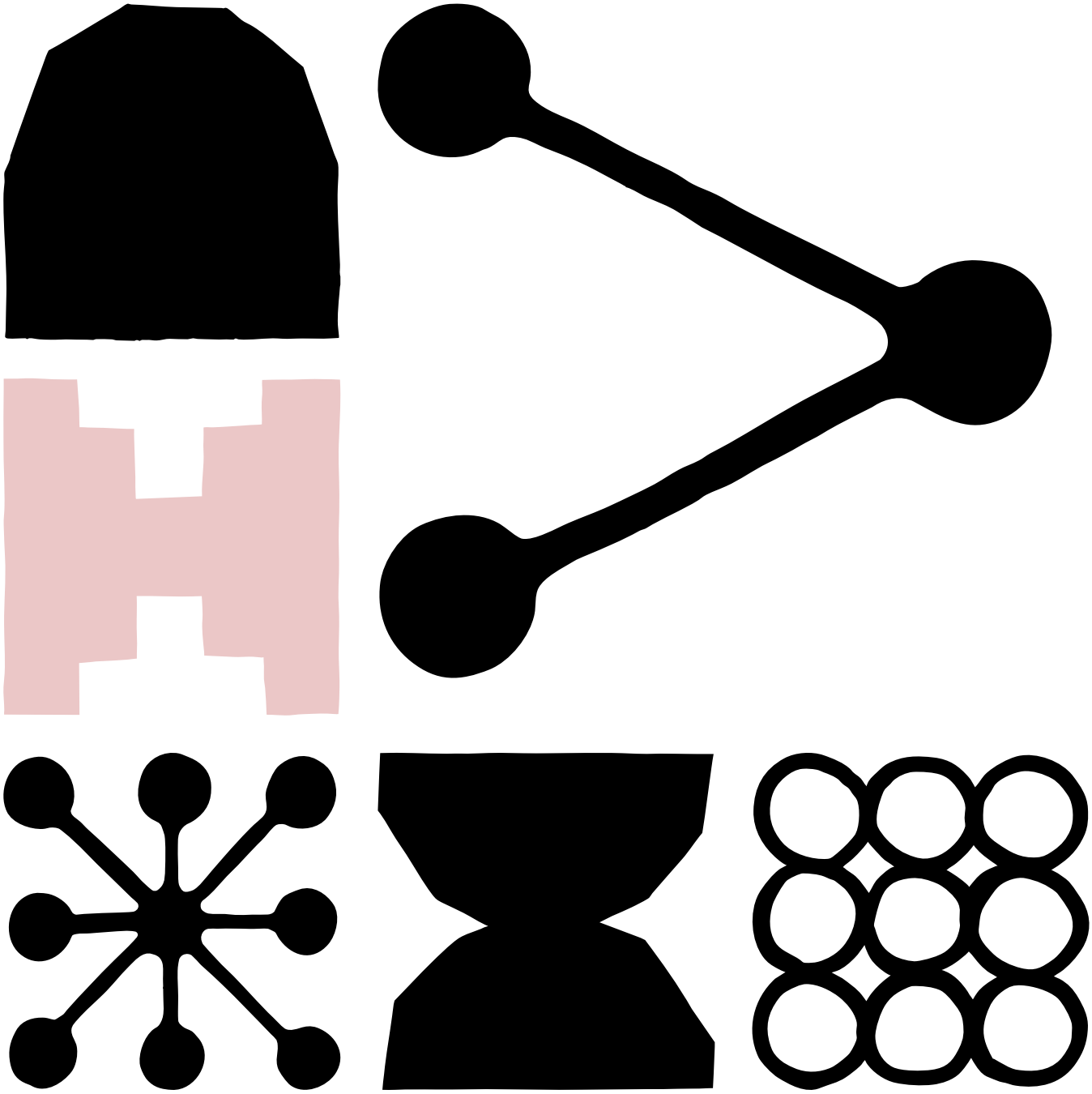


에이전트를 위한 효과적인 도구 작성 — 에이전트와 함께

게시됨2025년 9월 11일

에이전트는 우리가 제공하는 도구만큼만 효과적입니다. 고품질 도구와 평가를 작성하는 방법, 그리고 Claude가 자신을 위한 도구를 최적화하도록 하여 성능을 향상시키는 방법을 공유합니다.

번역할 텍스트가 "The"만 제공되었습니다. 완전한 문장이나 더 많은 맥락을 제공해 주시면 정확한 번역을 도와드리겠습니다. **모델 컨텍스트 프로토콜 (MCP)**LLM 에이전트가 실제 작업을 해결하기 위해 잠재적으로 수백 개의 도구를 활용할 수 있도록 지원할 수 있습니다. 하지만 이러한 도구들을 어떻게 최대한 효과적으로 만들 수 있을까요?

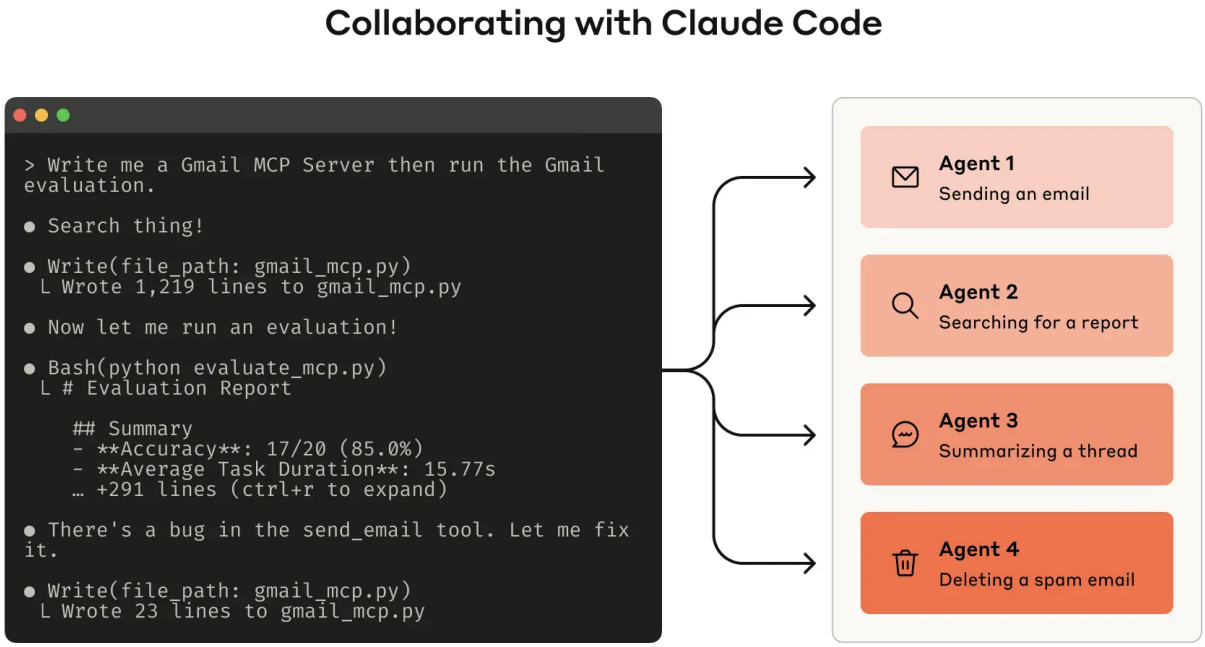
이 게시물에서는 다양한 에이전틱 AI 시스템에서 성능을 향상시키기 위한 가장 효과적인 기법들을 설명합니다1.

먼저 다음과 같은 방법을 다룹니다:

- 도구의 프로토타입을 구축하고 테스트하기
- 에이전트와 함께 도구의 포괄적인 평가를 생성하고 실행하세요
- Claude Code와 같은 에이전트와 협력하여 도구의 성능을 자동으로 향상시키세요

우리는 그 과정에서 확인한 고품질 도구 작성을 위한 핵심 원칙들로 마무리합니다:

- 구현할 도구(그리고 구현하지 않을 도구)를 올바르게 선택하기
- 기능의 명확한 경계를 정의하기 위한 도구 네임스페이싱
- 도구에서 에이전트로 의미 있는 컨텍스트 반환하기
- 토큰 효율성을 위한 도구 응답 최적화
- 도구 설명 및 사양에 대한 프롬프트 엔지니어링



평가를 구축하면 도구의 성능을 체계적으로 측정할 수 있습니다. Claude Code를 사용하여 이 평가에 대해 도구를 자동으로 최적화할 수 있습니다.

도구란 무엇인가요?

컴퓨팅에서 결정론적 시스템은 동일한 입력이 주어질 때마다 항상 같은 출력을 생성하는 반면, *비결정론적* 시스템인 에이전트와 같은 시스템은 동일한 시작 조건에서도 다양한 응답을 생성할 수 있습니다.

전통적으로 소프트웨어를 작성할 때, 우리는 결정론적 시스템들 간의 계약을 설정합니다. 예를 들어, `getWeather("NYC")` 와 같은 함수 호출은 호출될 때마다 항상 정확히 같은 방식으로 뉴욕시의 날씨를 가져올 것입니다.

도구는 결정론적 시스템과 비결정론적 에이전트 간의 계약을 반영하는 새로운 종류의 소프트웨어입니다. 사용자가 "오늘 우산을 가져가야 할까요?"라고 물으면, 에이전트는 날씨 도구를 호출하거나, 일반적인 지식으로 답변하거나, 심지어 위치에 대한 명확한 질문을 먼저 할 수도 있습니다. 때때로 에이전트는 환각을 일으키거나 도구 사용법을 파악하지 못할 수도 있습니다.

이는 에이전트를 위한 소프트웨어를 작성할 때 우리의 접근 방식을 근본적으로 재고해야 함을 의미합니다: 다른 개발자나 시스템을 위해 함수와 API를 작성하는 방식으로 도구와MCP 서버를 작성하는 대신, 에이전트를 위해 설계해야 합니다.

우리의 목표는 에이전트가 다양한 성공 전략을 추구하기 위해 도구를 사용함으로써 광범위한 작업을 해결하는 데 효과적일 수 있는 표면적을 늘리는 것입니다. 다행히도 우리의 경험상, 에이전트에게 가장 "인체공학적인" 도구들이 인간이 이해하기에도 놀랍도록 직관적인 것으로 나타났습니다.

도구 작성 방법

이 섹션에서는 에이전트에게 제공하는 도구를 작성하고 개선하는 데 있어 에이전트와 협력하는 방법을 설명합니다. 먼저 도구의 빠른 프로토타입을 구축하고 로컬에서 테스트하는 것부터 시작하세요. 다음으로, 후속 변경 사항을 측정하기 위해 포괄적인 평가를 실행하세요. 에이전트와 함께 작업하면서, 에이전트가 실제 작업에서 강력한 성능을 달성할 때까지 도구를 평가하고 개선하는 과정을 반복할 수 있습니다.

프로토타입 구축

에이전트가 어떤 도구를 사용하기 편하게 느끼고 어떤 도구를 그렇지 않게 느낄지는 직접 경험해보지 않고는 예측하기 어려울 수 있습니다. 도구의 빠른 프로토타입을 구축하는 것부터 시작하세요. Claude Code를 사용하여 도구를 작성하는 경우(잠재적으로 원샷으로), Claude에게 소프트웨어 라이브러리, API, 또는 SDK(MCP SDK) 당신의 도구들이 의존하게 될 것입니다. LLM 친화적인 문서는 일반적으로 공식 문서 사이트의 플랫 `llms.txt` 파일에서 찾을 수 있습니다 (여기 우리의 API의).

도구를 다음으로 감싸기로컬 MCP 서버 또는 데스크톱 확장 프로그램(DXT)를 사용하면 Claude Code나 Claude Desktop 앱에서 도구를 연결하고 테스트할 수 있습니다.

로컬 MCP 서버를 Claude Code에 연결하려면 다음을 실행하세요 `claude mcp add [args...]`.

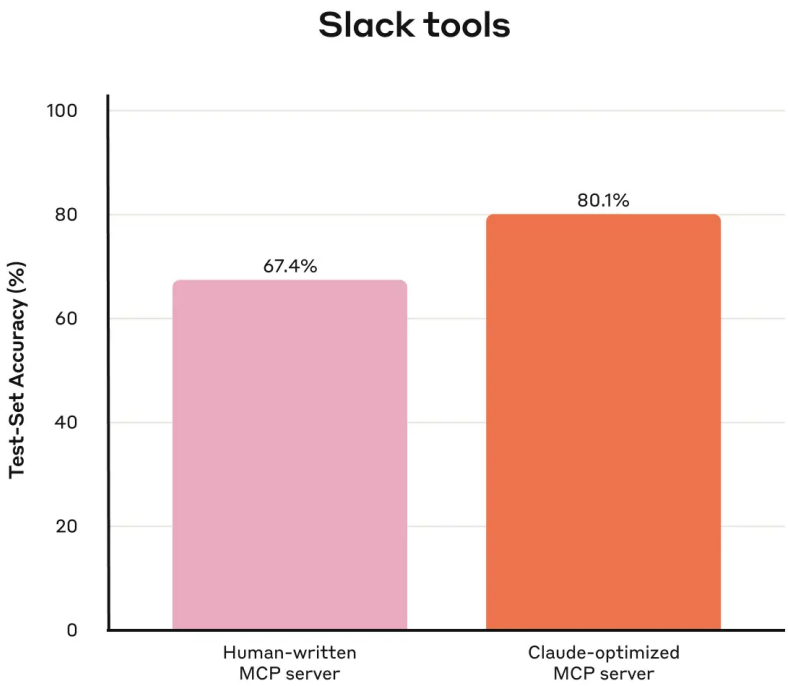
로컬 MCP 서버나 DXT를 Claude Desktop 앱에 연결하려면 각각 `설정 > 개발자` 또는 `설정 > 확장 프로그램` 로 이동하세요.

도구들은 프로그래밍 방식의 테스트를 위해 직접Anthropic API호출에 전달될 수도 있습니다.

도구들을 직접 테스트하여 문제점을 파악하세요. 사용자들로부터 피드백을 수집하여 도구들이 지원할 것으로 예상되는 사용 사례와 프롬프트에 대한 직관을 구축하세요.

평가 실행

다음으로, 평가를 실행하여 Claude가 당신의 도구를 얼마나 잘 사용하는지 측정해야 합니다. 실제 사용 사례에 기반한 많은 평가 작업을 생성하는 것부터 시작하세요. 결과를 분석하고 도구를 개선하는 방법을 결정하는 데 도움이 되도록 에이전트와 협력하는 것을 권장합니다. 이 전체 과정을 다음에서 확인하세요**도구 평가 쿡북**.



내부 Slack 도구의 홀드아웃 테스트 세트 성능

평가 작업 생성

초기 프로토타입을 통해 Claude Code는 도구를 빠르게 탐색하고 수십 개의 프롬프트와 응답 쌍을 생성할 수 있습니다. 프롬프트는 실제 사용 사례에서 영감을 받아야 하며, 현실적인 데이터 소스와 서비스(예: 내부 지식 베이스 및 마이크로서비스)를 기반으로 해야 합니다. 충분한 복잡성으로 도구를 스트레스 테스트하지 않는 지나치게 단순하거나 피상적인 "샌드박스" 환경은 피하는 것이 좋습니다. 강력한 평가 작업에는 여러 번의 도구 호출이 필요할 수 있으며, 잠재적으로 수십 번의 호출이 필요할 수도 있습니다.

다음은 강력한 작업의 몇 가지 예시입니다:

- 다음 주에 Jane과 최신 Acme Corp 프로젝트에 대해 논의하기 위한 회의를 예약하세요. 지난 프로젝트 계획 회의 노트를 첨부하고 회의실을 예약하세요.
- 고객 ID 9182가 한 번의 구매 시도에 대해 세 번 청구되었다고 신고했습니다. 모든 관련 로그 항목을 찾고 동일한 문제로 영향을 받은 다른 고객이 있는지 확인하세요.
- 고객 Sarah Chen이 방금 취소 요청을 제출했습니다. 고객 유지 제안을 준비하세요. 다음을 결정하세요: (1) 그들이 떠나는 이유, (2) 가장 매력적인 고객 유지 제안은 무엇인지, (3) 제안을 하기 전에 알아야 할 위험 요소가 있는지.

그리고 다음은 몇 가지 약한 작업들입니다:

- 다음 주에 jane@acme.corp와 회의를 예약하세요.

- 결제 로그에서 검색하세요 `purchase_complete` 그리고 `customer_id=9182`.
- 고객 ID 45892로 취소 요청을 찾으세요.

각 평가 프롬프트는 검증 가능한 응답이나 결과와 쌍을 이루어야 합니다. 검증자는 정답과 샘플 응답 간의 정확한 문자열 비교처럼 간단할 수도 있고, Claude를 활용해 응답을 판단하는 것처럼 고급스러울 수도 있습니다. 형식, 구두점, 또는 유효한 대안적 표현과 같은 허위 차이로 인해 올바른 응답을 거부하는 지나치게 엄격한 검증자는 피하세요.

각 프롬프트-응답 쌍에 대해, 평가 중에 에이전트가 각 도구의 목적을 성공적으로 파악하는지 측정하기 위해 작업을 해결할 때 에이전트가 호출할 것으로 예상되는 도구를 선택적으로 지정할 수도 있습니다. 하지만 작업을 올바르게 해결하는 데 여러 가지 유효한 경로가 있을 수 있으므로, 전략을 과도하게 구체화하거나 과적합하지 않도록 주의하세요.

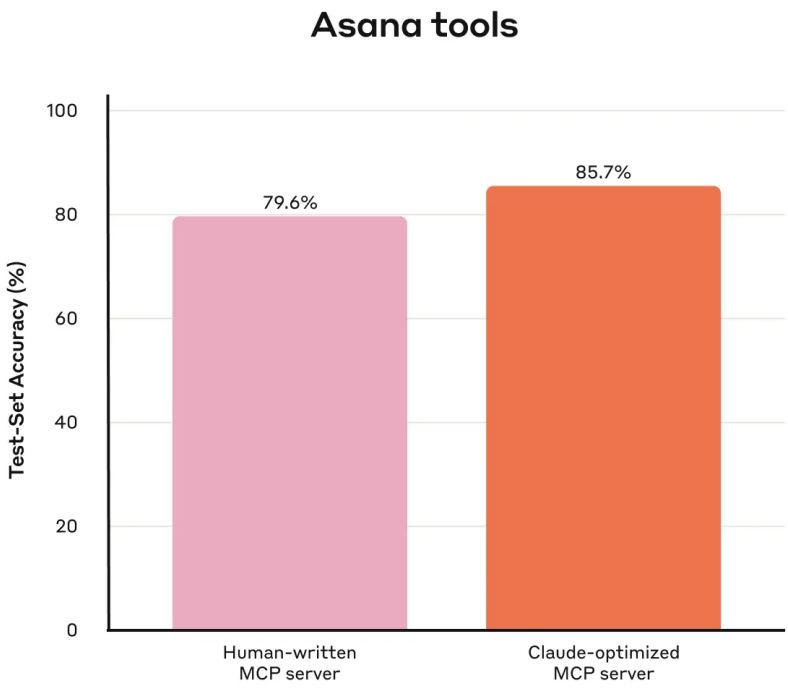
평가 실행

직접적인 LLM API 호출을 통해 프로그래밍 방식으로 평가를 실행하는 것을 권장합니다. 간단한 에이전트 루프를 사용하세요 (`동안` -루프는 교대로 LLM API와 도구 호출을 래핑합니다): 각 평가 작업마다 하나의 루프. 각 평가 에이전트는 단일 작업 프롬프트와 당신의 도구들을 제공받아야 합니다.

평가 에이전트의 시스템 프롬프트에서, 에이전트가 구조화된 응답 블록(검증용)뿐만 아니라 추론 및 피드백 블록도 출력하도록 지시할 것을 권장합니다. 에이전트가 이러한 것들을 출력하도록 지시하는 것은 *전에* 도구 호출 및 응답 블록은 연쇄 사고 (CoT) 행동을 유발함으로써 LLM의 효과적인 지능을 향상시킬 수 있습니다.

Claude로 평가를 실행하는 경우, interleaved thinking을 활성화하여 유사한 기능을 "즉시 사용"할 수 있습니다. 이는 에이전트가 특정 도구를 호출하거나 호출하지 않는 이유를 탐구하고 도구 설명 및 사양에서 개선이 필요한 특정 영역을 강조하는 데 도움이 됩니다.

최상위 정확도뿐만 아니라, 개별 도구 호출 및 작업의 총 실행 시간, 총 도구 호출 수, 총 토큰 소비량, 도구 오류와 같은 다른 지표들도 수집할 것을 권장합니다. 도구 호출을 추적하면 에이전트가 추구하는 일반적인 워크플로우를 파악하고 도구 통합을 위한 기회를 제공하는 데 도움이 될 수 있습니다.



내부 Asana 도구의 홀드아웃 테스트 세트 성능

결과 분석

에이전트는 모순된 도구 설명부터 비효율적인 도구 구현, 혼란스러운 도구 스키마에 이르기까지 모든 것에 대해 문제를 발견하고 피드백을 제공하는 유용한 파트너입니다. 하지만 에이전트가 피드백과 응답에서 생략하는 내용이 포함하는 내용보다 더 중요할 수 있다는 점을 명심하세요. LLM은 항상**자신이 의미하는 바를 말하지는 않습니다**.

에이전트가 막히거나 혼란스러워하는 지점을 관찰하세요. 평가 에이전트의 추론과 피드백(또는 CoT)을 읽어서 문제점을 파악하세요. 원시 대화 기록(도구 호출과 도구 응답 포함)을 검토하여 에이전트의 CoT에 명시적으로 설명되지 않은 행동을 포착하세요. 행간을 읽으세요. 평가 에이전트가 반드시 정확한 답변과 전략을 알고 있는 것은 아니라는 점을 기억하세요.

도구 호출 메트릭을 분석하세요. 많은 중복 도구 호출은 페이지네이션이나 토큰 제한 매개변수의 적절한 조정이 필요함을 시사할 수 있으며, 잘못된 매개변수로 인한 많은 도구 오류는 도구가 더 명확한 설명이나 더 나은 예시를 필요로 할 수 있음을 시사할 수 있습니다. Claude의**웹 검색 도구**를 출시했을 때, Claude가 도구의 `2025` 에 불필요하게 추가하고 있다는 것을 발견했습니다 `쿼리` 매개변수로 인해 검색 결과가 편향되고 성능이 저하됩니다 (도구 설명을 개선하여 Claude를 올바른 방향으로 유도했습니다).

에이전트와의 협업

에이전트가 결과를 분석하고 도구를 개선하도록 할 수도 있습니다. 평가 에이전트의 대화 기록을 연결하여 Claude Code에 붙여넣기만 하면 됩니다. Claude는 대화 기록을 분석하고 여러 도구를 한 번에 리팩토링하는 전문가입니다. 예를 들어, 새로운 변경사항이 적용될 때 도구 구현과 설명이 일관성을 유지하도록 보장할 수 있습니다.

실제로 이 게시물의 대부분의 조언은 Claude Code로 내부 도구 구현을 반복적으로 최적화하면서 얻어진 것입니다. 우리의 평가는 실제 프로젝트, 문서, 메시지를 포함한 내부 워크플로우의 복잡성을 반영하여 내부 작업 공간을 기반으로 만들어졌습니다.

우리는 "훈련" 평가에 과적합되지 않도록 보류된 테스트 세트에 의존했습니다. 이러한 테스트 세트를 통해 "전문가" 도구 구현으로 달성한 것 이상의 추가적인 성능 향상을 얻을 수 있음을 확인했습니다. 이는 해당 도구들이 연구자들이 수동으로 작성한 것이든 Claude 자체가 생성한 것이든 상관없이 마찬가지였습니다.

다음 섹션에서는 이 과정에서 배운 내용 중 일부를 공유하겠습니다.

효과적인 도구 작성을 위한 원칙

이 섹션에서는 효과적인 도구 작성을 위한 몇 가지 지침 원칙으로 우리의 학습 내용을 정리합니다.

에이전트를 위한 올바른 도구 선택

더 많은 도구가 항상 더 나은 결과로 이어지는 것은 아닙니다. 우리가 관찰한 일반적인 오류는 기존 소프트웨어 기능이나 API 엔드포인트를 단순히 래핑한 도구들입니다—그 도구들이 에이전트에게 적합한지 여부와 관계없이 말입니다. 이는 에이전트가 기존 소프트웨어와는 다른 "어포던스"를 가지고 있기 때문입니다—즉, 에이전트는 그러한 도구들로 취할 수 있는 잠재적 행동을 인식하는 방식이 다릅니다.

LLM 에이전트는 제한된 "컨텍스트"를 가지고 있는 반면(즉, 한 번에 처리할 수 있는 정보의 양에 제한이 있음), 컴퓨터 메모리는 저렴하고 풍부합니다. 주소록에서 연락처를 검색하는 작업을 생각해보세요. 기존 소프트웨어 프로그램은 연락처 목록을 한 번에 하나씩 효율적으로 저장하고 처리할 수 있으며, 다음으로 넘어가기 전에 각각을 확인합니다.

그러나 LLM 에이전트가 모든 연락처를 반환하는 도구를 사용한 다음 각각을 토큰별로 읽어야 한다면, 관련 없는 정보로 인해 제한된 컨텍스트 공간을 낭비하게 됩니다(주소록에서 연락처를 찾기 위해 각 페이지를 위에서 아래로 읽는 것, 즉 무차별 대입 검색을 통해 찾는 것을 상상해보세요). 더 나은 그리고 더 자연스러운 접근법은(에이전트와 인간 모두에게) 먼저 관련 페이지로 건너뛰는 것입니다(아마도 알파벳순으로 찾는 것).

특정 고영향 워크플로우를 대상으로 하는 몇 가지 신중한 도구를 구축하여 평가 작업과 일치시키고 거기서부터 확장해 나가는 것을 권장합니다. 주소록의 경우, `search_contacts` 또는 `메시지_연락처` 도구 대신 `연락처_목록` 도구를 구현하는 것을 선택할 수 있습니다.

도구는 기능을 통합하여 내부적으로 잠재적으로 *여러 개의* 개별 작업(또는 API 호출)을 처리할 수 있습니다. 예를 들어, 도구는 관련 메타데이터로 도구 응답을 풍부하게 하거나 자주 연결되는 다단계 작업을 단일 도구 호출로 처리할 수 있습니다.

다음은 몇 가지 예시입니다:

- 대신 `list_users`, `list_events`, 그리고 `create_event` 도구들을 고려할 때, 가용성을 찾고 이벤트를 예약하는 `schedule_event` 도구를 구현하는 것을 고려해보세요.
- 대신 `read_logs` 도구를 구현하는 것보다는, 관련 로그 라인과 주변 컨텍스트만 반환하는 `검색 로그` 도구를 구현하는 것을 고려해보세요.

- 구현하는 대신 `get_customer_by_id`, `거래 목록 조회`, 그리고 `메모 목록 조회` 도구들을 구현하는 대신, 고객의 최근 및 관련 정보를 모두 한 번에 컴파일하는 `get_customer_context` 도구를 구현하세요.

구축하는 각 도구가 명확하고 구별되는 목적을 가지도록 하세요. 도구들은 에이전트가 동일한 기본 리소스에 접근할 수 있는 인간과 같은 방식으로 작업을 세분화하고 해결할 수 있도록 하는 동시에, 중간 출력으로 인해 소모되었을 컨텍스트를 줄여야 합니다.

너무 많은 도구나 중복되는 도구들은 에이전트가 효율적인 전략을 추구하는 데 방해가 될 수 있습니다. 구축할 도구(또는 구축하지 않을 도구)에 대한 신중하고 선택적인 계획은 정말로 큰 효과를 가져올 수 있습니다.

도구 네임스페이싱

AI 에이전트는 잠재적으로 수십 개의 MCP 서버와 수백 개의 다양한 도구에 접근할 수 있으며, 여기에는 다른 개발자들이 만든 도구들도 포함됩니다. 도구들이 기능적으로 중복되거나 목적이 모호할 때, 에이전트는 어떤 도구를 사용해야 할지 혼란스러워 할 수 있습니다.

네임스페이싱(관련 도구들을 공통 접두사로 그룹화)은 많은 도구들 사이의 경계를 명확히 구분하는 데 도움이 될 수 있습니다. MCP 클라이언트는 때때로 기본적으로 이를 수행합니다. 예를 들어, 서비스별로 도구를 네임스페이싱하거나(예: `asana_search`, `jira_search`) 리소스별로 네임스페이싱하는 것(예: `asana_projects_search`, `asana_users_search`)은 에이전트가 적절한 시점에 올바른 도구를 선택하는 데 도움이 될 수 있습니다.

접두사 기반 네임스페이스와 접미사 기반 네임스페이스 중 선택하는 것이 도구 사용 평가에 중요한 영향을 미친다는 것을 발견했습니다. 효과는 LLM에 따라 다르므로 자체 평가에 따라 명명 체계를 선택하시기 바랍니다.

에이전트는 잘못된 도구를 호출하거나, 올바른 도구를 잘못된 매개변수로 호출하거나, 너무 적은 도구를 호출하거나, 도구 응답을 잘못 처리할 수 있습니다. 작업의 자연스러운 세분화를 반영하는 이름을 가진 도구들을 선택적으로 구현함으로써, 에이전트의 컨텍스트에 로드되는 도구와 도구 설명의 수를 동시에 줄이고 에이전트적 계산을 에이전트의 컨텍스트에서 도구 호출 자체로 다시 이전할 수 있습니다. 이는 에이전트가 실수를 범할 전반적인 위험을 줄입니다.

도구에서 의미 있는 컨텍스트 반환하기

같은 맥락에서, 도구 구현은 에이전트에게 높은 신호 정보만을 반환하도록 주의해야 합니다. 유연성보다는 맥락적 관련성을 우선시해야 하며, 저수준 기술적 식별자는 피해야 합니다(예: `uuid`, `256px_image_url`, `mime_type`). `name`, `image_url`, 그리고 `파일_타입`와 같은 필드들이 에이전트의 후속 작업과 응답에 직접적으로 정보를 제공할 가능성이 훨씬 높습니다.

에이전트들은 또한 암호화된 식별자보다 자연어 이름, 용어 또는 식별자를 훨씬 더 성공적으로 처리하는 경향이 있습니다. 임의의 영숫자 UUID를 더 의미론적으로 의미 있고 해석 가능한 언어(또는 심지어 0부터 시작하는 ID 체계)로 해결하는 것만으로도 환각을 줄여 검색 작업에서 Claude의 정확도가 크게 향상된다는 것을 발견했습니다.

일부 경우에는 에이전트가 자연어와 기술적 식별자 출력 모두와 상호작용할 수 있는 유연성이 필요할 수 있습니다. 이는 단지 다운스트림 도구 호출을 트리거하기 위해서라도 필요합니다(예를 들어, `search_user(name='jane')` → `send_message(id=12345)`). 도구에서 간단한 `response_format` 열거형 매개변수를 노출하여 두 가지 모두를 활성화할 수 있으며, 이를 통해 에이전트가 도구가 `"간결한"` 또는 `"상세한"` 응답을 반환할지 여부를 제어할 수 있습니다(아

래 이미지 참조).

더 큰 유연성을 위해 더 많은 형식을 추가할 수 있습니다. 이는 GraphQL과 유사하게 받고 싶은 정보를 정확히 선택할 수 있는 방식입니다. 다음은 도구 응답의 상세도를 제어하는 ResponseFormat 열거형의 예시입니다:

```
enum ResponseFormat {
   DETAILED = "detailed",
   CONCISE = "concise"
}
```

복사

다음은 상세한 도구 응답의 예시입니다 (206토큰):

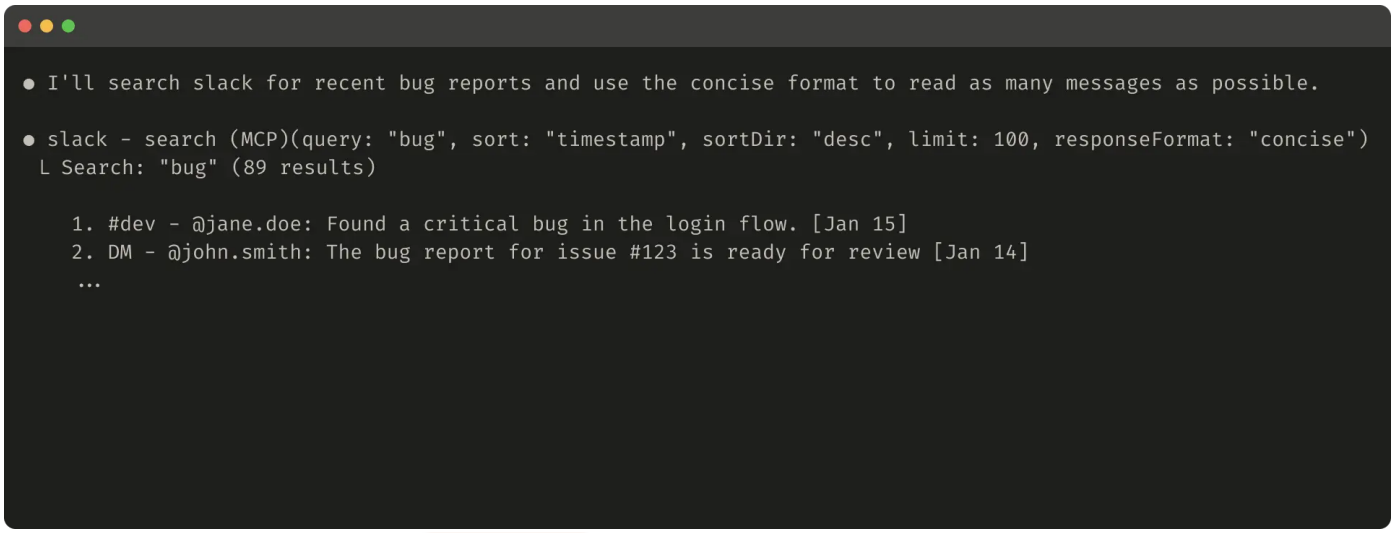
```
● I'll search slack for recent bug reports and use the detailed format to see which channel IDs and threads to
investigate further.

● slack - search (MCP)(query: "bug", sort: "timestamp", sortDir: "desc", limit: 100, responseFormat:
"detailed")
 L Search results for: "bug"

   ═══ Result 1 of 89 ═══
   Channel: #dev (C1234567890)
   From: @jane.doe (U123456789)
   Time: 2024-01-15 10:30:45 UTC
   TS: 1705316445.123456
   Text: Found a critical bug in the login flow.

   ═══ Result 2 of 89 ═══
   Channel: DM with @john.smith
   From: @john.smith (U987654321)
   Time: 2024-01-14 15:22:18 UTC
   TS: 1705247738.234567
   Text: The bug report for issue #123 is ready for review
   Files: bug-report-123.pdf
   …
```

다음은 간결한 도구 응답의 예시입니다 (72토큰):



Slack 스레드와 스레드 답글은 고유한 `thread_ts` 로 식별되며, 이는 스레드 답글을 가져오는 데 필요합니다. `thread_ts` 및 기타 ID들(`channel_id` , `user_id`)은 다음에서 검색할 수 있습니다 `"detailed"` 이러한 도구들이 필요한 추가 도구 호출을 가능하게 하는 도구 응답입니다. `"concise"` 도구 응답은 스레드 콘텐츠만 반환하고 ID는 제외합니다. 이 예시에서는 토큰의 약 ⅓을 사용합니다. `"concise"` 도구 응답입니다.

도구 응답 구조(예: XML, JSON 또는 Markdown)조차도 평가 성능에 영향을 미칠 수 있으며, 모든 상황에 적용되는 만능 해결책은 없습니다. 이는 LLM이 다음 토큰 예측으로 훈련되어 훈련 데이터와 일치하는 형식에서 더 나은 성능을 보이는 경향이 있기 때문입니다. 최적의 응답 구조는 작업과 에이전트에 따라 크게 달라집니다. 자체 평가를 바탕으로 최적의 응답 구조를 선택하시기 바랍니다.

토큰 효율성을 위한 도구 응답 최적화

컨텍스트의 품질을 최적화하는 것이 중요합니다. 하지만 도구 응답에서 에이전트에게 반환되는 컨텍스트의 *양* 을 최적화하는 것도 마찬가지로 중요합니다.

많은 컨텍스트를 사용할 수 있는 모든 도구 응답에 대해 합리적인 기본 매개변수 값과 함께 페이지네이션, 범위 선택, 필터링 및/또는 잘라내기의 조합을 구현할 것을 제안합니다. Claude Code의 경우, 기본적으로 도구 응답을 25,000개 토큰으로 제한합니다. 에이전트의 효과적인 컨텍스트 길이는 시간이 지남에 따라 증가할 것으로 예상하지만, 컨텍스트 효율적인 도구의 필요성은 여전히 남아있을 것입니다.

응답을 잘라내기로 선택한다면, 도움이 되는 지침으로 에이전트를 안내해야 합니다. 지식 검색 작업에서 하나의 광범위한 검색 대신 여러 개의 작고 목표가 명확한 검색을 수행하는 것과 같이, 더 토큰 효율적인 전략을 추구하도록 에이전트를 직접 권장할 수 있습니다. 마찬가지로, 도구 호출에서 오류가 발생하는 경우(예: 입력 검증 중), 불투명한 오류 코드나 트레이스백보다는 구체적이고 실행 가능한 개선사항을 명확하게 전달하도록 오류 응답을 프롬프트 엔지니어링할 수 있습니다.

다음은 잘린 도구 응답의 예시입니다:

```
● I'll find all of your transactions on Stripe and provide a summary for you.

● stripe - transactions_search (MCP)(limit: 5000, responseFormat: "concise")
 L ## Transaction Search Results

    Found **2,847 transactions** matching your query.

    The results are truncated. Showing first 3 results:

    | Date | Description | Amount | Category |
    ├──────┼─────────────┼────────┼──────────┤
    | 2024-01-15 | Payment from Acme Corp | +$5,200.00 | Revenue |
    | 2024-01-14 | Payment from TechStart | +$3,100.00 | Revenue |
    | 2024-01-13 | Payment from Cloud Co. | +$3,100.00 | Revenue |

    **Summary of all 2,847 results:**
    - Total Revenue: $458,291.00
    - Date Range: Jan 1 - Jan 15, 2024

    ## To refine these results, you can:

    - **Search for specific vendors**: Use `transactions_search(payee: "Acme Corp")` to see only Acme Corp
charges
    - **Filter by amount range**: Use `transactions_search(minAmount: 1000, maxAmount: 5000)`
    - **Get next page**: Use `transactions_search(query: <query>, page: 2)`
```

다음은 도움이 되지 않는 오류 응답의 예시입니다:

```
● Sure. I'll fetch John's contact information from his profile.

● asana - user_info (MCP)(userId: "john.doe@acme.corp")
 L {
     "error": {
       "code": "RESOURCE_NOT_FOUND",
       "status": 422,
       "message": "Invalid value",
       "details": {
         "field": "userId",
         "value": "john.doe@acme.corp",
         "type": "invalid_value"
       }
     }
   }
```

다음은 도움이 되는 오류 응답의 예시입니다:

```
● Sure. I'll fetch John's contact information from his profile.

● asana - user_info (MCP)(userId: "john.doe@acme.corp")
  L Tool Response:

    # Resource Not Found: Invalid `userId`

    ## Error Summary
    Your request to `/api/user/info` failed because the `userId` `john.doe@acme.corp` does not exist or is in
the wrong format.

    ## Valid User IDs
    Examples:
    - `1928298149291729`
    - `9381719375914731`

    ## Resolving a User ID
    - Call user_search()
```

도구 절단 및 오류 응답은 에이전트가 더 토큰 효율적인 도구 사용 행동(필터나 페이지네이션 사용)을 하도록 유도하거나 올바르게 형식화된 도구 입력의 예시를 제공할 수 있습니다.

도구 설명에 대한 프롬프트 엔지니어링

이제 도구를 개선하는 가장 효과적인 방법 중 하나인 도구 설명과 사양에 대한 프롬프트 엔지니어링에 대해 알아보겠습니다. 이러한 내용들이 에이전트의 컨텍스트에 로드되기 때문에, 이들은 집합적으로 에이전트가 효과적인 도구 호출 행동을 하도록 유도할 수 있습니다.

도구 설명과 사양을 작성할 때는 팀의 신입 직원에게 도구를 어떻게 설명할지 생각해보세요. 암묵적으로 가져올 수 있는 맥락—특수한 쿼리 형식, 전문 용어의 정의, 기본 리소스 간의 관계—을 명시적으로 만드는 것을 고려하세요. 예상되는 입력과 출력을 명확하게 설명하고(엄격한 데이터 모델로 강제하여) 모호함을 피하세요. 특히 입력 매개변수는 명확하게 명명되어야 합니다: `사용자`, 다음과 같은 매개변수를 시도해보세요 `사용자_아이디`.

평가를 통해 프롬프트 엔지니어링의 영향을 더욱 확신을 가지고 측정할 수 있습니다. 도구 설명에 대한 작은 개선만으로도 극적인 향상을 얻을 수 있습니다. Claude Sonnet 3.5는 도구 설명을 정밀하게 개선한 후 SWE-bench Verified 평가에서 최첨단 성능을 달성했으며, 오류율을 극적으로 줄이고 작업 완료율을 향상시켰습니다.

도구 정의에 대한 다른 모범 사례들은 저희 개발자 가이드에서 찾을 수 있습니다. Claude용 도구를 구축하고 있다면, 도구들이 Claude의 시스템 프롬프트에 동적으로 로드되는 방식에 대해서도 읽어보시기를 권합니다. 마지막으로, MCP 서버용 도구를 작성하고 있다면,도구 주석어떤 도구가 개방형 세계 접근을 필요로 하거나 파괴적인 변경을 수행하는지 공개하는 데 도움이 됩니다.

앞을 내다보며

에이전트를 위한 효과적인 도구를 구축하려면, 예측 가능하고 결정론적인 패턴에서 비결정론적인 패턴으로 소프트웨어 개발 관행을 재정향해야 합니다.

이 글에서 설명한 반복적이고 평가 중심적인 과정을 통해, 우리는 도구를 성공적으로 만드는 일관된 패턴들을 확인했습니다: 효과적인 도구는 의도적이고 명확하게 정의되며, 에이전트 컨텍스트를 신중하게 사용하고, 다양한 워크플로우에서 함께 결합될 수 있으며, 에이전트가 실제 작업을 직관적으로 해결할 수 있도록 합니다.

미래에는 에이전트가 세계와 상호작용하는 구체적인 메커니즘이 진화할 것으로 예상됩니다. MCP 프로토콜의 업데이트부터 기반이 되는 LLM 자체의 업그레이드까지 말입니다. 에이전트를 위한 도구를 개선하는 체계적이고 평가 중심적인 접근 방식을 통해, 에이전트가 더욱 능력을 갖추게 됨에 따라 그들이 사용하는 도구도 함께 진화할 수 있도록 보장할 수 있습니다.

감사의 말

Ken Aizawa가 작성했으며, Research(Barry Zhang, Zachary Witten, Daniel Jiang, Sami Al-Sheikh, Matt Bell, Maggie Vo), MCP(Theodora Chu, John Welsh, David Soria Parra, Adam Jones), Product Engineering(Santiago Seira), Marketing(Molly Vorwerck), Design(Drew Roper), Applied AI(Christian Ryan, Alexander Bricken) 부서의 동료들이 소중한 기여를 했습니다.

Writing effective tools for agents — with agents

Published Sep 11, 2025

Agents are only as effective as the tools we give them. We share how to write high-quality tools and evaluations, and how you can boost performance by using Claude to optimize its tools for itself.

The Model Context Protocol (MCP) can empower LLM agents with potentially hundreds of tools to solve real-world tasks. But how do we make those tools maximally effective?

In this post, we describe our most effective techniques for improving performance in a variety of agentic AI systems1.

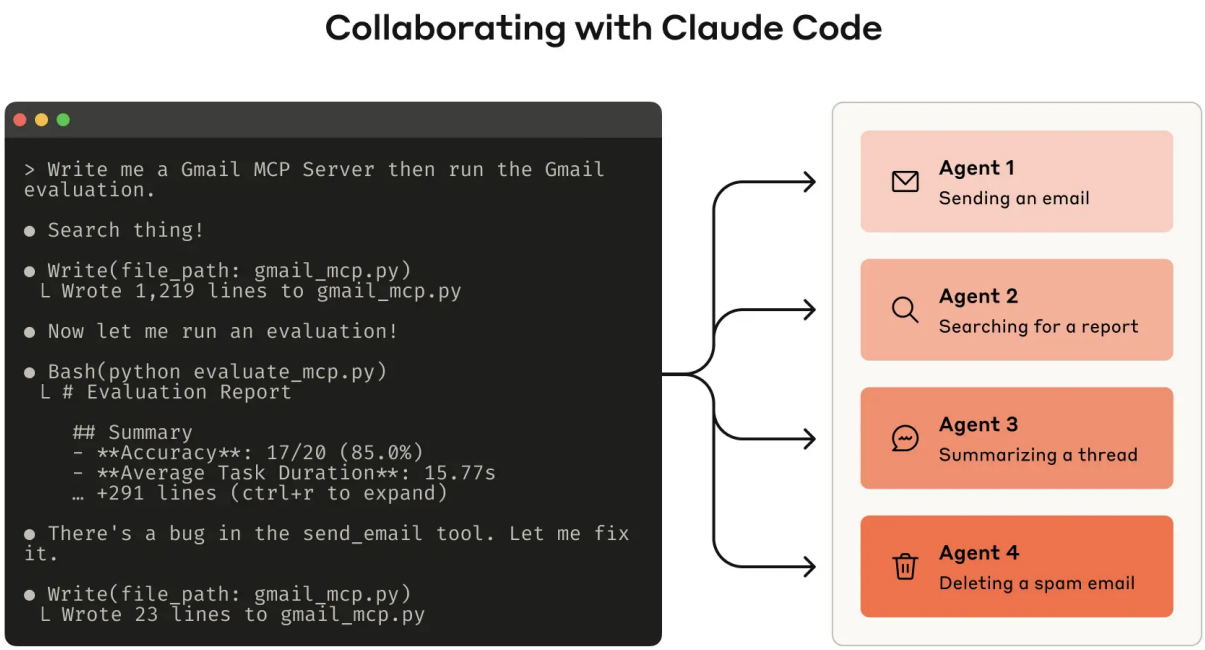
We begin by covering how you can:

- Build and test prototypes of your tools
- Create and run comprehensive evaluations of your tools with agents
- Collaborate with agents like Claude Code to automatically increase the performance of your tools

We conclude with key principles for writing high-quality tools we've identified along the way:

- Choosing the right tools to implement (and not to implement)
- Namespacing tools to define clear boundaries in functionality
- Returning meaningful context from tools back to agents

- Optimizing tool responses for token efficiency
- Prompt-engineering tool descriptions and specs



Building an evaluation allows you to systematically measure the performance of your tools. You can use Claude Code to automatically optimize your tools against this evaluation.

What is a tool?

In computing, deterministic systems produce the same output every time given identical inputs, while *non-deterministic* systems—like agents—can generate varied responses even with the same starting conditions.

When we traditionally write software, we're establishing a contract between deterministic systems. For instance, a function call like `getWeather("NYC")` will always fetch the weather in New York City in the exact same manner every time it is called.

Tools are a new kind of software which reflects a contract between deterministic systems and non-deterministic agents. When a user asks "Should I bring an umbrella today?," an agent might call the weather tool, answer from general knowledge, or even ask a clarifying question about location first. Occasionally, an agent might hallucinate or even fail to grasp how to use a tool.

This means fundamentally rethinking our approach when writing software for agents: instead of writing tools and MCP servers the way we'd write functions and APIs for other developers or systems, we need to design them for agents.

Our goal is to increase the surface area over which agents can be effective in solving a wide range of tasks by using tools to pursue a variety of successful strategies. Fortunately, in our experience, the tools that are most "ergonomic" for agents also end up being surprisingly intuitive to grasp as humans.

How to write tools

In this section, we describe how you can collaborate with agents both to write and to improve the tools you give them. Start by standing up a quick prototype of your tools and testing them locally. Next, run a comprehensive evaluation to measure subsequent changes. Working alongside agents, you can repeat the process of evaluating and improving your tools until your agents achieve strong performance on real-world tasks.

Building a prototype

It can be difficult to anticipate which tools agents will find ergonomic and which tools they won't without getting hands-on yourself. Start by standing up a quick prototype of your tools. If you're using Claude Code to write your tools (potentially in one-shot), it helps to give Claude documentation for any software libraries, APIs, or SDKs (including potentially the MCP SDK) your tools will rely on. LLM-friendly documentation can commonly be found in flat `llms.txt` files on official documentation sites (here's our API's).

Wrapping your tools in a local MCP server or Desktop extension (DXT) will allow you to connect and test your tools in Claude Code or the Claude Desktop app.

To connect your local MCP server to Claude Code, run `claude mcp add <name> <command> [args...]`.

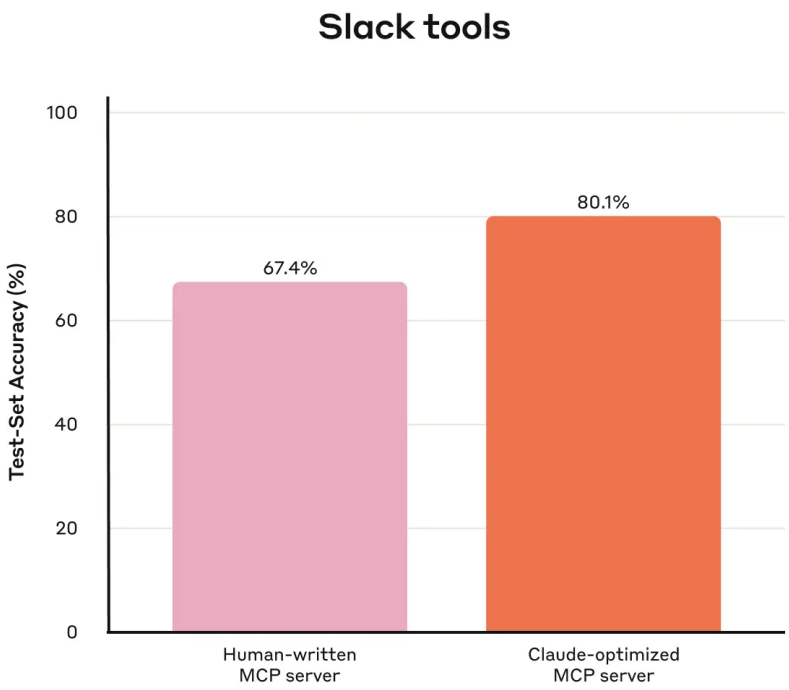
To connect your local MCP server or DXT to the Claude Desktop app, navigate to `Settings > Developer` or `Settings > Extensions`, respectively.

Tools can also be passed directly into Anthropic API calls for programmatic testing.

Test the tools yourself to identify any rough edges. Collect feedback from your users to build an intuition around the use-cases and prompts you expect your tools to enable.

Running an evaluation

Next, you need to measure how well Claude uses your tools by running an evaluation. Start by generating lots of evaluation tasks, grounded in real world uses. We recommend collaborating with an agent to help analyze your results and determine how to improve your tools. See this process end-to-end in our tool evaluation cookbook.



Held-out test set performance of our internal Slack tools

Generating evaluation tasks

With your early prototype, Claude Code can quickly explore your tools and create dozens of prompt and response pairs. Prompts should be inspired by real-world uses and be based on realistic data sources and services (for example, internal knowledge bases and microservices). We recommend you avoid overly simplistic or superficial "sandbox" environments that don't stress-test your tools with sufficient complexity. Strong evaluation tasks might require multiple tool calls—potentially dozens.

Here are some examples of strong tasks:

- Schedule a meeting with Jane next week to discuss our latest Acme Corp project. Attach the notes from our last project planning meeting and reserve a conference room.
- Customer ID 9182 reported that they were charged three times for a single purchase attempt. Find all relevant log entries and determine if any other customers were affected by the same issue.
- Customer Sarah Chen just submitted a cancellation request. Prepare a retention offer. Determine: (1) why they're leaving, (2) what retention offer would be most compelling, and (3) any risk factors we should be aware of before making an offer.

And here are some weaker tasks:

- Schedule a meeting with jane@acme.corp next week.
- Search the payment logs for `purchase_complete` and `customer_id=9182`.
- Find the cancellation request by Customer ID 45892.

Each evaluation prompt should be paired with a verifiable response or outcome. Your verifier can be as simple as an exact string comparison between ground truth and sampled responses, or as advanced as enlisting Claude to judge the response. Avoid overly strict verifiers that reject correct responses due to spurious differences like formatting, punctuation, or valid alternative phrasings.

For each prompt-response pair, you can optionally also specify the tools you expect an agent to call in solving the task, to measure whether or not agents are successful in grasping each tool's purpose during evaluation. However, because there might be multiple valid paths to solving tasks correctly, try to avoid overspecifying or overfitting to strategies.

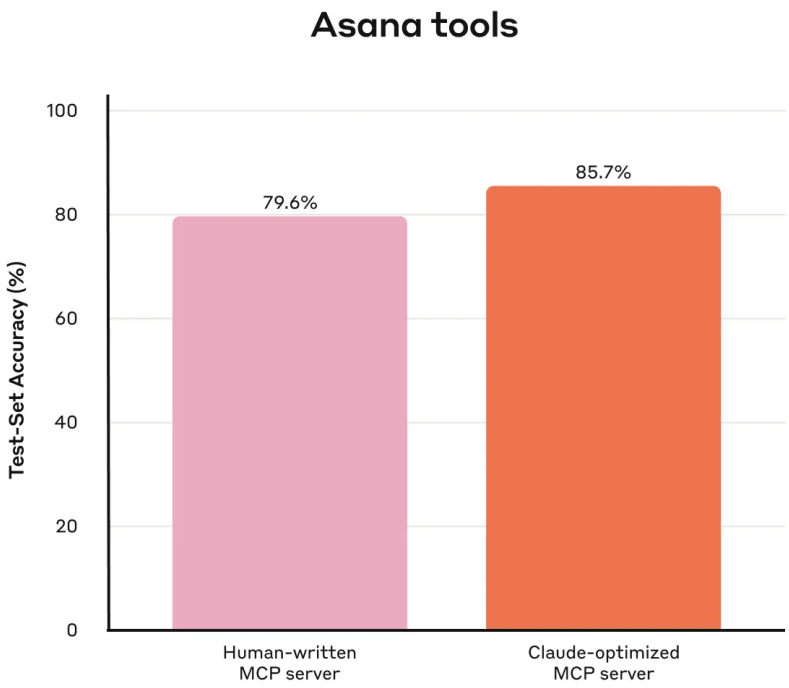
Running the evaluation

We recommend running your evaluation programmatically with direct LLM API calls. Use simple agentic loops (`while`-loops wrapping alternating LLM API and tool calls): one loop for each evaluation task. Each evaluation agent should be given a single task prompt and your tools.

In your evaluation agents' system prompts, we recommend instructing agents to output not just structured response blocks (for verification), but also reasoning and feedback blocks. Instructing agents to output these *before* tool call and response blocks may increase LLMs' effective intelligence by triggering chain-of-thought (CoT) behaviors.

If you're running your evaluation with Claude, you can turn on interleaved thinking for similar functionality "off-the-shelf". This will help you probe why agents do or don't call certain tools and highlight specific areas of improvement in tool descriptions and specs.

As well as top-level accuracy, we recommend collecting other metrics like the total runtime of individual tool calls and tasks, the total number of tool calls, the total token consumption, and tool errors. Tracking tool calls can help reveal common workflows that agents pursue and offer some opportunities for tools to consolidate.



Held-out test set performance of our internal Asana tools

Analyzing results

Agents are your helpful partners in spotting issues and providing feedback on everything from contradictory tool descriptions to inefficient tool implementations and confusing tool schemas. However, keep in mind that what agents omit in their feedback and responses can often be more important than what they include. LLMs don't always say what they mean.

Observe where your agents get stumped or confused. Read through your evaluation agents' reasoning and feedback (or CoT) to identify rough edges. Review the raw transcripts (including tool calls and tool responses) to catch any behavior not explicitly described in the agent's CoT. Read between the lines; remember that your evaluation agents don't necessarily know the correct answers and strategies.

Analyze your tool calling metrics. Lots of redundant tool calls might suggest some rightsizing of pagination or token limit parameters is warranted; lots of tool errors for invalid parameters might suggest tools could use clearer descriptions or better examples. When we launched Claude's web search tool, we identified that Claude was needlessly appending `2025` to the tool's `query` parameter, biasing search results and degrading performance (we steered Claude in the right direction by improving the tool description).

Collaborating with agents

You can even let agents analyze your results and improve your tools for you. Simply concatenate the transcripts from your evaluation agents and paste them into Claude Code. Claude is an expert at analyzing transcripts and refactoring lots of tools all at once—for example, to ensure tool implementations and descriptions remain self-consistent when new changes are made.

In fact, most of the advice in this post came from repeatedly optimizing our internal tool implementations with Claude Code. Our evaluations were created on top of our internal workspace, mirroring the complexity of our internal workflows, including real projects, documents, and messages.

We relied on held-out test sets to ensure we did not overfit to our “training” evaluations. These test sets revealed that we could extract additional performance improvements even beyond what we achieved with "expert" tool implementations—whether those tools were manually written by our researchers or generated by Claude itself.

In the next section, we’ll share some of what we learned from this process.

Principles for writing effective tools

In this section, we distill our learnings into a few guiding principles for writing effective tools.

Choosing the right tools for agents

More tools don’t always lead to better outcomes. A common error we’ve observed is tools that merely wrap existing software functionality or API endpoints—whether or not the tools are appropriate for agents. This is because agents have distinct “affordances” to traditional software—that is, they have different ways of perceiving the potential actions they can take with those tools

LLM agents have limited "context" (that is, there are limits to how much information they can process at once), whereas computer memory is cheap and abundant. Consider the task of searching for a contact in an address book. Traditional software programs can efficiently store and process a list of contacts one at a time, checking each one before moving on.

However, if an LLM agent uses a tool that returns ALL contacts and then has to read through each one token-by-token, it's wasting its limited context space on irrelevant information (imagine searching for a contact in your address book by reading each page from top-to-bottom—that is, via brute-force search). The better and more natural approach (for agents and humans alike) is to skip to the relevant page first (perhaps finding it alphabetically).

We recommend building a few thoughtful tools targeting specific high-impact workflows, which match your evaluation tasks and scaling up from there. In the address book case, you might choose to implement a `search_contacts` or `message_contact` tool instead of a `list_contacts` tool.

Tools can consolidate functionality, handling potentially *multiple* discrete operations (or API calls) under the hood. For example, tools can enrich tool responses with related metadata or handle frequently chained, multi-step tasks in a single tool call.

Here are some examples:

- Instead of implementing a `list_users`, `list_events`, and `create_event` tools, consider implementing a `schedule_event` tool which finds availability and schedules an event.
- Instead of implementing a `read_logs` tool, consider implementing a `search_logs` tool which only returns relevant log lines and some surrounding context.
- Instead of implementing `get_customer_by_id`, `list_transactions`, and `list_notes` tools, implement a `get_customer_context` tool which compiles all of a customer's recent & relevant information all at once.

Make sure each tool you build has a clear, distinct purpose. Tools should enable agents to subdivide and solve tasks in much the same way that a human would, given access to the same underlying resources, and simultaneously reduce the context that would have otherwise been consumed by intermediate outputs.

Too many tools or overlapping tools can also distract agents from pursuing efficient strategies. Careful, selective planning of the tools you build (or don't build) can really pay off.

Namespacing your tools

Your AI agents will potentially gain access to dozens of MCP servers and hundreds of different tools - including those by other developers. When tools overlap in function or have a vague purpose, agents can get confused about which ones to use.

Namespacing (grouping related tools under common prefixes) can help delineate boundaries between lots of tools; MCP clients sometimes do this by default. For example, namespacing tools by service (e.g., `asana_search`, `jira_search`) and by resource (e.g., `asana_projects_search`, `asana_users_search`), can help agents select the right tools at the right time.

We have found selecting between prefix- and suffix-based namespacing to have non-trivial effects on our tool-use evaluations. Effects vary by LLM and we encourage you to choose a naming scheme according to your own evaluations.

Agents might call the wrong tools, call the right tools with the wrong parameters, call too few tools, or process tool responses incorrectly. By selectively implementing tools whose names reflect natural subdivisions of tasks, you simultaneously reduce the number of tools and tool descriptions loaded into the agent's context and offload agentic computation from the agent's context back into the tool calls themselves. This reduces an agent's overall risk of making mistakes.

Returning meaningful context from your tools

In the same vein, tool implementations should take care to return only high signal information back to agents. They should prioritize contextual relevance over flexibility, and eschew low-level technical identifiers (for example: `uuid`, `256px_image_url`, `mime_type`). Fields like `name`, `image_url`, and `file_type` are much more likely to directly inform agents' downstream actions and responses.

Agents also tend to grapple with natural language names, terms, or identifiers significantly more successfully than they do with cryptic identifiers. We've found that merely resolving arbitrary alphanumeric UUIDs to more semantically meaningful and interpretable language (or even a 0-indexed ID scheme) significantly improves Claude's precision in retrieval tasks by reducing hallucinations.

In some instances, agents may require the flexibility to interact with both natural language and technical identifiers outputs, if only to trigger downstream tool calls (for example, `search_user(name='jane')` → `send_message(id=12345)`). You can enable both by exposing a simple `response_format` enum parameter in your tool, allowing your agent to control whether tools return `"concise"` or `"detailed"` responses (images below).

You can add more formats for even greater flexibility, similar to GraphQL where you can choose exactly which pieces of information you want to receive. Here is an example ResponseFormat enum to control tool response verbosity:

```
enum ResponseFormat {
   DETAILED = "detailed",
   CONCISE = "concise"
}
```

Copy

Here's an example of a detailed tool response (206 tokens):

```
● I'll search slack for recent bug reports and use the detailed format to see which channel IDs and threads to
investigate further.

● slack - search (MCP)(query: "bug", sort: "timestamp", sortDir: "desc", limit: 100, responseFormat:
"detailed")
 L Search results for: "bug"

    ═══ Result 1 of 89 ═══
    Channel: #dev (C1234567890)
    From: @jane.doe (U123456789)
    Time: 2024-01-15 10:30:45 UTC
    TS: 1705316445.123456
    Text: Found a critical bug in the login flow.

    ═══ Result 2 of 89 ═══
    Channel: DM with @john.smith
    From: @john.smith (U987654321)
    Time: 2024-01-14 15:22:18 UTC
    TS: 1705247738.234567
    Text: The bug report for issue #123 is ready for review
    Files: bug-report-123.pdf
    ...
```

Here's an example of a concise tool response (72 tokens):

```
● I'll search slack for recent bug reports and use the concise format to read as many messages as possible.

● slack - search (MCP)(query: "bug", sort: "timestamp", sortDir: "desc", limit: 100, responseFormat: "concise")
 L Search: "bug" (89 results)

    1. #dev - @jane.doe: Found a critical bug in the login flow. [Jan 15]
    2. DM - @john.smith: The bug report for issue #123 is ready for review [Jan 14]
    ...
```

Slack threads and thread replies are identified by unique `thread_ts` which are required to fetch thread replies. `thread_ts` and other IDs (`channel_id` , `user_id`) can be retrieved from a `"detailed"` tool response to enable further tool calls that require these. `"concise"` tool responses return only thread content and exclude IDs. In this example, we use ~⅓ of the tokens with `"concise"` tool responses.

Even your tool response structure—for example XML, JSON, or Markdown—can have an impact on evaluation performance: there is no one-size-fits-all solution. This is because LLMs are trained on next-token prediction and tend to perform better with formats that match their training data. The optimal response structure will vary widely by task and agent. We encourage you to select the best response structure based on your own evaluation.

Optimizing tool responses for token efficiency

Optimizing the quality of context is important. But so is optimizing the *quantity* of context returned back to agents in tool responses.

We suggest implementing some combination of pagination, range selection, filtering, and/or truncation with sensible default parameter values for any tool responses that could use up lots of context. For Claude Code, we restrict tool responses to 25,000 tokens by default. We expect the effective context length of agents to grow over time, but the need for context-efficient tools to remain.

If you choose to truncate responses, be sure to steer agents with helpful instructions. You can directly encourage agents to pursue more token-efficient strategies, like making many small and targeted searches instead of a single, broad search for a knowledge retrieval task. Similarly, if a tool call raises an error (for example, during input validation), you can prompt-engineer your error responses to clearly communicate specific and actionable improvements, rather than opaque error codes or tracebacks.

Here's an example of a truncated tool response:

```
● I'll find all of your transactions on Stripe and provide a summary for you.

● stripe - transactions_search (MCP)(limit: 5000, responseFormat: "concise")
  L ## Transaction Search Results

    Found **2,847 transactions** matching your query.

    The results are truncated. Showing first 3 results:

    | Date | Description | Amount | Category |
    ├──────┼─────────────┼────────┼──────────┤
    | 2024-01-15 | Payment from Acme Corp | +$5,200.00 | Revenue |
    | 2024-01-14 | Payment from TechStart | +$3,100.00 | Revenue |
    | 2024-01-13 | Payment from Cloud Co. | +$3,100.00 | Revenue |

    **Summary of all 2,847 results:**
    - Total Revenue: $458,291.00
    - Date Range: Jan 1 - Jan 15, 2024

    ## To refine these results, you can:

    - **Search for specific vendors**: Use `transactions_search(payee: "Acme Corp")` to see only Acme Corp
charges
    - **Filter by amount range**: Use `transactions_search(minAmount: 1000, maxAmount: 5000)`
    - **Get next page**: Use `transactions_search(query: <query>, page: 2)`
```

Here's an example of an unhelpful error response:

```
● Sure. I'll fetch John's contact information from his profile.

● asana - user_info (MCP)(userId: "john.doe@acme.corp")
 L {
     "error": {
       "code": "RESOURCE_NOT_FOUND",
       "status": 422,
       "message": "Invalid value",
       "details": {
         "field": "userId",
         "value": "john.doe@acme.corp",
         "type": "invalid_value"
       }
     }
   }
```

Here's an example of a helpful error response:

```
● Sure. I'll fetch John's contact information from his profile.

● asana - user_info (MCP)(userId: "john.doe@acme.corp")
 L Tool Response:

    # Resource Not Found: Invalid `userId`

    ## Error Summary
    Your request to `/api/user/info` failed because the `userId` `john.doe@acme.corp` does not exist or is in
the wrong format.

    ## Valid User IDs
    Examples:
    - `1928298149291729`
    - `9381719375914731`

    ## Resolving a User ID
    - Call user_search()
```

Tool truncation and error responses can steer agents towards more token-efficient tool-use behaviors (using filters or pagination) or give examples of correctly formatted tool inputs.

Prompt-engineering your tool descriptions

We now come to one of the most effective methods for improving tools: prompt-engineering your tool descriptions and specs. Because these are loaded into your agents' context, they can collectively steer agents toward effective tool-calling behaviors.

When writing tool descriptions and specs, think of how you would describe your tool to a new hire on your team. Consider the context that you might implicitly bring—specialized query formats, definitions of niche terminology, relationships between underlying resources—and make it explicit. Avoid ambiguity by clearly describing (and enforcing with strict data models) expected inputs and outputs. In particular, input parameters should be unambiguously named: instead of a parameter named `user`, try a parameter named `user_id`.

With your evaluation you can measure the impact of your prompt engineering with greater confidence. Even small refinements to tool descriptions can yield dramatic improvements. Claude Sonnet 3.5 achieved state-of-the-art performance on the SWE-bench Verified evaluation after we made precise refinements to tool descriptions, dramatically reducing error rates and improving task completion.

You can find other best practices for tool definitions in our Developer Guide. If you're building tools for Claude, we also recommend reading about how tools are dynamically loaded into Claude's system prompt. Lastly, if you're writing tools for an MCP server, tool annotations help disclose which tools require open-world access or make destructive changes.

Looking ahead

To build effective tools for agents, we need to re-orient our software development practices from predictable, deterministic patterns to non-deterministic ones.

Through the iterative, evaluation-driven process we've described in this post, we've identified consistent patterns in what makes tools successful: Effective tools are intentionally and clearly defined, use agent context judiciously, can be combined together in diverse workflows, and enable agents to intuitively solve real-world tasks.

In the future, we expect the specific mechanisms through which agents interact with the world to evolve—from updates to the MCP protocol to upgrades to the underlying LLMs themselves. 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.

Acknowledgements

Written by Ken Aizawa with valuable contributions from colleagues across Research (Barry Zhang, Zachary Witten, Daniel Jiang, Sami Al-Sheikh, Matt Bell, Maggie Vo), MCP (Theodora Chu, John Welsh, David Soria Parra, Adam Jones), Product Engineering (Santiago Seira), Marketing (Molly Vorwerck), Design (Drew Roper), and Applied AI (Christian Ryan, Alexander Bricken).

1Beyond training the underlying LLMs themselves.