프로MPT 엔지니어링

소프트웨어 엔지니어를 위한 AI 조작 원리

ThakiCloud (Hyojung Han)

프로MPT 엔지니어링

소프트웨어 엔지니어를 위한 AI 조작 원리

ThakiCloud (Hyojung Han)

Contents

1 프롬프트는 코드다

Figure 1.1: 프롬프트는 코드다

소프트웨어 엔지니어링에서 함수를 설계할 때 우리는 입력을 무엇으로 받을지, 출력을 무엇을 반환할지, 부수 효과는 무엇인지를 먼저 정의한다. 프롬프트도 본질적으로 같다. 프롬프트는 언어 모델이라는 시스템에 넘기는 함수 호출이며, 입력은 문맥이고 출력은 모델이 생성한 텍스트다. 이렇게 보면 프롬프트 작성은 단순한 글쓰기가 아니라 엔지니어링 작업이 된다.

1.1 프롬프트를 함수로 모델링하기

프롬프트를 함수 시그니처로 바라보자. 좋은 함수가 그렇듯 좋은 프롬프트도 이름이 명확하고 입력과 출력의 계약이 분명하다. 예를 들어 다음과 같이 쓸 수 있다.

```
def extract_people(text: str) -> list[Person]:
 """
```

```
    텍스트에서 인물 정보를 추출합니다.

    입력: 자연어로 기술된 텍스트
    출력: 사람 이름, 직함, 소속이 담긴 딕셔너리 리스트
    """
    prompt = f"""
    다음 텍스트에서 인물 정보를 추출하세요.

    텍스트: {text}

    결과를 JSON 배열로 반환하세요.
    """
    return call_model(prompt)
```

이 예제에서 눈여겨볼 점이 세 가지 있다. 입력을 텍스트로 제한해 모델이 받을 문맥의 범위를 명확히 하면 예측 가능한 출력을 기대할 수 있다. 출력 형태도 명시했다. 여기서는 JSON 배열을 요청했는데, 자연어 응답을 요구했다면 이 함수의 계약은 제멋대로 흐트러졌을 것이다. 마지막으로 계약 문서를 프롬프트 자체에 녹였다. 사람에게만 의도가 명확한 코드가 유지보수하기 어렵듯 프롬프트만으로는 계약이 모호해진다.

1.2 문맥은 매개변수다

함수에 어떤 인자를 넘기느냐에 결과가 달라지듯 프롬프트에 어떤 문맥을 주느냐가 결정적 차이를 만든다. 문맥 범주는 크게 세 가지다.

작업 지시는 모델에게 무엇을 해야 하는지 명령으로 전달하는 부분이고, 예제는 Few-Shot에서 다루는 영역이며, 입력 데이터는 그 자체로 세 번째 범주다. 이 셋을 어떻게 조합하느냐에 모델의 동작이 달라진다.

문맥을 구성할 때 흔한 실수는 모든 정보를 한 프롬프트에 담으려는 것이다.

모델의 문맥 창이 크다고 해서 전부 채우면 결과가 반드시 좋아지지는 않는다. 실제로는 관련 없는 정보가 많을수록 모델은 핵심 태스크에서 주의가 분산된다. 마치 함수가 필요 없는 매개변수까지 전부 받을 때 동작이 불안정해지는 것과 같다.

작업 지시는 선명하게 쓸수록 좋다. 모호한 표현보다 구체적인 동사를 쓴다. 예를 들어 “분석하세요”보다는 “긍정인지 부정인지 분류하고 그 근거를 한 문장으로 답하세요”가 결과의 분산을 줄인다. 이처럼 프롬프트도 타입을 명확히 하면 모델의 출력 포맷이 안정된다.

1.3 프롬프트도 디버깅한다

코드가 항상 처음부터 올바르게 동작하지 않듯 프롬프트도 대부분 실패와 수정을 반복한다. 다만 프롬프트의 실패는 컴파일 에러처럼 명쾌하지 않아서 체계적 접근이 필요하다.

실패 원인은 크게 세 갈래로 나뉜다. 지시가 모호하면(“간결하게 답하라”는 모델마다 다르게 해석된다) 결과가 흔들리고, 문맥이 부족하거나 과하면 출력이 어긋나며, 출력 형식 자체가 서로 맞지 않을 때도 있다. 한 프롬프트가 특정 입력에서만 실패한다면 그 입력의 엣지 케이스를 먼저 의심해야 한다.

디버깅 절차는 다음과 같다. 먼저 실패한 입력을 분리한다. 그 입력이 왜 다른지 문맥의 차이를 파악한다. 지시를 더 구체적으로 바꿔보고 결과를 비교한다. 이 과정을 반복하면서 프롬프트의 계약이 조금씩 단단해진다.

코드와 다른 점은 프롬프트의 동작이 모델 버전에 따라 달라질 수 있다는 사실이다. 같은 프롬프트라도 모델이 바뀌면 결과가 달라지곤 한다. 그래서 프롬프트 버전과 모델 버전을 함께 관리해야 한다.

1.4 정리

프롬프트는 함수의 계약과 같다. 입력을 제한하고, 출력을 명시하고, 문맥을 매개변수로 다루며, 실패를 체계적으로 디버깅한다. 이 관점을 갖추면 프롬프트 작성은 글쓰기가 아니라 엔지니어링 작업이 된다.

2 Few-Shot과 추론 체이닝

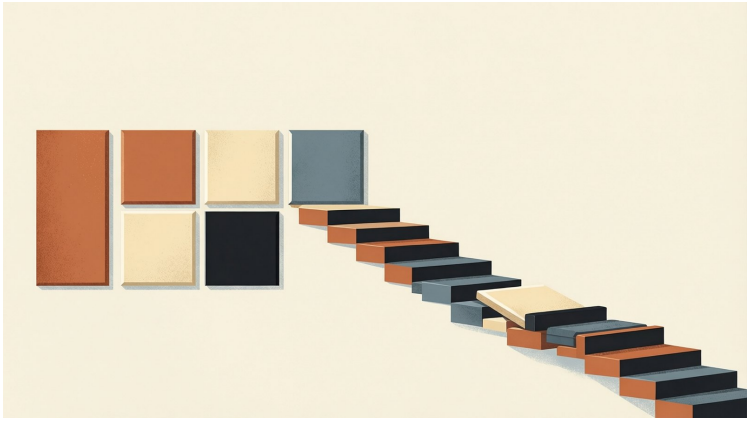

Figure 2.1: Few-Shot과 추론 체이닝

모델에게 예제를 어떻게 주입하느냐에 따라 결과의 품질이 크게 달라진다. Few-Shot 프롬프팅은 이 원리를 체계적으로 활용하는 기법이고, Chain-of-Thought는 추론 과정을 단계별로 유도하는 기법이다. 두 기법을 이해하면 모델이 단순히 답만 내놓는 수준을 넘어 올바른 추론 절차를 따라가도록 만들 수 있다.

2.1 Few-Shot의 원리

Zero-Shot은 명령만으로 결과를 요청하는 방식이다. 모델은 훈련 데이터에서 학습한 패턴을 스스로 판단해 답을 생성한다. Few-Shot은 여기에 예제 몇 개를 덧붙이는 기법이다. 예제로 모델에게 우리가 원하는 출력의 형태와 품질을 알려줄 수 있다.

예제는 단순한 참조가 아니라 모델의 출력 형식을 규정하는 계약이다. 예를 들어 감정 분류 태스크에서 다음과 같이 예제를 준다.

```
입력: 이 영화 정말 최고야
출력: 긍정

입력: 시간이 아깝다
출력: 부정

입력: 그저 그랬다
출력:
```

이 예제는 입력과 출력 사이의 관계를 보여준다. 모델은 이 패턴을 읽고 새로운 입력에 같은 규칙을 적용하며, 예제가 정확할수록 출력도 안정적으로 정확해진다.

예제를 고를 때 주의할 점이 있다. 예제는 작업의 경계 사례를 대표해야 한다. 쉬운 케이스만 예제로 넣으면 모델은 어려운 케이스에서 실패한다. 다양한 난이도와 레이블을 섞어야 한다. 예제의 수도 중요하다. 너무 적으면 패턴을 충분히 전달하지 못하고 너무 많으면 모델이 오히려 예제를 암기해 새로운 입력에 틀리게 적용하는 역효과가 생길 수 있다. 일반적으로 3개에서 8개 사이가 적당한 출발점이다.

2.2 Chain-of-Thought

Few-Shot이 출력의 형태를 가르쳐 준다면, Chain-of-Thought는 추론의 과정을 가르쳐 준다. 단계별 사고를 요청하면 모델은 최종 답변에 이르기까지의 중간 논리를 함께 생성한다.

다음은 산술 추론에서의 비교다. Chain-of-Thought 없이 요청하면 모델은 종종 잘못된 답을 바로 뱉는다. 하지만 다음과 같이 요청하면 정확도가 올라간다.

```
문장: 한 박스에 사과 12개가 들어있다. 5박스를 샀는데 3개가
↪ 썩어서 먹을 수 없게 되었다. 먹을 수 있는 사과는 총 몇
↪ 개인가?
단계별로 생각해 보세요.
```

모델은 다음과 같이 풀어간다. 5박스에 각각 12개이므로 총 60개이고, 그중 3개를 먹을 수 없으니 60에서 3을 빼면 57개다. 이 과정에서는 모델이 어디서부터 틀렸는지 추적할 수 있고, 필요하면 추론의 특정 단계만 수정하면 된다.

이 기법은 특히 복잡한 추론이 필요한 태스크에서 효과를 발휘한다. 수학 문제, 논리 퍼즐, 다단계 의사결정 문제 등에서 Chain-of-Thought를 쓰면 성능이 크게 오른다. 다만 추론 과정 자체가 필요 없는 단순 태스크에서는 오히려 불필요한 논리를 생성하게 만들어 효율이 떨어질 수 있다.

2.3 ReAct 패턴

ReAct는 “Reasoning”과 “Acting”을 결합한 패턴이다. 모델이 행동을 취하고 그 결과를 다시 추론에 반영하는 루프를 만든다. 단순한 텍스트 생성이나 분류가 아니라 환경과 상호작용하며 결론에 도달하는 종류의 작업에 적합하다.

예를 들어 질문에 답하면서 필요하면 도구를 사용하는 시스템에서 ReAct는 다음과 같이 동작한다. 먼저 질문을 분석하고 무엇을 모르는지 파악한다. 그다음 도구를 호출해서 정보를 가져온다. 결과를 바탕으로 다시 추론을 업데이트하고, 부족하면 다시 도구를 호출한다. 이 과정을 결론에 도달할 때까지 반복한다.

이 패턴에서는 모델이 스스로 판단해 언제 도구를 호출할지 결정한다. 사람이 미리 모든 경우를 프로그래밍하지 않아도 모델이 상황에 따라 행동을 선택한다. 다만 이 패턴은 모델이 자체 도구 호출을 지원해야 쓸 수

있어서 모든 API에 적용할 수 있는 것은 아니다.

2.4 실전 조합

이상의 기법은 단독으로도 효과적이지만 함께 쓸 때 시너지가 난다. Few-Shot이 출력 형식을 알려주고 Chain-of-Thought가 추론 과정을 유도하며 ReAct가 복합 작업을 처리하는 식이다.

구체적으로 적용하려면 먼저 태스크의 성격을 파악해야 한다. 단순 분류나 생성이라면 Few-Shot만으로 충분한 경우가 많다. 복잡한 추론이 필요하면 Chain-of-Thought를 추가한다. 환경과의 상호작용이 필요하면 ReAct를 고려한다. 대부분의 실무 태스크는 이 세 가지 기법을 조합해서 처리할 수 있다.

3 출력을 구조적으로 읽기

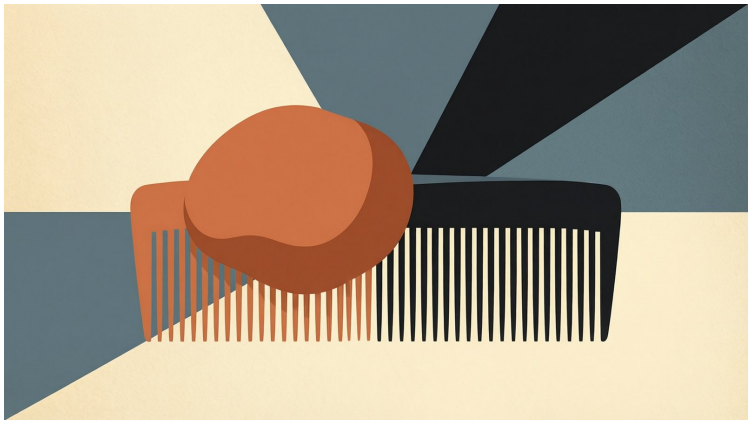

Figure 3.1: 출력을 구조적으로 읽기

프롬프트 결과가 자연어 텍스트로 돌아오면 사람이 읽기엔 편하지만 프로그램이 파싱하려면 손이 더 간다. 출력 형식을 강제할 수 있으면 파싱 오류를 미리 없애고 후속 시스템과의 통합도 안정적으로 가져갈 수 있다. 이 장에서는 JSON 모드, 스키마 기반 출력, 출력 검증 파이프라인을 다룬다.

3.1 JSON 모드와 강제 구조

대부분의 주요 모델 API는 출력을 JSON 형태로 강제하는 기능을 제공한다. 이 기능을 쓰면 모델이 자연어를 생성하다가도 지정한 JSON 구조를 따르게 된다. 예를 들어 인물 정보를 추출하는 태스크에서는 다음처럼 요청할 수 있다.

```
response = client.chat.completions.create(
 model="gpt-4o",
```

```
 messages=[{"role": "user", "content": prompt}],
 response_format={"type": "json_object"},
 schema={
 "type": "object",
 "properties": {
 "people": {
 "type": "array",
 "items": {
 "type": "object",
 "properties": {
 "name": {"type": "string"},
 "title": {"type": "string"},
 "organization": {"type": "string"}
 }
 }
 }
 }
 }
)
```

이렇게 하면 모델 출력은 항상 정의한 스키마를 따르는 JSON이 된다. 파싱 코드는 이 구조를 전제로 동작한다. 다만 이 모드에도 한계는 있다. 모델이 지시를 잘못 이해하면 스키마는 지키면서도 내용이 틀린 JSON을 만들 수 있다. 스키마가 있어도 내용의 정합성은 여전히 모델에게 맡기는 셈이다.

3.2 스키마 설계의 원칙

스키마는 출력의 계약이다. 스키마가 흐트러지면 후속 시스템이 파싱하지 못하거나 잘못된 데이터로 동작할 수 있다. 스키마 설계에서 중요한 원칙 몇 가지를 짚어보자.

필드 이름은 명확하게 짓는다. "result"보다는 "extracted_people"처럼 구체적으로 쓰는 편이 낫다. 각 필드의 타입과 범위도 명시해야 한다. 문자열인지 배열인지, 가능한 값의 목록은 무엇인지를 제한할수록 파싱이 안정된다. 필수 필드와 선택 필드를 구분하는 것도 중요한데, 필수 필드가 빠지면 그 자리에서 바로 실패로 처리할 수 있어야 한다.

스키마를 만드는 과정은 코드와 다르지 않다. 처음에는 간단한 구조로 시작해서 실패 케이스가 나올 때마다 필드를 추가하거나 타입을 강화해나간다. 그러다 보면 스키마가 실제 사용 패턴을 자연스럽게 반영하게 된다.

3.3 출력 검증 파이프라인

모델의 출력을 바로 신뢰해서는 안 된다. 반드시 검증 단계를 거쳐야 완전한 처리라 할 수 있다. 검증은 크게 세 갈래로 나뉜다.

구조 검증에서는 JSON이 유효한지, 스키마를 따르는지를 확인한다. 이 단계에서 대부분의 파싱 오류가 걸러진다. 의미 검증은 한 단계 더 들어가서 "name" 필드에 실제로 사람 이름이 들어 있는지, 날짜 필드가 유효한 범위인지를 살핀다. 이 단계는 정규 표현식이나 범위 검사로 처리할 수 있다. 마지막으로 비즈니스 로직 검증이 있는데, "주문 금액이 음수이면 안 된다"처럼 태스크에 특화된 규칙을 적용하는 단계다.

검증에 실패하면 어떻게 할 것인가. 가장 단순한 방법은 예외를 던지는 것이다. 하지만 실제로는 재시도하거나 기본값을 반환하거나 사용자에게 알려주는 등 상황에 맞는 전략이 필요하다. 중요한 것은 검증 실패를 조용히 넘기지 않는 것이다.

3.4 파싱 실패 디버깅

모델이 요청한 형식으로 출력하지 않는 경우가 있다. 원인은 여러 가지다. 지시가 충분히 명확하지 않았거나, 스키마가 너무 복잡했거나, 모델의 학습 데이터가 다른 방향을 선호했을 수 있다.

체계적인 디버깅 접근은 다음과 같다. 먼저 실패한 출력 자체를 로그에 남긴다. 원본 출력을 보는 것이 수정의 첫걸음이다. 그다음 실패 유형을 분류한다. 완전한 JSON이 아닌 경우가 있고, JSON이더라도 스키마를 따르지 않는 경우가 있다. 전자라면 지시를 더 명확하게 쓰거나 예제를 추가하고, 후자라면 스키마를 조정하거나 검증 로직에서 허용 범위를 넓힌다.

3.5 실전 패턴

구조화된 출력을 사용하는 실전 시스템에서 자주 보는 패턴이 있다. 하나는 응답을 "안전 게이트"로 감싸는 것이다. 모델의 출력을 검증해서 통과한 것만 실제 시스템에 넘기는 방식이다. 다른 하나는 fallback 구조다. JSON 모드에서 실패하면 일반 텍스트로 돌아가서 정규 표현식으로 파싱을 시도하는 이중 안전장치다.

이 패턴들의 공통점은 모델 출력을 신뢰 지점으로 다루지 않는다는 데 있다. 모델은 강력한 생성기이지만 정확한 구조까지 보장하지는 않는다. 그래서 항상 검증과 함께 써야 한다.

4 프롬프트도 버전 관리한다

Figure 4.1: 프롬프트도 버전 관리한다

코드와 마찬가지로 프롬프트에도 변경 이력이 필요하다. 같은 입력을 넣었을 때 프롬프트 버전에 따라 결과가 어떻게 달라지는지 추적할 수 있어야 하고, 문제가 생기면 이전 버전으로 되돌릴 수 있어야 하며, 팀원 누구나 지금 쓰이는 프롬프트가 무엇인지 알 수 있어야 한다. 이 장에서는 프롬프트 버전 관리의 실전 방법과 평가 프레임워크, 조직 차원의 운영까지 다룬다.

4.1 프롬프트도 Git처럼 관리한다

프롬프트 문자열을 데이터베이스나 설정 파일 대신 Git으로 관리하면 코드와 똑같은 워크플로를 그대로 쓸 수 있다. 변경 사항은 커밋으로 남고 Pull Request로 리뷰받으며, 필요하면 이전 버전으로 롤백한다.

실제로는 프롬프트를 템플릿 형태로 저장하는 방식을 쓴다. 템플릿에는 플레이스홀더를 넣어두고 런타임에 값을 채워 넣는다. 이렇게 분리해두면

변경 이력을 볼 때 지시문 자체가 바뀐 건지 파라미터만 바뀐 건지 구분할 수 있다.

예시 디렉터리 구조는 다음과 같다.

```
prompts/
 extraction/
 v1_extract_people.j2
 v2_extract_people.j2
 classification/
 v1_sentiment.j2
 CHANGELOG.md
```

각 버전의 프롬프트 파일은 Jinja2 템플릿으로 작성하고 CHANGELOG에는 변경 이유와 결과를 남긴다. 이렇게 구조를 잡아두면 나중에 문제가 생겼을 때 원인을 찾고 롤백하기가 수월하다.

4.2 평가 프레임워크

프롬프트도 테스트가 필요하다. 가장 기본적인 테스트는 정답 데이터셋으로 정확도를 재는 것이다. 입력과 기대 출력을 미리 준비해두고 현재 프롬프트가 어느 정도 정확도를 내는지 측정한다.

정확도 외에 다른 지표도 함께 봐야 한다. 일관성은 같은 입력을 여러 번 실행했을 때 매번 같은 출력이 나오는지 보는 지표이고, 구조 일치는 출력이 요청한 형태를 항상 따르는지 확인하는 지표다. 이 세 가지를 함께 추적하면 프롬프트 품질을 여러 각도에서 볼 수 있다.

평가는 자동화해야 의미가 있다. CI 파이프라인에 평가 단계를 넣어두면 새 프롬프트가 기존 성능보다 떨어질 때 자동으로 실패 처리할 수 있다. 이렇게 해두면 프롬프트 변경도 코드 변경 못지않게 품질 관리를 받는다.

4.3 A/B 테스트

개선하려는 프롬프트가 있다면 A/B 테스트로 효과를 검증한다. 트래픽을 절반씩 나눠 한쪽에는 기존 프롬프트를, 다른 쪽에는 새 프롬프트를 적용하고 결과를 비교하면 된다. 단순하지만 강력하다.

A/B 테스트에는 주의할 점이 있다. 테스트 기간이 너무 짧으면 결과가 우연에 좌우될 수 있다. 평가 지표도 미리 명확히 정해야 한다. 정확도를 높이려 했는데 응답 속도가 느려졌다면, 이걸 성공으로 볼지 실패로 볼지 판단할 기준이 있어야 한다. 그래서 테스트를 시작하기 전에 성공 조건부터 명시해둔다.

4.4 프롬프트 카탈로그 운영

팀에서 쓰는 프롬프트가 늘어나면 관리 시스템이 필요해진다. 프롬프트 카탈로그를 두면 팀원 누구나 지금 어떤 프롬프트가 쓰이는지, 각각 언제 업데이트됐는지, 어떤 평가 결과를 냈는지를 한눈에 볼 수 있다.

카탈로그에 담을 정보는 최소한 다음과 같다. 프롬프트의 이름과 용도, 현재 버전 번호, 마지막 업데이트 일자, 기대 정확도 또는 평가 결과, 사용 예시. 이런 정보가 한 곳에 모여 있으면 팀원들이 기존에 만든 것을 재활용할 수 있고 불필요한 중복도 줄어든다.

카탈로그 운영의 핵심은 업데이트를 부담 없이 하게 만드는 데 있다. 절차가 번거로우면 사람들은 금방 업데이트를 미룬다. 평가가 자동화돼 있고 변경 절차가 단순할수록 카탈로그는 건강하게 유지된다.

4.5 팀에서 프롬프트 협업

코드 품질을 관리하듯 프롬프트 품질도 팀 차원에서 관리해야 한다. 개인이 만든 프롬프트가 공유되지 않으면 같은 문제를 팀원 각자가 따로 풀게 되는

비효율이 생긴다.

협업은 먼저 팀이 지금 프롬프트를 어떻게 쓰고 있는지 파악하는 데서 시작한다. 그다음 공통적으로 많이 쓰이는 태스크부터 표준화하고, 표준 프롬프트의 기준선을 정해 개선 사항이 생기면 팀 전체에 공유하는 구조를 갖춘다.

이때 리뷰 프로세스도 함께 설계해두면 좋다. 중요한 프롬프트 변경은 코드 리뷰처럼 다른 팀원의 검토를 거치게 한다. 리뷰어는 프롬프트가 명확한지, 예상 출력이 적절한지, 기존 버전 대비 바뀐 부분이 타당한지를 살핀다. 이 과정이 습관으로 자리 잡으면 프롬프트 품질은 자연스레 올라간다.

4.6 정리

프롬프트는 만들어서 끝이 아니다. 버전을 관리하고 평가하고 개선하는 과정까지 거쳐야 비로소 엔지니어링 작업이라 부를 수 있다. 코드를 대하던 태도 그대로 프롬프트를 대하면, AI를 훨씬 정밀한 도구로 쓸 수 있게 된다.