AI API 엔지니어링

LLM을 신뢰할 수 있는 시스템 구성 요소로 사용하기

ThakiCloud (Hyojung Han)

AI API 엔지니어링

LLM을 신뢰할 수 있는 시스템 구성 요소로 사용하기

ThakiCloud (Hyojung Han)

Contents

1 JSON 스키마 강제: 응답 형식 통제

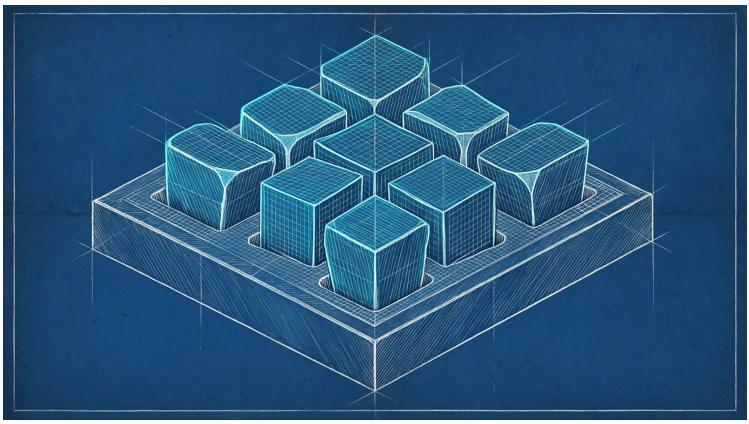

Figure 1.1: JSON 스키마 강제: 응답 형식 통제

LLM API를 시스템에 통합할 때 가장 먼저 부딪히는 벽이 있다. 모델이 반환하는 답변이 항상 다르다. 오늘은 `{"result": "ok"}`를 반환하던 모델이 내일은 `{"status": "success", "data": null}`을 반환한다. 이 변동성을 통제하지 못하면 그 아래의 모든 로직이 흔들린다.

이 장에서는 구조화된 출력을 강제하는 실질적 기법을 다룬다. 단순히 "system prompt에 지시하라"는 말이 아니다. 재시도 회로와 타입 안전성, 비용 요인까지 설계에 포함해 이야기한다.

1.1 스키마 설계의 세 가지 지주

스키마가 작동하려면 모델이 해당 구조를 이해할 수 있어야 한다. 그러나 이해 가능성과 제약 강도 사이에는 트레이드오프가 있다.

첫 번째 지주: enum으로 선택지를 줄인다

모델이 출력할 수 있는 값을 명시적으로 열거하면 그 선택지 밖의 응답이 올

확률이 급격히 줄어든다. 특히 상태 코드, 분류 라벨, 에러 유형에 enum이 효과적이다.

```
"status": {
 "type": "string",
 "enum": ["pending", "processing", "completed", "failed"]
}
```

이 스키마는 모델이 `"status": "maybe"`라고 반환할 가능성을 구조적으로 배제한다. 다만 enum 값이 10개를 넘기면 효과가 줄어든다 [추정]. 모델은 열거 크기에 비례해 실수를 늘린다.

두 번째 지주: const로 고정값 강제

응답의 특정 필드가 항상 같은 값이어야 할 때 const를 쓴다. API 버전, 고정 메타데이터, 구분자 역할을 하는 필드에 유용하다.

```
"version": {
 "type": "string",
 "const": "1.0"
}
```

세 번째 지주: description으로 의도 전달

LLM은 자연어를 이해한다. 스키마의 description 필드에 모델에게 해야 할 일을 자연어로 쓰면 지시 효과가 크게 올라간다.

```
"action": {
 "type": "string",
 "description": "사용자 요청의 의도를 파악하고, 가능한 한 짧게
 ↪ 한 문장으로 요약하세요. 해석할 수 없는 요청은 'unclear'로
 ↪ 응답하세요."
}
```

description이 없으면 모델은 자기 방식대로 해석한다. 있으면 그 해석을

description 방향으로 유도할 수 있다.

1.2 Pydantic + Instructor: Python의 타입 안전한 스키마 추출

Pydantic은 Python에서 가장 널리 쓰이는 데이터 검증 라이브러리다. Instructor는 Pydantic 모델만 정의하면 LLM이 해당 구조를 자동으로 반환하도록 하는 래퍼다.

```
from pydantic import BaseModel, Field
from openai import OpenAI
from instructor import from_openai

class ExtractionResult(BaseModel):
 name: str = Field(description="회사 이름")
 founded_year: int = Field(description="설립 연도, 숫자만")
 employee_count: str = Field(description="직원 수 범위: ~50,
↪  ~200, ~1000, ~5000+")

client = from_openai(OpenAI())
result = client.chat.completions.create(
 model="gpt-4o",
 messages=[{"role": "user", "content": "ThakiCloud에 대해
↪  알려줘"}],
 response_model=ExtractionResult
)
```

이 코드가 하는 일은 단순하다. ExtractionResult라는 스키마를 정의하면, Instructor가 그 스키마에 맞는 응답을 모델에게 요구하고, 모델이 반환한 텍스트에서 해당 필드를 추출까지 해준다.

재시도가 필요한 지점은 세 곳이다. 먼저 모델이 스키마를 따르지 않았을

때의 재시도다. Instructor는 기본적으로 유효성 검증에 실패하면 예외를 던지므로, 이를 캐치해 재시도하면 된다.

```
MAX_RETRIES = 3

for attempt in range(MAX_RETRIES):
 try:
 return client.chat.completions.create(
 model="gpt-4o",
 messages=messages,
 response_model=ExtractionResult,
 max_retries=0 # Instructor 내부 재시도 비활성화
 )
 except ValidationError:
 if attempt == MAX_RETRIES - 1:
 raise
 # 프롬프트에 스키마 위반 사례를 추가해 재시도
 messages.append({
 "role": "user",
 "content": f"이전 응답이 스키마를 따르지 않았습니다. 다시
 ↪ 시도하세요."
 })
```

두 번째는 rate limit 에러에 대한 재시도다. instructor의 retry 파라미터를 활용하면 자동으로 처리할 수 있지만, 지수 백오프를 직접 구현하면 더 세밀한 제어가 가능하다.

마지막은 모델이 스키마를 아예 어긴 게 아니라 일부만 어긴 경우다. 이때는 부분 유효성 검사(partial validation)를 허용하고, 누락된 필드에 기본값을 채우는 전략을 쓸 수 있다.

1.3 재시도 회로 설계: 스키마 실패를 위한 전용 회로

모든 재시도가 동일하게 작동하진 않는다. 스키마 validation 실패는 모델이 "길을 잃었을 때" 발생하므로, 재시도할 때마다 프롬프트를 보강해 주는 것이 효과적이다.

단순히 같은 프롬프트를 N번 보내는 것은 효과적이지 않다. 모델이 스키마를 잘못 해석했다면, 같은 프롬프트를 다시 보내도 같은 잘못된 해석을 반복한다.

효과적인 전략은 "스키마 실패 전용 프롬프트"를 별도로 준비하는 것이다.

```
SCHEMA_RETRY_PROMPT = """
응답이 요청된 JSON 스키마를 따르지 않았습니다.
요청된 형식:
{schema_description}

지시사항:
- response_schema에 명시된 필드만 사용해세요
- 추가 필드를 만들지 마세요
- 모든 필수 필드를 포함하세요
- 필드 값은 요청된 타입과 설명에 맞게 채우세요
"""
```

이 프롬프트를 재시도 시 맨 앞에 삽입하면 모델이 이전의 잘못된 해석에서 벗어날 가능성이 높아진다.

재시도 회로에는 circuit breaker도 함께 운영한다. 같은 엔드포인트에서 스키마 validation이 연속 N번 실패하면 일정 시간 동안 해당 요청을 차단하고 빠른 실패를 반환한다. 모델의 일시적 비정상 상태를 흡수하면서 연쇄 장애로 번지는 것을 막는 장치다.

1.4 비용 요인: 스키마 복잡성과 토큰 소모

스키마가 복잡해질수록 모델이 그 스키마를 따라 출력을 구성하는 데 더 많은 토큰을 소모한다. 이는 곧 비용과 직결된다.

예를 들어, 5개의 필드를 가진 스키마와 20개의 필드를 가진 스키마가 있다고 하자. 응답 길이의 차이는 평균적으로 2~3배에 이른다 [추정]. 모델은 항상 가장 간결한 유효한 출력을 생성하려 하지만, 스키마가 복잡해지면 선택지가 넓어져 불필요한 필드까지 채우려는 경향이 생긴다.

비용 최적화를 위한 원칙은 세 가지로 정리할 수 있다.

먼저 필드 수를 최소화한다. 실제로 필요한 데이터만 스키마에 포함시키고, 디버깅용 임시 필드는 프로덕션에 남기지 않는다.

다음으로 nested 구조를 평탄하게 만든다. 중첩된 객체 대신 단일 레벨 필드를 쓰면 토큰이 절약된다.

마지막으로 description을 간결하게 쓴다. description도 토큰으로 계산되므로, 핵심 의도만 남기고 과한 설명은 뺀다.

1.5 마무리

JSON 스키마 강제는 단순히 Pydantic 모델을 정의하는 일이 아니다. 모델의 행동 공간을 제한하는 설계이며, 재시도 회로와 비용 요인까지 포함한 시스템 설계다. 이 장의 핵심은 다음과 같다.

스키마는 세 가지 지주, enum, const, description으로 구성한다. Instructor로 Python에서 타입 안전한 추출을 구현하고, 스키마 실패 전용 재시도 프롬프트를 준비한다. Circuit breaker로 연쇄 장애를 방지하며, 스키마 복잡성은 비용과 직결되므로 필드 수와 깊이를 최소화한다.

2 Partial Streaming: 실시간 피드백 구현

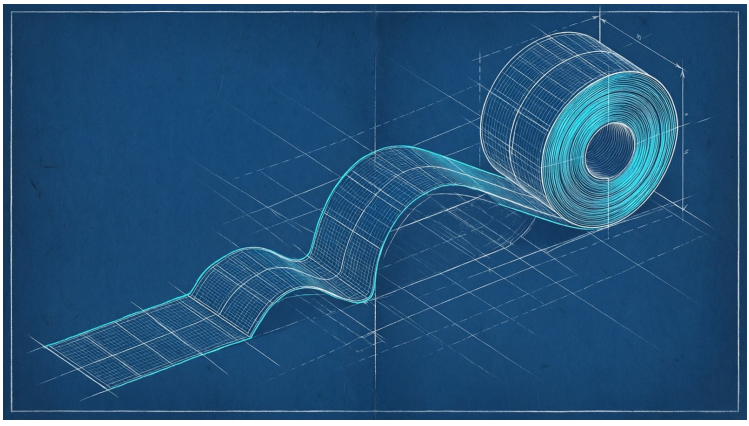

Figure 2.1: Partial Streaming: 실시간 피드백 구현

사용자는 긴 응답을 기다리는 동안 아무 피드백이 없으면 불안해진다. 로딩 스피너가 3초만 넘어가도 "뭔가 잘못된 거 아냐"라는 생각이 든다. LLM API 연동에서 이 문제를 푸는 방법이 스트리밍이다.

그러나 단순히 토큰을 하나씩 받는 것만으로는 부족하다. 부분 결과를 어떻게 의미 있게 구성하느냐가 실제 사용자 경험을 좌우한다. 이 장에서는 Server-Sent Events(SSE)와 WebSocket 중 무엇을 선택할지, 의미 단위로 어떻게 청킹할지, 재연결은 어떻게 처리할지, 그리고 스트리밍 환경의 Observability까지 다룬다.

2.1 SSE vs WebSocket: 선택 기준

실시간 통신 기술은 다양하지만, LLM API 연동에서 실질적인 선택지는 두 가지다.

Server-Sent Events(SSE)는 단방향 통신이다. 서버에서 클라이언트로만

데이터가 흐른다. 구현이 단순하고 HTTP/2를 쓰면 다중화도 지원된다. 대부분의 LLM API Provider가 SSE를 기본 전송 방식으로 쓴다. LLM 응답을 클라이언트로 스트리밍하는 용도라면 SSE로 충분하다.

WebSocket은 양방향 통신이다. 클라이언트가 서버로 데이터를 보낼 수 있어서 스트리밍 도중 사용자가 개입해야 하는 경우(중간 취소, 추가 지시 등)에 적합하다. 다만 구현 복잡도가 높고 별도의 유지보수 인프라가 필요하다.

결정 기준은 결국 "클라이언트가 서버에 보낼 것이 있는가"다. 대부분의 AI 채팅 인터페이스는 SSE만으로 충분하고, 사용자가 스트리밍 도중 끼어들어야 한다면 그때 WebSocket을 고려하면 된다.

SSE 구현은 Python의 FastAPI가 가장 간결하다.

```
from fastapi import FastAPI
from fastapi.responses import StreamingResponse
import openai

app = FastAPI()

async def stream_response(prompt: str):
 client = openai.AsyncOpenAI()

 async def event_generator():
 stream = await client.chat.completions.create(
 model="gpt-4o",
 messages=[{"role": "user", "content": prompt}],
 stream=True
 )

 for chunk in stream:
 delta = chunk.choices[0].delta.content
```

```
if delta:
# SSE 형식으로 전송
yield f"data: {delta}\n\n"

yield "data: [DONE]\n\n"

return StreamingResponse(
event_generator(),
media_type="text/event-stream"
)
```

핵심은 stream=True로 요청하고 chunk.choices[0].delta.content에서 토큰 단위 증분을 뽑아내는 데 있다. 이걸 SSE 형식으로 감싸 전송하면 된다.

2.2 의미 단위 청킹: 토큰 단위보다 나은 사용자 경험

토큰 단위 스트리밍의 문제는 사용자 눈에 보이는 결과가 문법적으로 깨져 있다는 데 있다. "안녕하세요"라는 문장이 4개의 토큰으로 나뉘어 전송되면 클라이언트에는 "안", "녕", "하", "세", "요"가 하나씩 따로 도착한다.

이 문제를 푸는 방법이 의미 단위 청킹이다. 문장이나 단어, 절 같은 의미적 경계에서만 클라이언트에 데이터를 넘긴다.

구현 방법은 크게 두 가지다. 하나는 서버 사이드 버퍼링으로, 토큰을 모아뒀다가 의미적 경계가 확인되면 한 번에 전송하는 방식이다.

```
import re

def split_into_sentences(text: str) -> list[str]:
 # 한국어 문장 분리
 sentences = re.split(r'(?<=[다고])[\.\?\!]|\n', text)
```

```
 return [s.strip() for s in sentences if s.strip()]

async def buffered_event_generator(prompt: str):
 buffer = ""
 MIN_CHUNK_SIZE = 5 # 최소 5토큰 이상 모이면 전송

 async for chunk in stream:
 buffer += chunk

 # 문장 경계에서 flush
 sentences = split_into_sentences(buffer)
 if len(sentences) > 1:
 # 마지막 문장은 버퍼에 남김
 for sentence in sentences[:-1]:
 yield f"data: {sentence}\n\n"
 buffer = sentences[-1]

 # 버퍼가 너무 커지면 강제 flush
 elif len(buffer) > MIN_CHUNK_SIZE * 3:
 yield f"data: {buffer}\n\n"
 buffer = ""
```

다른 하나는 클라이언트 사이드 버퍼링이다. 서버는 기존대로 토큰을 보내고 클라이언트가 모아서 문장 단위로 보여준다. 서버 부담은 줄지만 클라이언트 구현이 복잡해진다.

둘 중 무엇이 나은지는 use case에 따라 다르다. 서버 자원이 여유롭다면 서버 사이드 버퍼링 쪽이 사용자 경험 면에서 낫고, 클라이언트가 다양한 환경에서 돌아가야 한다면 클라이언트 사이드 버퍼링이 더 유연하다.

2.3 스트림 상태 관리

스트리밍 환경의 상태 관리는 단순해 보이지만 함정이 많다. 재연결, 부분 결과 처리, 정리(cleanup) 로직을 빠뜨리면 메모리 누수와 데이터 불일치로 이어진다.

먼저 **재연결 핸들링**을 본다. 클라이언트와 서버 사이 연결이 끊어지면 어디까지 처리됐는지 추적해야 한다. SSE에서는 `Last-Event-ID` 헤더로 마지막에 받은 이벤트 ID를 서버에 알려주면, 서버가 그 이후부터 다시 전송해준다.

```
async def stream_with_recovery(prompt: str, last_event_id: int =
↪  None):
 client = openai.AsyncOpenAI()
 stream = await client.chat.completions.create(
 model="gpt-4o",
 messages=[{"role": "user", "content": prompt}],
 stream=True
 )

 buffer = []
 start_index = last_event_id or 0

 async for idx, chunk in async_enumerate(stream,
 ↪  start=start_index):
 delta = chunk.choices[0].delta.content
 if delta:
 buffer.append(delta)
 yield f"id: {idx}\ndata: {delta}\n\n"

 # 스트림 종료 시 전체 결과를 메타데이터와 함께 전송
 full_text = "".join(buffer)
```

```
yield f"id: {idx + 1}\nevent: done\ndata: {full_text}\n\n"
```

부분 결과 처리는 클라이언트가 스트리밍 도중 데이터를 활용할 때 문제가 된다. 예를 들어 스트리밍되는 텍스트를 실시간으로 요약해 보여주고 싶다면 부분 결과를 다루는 로직이 필요하다. 이럴 때는 부분 결과를 상태로 저장해두고, 스트림이 완료된 시점에야 확정 처리를 하는 편이 안전하다.

정리 로직은 스트림이 정상 종료되든 오류로 중단되든 상관없이 실행돼야 한다. Python에서는 `try/finally` 블록이나 async context manager로 구현하면 된다.

```
class StreamContext:
 def __init__(self, prompt: str):
 self.prompt = prompt
 self.accumulator = []
 self.done = False

 async def __aenter__(self):
 self.stream = await client.chat.completions.create(
 model="gpt-4o",
 messages=[{"role": "user", "content": self.prompt}],
 stream=True
 )
 return self

 async def __aexit__(self, exc_type, exc_val, exc_tb):
 # 스트림 정리
 if hasattr(self, 'stream'):
 await self.stream.aclose()
 self.done = True
 return False # 예외를 먹지 않음
```

2.4 스트리밍 환경에서의 Observability

일반 HTTP 요청은 응답 시간, 상태 코드, 오류 여부만 추적하면 충분했다. 스트리밍에서는 그보다 훨씬 풍부한 데이터를 모을 수 있는데, 문제는 그 데이터를 어떻게 수집하고 분석하느냐다.

지연 측정은 스트리밍의 핵심 지표다. 첫 토큰까지의 시간(Time to First Token, TTFT)과 전체 스트림 완료 시간을 나눠서 재야 한다. TTFT가 길다면 모델의 처리 시간(processing time)이 긴 것이고, TTFT는 짧은데 완료 시간이 길다면 출력 결과 자체가 긴 것이다.

```
import time
from dataclasses import dataclass

@dataclass
class StreamMetrics:
 ttft_seconds: float
 total_duration_seconds: float
 total_tokens: int
 tokens_per_second: float
 error: str = None

async def measure_stream(prompt: str) -> StreamMetrics:
 start = time.perf_counter()
 ttft = None
 token_count = 0

 try:
 stream = await client.chat.completions.create(
 model="gpt-4o",
 messages=[{"role": "user", "content": prompt}],
 stream=True
```

```
)

async for chunk in stream:
if ttft is None:
ttft = time.perf_counter() - start
token_count += 1

total_duration = time.perf_counter() - start
return StreamMetrics(
ttft_seconds=ttft,
total_duration_seconds=total_duration,
total_tokens=token_count,
tokens_per_second=token_count / total_duration
)
except Exception as e:
return StreamMetrics(
ttft_seconds=time.perf_counter() - start,
total_duration_seconds=time.perf_counter() - start,
total_tokens=token_count,
tokens_per_second=0,
error=str(e)
)
```

품질 추적은 스트리밍 환경에서 특히 까다롭다. 스트림이 끝나기 전에는 전체 품질을 판단할 수 없기 때문이다. 대신 부분 결과의 품질 점수를 추정하는 방법을 쓸 수 있다. 비속어 검출, 문법 오류율, 반복 패턴 감지 같은 걸 실시간으로 돌려서 품질 저하가 잡히면 조기 종료를 시도하는 식이다.

2.5 마무리

Partial streaming은 단순히 데이터를 나눠 보내는 게 아니다. 의미 단위 청킹으로 사용자 경험을 끌어올리고, 재연결과 상태 관리로 안정성을 확보하고, TTFT와 품질 지표로 Observability까지 갖춰야 비로소 프로덕션에 어울리는 스트리밍이 된다. SSE와 WebSocket 사이의 선택은 양방향성이 필요한지로 갈리고, 서버 사이드 버퍼링 쪽이 사용자 경험에서 더 나은 경우가 많다.

3 재시도와 배압: 결함 허용 아키텍처

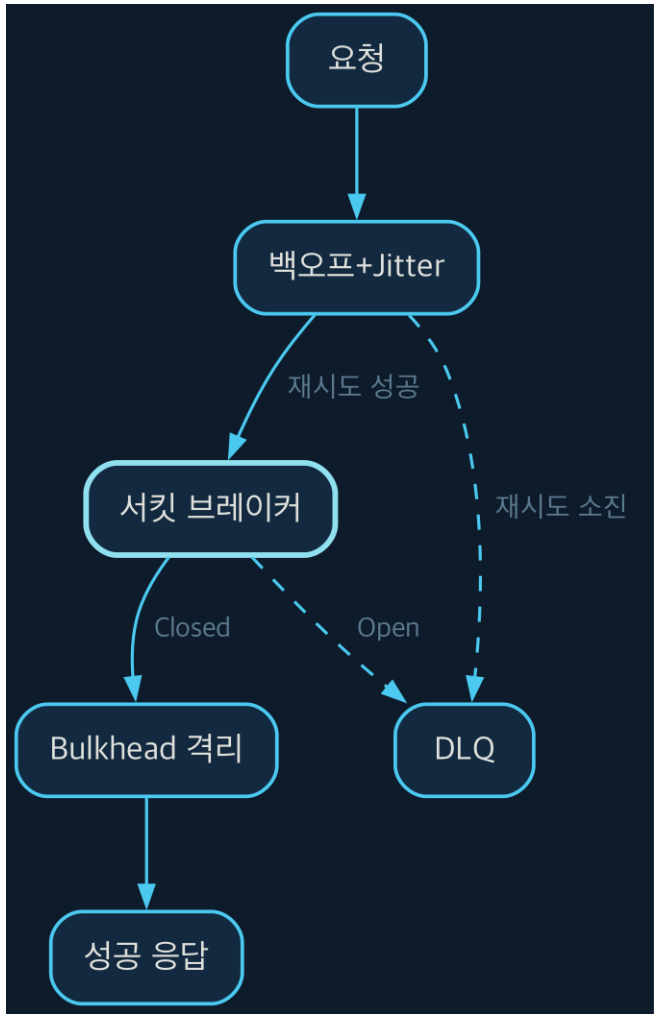

Figure 3.1: 재시도와 배압: 결함 허용 아키텍처

LLM API는 때때로 실패한다. Rate limit에 도달하거나, 서버가 일시적으로 불안정하거나, 네트워크 문제가 생긴다. 이때 시스템 전체가 마비되면 안 되므로, 재시도와 배압(backpressure) 메커니즘을 갖춰야 한다.

이 장에서는 단순한 재시도를 넘어서는 패턴들을 다룬다. 지수 백오프와 jitter의 정확한 작동 방식, circuit breaker의 상태 기계, bulkhead 패턴으로 리소스를 격리하는 방법, 그리고 실패한 요청을 잃어버리지 않는 dead letter queue까지 살펴본다.

3.1 지수 백오프와 jitter

단순히 1초 대기 후 재시도하면 문제가 있다. 실패한 요청이 동시에 대량으로 몰리면, 서버에 다시 부하를 줘서 연속 실패를 초래한다. 이 현상을 thundering herd problem이라 한다.

지수 백오프는 실패할 때마다 대기 시간을 2배로 늘리는 방식이다. 1초, 2초, 4초, 8초 식으로 대기한다.

```
import asyncio
import random

async def exponential_backoff(
 attempt: int,
 base_delay: float = 1.0,
 max_delay: float = 60.0
) -> float:
 """attempt 번째 재시도에서의 대기 시간 계산"""
 delay = min(base_delay * (2 ** attempt), max_delay)
 return delay

async def retry_with_backoff(coro, max_attempts=5):
 for attempt in range(max_attempts):
 try:
 return await coro()
 except RetryableError as e:
 if attempt == max_attempts - 1:
```

```
            raise

        delay = await exponential_backoff(attempt)
        await asyncio.sleep(delay)
```

이 코드에는 개선의 여지가 있다. 같은 시간에 재시도하는 요청들이 여전히 동시에 도달한다. 이를 해결하는 것이 jitter다.

Jitter는 대기 시간에 랜덤성을 추가한다. 세 가지 전략이 있다. Full jitter는 `base * 2^attempt` 범위 내에서 완전 랜덤, Equal jitter는 `base * 2^attempt / 2`에 랜덤 범위를 곱한다.

```
async def exponential_backoff_with_jitter(
    attempt: int,
    base_delay: float = 1.0,
    max_delay: float = 60.0,
    jitter_ratio: float = 0.5
) -> float:
    """Equal jitter가 적용된 백오프"""
    delay = min(base_delay * (2 ** attempt), max_delay)
    half = delay * jitter_ratio
    return half + random.uniform(0, half)
```

Equal jitter를 적용하면 재시도 요청들이 동일한 시간에 집중되지 않고 분산된다. 실무에서 가장 효과적인 것은 **Decorrelated jitter**다. 이전 대기 시간에 비례해서 랜덤성을 부여하는 방식으로, 기존 패턴보다 더 나은 분산을 보인다.

3.2 Circuit Breaker: 연쇄 장애 차단

재시도만으로는 불충분하다. 백오프 끝에서 재시도해도 계속 실패하는 요청이 있으면, 그 요청이 시스템 자원을 점유하며 다른 요청까지 끌어내린다. Circuit breaker는 이 문제를 해결한다.

Circuit breaker는 세 가지 상태를 가진다. **Closed**는 정상 동작 상태다. 요청이 통과하고, 실패율이 임계치를 넘으면 Open으로 전환된다. **Open**은 차단 상태다. 요청이 즉시 실패하고, 일정 시간이 지나면 Half-Open으로 전환된다. **Half-Open**은 테스트 상태다. 제한된 수의 요청을 통과시켜보고, 성공하면 Closed로, 실패하면 Open으로 돌아간다.

```
from enum import Enum
from datetime import datetime, timedelta
import threading

class CircuitState(Enum):
 CLOSED = "closed"
 OPEN = "open"
 HALF_OPEN = "half_open"

class CircuitBreaker:
 def __init__(
 self,
 failure_threshold: int = 5,
 recovery_timeout: float = 30.0,
 half_open_requests: int = 3
 ):
 self.state = CircuitState.CLOSED
 self.failure_count = 0
 self.failure_threshold = failure_threshold
 self.recovery_timeout = timedelta(seconds=recovery_timeout)
```

```
self.last_failure_time = None
self.half_open_success = 0
self.half_open_requests = half_open_requests
self._lock = threading.Lock()

def call(self, func, *args, **kwargs):
with self._lock:
if self.state == CircuitState.OPEN:
if datetime.now() - self.last_failure_time >
↪  self.recovery_timeout:
self.state = CircuitState.HALF_OPEN
self.half_open_success = 0
else:
raise CircuitOpenError("Circuit is OPEN")

if self.state == CircuitState.HALF_OPEN:
if self.half_open_success >= self.half_open_requests:
self.state = CircuitState.CLOSED
self.failure_count = 0
else:
pass # 허용된 수만큼 요청 통과

try:
result = func(*args, **kwargs)
self._on_success()
return result
except Exception as e:
self._on_failure()
raise

def _on_success(self):
with self._lock:
```

```
        if self.state == CircuitState.HALF_OPEN:
            self.half_open_success += 1
        else:
            self.failure_count = 0

    def _on_failure(self):
        with self._lock:
            self.failure_count += 1
            self.last_failure_time = datetime.now()
            if self.failure_count >= self.failure_threshold:
                self.state = CircuitState.OPEN

class CircuitOpenError(Exception):
    pass
```

핵심은 failure_threshold와 recovery_timeout을 실제 트래픽 패턴에 맞게 설정하는 것이다. LLM API의 경우 rate limit 에러가 많이 발생하므로, failureThreshold를 낮추는 것이 유리하다.

3.3 Bulkhead 패턴: 리소스 격리

Circuit breaker가 장애 파급을 차단한다면, bulkhead는 아예 장애 발생 확률을 줄인다. Bulkhead는 선박의 방수격벽에서 유래했다. 선박 한 칸에 구멍이 나도 다른 칸으로 물이 번지지 않도록 하는 구조다.

소프트웨어에서는 서로 다른 작업의 리소스를 분리하는 것으로, 한 작업이 리소스를 독점하지 못하게 막는다.

LLM API 연동에서는 두 가지 방식으로 bulkhead를 구현한다. 첫째, **연결 풀 분리**다. 중요도별로 다른 API 클라이언트를 사용하면, 중요도 높은 요청이 rate limit에 도달해도 낮은 요청에 영향이 가지 않는다.

```
from openai import OpenAI

class LLMClientPool:
 """우선순위별로 분리된 클라이언트 풀"""

 def __init__(self):
 self.priority_client = OpenAI(
 api_key=os.environ["OPENAI_API_KEY"],
 max_retries=0 # 재시도는 caller에서 관리
 )
 self.bulk_client = OpenAI(
 api_key=os.environ["OPENAI_API_KEY"],
 max_retries=0
 )
 self.low_priority_client = OpenAI(
 api_key=os.environ["OPENAI_API_KEY"],
 max_retries=0
 )

 async def call(self, prompt: str, priority: str = "normal"):
 client = {
 "high": self.priority_client,
 "normal": self.bulk_client,
 "low": self.low_priority_client
 }.get(priority, self.bulk_client)

 return await client.chat.completions.create(
 model="gpt-4o",
 messages=[{"role": "user", "content": prompt}]
 )
```

둘째, **Semaphore로 동시 요청 수 제한**이다. Python에서는 asyn-

cio.Semaphore로 동시 실행 수를 제어한다.

```
import asyncio

class PrioritySemaphore:
 def __init__(self):
 self.high = asyncio.Semaphore(10) # 우선순위 높음: 10개 동시
 self.normal = asyncio.Semaphore(5) # 보통: 5개 동시
 self.low = asyncio.Semaphore(2) # 낮음: 2개 동시

 async def acquire(self, priority: str):
 sem = getattr(self, priority, self.normal)
 return await sem.acquire()

 def release(self, priority: str):
 sem = getattr(self, priority, self.normal)
 sem.release()
```

이렇게 우선순위별로 동시성을 제한하면, 우선순위 높은 요청이 대량의 동시 요청에 가려지지 않는다.

3.4 Dead Letter Queue: 실패의 안전한 보관

재시도에도 실패한 요청은 어떻게 될까? 그냥 사라지면 안 된다. Dead Letter Queue(DLQ)는 실패한 메시지를 보관해서 나중에 처리할 수 있게 한다.

```
import json
import time
from pathlib import Path
from datetime import datetime
```

```
class DeadLetterQueue:
 def __init__(self, storage_path: str = "./dlq"):
 self.storage_path = Path(storage_path)
 self.storage_path.mkdir(parents=True, exist_ok=True)

 def store(self, message: dict, error: Exception, context: dict):
 """실패한 메시지를 DLQ에 저장"""
 entry = {
 "original_message": message,
 "error_type": type(error).__name__,
 "error_message": str(error),
 "context": context,
 "failed_at": datetime.now().isoformat(),
 "retry_count": context.get("retry_count", 0)
 }

 filename = f"{int(time.time() *
↪  1000)}-{context.get('request_id', 'unknown')}.json"
 filepath = self.storage_path / filename

 with open(filepath, "w") as f:
 json.dump(entry, f, ensure_ascii=False, indent=2)

 return filepath

 def load_pending(self, limit: int = 100):
 """DLQ에 보관된 실패 메시지 목록 반환"""
 files = sorted(self.storage_path.glob("*.json"))[:limit]
 results = []
 for f in files:
 with open(f) as fp:
 results.append({"path": str(f), **json.load(fp)})
```

```
        return results

    def retry(self, filepath: str) -> dict:
    """DLQ 메시지를 재처리"""
    with open(filepath) as f:
    entry = json.load(f)

    # 원래 로직 재시도
    result = process_message(entry["original_message"])

    # 성공 시 DLQ에서 제거
    Path(filepath).unlink(missing_ok=True)
    return result
```

DLQ 운영에서 중요한 것은 모니터링이다. DLQ에 메시지가 쌓이면 시스템에 문제가 있다는 신호다. 매일 DLQ를 확인하고, 반복해서 실패하는 메시지는 패턴을 분석해 근본 원인을 제거해야 한다.

3.5 마무리

지금까지 살펴본 네 가지 패턴은 각각 다른 실패 지점을 겨냥한다. 지수 백오프와 jitter는 재시도가 한꺼번에 몰리는 순간의 부하를 흩어놓고, circuit breaker는 계속 실패하는 요청이 다른 요청까지 끌어내리지 않도록 막는다. bulkhead는 중요도가 다른 작업의 리소스를 애초에 나눠두며, DLQ는 그래도 실패한 요청을 버리지 않고 남겨둔다. 네 가지를 함께 갖췄을 때 LLM API 연동은 일부 요청이 실패하더라도 전체가 흔들리지 않는 구조로 자리 잡는다.

4 비용 관리와 옵저버빌러티

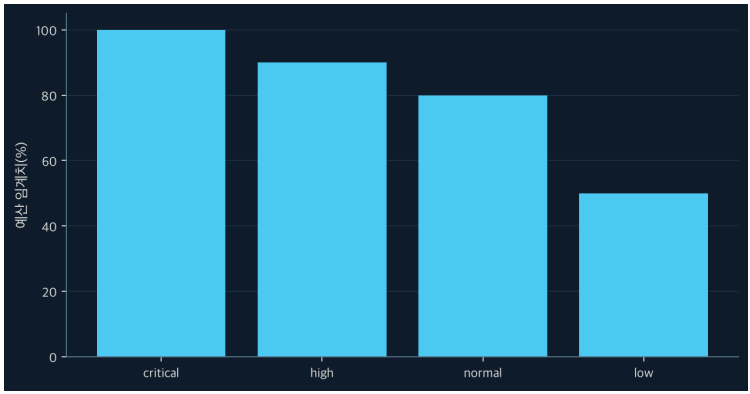

Figure 4.1: 비용 관리와 옵저버빌러티

LLM API 비용은 예측 불가능하게 쌓인다. 긴 대화 한 번, 의도하지 않은 반복 출력, 잘못된 토큰 계산 하나만으로도 청구서가 순식간에 불어난다. 이 장에서는 토큰 예산 시스템과 의미론적 캐싱, 분산 추적 연동을 다루고, 마지막으로 모델이 실패했을 때 사용자에게 어떻게 대응할지도 짚는다.

4.1 토큰 예산 시스템

LLM API의 비용은 입력 토큰과 출력 토큰으로 구성된다. 입력 토큰은 프롬프트와 대화 이력, 출력 토큰은 생성된 응답이다. 두 가지 모두 예산 관리의 대상이다.

입력 토큰 관리의 핵심은 대화 이력의 크기를 제한하는 것이다. 대화 이력이 끝없이 쌓이면 입력 토큰 수도 그만큼 늘어난다. 최대 메시지 수를 제한하는 것이 가장 간단한 전략이다.

```
from collections import deque
from dataclasses import dataclass
from typing import Optional

@dataclass
class ConversationBudget:
 max_tokens: int = 64000 # 입력 토큰 상한
 max_messages: int = 20 # 메시지 수 상한
 warning_threshold: float = 0.8 # 80% 이상에서 경고

class ManagedConversation:
 def __init__(self, budget: ConversationBudget):
 self.budget = budget
 self.messages = deque(maxlen=budget.max_messages)
 self.token_count = 0

 def add(self, role: str, content: str, tokens: int):
 self.messages.append({"role": role, "content": content})
 self.token_count += tokens

 if self.token_count > self.budget.max_tokens *
 ↪  self.budget.warning_threshold:
 self._trigger_warning()

 def _trigger_warning(self):
 # 토큰 사용량이 상한의 80%를 넘으면 오래된 메시지 제거
 while (self.token_count > self.budget.max_tokens * 0.6
 and len(self.messages) > 4):
 removed = self.messages.popleft()
 self.token_count -= estimate_tokens(removed["content"])

 def get_messages(self):
```

```
 return list(self.messages)
```

이 코드의 핵심은 프롬프트 압축이 아니라 대화 이력의 간결한 관리다. LLM에게 보내는 대화 이력이 줄어들면 입력 토큰이 줄어든다.

출력 토큰 관리는 max_tokens 파라미터로 제어한다. 예상 출력보다 크게 잡으면 비용 낭비가, 작게 잡으면 출력이 잘린다. 이를 해결하려면 출력 길이를 예측해야 한다.

```
async def call_with_adaptive_max_tokens(
 prompt: str,
 model: str = "gpt-4o",
 min_output: int = 100,
 max_output: int = 4000
) -> str:
 """초기 추정에 실패하면 출력 길이를 조정해서 재시도"""
 for attempt in range(3):
 response = await client.chat.completions.create(
 model=model,
 messages=[{"role": "user", "content": prompt}],
 max_tokens=max_output,
 stop=None if attempt == 0 else ["
---
"]
 )

 content = response.choices[0].message.content

 # 출력이 잘렸는지 확인
 if response.choices[0].finish_reason == "length":
 # 토큰이 부족했던 것이므로 길이를 늘려 재시도
 max_output = min(max_output * 2, 16000)
```

```
continue

# 출력이 너무 짧은 경우
token_estimate = estimate_tokens(content)
if token_estimate < min_output:
# 더 긴 출력을 요청 (간결 모드 해제 등)
max_output = min(max_output * 2, 16000)
continue

return content

return content # 마지막 시도 결과 반환
```

예산 고갈 대응도 설계해야 한다. 월간 예산에 도달하거나 일일 할당량을 넘으면 어떻게 할 것인가. 단순히 요청을 거절하면 서비스가 중단된다. Tiered 접근법이 효과적이다.

```
class TieredBudgetManager:
 def __init__(self):
 self.tiers = {
 "critical": BudgetTier(limit=1.0, # 100% 허용
 action="proceed"),
 "high": BudgetTier(limit=0.9, # 90% 이상 시 경고
 action="warn"),
 "normal": BudgetTier(limit=0.8, # 80% 이상 시 우회
 action="fallback"),
 "low": BudgetTier(limit=0.5, # 50% 이상 시 차단
 action="reject")
 }
 self.current_usage = 0.0

 def check(self, priority: str, estimated_cost: float) ->
 ↪ Decision:
```

```
tier = self.tiers.get(priority, self.tiers["normal"])
projected = self.current_usage + estimated_cost

if projected > tier.limit:
return Decision(action=tier.action, proceed=False)

self.current_usage = projected
return Decision(action="proceed", proceed=True)
```

4.2 의미론적 캐싱: Embedding 기반 요청 Deduplication

동일한 질문에 동일한 응답을 여러 번 생성하면 비용이 불필요하게 발생한다. 전통적 캐싱은 URL이나 쿼리 문자열 같은 정확한 키를 사용하지만, LLM에서는 프롬프트가 완전히 동일하지 않아도 의미가 같은 경우가 많다.

Embedding 기반 캐싱은 요청의 의미적 유사도를 계산해 캐시 히트를 판단한다.

```
import numpy as np
from openai import OpenAI

class SemanticCache:
 def __init__(
 self,
 embedding_model: str = "text-embedding-3-small",
 similarity_threshold: float = 0.95,
 max_cache_size: int = 10000
 ):
 self.client = OpenAI()
```

```
self.embedding_model = embedding_model
self.similarity_threshold = similarity_threshold
self.cache = [] # (embedding, response, count)
self.cache_count = {}

async def get_or_compute(self, prompt: str, compute_fn):
"""캐시에서 유사 요청을 찾거나 새로 계산"""
# Embedding 생성
embedding = await self._get_embedding(prompt)

# 캐시에서 유사한 요청 탐색
for idx, (cached_emb, response, _) in enumerate(self.cache):
similarity = self._cosine_similarity(embedding, cached_emb)

if similarity >= self.similarity_threshold:
# 캐시 히트
self.cache_count[idx] = self.cache_count.get(idx, 0) + 1
return {"hit": True, "response": response, "similarity":
↪  similarity}

# 캐시 미스 - 새로 계산
response = await compute_fn(prompt)

# 캐시에 추가
if len(self.cache) >= 1000: # 캐시 크기 제한
self._evict_least_used()

self.cache.append((embedding, response, 0))
return {"hit": False, "response": response, "similarity": 1.0}

async def _get_embedding(self, text: str) -> list[float]:
result = self.client.embeddings.create(
```

```
model=self.embedding_model,
input=text
)
return result.data[0].embedding

@staticmethod
def _cosine_similarity(a: list[float], b: list[float]) -> float:
a = np.array(a)
b = np.array(b)
return float(np.dot(a, b) / (np.linalg.norm(a) *
↪  np.linalg.norm(b)))

def _evict_least_used(self):
"""사용 횟수가 가장 적은 캐시 항목 제거"""
if not self.cache_count:
self.cache.pop(0)
return

min_idx = min(self.cache_count, key=self.cache_count.get)
del self.cache[min_idx]
del self.cache_count[min_idx]
# 인덱스 재정렬
self.cache_count = {
k - 1 if k > min_idx else k: v
for k, v in self.cache_count.items()
}
```

이 캐시 전략에서 가장 중요한 값은 `similarity_threshold`다. 0.95는 매우 높은 유사도를 요구해서 히트가 자주 나지 않는다. 0.85로 낮추면 히트율은 오르지만 잘못된 응답을 반환할 가능성도 함께 커진다. 실제 트래픽 패턴을 보면서 값을 조정해야 한다.

4.3 분산 추적: LangSmith, LangFuse, OTel

LLM 기반 시스템이 복잡해지면 하나의 요청이 여러 모델 호출, 캐시 조회, 벡터 저장소 질의 등을 거친다. 이 과정에서 어디서 병목이 발생하는지 파악하려면 분산 추적이 필요하다.

LangSmith는 LangChain이 운영하는 매니지드형 추적 서비스다. 설정이 간단하고, OpenAI SDK를 비롯한 여러 LLM SDK와 쉽게 통합된다. 그러나 외부 서비스 의존이 싫으면 다른 옵션이 필요하다.

```
from langsmith import traceable
from openai import OpenAI

client = OpenAI()

@traceable(name="llm-summarize")
async def summarize(text: str, max_length: int = 100) -> str:
 response = client.chat.completions.create(
 model="gpt-4o-mini",
 messages=[
 {"role": "system", "content": f"{max_length}자 이하로 요약"},
 {"role": "user", "content": text}
 ]
 )
 return response.choices[0].message.content
```

LangFuse는 자체 호스팅이 가능한 오픈소스 옵션이다. PostgreSQL과 Redis만 있으면 운영할 수 있고, 모든 추적 데이터를 직접 관리한다.

```
from langfuse import Langfuse

langfuse = Langfuse(
 public_key=os.environ["LANGFUSE_PUBLIC_KEY"],
```

```
 secret_key=os.environ["LANGFUSE_SECRET_KEY"],
 host="https://langfuse.your-company.com"
)

@langfuse.observe(name="llm-summarize")
async def summarize(text: str) -> str:
 # 추적이 자동으로 기록됨
 response = client.chat.completions.create(
 model="gpt-4o-mini",
 messages=[{"role": "user", "content": text}]
 )
 return response.choices[0].message.content
```

OpenTelemetry(OTel)는 업계 표준 추적 프레임워크다. LangFuse, LangSmith, Weights & Biases 등 대부분의 추적 도구가 OTel 스팬을 지원하므로, OTel 기반으로 통합하면 추적 백엔드를 자유롭게 교체할 수 있다.

```
from opentelemetry import trace
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor
from opentelemetry.exporter.otlp.proto.grpc.trace_exporter
↪  import OTLPSpanExporter

trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer(__name__)

# OTel 스팬으로 LLM 호출 래핑
async def traced_llm_call(prompt: str, model: str = "gpt-4o") ->
↪  str:
 with tracer.start_as_current_span("llm_call") as span:
 span.set_attribute("llm.model", model)
```

```
    span.set_attribute("llm.prompt_length", len(prompt))

    start = time.perf_counter()
    response = client.chat.completions.create(
    model=model,
    messages=[{"role": "user", "content": prompt}]
    )
    duration = time.perf_counter() - start

    span.set_attribute("llm.tokens_used",
    ↪  response.usage.total_tokens)
    span.set_attribute("llm.duration_ms", duration * 1000)

    return response.choices[0].message.content
```

핵심 속성(Attribute)을 추적해두면 나중에 분석하기 쉬워진다. 모델 이름, 입력 토큰 수, 출력 토큰 수, TTFT, 전체 소요 시간은 반드시 기록해야 하고, 오류가 발생했다면 오류 유형과 메시지도 함께 남겨야 한다.

4.4 Graceful Degradation: 모델 실패를 사용자 경험으로 전환

재시도에도 실패하고 circuit breaker까지 열렸다면 어떻게 할 것인가. 사용자에게 "일시적 오류"라는 말만 하면 신뢰가 떨어진다. Graceful degradation은 모델 실패를 의미 있는 사용자 경험으로 전환하는 설계다.

순차적 대체 전략이 가장 기본이다. 주력 모델이 실패하면 단순하지만 확실한 대안을 제시한다.

```
async def robust_complete(prompt: str) -> str:
    """여러 모델을 순차적으로 시도"""
```

```
models = ["gpt-4o", "gpt-4o-mini", "gpt-3.5-turbo"]

errors = []

for model in models:
try:
response = client.chat.completions.create(
model=model,
messages=[{"role": "user", "content": prompt}]
)
return response.choices[0].message.content

except RateLimitError:
# Rate limit은 잠시 후 재시도
await asyncio.sleep(5)
continue

except APIError as e:
errors.append({"model": model, "error": str(e)})
continue

# 모든 모델 실패 시 최종 대체 응답
return fallback_response(prompt, errors)
```

기능 축소는 모델 실패가 전체 기능 실패로 이어지지 않게 하는 방법이다. AI 요약이 실패하면 기존 키워드 기반 요약을 표시하고, AI 번역이 실패하면 기본 번역을 보여주는 식이다.

```
async def get_summary(text: str, use_ai: bool = True) -> str:
 if not use_ai:
 return naive_keyword_summary(text)
```

```
 try:
 return await ai_summary(text)
 except LLMError:
 # AI 실패 시 키워드 기반 요약으로 축소
 logger.warning("AI summary failed, falling back to keyword
↪  summary")
 return naive_keyword_summary(text)
```

사용자 경험으로서의 실패는 가장 중요한 설계 원칙이다. 실패 메시지는 "일시적 오류가 발생했습니다"가 아니라, 구체적으로 무엇이 안 되었고 무엇을 시도할 수 있는지 알려줘야 한다.

```
@dataclass
class GracefulDegradationResponse:
 success: bool
 primary_content: Optional[str]
 fallback_content: Optional[str]
 message: str # 사용자에게 보여줄 메시지
 retry_after: Optional[int] # 권장 재시도 시간(초)

def fallback_response(errors: list) ->
↪  GracefulDegradationResponse:
 if is_rate_limit_error(errors):
 return GracefulDegradationResponse(
 success=False,
 primary_content=None,
 fallback_content=None,
 message="현재 요청이 많아 응답이 지연되고 있습니다. 잠시 후 다시
↪  시도해 주세요.",
 retry_after=30
 )
 elif is_timeout_error(errors):
```

```
return GracefulDegradationResponse(
success=False,
primary_content=None,
fallback_content=None,
message="요청이 너무 복잡합니다. 더 짧게 입력해 주시면 처리할 수
↪ 있습니다.",
retry_after=None
)
else:
return GracefulDegradationResponse(
success=False,
primary_content=None,
fallback_content=None,
message="일시적인 오류가 발생했습니다. 잠시 후 다시 시도해
↪ 주세요.",
retry_after=5
)
```

실패를 숨기지 않는 것이 가장 중요하다. 실패를 숨기면 사용자만 모를 뿐 문제 자체는 그대로 남는다. 현재 상태를 투명하게 전하고 다음 행동을 제시해야 신뢰를 유지할 수 있다.

4.5 마무리

비용 관리와 옵저버빌러티는 LLM API 연동에서 반드시 갖추어야 할 시스템 품질의 두 축이다. 토큰 예산 시스템은 비용의 예측 가능성을 높이고, 의미론적 캐싱은 불필요한 API 호출을 줄인다. 분산 추적은 병목과 비용 원인을 짚어내고, graceful degradation은 모델 실패가 사용자 경험의 실패로 번지지 않게 막아준다. 이 네 가지가 함께 갖춰져야 비로소 LLM을 프로덕션 시스템의 신뢰할 수 있는 구성 요소로 쓸 수 있다.