무너지지 않는 기술: 레이트 리미팅, 백프레셔, 그리고 과부하를 막는 규율

트래픽 폭주와 LLM API 비용 폭증이 올 때 일부러 늦추고 거절하는 1인 개발자의 실무 가이드

ThakiCloud (Hyojung Han)

무너지지 않는 기술: 레이트 리미팅, 백프레셔, 그리고 과부하를 막는 규율

트래픽 폭주와 LLM API 비용 폭증이 올 때 일부러 늦추고 거절하는 1인 개발자의 실무 가이드

ThakiCloud (Hyojung Han)

Contents

1 의도적으로 늦추는 기술: 과부하의 규율 1

2 서버 측면: 토큰 버킷과 429 8

3 클라이언트 측면: 백오프와 로드 셰딩 16

4 LLM API: 비용과 트래픽을 혼자 막기 24

1 의도적으로 늦추는 기술: 과부하의 규율

토요일 밤 11시. 부업으로 만들고 배포한 서비스가 화제가 되었다. 한 시간 만에 트래픽이 열 배가 됐다. 로그에 처음 올라오는 건 타임아웃 에러. 그다음엔 워커 프로세스가 전부 대기 중인 요청에 붙잡힌다. 데이터베이스 연결 풀이 비어간다. 마지막엔 헬스체크조차 답이 없다. 대시보드를 새로고르면 에러율이 100퍼센트다.

죽은 원인이 트래픽이 빨리 들어온 것은 아니다. 시스템 안 어디에서도 "지금은 안 돼, 조금 후에 와"라고 말한 존재가 없었던 것이다. 이 책은 그 말을 만드는 법에 관한 책이다. 제목의 세 단어, 레이트 리미팅, 백프레셔, 과부하 규율이 각자 담당하는 구역을 먼저 짚자. 레이트 리미팅은 문 앞에서 받는 사람 수를 정하는 일이다. 백프레셔는 뒤에서 오는 압력을 앞으로 신호로 돌려보내는 일이며, 규율은 사고가 나기 전에 이 둘을 적용하는 태도다.

1.1 과부하는 오지 않고 자란다

과부하를 "서버가 감당할 트래픽보다 많은 것"으로만 떠올리기 쉽다. 그런데 실제 전개는 좀 다르다.

1. 용량보다 조금 많은 요청이 들어온다.
2. 일부 요청이 대기하기 시작한다. 처리 시간이 늘어난다.
3. 처리가 늘어난 만큼 동시에 쌓이는 요청이 더 많아진다.
4. 클라이언트와 로드 밸런서에서 답이 없자 타임아웃이 발생한다.
5. 타임아웃은 재시도로 바뀐다. 트래픽이 다시 늘어난다.
6. 3번으로 돌아간다.

치명적인 지점은 5번이다. 한 번의 실패가 재시도를 통해 두 번째 트래픽

물결로 돌아온다. 시스템은 원래 부하보다 자기가 부른 부하에 죽는 경우가 대부분이다. 이 순환을 캐스케이드 실패라고 부르는데, 상한이 없는 서비스는 규모나 스택과 무관하게 모두 여기서 끝난다.

구체적으로 하나 세워보자. 10개의 워커가 있고 요청 하나를 처리하는 데 평소 100밀리초가 걸린다고 하자. 초당 100개는 여유다. 그런데 초당 150개가 오면, 50개가 매 턴 대기 줄에 쌓인다. 대기 줄이 자라면서 요청 하나가 시스템을 관통하는 시간이 300밀리초, 800밀리초로 늘고 클라이언트가 1초에 타임아웃을 걸면 재시도 물결이 그 위에 겹친다. 처음에는 "조금 느리네"였던 상태가 몇 분 안에 "전부 죽었다"가 되는 데 필요한 건 더 많은 트래픽이 아니라, 이 순환이 한 바퀴만 도는 시간이다.

한도까지 두고 똑같이 돌려보자. 워커 열 개면 초당 100개를 처리할 수 있고 한도를 그 80퍼센트인 초당 80개로 놓았다고 하자. 초당 150개가 오면 80개는 들어오고 70개는 즉시 429로 돌아간다. 워커는 평소 속도로 일하고 대기 줄은 거의 제로에 머물러 있다. 429를 받은 사용자는 2, 3초를 기다린 뒤 다시 오는데, 그때쯤 자리가 난다. 시스템은 "자라는" 구간을 한 번도 통과하지 못한다. 대기의 순간이 없으니 타임아웃의 순간이 없고 재시도 폭풍의 순간도 없다. 거절된 70개는 손실이다. 다만 보이는 손실이다. 무너짐의 손실은 보이지 않는다. 온 150개가 다 실패하고 사용자는 떠나고, 청구서까지 남는다. 한도의 가치는 전부 처리하는 데 있지 않다. 보이지 않는 큰 손실을, 작고 회복 가능한 손실로 바꾸는 데 있다.

그래서 첫 번째 원칙이 나온다. 거절하지 못하는 시스템은 죽고 일찍 거절하는 시스템은 산다.

1.2 거절은 설계 결정이다

"거절"은 스스로 일어나지 않는다. 무슨 에러를 선호할지는 아무 일도 일어나기 전에 정해야 한다.

- 429 Too Many Requests. “나는 살아 있는데 너는 너무 많다. 잠시 후에 와”라는 말.
- 5xx Server Error. “나는 무너지고 있으니 지금 당장 멈춰”라는 말.
- 타임아웃. 답이 없는 상태. 해석은 호출 쪽이 알아서 한다.

건강한 시스템은 과부하 시 429를 고른다. 429는 용량 문제라는 정보를 전달한다. 클라이언트는 백오프해서 나중에 다시 오면 되고 서버는 처리할 수 없는 요청에 자원을 태우지 않아도 된다. 반대로 과부하에서 나온 5xx는 시스템이 무너지는 중이라고 스스로 고백하는 것이다. 더 나쁜 건 많은 클라이언트가 5xx를 보면 재시도한다는 점이다. 무너지고 재시도당하고, 그 재시도가 또 무너짐을 키운다.

그러니까 레이트 리미팅은 배포 뒤에 붙이는 부가 기능이 아니다. 에러 처리 명세서의 한 줄이다. “요청이 너무 많으면 429와 이 바디를 돌려준다”는 문구가 설계 문서에 있어야 한다. 이 문구가 없으면, 폭주가 온 날 첫 결정이 검토가 아니라 당황이 된다.

현장에서 듣는 변명을 세 가지 모아두자. “아직 트래픽이 없어요.” 트래픽이 없는 지금은 숫자를 정하기 가장 싼 시점이고 바쁜 때는 가장 비싼 시점이다. “스케일업하면 되겠죠.” 스케일업은 시간이 걸리고 돈이 들고, 그 사이에 시스템은 이미 죽어 있다. “게이트웨이가 해줄 거예요.” 게이트웨이가 전역 제한을 해도, 내가 부르는 LLM과 데이터베이스까지 지켜주지는 못한다. 세 변명 모두 맞을 수 있다. 다만 셋 다 “이번 주에 내가 숫자를 정하는 일”을 대신하지는 못한다.

1.3 가장 원시적인 거절: 타임아웃

레이트 리미팅보다 오래된 메커니즘이 하나 있다. 타임아웃이다. 타임아웃은 호출자가 한도를 정하는 거절이다. “나는 2초까지 기다릴 수 있어”라는 말. 그때까지 답이 없으면 그 요청은 죽는다.

타임아웃은 이 책의 모든 메커니즘의 바닥이다. 타임아웃이 없으면 재시도도 일어나지 않기 때문이다. 클라이언트가 영원히 기다리면, 캐스케이드 실패의 4단계는 도래하지 않는다. 반대로 타임아웃이 위험한 이유는, 대기를 트래픽으로 바꿔버린다는 점이다. 그래서 규칙은 두 개인다. 모든 호출에 타임아웃을 걸고 무한대는 금지. 그리고 타임아웃은 사용자가 기다릴 수 있는 값보다 짧아야 한다. 사용자가 보는 화면이 5초까지 기다린다면, 하류 호출이 10초를 기다리는 것은 의미가 없다. 워커를 태워서 아무것도 만들지 않는 일이다.

이 책에서 한도(얼마나 받을지)와 타임아웃(얼마나 기다릴지)는 쌍으로 나온다. 같은 코인의 양 면이다.

1.4 세 층의 방어

방어선은 한 줄로는 버티지 못한다. 실무에서는 세 층을 만든다.

층	답하는 질문	위치
진입	초당 얼마를 들여보낼까	게이트웨이 또는 앱 시작점
인플라이트	동시에 몇 개를 처리할까	워커 풀, 연결 풀
의존성	외부 호출에 하루 얼마를 쓸까	LLM, 결제, 문자 래퍼

각 층이 지키는 자원이 다르다. 진입 층은 CPU와 메모리를 지키고 인플라이트 층은 연결 풀과 큐를 지키며 의존성 층은 돈과 외부 서비스와의 관계를 지킨다. 진입 제한만 만들면, 소수의 큰 사용자가 LLM 예산을 전부 태우는 동안 박스 자체는 멀쩡해 보인다. 예산만 만들면, 트래픽 물결이 먼저 데이터베이스를 쓰러뜨린다.

백프레셔라는 말이 여기서 의미를 갖는다. 압력이 뒤에서 오면, 뒤로 전파되는 신호를 만들어야 한다. 큐가 80퍼센트 차면 새 요청을 받지 않는 것, 외부 API가 느려지면 내 응답도 일부러 느리게 주는 것, 워커가

다 차면 429를 보내는 것. 이것들이 모두 백프레셔다. 압력을 무시하고 "빨리빨리"만 외치면, 압력은 가장 약한 곳, 대개는 데이터베이스나 결제 게이트웨이에서 터진다.

1인 운영자에게는 세 층을 모두 자기 손으로 다룰 수 있다는 점이 있다. "그건 다른 팀 영역이다"라는 벽이 없다. 진입, 인플라이트, 의존성. 숫자 세 개를 하나씩 정하는 것만으로도, 서비스는 어느 한 방향으로만 무너지지 않게 된다.

1.5 출시 전에 숫자를 정한다

실패 후에 세운 한도는 벌금이고 전에 세운 한도는 정책이다. 정확할 필요는 없다. 정할 필요만 있다. 거친 숫자는 얼마든 고칠 수 있지만 숫자 자체는 고칠 수 없다.

출시 전에 아래 여덟 질문에 답하고 그 답을 한곳에 적어둔다.

1. 요청을 최장 얼마나 기다리게 할 수 있는가. 큐 대기 상한이다.
2. 동시에 처리할 수 있는 요청의 최대 개수는 몇 개인가.
3. 어떤 요청을 먼저 떨어뜨려도 되는가. 분석 로그, 캐시 갱신, 비유형 작업 같은 것들.
4. 외부 호출 하나당 비용이 얼마인가. 하루 상한은 얼마인가.
5. 한도에 닿으면 어떤 에러를 선호하는가. 429인가, 5xx인가.
6. 내 API는 멱등한가. 클라이언트가 안전하게 재시도할 수 있는가.
7. 한도를 어떻게 알릴 것인가. Retry-After 헤더, 잔여 한도 헤더.
8. 한도에 닿는 순간 누가 가장 먼저 알 것인가.

여덟 질문에 한 시간 안에 답할 수 있다면, 그 서비스는 현장에서 보는 대부분의 서비스보다 과부하에 강하다. 8번의 답은 거의 항상 "나"인데, 1인 운영자는 한도가 트립되는 순간 알림 채널을 남겨두어야 한다. 문자 한 통이면 된다. 한도가 트립했다는 것은 내가 아는 세계가 이미 바꼈다는

신호이기 때문에, 그 사실을 그 시점에 몰라서 되는 일은 하나도 없다. 알림의 대상도 잊지 말자. "429가 첫 번째로 나왔을 때"가 적다. 429가 나오는 건 시스템이 방어하고 있다는 증거고, 그걸 고요하게 넘기면 한도가 무너지는 순간을 처음 알게 되는 채널이 프로바이더의 정지 메일이 된다.

1.6 거절의 경제학

세 가지 결과를 비교해보자.

결과	사용자 비용	내 비용
일찍 거절 (429)	잠깐 대기	평판 조금
버티기	느려짐	자원 소진
무너짐 (5xx)	실패, 이탈	정지+재시도 폭풍

일찍 거절하면 사용자가 잠깐 대기한다. 솔직한 비용이다. 그리고 클라이언트는 나중에 다시 오면 된다는 걸 안다. 한도를 넘어서 버티는 것이 나쁜 선택인 이유는, CPU와 연결과 예산을 태우는 동안 사용자 경험은 한 초도 빠지지 않기 때문이다. 무너짐은 전부 정지이고 재시도 폭풍까지 부추긴다. 이 책의 규율은 첫 번째를 고르는 일이다. 넘으면 거절한다. 빨리 거절하고 분명히 거절한다.

한 가지만 덧붙이자. "분명히"가 429를 쓸모 있게 만든다. 클라이언트는 얼마를 기다려야 하는지(Retry-After)와 얼마나 넘었는지(잔여 한도 헤더)를 알아야 백오프를 제대로 할 수 있다. 아무 정보도 없는 429는 장애물이다.

숫자 감각을 위해 하나 더. 429를 하루에 100건 보낸 서비스가 있다고 하자. 각 사용자가 30초를 기다리고 다시 와서 성공하면, 사용자는 "조금 밀린다"고 느낀다. 반대로 그 100건을 받아들이려다 5분간 전체 응답이 500으로 변하는 경우, 사용자는 "서비스가 죽었다"고 느낀다. 같은

100건의 실패인데, 체감은 하늘과 땅이다. 한도 설계의 목표는 실패의 체감을 "전체 정지"에서 "약간의 대기"로 옮기는 일이다.

1.7 첫 429는 축하일이다

서비스 초기, 로그에 429가 처음 찍힐 때의 기분은 버그를 발견한 것과 비슷하다. "내 한도가 사용자를 막았네" 하는 불편함이 든다. 그런데 이 사건은 축하할 사건이다. 한도가 실제로 작동한다는 증거이기 때문이다.

한 번도 트립하지 않는 한도는 너무 높게 놓았거나, 처음부터 끼워지지 않은 것이다. 429가 반년 동안 한 번도 안 나온다면, 그것이 동작한다는 증거가 없다. 처음 429가 나올 때 확인할 것은 네 가지다. 하나, 응답에 Retry-After가 있는지. 둘, 바디에 사용자가 읽을 수 있는 메시지가 있는지. 셋, 여덟 번째 질문에 두고 온 알림이 왔는지. 넷, 통과한 요청들의 에러율이 여전히 평평한지. 네 개 모두 예라면, 한도는 이제 시스템의 일부다. 하나라도 아니라면, 그날 바로 고친다. 현장이 아직 선명할 때.

시각도 적어둔다. "10월 어느 화요일, 마케팅 메일 나간 시간에 한도가 트립했다"는 게 데이터다. 다음에 rate를 바꿀 때, 그 기억이 여유가 얼마나 있었는지 알려준다. 1인 운영자의 데이터 과학은 이런 일기다.

1.8 이번 장 연습

지금, 내가 돌리는 서비스에 숫자 세 개를 적어보자.

- 진입. 사용자당 초당 X개, 전체 분당 Y개.
- 인플라이트. 동시에 Z개까지.
- 의존성. 외부 호출에 하루 W달러, 또는 W원까지.

추정이어도 좋다. 적는 순간부터 그것은 정책이 된다. 다음 장에서는 X와 Y라는 숫자를 거짓말하지 않는 형태로 만드는 도구, 즉 토큰 버킷을 다룬다.

2 서버 측면: 토큰 버킷과 429

한도는 “지금은 되고 지금은 안 된다”를 정하는 알고리즘만큼만 정확하다. 이 장에서는 고전적인 방식 세 가지, 고정 윈도우, 슬라이딩 윈도우, 토큰 버킷을 차례로 훑고 실무의 일꾼인 토큰 버킷을 구현한다. 마지막에는 429를 보내는 정석이다.

2.1 고정 윈도우: 단순하지만 경계에서 거짓말한다

가장 직관적이다. 1분 단위로 요청을 세고 100개를 넘으면 429를 보낸다. 매 분의 시작에서 카운터를 제로로 초기화한다.

문제는 경계에서 나온다. 성실한 클라이언트가 N분 마지막 1초에 100개를 보내면, N+1분 첫 1초에 또 100개를 보낼 수 있다. 2초 사이에 200개. 분당 평균으로 보면 한도인데, 순간적 피크로는 한도의 두 배가 통과한다. 그 순간 피크를 데이터베이스가 못 버틴다면, 경계 문제는 주말 밤 11시다.

고정 윈도우의 가치는 단순함에 있다. 카운터 하나면 되고 저트래픽 서비스에 거친 한도면 충분하다. 그러나 버스트가 의미 있는 순간에는 다음을 보아야 한다.

2.2 슬라이딩 윈도우: 정확하지만 메모리가 늘어난다

경계 문제를 고치려면 “지금부터 거꾸로 60초”를 본다. 모든 요청에 시각을 붙이고 지금 끝나는 윈도우 안에 몇 개가 있는지 센다.

결과는 정확하다. 2초 경계에서 200개는 어디에서도 발생할 수 없다. 그러나 대가가 있다. 최근 요청들의 시각을 윈도우에서 벗어날 때까지 보관해야 한다. 트래픽이 많은 엔드포인트에서는 사용자당 수천 개의

엔트리가 쌓이고, 한도 자체가 자원을 먹는 존재가 된다. 시각을 압축하는 방법도 있다. 비트맵이나 근사 카운터처럼. 하지만 그것들은 복잡도를 올린다. 1인 개발자의 결론은 이렇다. 슬라이딩 윈도우는 토큰 버킷으로는 부족해져서야 갈 때의 도구다. 첫 선택지는 아니다.

2.3 토큰 버킷: 숫자 두 개, 거짓말 하나

토큰 버킷은 프로덕션 서비스 대부분이 쓰는 일꾼이다. 숫자 두 개로 돌아간다.

- rate. 초당 버킷에 토큰이 몇 개 채워지는가. 곧 지속할 수 있는 속도다.
- capacity. 버킷에 몇 개까지 들어 있는가. 곧 버스트의 천장이다.

버킷은 capacity만큼 채워져서 시작한다. 요청이 오면 토큰 하나를 빼고 통과시킨다. 토큰이 없으면 429. 토큰은 rate의 속도로 채워지지만 capacity를 넘어서는 순간 멈춘다.

실무에서 살리는 두 성질이 있다. 첫째, 평균 속도가 rate로 꽁꽁 박혀 있다. 채워지는 속도가 rate이기 때문에, 요청을 어떻게 배치해도 장기 평균은 rate를 넘지 못한다. 둘째, 버스트가 내 선택이다. 클라이언트가 요청을 모아 한꺼번에 쏘아도, capacity만큼만 통과한다. capacity가 큰 날에는 "허락한 폭주"가 되고, capacity가 작은 날에는 "허락하지 않은 폭주"가 된다.

capacity를 어떻게 정하나. 실용 규칙은 rate를 허용 버스트 지속 시간에 곱한 것이다. 2초짜리 버스트를 허용하고 싶으면 capacity = rate x 2. 버스트를 전부 죽이고 싶으면 capacity = rate. 이 때 버킷은 매 초 한 개씩만 내어주는 형태, 즉 리키 버킷처럼 굴는다. capacity를 크게 하는 것은 공짜가 아니다. 거절하고 싶었던 과부하를 그대로 들여보내는 일이다.

2.4 손으로 계산을 한번

숫자를 정해서 따라가보자. rate 10, capacity 20. t=0에 버킷이 가득(20) 있고 클라이언트가 동시에 25개를 쏘았다.

시간	버킷	판정
t=0	20에서 0	앞 20개 통과
t=0	0	뒤 5개는 429
t=0.5	5	5개 통과 가능
t=1.0	10	다시 10개 여력

동시에 온 25개는 "20개는 지금, 5개는 잠시"로 갈린다. 0.5초 뒤면 토큰이 5개 생기고 429를 받은 쪽이 그 때 재시도하면 통과한다. 밖에서 보면 "다 처리됐는데 조금 늦었다"이고 안에서 보면 "워커는 초당 10개를 한 번도 넘지 않았다"다.

핵심은 그 5개가 실제로 기다린다는 것이다. Retry-After: 0.5를 받은 클라이언트가 0.5초에 지터를 약간 더 얹어서 자고 재시도해야 한다. 바로 재시도하면 429다. 앞에서 세운 헤더 규칙이 여기서 첫 결실을 맺는다.

2.5 구현: 인프로세스 버전부터

첫 버전에는 Redis가 필요 없다. 동작이 옳은 것이 배포보다 우선이다.

```
import time

class TokenBucket:
 def __init__(self, rate: float, capacity: float):
 self.rate = rate
 self.capacity = capacity
```

```
self.tokens = capacity
self.updated = time.monotonic()

def take(self) -> bool:
now = time.monotonic()
self.tokens = min(self.capacity, self.tokens + (now -
↪  self.updated) * self.rate)
self.updated = now
if self.tokens >= 1:
self.tokens -= 1
return True
return False
```

사용자마다, 또는 API 키마다 버킷 하나를 둔다. take()가 True면 통과, False면 429. 프로세스 하나, 스레드 하나면 그대로 쓰고 멀티스레드 런타임이면 락으로 지키면 된다. 핵심은 두 줄이다. 먼저 지난 시간만큼을 채우고 capacity를 넘으면 자르고 그 다음에 하나를 빼는 것. 순서가 바뀌면 한도가 새거나 과하게 막힌다.

인스턴스가 여러 개가 되면 공유 상태를 Redis로 옮긴다. 패턴은 같다. 토큰 수와 마지막 채움 시각을 읽고 채움분을 더하고 하나를 빼고 다시 쓴다. 다만 이 네 가지를 하나의 원자적 단위, 즉 Lua 스크립트 하나로 실행해야 한다. 읽기와 쓰기를 나누면, 두 요청이 동시에 "토큰이 있겠다"고 판단해서 둘 다 통과한다. 한도가 새는 사고는 대부분 읽기와 쓰기를 나누는 구현에서 난다. 키는 사용자 ID의 해시로 만든다.

```
local key = KEYS[1]
local rate, capacity, now = tonumber(ARGV[1]), tonumber(ARGV[2]),
↪  tonumber(ARGV[3])
local data = redis.call('HMGET', key, 'tokens', 'updated')
local tokens, updated = tonumber(data[1]), tonumber(data[2])
```

```
if tokens == nil then tokens, updated = capacity, now end
tokens = math.min(capacity, tokens + (now - updated) * rate)
local allowed = 0
if tokens >= 1 then tokens = tokens - 1; allowed = 1 end
redis.call('HMSET', key, 'tokens', tokens, 'updated', now)
redis.call('PEXPIRE', key, math.ceil(capacity / rate) * 1000 +
↪  1000)
return allowed
```

스크립트는 Redis 안에서 원자적으로 실행된다. 앞에서 읽기, 계산, 쓰기를 나누면 안 된다고 한 이유가 바로 이것이다. 두 요청이 동시에 같은 토큰을 "보았을" 수는 있어도, 둘 다 "뺀" 적은 없다. 마지막 줄의 PEXPIRE도 빼먹지 말자. 버킷 키에 수명을 걸어두면, 한 번도 활동하지 않는 사용자의 키가 Redis에 쌓이지 않는다. 수명은 capacity를 다 채우는 시간, 즉 capacity 나 rate에 약간의 여유를 더한 정도면 된다.

2.6 429를 보내는 정석

429의 바디보다 헤더가 먼저다. 정석대로 백오프하는 클라이언트는 정보 세 개가 필요하다.

- Retry-After. 몇 초 후면 다시 와도 되는지.
- X-RateLimit-Limit. 이 윈도우에 허용한 총량.
- X-RateLimit-Remaining. 지금 남은 양.

이 헤더 이름들은 많은 서비스가 공유하는 관행이다. 내 서비스도 같은 이름을 쓰면, 클라이언트가 이미 만들어둔 처리 코드를 그대로 쓸 수 있다. 관행을 깨는 이득은 없다.

바디는 사람이 읽는 한 문장이다. "요청이 너무 많습니다. 30초 후 다시 시도하세요." 스택 트레이스와 내부 상태는 넣지 않는다. 429 바디는 상대

로그를 위한 것이다.

아주 중요한 습관이 하나 더 있다. 거절할 때만 헤더를 보내지 말고 통과할 때에도 보낸다. 클라이언트가 벽에 부딪히기 전에 "100개 중 90개를 썼다"는 걸 알게 된다. 실패할 때만 불이 들어오는 속도 제한 표지판은 함정이다. 잔여 헤더가 있으면 성실한 클라이언트가 스스로 속도를 늦춘다. 그 결과 내 429는 줄어든다.

2.7 사용자당, 그리고 전역

사용자 버킷을 먼저 건다. 사용자가 통과해야 전역 버킷을 거친다. 전역 버킷은 박스의 보험이다.

범위	목적	정하는 법
사용자당	공평	전체가 다 써도 내가 살아남을 속도
전역	생존	측정된 능력보다 아래 천장

전역 숫자는 사용자당 총합보다 작아야 한다. 기대하는 1만 사용자의 한도를 더하면 초당 10만 개가 나와도, 그 박스는 초당 10만 개를 못 쓴다. 전역 한도는 박스가 실제로 버티는 선이고 사용자 한도는 그것을 어떻게 나눌지의 문제다.

키로 IP를 쓰지 말자. 혼자 돌리는 서비스의 클라이언트는 NAT 뒤에서 하나의 IP로 수 십 명의 사람을 대표한다. 모바일 네트워크의 IP는 하루에도 몇 번씩 바뀐다. 안정적인 키는 사용자 ID, 또는 API 키의 해시다. 익명 계층이 있으면 IP별로 묶되, 한도는 훨씬 낮게 잡고 그것을 스팸 대책으로 취급한다.

2.8 rate는 추측이 아니라 측정에서

rate를 어떻게 정하나. 측정한다. 프로덕션을 흉내 낸 스테이징에 부하를 걸어(단순한 루프 스크립트든, k6이든) 지연이 꺾이기 시작하는 지점을 찾는다. 그게 지속 가능한 능력이다. 한도를 그 능력의 70, 80퍼센트에 놓는다. 백 퍼센트에 놓지 않는 이유는, 반드시 넘는 날이 있기 때문이다. 배포 순간에 능력이 빠지고 백그라운드 잡이 CPU를 먹고 예상 못한 트래픽이 친구를 데리고 온다. 20, 30퍼센트의 여유가 429를 일상의 사건에서 드물고 정직한 에러로 만든다.

한 가지만 더. 리미터 자신이 병목이 되는 실패 모드다. 요청 하나마다 Redis 왕복을 한 번씩 하면, 지키려는 요청보다 한도가 비싸진다. 인프로세스 버킷은 거의 공짜이고 공유 상태 버킷은 원자 연산 한 번이면 된다. 한도 검사 자체의 p99 지연을 측정해두라. 핸들러의 지연에 비해서 보이기 시작하면, 과잉 설계다.

2.9 튜닝은 5단계 절차로

rate를 추측으로만 끝내지 않으려면 순서를 고정하자.

1. 측정한다. 보호할 엔드포인트의 유휴 상태 p50과 p99를 잰다.
2. 밀어본다. 부하를 latency가 꺾이는 순간까지 올리고, 그 순간의 RPS를 적는다.
3. 놓는다. rate = 2번의 RPS의 70, 80퍼센트.
4. 버스트를 정한다. capacity = rate x 허용 버스트 초.
5. 검증한다. 10초에 500개 테스트. 통과와 429가 섞이고 p99가 그대로인지 확인.

5단계를 돌리면 한도가 증거가 생긴다. 다음에 rate를 바꿀 때 "그날 그 부하에서 측정했다"는 말이 있고 그 말로 대화가 끝난다. 혼자 운영할 때 이

절차의 가치는 더 크다. 결정은 내가 하고, 그 결정의 근거도 내가 쓴 문장 안에 있기 때문이다.

2.10 흔한 버그, 셋

리미터는 단순해 보이니까 리뷰를 건너뛰기 쉽다. 현장에서 나오는 버그는 대개 아래 셋이다.

1. 벽시계. time.time()을 쓰는 코드는 NTP가 시계를 교정하는 순간 뒤틀린다. 버킷이 한 번에 10시간치를 채워 터져 나오고, 반대로 한 시간 동안 아무도 통과 못 하는 날도 있다. 경과 시간에는 단조 시계를 써야 한다. 앞의 코드가 monotonic을 쓴 이유가 이것이다.
2. 설정 변경이 전부 리셋. rate를 바꾸는 순간 기존 버킷들이 새 값으로 초기화되면, 모든 사용자가 갑자기 버스트를 쏠 수 있다. 토큰 잔량은 살리면서 refill 속도만 바꾸면 변경이 부드러워진다.
3. 429 로그를 안 남기는 것. 429가 나오면 "아, 한도가 일했네" 하고 넘어가지만 그 수가 몇 개였는지는 모른다. 429 건수를 메트릭으로 남겨두지 않으면 "한도가 조금 빡빡한 걸까, 한도에 여유가 전혀 없는 걸까"를 구분할 수 없다. 시간당 429 수는 튜닝의 가장 중요한 지표다.

2.11 이번 장 연습

가장 많이 불리는 엔드포인트에 사용자당 토큰 버킷을 붙이자. rate는 측정으로, 없으면 보수적인 추측으로. capacity = rate x 2. 429에는 Retry-After를, 모든 응답에는 잔여 헤더를. 그리고 두 번째 머신, 또는 스크립트로 10초에 500개를 쏴서, 로그에 통과와 429가 함께 찍히고 프로세스가 죽지 않는지 확인하자. 그 확인을 하는 순간, 한도는 사실이 된다.

3 클라이언트 측면: 백오프와 로드 셰딩

서버에서 429를 보내는 것은 이야기의 반쪽이다. 나머지 반쪽은 그것을 받는 클라이언트다. 내 서비스가 결제 제공자, LLM, 데이터베이스를 부를 때, 나는 이미 누군가의 레이트 리미트 뒤에 서 있는 클라이언트다. 이 장에서 클라이언트 쪽 근육 두 개를 만든다. 실패했을 때 물러서는 법(백오프)과, 혼잡할 때 버리는 법(로드 셰딩).

3.1 무심한 재시도는 두 번째 재앙

현장에서 가장 흔한 재시도 코드는 이 모양이다.

```
try:
 call_provider()
except Exception:
 call_provider()
```

제공자가 과부하로 흔들리기 시작하면, 모든 요청이 거의 같은 순간에 실패한다. 그리고 모든 클라이언트가 거의 같은 순간에 재시도한다. 가까스로 버티고 있던 제공자가 두 배 크기의 물결을 또 받고, 또 무너진다. 재시도가 또 도는 동안, 트래픽은 매 라운드 두 배씩 불어난다. 제공자가 회복되는 데 분 단위 시간이 걸리는 이유는 이 재시도 폭풍 때문이다.

재시도가 안전하려면 두 조건 중 하나가 깨져야 한다. 재시도가 실패보다 충분히 늦거나, 다른 클라이언트들과의 시점이 어긋나거나. 지수 백오프가 전자를, 지터가 후자를 담당한다.

숫자를 넣어보자. 1,000개의 클라이언트가 흔들리는 제공자를 부르고 있고, t=0에 500개가 실패했다. 모두 t=0.5에 같은 시점에 재시도하면, 제공자는 나머지 500개의 정상 요청에 재시도 500개를 더해, 같은 0.5초

간격에 1,000개를 받는다. 또 실패하면 t=1.0에 1,000개의 재시도가 돌아온다. 제공자의 부하는 줄지 않고 매 라운드 두 배씩 늘어난다. 원래 사고는 "클라이언트의 절반이 실패"였는데, 클라이언트들이 만든 사고는 "전체가 실패"다. 제공자가 필요한 것은 시간이다. 그리고 그 시간을 줄 수 있는 쪽은, 백오프하는 클라이언트다.

3.2 지수 백오프: 기다림이 커지는 법

아이디어는 단순하다. 실패가 계속될수록 더 오래 기다린다. 첫 재시도는 1초 후, 둘째는 2초 후, 셋째는 4초 후. 제공자가 3초 만에 회복되면, 대부분의 클라이언트는 네 번째 시도에 이르러서는 이미 줄을 서서 차례를 기다리고 있다. 회복 순간의 트래픽이 평시 수준을 다시 넘지 못한다.

```
import random
import time

def call_with_backoff(fn, retryable, max_retries=5, base=1.0,
↪  cap=30.0):
 for attempt in range(max_retries):
 try:
 return fn()
 except Exception as e:
 if attempt == max_retries - 1 or not retryable(e):
 raise
 wait = min(cap, base * (2 ** attempt))
 time.sleep(random.uniform(0, wait))
```

수비막이 두 개인다. 재시도 횟수 상한(max_retries). 영원히 재시도하는 루프는 좀비다. 죽은 제공자에게 계속 트래픽을 보내는 좀비 클라이언트는 장애를 연장하는 존재다. 기다림 상한(cap)도 그렇다. 256초를 자고 한 번 더 시도하는 것은 형식이다. cap 이후에는 기다림의 가치가 늘지 않으므로,

상한을 찍고 일찍 포기하는 쪽이 시스템에 공헌한다.

3.3 지터: 줄을 깨는 무작위

모두가 같은 공식으로 백오프하면, 모두가 같은 시각에 깨어난다. 1, 2, 4, 8초. 천 개의 클라이언트가 동시에 8초 뒤에 일어나면, 다시 스파이크가 하나 생긴다. 패턴은 그대로인데, 늘어진 버전일 뿐이다.

지터는 기다림을 무작위로 흔든다. 앞의 코드가 full jitter다. 기다림을 base의 2의 시도번째 거듭제곱 값, 정확히 그만큼으로 두는 대신, 0부터 그 값 사이에서 무작위로 고르는 것. 스파이크가 낮은 평탄한 흐름으로 퍼진다. "조금 기다리자"에서 "무작위로 기다리자"로 바꾸는 한 단어의 차이가, 제공자의 회복 시간을 분에서 초 단위로 바꾼다.

서버가 Retry-After를 보내면, 그 값을 지키자. 그것은 서버가 자기 상황으로 계산한 시간이다. 최소한 그 만큼은 기다리고 그 위에 지터를 얹는다. Retry-After를 무시하는 클라이언트는, 조용한 방에서 소리치는 사람이다.

3.4 어떤 에러를 재시도하나

모든 에러가 재시도의 자격을 갖지는 못한다.

에러	재시도	이유
429, 503	예	용량 문제. 나중에 통할 수 있다
5xx (나머지)	1, 2회	일시적일 수 있다
4xx (기타)	아니오	그대로 다시 실패한다
타임아웃	주의	이미 성공했을 수 있다

4xx, 특히 검증 에러는 "요청 자체가 틀렸다"는 말이고 다시 보내어도

달라지는 것이 없다. 예산만 태운다. 타임아웃은 더 까다롭다. 요청은 서버에 도착해 처리됐을 수 있고 답만 잃어버렸을 수 있다. 결제를 타임아웃에 재시도하면 두 번 과금될 수 있다. 그래서 다음 소절이 필요하다.

3.5 멱등: 재시도의 전제

멱등한 요청은 몇 번을 반복해도 결과가 같은 요청이다. "사용자 42번 이름을 김으로 고른다"는 멱등하다. "새 결제를 만든다"는 멱등하지 않다.

멱등하지 않은 호출을 안전하게 재시도되게 하는 방법은 두 개인다. 하나, 서버 쪽 멱등 키를 쓴다. 클라이언트가 논리적 조작 하나당 UUID를 만들고 헤더로 보내면, 서버는 그 키를 기억하고 같은 키가 다시 오면 처음 결과를 다시 돌려준다. 결제 제공자들은 정확히 이걸 제공하니까, 쓰는 것이 공짜다. 둘, 조작 자체를 중복 탐지 가능하게 만든다. 주문 번호에 고유 제약, 또는 쓰기 전에 "이미 처리됐나" 확인.

규칙으로 줄인다. 뭔가를 만드는 POST에는 눈먼 재시도를 금지한다. 키가 없으면, 확인이 있어야 한다. 둘 중 하나를 붙이지 못하면, 그 호출은 재시도 대상에서 제외된다.

3.6 백오프의 마지막 형상

재시도 예산이 다 빠지면 어떻게 되나. 많은 서비스에서 백오프의 이야기는 범용 500으로 끝나고 그건 낭비다. 나를 부르는 클라이언트도 누군가의 서비스고, 내가 포기하는 형식은 내 API의 일부다.

포기할 때는 두 가지를 말하자. 첫째, 무엇이 일어났는지 내 서비스의 어휘로. "결제 제공자를 지금 쓸 수 없습니다. 잠시 후 다시 시도하세요." 내부 예외 이름이 아니라, 그 문장이다. 둘째, 클라이언트가 무엇을 해야 하는지. 지금은 재시도하지 말 것, N분 후에 재시도할 것. 내 API에

그 작업의 비동기 버전이 있으면, 거기로 유도하자. "주문은 접수해 두겠습니다. 백그라운드로 처리해서 알려드릴게요"는, 장애 중에 실제로 동작하는 유일한 형태인 경우가 많다.

포기는 기능의 마지막 단계다. "어떻게 포기할지"를 설계 질문으로 여긴 서비스가, 최악의 날에도 서 있다.

3.7 서킷 브레이커: 죽은 의존성에 전화를 멈추는 법

백오프는 일시적 실패를 다룬다. 의존성이 30분간 죽어 있으면, 백오프는 정중하게 시간을 낭비하는 방법이다. 서킷 브레이커는 잠시 전화를 멈추는 패턴이다.

브레이커는 세 상태를 가진다.

상태	행동	전환
닫힘 (closed)	호출을 통과시킨다	실패율 임계 초과 시 열림
열림 (open)	즉시 거절한다	쿨다운 만료 시 반열림
반열림 (half-open)	탐침 하나를 통과	성공하면 닫힘, 실패면 열림

닫힘은 평시 운영이다. 최근 호출을 세고 예컨대 최근 20개 중 50퍼센트가 실패하면 임계에 닿아 열림으로 튄다. 이 숫자는 튜닝 예시일 뿐 법이 아니다. 내 서비스의 실패 패턴에 맞게 바꾼다. 열림은 호출을 멈추고 빠르게 실패한다는 뜻이다. 내 에러는 분명하고 죽은 박스에 트래픽을 보내지 않는다. 쿨다운, 예컨대 10, 30초가 지나면 반열림으로 가서 탐침 하나를 보낸다. 탐침이통과하면 닫힘으로 돌아가고 실패하면 다시 열림으로, 쿨다운은 재시작된다.

왜 기다림 대신 빠르게 실패하는가. 죽은 의존성에 대한 호출은 타임아웃이 될 때까지 워커 하나, 연결 하나, 사용자의 인내 하나를 태운다. 즉시

거절하면, 성공할 수 없는 요청이 거의 무값이 되고 그 워커는 성공할 수 있는 다른 요청을 받아들인다. 혼잡 시에 이것은 생존이다.

브레이커 숫자에 함정이 하나 있다. 저트래픽 서비스다. "최근 20개 호출 중 50퍼센트"라는 규칙을, 분당 10개 호출밖에 안 하는 내가 쓰면, 20개 호출의 윈도우는 2분이다. 제공자의 30초 장애가 "윈도우 전부 실패"로 변하고 브레이커가 열려 있는 동안 회복은 몇 분을 기다린다. 반대로 분당 1,000개 호출이면 20개 윈도우는 2초고, 단일 블립에 트립하고 풀림을 반복한다.

수정법은 시간으로 생각하는 것이다. "최근 N초의 실패율이 X퍼센트면" 이면, 트래픽 규모와 무관하게 같은 규칙이 된다. 개수 기반 윈도우는, "최근 20개 호출"의 의미가 시간적으로 안정된 트래픽에서만 쓰는 도구다.

1인 개발자라면 브레이커를 카운터와 타임스탬프 몇 개로 만들 수 있다. 라이브러리가 필요 없다. 다만 언어에 검증된 브레이커가 있으면 그것을 쓰는 것이 낫다. 손으로 만든 상태 머신은 재미보다 버그가 많다.

3.8 429 핸드셰이크를 처음부터

핸드셰이크를 순서대로 한 번 걸어보자.

1. 클라이언트가 요청을 보낸다.
2. 서버가 용량을 넘은 상태라 429와 Retry-After: 30을 돌려준다.
3. 클라이언트가 실패 횟수를 세고 기다림 = 30 + random(0, 5)을 고른다.
4. 잔다. 이 동안 워커를 잡지 않는다. 스레드는 풀린다.
5. 깨어나서 재시도한다.
6. 또 429면 3번으로. 기다림의 기저는 지수처럼 두 배씩 커진다.
7. 횟수 상한이면, 백오프의 마지막 형상으로 자신의 클라이언트에게 답한다.

이 순서에 세부 두 가지가 있다. 3번의 기다림은 서버 값을 기준으로 삼아야

한다. 내 지수 공식이 아니라. 서버는 자기 회복 시간을 클라이언트보다 잘 안다. 4번도 그렇다. 자는 동안 연결을 쥐고 있으면, 조용히 매달리는 것이다. 백오프는 "일부러 트래픽을 보내지 않는 시간"이고 그 시간의 전제는 워커를 푸는 일이다.

재시도의 대가도 다시 말하자. 내 재시도는 제공자의 한도에 내 트래픽으로 지불되는 것이다. 재시도를 무심하게 쓰면, 내 서비스의 예산으로 제공자의 한도를 태우는 셈이다. 재시도 횟수 상한은 내 429를 위한 수비다.

3.9 로드 셰딩: 무엇을 버릴지 정해둔다

백오프와 브레이커는 실패를 다룬다. 로드 셰딩은 혼잡을 다룬다. 의존성은 멀쩡한데, 나는 따라가지 못하는 상황. 큐가 자라고 지연이 늘어나고 무언가가 양보해야 한다.

규율은, 억지로 결정당하기 전에 내가 버릴 것을 정해두는 일이다. 무작위 버림은 동전 던지기이고 계획된 버림은 정책이다. 대개의 우선순위 순은 이렇다.

1. 먼저 버린다. 분석 로그, 캐시 갱신, 메일 발송, 긴급하지 않은 백그라운드 작업.
2. 다음으로 버린다. 비싼 읽기 기능. 요약 생성, 추천 목록, 어제 데이터로 대체될 수 있는 것.
3. 가장 마지막까지 지킨다. 핵심 거래. 결제, 인증, 사용자가 서비스 존재의 이유로 온 요청.

구현은 생각보다 단순하다. 버려도 되는 작업 주위에 체크를 돌린다. "부하가 높으면, 건너뛰거나 큐에 넣는다"는 한 줄이다. 부하가 높다는 신호는 내가 정한다. 큐 길이, 인플라이트 수, CPU, 또는 비상시에 내가 직접 내리는 수동 플래그. 수동 플래그는 과소평가된다. 트래픽이 폭주할 때, "지금부터 분석 로그를 꺼둔다"고 사람이 결정하는 제어는 오탐이

제로인 단추다.

샘플링도 있다. 기능을 전부 끌 수 없다면, 일부를 끈다. 부하가 높을 때 로그를 요청의 10분의 1만 남긴다. 기능은 살아 있고 비용은 90퍼센트 내려간다.

그리고 무엇을 절대 안 버릴지의 한 줄. 핵심 경로는 정체성의 문제다. 내 서비스의 핵심이 체크아웃이면, 요약 기능을 살리려고 체크아웃을 버린 순간, 나는 "약화된 모드의 서비스"가 아니라 "다른 서비스"를 돌리고 있다. 결제하러 온 사용자가 "결제는 바쁘고 추천은 잘 보입니다"라는 메시지를 보면 이해하지 못한다. 그냥 떠난다. 셰딩은, 남은 것만으로 원래의 서비스인지 알아볼 수 있게 해야 한다.

3.10 이번 장 연습

가장 많이 부르는 의존성을 하나 고른다. 세 가지를 붙인다. 지터가 있는 지수 백오프와 재시도 횟수 상한. 빠르게 실패할 만큼 짧은 타임아웃. 뭔가를 만드는 호출이면 멱등 키. 그리고 의존성이 죽은 상황을 의도적으로 만들어본다. 설정을 죽은 포트로 돌리면 된다. 내 서비스가 어떻게 degrade하는지 관찰한다. 요청이 분명하게 실패하고 워커는 비어 있고 의존성을 회복시키면 트래픽이 벽이 아니라 완만하게 돌아와야 한다. 벽이 보인다면, 지터가 일하고 있지 않다.

4 LLM API: 비용과 트래픽을 혼자 막기

LLM을 서비스에서 부르기 시작하면, 위험의 모양이 바뀐다. 보통 API 호출은 비용이 상한이 있다. 밀리초 몇의 계산, 정가 하나. LLM 호출은 토큰당 과금이고 응답은 몇 초를 먹고 길이까지 모델이 정한다. 하루에 100개 요청만 처리하던 기능이, 사용자가 문서를 통째로 붙여넣는 날이면 한 시간 안에 월 예산을 태울 수 있다. 이 장에서는 두 가지를 지킨다. 청구서, 그리고 트래픽 폭주 중인 내 서비스.

4.1 비용 문제는 예산 문제

가장 흔한 첫 번째 실수는, LLM 비용을 모니터링 문제로 푸는 것이다. 지출이 올라가는 것을 보여주는 대시보드는 사후에 쓸모가 있다. 자고 있으면 쓸모가 없다.

동작하는 모델은, 예산이 집행되고 지출에 천장이 있다는 것이다. 예산은 세 개다.

- 일일 예산. 하루에 잃어도 되는 절대 상한. 튜닝 예시로 50달러를 적어두지만 내 수면과 상관없는 숫자여야 한다.
- 요청당 예산. 한 호출이 태울 수 있는 최대값. 입력은 컨텍스트 길이로, 출력은 max token으로, API 파라미터로 씌운다. 파라미터로 제어한다.
- 사용자당 예산. 한 계정이 하루에 태울 수 있는 상한. 나를 깨뜨리는 건 대개 큰 사용자 하나인데, 사용자당 천장이 그 한 마리를 상한 있는 계좌로 바꾼다.

세 개 모두 서버 쪽, 호출 직전에 집행한다. 일일 예산이 다 쓰이면, 핵심이 아닌 LLM 기능은 정중한 문장으로 대체된다. “AI 답변은 오늘 자리가

찼습니다. 내일 다시 시도해 주세요.” 문장이 청구서보다 낫고 침묵보다 낫다.

계산을 하나 해두자. 단가를 예시로 쓴다. 입력 1M tokens당 3달러, 출력 1M tokens당 15달러. [예시 단가다. 내가 쓰는 모델의 표를 대입해야 한다.] Q&A 호출 하나를 입력 2,000 tokens, 출력 300 tokens로 잡으면, (2000 x 3 + 300 x 15) / 1,000,000 = 0.0105달러. 호출당 약 1센트다. 하루 1,000 call이면 10달러 50센트. 일일 예산 50달러는 약 4,700 call의 여유다.

이 계산이 주는 것은 숫자만이 아니다. “무거운 사용자”의 의미를 준다. 문서를 통째로 붙여넣는 호출은 입력이 2,000 tokens가 아니라 12,000 tokens이고 그걸 루프로 돌리면 1분 만에 50달러 예산의 절반이 사라진다. 요청당 예산이 컨텍스트 천장으로 이 호출을 8,000 tokens까지 자른다면, 같은 루프도 예산의 일부 안에서 끝나게 된다. 천장은 추상적이지 않다. 센트로 읽힌다.

4.2 비용 서킷 브레이커

3장의 브레이커는 에러에 튄다. LLM에는 돈에 튀는 두 번째 브레이커가 필요하다.

최근 1분간 지출을 센다. 임계, 예컨대 평소 분당 지출의 3배를 넘으면, 비용 브레이커가 10분간 튄다. 열려 있는 동안 핵심 경로 호출만 통과하고 나머지는 모두 shed된다. 임계와 시간은 튜닝 예시다. 중요한 것은 브레이커가 존재하고, 어떤 호출이 살아남는지 내가 정했다는 점이다.

왜 창이 1분인가. 비용 이상은 “지금”의 문제라서다. 10분, 1시간 평균은 지금 돈을 태우는 스파이크를 감춘다. 일일 예산은 느린 난간이고 비용 브레이커는 빠른 난간이다. 둘 다 필요하다.

4.3 호출 전의 세 관문

요청이 제공자에게 닿기 전에 관문을 세 개 지나는 구조다.

관문	검사	효과
예산	남은 예산에 들어맞나	지출 전 정지
캐시	같은 질문의 답이 있나	비용 제로
동시성	빈 자리가 있나	지출 속도 제어

동시성 상한은 세마포어(semaphore), 또는 고정 크기 큐다. LLM 호출을 동시에 N개까지만 띄우고, 나머지는 기다리거나 거절한다. N은 두 주인을 모신다. 제공자의 동시성 한도(RPM, TPM)와, 내 지출 속도. 플랜이 분당 500 requests를 허락해도, 호출당 2,000 tokens를 쓰면 TPM 천장이 RPM보다 먼저 온다. 그래서 N은 토큰으로 정한다.

캐시는 1인 운영자가 가장 큰 수익을 보는 곳이다. Q&A 기능의 질문은 생각보다 많이 겹친다. 거의 동일한 입력, 모델과 시스템 프롬프트와 사용자 입력의 해시로 키를 만들어 조회한다. 명중했다면, 저장된 답을 제로 비용으로 돌려준다. TTL은 짧아도 된다. 10, 60분이면 반복되는 질문이 거의 공짜가 된다. [서포트 유형의 Q&A 기능에서 30, 70퍼센트의 호출이 캐시 명중한다는 정도는 대략적 추정이다. 내 기능에서 재라.]

예산 관문의 토큰 추정도 공짜다. 글자 수를 4로 나누면 대부분의 문자열에서 토큰 수의 근사가 된다. [모든 모델에 정확한 규칙은 아니다. 근사다.] 토큰당 단가와 곱해서 남은 예산에 비교한다. 들어맞지 않으면, 호출 전에 거절한다. 보내지 않은 요청은 비용이 없다.

4.4 캐시를 조금 더 설계하자

캐시의 키는 (model, system prompt, user input)의 해시다. 여기에 하나 더 섞자. 프롬프트 버전이다. 시스템 프롬프트를 고친 날, 옛 답을 새 프롬프트의 답인 것처럼 돌려주면 안 되므로, 버전이 바뀌면 키가 바뀌어야 한다. 모델 이름도 키에 들어가야 한다. 모델 이름이 바뀌면 같은 프롬프트에도 답이 달라지기 때문이다.

캐시하지 말아야 할 것에도 규칙이 있다. 사용자 이름, 주문 번호 같은 개인화된 값이 답에 섞이는 호출은, 그대로 캐시하면 A 사용자의 답이 B 사용자에게 돌아간다. 개인화된 부분을 템플릿에서 빼고 "이 부분만 매번 실행한다"는 구조로 고치는 것이 낫다. 또는 그 호출의 캐시를 무효화하는 것이다.

TTL 정책은 기능마다 다르다. FAQ는 하루, "지금 날씨" 같은 실시간성은 5분, 내 서비스의 문서 요약은 1시간. TTL을 정할 때 기준은 "얼마나 자주 변하나"가 아니라, "새로운 답이 필요한 순간을 사용자가 알아챌 수 있나"다. 알아챌 수 없는 기능은 짧은 TTL이 안전하다.

4.5 타임아웃, 재시도, 과금 함정

3장의 재시도 규율이 그대로 적용되지만, LLM에는 규칙이 하나 더 있다. 성공을 확인할 수 없는 호출에는 재시도하지 않는다.

제공자의 타임아웃은, 요청이 처리되고 과금됐을 수 있다는 뜻이다. 읽기 타임아웃마다 재시도하면, 같은 답에 두 번 지불한다. 그래서 재시도 신호를 명확한 것으로만 좁힌다. 429와 5xx, 명시적 에러, 요청이 나가기 전에 터진 연결 에러(DNS, 거부). 요청이 나간 뒤의 읽기 타임아웃은 재시도 신호가 아니라, "성공했는지 확인하라"는 신호다. 제공자가 요청 상태를 조회하거나 중복을 제거하는 수단을 주면, 그것을 써라. 주지

않으면, 중복이 해롭지 않은 형태로 설계하라. 캐시된 답은 멱등이고 중복의 대가는 청구서 한 장이다.

429에는 지터가 있는 백오프로, Retry-After가 있으면 그 값으로. LLM 제공자들은 레이트 리미트에서 보내주는데, 무시하는 것은 키를 더 강하게 조여 달라고 요청하는 것과 같다.

4.6 비용 로그: 사후의 유일한 증인

브레이커가 튄 이유를, 청구서가 왜 그 크기인지를, 나중에 설명할 수 있어야 한다. 그러려면 호출당 비용 로그가 필요하다. 다음 값들을 한 줄로 남기면 된다.

- 시각, 사용자(또는 익명 식별자), 기능
- 입력 token 수, 출력 token 수
- 계산된 비용
- 캐시 명중 여부

이 다섯 값이 있으면, "어떤 기능이 예산의 몇 퍼센트를 쓰는지"를 다음 날 아침에 알 수 있다. 비용 브레이커의 임계도 이 로그 위에서 조정한다. 평소 분당 지출이라는 값은 추정이 아니라, 이 로그의 평균이다. 로그를 안 남기면, 예산 튜닝은 매번 운이다. 1인 운영자에게 이 로그는 회계 장부이고 대시보드의 원료이고, 폭주 때의 나침반이다. 세 개의 역할이 한 테이블에 들어간다.

4.7 트래픽 폭주 플레이북

그날이 온다. 서비스에 대한 글이 퍼지고 아침에 트래픽이 열 배다. 1, 2, 3장을 세웠다면, 내 서비스는 문 앞에서 거절하고(429), 핵심 밖은 버리고(로드 셰딩), 실패는 물러서고(백오프) 있다. 남은 일은, 청구서를 지키면서 핵심을 살리는 순서다.

1. 비용 대시보드와 에러율을 본다. 2분 안에 현재 상태를 세운다.
2. 동시성 상한을 조인다. N을 절반으로. 응답은 느려지지만 지출 속도는 준다.
3. 비용 브레이커를 낮은 우선순위로 먼저 튼다. 요약 생성, AI 댓글 답변을 끄고 핵심 대화는 살린다.
4. 캐시 TTL을 늘린다. 트래픽이 몇 가지 뜨거운 질문에 몰려 있으면, 캐시 명중은 오르고 제공자 호출은 줄어든다.
5. 비대화형 작업을 큐로 옮긴다. 배치 보고서, 메일 다이제스트는 나중에 전달한다. 사용자는 "바빠요. 1시간 안에 드립니다"를 보고 실패를 보지 않는다.

순서가 중요하다. 먼저 지출 속도를 줄이고 다음에 핵심을 지킨다. 잃은 처리 능력의 보상은 그 다음 생각이다. 흔히 하는 순서 오류는 거꾸다. 트래픽을 받으려고 박스를 늘리고 청구서가 트래픽과 함께 자란다. 트래픽이 늘어난 것은 더 많이 지불하라는 이유가 아니라, 더 많이 거절하라는 이유다.

폭주에서 하나 더. 사용자당 예산이 있으면, 큰 사용자는 나의 최대 위험이다. 계정 하나가 무거운 프롬프트를 루프로 돌리면, 일반 사용자 천 명이 넘는 돈을 태울 수 있다. 사용자당 천장이 그 공격을 상한 있는 청구서로 바꾼다.

4.8 약화 모드의 문장

LLM이 비어 있을 때 사용자에게 보여줄 문장도 미리 써두자. 사고가 난 뒤에는 문장을 쓸 시간이 없다. 미리 써둘 문장 셋.

1. 기능 전체가 비었을 때. "AI 기능은 잠시 쉬어갑니다. 보통은 10분 안에 돌아옵니다." 다른 기능이 작동 중임을 보여주는 문장이 신뢰를 살린다.
2. 답이 늦을 때. "요청이 많아서 답이 평소보다 걸립니다. 기다려

주시거나, 더 짧은 질문을 보내 주셔도 됩니다." 사용자가 스스로 조율할 수 있는 문장이다.

3. 특정 사용자의 한도에 닿았을 때. "오늘 AI 사용 상한에 도달했습니다. 내일 0시에 초기화됩니다." 언제 돌아오는지 말한 문장은, 이탈을 줄인다.

셋 공통의 규칙은 하나다. 내 상황을 책임으로 돌리지 않고 사용자가 취할 다음 행동을 말한다. "죄송합니다"만 있는 문장은 사용자가 아무것도 할 수 없게 만든다. 폭주 중에 이 문장들이 자동으로 나오려면, 429와 비용 브레이커의 응답 경로에 이 문장들이 이미 코드에 들어가 있어야 한다.

4.9 제공자도 천장이 있다

레이트 리밋은 내 박스만의 문제가 아니다. 제공자도 자기 한도를 갖고 있다. 플랜의 RPM, TPM, 동시성은 계약이고 그 위에서 429를 받는다. "내가 먼저 거절"과 "상대가 먼저 거절"의 차이는 이렇다. 전자는 내가 통제하는 메시지이고 후자는 내 사용자가 보는 실패다.

그래서 동시성 관문의 N은 계약 아래에 놓이는 숫자다. 제공자 429를 받는 순간은, 내 관문이 일하지 못했다는 신호다. 그 때 할 일은 N을 낮추고 로그를 보는 일이지, 플랜 업그레이드를 주문하는 일이 아니다. [업그레이드가 필요한 날도 있다. 그러나 그 판단은 로그가 먼저여야 한다.]

4.10 마지막 조각: 자동으로 만들 것

이 장에 쓴 모든 것이, 없어도 돌아가야 한다. 비용 브레이커가 알아서 튀고 429가 알아서 나가고 문장이 알아서 보여야 한다. 트립의 순간에 내가 대시보드를 보고 수동으로 스위치를 꺾어야 한다면, 방어는 내가 보는 시간에만 있다.

실무의 선은 이렇다. 전부 "값을 계산해서 비교"로 만든다. 판단과 눈대중을

넣지 않는다. 일일 예산은 “쓸 수 있는 것이 남아 있는가”, 비용 브레이커는 “최근 1분 지출이 임계 아래인가”, 동시성은 “빈 자리가 있는가”. 셋이 모두 이어져 있으면, 서비스는 내가 자는 동안에도 스스로를 지킨다. 아침에 깨서 “새벽 3시에 브레이커가 트립했고 3시 12분에 회복됐다”는 로그를 읽는 것. 이 책의 규율이 생겼을 때의 모양이 바로 그 로그 한 줄이다.

4.11 이번 장 연습

사용자 한 명의 최악의 경우를 계산한다. 보낼 수 있는 가장 긴 프롬프트(컨텍스트 천장), 가장 긴 답(max token 천장), 토큰당 단가, 시간당 최대 호출 수(허용한 rate). 네 개를 곱하면, 한 사용자가 한 시간에 내게 태울 수 있는 최대값이 나온다. 그 숫자가 불편하다면, 네 개 중 하나를 낮춘다. 그리고 사용자당 일일 예산을 그 배수로 놓고 비용 브레이커를 내 알림 채널에 이어붙인다. 규율은 스파이크가 오지 않는다는 것이 아니다. 스파이크가 올 때, 천장이 무엇인지 알고, 어떤 기능을 살릴지 정해져 있다는 것이다.