재는 기술

믿을 수 있는 성능 측정과 벤치마크의 규율

ThakiCloud (Hyojung Han)

재는 기술

믿을 수 있는 성능 측정과 벤치마크의 규율

ThakiCloud (Hyojung Han)

Contents

1 무엇을 잴 것인가: 지표는 관측이 아니라 선택이다

성능 측정은 자연을 관찰하는 일처럼 보이지만, 실제로는 선택하는 일입니다. 시스템은 초당 수백만 개의 사건을 만들어내고 우리는 그중 극히 일부를 골라 숫자로 요약합니다. 무엇을 고르느냐가 결론의 절반을 이미 결정합니다. 그래서 측정의 규율은 장비나 도구에서 시작하지 않고 "우리는 무엇을 재기로 했는가"를 문장으로 적는 데서 시작합니다.

이 장에서 다루는 것은 딱 하나입니다. 숫자를 만들기 전에 지표를 고르는 절차를 갖추는 일입니다. 절차가 없으면 팀은 대개 도구가 기본으로 뿌려주는 값을 그대로 씁니다. 부하 도구가 평균 응답 시간을 찍어주니 평균을 보고하고 대시보드에 초당 요청 수 그래프가 있으니 그것을 성능이라 부릅니다. 도구의 기본값이 조직의 성능 정의가 되는 순간, 그 조직은 자기가 무엇을 최적화하는지 모르게 됩니다.

1.1 지표를 세 칸으로 나눈다

지표를 한 덩어리로 다루면 논쟁이 끝나지 않습니다. 회의에서 "지연이 줄었는데 왜 반대하냐"와 "대신 메모리가 늘지 않았냐"가 평행선을 달리는 이유는 두 사람이 서로 다른 역할의 지표를 같은 무게로 놓고 있기 때문입니다. 지표를 역할별로 세 칸에 나눠 담으면 이 논쟁이 구조적으로 사라집니다.

첫째 칸은 목표 지표입니다. 이번 작업이 개선하려는 대상이며 딱 하나여야 합니다. 둘이 되는 순간 트레이드오프를 판단할 기준이 사라집니다. 둘째 칸은 가드레일 지표입니다. 개선하지 않아도 되지만 나빠지면 안 되는 값입니다. 셋째 칸은 진단 지표입니다. 목표가 왜 움직였는지 설명하기

위한 값이며 이 값 자체를 목표로 삼지는 않습니다.

역할	개수	판정 방식
목표	하나	개선 폭
가드레일	둘에서 넷	악화 한도
진단	제한 없음	설명용

추론 서빙을 예로 들면, 목표는 동시 사용자 300명에서의 응답 지연 95백분위, 가드레일은 요청 실패율과 첫 토큰까지 걸리는 시간과 시간당 비용, 진단은 큐 대기 시간과 배치 크기 분포와 캐시 적중률이 됩니다. 이렇게 적어두면 "지연은 좋아졌는데 비용이 두 배"라는 결과가 나왔을 때 논쟁 없이 판정됩니다. 가드레일 한도를 넘었으니 채택하지 않는 것이지요.

1.2 평균은 왜 거의 항상 틀린 답인가

평균 응답 시간은 측정 문화에서 가장 오래 살아남은 나쁜 습관입니다. 평균이 나쁜 이유는 부정확해서가 아니라 사용자가 경험하는 것과 다른 것을 측정하기 때문입니다.

응답 시간 분포는 대개 오른쪽으로 길게 늘어집니다. 대부분의 요청은 빠르고 소수가 아주 느립니다. 이 분포에서 평균은 느린 소수에 끌려다니면서도 그 소수가 얼마나 느린지는 감추는 이상한 자리에 놓입니다. 평균 120밀리초라는 문장은 5백분위가 20밀리초이고 99백분위가 3초인 상황과 모든 요청이 120밀리초인 상황을 구분하지 못합니다. 두 시스템은 사용자 입장에서 완전히 다른 제품입니다.

여기에 곱셈 효과도 있습니다. 화면 하나를 그리는 데 백엔드 호출이 열 번 들어간다고 해봅시다. 각 호출이 독립적으로 99백분위 지연을 겪을 확률이 1퍼센트라면, 열 개 중 적어도 하나가 꼬리에 걸릴 확률은 대략 10퍼센트에 가깝습니다. 개별 서비스에서는 드문 사건이 화면 단위에서는 열 명 중 한

명의 일상이 됩니다. 마이크로서비스가 늘어날수록 꼬리 지연이 곧 제품의 얼굴이 되는 이유가 여기 있습니다.

그래서 기본값을 바꿔야 합니다. 지연을 보고할 때는 중앙값과 95백분위와 99백분위를 함께 적고 평균은 굳이 필요할 때만 덧붙입니다. 백분위를 여러 구간에서 합칠 때는 주의가 필요합니다. 서버 열 대의 99백분위를 산술 평균한 값은 전체의 99백분위가 아닙니다. 백분위는 평균을 낼 수 있는 양이 아니며 합치려면 원본 분포나 히스토그램을 합쳐야 합니다.

1.3 처리량과 지연은 한 쌍이다

"우리 시스템은 초당 5천 건을 처리합니다"라는 문장은 그 자체로는 아무 정보도 담고 있지 않습니다. 어떤 지연을 허용했을 때의 5천 건인지가 빠져 있기 때문입니다. 큐를 무한정 길게 두면 거의 모든 시스템이 처리량 숫자를 올릴 수 있습니다. 대기열에 쌓인 요청은 언젠가 처리되고 처리된 건수는 처리량에 잡히며 그동안 사용자는 화면을 보며 기다립니다.

반대 방향도 마찬가지입니다. "지연 20밀리초"라는 문장도 부하 조건이 없으면 무의미합니다. 아무도 쓰지 않는 시스템은 언제나 빠릅니다.

그래서 성능은 점이 아니라 곡선으로 말해야 합니다. 부하를 단계적으로 올리면서 각 단계의 지연 백분위를 기록하면, 지연이 완만하게 오르다가 갑자기 꺾이는 지점이 보입니다. 그 무릎 지점이 시스템의 실질 용량이고 운영 목표는 대개 그보다 여유 있게 잡습니다. 용량 계획 회의에 가져가야 할 산출물은 하나의 숫자가 아니라 이 곡선입니다.

곡선을 그릴 때는 부하를 만드는 쪽의 한계도 살펴야 합니다. 부하 생성기가 먼저 포화되면 우리는 대상 시스템이 아니라 측정 도구의 성능을 재게 됩니다. 부하 생성기를 두 배로 늘렸을 때 결과가 달라진다면 이전 측정은 대상이 아니라 도구를 측정한 것입니다.

1.4 비율 지표는 분모부터 적는다

효율을 말할 때 우리는 비율을 씁니다. 요청당 비용, 토큰당 에너지, 달러당 처리량 같은 값들입니다. 비율 지표는 강력하지만 분모가 조용히 바뀌면 완전히 다른 이야기가 됩니다.

토큰당 비용을 예로 들면, 분모의 토큰이 입력과 출력을 합친 것인지 출력만인지에 따라 값이 몇 배 차이 납니다. 토크나이저가 다르면 같은 문장도 토큰 수가 다릅니다. 요청당 비용도 요청의 정의가 재시도를 포함하는지, 실패한 요청을 세는지에 따라 흔들립니다. 비율 지표를 쓸 때는 분자와 분모를 각각 한 문장으로 정의하고 그 정의를 코드의 한 곳에서만 계산하도록 만들어야 합니다. 정의가 두 군데 있으면 언젠가 갈라집니다.

1.5 지표 카드라는 작은 습관

지금까지의 이야기를 실행 가능한 형태로 압축하면 지표 카드가 됩니다. 지표 하나마다 다음 여섯 줄을 채웁니다. 이름, 한 문장 정의, 분자와 분모, 측정 지점, 역할과 한도, 오해 위험입니다.

측정 지점을 적는 칸이 특히 중요합니다. 같은 "응답 시간"이라도 클라이언트에서 재는지 로드밸런서에서 재는지 애플리케이션 안에서 재는지에 따라 다른 값이 나오고 그 차이가 종종 우리가 찾던 문제입니다. 애플리케이션 내부에서 재면 대기열에서 보낸 시간이 통째로 빠지고 그러면 시스템이 포화될수록 측정값은 오히려 좋아 보입니다. 측정 지점을 카드에 적어두는 습관 하나가 이 함정을 막습니다.

오해 위험 칸에는 이 지표가 어떤 식으로 잘못 읽힐지를 미리 적습니다. 캐시 적중률은 높을수록 좋다고 읽히지만 캐시가 커져서 오래된 값을 오래 들고 있는 상황에서도 올라갑니다. 이런 문장을 미리 적어두면 나중에 그 숫자로 잘못된 결정을 내리는 일이 줄어듭니다.

1.6 재지 않기로 한 것도 적는다

제외 목록도 지표 목록 못지않게 중요합니다. 이번 측정에서 다루지 않기로 한 항목을 한 줄씩 적어두면, 나중에 "왜 그건 안 봤냐"는 질문에 방어가 아니라 설명으로 답할 수 있습니다.

제외에는 두 종류가 있습니다. 하나는 이번 변경과 무관하다고 판단해 뺀 것이고 다른 하나는 재고 싶었지만 잴 방법이 없어서 뺀 것입니다. 둘을 구분해 적는 습관이 유용합니다. 앞의 것은 판단이고 뒤의 것은 부채이기 때문입니다. 잴 방법이 없어서 뺀 항목이 계속 쌓이면, 그 목록 자체가 다음 분기에 계측을 보강할 우선순위가 됩니다.

1.7 예시 하나를 끝까지 따라가기

추상적인 원칙은 예시 하나로 훨씬 빨리 이해됩니다. 사내 검색 서비스에 벡터 인덱스를 도입하는 작업을 생각해봅시다. 목적은 검색 품질 향상이고 성능 쪽 우려는 지연과 메모리입니다.

지표 카드를 채워보면 이렇게 됩니다. 목표 지표는 실사용 질의 분포에서의 검색 응답 지연 95백분위이고 측정 지점은 API 게이트웨이 진입 시각부터 응답 완료까지입니다. 가드레일은 세 개를 둡니다. 검색 결과 관련도 점수가 기존보다 떨어지지 않을 것, 인스턴스당 상주 메모리가 한도 아래일 것, 인덱스 갱신 지연이 정해진 시간을 넘지 않을 것입니다. 진단 지표로는 후보 생성 단계와 재순위 단계의 시간 분해, 캐시 적중률, 인덱스 크기를 둡니다.

목표 지표는 "빨라진다"가 아닙니다. 이 작업의 목적은 품질이고 성능은 나빠지지 않아야 하는 조건입니다. 그런데 성능 팀이 참여하면 자연스럽게 지연이 목표처럼 다뤄지기 시작합니다. 카드에 역할을 명시해두면 이런 표류를 막을 수 있습니다.

그리고 제외 목록에는 이렇게 적습니다. 색인 재구축 시의 최대 부하는

이번에 다루지 않음, 다국어 질의 성능은 계측 부재로 이번 범위 밖. 두 번째 항목이 곧 다음 작업의 씨앗이 됩니다.

1.8 지표는 관측 비용도 함께 고른다

지표를 고를 때 자주 빠지는 항목이 계측 비용입니다. 어떤 값은 이미 로그에 있어서 공짜에 가깝고 어떤 값은 추가 계측을 붙여야 하며 어떤 값은 붙이는 순간 대상이 느려집니다.

그래서 지표 후보를 놓고 세 가지를 함께 봅니다. 이 값을 얻는 데 드는 구현 비용, 상시로 켜둘 때의 성능 부담, 그리고 값의 신선도입니다. 매 요청마다 상세 추적을 남기면 정보는 풍부해지지만 처리량이 깎이고 저장 비용이 붙습니다. 이럴 때는 전량 계측 대신 표본 추출로 내려가는 선택지가 있습니다. 저렴한 집계 지표는 100퍼센트로 켜두고 비싼 상세 추적은 몇 퍼센트만 켜는 조합이 실무에서 가장 자주 쓰이는 균형점입니다.

표본 추출을 쓸 때는 한 가지를 조심해야 합니다. 무작위 표본은 중앙값을 잘 추정하지만 극단 꼬리는 잘 놓칩니다. 99.9백분위를 봐야 하는 시스템이라면, 느린 요청은 무조건 남기는 편향 표본을 함께 두어야 합니다. 이 경우 표본이 편향됐다는 사실을 지표 카드에 적어두고 그 데이터로 평균을 내지 않도록 주의합니다.

1.9 골디락스 규칙

지표는 적어야 합니다. 대시보드에 마흔 개의 그래프가 있는 팀은 대개 아무것도 보지 않습니다. 목표 하나, 가드레일 서넛, 그리고 필요할 때 펼쳐 보는 진단 묶음. 이 정도면 대부분의 작업에 충분합니다.

그리고 지표 목록은 작업마다 다시 정합니다. 캐시 계층을 바꾸는 작업과 모델을 양자화하는 작업은 같은 지표로 판정할 수 없습니다. 조직 공통 대시보드는 상태를 감시하는 용도이고 작업의 성패를 가르는 지표 카드는

그 작업을 시작할 때 새로 씁니다. 이 두 가지를 구분하지 않으면, 공통 대시보드에 없는 개선은 아무도 인정해주지 않는 이상한 상태가 됩니다.

다음 장에서는 고른 지표를 실제로 재는 순간의 문제로 넘어갑니다. 같은 코드, 같은 장비, 같은 명령인데도 숫자가 달라지는 이유들입니다.

2 숫자를 흔드는 것들: 워밍업, 베이스라인, 분산, 공유 자원

같은 명령을 두 번 실행했는데 결과가 다릅니다. 이 흔한 경험을 대하는 태도가 측정 문화의 수준을 가릅니다. 대개는 두 값 중 마음에 드는 쪽을 고르거나, 평균을 내고 넘어갑니다. 규율이 있는 팀은 다르게 반응합니다. 왜 달라졌는지를 먼저 밝히고, 그 원인을 측정 절차 안으로 흡수합니다.

숫자를 흔드는 요인은 무한하지 않습니다. 실무에서 마주치는 대부분은 네 갈래로 정리됩니다. 시스템이 아직 정상 상태가 아니거나, 기준선을 잘못된 시점에 쟀거나, 표본이 하나뿐이거나, 우리 것이 아닌 자원의 소비를 우리 것으로 세고 있는 경우입니다.

2.1 워밍업은 예의가 아니라 정의의 문제다

첫 실행이 느린 이유는 많습니다. 런타임이 아직 기계어로 컴파일하지 않았고, 페이지 캐시가 비어 있고, 커넥션 풀이 비어 있고, 커널이 아직 필요한 페이지를 매핑하지 않았습니다. GPU 쪽으로 가면 커널 자동 튜닝과 메모리 할당자 예열이 더해지고, 프로세서는 부하가 들어오고 나서야 클럭을 끌어올립니다.

여기서 흔한 오해는 워밍업을 "정확도를 높이기 위한 배려"로 이해하는 것입니다. 그렇지 않습니다. 워밍업은 우리가 측정하려는 상태가 무엇인지 정의하는 행위입니다. 콜드 스타트 지연이 궁금한 사람에게 워밍업된 숫자는 틀린 답이고, 정상 운영 지연이 궁금한 사람에게 콜드 숫자는 틀린 답입니다. 그래서 측정 스펙에는 "워밍업 몇 회"가 아니라 "우리가 재려는 상태는 무엇인가"가 먼저 적혀야 합니다.

정상 상태에 도달했는지는 눈으로 판단하지 말고 수치로 판정합니다.

실용적인 방법은 이렇습니다. 측정 구간을 여러 개의 창으로 쪼갠 뒤, 최근 세 창의 중앙값이 서로 몇 퍼센트 안에 들어오면 정상 상태로 봅니다. 이 조건을 만족하기 전의 데이터는 버립니다. 이렇게 하면 워밍업 횟수를 손으로 정하지 않아도 되고, 환경이 바뀌어도 절차가 그대로 작동합니다.

2.2 베이스라인을 언제 쟀는지가 결과를 몇 배로 바꾼다

차이를 재려면 기준선이 필요합니다. 그런데 기준선을 언제 재는지는 스펙에 잘 적히지 않습니다. 여기서 조용한 사고가 납니다.

우리 팀이 겪은 사례를 하나 옮겨봅니다. 같은 모델, 같은 노드, 같은 코드로 GPU 전력을 재는데 순수 증가분이 어떤 날은 61와트, 어떤 날은 8와트 근처로 나왔습니다. 여덟 배 차이입니다. 코드는 한 줄도 다르지 않았고, 다른 것은 기준선을 재는 시점 하나뿐이었습니다. 모델을 올리기 전 차가운 상태에서 기준선을 재면 아이들 값이 낮게 잡히고, 방금 작업을 끝낸 뜨거운 상태에서 재면 아이들 값이 높게 잡힙니다. 절대값은 두 경우 모두 비슷했는데, 기준선을 뺀 순증가분만 극단적으로 갈렸습니다.

이 사고에서 얻은 교훈은 세 가지입니다.

기준선을 재는 시점을 스펙에 못 박고 결과에 함께 기록합니다. 유휴 상태로 몇 초를 쉰 뒤에 잰다는 조건을 코드가 강제하게 만듭니다. 절대값도 함께 보고합니다. 절대값은 기준선 선택에 둔감하고, 자원을 통째로 빌려 쓰는 상황에서는 어차피 유휴 소비까지 우리가 부담하기 때문입니다. 순증가분은 "요청 하나를 더 처리할 때의 한계 비용"이라는 다른 질문의 답이므로, 둘 다 내고 각각 이름을 붙입니다. 앞선 작업의 잔열이 다음 측정을 오염시킨다는 점도 놓치기 쉽습니다. 대상이 바뀌면 기준선을 다시 재야 하고, 배치로 열 가지를 연달아 재면서 기준선을 처음에 한 번만 잡으면 뒤로 갈수록 결과가 무너집니다.

2.3 한 번 잰 값은 측정이 아니다

단일 실행에서 나온 숫자는 표본 하나입니다. 그 값으로 두 구성을 비교하는 것은 동전을 한 번씩 던져 어느 동전이 더 앞면이 잘 나오는지 판단하는 일과 같습니다.

최소한의 규율은 이렇습니다. 같은 조건을 최소 다섯 번 반복하고, 대푯값으로 중앙값을 쓰고, 흔들림의 크기로 변동계수를 함께 봅니다. 변동계수는 표준편차를 평균으로 나눈 값이며, 이 값이 크면 그 환경에서 나온 비교는 신뢰할 수 없습니다. 경험적으로 변동계수가 5퍼센트를 넘으면 측정 환경을 먼저 고치고, 그 전에는 어떤 결론도 내지 않는 편이 낫습니다. 이 임계값 자체도 팀마다 환경마다 달라야 하므로, 한 번은 실제로 재보고 정합니다.

차이를 볼 때는 두 가지를 구분해야 합니다. 하나는 차이가 있는지이고, 다른 하나는 그 차이가 의미 있는 크기인지입니다. 반복을 충분히 늘리면 0.3퍼센트 차이도 통계적으로는 유의해집니다. 그런데 0.3퍼센트를 위해 아키텍처를 바꾸지는 않습니다. 그래서 측정을 시작하기 전에 “이만큼 이상 좋아져야 채택한다”는 최소 유의미 차이를 먼저 적어둡니다. 결과를 보고 나서 기준을 정하면 사람은 언제나 자기가 원하는 쪽으로 기준을 옮깁니다.

2.4 공유 자원에서 나온 숫자는 우리 것이 아니다

측정 대상이 혼자 쓰는 장비 위에 있는 경우는 드뭅니다. 클라우드 인스턴스는 물리 서버를 이웃과 나눠 쓰고, 사내 클러스터의 노드에는 다른 팀의 작업이 함께 돌고, 하나의 프로세서 패키지 안에서 수십 개의 코어가 여러 프로세스를 처리합니다.

이 상황에서 시스템 전체를 대상으로 하는 계측기를 쓰면 남의 소비가 우리 숫자에 섞입니다. 프로세서 패키지 단위로 읽는 전력 값이 대표적입니다. 코어가 256개인 노드에서 우리 프로세스가 32코어어치를 썼다면, 패키지

전력에서 우리 몫은 대략 8분의 1입니다. 이 사실을 무시하고 패키지 전력을 그대로 인용하면 우리 작업의 에너지를 여덟 배 부풀려 보고하게 됩니다.

여기서 권하는 태도는 숫자를 지우는 것이 아니라 신뢰도를 함께 내는 것입니다. 프로세스가 실제로 소비한 코어 시간, 벽시계 시간, 노드 전체 코어 수를 함께 기록해 귀속률을 계산하고, 귀속률이 낮으면 그 지표를 참고값으로 강등합니다. 반대로 가속기처럼 프로세스가 독점하는 자원은 귀속이 명확하므로 주 지표로 씁니다. 어떤 값이 얼마나 우리 것인지를 명시하는 것만으로도 보고의 정직함이 크게 올라갑니다.

이웃 소음을 완전히 없앨 수는 없지만 줄일 방법은 있습니다. 측정 전에 노드의 부하와 사용 중인 자원을 읽어 임계 이상이면 측정을 거부하도록 만들면 됩니다. 오염된 조건에서 나온 예쁜 숫자보다, 측정을 건너뛰었다는 기록이 훨씬 쓸모 있습니다.

2.5 순서와 관측자

두 구성을 비교할 때 언제나 A를 먼저 재고 B를 나중에 재면, 잔열과 캐시 상태와 시스템 드리프트가 전부 B에게 유리하거나 불리하게 작용합니다. 순서를 번갈아 바꿔가며 재는 것만으로 이 편향의 상당 부분이 상쇄됩니다. 하루 중 시간대에 따라 클러스터 혼잡도가 다른 환경이라면, 두 구성을 짧은 간격으로 교대 실행하는 편이 각각을 몰아서 재는 것보다 훨씬 공정합니다.

계측 자체의 비용도 잊기 쉽습니다. 상세 프로파일러를 켜면 대상이 느려지고, 우리는 프로파일러가 켜진 시스템의 성능을 재게 됩니다. 계측을 켠 상태와 끈 상태의 기준 성능을 한 번 비교해두면, 프로파일 결과를 어느 정도까지 믿을지 판단할 수 있습니다.

2.6 데이터 상태와 무작위성

환경만 통제한다고 끝나지 않습니다. 측정 대상이 다루는 데이터의 상태도 결과를 크게 흔듭니다.

데이터베이스 성능을 재면서 테이블에 행이 만 개일 때와 천만 개일 때를 같은 조건이라 부를 수는 없습니다. 인덱스가 메모리에 다 올라가느냐 아니냐로 성능 특성이 통째로 바뀌기 때문입니다. 그런데 반복 실행을 하다 보면 데이터가 조금씩 늘어나거나, 앞선 실행이 남긴 레코드가 다음 실행의 조건을 바꿉니다. 그래서 반복마다 데이터 상태를 같은 지점으로 되돌리는 절차가 필요합니다. 스냅샷에서 복원하거나, 실행 후 정리하거나, 최소한 데이터 규모를 결과에 기록해야 합니다.

무작위성도 마찬가지입니다. 부하 생성기가 요청을 무작위로 고르고, 샘플링이 무작위로 일어나고, 모델 추론이 확률적으로 동작합니다. 이 무작위성을 없앨 필요는 없지만 재현할 수는 있어야 합니다. 난수 시드를 고정하고 결과에 함께 남기면, 이상한 값이 나왔을 때 그 실행만 정확히 다시 만들어 원인을 볼 수 있습니다. 반대로 시드를 매번 다르게 두고 여러 번 반복하는 것이 목적인 경우도 있습니다. 중요한 것은 어느 쪽인지 의식적으로 정하고 기록하는 일입니다.

2.7 측정 대상을 격리하는 값싼 방법들

큰 장비를 새로 사지 않고도 잡음을 줄이는 방법이 여럿 있습니다.

프로세스를 특정 코어에 고정하면 스케줄러가 코어 사이를 옮겨다니며 만드는 캐시 무효화가 줄어듭니다. 전력 관리 정책을 성능 우선으로 고정하면 클럭이 오르내리며 생기는 변동이 줄어듭니다. 측정 시간대를 배치 작업이 없는 창으로 옮기는 것만으로 변동계수가 절반이 되는 경우도 흔합니다. 네트워크가 개입하는 측정이라면 같은 랙 안으로 클라이언트를 옮겨 경로를 단순하게 만듭니다.

이 조치들에는 대가가 따릅니다. 격리를 강하게 할수록 측정 환경이 실제 운영 환경에서 멀어집니다. 코어를 고정하고 전력 정책을 바꾼 상태에서 잰 값은 재현성은 높지만 프로덕션의 현실은 아닙니다. 그래서 목적에 따라 나눠 씁니다. 두 구성을 비교하는 실험은 잡음이 적은 격리 환경에서 하고, 실제 용량을 판단하는 측정은 운영과 최대한 같은 조건에서 합니다. 두 종류를 섞어 인용하지 않는 것이 핵심입니다.

2.8 재현할 수 없다면 아직 측정이 아니다

이 장의 모든 항목은 하나의 문장으로 압축됩니다. 다른 사람이 같은 절차로 같은 범위의 값을 얻을 수 없다면, 우리가 가진 것은 측정이 아니라 목격담입니다.

재현성을 확인하는 가장 값싼 방법은 동료에게 같은 명령을 돌려보게 하는 것입니다. 값이 크게 다르면 그 차이 자체가 우리가 통제하지 못한 변수를 알려줍니다. 머신이 다르거나, 환경 변수가 다르거나, 캐시 상태가 다르거나, 무엇인가 스펙에 안 적힌 것이 있다는 뜻입니다. 이 확인을 한 번 거치는 것만으로 발표 자리에서 무너지는 숫자를 대부분 걸러낼 수 있습니다.

2.9 시간이라는 변수

마지막으로 자주 잊히는 변수가 시간입니다. 같은 클러스터도 오전과 심야의 혼잡도가 다르고, 클라우드 인스턴스는 같은 이름이어도 배치되는 물리 서버 세대가 다를 수 있습니다. 며칠에 걸쳐 조금씩 모은 결과를 한 표에 나란히 놓으면, 그 표에는 우리가 의도하지 않은 시간 축이 숨어 들어갑니다.

해법은 단순합니다. 비교할 값들은 가능한 한 같은 시간 창 안에서 모으고, 그럴 수 없다면 각 값의 측정 시각을 표에 함께 적습니다. 시각이 적혀 있으면 나중에 이상한 값을 발견했을 때 그날 무슨 일이 있었는지 추적할

수 있습니다.

2.10 측정 직전 체크리스트

- 우리가 재려는 상태가 콜드인지 정상 운영인지 문장으로 적었는가
- 정상 상태 도달을 수치 조건으로 판정하는가
- 기준선을 언제 재는지 스펙에 있고 결과에 기록되는가
- 반복 횟수가 다섯 이상이고 변동계수를 함께 보는가
- 최소 유의미 차이를 결과를 보기 전에 정했는가
- 이 자원이 공유 자원인지, 우리 몫의 비율을 아는가
- 비교군의 실행 순서를 교대했는가

이 일곱 줄을 통과하지 못한 숫자는 회의에 가져가지 않는 편이 낫습니다. 잘못된 숫자는 없는 숫자보다 비싸기 때문입니다.

3 벤치마크가 거짓말하는 여덟 가지 방식과 코드로 막는 법

벤치마크가 거짓말을 할 때, 대부분은 누가 속이려 해서가 아닙니다. 측정을 설계한 사람은 대개 성실했고, 결과를 발표한 사람도 자기 숫자를 믿었습니다. 문제는 측정이라는 작업에 사람의 기대가 스며들 통로가 아주 많다는 데 있습니다. 그 통로를 하나씩 이름 붙여 막는 것이 이 장의 목적입니다.

여기서 다루는 여덟 가지는 특정 도구나 분야에 국한되지 않습니다. 웹 서비스든 데이터베이스든 모델 추론이든, 성능 숫자가 오가는 곳에서는 같은 방식으로 반복됩니다. 각 항목이 어떻게 생겼는지, 왜 자연스럽게 발생하는지, 산문 규칙 대신 무엇을 코드로 막을지를 함께 적습니다. 규칙을 문서에 적어두는 것으로는 부족합니다. 사람이 지켜야 하는 규칙은 바쁜 날 가장 먼저 무너지고, 하필 바쁜 날의 측정이 가장 중요한 의사결정에 쓰입니다.

3.1 첫째, 워크로드가 현실이 아니다

가장 흔하고 가장 큰 왜곡입니다. 합성 부하는 만들기 쉽고 재현하기 쉬워서 매력적이지만, 현실의 분포를 닮지 않으면 결과 전체가 무의미해집니다.

전형적인 형태는 이렇습니다. 같은 요청을 반복해서 보내면 캐시 적중률이 100퍼센트에 수렴하고, 시스템은 실제로는 존재하지 않는 성능을 보여줍니다. 요청 크기를 전부 동일하게 맞추면 큰 요청이 만드는 메모리 압박과 꼬리 지연이 사라집니다. 사용자 도착 시간을 완전히 균등하게 만들면 실제 트래픽의 몰림 현상이 지워지고, 큐가 쌓이는 구간을 영영 보지 못합니다.

막는 방법은 프로덕션 로그에서 분포를 뽑아 쓰는 것입니다. 요청 종류의 비율, 크기 분포, 도착 간격의 분포를 실제 데이터에서 추정해 부하 생성기의 입력으로 씁니다. 완벽하게 재현할 필요는 없고, 최소한 분포의 모양이 닮으면 됩니다. 벤치마크 스펙에는 "이 워크로드는 몇 월 며칠의 어떤 로그에서 유도했다"는 한 줄을 남깁니다. 이 한 줄이 없으면 반년 뒤에 그 벤치마크가 무엇을 흉내 낸 것인지 아무도 모릅니다.

3.2 둘째, 비교군을 공평하게 대하지 않았다

새 구성을 평가할 때 우리는 새 쪽을 며칠 동안 튜닝합니다. 파라미터를 만지고, 배치 크기를 바꾸고, 컴파일 옵션을 조정합니다. 기존 구성은 기본값 그대로 둡니다. 두 숫자를 나란히 놓습니다.

이런 비대칭은 악의 없이 발생합니다. 새것에 시간을 쓰는 것은 자연스럽고, 기존 것은 이미 안다고 생각하기 때문입니다. 하지만 결과는 명백한 편향입니다.

막는 방법은 튜닝 예산을 명시적으로 정하는 것입니다. 양쪽에 같은 시간 또는 같은 횟수의 탐색을 배정하고, 각 구성에 어떤 설정을 썼는지 결과와 함께 저장합니다. 튜닝 시간을 맞추기 어렵다면 최소한 "새 구성은 여덟 시간, 기존 구성은 기본값"이라고 결과에 적습니다. 불공정을 없애지 못할 때는 드러내는 것이 차선입니다.

3.3 셋째, 측정 구간에서 대기 시간이 빠졌다

성능 측정에서 가장 교묘한 함정입니다. 부하 생성기가 요청을 보내고 응답을 받을 때까지의 시간을 재는데, 요청을 보낼 스레드가 앞의 응답을 기다리느라 묶여 있으면 어떻게 될까요. 시스템이 느려질수록 부하 생성기는 요청을 덜 보내게 되고, 실제로는 늦어졌어야 할 요청들이 아예 발생하지 않은 것으로 처리됩니다. 시스템이 막히는 그 순간에 측정값은

오히려 안정적으로 보입니다.

같은 문제가 서버 내부 계측에서도 생깁니다. 처리 함수에 들어온 시점부터 시간을 재면, 스레드 풀에서 대기한 시간이 통째로 빠집니다. 포화 상태에서 이 값은 실제 사용자 경험과 완전히 분리됩니다.

막는 방법은 두 가지입니다. 부하 생성기는 응답을 기다리지 않고 정해진 속도로 요청을 발사하는 모드를 쓰고, 예정된 발사 시각과 실제 응답 시각의 차이로 지연을 계산합니다. 서버 쪽에서는 요청이 프로세스 경계에 도착한 시각을 첫 지점으로 삼고, 큐 대기 시간을 별도 지표로 항상 노출합니다. 큐 대기 시간이 진단 지표 목록에 늘 들어 있어야 하는 이유가 이것입니다.

3.4 넷째, 분모가 조용히 바뀌었다

비율 지표의 분모는 소리 없이 이동합니다. 초당 처리량을 비교하는데 한쪽은 재시도를 성공으로 세고 다른 쪽은 세지 않습니다. 토큰당 비용을 비교하는데 토크나이저가 다릅니다. 요청당 지연을 비교하는데 한쪽의 요청 하나가 다른 쪽의 요청 세 개에 해당합니다.

이 실패는 두 팀이 각자 자기 코드에서 지표를 계산할 때 거의 확실하게 발생합니다.

막는 방법은 계산을 한 곳으로 모으는 것입니다. 원시 사건 기록만 각자 남기고, 지표 계산은 공통 함수 하나가 담당하게 합니다. 그 함수는 분모의 정의를 인자가 아니라 코드로 고정합니다. 비교 결과를 낼 때 분모 자체의 값도 함께 출력하게 하면, 분모가 다르다는 사실이 표에 그대로 드러나서 오해가 생기기 전에 잡힙니다.

3.5 다섯째, 좋은 실행만 골랐다

열 번 돌리고 가장 좋은 값을 보고하는 일은 생각보다 흔합니다. 대놓고 고르지 않더라도, 이상하게 나쁜 실행을 "뭔가 잘못됐겠지"라며 다시 돌리는 습관이 같은 결과를 만듭니다. 나쁜 값만 재실행하고 좋은 값은 그대로 두면, 그 자체가 편향입니다.

막는 방법은 실행을 버리는 기준을 미리 정하고 코드가 판정하게 하는 것입니다. 노드 부하가 임계를 넘었거나, 오류율이 기준을 넘었거나, 워밍업 판정을 통과하지 못한 경우처럼 객관적 조건만 폐기 사유로 인정합니다. 폐기한 실행의 개수와 사유는 결과에 남깁니다. 폐기율이 높다는 사실 자체가 중요한 정보입니다.

3.6 여섯째, 통계 없이 하나를 비교했다

앞 장에서 다룬 문제지만 벤치마크 문맥에서 다시 짚어볼 만합니다. 표 안에 숫자 하나만 있고 반복 횟수도 흔들림도 없는 비교표는 읽는 사람이 검증할 방법이 없습니다.

막는 방법은 보고 포맷을 강제하는 것입니다. 결과 표를 만드는 함수가 반복 횟수와 중앙값과 사분위 범위를 함께 넣지 않으면 아예 표를 만들지 못하게 합니다. 포맷을 사람의 성실함에 맡기지 않고 코드가 소유하면, 바쁜 날에도 형식이 무너지지 않습니다.

3.7 일곱째, 기준선이 오염됐다

기준선은 한번 정하면 오래 쓰기 때문에 조용히 낡습니다. 세 달 전 커밋으로 잰 기준선을 계속 인용하는 동안, 그 사이 라이브러리가 올라갔고 커널이 패치됐고 클러스터의 노드 구성이 바뀝니다. 그 상태에서 나온 개선 폭은 우리 코드의 기여와 환경 변화가 섞인 값입니다.

막는 방법은 기준선을 저장하지 말고 매번 함께 재는 것입니다. 비교가 필요할 때마다 두 구성을 같은 세션 안에서 교대로 실행하면, 환경 변화가 양쪽에 동일하게 작용해 상쇄됩니다. 저장된 기준선을 꼭 써야 한다면 유효 기간을 두고, 기간이 지나면 코드가 경고를 띄우게 합니다.

3.8 여덟째, 벤치마크 자체가 목표가 됐다

마지막 항목은 성격이 다릅니다. 앞의 일곱 개가 측정 기술의 문제라면, 이것은 조직의 문제입니다.

어떤 지표를 평가 기준으로 삼는 순간, 사람과 시스템은 그 지표를 올리는 방향으로 최적화됩니다. 벤치마크 워크로드에만 유리한 캐시 정책이 들어가고, 평가셋에 맞춘 튜닝이 쌓이고, 시간이 지나면 벤치마크 점수는 올라가는데 실제 사용자 경험은 그대로이거나 나빠집니다. 지표가 목표가 되면 좋은 지표이기를 그만둡니다.

이 문제는 완전히 없앨 수 없고 관리할 수 있을 뿐입니다. 실무적인 방어는 세 가지입니다. 평가용 워크로드의 일부를 공개하지 않고 보관해 두었다가 주기적으로 교체합니다. 목표 지표와 별개로, 사용자 행동에서 나오는 지표를 함께 봅니다. 마지막으로, 벤치마크 점수가 크게 올랐는데 사용자 지표가 움직이지 않을 때, 그것을 성공이 아니라 경고 신호로 읽습니다.

3.9 규칙을 게이트로 옮기기

여덟 가지를 모두 기억하는 것은 불가능하고, 기억해도 매번 적용되지 않습니다. 그래서 이 장의 결론은 목록이 아니라 구조입니다. 측정 코드에 다음 네 개의 게이트를 심으면 위 항목의 대부분이 자동으로 걸립니다.

게이트	막는 실패
환경 검사	오염, 체리피킹

게이트	막는 실패
정상 상태 판정	워밍업, 기준선
반복과 분산 검사	단일 실행
포맷 강제	분모, 보고 누락

게이트는 통과 여부를 종료 코드로 돌려주어야 합니다. 사람이 로그를 읽고 판단하는 방식은 결국 지켜지지 않습니다. 게이트가 한 번도 아무것도 막지 않는다면, 그것은 우리가 완벽하다는 뜻이 아니라 게이트가 고장 났다는 뜻일 가능성이 높습니다. 새 게이트를 만들면 일부러 실패시켜 보고, 실패 메시지가 원인을 짚는지 확인하고 나서 배포합니다.

3.10 게이트를 만들 때의 실무 요령

게이트를 실제로 도입해보면 몇 가지가 반복해서 문제가 됩니다.

첫째, 게이트가 너무 엄격하면 팀이 우회합니다. 통과하지 못하는 날이 계속되면 사람들은 게이트를 끄는 방법부터 찾습니다. 임계값은 과거에 정상적으로 통과했던 실측 이력을 근거로 잡고, 새 게이트는 처음 몇 주 동안 경고만 내도록 두는 편이 정착률이 높습니다.

둘째, 게이트가 실패했을 때 이유를 알 수 없으면 없느니만 못합니다. “검증 실패”라는 메시지만 남기고 종료하는 게이트는 두 번 헛돌게 만듭니다. 어떤 조건을 기대했고 실제 값이 얼마였는지를 한 줄로 출력해야 합니다.

셋째, 게이트는 측정 파이프라인의 앞쪽에 놓을수록 이득이 큽니다. 비싼 단계를 다 돌리고 나서 환경 오염을 발견하면 그 시간이 통째로 낭비됩니다. 환경 검사처럼 몇 초면 끝나는 검증은 항상 맨 앞에 둡니다.

3.11 남의 벤치마크를 읽는 법

우리가 재는 쪽이 아니라 읽는 쪽일 때도 같은 목록이 쓸모 있습니다. 벤더 자료나 공개 비교표를 볼 때 다음 질문을 순서대로 던지면 대부분의 과장이 걸러집니다.

워크로드가 무엇이고 어디서 유도했는가. 비교 대상은 어떤 설정으로 돌렸는가. 반복 횟수와 흔들림이 적혀 있는가. 지연을 어느 지점에서 쟀는가. 비율 지표의 분모가 명시되어 있는가. 이 결과를 재현할 명령이나 스크립트가 공개되어 있는가.

여섯 개 중 서넛에 답이 없다면, 그 표는 성능 정보가 아니라 마케팅 자료로 읽는 편이 맞습니다. 그렇다고 무조건 무시할 필요는 없습니다. 방향성 참고로 쓰되, 우리 워크로드로 직접 재보기 전에는 결정의 근거로 삼지 않으면 됩니다.

4 정직하게 보고하고 결정에 쓰기

여기까지 왔다면 지표를 골랐고, 환경을 통제했고, 흔한 왜곡을 게이트로 막았습니다. 이제 남은 일은 그 숫자를 사람에게 전달하고 결정으로 바꾸는 것입니다. 많은 팀이 여기서 앞의 노력을 잃습니다. 정교하게 잰 값이 슬라이드 한 장에 옮겨지면서 조건이 떨어져 나가고, 조건 없는 숫자는 몇 주 뒤에 전혀 다른 맥락에서 인용됩니다.

측정의 마지막 단계는 통계가 아니라 서술입니다. 숫자를 어떻게 적느냐가 그 숫자의 수명을 결정합니다.

4.1 값과 불확실성과 조건은 한 묶음이다

보고의 최소 단위는 숫자 하나가 아니라 세 가지의 묶음입니다. 얼마인지, 얼마나 흔들리는지, 어떤 조건에서였는지입니다. 셋 중 하나라도 빠지면 그 문장은 재사용될 수 없습니다.

"응답 지연 95백분위 180밀리초"는 절반짜리 문장입니다. "동시 사용자 300명, 요청 크기 중앙값 4킬로바이트 조건에서 응답 지연 95백분위 180밀리초, 다섯 회 반복의 사분위 범위 174에서 191"이 완성된 문장입니다. 길어 보이지만 이 문장은 반년 뒤에도 그대로 쓸 수 있고, 앞의 문장은 다음 주면 이미 무슨 뜻인지 아무도 모릅니다.

효율 지표라면 여기에 한 가지가 더 붙습니다. 절대값과 순증가분을 함께 내고 각각 이름을 붙이는 일입니다. 서비스 원가를 이야기하는 자리에서는 유휴 소비까지 포함한 절대값이 맞고, 요청 하나를 더 받을 때의 한계 비용을 이야기하는 자리에서는 순증가분이 맞습니다. 어느 쪽이 맞느냐는 질문에 달려 있으므로, 둘 다 내고 라벨을 붙이는 편이 언제나 안전합니다.

4.2 측정 원장과 재현 명령

숫자에 조건을 붙였더라도, 그 조건이 산문으로만 존재하면 언젠가 부정확해집니다. 그래서 측정할 때마다 기계가 읽을 수 있는 원장을 남깁니다. 원장에는 최소한 다음이 들어갑니다.

- 실행 시각과 실행자
- 코드 커밋 해시와 설정 파일의 해시
- 하드웨어와 드라이버 버전, 노드 이름
- 워크로드 정의와 그 출처
- 반복 횟수, 폐기한 실행 수와 사유
- 기준선을 잰 시점과 그때의 유휴값
- 원시 결과 파일의 경로

이 목록을 다 채우는 데 걸리는 시간은 몇 분이지만, 채우지 않으면 나중에 그 측정을 다시 하기 위해 며칠이 듭니다.

원장보다 더 강력한 것은 재현 명령입니다. 결과 파일 맨 위에 이 결과를 다시 만들어내는 명령 한 줄이 적혀 있으면, 의심하는 사람이 직접 확인할 수 있습니다. 반박 가능한 숫자만이 신뢰를 얻습니다. 재현할 수 없는 성능 주장은 아무리 정교해도 결국 일화로 남습니다.

4.3 회귀 게이트는 노이즈를 먼저 재고 건다

성능을 지속적으로 지키려면 자동화된 회귀 검사가 필요합니다. 다만 순서를 지켜야 합니다. 많은 팀이 임계값을 먼저 정하고 게이트를 걸었다가, 매일 울리는 경보에 지쳐 게이트를 꺼버립니다.

올바른 순서는 이렇습니다. 먼저 아무것도 바꾸지 않은 상태에서 같은 측정을 여러 번 반복해 그 환경의 자연스러운 흔들림 폭을 잽니다. 그 폭이 3퍼센트라면, 3퍼센트짜리 회귀를 잡겠다는 게이트는 절반이 거짓 경보가

됩니다. 임계값은 노이즈 폭보다 확실히 위에 놓고, 그보다 작은 변화를 잡고 싶다면 임계값을 낮추기 전에 노이즈를 먼저 줄여야 합니다.

작은 변화를 잡는 다른 방법도 있습니다. 한 번의 실행으로 판정하지 않고 여러 실행의 추세로 판정하는 것입니다. 연속된 세 번의 측정이 모두 같은 방향으로 기울면 경보를 울리는 방식은, 단일 임계값보다 거짓 경보가 적으면서도 완만한 성능 저하를 잘 잡아냅니다. 하루하루는 1퍼센트씩 나빠지는데 아무도 눈치채지 못하다가 분기 말에 20퍼센트가 빠져 있는 상황이 이렇게 막힙니다.

경보가 울렸을 때 무엇을 할지도 미리 정해둡니다. 담당자가 없는 경보는 두 번째부터 소음이 됩니다. 회귀가 확인되면 되돌릴지, 예외로 승인할지, 다음 스프린트로 넘길지의 선택지를 정해두고, 승인한 예외는 기한과 함께 기록합니다.

4.4 숫자를 결정으로 바꾸는 마지막 단계

성능 숫자는 그 자체로 결정이 되지 않습니다. 결정은 언제나 비교입니다. 그래서 마지막에 한 번의 환산이 필요합니다.

지연 20퍼센트 개선이 무엇을 의미하는지 답하려면, 그 개선이 사용자 이탈률이나 전환율이나 서버 대수로 환산되어야 합니다. 처리량 개선은 같은 트래픽을 몇 대로 감당할 수 있는지로 바꿔야 예산 회의에서 쓰입니다. 에너지 효율 개선은 시간당 비용과 냉각 여유로 바꿔야 데이터센터 계획에 들어갑니다. 환산 계수가 정확할 필요는 없고, 자릿수만 맞아도 대화가 완전히 달라집니다. 다만 추정한 값에는 반드시 추정이라고 적습니다.

환산을 해보면 종종 예상 밖의 결론이 나옵니다. 몇 주를 들여 얻은 8퍼센트 개선이 서버 반 대 값어치이고, 아무도 손대지 않던 설정 하나가 서버 세 대 값어치인 경우가 실제로 흔합니다. 측정의 목적은 우리가 잘했다는 증거를 모으는 것이 아니라, 다음에 어디에 시간을 쓸지 정하는 데 있습니다.

4.5 표와 그림에 관한 최소 규칙

보고서에서 숫자가 왜곡되는 마지막 지점은 시각화입니다. 악의 없이도 그림은 쉽게 거짓말합니다.

세로축을 0에서 시작하지 않으면 3퍼센트 차이가 세 배 차이처럼 보입니다. 막대그래프에서는 축을 자르지 않는 편이 안전하고, 미세한 차이를 보여줘야 한다면 막대 대신 점과 오차 막대를 쓰는 편이 정직합니다. 흔들림을 표시하지 않은 막대 두 개를 나란히 놓는 순간, 읽는 사람은 그 차이가 확실하다고 믿게 됩니다.

표를 쓸 때는 열을 늘리고 싶은 유혹을 참아야 합니다. 한 표에 열 개의 지표를 넣으면 아무도 읽지 않고, 읽더라도 무엇이 목표 지표인지 알 수 없습니다. 표 하나에 목표 지표와 가장 중요한 가드레일만 남기고 나머지는 부록으로 보냅니다.

항목	권장
세로축	0에서 시작
흔들림	항상 표기
표 열 수	셋 이하

4.6 언제 측정을 멈추는가

측정에도 수확 체감이 있습니다. 반복을 다섯 번에서 쉰 번으로 늘리면 신뢰 구간은 좁아지지만, 그 좁아진 구간이 결정을 바꾸지 않는다면 그 시간은 낭비입니다.

멈춰야 할 때를 판단하는 질문은 간단합니다. 지금 숫자가 조금 더 정확해지면 우리 선택이 달라지는가. 답이 아니라면 멈춥니다. 두 후보의 신뢰 구간이 겹치지 않고 이미 한쪽이 이겼다면 더 재지 않습니다.

반대로 구간이 크게 겹쳐 판정이 안 되면 반복을 늘리기 전에 환경 잡음부터 줄입니다. 잡음이 큰 환경에서 반복만 늘리는 것은 비싸고 느린 해법입니다.

이 판단을 미리 정해두는 방법도 있습니다. 측정을 시작할 때 최대 반복 횟수와 조기 종료 조건을 함께 적어두면, 결과에 끌려다니며 계속 재는 일이 줄어듭니다.

4.7 독자에 따라 다르게 쓴다

같은 측정 결과라도 읽는 사람에 따라 필요한 것이 다릅니다. 한 벌의 문서로 모두를 만족시키려다 보면 어느 쪽에도 쓸모없는 글이 나옵니다.

엔지니어에게는 조건과 원시 데이터 경로가 필요합니다. 어떤 커밋에서, 어떤 노드에서, 어떤 워크로드로 쟀는지가 있어야 이어서 파고들 수 있습니다. 여기서는 문장을 줄이고 표와 경로를 늘리는 편이 낫습니다.

기술 리더에게는 판정과 트레이드오프가 필요합니다. 목표 지표가 얼마나 움직였고, 가드레일 중 어디가 위태로운지, 채택할 경우 무엇을 포기하는지가 첫 문단에 있어야 합니다. 여기서는 결론을 먼저 쓰고 근거를 뒤에 붙입니다.

예산을 다루는 자리에서는 환산된 값이 필요합니다. 서버 대수, 월 비용, 확보되는 여유 용량 같은 언어로 옮겨야 합니다. 이때 원 지표를 함께 각주로 남겨두면, 나중에 환산 계수를 고칠 때 처음부터 다시 하지 않아도 됩니다.

세 벌을 따로 쓰는 것이 아니라, 하나의 원장 위에 세 개의 요약을 얹는 구조가 실용적입니다. 원장은 하나고 요약은 여럿이라는 원칙을 지키면, 어느 자리에서 인용된 숫자든 같은 근거로 되돌아갈 수 있습니다.

4.8 측정 결과에 유효 기간을 둔다

숫자는 늙습니다. 반년 전 측정이 오늘의 결정에 인용되는 일은 생각보다 흔하고, 그 사이 트래픽 패턴과 데이터 규모와 인프라가 모두 바뀌었을 가능성이 큽니다.

중요한 측정에는 유효 기간을 적어두기를 권합니다. 이 값은 언제까지 유효하다고 보는지, 무엇이 바뀌면 다시 재야 하는지를 한 줄로 남깁니다. 트래픽이 두 배가 되면, 모델이 교체되면, 노드 종류가 바뀌면 재측정이라는 식입니다. 조건을 적어두면 재측정 시점을 사람의 기억이 아니라 사건이 알려주게 됩니다.

4.9 나쁜 숫자를 말할 수 있는 조직

마지막은 기술이 아닌 이야기입니다. 지금까지 다룬 모든 규율은 한 가지 조건 위에서만 작동합니다. 나쁜 결과를 그대로 보고해도 괜찮아야 한다는 조건입니다.

성능 개선을 약속하고 시작한 작업에서 개선이 없었다는 결과가 나왔을 때, 그 결과를 그대로 말할 수 있는 팀만이 신뢰할 수 있는 측정을 갖습니다. 그렇지 않은 팀에서는 게이트가 조용히 완화되고, 워크로드가 유리한 쪽으로 조정되고, 폐기한 실행이 기록되지 않습니다. 어떤 도구도 이것을 막지 못합니다.

실무적으로 권하는 습관이 하나 있습니다. 측정을 시작하기 전에 예상 결과를 적어두는 것입니다. 얼마나 좋아질 것으로 보는지, 어떤 결과가 나오면 이 방향을 접을 것인지를 미리 씁니다. 결과가 예상과 다르면 그 자체가 배움이 되고, 예상을 미리 적었기 때문에 “원래 그럴 줄 알았다”는 사후 합리화가 끼어들 자리가 없어집니다.

측정의 규율은 결국 정직함을 절차로 바꾸는 일입니다. 사람의 성실함에

기대는 대신, 잘못 재기 어렵고 잘못 보고하기 번거로운 구조를 만드는 것입니다. 그 구조가 자리 잡으면 숫자를 둘러싼 논쟁이 줄고, 팀은 무엇이 실제로 효과가 있었는지를 두고 이야기하기 시작합니다. 그 대화가 시작되는 지점에서 성능 작업은 비로소 공학이 됩니다.