무언가 말해주는 시스템

로그를 쓰고, 찾고, 믿는 엔지니어의 규율

ThakiCloud (Hyojung Han)

무언가 말해주는 시스템

로그를 쓰고, 찾고, 믿는 엔지니어의 규율

ThakiCloud (Hyojung Han)

Contents

1 1장. 로그가 유일한 증인인 밤

1.1 새벽 2시의 질문

새벽 2시. 휴대폰이 진동한다. 알림 문장은 짧다. 결제 API 에러율이 5퍼센트를 넘겼다.

전화로 물어볼 사람이 없다. 온콜 동료는 존재하지 않고 벤더의 지원센터는 오전 9시에 문을 연다. 이 시스템을 아는 사람은 오직 나인데, 새벽 2시의 나는 가장 신뢰할 수 없는 증인이다. 기억이 희미하고 피곤하고 확신이 없다. 어제 뭘 바꿨는지 반쯤만 기억한다.

그러면 처음 하는 일이 무엇인가. 터미널을 열고 로그를 찾는 일이다. 그 뒤의 30분은 거의 매일 같은 순서로 간다.

1. 에러율이 언제부터 올라왔나.
2. 어떤 요청이 실패하고 있나.
3. 실패는 내 코드에서 일어나는가, 외부에서 오는가.
4. 어제 내가 건드린 것과 관련이 있나.

이 네 질문의 답이 로그에 남아 있으면 새벽은 견딜 만하다. 답이 남아 있지 않으면 새벽은 추측 게임. 추측 게임에서 이기는 사람은 운 좋은 사람뿐인데, 운은 준비가 되지 않는 한 반복되지 않는다.

이 책은 그 단 하나의 행위를 위해 미리 준비하는 책이다. 로그를 어떻게 쓰느냐, 어떻게 찾느냐, 그리고 어떻게 믿느냐. 세 가지 전부 새벽 2시의 질문을 위해서다.

1.2 물어볼 사람이 없는 개발자

팀에서 일하면 고장이 났을 때 첫 번째 수단은 질문이다. 어젯밤 배포한 거야? 설정을 건드린 사람 있어? 두세 번 물으면 원인이 좁혀진다. 동료는 살아 있는 문서다. 회의록도, 채널 대화도 문서.

1인 개발자에게 살아 있는 문서는 자신뿐이다. 그런데 새벽 2시의 자신은 위험한 출처다. 어젯밤 배포했다는 기억은 있는데, 워커를 재시작했는지는 기억나지 않는다. 설정을 바꾼 순서도, 그 사이에 브라우저 탭에서 뭘 클릭했는지도 모른다. 더 위험한 것은 기억이 사실처럼 느껴진다는 점이다. 확신과 사실은 별개의 것인데, 피곤한 새벽에는 둘을 구분할 에너지가 없다.

그래서 잊지 않는 증인이 필요하다. 그게 로그다.

로그는 코드의 부산물이 아니다. 나중에 시스템에게 물을 질문의 답변을 미리 써두는 행위다. 뭐를 했어. 언제부터 이상해졌어. 이 요청이 결제 벤더까지 닿았어? 지금 시스템에게 물을 수는 없다. 시스템은 이미 지난 일을 겪었고 지난 일은 로그에 남겨진 만큼만 남는다. 남겨지지 않은 일은 일어난 일일지라도 증명이 안 된다. 증명이 안 되는 일은 새벽 2시에는 없는 일과 다르지 않다.

여기서 한 가지를 분명히 하자. 로그는 설계다. print를 지우는 순간 로그가 만들어지는 것이 아니다. 어떤 줄이 어디에, 어떤 형식으로, 어떤 레벨로 남을지를 코드를 쓸 때 정한다. 정하지 않으면 시스템은 아무 말도 하지 않는다. 아무 말도 하지 않는 시스템은 고장 났을 때 침묵으로 응답하고, 침묵은 최악의 증언이다.

이 책을 관통하는 첫 번째 규율은 여기 있다.

> 혼란스러운 새벽의 나를 읽을 사람이라고 생각하고 로그를 쓴다. 나중에 누가 볼까 하는 수준으로는 쓰지 않는다.

읽는 사람이 명확해야 쓸 때의 판단이 선명해진다. 그리고 그 독자는 결국 나 자신이다. 미래의 내가 지금의 나보다 이 로그를 더 자주 읽는다는 사실을 전제로 두자.

1.3 1인이라는 규모의 의미

1인이라는 것은 로그 측면에서 특이한 규모다. 큰 조직에서는 로그 규율이 조직 문제로 풀린다. 누가 읽을지, 어떤 필드를 표준으로 할지, 보존 기간을 어디서 끊을지를 팀끼리 협상한다. 협상에는 시간이 들고 실패하면 타협이 남는다.

1인 개발자에게 이 협상은 사라진다. 읽는 사람은 하나, 정하는 사람도 하나다. 표준을 잡는 비용이 거의 없다. 반대로, 표준이 없으면 누구도 만들어 주지 않는다. 규율은 내가 스스로 결정한 규칙이다. 이 책의 모든 권고는 이 전제 위에 있다. 1인 규모에서 가장 싼 신뢰를 사는 법이다.

규모가 작다는 것은 도구도 적게 쓴다는 뜻이다. 이 책에서 권하는 구조는 파일 한 줄과 grep으로도 성립한다. 클라우드 로그 플랫폼이 없어도 된다. 플랫폼은 3장에서 다루되, 원칙은 플랫폼 없이도 먼저 성립하게 써 둔다.

1.4 로그와 print의 차이

서버를 시작했습니다. 준비 완료.

많은 사람의 첫 로그는 콘솔에 찍는 print다. 개발 중에는 그게 전부다. 터미널을 바라보고 있으면 충분해 보인다. 나쁘지는 않다. 다만 print는 로그가 아니다. print는 콘솔로 가는데, 콘솔에는 치명적인 약점이 세 개 있다. 사라진다는 점, 멈춘다는 점, 그리고 시간이 없다는 점이다.

터미널을 닫으면 화면 내용이 없다. ssh 세션을 두 개 띄워 두고 있었다면, 한 세션에서 지운 줄은 다른 세션에도 없다. 프로세스가 죽으면 print도

죽는다. 죽은 프로세스가 마지막에 뭘 말했는지 부를 수 없다. 새벽의 대다수 사고는 바로 이 지점에서 시작한다. 프로세스가 죽었고 죽기 직전에 말한 것이 콘솔에 있었다. 콘솔은 닫혀 있다.

진짜 로그는 프로세스보다 오래 사는 곳으로 간다. 파일이나 로그 서비스. 프로세스가 새벽 2시에 죽어도, 그 전까지 쓴 줄은 찾을 수 있는 자리에 남아 있어야 한다.

두 번째 차이: 로그에는 시간이 있다. 콘솔 줄에는 신뢰할 시간이 없다. 아까쯤 떴겠지 정도만 추측한다. 로그 줄에는 타임스탬프가 있어서 어떤 일이 먼저였느냐는 질문을 답할 수 있다. 순서를 모르면 인시던트 분석은 추측이 된다. 배포가 에러보다 먼저였는지, 에러가 배포보다 먼저였는지는 전혀 다른 이야기다.

세 번째 차이: 로그는 검색할 수 있다. 100만 줄 가운데서 14일에 데이터베이스로 실패한 요청만 골라낼 수 있다. 같은 문장을 다른 단어로 쓴 줄까지 패턴으로 낚아챌 수 있다. 콘솔에서는 불가능하다. 콘솔은 읽는 곳이지, 찾는 곳이 아니다.

정리하면: 콘솔은 지금 내가 보는 것, 로그는 나중에 내가 찾는 것이다. 새벽 2시에 로그를 믿고 싶다면, 코드는 로그 저장처로 계속 써야 한다. 개발 중에도, 배포 후에도, 같은 저장처로. 저장처를 상황별로 바꾸는 순간, 상황 전환 지점의 로그가 빈틈이 된다.

1.5 하나의 구체적 사례

차이를 추상에서 끝내지 않기 위해, 같은 상황을 두 버전으로 보여준다. 상황: 결제 요청이 15초 뒤 성공했고 사용자는 지연을 불평했다. 원인은 벤더의 일시적 오류와 재시도였다.

로그가 빈약한 시스템에서는 이런 기록이 있다.

[14일 09:12] request ok [14일 09:12] request ok

두 줄. 성공했다 정도만 알 수 있다. 15초가 어디에 갔는지는 기록에 없다. 불평을 조사하려면 코드에서 재시도 로직을 다시 읽고 벤더 대시보드를 보고 기억을 더듬는다. 결론은 나온다. 다만 결론까지 40분이 걸리고, 그 40분은 나 자신에서 나온 것이다.

로그가 규율대로 쓰인 시스템에서는 같은 상황의 기록이 이렇게 보인다.

09:12:31 INFO payment call start provider=stripe request_id=req_9f2k attempt=1 09:12:33 ERROR payment call failed provider=stripe request_id=req_9f2k status=502 latency_ms=3120 attempt=1 09:12:34 INFO retry scheduled request_id=req_9f2k next_in_ms=15000 09:12:47 INFO payment call succeeded provider=stripe request_id=req_9f2k latency_ms=210 attempt=2

네 줄. 15초는 재시도 대기였다. 벤더가 한 번 502를 날렸고 두 번째 시도는 210ms에 성공했다. 사용자에게 해명을 쓸 수도 있고 벤더에 제보할 수도 있다. 질문은 로그에서 나왔다.

이게 이 책이 말하는 전부다. 줄의 수는 네 줄이든 사십 줄이든 상관없다. 관련이 있는 줄이, 관련을 알 수 있는 형태로, 나중에 찾을 수 있는 자리에 남아 있는가가 관점이다.

1.6 로그가 사실이다는 무슨 뜻인가

로그가 사실이다. 이 문구를 구호로 삼으면 안 된다. 분해해야 한다. 로그가 사실로 취급받으려면 다섯 속성이 필요하다.

첫째, 시간. 모든 줄에 신뢰할 수 있는 타임스탬프가 있어야 한다. 서버 시계가 1시간 틀어 있으면, 뭐가 먼저였느냐에 답할 수 없고 로그는 소문이 된다. 시간의 신뢰는 시계에서 오고 시계는 동기화에서 온다. 이 부분은

4장에서 다시 다룬다.

둘째, 위치. 어떤 서비스, 어떤 컴포넌트에서 나온 줄인지 알려야 한다. 두 서비스가 같은 형식으로 쓰면 로그는 뒤섞인 산더미가 된다. 이 줄은 결제 API에서 나왔다는 것을 한 번에 알아볼 수 있어야 한다. 위치가 없는 줄은 증인이 없는 증언이다.

셋째, 상관관계. 하나의 요청이 세 서비스를 지나면, 그 세 줄은 하나의 ID로 이어져야 한다. 이어지지 않으면 요청은 세 개의 단편이 되고 단편은 이야기가 되지 않는다. 상관관계 ID는 로그를 줄의 모음에서 이벤트의 서사로 바꾸는 도구다.

넷째, 부연만 허용. 쓴 줄은 다시 쓰지 않는다. 로그를 고치는 것은 증거를 고치는 것과 같다. 바뀔 수 있는 로그는 신뢰할 수 없고 바꿀 수 있는 로그를 가진 시스템은 믿을 수 없다. 추가는 해도 수정은 하지 않는다는 원칙, append only,가 여기 있다.

다섯째, 독자 적합. 저자만 이해하는 레벨과 메시지로 쓴 로그는 독자에게 사실이 아니다. 새벽의 나는 어떤 필드가 중요했는지 모른다. level=ERROR인데 문장이 요청 처리중 이라면, 이 줄이 왜 에러인지 알 수 없다. 레벨과 문장이 저자의 머리 없이도 읽힐 수 있어야 한다.

지금 내 시스템의 로그를 이 다섯 속성으로 점검해보면, 대부분 두세 개는 빠져 있을 것이다. 위치가 없다, 상관관계가 없다, 시간이 콘솔에만 있다. 빠져 있는 속성만큼 로그는 추측에 가깝다. 나머지 장들은 빠진 것을 하나씩 채우는 순서로 가 있다. 2장은 레벨과 문장, 3장은 위치와 상관관계, 4장은 시간과 불변이다.

1.7 로그가 아니라서 이 책에서 다루지 않는 것

범위를 정하자. 이 책은 로그다. 인접한 도구들을 전부 다루면 책이 두 배가 되고, 규율은 반이 된다.

지표, 즉 metrics은 몇 개냐를 말한다. 에러율, 지연, 처리량. 새벽의 첫 질문인 에러율이 5퍼센트를 넘겼다 는 알림이 바로 지표에서 온 것이다. 지표를 만들지 않아도 된다. 다만 지표가 알려주는 이상에 대해 로그가 설명을 이어받아야 한다.

트레이스는 요청이 어디를 지나갔는지를 말한다. 서비스가 열 개가 넘어가고 각 단계의 지연 분해가 필요해지면 로그보다 트레이스가 정확한 도구다. 1인 규모에서는 상관관계 ID로 이어진 로그가 트레이스의 8할[추정]을 대신한다.

알림은 로그를 울리는 것이다. 알림의 문장은 짧고 짧아야 한다. 자세한 것은 로그가 한다.

이 책의 약속은 이렇다. 로그를 쓰게 하고 찾게 하고 믿게 만든다. 나머지는 다른 도구의 일이다.

1.8 오늘 끝내는 점검

책 읽기 전에 10분만 내서, 지금 내 시스템이 다섯 속성을 몇 개 가졌는지 세어보길 권한다.

1. 타임스탬프가 모든 줄에 있는가. 콘솔에만 있는 것은 세지 않는다.
2. 서비스 이름이 줄에서 보이는가.
3. 하나의 요청을 하나의 ID로 쫓아갈 수 있는가.
4. 쓴 로그를 뒤에서 고칠 수 있는 코드가 있는가. 있으면 그 코드가 위험하다.
5. ERROR 줄을 보면 저자가 아니어도 알 수 있는가.

다섯 개를 세서 적어두자. 4장 끝에서 다시 센다. 숫자가 달라졌으면, 이 책은 내 시스템에 적용된 것이다.

2 2장. 무엇을, 어떤 레벨로 쓸 것인가

2.1 레벨은 장식이 아니라 약속

처음에는 모든 것이 중요하다. 그래서 모든 줄이 INFO다. 서버를 시작했습니다, 요청을 받았습니다, 재시도를 스케줄했습니다, 배치 작업을 마쳤습니다. 다 같은 눈높이. 결과는 하루 10만 줄의 INFO이고 그 가운데 새벽에 필요한 줄이 있는데 찾을 수 없다. 찾지 못한 이유는 줄이 너무 많아서다.

레벨은 독자에게 하는 약속이다. 이 레벨의 줄을 보면 얼마나 급하게 행동해야 하는지. ERROR를 보면 일어나야 하고 WARN을 보면 오늘 중에 확인하고 INFO는 흐름을 파악할 때 읽는다. 이 약속이 지켜지면 레벨은 탐색의 계단이 된다. 약속이 깨지면, ERROR를 봐도 다시 확인해야 하나 하는 마음이 들고, 다시 확인하면 또 ERROR다. 그 순간 독자는 레벨을 믿지 않는다. 레벨을 믿지 않는 시스템은 로그가 없는 것과 다름없다.

1인 개발자는 독자가 한 명뿐이라 약속을 단순하게 정의할 수 있다.

레벨	의미	빈도 기대
DEBUG	개발용 상세	요청당 수십 개
INFO	정상 이벤트	요청당 몇 개
WARN	이상인데 자회복	가끔
ERROR	행동이 필요한 실패	목표는 제로

표의 실질적 가치는 마지막 줄이다. ERROR가 매일 여러 번 나는 시스템은 건강한 시스템이 아니다. ERROR의 의미가 희석된 시스템이다. 이 줄이 또 나왔군, 이 정도면 되는 줄이면, 그 시스템의 ERROR는 약속을 지키지 못하고 있다.

규칙을 하나만 남기자. ERROR는 일어나고 싶은 줄이다. 새벽 2시에 이 줄을 보면 잠에서 깨야 할 만큼 중요한가. 기준에 못 미치면 WARN이다. WARN도 아니라면 INFO다.

레벨과 함께, 문장의 위치도 정하자. 같은 레벨이라도 문장이 다르다. INFO는 일이 정상적으로 일어났음을 말한다. 시작, 성공, 완료. WARN은 이상한 일이 일어났지만 시스템은 계속 돌아감을 말한다. 재시도, 시간 초과 후 성공, 용량 임계 근접. ERROR는 행동이 필요함을 말한다. 실패, 거부, 데이터 정합성 깨짐.

2.2 새벽 2시 테스트

각 로그 줄을 쓸 때 한 번만 물어본다. 새벽 2시에 이 줄만 보면, 행동할 수 있는가.

예를 들어 보자. 결제 벤더 호출이 실패했다.

나쁜 줄:

ERROR 결제 실패

좋은 줄:

ERROR payment call failed provider=stripe status=502 latency_ms=3120 request_id=req_9f2k attempt=2

첫 번째 줄은 뭔가 실패했다는 것만 알려준다. 두 번째 줄은 뭘, 어디에서, 어떻게, 어느 요청이 실패했는지를 알려준다. 첫 번째 줄을 보면 조사를 시작해야 한다. 두 번째 줄을 보면 stripe가 502를 던졌고, 이건 두 번째 시도였다는 사실이다. 시작과 판단 사이의 차이가 이 두 줄 사이의 거리다.

테스트를 통과하지 못하는 줄은 독자에게 빈칸을 요구하는 줄이다. 빈칸은 컨텍스트로 채워지는데, 새벽 2시의 컨텍스트는 최악이다. 코드베이스가

머릿속에 없고 브라우저 탭이 열려 있지 않고 어제 뭘 바꿨는지도 반쯤만 기억난다. 빈칸을 만들지 않는 줄을 쓴다.

2.3 좋은 한 줄의 해부

좋은 줄은 다섯 부분으로 이루어진다.

who: 서비스 이름. 결제 API, 워커, 크론 잡. 줄 하나가 어떤 컴포넌트에서 나왔는지 바로 알 수 있어야 한다. 한 줄에 who가 없으면 그 줄은 공용 도로에 버려진 편지다.

when: 타임스탬프. 개발자가 직접 넣지 말고 로깅 프레임워크가 넣게 한다. 직접 넣으면 형식이 사람마다 다르다. UTC로 저장하고 표시할 때 로컬로 바꾼다.

level: 위에서 정한 약속.

what: msg. 어떤 일이 일어났는지, 구문 한 줄로. 동어로 시작하고 마침표 없이 끝낸다. payment call start, payment call succeeded, retry scheduled. msg는 사람이 읽는 부분이다.

context: 독자가 필요한 필드. request_id, status, latency_ms, attempt, provider. context는 기계가 찾는 부분이다.

순서는 크게 중요하지 않다. 중요한 것은 같은 종류의 이벤트가 같은 구조를 갖는다는 점이다. 재시도가 한 줄에서는 attempt=2이고 다른 줄에서는 retry: 2라면, 검색은 추측이 된다. 이벤트당 하나의 형식, 그리고 그 형식을 영원히 반복한다. 이 습관이 다음 장에서 로그를 검색 가능하게 만든다.

구문의 규칙도 간단하다. msg에는 값이 들어가지 않는다. status=502는 context에 있다. msg는 이벤트의 이름, context는 이벤트의 내용이다. msg에 값을 섞으면 검색의 대상이 사라진다.

2.4 같은 이벤트, 다른 레벨

레벨은 줄의 속성이지, 이벤트의 속성이 아니다. 같은 일이라도 누가 보느냐에 따라 레벨이 달라진다.

벤더가 502를 반환했다. 벤더 쪽에서는 그 것이 ERROR다. 내 결제 API에서 볼 때는? 재시도 한 번으로 성공했다면, 그 502는 결국 성공한 요청의 중간 지점이다. 두 번째 시도에 성공했다,는 INFO 한 줄에 첫 시도의 실패는 context로 들어간다. 다만 재시도가 모두 실패하고 요청이 죽으면, 그때 처음 502는 ERROR로 재평가된다.

결국 규칙은 두 단계다. 이벤트가 일어난 순간에는, 그 순간의 사실대로 레벨을 매긴다. 재시도, 타임아웃, 회피. 그리고 요청이 최종적으로 끝나면, 최종 결과를 한 줄로 쓴다. payment request succeeded, attempts=2. 이 마지막 줄이 없으면, 여러 줄의 중간 상태를 읽어서 결과를 추론해야 한다. 추론은 새벽에 하지 않는다.

반대 경우도 있다. WARN을 쌓아두면 ERROR가 된다. 한 벤더의 502가 1시간에 3번이면 WARN이다. 1시간에 50번이면 그 벤더는 사실상 죽은 것이니, ERROR다. 문턱 값을 정해 두고 문턱을 넘으면 레벨을 올린다. 문턱 값은 시스템의 성격마다 다르다. 수치를 정해서 코드에 넣어라. 새벽에 문턱을 정하는 것은 불가능하다.

2.5 모든 서비스의 첫 줄

서비스는 시작할 때 자기 자신을 기록해야 한다. 첫 줄은 언제나 같은 형태다.

INFO process started service=payment-api version=1.4.2 pid=831
INFO config loaded service=payment-api source=/etc/app/config.yaml
INFO listening service=payment-api port=8080

세 줄이면 충분하다. version이 있어야 배포 후의 로그를 읽을 때 어느 코드에서 나온 줄인지 안다. 두 번 배포를 놓고 이 줄이 1.4.2인지 1.4.1인지를 구분하지 못하면, 인시던트의 절반은 버전을 놓고 추측 게임이 된다.

첫 줄은 또, 살아 있는 증거다. 서비스가 재시작되지 않으면 첫 줄이 없거나, 오래전 첫 줄이 마지막이다. 프로세스가 죽고 다시 살아난 횟수는 첫 줄의 개수로 센다. 크래시 루프를 진단하는 첫수가 바로 이것이다. 같은 서비스의 첫 줄이 5분 사이에 6개면, 그 서비스는 죽고 나서는 루프에 있다.

2.6 절대 쓰지 말아야 할 것

레벨과 무관하게 일부는 영원히 로그에 들어가면 안 된다.

시크릿. API 키, 토큰, 비밀번호. 시크릿을 한 번 실어 보낸 로그는 영원히 오염된다. 로그 저장 서비스로 복제되고 백업으로 복제되고 아카이브로 복제된다. 오염이 퍼진 경로를 역추적할 수 없다. 마지막 4자만, 또는 아예 안 쓰는 쪽으로 간다. 실수는 한 번이면 충분하다.

개인정보의 전체. 결제 카드 번호, 주소, 전화번호. 진단에 필요한 것은 어느 요청이었느냐이지, 어느 카드였느냐가 아니다. 식별자와 길이면 된다. 길은 16자리 카드임을, 식별자는 어느 사용자인지를 말한다. 번호 자체는 없다.

페이로드 전체. 요청 바디가 500KB면 전부 쓰면 스토리지가 부풀고, 줄은 노이즈로 가득 찬다. 크기, 식별자, 그리고 중요한 필드. 나머지는 필요할 때 다시 만들 수 있는 것이라 기록하지 않는다.

2.7 쓰지 말아야 할 것: 안전이 아니라 신호

두 번째 종류는 신호 문제다.

가장 큰 노이즈는 재시도와 폴링이다. 폴링 워커가 1분마다 새 작업이

없다고 쓰면, 하루 1,440줄, 한 달에 4만 3천 줄[추정]이 난다. 이 줄들은 사실이 아니다. 검색할 가치가 없는 반복이다. 정책은 첫 발생만 쓰고 이후는 집계한다.

재시도는 번마다 줄을 쓰지 않는다. 이렇게 쓴다.

09:12:33 ERROR payment call failed provider=stripe status=502 attempt=1 request_id=req_9f2k 09:12:48 WARN payment retries exhausted provider=stripe attempts=3 request_id=req_9f2k last_status=502 09:12:48 ERROR payment request failed request_id=req_9f2k total_ms=21400 attempts=3

첫 줄은 처음 실패다. 두 번째 줄은 재시도 도중의 요약이고 세 번째 줄은 최종 결과다. 중간에 재시도가 몇 번 더 있었는지 일일이 쓰지 않아도 된다. attempts 필드가 대신한다. 반복을 줄마다 쓰면, 반복의 횟수만큼 신호가 희석된다. 횟수는 필드로, 사건은 줄로.

두 번째는 컨텍스트 없는 예외다. 스택 트레이스는 유용하다. 다만 뭘 하다가 터졌는지가 함께 있을 때다. request_id가 없는 스택 트레이스는 이름표 없는 시신이다. 몸은 있는데 누구의 몸인지 모른다.

세 번째는 성공만 쓰는 코드다. 실패만 쓰면, 실패한 요청은 알 수 있어도, 그 요청이 어디까지 갔는지 알 수 없다. 진입과 출구를 쓴다. payment call start와 payment call succeeded. 둘 다 없으면 호출 자체가 일어나지 않았는지, 일어났는데 기록을 잃었는지를 구분할 수 없다.

2.8 예산: 한 요청당 몇 줄이면 좋은가

줄의 개수에도 예산이 필요하다. 예산이 없으면, 로그는 점점 두꺼워지고 두꺼워진 로그는 다시 찾아내기 어렵다.

권하는 예산은 이렇다. 하나의 요청이 정상적으로 끝나면, INFO는 3줄

안팎이다. 시작, 중간 경계(외부 호출이나 결제 같은), 성공. DEBUG는 개발 중에만 켜는 것이라 예산에 넣지 않는다. WARN은 0줄이 목표고 ERROR도 정상 요청에서는 0줄이다.

예산의 역할을 하나만 꼽자면, 해피 패스의 떠든 줄을 DEBUG로 내려 보내는 기준을 주는 것이다. 한 요청이 INFO 20줄을 만든다면, 그 20줄 중에 새벽에 필요한 줄은 몇 줄인가. 대부분 3줄보다 적다. 나머지는 DEBUG다. 예산을 정하고 나면 줄을 줄이는 일이 죄책감 없이 허용된다. 줄이는 것은 신호를 끌어올리는 일이다.

예산은 감각이다. 여러 컴포넌트를 거치는 요청은 당연히 3줄보다 많다. 그럴 때는 각 줄이 제 값을 하는지 확인한다. 줄이 많고 각 줄이 다르다면, 솔직한 기록이다. 줄이 많고 내용이 겹겹이라면, 겹치는 쪽을 DEBUG로 내린다.

2.9 오늘 점검하는 체크리스트

프로젝트의 로깅 코드를 오늘 열어 여섯 가지를 확인한다.

1. print와 콘솔 출력은 전부 프로세스보다 오래 사는 저장처로 갔는가.
2. 모든 줄에 타임스탬프, 서비스 이름, 레벨이 있는가.
3. ERROR 줄은 새벽 2시 테스트를 통과하는가. 혼자 읽어도 행동할 수 있는가.
4. 반복되는 이벤트는 한 줄씩 쏟아지지가 않고, 집계되는가.
5. 로그에 시크릿이 섞여 있지 않은가. 한 번이라도 실려 갔으면 그 로그를 오염된 것으로 취급한다.
6. 같은 종류의 이벤트가 같은 구조인가. 필드명이 매번 다른가.

여섯 개 중 셋을 통과하면 로그가 있다는 단계다. 여섯 개를 통과하면 로그를 찾을 수 있다는 단계다. 후자부터 다음 장이 시작한다. 점검은 10분이면 끝난다. 결과는 표로 남겨 두지 않아도 된다. 머리 속에 여섯 칸을 만들고

하나씩 칸을 채워 가라. 채워질수록 새벽이 부드러워진다.

3 3장. 로그를 찾을 수 있게 만들기

3.1 텍스트 줄이 고꾸라지는 두 질병

결제 API 2026-03-14 09:12:33 ERROR request req_9f2k failed

이런 텍스트 줄은 사람이 읽기는 한다. 사람만이 읽는다. 14일에 실패한 요청을 전부 찾으려면, 줄을 읽고 request ID가 어디쯤 있는지 기억해야 한다. 패턴을 맞춰야 한다. 형식이 조금만 바뀌면 기억이 무효하다. 필드의 순서가 한 칸만 어긋나면 grep이 끊긴다.

텍스트 로그는 두 질병으로 고꾸라진다.

첫 번째, 포맷 추측. 필드의 위치가 줄마다 다르다. request ID가 때로는 세 번째 단어이고, 때로는 마지막 단어다. 검색할 때마다 포맷을 눈으로 확인해야 한다. 확인하는 동안 새벽은 흘러간다.

두 번째, 어휘 추측. 같은 이벤트가 줄마다 다른 단어로 쓰인다. failed, error occurred, could not complete, 실패. 검색은 패턴이 되고 패턴은 변형에 취약하다. 한 단어만 누락되면, 그 단어의 줄은 증거에서 사라진다.

두 질병의 치료제는 같다. 로그에 이름을 붙여라. 값 뒤에 어떤 값인지 말하게 해라.

3.2 구조화 로그

구조화 로그는 값에 이름을 붙인 줄이다.

{“ts”:“2026-03-14T09:12:33Z”,“service”:“payment-api”,“level”:“ERROR”,“msg”:“pay
call failed”,“provider”:“stripe”,“status”:502,“latency_ms”:3120,“request_id”:“req_9f2

이 줄에는 두 개의 독자가 있다. msg 필드는 사람이 읽는 부분이다. payment call failed, 결제 호출이 실패했다. 나머지는 기계가 찾는 부분이다. status가 502이고 provider가 stripe인 줄을 전부, request_id가 req_9f2k인 줄을 전부. 사람이 읽는 것과 기계가 찾는 것이 한 줄에 공존한다.

형식은 거의 언제나 한 줄 한 JSON이다. 한 줄 한 JSON은 줄 기반 도구가 안 깨지기 때문이다. JSON이 여러 줄에 걸쳐 있으면, 그 줄은 세지 못하고 카운트되고 rotation되지 않는다. 형식화 한 줄에 모든 필드를 눌러 넣는 것이 구조화 로그의 기본 포즈다.

표는 이렇게 읽힌다.

필드	역할	예시
ts	시간, UTC	2026-03-14T09:12:33Z
service	컴포넌트	payment-api
level	약속	ERROR
msg	사람이 읽는 구문	payment call failed
request_id	상관관계	req_9f2k

이 다섯 필드는 모든 줄이 갖는 세트다. 이벤트마다 추가 필드, provider나 status나 latency_ms,가 그 위에 얹힌다. 모든 줄이 같은 다섯을 갖고 나머지는 이벤트가 정한다. 이 모양이 검색이 쉬운 모양이다. 필드가 없으면 값만 있고 값이 있으면 이름도 있는 것. 둘 다 지켜지는 순간, 검색은 더 이상 추측이 아니다.

3.3 필드 이름의 약속

구조화 로그에서 이름은 형식이다. request_id를 한 서비스에서는 req로 쓰고, 다른 서비스에서는 correlation_id로 쓰면, 다섯 필드 세트가 무너진다.

값은 같아도 이름이 다르면 검색은 다시 추측이 된다.

규칙은 세 개다.

첫째, 소문자 스네이크 케이스. request_id, latency_ms, attempt. 혼합 표기는 검색에서 대소문자 문제를 만들고 문제는 늘 검색 후에 발견된다.

둘째, 흔한 필드에는 흔한 이름을. ts, service, level, msg, request_id, status, latency_ms, attempt. 대부분의 생태계에서 통하는 이름이다. 통하는 이름을 쓰면, 도구 전환이나 외부 도움이 필요할 때 재학습 비용이 줄어든다.

셋째, 이름 변경은 브레이킹 체인지다. 필드 이름을 바꾸는 순간, 그 이름으로 된 과거 로그는 그 필드를 잃는다. 이름을 바꾸면, 새 이름으로 쓴 줄과 옛 이름으로 쓴 줄이 동시에 존재한다. 둘 다 검색해야 한다. 그래서 이름은 한 번 정하면 바꾸지 않는다. 고쳐야 할 때는 이름을 유지하고 값을 고친다.

3.4 상관관계 ID: 하나의 요청을 잇는 열쇠

하나의 요청은 흔히 여러 서비스를 지난다. 웹 API가 받고 결제 벤더를 부르고 워커가 결과를 다룬다. 세 서비스, 세 로그 흐름. 열쇠가 없으면 이 셋은 서로 모르는 세 개의 산이다.

상관관계 ID가 이 산을 잇는다. 요청의 진입점에서 고유 ID를 만들고 그 줄에 request_id로 쓴다. 서비스가 바뀌는 경계에서는 ID를 요청과 함께 넘긴다, 헤더든 파라미터든. 그러면 req_9f2k를 하나로 검색하면, 세 서비스에 흩어진 줄이 시간 순서로 한 줄로 모인다.

req_9f2k payment-api 09:12:31 INFO request received payment-api 09:12:33 ERROR provider call failed status=502 attempt=1 worker 09:12:34 INFO retry scheduled next_in_ms=15000 payment-api 09:12:47 INFO payment succeeded attempt=2

네 줄의 순서를 읽으면, 사용자가 15초의 지연을 본 이유는 재시도 대기였다. 그 15초는 벤더의 첫 502에서 비롯되었고 두 번째 시도에서 끝났다. 아무에게도 물었고 추측도 않았다. 하나의 ID를 찾고 하나의 순서를 읽었다.

ID는 진입점에서 만든다. 크론 잡처럼 요청이 없는 작업은, 잡 ID나 실행 ID를 같은 역할로 쓴다. 모든 줄이 다른 줄과 이어질 열쇠를 갖는지가 핵심이다. 이어질 수 없는 줄은 섬이다. 섬은 지도가 아니다.

3.5 검색 쿠킹북: 다섯 질문

검색은 질문을 알 때만 유용하다. 실제로 쓸 질문 다섯 개를 아는 것이, 쓸 일이 없는 질문 100개보다 낫다.

1. 지금 이상한 것이 있는가. 지난 15분, ERROR만, 최신순. 알림이 울린 뒤의 첫 질문이다. 아무도 없으면, 알림의 원인은 지표나 인프라의 다른 곳에 있다. 로그는 무죄다.
2. 이 요청은 뭘 겪었는가. request_id로 검색. 하나의 요청의 모든 줄, 시간 순. 상관관계 ID가 이 질문을 가능하게 한다.
3. 언제부터 이상해졌는가. 특정 ERROR를 시간 버킷, 5분 단위로, 그룹화. 개수가 껑충 뛰어오르는 순간을 찾는다. 그 순간이 원인 후보의 경계다, 배포, 설정 변경, 벤더의 장애.
4. 새 버전이 정말 돌아가는가. 마지막 배포 후의 process started 줄. 각 서비스마다 version 필드가 한 번씩, 같은 값으로 나와야 한다. 한 서비스가 없으면, 그 서비스는 재시작되지 않았다. 배포가 되면서 재시작이 안 되는 것은 흔한 사고의 한 종류다.
5. 얼마나 느린가. latency_ms가 임계값을 넘는 줄. 평균을 보지 않는다. 최악의 줄을 본다. 느린 1퍼센트[추정]가 평균보다 더 많은

것을 말한다.

이 다섯 질문은 각각 10초 안에 답해야 한다. 10초가 넘으면, 구조의 문제다. 필드 세트와 상관관계 ID로 돌아가라. 도구가 느리면 도구를 바꾸고 구조가 흐리면 구조를 다시 잡는다.

3.6 검색을 구체적으로: 파일과 grep, jq

플랫폼이 없어도 구조화 로그는 검색된다. 로그가 한 파일에 한 줄 JSON으로 쌓여 있다면, 다섯 질문은 grep과 jq로 답한다.

질문 1, 지금 이상한 것이 있는가:

grep ' "level":"ERROR" ' app.log | tail -20

질문 2, 이 요청은 뭘 겪었는가:

grep 'req_9f2k' app.log

질문 3, 언제부터인가:

grep ' "level":"ERROR" ' app.log | jq -r.ts | cut -c1-16 | sort | uniq -c

ts의 첫 16자, 즉 분 단위까지를 잘라서 센다. 개수가 껑충 도는 분을 찾으면, 그 분이 경계다.

질문 4, 새 버전이 돌아가는가:

grep ' "msg":"process started" ' app.log | grep ' "version":"1.4.2" '

질문 5, 얼마나 느린가:

jq -r 'select(.latency_ms > 2000) |.ts + " " +.request_id' app.log

이 다섯은 외울 수준이다. 새벽에는 새로운 도구를 배우는 시간이 없다. 아는 도구를 아는 형식에 쓰는 것. 구조화 로그의 가치는 바로 이 한 줄에 있다. 형식이 같으면, 도구가 달라도 명령은 같다.

jq가 없다면, grep과 cut만으로 질문 1, 2, 4는 답한다. 질문 3과 5는 awk로 대치한다. 도구의 선택은 환경의 문제고 형식의 약속은 내 문제다. 내 문제를 먼저 끝낸다.

3.7 로그를 어디에 두는가

저장소는 비용의 결정이다.

규모	저장소	검색
작음	파일, rotation 14일[추정]	grep, 작은 도구
중간	로그 집계 서비스	대시보드
큼	관리형 로그 플랫폼	필드 인덱스

작음은 파일 하나, 구조화 줄, 14일 rotation. 다섯 질문을 전부 답할 수 있으면, 규모가 작을 때 필요한 것은 이것이다. 중간은 Loki나 CloudWatch 같은 집계 서비스다. 다섯 질문이 대시보드가 되고 경계가 시각이 된다. 큼은 1인 규모를 넘어선 이야기이므로, 여기서는 끝까지 작음으로 간다.

모든 규모에서 같은 원칙이 있다. ERROR는 오래, INFO는 보통, DEBUG는 기본 꺼짐. DEBUG는 일시적 조사의 도구다. 기록이 아니다. DEBUG를 영구히 켜면 스토리지 비용과 노이즈가 함께 자라고 2시 테스트가 어려워진다. 조사가 필요할 때 켜고, 끝나면 끈다.

보존 기간은 스스로 정할 수 있는 숫자다. 질문은 하나다. 재조사하고 싶은 가장 오래된 인시던트는 며칠 전인가. 1인 개발자에게는 14일에서 30일이[추정] 거의 모든 것을 덮는다. 그 이상은 아카이브다. 아카이브는 싸게 두되, 빨리 찾지 않는다.

부피도 미리 셈해 두자. 수식은 단순하다. 초당 요청 수에, 요청당 줄 수를 곱하고 3,600을 곱하면 시간당 줄 수다. 초당 3개 요청, 요청당 INFO 4줄이면, 시간당 4만 3,000줄[추정]이다.이 숫자는 무섭지 않다. 다만

저장소 결정을 바꾼다. 이 숫자라면 파일 하나와 rotation으로 충분하고, 10배라면 rotation 기간을 줄이거나, 100배라면 플랫폼을 생각해야 한다. 그리고 요청당 줄 수는 2장의 예산이 조절한다. 예산이 3줄에서 6줄로 오르면, 부피도 두 배다. 예산과 보존 기간, 둘 다 숫자로 정해 두면, 로그는 돌발 비용 없이 자란다.

DEBUG를 켜고 끄는 것은 런타임 스위치여야 한다. 설정 한 줄이면 된다. 환경 변수든, config 필드든, 재시작 없이 바뀌는 것이 좋다. DEBUG를 켜려면 배포를 해야 하면, 새벽에 아무도 켜지 않는다. flag 하나면, 조사 중에 임시로 켜고 끝나면 끄는 것이 습관이 된다. 습관이 되어야, DEBUG는 조사의 도구가 된 채, 기록은 남지 않는다.

3.8 이 장에서 얻는 것

다섯 필드, 하나의 ID, 다섯 질문. 이 셋을 적용하면 로그는 찾는 것이 된다.

그러자 다음 질문은 가장 무거운 질문이다. 찾은 것을 믿어도 되는가.

찾을 수 있지만 거짓인 로그는, 없는 로그보다 위험하다. 잘못된 확신을 주기 때문이다. 확신이 잘못되면 행위가 잘못되고 행위가 잘못되면 인시던트는 길어진다. 4장은 이 위험을 정면으로 다룬다. 로그가 거짓말하는 다섯 길, 그리고 그 거짓을 드러내는 규율.

4 4장. 로그가 사실임을 증명하는 규율

4.1 로그가 거짓말하는 다섯 길

로그가 거짓말하는 것은 구조다. 그리고 그 구조는 이미 알려져 있다. 흔한 거짓말은 다섯 개인데, 각자의 해독제가 있다.

첫째, 시계 오차. 두 서버의 시계가 30초씩 틀리면, 뭐가 먼저였는지가 뒤집힌다. A가 B를 불렀는데 로그에는 B가 먼저 보인다. 원인 분석은 순서에서 시작하므로, 순서가 틀리면 원인도 틀린다. 해독제는 NTP. 모든 기계가 시간 동기화를 쓰고 가끔 공공 시간 소스와 비교한다. 서버 두 대가 서로 10초 이상 차이 나는 것은 사고의 전조.

둘째, 유실된 꼬리. 프로세스가 강제 종료되면, 버퍼에 남은 줄은 디스크에 도착하지 않는다. 마지막 몇 줄이 사라진다. 그리고 죽기 직전의 줄은 가장 중요한 줄인 경우가 많다. 해독제는 플러시. ERROR 레벨의 줄은 쓰자마자 내린다, 또는 버퍼 플러시 간격을 짧게 한다. 메모리에만 있는 줄은 아직 쓰이지 않은 줄.

셋째, 시간대 혼합. 어떤 줄은 UTC, 어떤 줄은 로컬. 09:12이 두 개의 순간을 뜻한다. 검색으로 두 시간대를 섞으면, 로그는 두 개의 사실이 겹친 그림자가 된다. 해독제는 하나의 규칙. 저장할 때는 UTC, 볼 때는 로컬. 1장이 말한 시간 속성의 실체다.

넷째, 에러를 먹는 샘플링. 스토리지를 아끼려고 줄을 골라 내리면, 골라진 줄이 필요한 줄일 수 있다. 해독제는 레벨별 샘플링. ERROR와 WARN은 절대 샘플링하지 않는다. INFO와 DEBUG만 내린다. 드문 줄이 값진 줄. 드문 줄을 내리면, 로그는 흔한 일의 기록이 되고 필요한 순간만 잃는다.

다섯째, 없는 로그. 내린 줄도, 죽은 줄도 아니다. 코드가 그 줄을 쓰지

않았다. 코드 패스에 로그가 없는 것. 요청이 걸리면 마지막 줄이 없고 답은 아무것도 없다. 해독제는 경계. 각 단계의 진입과 출구를 쓴다, 2장의 규칙. 그러면 없는 줄이 어디에 있는지가 보인다. 죽은 자리는 두 줄 사이의 간격.

네 가지는 인프라의 문제. 마지막 하나는 설계의 문제. 설계의 문제는 2장의 체크리스트가 해독제다.

다섯 길을 한 눈에 본다.

거짓말	증상	해독제
시계 오차	순서가 뒤집힌다	NTP, 주기적 비교
유실된 꼬리	마지막 줄이 없다	ERROR 즉시 플러시
시간대 혼합	같은 시각이 둘이다	UTC 저장, 로컬 표시
샘플링	드문 줄이 사라진다	레벨별 샘플링
없는 로그	간격이 설명되지 않는다	경계에서 쓰기

증상열이 진단의 실마리다. 순서가 뒤집혀 보이면, 먼저 시계를 의심한다. 마지막 줄이 없으면, 버퍼를 의심한다. 같은 시각이 두 개면, 형식을 의심한다. 로그의 거짓말은 얼굴이 다르다. 얼굴을 알면, 의심할 곳이 줄어든다.

4.2 마지막 로그: 쓰이지 않은 것을 읽는 법

존재하는 줄만 읽는 인시던트 분석은, 가장 중요한 증거를 놓친다. 증거는 없는 줄에도 있다.

기법은 이렇다. 문제 서비스의, 시간 창 안의, 마지막 줄을 찾는다. 만약 마지막 줄이 09:12:31의 request received이고, 다음으로 기대되는 줄이 09:12:33의 provider call start라면, 프로세스는 그 둘 사이에 죽었다. 죽음의 위치는, 두 줄 사이의 간격.

이 기법이 통하는 이유는, 좋은 로그가 경계를 갖고 있기 때문이다. 각 단계에 진입과 출구가 있다. 출구가 없는 단계는, 거기서 죽었다는 뜻이다. 실패에만 로그를 쓴다면, 마지막 로그는 약한 힌트다. 경계에 로그를 쓴다면, 마지막 로그는 위치.

그러므로 줄의 위치의 규칙성이 로그의 가치다. 적지만 맞는 자리의 로그가, 많지만 흩어진 로그보다 낫다. 2장의 예산이 이 원리를 숫자로 만든 것이다.

간격을 읽는 연습을 하나하자. 서비스 두 개를 시간으로 정렬하고 한 쪽의 마지막 줄과 다른 쪽의 첫 줄을 본다. A가 09:12:31에 provider call start를 썼는데, B에 09:12:31의 request received가 없다면, B는 그 요청을 받지 못했다. A와 B 사이의 네트워크에 문제가 있다. A가 받았는데 B가 시작을 안 썼다면, B는 시작 직전에 죽었다. 같은 간격이라도, 읽는 방향이 다르다.

구체적으로 하나 더. 09:12에 결제가 멈췄다고 하자.

09:12:29 INFO worker heartbeat 09:12:31 INFO payment call start request_id=req_77a1 09:12:58 INFO worker heartbeat

req_77a1의 start는 있고 succeeded도 failed도 없다. 다음 heartbeat까지 27초의 침묵. 그리고 그 침묵 동안, 다른 req도 start만 있고 끝이 없다. 읽을 수 있는 결론: 워커는 살아 있고(heartbeat), 결제 호출은 들어가는데 돌아오지 않는다(provider). 내 호출이 걸린 것이거나, 내 프로세스가 호출 직전에 죽은 것 둘 중 하나. heartbeat가 그 사이에서 이어졌으니, 둘째는 반박된다. 벤더 쪽으로 시선이 간다.

여기서 로그가 한 일은, 가능성을 두 개에서 하나로 줄인 것이다. 사실은 여전히 벤더 쪽에서 확인해야 한다. 다만 확인하기 전에, 확인할 곳을 고르는 비용이 줄었다. 그 비용이 새벽의 전부다.

4.3 로그의 탓이 아닐 때

인시던트가 끝나고 로그가 불완전하다고 자책하는 것이 습관이 되면, 규율은 반대가 된다. 모든 사고를 로그 탓으로 돌리지 않는다.

분류를 하자. 첫 번째, 로그가 없어서 몰랐던 사고. 이 것은 2장, 3장의 일이다. 다음 주일에 줄을 추가한다.

두 번째, 로그는 있는데, 사고가 시스템 밖이었다. 벤더의 장애, 네트워크 분할, DNS. 이 경우 로그는 자신의 역할. 시스템 안의 일은 전부 말했고 밖은 밖. 밖의 사실을 로그가 말해줄 수는 없다. 벤더 대시보드나 상태 페이지가 밖의 증인이다. 이 분류에 놓이면, 로그를 더 쓰라는 강박이 줄어든다.

세 번째, 로그가 있어서 속은 사고. 시계 오차, 샘플링, 유실. 이 것은 4장의 첫 섹션의 일이다. 해독제를 적용한다.

분류를 쓰는 이유는, 다음 인시던트에서 같은 착각을 하지 않기 위해서다. 로그를 믿지 못하는 이유는 세 가지가 다르다. 불완전, 시스템 밖, 거짓말. 각각의 대응이 다르다. 대응을 바꾸려면, 원인을 분류해야 한다.

4.4 인시던트 리플레이: 5단계 절차

무언가 부러졌을 때, 절차를 따른다. 절차를 따라하면, 느낌으로 하는 것보다 빠르고 느낌으로 한 것은 미래의 나에게 넘길 수 없기 때문이다.

1단계, 시간 창을 고정한다. 09:10에서 09:30 사이에 부러졌을 것이다. 먼저 경계를 세운다. 창은 작을수록 좋다. 창이 작으면 줄이 적고 줄이 적으면 읽기가 빠르다. 알림 시각이 있으면, 그 시각을 중심으로 ±10분. 없으면, 첫 보고를 받은 시각을 중심으로.

2단계, 범위를 확인한다. 영향을 받은 요청의 수. 모든 사용자, 특정 플랜, 특정 국가, 혹은 한 사용자의 한 요청. 범위가 첫 번째 단서다. 범위가

전체이면 인프라나 벤더의 가능성 높고, 범위가 좁으면 설정이나 데이터의 가능성이 높다.

3단계, 순서를 본다. 레벨로, ERROR부터. 그리고 대표 요청 3개에서 5개를, ID로 끝까지 읽는다. 전부 읽지 않는다. 대표를 읽는다. 대표를 읽으면, 전체의 모양이 보인다.

4단계, 경계를 찾는다. 개수가 껑충 도는 순간, 새 버전이 시작하는 순간, 벤더의 ERROR가 시작하는 순간. 원인은 거의 항상 경계에 있다. 경계에서 원인 후보를 고르고 후보를 시간 창 안에서 검증한다.

5단계, 메모를 쓴다. 3줄이면 된다. 뭐가, 어떤 증거, 뭐를 바꿨다. 증거는 느낌으로 적지 않는다. 쿼리로 적는다. 지난 15분 ERROR 검색, provider 502 확인, 이 것이 메모다. 결제 벤더 같았어, 이것은 메모가 아니다.

메모는 다음 인시던트를 위한 것이다. 6개월 뒤에, 오늘 쓴 3줄에 고맙게 될 것이다. 메모의 형식은 고정하자. 인시던트 이름, 시간 창, 원인 한 줄, 쿼리 한 줄, 조치 한 줄. 형식이 같으면, 메모들은 나중에 서로 검색된다.

4.5 주 15분 훈련

신뢰는 읽기로 자라지 않는다. 습관으로 자란다. 훈련은 하나다.

매주, 지난 달의 인시던트 하나를, 또는 최근의 알림 하나를 가져와서, 로그만으로 답을 해본다. 뭐가, 언제, 어떤 요청, 왜. 답이 나오면, 로그는 사실이다. 답이 나오지 않으면, 원했는데 없는 줄을 적는다. 그 줄이 다음 주일의 일.

이 훈련은 회의보다 싸다. 그리고 다른 누가 찾을 수 없는 빈틈을 찾는다. 이 로그를 쓰는 사람은 나뿐이니까.

두 번째 훈련은 더 작다. 배포 후, 새 버전이 돌아가는가 쿼리를 돌린다, 3장의 질문 4. 각 서비스 한 줄. 줄이 없으면, 배포는 불완전하다. 배포는

되면서 재시작이 안 되는 사고의 한 종류를, 이 훈련이 잡는다.

4.6 끝장 체크리스트

책이 끝나는 자리에, 점검을 둔다. 이번 주 안에 하나씩 답할 수 있어야 한다. 답이 아니면, 그 항목이 다음 주일의 일.

1. ERROR 줄은 새벽에 일어나게 만드는가. 매일 나면, 그 줄의 레벨을 내린다.
2. 줄 하나만으로 행동할 수 있는가. 2시 테스트를 대표 줄 몇 개에 적용해 본다.
3. 모든 줄에 다섯 필드가 있는가. ts, service, level, msg, request_id.
4. 각 단계에 진입과 출구가 있는가. 없는 단계가 마지막 로그를 약하게 만든다.
5. 반복 줄은 집계되는가. 폴링과 재시도가 줄마다 쓰이지는 않는가.
6. ERROR는 즉시 디스크에 내리는가. 버퍼에 머무는 줄은 아직 쓰이지 않은 줄이다.
7. 모든 기계의 시계가 동기화되는가. 서버 두 대를 비교해 본다.
8. 보존 기간은 숫자로 정해져 있는가. 14일에서 30일, 그 이상은 아카이브다.
9. 인시던트 메모가 쿼리로 쓰여 있는가. 느낌으로 쓴 메모는 다음 인시던트에 남지 않는다.
10. 이번 주 훈련을 했는가. 15분. 로그만으로 한 인시던트에 답한 것.

열 개다. 열 개를 전부 답하기보다, 하나를 답하는 것이 먼저다. 하나를 답할 때마다, 새벽은 조금 부드러워진다.

4.7 규율

이 책이 말한 것을 모은다.

로그는 잊지 않는 증인이다. 나중에 시스템에게 물을 질문의 답변을, 미리 써두는 행위다. 1장.

레벨은 약속이다. ERROR는 일어나고 싶은 줄이고, 2시 테스트가 기준이다. 2장.

구조는 검색이다. 다섯 필드, 하나의 ID, 10초에 답하는 다섯 질문. 3장.

신뢰는 검증이다. 다섯 가지 거짓말, 마지막 로그, 리플레이, 주 훈련. 4장.

네 가지는 하나다. 말하는 시스템은 말하는 것을 믿을 수 있는 시스템. 1인 개발자는 모니터링 팀이 필요하지 않다. 새벽에 진짜를 말하는 시스템이 필요하다.

로그는 그 시스템의 입이다. 입은 규율로 만든다. 오늘 첫 줄부터.