# 백업의 규율

잃지 않아야 하는 데이터, 그리고 되돌릴 수 있다는 증명

ThakiCloud (Hyojung Han)

# 백업의 규율

잃지 않아야 하는 데이터, 그리고 되돌릴 수 있다는 증명

ThakiCloud (Hyojung Han)

# Contents

# 1 증명되지 않은 백업은 백업이 아니다

목요일 새벽 2시, 알림이 온다. 결제 데이터베이스가 응답을 멈췄다는 내용.

서버에 접속해 상황을 본다. 디스크가 죽었다. 프로덕션 데이터가 있던 볼륨이 통째로 읽히지 않는다. 마음은 이미 서랍에 넣어 둔 백업으로 향한다. 백업은 존재했다. 매일 새벽 4시에 돌던 스크립트가, 객체 스토리지에 파일을 하나씩 남기던 그 백업.

그 파일을 열려고 한다. 날짜를 보자마자 숨이 막힌다. 3개월 전 것이다. 그 뒤로는 한 번도 성공한 적이 없었다. 스크립트는 매일 돌았지만 디스크가 차서 조용히 실패하고 있었다. 실패를 알려 주는 알림은 없었다. 매일 초록을 확인하던 대시보드도, 사실은 마지막 성공한 파일이 있는지 확인만 하고 초록을 표시하고 있었다.

이 이야기는 실제 여러 건의 사고를 섞어 재구성한 것이다. 패턴은 늘 비슷하다. 백업을 하고 있다고 확신하던 지점, 실패가 조용히 멈추지 않던 지점, 그리고 그 두 지점을 아무것도 이어 주지 않던 지점. 이 책이 다루는 것은 바로 그 빈 지점이다.

핵심 구분을 하나 세워 보자. "백업을 하고 있다"는 것과 "되돌릴 수 있다"는 것은 전혀 다른 두 가지 상태다. 전자는 희망이고 후자는 증명이다. 대부분의 1인 운영 시스템은 희망의 상태에 머물러 있다. 백업 파일이 존재한다는 사실, 크기가 들어 보인다는 사실, 목록이 나온다는 사실. 이 세 가지는 모두 "백업이 있다"를 증명할 뿐, "되돌릴 수 있다"를 증명하지는 않는다.

이 장의 나머지는 이 희망을 증명으로 바꾸는 첫 단계를 다룬다. 백업이 어떻게, 왜, 어떤 얼굴로 실패하는지를 먼저 알아야, 그것을 막는 구조를 세울 수 있다.

## 1.1 백업, 복제, 미러를 섞어 쓰지 말자

데이터를 지킨다는 말에는 세 가지 도구가 숨어 있다. 백업, 복제, 미러. 셋 다 데이터를 지킬 수 있지만 실패하는 방식이 다르다. 그리고 이 차이를 모르면, "데이터는 지켰는데 잃어버렸다"는 일만 생긴다.

복제와 미러는 원본이 바뀔 때마다 따라가는 실시간 사본이다. 원본과 같은 운명을 공유한다. 같은 디스크가 죽으면 둘 다 사라진다. 같은 클라우드 지역에서 장애가 나면 둘 다 멈춘다. 그리고 가장 무서운 경우, 사람의 실수나 잘못된 마이그레이션으로 원본이 지워지는 경우에는 따라가는 사본도 함께 지워진다. 실수로 삭제한 것을, 실시간으로 복제받는 쪽까지 전달해 버리는 구조다.

백업은 그 반대다. 백업은 특정 시점의 데이터를 독립적으로 복사해 두는 것이다. 원본에 지금 이 순간에 무슨 일이 일어나도, 백업은 과거의 깨끗한 시점을 유지한다. 그래서 데이터 손실로부터 진정으로 구해 주는 것은 백업이다. 복제는 가용성, 즉 멈추지 않는 것을 위한 도구다. 둘을 같은 서랍에 넣지 말라. 가용성을 위해 복제를 쓰고 손실을 위해 백업을 쓴다. 각각이 담당하는 실패가 다르다.

| 도구 | 지키는 실패 | 놓치는 실패 |
|---|---|---|
| 복제, 미러 | 하드웨어, 장애 | 같은 시점 오염 |
| 백업 | 손실, 오염 | 시차, 시간 |

## 1.2 백업 실패의 네 가지 얼굴

백업이 실패하는 방식은 대개 네 갈래다.

첫째, 사라짐이다. 백업 파일 자체가 없다. 스크립트가 조용히 죽었거나, 디스크가 차서 쓰지 못했거나, 백업 대상 경로가 바뀌어 빈 폴더를 복사하고

있었다. 흔한 것은, 실패를 아무도 모른다는 점이다.

둘째, 손상이다. 파일은 있는데 깨져 있다. 전원이 나가면서 쓰다 만 파일, bit rot, 압축 도구가 만든 깨진 아카이브. 열려야 할 바로 그 순간, 열리지 않는다.

셋째, 오래됨이다. 백업은 있지만 수개월 전 것이다. 복원하면 시스템은 되살아나지만 그 사이 쌓은 데이터가 통째로 날아간다. "되돌릴 수 있다"는 말은 참이되, 돌아간 시점이 쓸모없다면 거짓과 다르지 않다.

넷째, 못 읽음이다. 파일은 멀쩡히 있는데, 그 파일을 열 수 있는 도구가 더 이상 없다. 폐기된 소프트웨어 버전, 더 이상 구할 수 없는 독점 형식, 열쇠를 잃은 암호화된 아카이브. 백업은 있는데, 그것을 읽을 수 있는 손이 사라진 상태다.

이 네 가지를 구분해야 하는 이유는, 각각이 요구하는 방어책이 다르기 때문이다. 사라짐은 알림으로, 손상은 무결성 검사로, 오래됨은 보관 정책으로, 못 읽음은 도구의 지속 가능성과 키 관리로 막는다. 하나를 막는다고 나머지가 막히는 것은 아니다.

| 얼굴 | 흔한 징후 | 막는 방법 |
|---|---|---|
| 사라짐 | 파일이 안 보인다 | 실패 알림 |
| 손상 | 열릴 때 깨짐 | 무결성 검사 |
| 오래됨 | 날짜가 오래됨 | 보관 주기 |
| 못 읽음 | 도구가 사라짐 | 키와 형식 관리 |

## 1.3 시스템이 없는 한 주

추상적으로 말하지 말고 백업 체계가 없는 1인 시스템의 일주일을 따라가 보자.

월요일, 기능을 추가한다. 데이터 스키마에 열을 하나 넣는다. 마이그레이션

스크립트를 작성하고 프로덕션에 실행한다. 백업 대상에 무엇이 새로 들어왔는지는 생각하지 않는다. 화요일, 저장소 구조를 정리한다. 업로드 파일을 다른 경로로 옮긴다. 백업 스크립트는 여전히 오래된 경로를 복사하고 있지만, 아무도 모른다.

수요일, 객체 스토리지 용량이 한계에 근접한다. 요금을 줄이려고 오래된 버전을 정리한다. 실수로 최근 백업과 함께, 보관하던 오래된 백업도 지운다. 목요일, 스키마 마이그레이션이 문제였다. 되돌리고 싶다. 하지만 되돌릴 스키마를 담은 백업은, 화요일 경로 정리 이후로는 빈 폴더의 복사물뿐이다. 금요일, 상황을 정리한다. 잃어버린 것은 되돌릴 수 있다는 확신이다.

이 일주일에서 터진 것은 하나다. 백업과 현실 사이에 아무도 서 있지 않았다. 스키마가 바뀌어도, 경로가 바뀌어도, 용량이 차도, 백업은 그것을 몰랐다. 그리고 백업이 몰랐다는 사실조차, 아무도 몰랐다.

## 1.4 조용히 실패하는 데는 이유가 있다

혼자서 시스템을 돌리는 사람에게는, 실패를 알려 줄 사람이 없다. 팀이 있으면 백업이 죽어도 누군가의 대시보드가 빨갛게 변하고, 누군가가 그 빨간 것을 본다. 하지만 1인 시스템에서는 대시보드를 보는 것도, 빨간 것을 알아채는 것도 결국 같은 한 사람이다. 그리고 그 사람은 대개 다른 일을 하고 있다.

그래서 조용한 실패는 구조적으로 발생한다. 백업 스크립트는 매일 새벽에 돌고, 아무도 그 시간에는 화면을 안 본다. 실패가 쌓여도 아무에게도 알려지지 않는다. 대시보드가 초록을 보인 것은, 마지막 성공 파일이 여전히 존재해서인 경우도 많다. 이 초록은 "한 번은 잘 되었던 적이 있다"를 보여 줄 뿐이다.

이 장이 말하는 규율은 여기에 있다. 실패를 조용히 두지 말라는 것이다. 백업이 실패하면, 그 실패는 다음 사고를 준비하는 신호다. 신호를 보내는

경로를 반드시 하나 이상 만들어 두어야 한다. 알림이 없다면, 최소한 다음에 백업을 열었을 때 실패가 보이도록 남겨 두어야 한다. 초록을 믿는 습관이, 3개월 뒤에 식은땀을 만든다.

## 1.5 백업의 유효성은 속성이 아니라 연습이다

핵심 명제를 하나 세워 보자. 백업이 동작한다는 증거는 오직 하나, 성공한 복원이다.

이것은 지루한 진리처럼 보이나, 대부분의 실패는 바로 이 지루한 진리를 생략한 데서 온다. 파일을 만들어 두었다는 사실은 "백업이 있다"를 증명한다. 하지만 "되돌릴 수 있다"는 말은, 실제로 되돌려 보았을 때만 성립한다.

그렇다면 복원을 언제 연습하는가. 이 책의 답은 간단하다. 복원 연습은 백업과 같은 주기로, 백업만큼 당연하게 돌아야 한다. 매일 백업을 한다면, 일주일에 한 번은 샘플 복원을 점검하고, 한 달에 한 번은 전체 복원을 실행한다. 이 비용은 작다. 하지만 사고 날 때 이 연습을 안 한 대가는, 잃어버린 데이터 그 자체다.

구체적으로 무엇을 하는지 보자. 매일 밤 데이터베이스를 덤프하고 객체 스토리지에 올린다고 치자. 일주일에 한 번의 샘플 복원은, 그 덤프 파일을 임시 스키마에 실제로 다시 올려 보는 일이다. 파일을 내려받고 복원 명령을 실행하고 표의 개수와 행의 수가 원본과 일치하는지 확인한다. 한 달에 한 번의 전체 복원은, 새 인스턴스를 만들고 덤프를 올려, 앱을 그 위에 다시 띄워, 핵심 기능을 하나씩 눌러 보는 일이다. 이 전체 복원에서 앱이 뜨지 않는다면, 그것은 뜨지 않는 백업이다.

## 1.6 스스로에게 물어볼 다섯 질문

매달 10분만 내서 이 다섯 질문에 답해 보자. 답이 막히는 순간이 곧 백업의 허점이다.

첫째, 가장 마지막에 백업이 성공한 시점이 언제인가. 둘째, 가장 마지막에 복원이 성공한 시점이 언제인가. 셋째, 지금 복원하면 어느 시점의 데이터를 되찾을 수 있는가. 넷째, 그 백업을 지금 열 수 있는 도구가 내 손에 있는가. 다섯째, 백업이 암호화되어 있다면, 그 열쇠를 지금 꺼낼 수 있는가.

이 다섯 중 하나라도 "잘 모르겠다"가 나오면, 아직 희망인 것이다.

## 1.7 RPO와 RTO를 숫자로 세우자

마지막으로, "얼마나 빠른 시점까지"와 "얼마나 빨리"를 숫자로 정해 두자.

RPO는 복원 시점 목표다. 잃어도 되는 데이터의 양을 시간으로 쓴 것이다. 매일 밤 백업한다면 RPO는 최대 24시간이다. 1시간마다 하면 1시간이다. RTO는 복구 시간 목표다. 사고를 알아차린 뒤 실제 서비스를 다시 올리는데 걸리는 시간이다.

| 목표 | 의미 | 실측 방법 |
|---|---|---|
| RPO | 잃어도 되는 시간 | 백업 간격 |
| RTO | 복구에 쓰는 시간 | 복원 연습 기록 |

구체적으로 하나 잡아 보자. 온라인으로 예약을 받는 작은 서비스라고 하자. 고객이 예약을 남기고 운영자가 그 예약을 확인해 답을 보내는 구조다. 이 서비스에서 RPO가 24시간이라면, 하루치 예약을 잃을 수 있다는 뜻이다. 하루 예약이 10건이라고 하면, 최악의 경우 하루치가 통째로 사라진다[추정]. 그것이 받아들여진다면, 매일 백업이면 된다. 받아들여지지 않다면,

백업 간격을 줄여야 한다. RTO도 마찬가지다. 새벽 2시에 디스크가 죽어, 오전 9시에 고객을 다시 맞아야 한다면, RTO는 그보다 짧아야 한다.

이 숫자는 사후에 정하기 쉬운데, 그때 정하면 이미 늦다. 서비스의 성격에 맞춰 미리 쓰고, 백업 주기와 복원 연습을 그 숫자에 맞춰 설계한다. 1인 운영이라면, RPO 24시간, RTO 반나절 같은 현실적인 목표가 출발점이 될 수 있다 [추정]. 중요한 것은 목표가 숫자라는 사실, 그리고 그 숫자를 실제로 검증했다는 사실이다.

## 1.8 오늘 밤에 해 볼 일

책 전체를 이해한 다음에 시작할 필요는 없다. 오늘 밤 10분이면 된다.

가장 마지막에 백업이 성공한 시각을, 가장 마지막에 복원이 성공한 시각을, 각각 적어 보라. 두 줄이면 된다. 복원의 시각이 빈 칸이라면, 그 백업은 아직 희망인 것이다. 백업 시각도 빈 칸이라면, 이 책의 나머지 장을 처음부터 따라오면 된다.

# 2 무엇을 백업하고 무엇을 놓치는가

데이터 손실을 복기해 보면, 의외로 데이터베이스 그 자체가 원인이 적다. 진짜로 날아간 것은 그 옆에 작게 놓여 있던 것들이다. 설정, 키, 스케줄러, 연결 문자열. 데이터베이스는 크고 눈에 뜨여 백업을 떠올리게 하지만, 작은 것들은 잊힌다. 그래서 이 장에서는 무엇을 백업할지 정하는 분류법과, 특히 놓치기 쉬운 것들의 목록을 다룬다.

## 2.1 다시 만들 수 있는 것과 없는 것

데이터를 세 갈래로 나눠 보자.

첫째, 다시 만들 수 없는 것이다. 고객이 남긴 데이터, 거래 기록, 생성한 결과물. 이것은 소실되면 회복이 불가능하거나, 회복 비용이 사업 전체를 건드린다. 백업의 최우선이다.

둘째, 다시 만들 수 있는 것이다. 빌드 결과물, 캐시, 다른 곳에서 다시 받아올 수 있는 패키지. 소실해도 재생산 가능. 백업 우선순위는 낮다. 다만 재생산에 드는 시간과 비용을 고려해 일부는 제외한다.

셋째, 작지만 재구성 비용이 큰 것이다. 설정 파일, 인증 키, 배포 스크립트, 스키마 마이그레이션 역사. 용량은 기가바이트가 안 되지만 이것들이 없으면 첫째를 돌려도 서비스가 다시 서지 않는다.

실수의 대부분은 셋째를 백업 목록에서 뺀 데서 온다. "이건 텍스트 파일이잖아" 하고 넘기면, 어느 날 그 텍스트 파일이 서비스 전체를 묶는 마지막 끈이 되는 것을 알게 된다.

## 2.2 보이는 데이터와 숨은 데이터

서버를 뒤져 보면 데이터는 두 층으로 나뉜다. 보이는 층과 숨은 층.

보이는 층은 데이터베이스와 업로드된 파일이다. 숨은 층은 다음 것들이다.

- 환경변수와 연결 문자열
- 인증서, 키, 토큰
- 스케줄러 설정
- 도메인과 DNS 기록
- 방화벽과 접근 규칙
- CI/CD 설정
- 스키마 마이그레이션 역사
- 외부 API 인증 정보
- 인프라 코드, 그리고 그 바깥의 수작업 변경

숨은 층의 공통점은, 따로 모아 놓은 곳이 없다는 것이다. 여기 한 줄, 저기 한 줄, 그리고 누구의 머릿속 한 줄. 그래서 사고가 나면 "그 값이 뭐였더라" 하는 순간부터 복구 시간이 도둑맞기 시작한다.

구체적으로 하나 보자. 결제 서비스를 돌리는 1인 시스템이라고 하자. 보이는 층에는 주문 데이터베이스와 업로드한 영수증 파일이 있다. 숨은 층에는, 결제 게이트웨이의 API 키, Webhook 시그니처를 검증하는 값, 타임존 설정, 도메인과 DNS 기록, 그리고 그 도메인을 어떤 registrar에서, 어떤 값으로 인증했다는 기록이 있다. 주문 데이터베이스를 그대로 돌려도, Webhook 시그니처 값이 없으면 결제 성공 통보가 들어오지 않는다. 데이터는 온전하지만 서비스는 죽은 셈이다.

## 2.3 데이터 자산 지도를 그리는 세 단계

"무엇을 백업할지"를 결정하는 반복 가능한 절차를 세 단계로 정리해 보자.

첫 단계, 쓰기 경로를 추적한다. 새 데이터는 어디로 들어오는가. 사용자가 폼을 제출하면, Webhook이 오면, 예약 작업이 돌면, 각각 어디에 쓰이는가. 쓰이는 모든 위치가 백업 후보다.

두 단계, 의존성으로 확장한다. 앱이 멈췄다고 치자. 다시 실행하게 하려면 어떤 데이터, 어떤 설정, 어떤 키가 필요한가. 그 목록이 진짜 백업 범위다.

세 단계, 소실 시나리오를 하나씩 통과시킨다. 자산마다 "이것이 사라졌다고 치면"을 붙여 본다. 한 시간 안에 다시 만들 수 있으면 백업에서 빼도 된다. 아니면 백업 대상이다. 이 질문은 한 번도 해본 적이 없지만, 실제로는 매일 서비스의 목을 조르는 질문이다.

이 세 단계의 결과를 한 장에 모으면, 그것이 데이터 자산 지도다. 지도에는 자산 이름, 쓰는 위치, 다시 만들 수 있는지, 백업에 포함되어 있는지, 마지막 확인일이 들어간다. 한 장이면 된다.

| 자산 | 다시 만들 수 있나 | 백업 |
|---|---|---|
| 고객 데이터 | 불가 | 포함 |
| 설정, 키 | 어려움 | 포함 |
| 캐시, 빌드 | 가능 | 제외 |

## 2.4 작은 SaaS의 자산 지도를 채워 보기

추상적으로 끝내지 말고 작은 SaaS의 지도를 하나 채워 보자. 이 서비스는 사용자가 가입하고 계정을 만들고 월로 과금되는 구조다.

데이터베이스는 고객과 계정과 과금 기록을 담는다. 다시 만들 수 없다. 매일 백업. 업로드 폴더는 사용자 프로필 이미지와 첨부 파일을 담는다. 다시 만들 수 없다. 매일 백업. 코드베이스는 git에 있다. git은 백업이 아니므로, 리포지토리 자체를 주기적으로 원격 저장소에 푸시한다. 환경변수와 연결 문자열은 서버와 프로세스 매니저에 흩어져 있다. 다시

만들기 어렵다. 월 한 번, 런북에 적어 둔다. 결제 게이트웨이의 키와 Webhook 시그니처 값은, 사라지면 과금이 멈춘다. 다시 만들기 어렵다. 월 한 번, 런북에 적어 둔다. 도메인과 DNS 기록은 registrar에 있다. 이 기록이 사라지면 서비스가 안 열리지만, registrar는 남아 있다. 백업이 아니라, 런북에 위치만 적어 두면 된다. CI/CD 설정은 코드 저장소에 있다. 코드와 함께 푸시된다.

이 지도를 보면, 백업 대상이 데이터베이스와 업로드뿐이 아니라, 환경변수와 키와 설정까지 포함된다. 그리고 git이 코드의 백업은 되지만, 데이터와 값의 백업은 되지 않는다. 이 구분이 1인 시스템에서 가장 먼저 넘어지는 자리다.

## 2.5 백업이라고 착각하는 것들

1인 시스템에서 자주 “백업이다”라고 불리지만 사실은 다른 것들이 있다.

git은 버전 관리다. 코드의 변이를 기록하지만 git 리포지토리 자체가 사라지면 그것도 함께 사라진다. 게다가 git은 데이터베이스의 내용, 환경변수의 값, 업로드된 파일을 담지 않는다. git을 백업한다고 생각하면, 코드는 남아도 데이터는 없는 상태가 된다.

클라우드 스냅샷은 백업처럼 보이지만 대개 같은 계정, 같은 지역에 있다. 계정에서 지워지면 스냅샷도 함께 지워진다. 지역이 죽으면 스냅샷도 읽을 수 없다. 스냅샷은 백업이 아니라, 같은 실패 영역 안의 또 다른 복사일 뿐이다.

이메일로 받아 두면도 마찬가지다. 이메일에는 값이 있지만 이메일 계정 자체가 사라지면, 그리고 이메일을 찾지 못하면, 그것도 없다. 더 나쁜 것은, 이메일은 구조가 없어 복구가 어렵다.

이것들의 공통점은, 원본과 같은 운명을 공유한다는 것이다. git, 스냅샷, 이메일. 셋 다 여기에 있으니 안전하다는 착각을 만들어 준다. 하지만

안전은, 원본이 사라져도 남아 있을 때만 성립한다.

## 2.6 자산별로 주기를 달리하자

모든 데이터에 같은 백업 주기를 쓰면 비용만 올라간다. 자산의 성격에 따라 주기를 달리하자.

다시 만들 수 없는 고객 데이터는 주기를 짧게 한다. 매일, 혹은 그 이상. 작지만 재구성 비용이 큰 설정과 키는, 바뀔 때마다, 또는 적어도 주 한 번에 한다. 다시 만들 수 있는 캐시는, 백업에서 제외해도 된다.

이 차등화가 중요한 이유는, "모든 것을 매일 백업한다"는 말의 뒤에 숨은 비용이다. 용량, 요금, 그리고 더 나쁜 것은, 백업이 너무 많아서 중요한 백업이 묻혀버리는 것이다. 자산의 가치에 맞춰 주기를 세우면, 백업은 줄고 되돌릴 수 있다는 확신은 커진다.

## 2.7 소실 시나리오별로 대상을 달리하자

무엇을 백업할지는, 무엇에서 잃는지에 따라 달라진다. 같은 자산도, 소실의 종류에 따라 다른 대우를 받는다.

디스크가 죽으면, 그 디스크에 있는 모든 것이 사라진다. 데이터베이스, 업로드, 설정. 그래서 디스크 실패에는, 그 디스크에 있는 것을 전부 다른 곳에 두어야 한다.

지역이 죽으면, 그 지역의 모든 것이 읽히지 않는다. 클라우드 서비스, 객체 스토리지, 그리고 그곳에 둔 백업. 그래서 지역 실패에는, 백업을 다른 지역에 두어야 한다.

계정이 지워지면, 그 계정에 붙어 있는 것이 사라진다. 클라우드 리소스, 스냅샷, 객체 스토리지. 그래서 계정 실패에는, 백업을 다른 계정에 두어야 한다.

사람이 실수하면, 원본과 따라가는 사본이 함께 오염된다. 그래서 실수에는, 독립된 시점의 백업이 있어야 한다.

같은 백업이라도, 막고 싶은 소실의 종류가 다르다면, 놓이는 곳도 달라야 한다. 이것이 다음 장에서 실패 영역이라고 부를 것의 첫 조각이다. 여기서는, 무엇을 백업할지를 정할 때, 그것을 어디에 놓을지를 함께 생각해 두라는 뜻이다.

## 2.8 머릿속이 가장 위험한 저장소

가장 취약한 저장소는 서버도 클라우드도 아니다. 운영자의 머릿속이다. "그 포트는 뭐였지", "그 도메인은 어디에서 설정했지", "그 키는 왜 이 값이었지". 이 지식은 백업될 수 없고 사람이 멈추면 사라진다. 1인 시스템에서는 운영자가 곧 유일한 백업이다. 그런데 그 백업은 가장 잘 망가진다. 피곤할 때, 긴장할 때, 사고가 난 바로 그 순간, 머릿속 백업은 먼저 읽히지 않는다.

대안은 단순하다. 복구를 위해 필요한 절차와 값을, 백업과 같은 장소에, 백업과 같은 주기로 문서로 남겨 두라는 것이다. 런북 한 페이지가, 잃어버린 값을 한 시간 동안 헤매는 것보다 싸다.

## 2.9 월 30분의 재조사

데이터 자산은 고정되어 있지 않다. 새 기능을 올리면 새 쓰기 경로가 생기고, 새 서비스가 의존하는 설정이 늘어난다. 그래서 매월 30분만 내서 자산 지도를 다시 그린다. 지난 달과 무엇이 바뀌었는가. 새로 생긴 쓰기 경로는 백업에 포함되었는가. 숨은 층의 값은 아직인가. 이 30분이, 다음 사고 때의 몇 시간을 사고낸다.

## 2.10 백업 전에 확인하는 체크리스트

자신이 돌리는 시스템에 대해, 다음 질문에 하나씩 답해 보라. 답이 막히는 곳이 곧 백업의 허점이다.

- 사용자의 입력은 어디로 쓰이는가.
- Webhook과 예약 작업은 어떤 데이터를 만드는가.
- 서버에 있는 설정 파일은, 원본과 다른 값이 있는가.
- 프로덕션에만 있는 값, 그 원본은 어디에 있는가.
- 이 시스템이 죽으면, 다시 세우려면 무엇이 필요한가.
- 그 "무엇"은 지금 백업에 포함되어 있는가.
- 백업이 오늘 밤에 죽어도, 내일 아침에 알아채는가.

이 질문의 답이 모두 "알고 있다"가 아니라면, 백업의 대상은 아직 다 찾지 못한 것이다. 체크리스트는 10분이면 된다. 하지만 10분의 답이, 다음 사고 때의 방향을 정한다.

## 2.11 오늘 밤에 해 볼 일

자산 지도의 첫 장을 오늘 밤에 만들어 보라. 쓰는 위치, 다시 만들 수 있는지, 백업에 포함되어 있는지를, 눈에 보이는 데이터부터 5개만 적어 보라. 5개를 다 적었다면, 이제 숨은 층으로 넘어간다. 환경변수, 키, 스케줄러, 도메인. 이 4개 그룹만 확인해도, 대부분의 1인 시스템은 이미 몰랐던 백업 대상을 하나씩 발견하게 된다. 발견한 것 하나하나가, 다음에 잃지 않을 데이터를 하나씩 더해 준다.

# 3 어디에, 얼마나 오래, 어떻게 복제하는가

1장에서는 백업이 왜 조용히 죽는지, 2장에서는 무엇을 백업할지를 정했다. 이번 장은 그 백업을 어디에, 얼마나 오래, 어떤 방식으로 놓는가 다룬다. 핵심은 하나. 백업은 원본과 같은 실패 영역에 있으면, 원본과 함께 죽는다. 그래서 백업의 위치는, 막고 싶은 실패의 종류를 먼저 정한 다음에 결정한다.

## 3.1 실패 영역부터 세다

"다른 곳에 놓아라"는 말은 모호하다. 다른 곳이 무엇을 뜻하는지, 실패 영역으로 정해 보자.

- 같은 호스트: 디스크 하나가 죽으면 원본과 백업이 함께 사라진다.
- 같은 지역: 클라우드 지역 장애가 나면 둘 다 읽히지 않는다.
- 같은 계정: 실수나 탈취로 계정이 지워지면 둘 다 날아간다.
- 같은 시간: 잘못된 배포나 마이그레이션이 원본을 지우면, 따라가는 사본도 함께 지워진다.

백업의 각 계층은 이 실패 영역을 하나씩 다른 데 놓아야 한다. 이것이 이 장의 전부다. 나머지는, 이 원칙을 돈과 노력이 허용하는 한도 안에서 구체화하는 일이다.

구체적으로, 원본이 지역 A의 한 호스트에 있고 백업이 같은 호스트의 다른 디스크에 있다면, 실패 영역은 하나다. 그 디스크가 같은 저장 장치에, 같은 전원에, 같은 지역에 붙어 있으므로, 한 번의 사고로 둘 다 잃는다. 백업이 같은 지역 A의 다른 호스트라면, 실패 영역은 두 개다. 호스트 실패는 막지만, 지역 실패는 막지 못한다. 백업이 지역 B라면, 실패 영역은 세 개다. 호스트, 지역, 그리고 다른 계정이면 계정까지 나뉜다.

## 3.2 3-2-1을 구호가 아니라 계산으로

데이터의 세 사본, 두 가지 다른 매체, 하나를 원본에서 멀리. 3-2-1은 널리 알려진 공식이다. 하지만 1인 운영자가 그대로 따르기엔, 사본 세 개를 모두 유지하는 비용과 노력이 커진다. 그래서 현실적 변형으로 이렇게 세운다.

- 원본: 프로덕션 시스템
- 사본 1: 클라우드 객체 스토리지, 다른 지역
- 사본 2: 다른 계정, 또는 주기적 오프라인

핵심은 계층마다 실패 영역이 겹치지 않는 것이다. 사본이 세 개든 둘이든, 같은 실패 영역에 두 개를 넣으면 실제로는 하나밖에 없는 셈이다. 3-2-1의 숫자를 외우지 말고 실패 영역을 겹치지 않게 두는다는 뜻으로 읽자.

## 3.3 저장소를 고르는 일

각 계층을 어디에 놓을지, 선택지를 하나씩 보자.

로컬 서버의 백업은 가장 빠르지만 실패 영역이 원본과 겹친다. 같은 호스트, 같은 디스크, 같은 지역. 그래서 로컬 백업은 최속 계층으로는 쓸 수 있어도, 마지막 계층으로는 쓸 수 없다. 원본이 죽으면 함께 죽는 구조다.

클라우드 객체 스토리지는 1인 운영자에게 좋은 첫 번째 사본 장소다. 이유는 관리가 거의 없고 버저닝과 수명 주기 규칙이 내장되어 있기 때문이다. 주의할 것도 있다. 버저닝은 인간 실수의 적이 될 수 있다. 파일을 실수로 지우면, 새 삭제 마커 버전이 만들어져 원래 파일은 버전 역사로 숨는다. 그리고 수명 주기 규칙을 잘못 설정하면, 보관하기로 한 것을 조용히 삭제한다. 규칙을 쓸 때는 반드시 삭제가 아니라, 더 싼 보관으로 옮기는 저장 클래스 전환으로 설계한다.

오프라인, 또는 다른 계정의 매체는 마지막 계층이다. 접근은 어렵지만 실패 영역은 가장 멀리 있다. 주기적으로, 한 달에 한 번, 같은 것을

밀봉해서 옮겨 둔다. 이 계층이 있기에, "계정이 통째로 지워졌다"는 최악의 시나리오에서도 살아 남는 사본이 하나 있다.

구체적으로 객체 스토리지 계층을 하나 잡아 보자. 매일 밤, 데이터베이스 덤프를 만들어, 암호화해서, 버킷의 날짜 폴더에 올린다. 버킷은 다른 지역에 두고 버전 관리를 켜 둔다. 수명 주기 규칙은, 최근 7일 동안의 객체를 표준 보관에 두고, 30일 이상 지난 것은 저비용 보관으로 전환하고, 1년 이상은 아카이브로 전환한다. 삭제는 수명 규칙이 아니라, 사람이 명시적으로 한다. 버저닝이 켜져 있어도, 삭제 마커가 쌓이지 않도록, 주기적으로 오래된 버전을 정리하는 별도 작업을 둔다.

| 저장소 | 강점 | 약점 |
|---|---|---|
| 로컬 서버 | 빠름 | 같은 실패 영역 |
| 객체 스토리지 | 관리 없음, 버저닝 | 실수로 삭제 |
| 오프라인, 다른 계정 | 가장 멀리 | 접근 어려움 |

## 3.4 보관 기간: 오래된 백업이 왜 필요한가

"백업은 오래될수록 쓸모없다"는 통념을 한 번 뒤집어 보자. 오래된 백업이 필요한 이유는 두 가지다.

첫째, 논리적 오류는 나중에야 드러난다. 잘못된 배포나 마이그레이션은 그날 밤이 아니라, 일주일 뒤에, 또는 한 달 뒤에 누군가 요즘 데이터가 이상하다 하고 말하며 발견된다. 그때 원본과 최신 백업은 모두 이미 오염되어 있다. 오염되기 직전의 오래된 백업만이 깨끗하다.

둘째, 암호화 공격, 즉 랜섬웨어는 아직 살아 있다는 것을 몇 주 뒤에야 알아채는 경우가 있다. 공격자가 최신 백업까지 지우거나 오염시켰을 때, 몇 주 전의 깨끗한 백업만이 살아 있다. 그래서 오래된 백업을 반드시 두어야 한다 [추정].

그래서 보관은 최근만이 아니라, 최근, 주 단위, 월 단위, 해 단위를 겹쳐 놓는 세대 구조가 좋다. 매일 백업의 최근 며칠, 매주 백업의 최근 몇 주, 매월 백업의 최근 몇 달, 그리고 가끔 해 단위까지. 이 세대 구조가, "오래됨"이라는 실패의 얼굴을 가격과 교환해 준다.

| 세대 | 주기 | 보관 |
|---|---|---|
| 일 | 매일 | 최근 7일 |
| 주 | 매주 | 최근 4주 |
| 월 | 매월 | 최근 12개월 |
| 년 | 연 1회 | 3년 |

객체 스토리지 중 일부는, 객체 잠금, 즉 임마셔빌리티를 제공한다. 일정 기간 동안 삭제 자체가 불가능하도록 객체를 잠가 두는 기능이다. 랜섬웨어 시나리오에서 이것은 강력하다. 공격자가 아무리 백업 버킷에 접근해도, 잠긴 객체는 지울 수 없다. 다만, 잠그면 본인도 지울 수 없으므로, 잠금 기간은 오래됨을 막는 수준에서, 위 세대의 보관 기간과 맞추어야 한다. 잠금을 켜는 것은 선택이지만 1인 시스템에서 지워질 수 있는 백업과 지워지지 않는 백업의 차이를 만들어 주는 단 하나의 도구다.

## 3.5 암호화: 원본을 떠나기 전에

백업은 원본보다 위험한 위치에 놓인다. 원본은 접근이 제한된 프로덕션인데, 백업은 언젠가 쓸 거니까 하고 느슨하게 두기 쉽다. 그래서 백업이 공격자의 두 번째 표적이 된다.

방어는 원본을 떠나기 전에 암호화하는 것이다. 서버에서 평문으로 백업을 만든 뒤 올리는 것이 아니라, 먼저 로컬에서 암호화하고 그 암호화된 것을 옮겨라. 암호화 키는 백업과 같은 장소에 두지 않는다. 키가 백업과 함께 죽으면, 백업은 영문이다.

키 관리의 핵심 질문은, 키를 어디에 두는가다. 키도 백업해야 하고 백업과 다른 실패 영역에 두어야 한다. 그래서 키는 별도 저장소, 또는 신뢰하는 사람의 손에, 또는 비밀 관리 도구의 별도 계정에 둔다. 백업과 키를 동시에 잃으면 복구 불가능하다. 이 둘을 같은 실패 영역에 묶어 두지 말라.

## 3.6 계정: 유리문을 깨는 별도의 손

백업을 올리는 계정은, 매일 앱이 쓰는 계정을 재사용하지 않는 것이 좋다. 별도 계정, 최소 권한, 긴 수명의 토큰을 피하는 구조가 안전하다.

이유는 탈취 시나리오다. 서버에서 매일 쓰는 접근 키가 도난당하면, 공격자는 그 키로 프로덕션만 아니라, 그 키가 닿는 모든 것, 즉 백업 저장소까지 지울 수 있다. 백업은 유리문을 깨고 들어가는 별도의 계정에만 닿게 해 두면, 원본 계정이 털려도 백업은 살아 있다.

## 3.7 백업의 실패 영역을 문서로 남겨 둔다

백업이 어디에, 어떻게, 얼마나 있는지, 그 사실이 어디에 적혀 있는가. 1인 시스템에서는 그 답이 백업과 함께, 백업과 다른 실패 영역에 있어야 한다.

백업의 설계, 즉 실패 영역을 나누는 도면을, 한 페이지로 만들어 둔다. 원본이 어디에 있는가, 사본 1이 어디에 있는가, 사본 2와 오프라인이 어디에 있는가, 키는 어디에 있는가, 각 계층을 올리는 주기와 보관 기간은 무엇인가. 이 도면은 백업 스크립트와 같은 저장소에, 그리고 백업 저장소와 다른 곳에 하나씩 둔다.

이 도면이 없으면, “백업을 하고 있다”는 말은 그 사람의 기억에만 존재한다. 기억은 1장의 마지막 얼굴, 못 읽음과 같은 위험을 공유한다. 사람이 멈추면, 백업도 멈춘다.

## 3.8 1인 시스템의 백업 예산

실패 영역을 나누면 비용이 든다. 그리고 1인 시스템의 예산은 작다. 그래서 모든 계층을 최상이 아니라, "실패 영역이 겹치지 않을 만큼"으로 세운다.

대략적인 감각을 잡아 보자. 원본의 데이터가 50GB라고 하자. 매일 덤프는 증분으로, 50GB 전부가 아니라 바뀐 부분만 옮기면, 일주일 분량이 20GB가량 된다 [추정]. 객체 스토리지의 저비용 보관으로 두면, 한 달 수 GB의 가격에 든다 [추정]. 오프라인이나 다른 계정은 한 달에 한 번, 밀봉해서 옮기므로 거의 비용이 없다.

이 수식이 주는 교훈은, 실패 영역을 나누는 데 드는 돈이 생각보다 작다는 것이다. 3-2-1이 비싸다고 포기하는 팀은, 대개 최신 백업 하나만 들고 같은 실패 영역에 세 번 쓴 셈이다.

## 3.9 1인 SaaS를 위한 하나의 구성

구체적으로 하나의 구성을 잡아 보자.

- 원본: 프로덕션 서버, 지역 A
- 사본 1: 객체 스토리지, 지역 B, 매일, 수명 규칙으로 월 보관으로 전환
- 사본 2: 다른 계정의 저장소, 매주, 월 단위로 보관
- 오프라인: 한 달에 한 번, 별도 매체 또는 별도 계정에 밀봉
- 키: 백업과 다른 실패 영역의 별도 장소
- 런북: 백업과 같은 장소, 같은 주기

이 구성의 공통 분모는, 각 계층이 다른 실패 영역에 있다는 것이다. 지역, 계정, 시간, 매체 중 하나라도 겹치지 않게 배치한다.

이 구성은 규모가 커질 때만 바뀐다. 서비스가 1인에서 2인, 3인이 되면, 실패 영역은 그대로지만 그 영역을 아는 사람이 바뀐다. 그래서 이 도면은 백업이 아니라, 백업에 대한 합의가 된다. 1인 시스템이 성장할 때, 백업

설계가 아니라, 백업의 소유자가 바뀌는 순간을 준비하는 것이다.

## 3.10 오늘 밤에 해 볼 일

지금 백업이 놓여 있는 실패 영역을 하나씩 적어 보라. 원본과 백업이 같은 호스트에 있는가, 같은 지역에 있는가, 같은 계정에 있는가. 이 세 줄을 적다 보니, 어느 줄이 모두 "네"라면, 그 백업은 아직 실패 영역을 나누지 못한 것이다. 그 줄 하나를 이번 주에라도 다른 실패 영역으로 옮겨 보라. 옮기는 비용은 낮지만 옮기지 않은 비용은 다음 사고다.

# 4 되돌릴 수 있다는 것을 증명한다

첫째 주 월요일, 전체 복원 연습을 한다. 프로덕션을 건드리지 않고 새 스크래치 인스턴스에 가장 마지막 백업으로 시스템을 통째로 다시 세운다. 40분이 걸린다. 로그인은 되고 핵심 조회는 원본과 같은 값을 반환한다. 한 건의 쓰기도 정상적으로 저장된다. 연습은 성공했고 그 40분이라는 숫자를 기록에 남긴다.

그리고 세 번째 주 금요일, 스키마 마이그레이션이 문제다. 되돌려야 한다. 이번에는 실전이다. 기록을 펴 본다. 지난 월요일, 40분이면 복원이 가능하다는 것을 증명해 두었다. 절차는 런북에 있다. 원본을 건드리지 않고 그 직전의 백업으로 새 인스턴스를 세우고, 확인을 거치면, 서비스는 돌아간다. 식은땀 대신, 달력을 보는 일만 남았다.

이 대비가 이 장의 전부다. 앞의 세 장은 백업을 만드는 일이었다. 이번 장은, 백업이 진짜로 되돌릴 수 있다는 것을, 사고가 오기 전에 증명하는 일이다. 반복해서 갱신해야 하는 증거가 핵심이다.

## 4.1 복원 연습은 백업과 같은 주기로

결정적인 원칙 하나. 복원 연습은 백업과 같은 주기로, 백업만큼 당연하게 돌아야 한다. 백업만 하고 복원은 필요할 때 하면, 필요할 때가 곧 처음 해 보는 순간이 된다. 그때는 손이 떨리고 시간이 부족하고 문서도 기억도 없다.

복원 연습은 크게 세 단계로 나누어 주기를 달리하자.

- 샘플 파일 복원: 매일. 한 파일만 꺼내, 무결성을 확인한다.
- 표 단위 복원: 주 1회. 데이터베이스의 한 표, 또는 한 구간을 복원한다.

- 전체 시스템 복원: 월 1회. 스크래치 환경에 시스템을 통째로 다시 세운다.

전체 복원이 핵심이다. 파일을 목록으로 확인하는 것은, 있음만 증명할 뿐, 서기름은 증명하지 않는다. 매일의 샘플 복원은, 백업이 열릴 수 있는지 확인하는 것. 주 1회의 표 복원은, 특정 구간이 복원될 수 있는지 확인하는 것. 월 1회의 전체 복원은, 시스템이 다시 설 수 있는지 확인하는 것이다. 세 단계가 다르다.

## 4.2 복원을, 사고의 반대로 설계한다

복원 연습은, 백업 절차의 거꾸로다. 백업이 원본에서 백업으로 데이터를 옮기는 절차라면, 복원은 백업에서 새 원본으로 데이터를 옮기는 절차다. 이 거꾸로를 한 줄씩 써 두면, 그것이 복원 설계다.

구체적으로, 데이터베이스 덤프를 쓰는 시스템이라면, 복원 설계는 다음 순서다. 새 인스턴스를 만든다. 덤프 파일을 그 인스턴스로 옮긴다. 복원 명령을 실행한다. 스키마와 데이터가 들어간 것을 확인한다. 앱을 그 인스턴스에 연결해 띄운다. 핵심 기능을 확인한다. 사고 당시에 따라갈 것이 생긴다.

여기서 중요한 것은, 새 인스턴스다. 복원은 백업 위에서, 새로운 곳에서 다시 세우는 것이다. 원본을 덮어쓰는 복원은, 실패하면 원래 상태마저 잃는다. 그래서 복원 연습은 항상 스크래치 환경, 즉 버려도 되는 곳에서 한다.

## 4.3 스크래치 환경을 만드는 일

전체 복원이 필요하려면, 버려도 되는 환경이 필요하다. 1인 시스템에서 이것을 만드는 것은 생각보다 싸다.

가장 싼 방식은, 복원 전용 인스턴스를 필요할 때만 만들고, 끝나면 지우는 것이다. 월 1회, 한 시간만 쓰는 인스턴스라면, 비용은 거의 없다. 조금 더 편한 방식은, 항상 떠 있는 소형 스크래치 인스턴스를 하나 두는 것이다. 프로덕션보다 작아도 된다. 복원은 프로덕션과 같은 규격을 요구하지 않으므로, 작아도 절차는 증명된다.

중요한 것은, 스크래치 환경이 프로덕션과 같은 실패 영역에 있지 않다는 것이다. 프로덕션과 같은 호스트, 같은 지역, 같은 계정에 스크래치를 두면, 프로덕션이 죽은 상황에서 스크래치도 읽히지 않을 수 있다. 복원 연습은, 3장에서 세운 실패 영역을 거스를 수 있는 곳에서만 의미가 있다.

## 4.4 복원 도구는, 지금도 살아 있어야 한다

1장의 네 번째 얼굴, 못 읽음을 다시 떠올려 보자. 백업 파일은 있지만 그것을 열 수 있는 도구가 사라진 경우다. 복원 연습은 이 얼굴에 대한 유일한 방어다.

도구가 사라지는 방식은 두 가지다. 첫째, 도구가 더 이상 배포되지 않는다. 폐기된 버전, 구할 수 없는 바이너리. 둘째, 도구가 있지만 그 도구가 쓰는 형식이 바뀐다. 구 버전 도구가 새 형식을 읽고 새 버전 도구가 구 형식을 읽지 못하는 경우. 그래서 복원 연습은, 복원 도구가 지금도 동작하는지 확인하는 일이다. 복원 도구의 지속 가능성을 증명한다.

복원 도구는, 백업 도구와 같은 버전, 같은 형식을 유지해야 한다. 백업 도구가 업그레이드되면, 복원 도구도 함께 업그레이드되고, 그 사이에는 복원 연습을 한 번 더 한다. 이 작은 규칙이, 파일은 있는데 열 수 없는 상황을 막아 준다.

## 4.5 복원이 성공한 기준

복원이 성공했다는 말을 쓰기 전에, 다음 네 가지를 확인하자.

첫째, 무결성이다. 체크섬이 맞고 앱이 뜨고 조회가 기대한 데이터를 반환한다. 둘째, 시간이다. 실측된 복원 시간이 목표, 즉 RTO를 넘지 않았다. 셋째, 신선도다. 복원된 데이터가 RPO가 약속한 시점까지 최신이다. 넷째, 기능이다. 복원된 데이터로 서비스가 실제로 동작한다. 앱이 된다.

이 네 가지를 전부 통과해야, 복원이 성공했다고 쓸 수 있다. 하나라도 빠지면, 그것은 시도일 뿐이다.

| 기준 | 확인 방법 | 실전 신호 |
|---|---|---|
| 무결성 | 체크섬, 조회 | 데이터가 맞다 |
| 시간 | 복원 시작 종료 | RTO 이내 |
| 신선도 | 복원 시점 확인 | RPO 이내 |
| 기능 | 핵심 동작 누름 | 앱이 된다 |

## 4.6 자동화: 백업 후에 스스로 확인하게

사람이 매번 확인하는 것은 지치므로, 검증을 자동화하자.

각 백업 직후, 도구의 내장 검증으로 무결성을 확인한다. 덤프 도구는 대부분 복원 가능 여부를 자체 검사하는 옵션을 제공한다. 오늘 백업이 열 수 있다는 것을 기계가 먼저 확인하게 한다.

정기적으로, 스크래치 버킷이나 스크래치 호스트에 자동 복원 테스트를 실행한다. 주 1회, 표 단위 복원과 무결성 검사를 자동화한다. 월 1회, 전체 복원은 사람이 직접 하지만 그 전까지의 점검은 기계가 한다.

실패 시 알림. 백업 실패, 백업이 너무 오래됨, 검증 실패, 복원 연습 기한 초과. 이 네 가지를 알림으로 만든다. 1장에서 본 대로, 조용히 실패한 백업은 없다 것과 같다.

| 자동화 | 주기 | 확인하는 것 |
|---|---|---|
| 무결성 검사 | 매일 | 백업이 열림 |
| 표 단위 복원 | 주 1회 | 구간 복원 |
| 전체 복원 | 월 1회 | 시스템 재구성 |

## 4.7 복원 연습의 기록

복원 연습은, 해 본 것만으로는 충분하지 않다. 언제, 얼마나 걸렸고 무엇을 확인했는지, 무엇이 실패했는지를 기록으로 남겨 두자.

기록에는 다음이 들어간다. 날짜, 복원한 백업의 시점, 걸린 시간, 통과한 기준, 실패한 지점. 이 기록이 쌓이면, 복원 시간이 점점 줄어드는지, 혹은 점점 늘어드는지 보여 준다. 그리고 다음 전체 복원 때, 이전과 무엇이 달라졌는지 비교할 수 있다. 기록이 없는 복원 연습은, 다음에 다시 처음부터 하는 것이다.

구체적으로, 한 달의 기록을 한 장에 모으면 이렇게 생긴다. 월 1일 전체 복원, 42분, 통과. 주 2일 표 복원, 통과. 월 3일 전체 복원, 40분, 통과. 월 1일보다 2분 빠르다. 주 4일 표 복원, 실패. 백업 파일의 경로가 바뀌어 찾을 수 없다.

이 기록에서, 주 4일의 실패가 중요했다. 1장, 2장에 나오지 않은, 실제로 백업이 어딘가로 이동한 일이 드러났다. 연습에서 발견했다. 이 차이가, 전체 복원 연습의 전부다.

## 4.8 복원 실패를 만났을 때

마지막으로, 복원이 안 될 때를 준비하자. 복원 연습에서 실패하는 것은 나쁜 일이 아니다. 사고 전에 실패하는 것이, 사고 중에 실패하는 것보다 훨씬 싸다.

실패를 네 갈래로 분류하자.

- 도구: 복원 도구가 없거나 버전이 다르다.
- 경로: 원본 경로가 바뀌어 찾을 수 없다.
- 데이터: 파일이 손상되어 열리지 않는다.
- 시간: 복원이 목표 시간보다 너무 늦다.

각 갈래의 대응은 1장의 네 얼굴과 다르지 않다. 중요한 것은, 이 분류와 대응을 사고 당시에 처음 생각하지 않게, 런북에 미리 써 놓는 것이다.

## 4.9 복원 런북은 백업 스크립트만큼 중요하다

복구 절차를 문서로 남겨 두자. 어떤 명령으로, 어떤 순서로, 어디에서 무엇을 가져와서 어떻게 다시 세우는가. 이 런북은 백업 스크립트만큼, 어쩌면 그보다 더 중요하다.

이유는, 백업 스크립트는 매일 돌아가며 스스로의 오류를 드러내지만, 런북은 쓰지 않으면 아무도 그 오류를 모른다는 것이다. 런북의 한 줄이 바뀐 경로 때문에 한 시간을 버리는 것을, 한 달에 한 번의 전체 복원이 막아 준다. 런북은 백업의 설계도와 같다. 설계도가 없으면, 백업이 아무리 있어도 다시 세울 수 없다.

## 4.10 복원이 곧 롤백이다

마지막으로, 복원이 왜 배포에도 중요한지 한 마디 하자. 배포를 되돌리는 것, 즉 롤백은, 데이터의 복원이어야 한다. 코드만 되돌리면, 데이터는 새로운 스키마를 가린 채로 남는다. 그래서 배포를 되돌릴 수 있으려면, 데이터베이스를 되돌릴 수 있어야 한다.

이 책의 백업 규율은, 배포의 롤백을 가능하게 하는 토대다. 배포와 운영의 전부다.

## 4.11 되돌릴 수 있다는 것은 상태가 아니라 유지다

이 책의 결론을 한 문장으로 다시 쓰자. 측정되고 증명되고 유지되는 상태다.

그리고 그 상태를 유지하는 비용은 매일 작다. 백업 한 번, 검증 한 번, 주 한 번의 표 복원, 월 한 번의 전체 복원. 이 작은 반복이 쌓이면, 어느 날 새벽 2시에 식은땀을 흘릴 일이 사라진다. 사고가 나도 되돌릴 수 있다는 말이 참이도록 해 주는 것이다.

## 4.12 오늘 밤에 해 볼 일

오늘 밤에, 월 1회 전체 복원의 다음 일자를 달력에 적어 보라. 그리고 복원 절차의 첫 세 줄, 새 인스턴스를 만들고 백업을 옮기고 복원 명령을 실행하는 줄을, 문서의 한 장에 적어 보라. 세 줄이면 된다. 다음 달에 따라갈 것이 생기는 것이다.

그리고 다음 달에, 그 세 줄을 실제로 한 번 써 보라. 써 보는 것이, 이 책의 전부다.