되돌릴 수 있어야 보내는 것: 프로덕션 배포의 기술

빌드, 환경, 파이프라인, 롤백까지: 1인 개발자와 소규모 팀을 위한 프로덕션 배포 실무

ThakiCloud (Hyojung Han)

되돌릴 수 있어야 보내는 것: 프로덕션 배포의 기술

빌드, 환경, 파이프라인, 롤백까지: 1인 개발자와 소규모 팀을 위한 프로덕션 배포 실무

ThakiCloud (Hyojung Han)

Contents

1 1장. 코드가 아니라 빌드가 나간다

코드가 머신을 떠나는 것은 아닙니다. 떠나는 것은 빌드입니다.

코드를 push 하면, 실제로 머신을 떠나는 것은 코드 그 자체가 아닙니다. Git에 들어 있는 것은 텍스트이고 텍스트만으로는 어디에서도 실행되지 않습니다. 그 텍스트를 특정 실행 가능 형태로 바꾸는 작업이 바로 빌드이며, 배포는 그 빌드의 결과물을 프로덕션에 옮겨 실행하는 행위입니다.

이 구분이 학술적으로만 들릴 수 있지만 프로덕션 사고를 다루는 방식은 이 구분이 정해 줍니다.

1.1 밤 10시 이야기

먼저 짧은 이야기 하나. 이건 픽션이 아니라, 너무 흔한 패턴입니다.

밤 9시 40분. 누군가 절반만 고친 기능을 main에 머지합니다. 개발에서는 작동했으니까 괜찮다고 여깁니다.

밤 10시. 여러분은 프로덕션 서버에서 git pull을 하고 의존성을 설치하고, 프로세스를 재시작합니다. 배포가 "완료"되었습니다.

밤 10시 20분. 에러 로그가 차기 시작합니다. 로그를 보고 익숙한 오류를 보고 되돌리려 합니다. 그런데 되돌릴 수가 없습니다. 서버가 그전에 어떤 버전이었는지 아무도 모릅니다. 10시에 설치한 의존성도 최신 버전이라, 1시간 전의 상태 자체가 불분명합니다.

이 과정에서 빠진 것은 코드도, 실력도 아닙니다. 신원이 있는 아티팩트입니다.

1.2 서버에서 빌드하면 안 되는 이유

서버에서 빌드할 때 일어나는 일은 두 가지입니다.

첫째, 실행 대상이 불분명해집니다. 서버에서 git pull하면, 서버는 main 브랜치의 최신 커밋을 가져옵니다. 이 순간 실행되는 것은 "내가 테스트한 버전"이 아니라 "지금 브랜치에 있는 무엇이든"이 됩니다. 밤 9시에 누군가 절반만 고친 기능을 머지했고 밤 10시에 여러분이 배포를 하면, 둘 다 프로덕션이 울리기 시작해서야 알 수 있습니다.

둘째, 실행 환경이 불분명해집니다. 의존성도 서버에서 새로 설치됩니다. 서버의 패키지 매니저 버전, 기반 OS 버전, 네트워크 환경은 개발 머신과 다릅니다. 개발 머신에서 돌아가는 것과 서버에서 돌아가는 것은, 소스 코드만 공유하는 사실은 다른 프로그램 둘이 됩니다.

그래서 첫 번째 규칙은 단순합니다.

프로덕션에서 빌드하지 마세요. 한 곳에서 빌드하고 그 결과물을 가져다 실행하세요.

1.3 아티팩트는 신원이 있는 패키지

빌드의 결과물, 즉 아티팩트는 신원이 필요합니다. 롤백을 하거나 사고를 조사할 때 우리가 묻는 질문은 "어떤 코드가 돌았나요"가 아니라 "어떤 빌드가 돌았나요"입니다. 아티팩트에 신원이 없으면 이 질문에 답할 방법이 없습니다.

모든 아티팩트는 다음 메타데이터를 갖습니다.

- 버전 번호
- Git 커밋 해시
- 빌드 시각

- 빌드한 쪽 (사람이든 파이프라인이든)

Docker 이미지라면 이름이 이렇게 붙습니다.

myapp:1.4.2-3f9a1c2

버전 번호는 사람이 읽기 위한 것이고 커밋 해시는 기계가 검증하기 위한 것입니다. 롤백할 때 "작주 버전"으로 돌아가는 것이 아니라, 특정 이미지 ID를 다시 가리킵니다. 아티팩트가 이렇게 식별되면, 롤백은 "이전 아티팩트를 다시 선택하는 일"로 줄어듭니다.

아티팩트 안에 무엇이 들어가야 하는지도 정해 두어야 합니다.

아티팩트가 담을 것:

- 코드 (소스이거나 컴파일된 형태)
- 실행에 필요한 모든 의존성
- 실행 스크립트 (시작, 종료 방법)

아티팩트가 담지 말아야 할 것:

- 프로덕션 특정 설정 값 (DB 주소, API 키)
- 데이터
- 상태

설정과 데이터는 아티팩트가 아니라 환경이 제공합니다. 이 분리 덕분에 "아티팩트만 바꾸고 환경은 그대로 둔다"는 일이 가능해집니다. 2장과 4장에서 이 원칙을 다시 만나게 됩니다.

아티팩트의 형태는 스택에 따라 달라집니다.

아티팩트	사용 예	주의점
Docker 이미지	컨테이너 서비스	베이스 이미지 고정
tarball	서버리스 함수	체크섬 함께 저장
바이너리	Go, Rust 서비스	정적 링크 확인

1.4 재현 가능한 빌드

신원이 있는 아티팩트가 의미를 갖려면 다시 만들 수 있어야 합니다. 재현 가능한 빌드는, 같은 소스와 같은 설정을 넣으면 같은 아티팩트가 나오는 것을 뜻합니다. 소규모 팀에게는 집착처럼 들릴 수 있지만 실제로는 매일 쓰는 안전장치입니다.

재현 가능한 빌드에는 세 가지가 필요합니다.

첫째, 고정된 의존성입니다. lockfile을 소스 관리에 함께 커밋해야 합니다. package-lock.json, poetry.lock, go.sum은 빌드의 부속산물이 아니라 코드입니다. lockfile이 Git에 없다면, "작주에 사용된 의존성 버전"을 특정할 방법이 없습니다.

둘째, 고정된 빌드 환경입니다. Dockerfile의 FROM python:3.12는 약한 고정입니다. 3.12 태그는 언제든 OS가 바뀔 수 있습니다. 더 안정적인 형태는 구체 태그나 digest까지 고정하는 것입니다. 컨테이너가 아닌 빌드라면, 컴파일러와 빌드 도구의 버전을 어딘가에 기록해 둡니다.

셋째, 결정적인 빌드 순서입니다. 빌드 결과가 파일 처리 순서나 파일시스템의 타임스탬프에 의존하면 안 됩니다. 많은 언어 생태계는 이 문제를 이미 해결해 놓았고, 일부 웹 번들러는 그렇지 않습니다. 번들에 생성 시각을 넣으면 매번 파일 내용이 달라지고, "동일한 빌드인가"를 확인할 길이 사라집니다.

재현 가능한 빌드의 가치는 사고 현장에서 드러납니다. 화요일에 사용자가 버그를 보고했고 수요일에 로컬에서 같은 버전을 다시 구동해봐야 합니다. 커밋 해시로 정확한 아티팩트를 다시 빌드할 수 있다면, 디버깅은 같은 상태에서 시작합니다. 그렇지 않다면, "실제로 돌던 게 뭐였지"를 찾는 데 한 시간을 먼저 쓰게 됩니다.

1.5 버전 매김: 빌드를 어떻게 이름 붙일 것인가

아티팩트에 신원이 있다면, 버전 번호는 그 신원 중 가장 잘 보이는 부분입니다. 소규모 팀에는 현실적인 방식이 두 가지 있습니다.

첫째는 의미 버전(semver), 즉 MAJOR.MINOR.PATCH입니다. 1.4.2처럼. 정기적으로 릴리스를 하고 바깥에 배포를 한다면 좋습니다. 나쁜 점은 사람이 직접 올리기 쉽다는 것, 그리고 그래서 틀리기 쉽다는 것입니다.

둘째는 커밋 해시를 주된 식별자로 쓰는 것입니다. push될 때마다 빌드 번호가 자동 배정되거나, 커밋 해시 짧은 형태를 그대로 씁니다. "지금 이게 몇 버전이지"를 생각할 사람이 사라집니다.

둘 다에는 한 가지 규칙이 공통입니다. 버전 번호는 사람이 아니라 빌드가 매깁니다. 사람이 매기면 실수하고 기계가 매기면 같은 소스에는 항상 같은 번호가 갑니다.

그리고 버전 번호는 한 번 쓰면 재사용하지 않습니다. 1.4.2에서 버그를 찾았다고 1.4.2를 고쳐 다시 쓰는 것이 아니라, 1.4.3을 만듭니다. 이 규칙은 곧 저장소 불변성 규칙과 같은 것입니다.

1.6 빌드 명령: 파이프라인이 없을 때

"빌드를 한 곳에서 하자"는 원칙을 지키려면, 최소한 빌드하는 명령 자체를 파일로 남겨 둬야 합니다. 파이프라인이 없다면, 리포지토리의 build.sh 하나면 됩니다.

#!/bin/sh set -e COMMIT=$(gitrev−parse−−shortHEAD)VERSION =
VERSION) NAME="myapp-$VERSION−${COMMIT}"

1. 의존성 고정 상태로 설치 npm ci

2. 번들/컴파일 npm run build

3. 아티팩트 포장 tar czf “dist/$NAME.tar.gz" -C dist .sha256sum "dist/{N
> “dist/${NAME}.tar.gz.sha256”

4. 메타데이터 기록 echo “{”name”:“$NAME", "$commit" : "{COMMIT}”,“built_at”:“$(date -u+{NAME}.meta.json”

이 스크립트가 만들어 주는 것은 단순히 파일이 아닙니다. 이름(NAME)에 버전과 커밋이 들어 있고 체크섬이 함께 있고 빌드 시각이 기록됩니다. 이 스크립트가 있으면, 파이프라인이 생겼을 때도 그 파이프라인의 “build 단계”가 무엇인지 정해져 있습니다.

여기서 한 가지 유혹을 거절해야 합니다. “빌드는 개발 머신에서, 배포는 서버에서”처럼 중간에 끊는 방식입니다. 빌드와 배포는 같은 기계에서, 같은 순서로, 가능한 한 같은 계정으로 실행되어야 합니다. 중간에 사람이 파일을 우회 전송하면, 신원이 있는 아티팩트라는 약속이 깨집니다.

1.7 아티팩트 저장소

아티팩트가 만들어지면, 그것을 보관할 곳이 필요합니다. 아티팩트 저장소(registry)는 서비스의 모든 빌드가 쌓이고, 배포할 때 고르는 곳입니다. 소규모 팀에서는 이 “곳”이 생각보다 단순해도 됩니다.

- 컨테이너라면 Docker Hub나 클라우드 제공사의 이미지 저장소
- tarball이나 바이너리라면 S3, GCS 같은 객체 저장소
- 언어 패키지라면 사설 레지스트리

저장소에 대한 가장 중요한 규칙은 불변성입니다. 같은 이름을 가진 아티팩트는 절대 바뀌면 안 됩니다. v1.4.2가 잘못되었다면, 그 내용을 바꾸는 것이 아니라 새 번호를 줍니다. 이유는 명확합니다. 프로덕션이 지금 v1.4.2를 가리키고 있을 수 있고 그 내용이 조용히 바뀌면 롤백할 곳이 사라지기 때문입니다.

또한 저장소는 "지금 뭘 배포할 수 있는지"가 한눈에 보여야 합니다. 서비스별로 최근 빌드가 시간순으로 보여야 하고 각 빌드 옆에 커밋, 시각, 누가 배포했는지가 붙어야 합니다. "지금 존재하는 버전이 뭐야?"를 30초 안에 답하지 못한다면, 그 저장소는 역할을 하지 못하고 있습니다.

또 하나, 보존 기간이 필요합니다. 모든 빌드를 영원히 둘 필요는 없지만 "지금 도는 것"과 "최근 몇 개"는 반드시 남겨 둡니다. 보통은 릴리스된 것을 수개월간 두고 나머지를 지우는 식입니다 [추정]. 아티팩트를 지우는 순간은, 그 버전으로 되돌릴 수 있는 권리를 포기하는 순간입니다.

1.8 서비스가 하나뿐인 때

소규모 팀의 흔한 질문이 있습니다. "서비스가 하나뿐인데, 이걸 전부 해야 하나요?" 네, 해야 합니다. 그리고 오히려 쉽습니다. 범위가 작으니까요.

최소 구성은 이렇습니다.

빌드는 CI 기계(GitHub Actions 같은 것)에서 합니다. 개발 머신에서, 서버에서 하는 것이 아닙니다. 서비스가 하나뿐이어도, 빌드가 별도의 장소에서 일어나는 순간 아티팩트의 신원이 의미를 갖습니다.

저장소는 버킷 하나면 됩니다. 서비스 이름과 버전 키로 정리한 S3 버킷이 바로 아티팩트 저장소입니다.

배포는 버킷에서 아티팩트를 내려와 시작하는, 한 줄의 명령입니다.

이 최소 구성으로 얻는 것은 세 가지입니다. "지금 도는 버전"을 알 수 있고 "이전 버전"을 다시 가져올 수 있고 "롤백"이 한 줄의 조작이 됩니다. 도입 비용은 며칠이고 효력은 서비스의 수명만큼 갑니다.

1.9 지금 그만둬야 할 습관

이 장을 읽다가, 현재 프로덕션을 잠깐 떠올려 보세요. 소규모 팀에서 흔한 습관들이고 하나씩 끊는 비용은 낮습니다.

- 서버 위에서 코드를 직접 고치는 일 (주석 한 줄이어도)
- 프로덕션 설정 파일을 손으로 바꾸는 일
- lockfile 없이 서버에서 의존성을 설치하는 일
- 버전 번호를 손으로 올리는 일
- 배포 대신 재시작으로 문제를 넘기는 습관

이것들은 하나씩 보면 작은 지름길입니다. 하지만 다 합치면, "지금 프로덕션의 상태"를 설명할 수 없는 상태가 됩니다. 설명할 수 없는 상태는, 되돌릴 수 없는 상태입니다.

1.10 이 장의 연습

지금 서비스에서 이번 주에 다음 세 가지만 해 보세요.

1. 마지막 배포한 빌드에 신원을 부여합니다. tarball이든 바이너리든, 파일명에 커밋 해시와 빌드 시각을 붙입니다.
2. "지금 도는 아티팩트가 어디 있고 이전 것은 어디 있는가"를 한 줄로 적습니다. 답하지 못한다면 저장소부터 시작합니다.
3. lockfile을 Git으로 옮깁니다. 없다면 만들어 넣고 있는 것이라도 다시 생성해 확인합니다.

세 가지 모두 작습니다. 하지만 다음 배포부터 "지금 뭐가 돌고 있나"라는 질문에 답이 생깁니다.

1.11 이 장의 요점

배포는 “머신을 떠나는 것”을 신원이 있는 패키지로 대하는 순간부터 시작됩니다. 빌드가 한 번, 한 곳에서 일어나고 그 결과가 식별되고 이름이 붙고 언제든 다른 결과물로 바꿀 수 있다면, 그 다음 세 장이 서는 기초가 단단해집니다.

2 2장. 환경은 장소가 아니라 계약이다

사람들은 환경을 기계로 떠올립니다. 개발 서버, 스테이징 서버, 프로덕션 서버. 하지만 코드에게는 기계가 보이지 않습니다. 코드가 보는 것은 대화할 수 있는 서비스 목록, 읽을 수 있는 값, 건드릴 수 있는 데이터입니다. 환경을 "계약"으로 보면, 신경 써야 할 것이 달라집니다. "서버가 같은가"가 아니라, "코드가 같은 질문을 했을 때 같은 답을 듣는가"가 핵심입니다.

먼저 구체적인 예로 형태를 잡아 봅시다.

2.1 작은 서비스의 세 환경

소규모 SaaS를 떠올려 보세요. 웹 앱 하나, 데이터베이스, 이메일을 보내는 백그라운드 워커가 있는 서비스.

개발 환경은 개발자 노트북입니다. DB는 로컬, 이메일 워커는 메일을 보내지 않고 콘솔에 인쇄합니다. API 키는 테스트 키입니다.

스테이징은 클라우드 VM 하나입니다. DB는 프로덕션과 같은 제공사의 매니지드 서비스지만 작은 규격입니다. API 키는 진짜이지만 테스트 모드로 발급된 것입니다.

프로덕션은 실제 사용자의 데이터가 있는 곳입니다.

세 곳의 코드는 같은 빌드입니다. 다른 것은 환경이 주는 값들. DB 주소, API 키, 이메일 워커의 "버리기 아니면 보내기", 로그 레벨. 이것이 계약의 형태입니다.

이제 질문입니다. 환경은 몇 개면 좋은가.

흔한 답은 셋입니다. 로컬 개발, 스테이징, 프로덕션. 그런데 1인 개발자에게 스테이징은 사치에 가깝습니다. 비용도 들고 관리도 해야 하고 방치하면

프로덕션과 멀어집니다.

그래서 실질적인 질문은 이것입니다. 어떤 계약은 따로 둔 환경 없이는 검증할 수 없는가.

통상 다음 세 가지가 해당됩니다.

- 프로덕션에만 존재하는 설정 (실제 API 키, 실제 DB 연결)
- 데이터 마이그레이션의 동작
- 실제 트래픽에서만 나타나는 지연과 부하

서비스가 돈을 다루지 않고 마이그레이션이 작다면, 스테이징을 생략해도 됩니다. 대신 배포 직후 검증 스크립트를 돌리고 되돌리는 시간이 짧은 방식으로 배포하세요. 4장에서 더 다룹니다. 반대로 코드에 돈, 개인정보, 긴 마이그레이션이 걸려 있다면 스테이징이 필요합니다. 프로덕션의 복사본으로서가 아니라, 같은 모양의 설정을 쓰고 같은 빌드를 돌리고, 같은 테스트를 거치는 곳으로서입니다.

2.2 설정과 시크릿: 어디에 무엇을 둘 것인가

환경 계약의 핵심 원칙은, 코드는 같고 환경이 다르다는 것입니다. 설정 값은 코드 안에 있는 것이 아니라, 환경이 시작할 때 주는 입력이어야 합니다. 12-factor app이 말하는 정신이기도 하지만, 이름보다 기계가 중요합니다.

소규모 팀에서는 설정을 세 바구니로 나누면 잘 동작합니다.

구분	예시	보관 곳
빌드 시 상수	기능 이름	코드
런타임 설정	DB 주소	환경 변수
시크릿	API 키	시크릿 매니저

구체적인 기계는 다음과 같습니다.

런타임 설정은 전부 환경 변수에서 읽습니다. 변수 이름과 기본값을 매핑하는 파일은 리포지토리에 커밋합니다. 예컨대 서비스에게 필요한 모든 변수를 빈 값으로 나열한.env.example을 두고 변수가 하나 바뀌면 그 파일도 같이 바뀌게 하는 것입니다. 새로운 동료, 혹은 다음 달의 본인이 그 파일만 보면 서비스가 무엇을 요구하는지 알 수 있습니다.

프로덕션의.env 파일은 Git에 넣지 않습니다. 꼭 넣어야 한다면, 파일을 암호화하는 도구로 감싸서 암호화된 산출물만 커밋합니다.

CI에서 쓰는 시크릿은 플랫폼 측에 두어, 배포 단계에서만 주입합니다. 이렇게 하면 시크릿이 빌드 산출물이나 로그에 남지 않습니다.

시크릿은 주기적으로 회전해야 합니다. 최소한 “키 하나를 바꾸는 절차”를 알아야 합니다. 절차를 쓰지 않았다면, 그 날이 오면 당황하게 될 것입니다.

흔한 실수는 코드에 환경 이름을 넣는 것입니다. if env == “prod” 같은 분기는, 개발과 프로덕션에서 같은 코드가 돌아도 행위가 달라지는 곳이 됩니다. 대신 설정 값만 바꾸고 코드는 같은 경로를 걷게 하세요. 환경 분기를 쓰게 된다면 한 번 멈추고, “이 설정은 각 환경에서 어떤 값이어야 하나”를 먼저 물어 보세요.

2.3 데이터베이스: 가장 바꾸기 어려운 계약

계약 중 데이터가 가장 바꾸기 어렵습니다. 코드는 몇 초 만에 바꿀 수 있지만 DB의 데이터는 몇 달 쌓인 것입니다. 잘못된 스키마 변경은 “이전 아티팩트를 다시 가리킨다”로는 되돌릴 수 없는, 드문 배포 실수입니다.

이에 대한 표준적인 답은 expand/contract 패턴입니다.

구체적인 예로 살펴봅시다. users 테이블에 email_verified_at 컬럼을 추가해야 한다고 하면, 한 방에 “추가하고 옛것 지우기”가 아니라 이렇게 갑니다.

확장 단계: 컬럼을 추가합니다. nullable로 두고 기존 구조는 손대지 않습니다. 새 코드를 배포합니다. 새 코드는 기존 JSON 필드에 email_verified가 있으면 읽고, 쓰기는 옛 필드와 새 컬럼 둘 다에 하게 합니다. 이 순간, 옛 코드와 새 코드가 같은 테이블 위에서 함께 삽니다.

백필: 기존 데이터를 새 컬럼으로 옮깁니다. 큰 테이블이면 배치로, 백그라운드 잡으로, 테이블을 잠그지 않는 방식으로 합니다. 며칠에 걸쳐 돌아가도록 해도 됩니다.

축약 단계: 옛 필드를 읽는 모든 쪽이 새 컬럼으로 옮긴 뒤에, 옛 필드를 제거합니다. 같은 릴리스가 아니라, 다음 릴리스에서 합니다.

세 단계로 나누는 이유는, 언제든 "코드 롤백"과 "스키마 롤백"이 둘 다 가능하게 하려는 것입니다. "컬럼을 추가하고 옛 것을 지운" 단일 릴리스는, 한 번 적용되면 되돌릴 길이 사라집니다.

마이그레이션에서 하지 말아야 할 것도 분명합니다.

큰 테이블에서 긴 ALTER: 테이블을 잠그게 되면, 반드시 피크 시간에 걸립니다. 조작을 쪼개서 배치로 실행합니다.

데이터를 지우는 파괴적 작업: 컬럼을 drop할 수 있다면, 프로덕션에서는 drop하지 마세요. 코드가 쓰지 않는 것으로 표시해 두고 끝냅니다. 지우는 행위는 다음 릴리스, 혹은 확실해진 시점으로 미룹니다.

앱의 상태에 의존하는 마이그레이션: 마이그레이션은 단독으로 실행될 수 있어야 합니다. 단독으로 안 돌아가면, 마이그레이션과 코드 변경이 엮여 있다는 뜻입니다. 다시 나눠 보세요.

한 가지 더, 스냅샷이 있습니다. 개발 환경에 프로덕션 모양에 가까운 데이터를 가져오면, 테스트의 힘이 크게 달라집니다. 전부 복사할 필요는 없습니다. 개인정보를 익명화하고 필요한 부분만 떼어 내는 덤프면 충분합니다. 소규모 서비스라면 한 달에 한 번 만드는 정도가 현실적인 리듬입니다 [추정].

2.4 환경 드리프트: 스테이징과 프로덕션이 다른 말을 할 때

환경을 둘 이상 유지하면, 어느 순간부터 어긋나기 시작합니다. 의도적으로가 아니라, 프로덕션 DB가 먼저 업그레이드되고 스테이징에만 도메인이 붙고 설정 값이 한 달간 안 바뀌어 있는 식으로 어긋납니다. 그리고 사고가 나면 첫 의심은 항상 "스테이징에선 됐던 건데"가 됩니다.

드리프트는 성실함의 문제가 아니라 구조의 문제입니다. 두 환경을 다른 방식으로 관리하고 있는 한, 어긋남은 시간문제입니다. 그래서 답은 "신경 쓰세요"가 아니라 "두 환경을 같은 방식으로 만들게 하세요"입니다.

세 가지 규칙이면 충분합니다.

첫째, 인프라를 코드로 씁니다. 서버, DB, 도메인 모두 스크립트(Terraform, CloudFormation, Pulumi 등)가 만듭니다. 스크립트가 완벽하지 않아도, 환경을 diff로 말할 수 있다는 것만으로도 사고 조사의 방식이 달라집니다.

둘째, 같은 빌드를 씁니다. 스테이징과 프로덕션은 저장소의 같은 아티팩트를 끌어와 실행해야 합니다. 스테이징은 main의 빌드를, 프로덕션은 tag의 빌드를 돌리면, 테스트한 것과 배포한 것이 다른 프로그램입니다.

셋째, 같은 테스트를 씁니다. 배포 후 검증에 쓰는 스모크 테스트는 두 환경에서 모두 돌아야 합니다. 스테이징에서만 통과하는 테스트는 프로덕션을 보증하지 못합니다.

2.5 스테이징의 형태: 건너뛰지 않을 만큼 싸게 만드는 법

스테이징을 건너뛰게 만드는 것은 기능 부족이 아니라 비용입니다. 프로덕션급으로 만들면 비싸고 너무 작게 만들면 신뢰할 수 없습니다. 균형을 잡는 실용적인 형태는 이렇습니다.

첫째, 빌드는 프로덕션과 같은 아티팩트를 씁니다. 저장소에서 같은 이미지, 같은 tarball을 끌어옵니다. 여기서 절약할 것은 없습니다.

둘째, 규격은 작게 합니다. DB는 프로덕션의 반 규격이면 충분합니다. 트래픽도 프로덕션의 일부이지 않으니까요. 절약이 일어나는 곳은 규격입니다.

셋째, 값은 진짜 모양으로 합니다. API 키는 테스트 모드의 진짜 키, 도메인은 staging.로 시작하는 진짜 도메인, 이메일 워커는 실제로 보내되 수신자가 팀뿐인 설정. 값의 "진짜 여부"는 규격보다 중요합니다.

넷째, 데이터는 위 스냅샷을 씁니다. 프로덕션 덤프의 익명화 버전이 있으면, 스테이징은 "작은 프로덕션"이 아니라 "같은 프로덕션"이 됩니다.

이 네 가지만 지켜도, 스테이징의 월 비용은 프로덕션의 아주 작은 부분이 됩니다 [추정: 규격에 따라 다름]. 대신 얻는 것은, 프로덕션에서야 알 일을 한 단계 앞에서 막아 주는 자리입니다.

2.6 배포의 순서: 계약의 변경은 순서가 있다

환경 계약의 변경에는 순서가 있습니다. 순서를 지키면 배포는 가역적이고 순서를 어기면 되돌릴 길이 사라집니다.

가장 안전한 순서입니다.

1. 스키마 확장 (새 컬럼 추가, nullable)

2. 아티팩트 배포 (새 코드, 옛 형식과 새 형식 둘 다 다룸)
3. 트래픽 확인, 백필 실행
4. 다음 릴리스에서 축약 (옛 구조 제거)

여기서 중요한 것은, 1과 2가 서로를 필요로 하지 않는다는 점입니다. 스키마만 먼저 바뀌어도 옛 코드는 잘 돌고, 코드만 먼저 바뀌어도 (새 컬럼이 없으면) 옛 경로로 잘 돕니다. 각 단계가 단독으로 안전해야, 그 중간에서 롤백이 가능합니다.

반대로 위험한 순서입니다.

1. 스키마 축약 (옛 컬럼 제거)
2. 아티팩트 배포 (새 컬럼만 읽는 새 코드)

1이 먼저 나가는 순간, 1 이전의 코드는 돌 수 없습니다. 즉, 새 코드를 배포하기 전에 롤백할 곳이 사라진 것입니다. "축약"은 항상 "확장된 것을 충분히 배포하고 나서"의 일입니다.

설정 변경도 같은 순서를 따릅니다. 먼저 환경에 새 값을 놓고 (옛 코드에게는 무시되는 값), 그 다음에 새 코드를 올립니다. 새 코드만의 값을 요구하는 순간이 오기 전에, 값은 이미 해당 환경에 있어야 합니다.

2.7 계약이 깨졌을 때: 환경 관련 버그의 조사법

코드 버그가 아니라 "환경이 달라"서 나는 버그는, 모양이 특징적입니다. 로컬에서는 되고 스테이징에서는 되고 프로덕션에서는 안 되는 것. 혹은 그 반대.

이런 버그의 조사는 두 단계면 됩니다.

첫째, 값을 diff합니다. 환경이 주는 값들이 실제로 같거나, 다른지 확인합니다. 같은 변수 이름이라도 값이 다른 것이 가장 흔한 원인입니다. 변수가 누락된 것, 기본값이 각 환경마다 다른 것도 포함됩니다.

둘째, 스키마를 diff합니다. DB의 구조가 각 환경에서 같은지 확인합니다. 컬럼 하나가 늦게 적용된 것이 원인인 경우도 흔합니다. 마이그레이션이 "프로덕션에서는 돌았는데, 스테이징에서는 아직"인 식의 어긋남입니다.

두 diff를 할 수 있는 전제 조건은, 1장에서 말한 것과 같습니다. 각 환경이 "무엇을 쓰고 있는지"를 기계가 말할 수 있어야 합니다. 인프라가 코드이고 마이그레이션이 기록되고 설정이 문서화되어 있을 때만, diff가 가능합니다. 환경 관련 버그가 늘어난다면, 먼저 코드 버그를 찾지 말고 이 두 diff를 가능한 방향으로 환경을 고치세요.

2.8 이 장의 연습

현재 서비스의 설정을 세 바구니에 나눠 보세요. 빌드 시 상수, 런타임 설정, 시크릿. 각각이 어디에 보관되는지도 적습니다. 적는 순간, 바구니가 틀린 것이 보입니다. 예컨대 코드 안에 있는 API 키, 혹은 환경 변수로만 존재해서 문서화도 안 된 DB 주소 같은 것.

다음으로, 가장 바꾸기 어려운 계약 두 가지를 확인하세요. 이 서비스에 마이그레이션이 있는가. 있다면 expand/contract 형태인가, 아니면 "추가와 제거가 한 번에" 형태인가. 후자라면 다음 마이그레이션부터 바꿉니다.

2.9 이 장의 요점

환경은 코드가 듣는 답들의 집합입니다. 코드도, 빌드도 같게 두고 답만 다르게 하면, 배포는 예측 가능해집니다. 그리고 예측이 가장 잘 깨지는 두 곳은 설정과 데이터 마이그레이션입니다. 파이프라인을 설계하기 전에, 이 둘을 먼저 정리해 두세요.

3 3장. 파이프라인: 사람이 아니라 순서가 배포한다

손으로 하는 배포는, 릴레이 경주에서 바통 인계 규칙이 없는 것과 같습니다. 모든 사람이 집중하고 있는 동안은 잘 돌아가고, 한 번의 허술이 그대로 사고가 됩니다. 파이프라인은 "어떤 아티ifacts를, 어떤 환경에, 어떤 순서로"를 사람의 기억에서 실행 가능한 파일로 옮기는 기계입니다.

파이프라인은 결국 단계의 나열입니다. 가치는 단계의 복잡함이 아닌, 각 단계에 통과 기준이 있고 전체가 하나의 명령으로 다시 돌아간다는 사실에서 옵니다.

3.1 파이프라인의 뼈대

전형적인 파이프라인은 이 순서를 갖습니다.

1. 린트와 정적 분석
2. 단위 테스트, 통합 테스트
3. 빌드와 아티팩트 생성
4. 아티팩트를 저장소에 push
5. 환경에 배포
6. 배포 후 검증
7. 알림

모든 단계가 필요하지는 않습니다. 작은 서비스라면 "테스트, 빌드, 배포, 검증"만으로도 충분합니다. 하지만 순서가 중요합니다. 빌드 뒤에 테스트를 하는 것은 의미가 없고 아티팩트를 저장소에 올리기 전에 배포하면 롤백할 곳이 사라집니다.

대략적인 모습은 이렇게 생겼습니다.

jobs: test: steps: - run: npm ci && npm test build: needs: test steps: - run: docker build -t myapp:$COMMIT .push : needs : buildsteps : —run : dockerpushregistry.example/myapp :COMMIT deploy: needs: push environment: production steps: - run:./deploy.sh $COMMIT - run:./smoke.sh

여기서 deploy가 push를 필요로 하는 것이 중요합니다. "저장소에 없는 아티팩트는 배포할 수 없다"는 수호 장치입니다. 그리고 environment: production 한 줄은, 플랫폼 기능으로 사람 한 번의 승인을 요구할 수 있는 곳입니다. 소규모 팀에게는 이 "사람 승인" 한 칸이 가장 싸고 확실한 리스크 통제가 됩니다.

한 가지 더 넣어야 할 것이 있습니다. 스테이징 배포입니다. 스테이징이 있다면, 같은 파이프라인이 먼저 스테이징으로 배포하고 스모크를 통과한 뒤에 프로덕션 승인으로 넘어가는 것이 좋습니다. 이렇게 하면 "프로덕션에 올라가는 빌드"와 "스테이징에서 방금 통과한 빌드"가 같은 아티팩트가 됩니다.

3.2 각 단계의 통과 기준

단계가 나열되어 있는 것만으로는 파이프라인이 아닙니다. 각 단계가 "통과했다"는 것을 어떻게 판단하는지가 파이프라인입니다.

테스트 단계: 실패한 테스트가 하나라도 있으면 실패입니다. 스킵된 테스트가 쌓이는 순간, 이 기준은 무너집니다.

빌드 단계: 산출물이 만들어졌고 이름에 커밋이 들어 있고 저장소에 이미 같은 이름의 아티팩트가 없으면 실패입니다. 같은 이름 덮어쓰기는 1장의 불변성 규칙 위반입니다.

push 단계: 저장소에서 그 아티팩트를 다시 pull할 수 있어야 통과입니다. push 성공 메시지만 보고 넘어가면 안 됩니다.

배포 단계: 헬스체크가 통과한 뒤에만 배포를 "성공"이라 합니다. 프로세스가 올랐다고 성공이 아닙니다.

검증 단계: 스모크 테스트가 통과해야 성공입니다. 실패하면 파이프라인은 실패 상태로 멈추고 알림이 나갑니다.

이 기준들을 다 합치면, 파이프라인은 "성공했다"는 말이 나올 때마다, 프로덕션이 실제로 새 버전으로, 실제로 돌아간다는 보장을 갖게 됩니다.

3.3 배포 단계의 세 가지 선택지

아티팩트가 저장소에 도착하면, "돌고 있는 것과 어떻게 바꿀 것인가"가 전략의 문제가 됩니다. 흔한 선택지는 세 가지입니다.

롤링 배포: 프로세스를 하나씩 바꿉니다. 인스턴스 하나를 내리고 새 아티팩트를 올리고 헬스를 확인하고 다음으로 갑니다. 리플리카가 여러 개인 서비스에 맞고 단순합니다. 약점은 교체하는 도중에 옛 코드와 새 코드가 같은 트래픽을 나눈다는 것인데, 둘이 함께 살아갈 수 있어야 합니다. 4장에서 더 다룹니다.

블루/그린: 같은 스택을 두 개 두고 트래픽은 항상 한 쪽에만 흘려 보냅니다. 지금 도는 쪽(블루) 옆에 새 것(그린)을 준비하고, 검증이 통과하면 트래픽을 한꺼번에 전환합니다. 전환 후에는 블루를 잠시 남겨 두다가 확신이 서면 내립니다. 롤백이 "트래픽을 다시 뒤집는 것"으로 끝난다는 점에서, 소규모 팀에게 가장 친숙한 형태입니다.

캐너리: 트래픽의 5퍼센트만 새 아티팩트로 보냅니다. 지표를 보고 이상이 없으면 비율을 올리고 문제가 있으면 0으로 돌립니다. "어떤 사용자가 새 코드를 탔는지"를 볼 수 있는 계측이 있을 때만 의미가 있는 방식이며,

소규모 팀에는 보통 기계가 너무 큽니다.

선택 기준은 의외로 단순합니다.

전략	필요 조건	롤백의 모양
rolling	리플리카, 헬스체크	개별 재배포
blue/green	두 스택 여력	트래픽 되돌리기
canary	트래픽 분할, 지표	비율 0%

인스턴스가 하나인 서비스라면, 블루/그린을 단순화할 수 있습니다. 같은 머신에서 옛 것, 새 것을 서로 다른 포트에 올리고 검증이 지나면 리버스 프록시(nginx, HAProxy, 로드 밸런서) 한 줄만 바꿔 겁니다. 두 대의 서버만큼 정밀하지는 않지만 "먼저 검증하고 나중에 전환한다"는 논리는 같습니다.

예를 들어, nginx를 쓰는 단일 머신이면 전환 절차는 이 모양입니다.

1. 새 아티팩트를 끌어와 새 포트에 올립니다
2. 새 포트에서 스모크 테스트를 돌립니다
3. nginx upstream을 옛 포트에서 새 포트로 바꾸고 reload합니다
4. 몇 분을 관찰합니다 (에러 로그, 지연)
5. 문제가 있으면 upstream을 다시 뒤집고 없으면 옛 것을 내립니다

사용자 트래픽을 실제로 건드리는 것은 3번 한 줄입니다. 나머지는 전부 준비와 검증입니다. 이 모양이 바로, 블루/그린이 "서비스를 바꾸는 것"이 아닌 "트래픽을 바꾸는 것"인 이유입니다.

3.4 배포 후 검증: 살아 있는지를 확인하는 순간

"배포가 성공했다"는 "서비스가 건강하다"를 뜻하지 않습니다. 프로세스는 올랐고 포트는 열려 있는데, 첫 요청에서 500이 날아올 수 있습니다. 배포

후 검증은 이 틈을 막는 단계입니다.

두 레벨이 있습니다.

헬스체크: 프로세스가 살아 있는가, 포트가 열려 있는가, 간단한 요청에 응답하는가. 이것은 liveness 확인이며 배포 도구가 이 단계를 완료하기 전에 기다립니다.

스모크 테스트: 핵심 경로가 실제로 동작하는가. 테스트 계정으로 로그인하고 메인 API를 두세 번 호출하고 DB가 읽히는지를 확인합니다. 몇 줄짜리 스크립트면 충분합니다. 중요한 것은, 모든 환경에서 같은 스크립트를 돌린다는 것입니다.

#!/bin/sh # smoke.sh set -e curl -fsS “$BASE/healthz"|grep -q '"ok"' curl -fsS -X POST "BASE/api/login” -d “$TEST_CREDENTIAL /dev/null curl -fsS "BASE/api/orders” | head -c 100 echo “smoke passed”

이 스크립트가 실패할 때는 두 가지 이유뿐입니다. 새 빌드가 깨졌거나, 환경이 변했거나. 둘 다 배포 단계에서 알아야 할 일이지, 다음 아침에 알아야 할 일이 아닙니다.

통과 기준을 정할 때 관대하지 마세요. “뭐든 답하면 됨”은 기준이 아닙니다. “핵심 경로가 특정한 상태 코드와 형식으로 돌아옴”이 기준입니다. 기준이 느슨할수록, 사고가 “어느 누가 눈치 챌 때까지” 밀려납니다.

스모크 테스트 외에도, 배포 후 몇 분간 지표를 봅니다. 거창한 대시보드 대신, 숫자 셋이면 됩니다. 5xx 에러율, 요청 지연, 백그라운드 작업 큐 길이. 셋 중 어느 하나라도 배포 전보다 나빠지는 방향이라면, 그것은 “배포가 끝난 것”이 아닌 “검증 중인 것”입니다.

3.5 처음 파이프라인을 만들 때

아무런 파이프라인이 없는 상태라면, 처음부터 완성형을 만들지 마세요. 두 가지부터 시작합니다.

첫째, 아티팩트 push입니다. 빌드가 끝나면 저장소에 자동 push되는 단계 하나. 이 단계 하나로 "롤백할 곳"이 저장되기 시작합니다.

둘째, 배포 후 스모크 테스트입니다. 배포가 끝난 뒤 핵심 경로를 호출하고 실패하면 nonzero로 끝나는 스크립트. 처음에는 손으로 돌려도 괜찮습니다. 파이프라인 안으로 넣는 것은 다음 일입니다.

이 두 가지만으로도 배포의 대부분이 달라집니다. 그 뒤에 차례로, 테스트 단계, 린트, 알림, 스테이징 배포를 더해 가세요. 파이프라인은 한 번에 만들어지지 않습니다. 코드처럼 한 단계씩 자랍니다.

그리고 첫 파이프라인은 "스테이징 배포"에 걸어야 합니다. 실수해도 프로덕션이 아닌 스테이징이 깨지는 곳에서, 파이프라인을 익히는 것입니다. 파이프라인이 안정적으로 돌아간다는 확신이 섰을 때, 프로덕션을 만지게 합니다.

3.6 단계가 실패할 때

실패하지 않는 파이프라인은 없습니다. 설계해야 하는 것은 실패했을 때입니다.

첫째, 실패를 보이게 합니다. 파이프라인은 끝나면 알림을 보내야 하고 특히 실패하면 보내야 합니다. Slack이든 메일이어든, 사람에게 도달해야 합니다. 조용히 실패하는 배포가 가장 나쁩니다. 프로덕션은 옛 버전으로 돌고 팀은 새 버전으로 알고 있기 때문입니다.

둘째, 자동 롤백의 범위를 정합니다. 소규모 팀의 안전한 기본값은

"자동으로 멈추고 사람이 롤백한다"입니다. 파이프라인이 실패한 단계에서 멈추고 알림이 오고, 사람이 로그를 보고 롤백 명령을 칩니다. 저장소에 이전 아티팩트가 있으니까, 그 명령은 한 줄입니다. 확신이 쌓이면 단계적으로 올립니다. 예컨대 "스모크 테스트가 실패하면 자동 롤백"처럼.

셋째, 런북을 둡니다. 런북은 한 페이지짜리 문서입니다. 로그를 어디서 보다, 롤백은 무엇으로 하느냐, 되돌렸는지를 어떻게 확인하느냐, 누구에게 말하느냐. 서두르는 순간에만 읽을 문서이므로, 짧고 정확해야 합니다. 런북은 파이프라인과 같은 리포지토리에, 옆에 둡니다.

3.7 피해야 할 파이프라인의 모양

설계하다가 빠지기 쉬운 모양이 몇 있습니다.

단계가 너무 많은 배포 파이프라인: 추가하는 단계마다 실패할 곳이 하나 더 생깁니다. 20단계인데 어떤 단계가 필수인지 모르겠다면, 절반을 지워 보세요.

한 방향으로만 가는 파이프라인: "배포"는 있는데 "되돌리기"가 한 명령이 아니라면, 그 파이프라인은 편도 티켓입니다. 롤백도 실행 가능한 단계여야 합니다.

사람의 로컬 머신에 의존하는 단계: "우리" 머신에서만 돌아가는 단계가 있으면, 그 단계는 여러분이 쉬는 순간 막힙니다. 모든 단계는 CI 환경에서 돌아야 합니다.

성공지시에만 알림이 가는 파이프라인: "배포 성공" 알림은 보너스이고, "배포 실패" 알림이 필수입니다. 실패 알림이 없으면, 파이프라인은 여러분이 보지 않는 곳에서 조용히 멈춥니다.

3.8 전체를 돌리는 한 줄

파이프라인이 완성되면, "배포를 한다"는 한 줄이 됩니다.

./deploy.sh

이 한 줄이 안에 넣는 것은, 앞에서 다 본 것들입니다. 테스트를 돌리고 빌드하고 아티팩트를 push하고 배포하고 스모크를 돌리고 알림을 보냅니다. 이 한 줄이 가능한 조건은 두 가지입니다.

첫째, 모든 단계가 멱등(idempotent)이라는 것입니다. 같은 명령을 두 번 실행해도, 세 번 실행해도 결과는 같습니다. 이미 push한 아티팩트를 다시 push해도 무해하고, 이미 배포된 것을 다시 배포해도 무해합니다. 멱등이 아니면, "중간에 끊겼다"는 상태가 생기는데, 그 상태를 사람이 머리로 기억하게 됩니다.

둘째, 모든 단계를 CI 환경에서 돌릴 수 있다는 것입니다. 여러분의 노트북이 없어도, 이 한 줄이 돌아야 합니다.

이 한 줄이 생기고 나서, 배포는 "이벤트"에서 "명령"으로 바뀝니다. 이벤트는 긴장하고 명령은 반복합니다. 긴장을 반복으로 바꾸는 것이 파이프라인의 본질입니다.

그리고 이 한 줄은 런북의 롤백 명령과 짝이 됩니다. 배포는./deploy.sh, 롤백은./rollback.sh. 둘 다 한 줄이고 둘 다 저장소의 아티팩트에서 동작합니다.

3.9 이 장의 연습

지금 배포 절차를 종이에 그려 보세요. "지금, 손으로, 어떤 순서로 무엇을 하는가"입니다. 각 단계에 "입력, 산출물, 통과 기준"을 적어 보세요. 통과 기준을 적지 못 하는 단계가, 기계화가 가장 먼저 필요한 곳입니다.

스테이징이 있다면, “스테이징 배포 후 스모크” 단계를 파이프라인에 넣으세요. 프로덕션 배포가 여전히 손이라면, 최소한 “아티팩트 push”와 “스모크”를 명령으로 만드세요.

3.10 이 장의 요점

파이프라인은 배포를 사건에서 절차로 바꿉니다. 각 단계에는 입력이, 산출물이, 통과 기준이 있습니다. 그리고 파이프라인을 가장 크게 정의하는 순간은 둘입니다. 아티팩트가 저장소에 도착한 순간, 그리고 스모크 테스트가 통과한 순간. 이 둘을 지키면, 배포의 대부분을 지킨 것입니다.

4 4장. 롤백은 실패가 아니라 설계다

많은 팀에서 롤백은 비상 조치입니다. “깨졌으니 돌아가자.” 하지만 배포를 설계할 때, 롤백은 본래 경로에 속한 것입니다. 배포는 시작할 때부터 롤백할 곳이 정해져 있어야 합니다. 그래서 질문은 “얼마나 빨리, 얼마나 깨끗하게 되돌릴 수 있을까”입니다.

이 장은 롤백을 깨끗하게 만들기 위해 무엇을 설계해야 하는지를 다룹니다.

4.1 롤백은 되돌리는 것이 아니라, 교체하는 것이다

롤백 설계에서 가장 중요한 구분 하나. 롤백은 이전 아티팩트로 스위칭하는 것입니다.

이 둘은 완전히 다릅니다. 되돌리기(undo)는 코드 변경을 거두고 데이터를 지우고 마이그레이션도 거꾸로 실행하는 것입니다. 순서가 복잡하고 매 단계마다 실패할 수 있습니다. 교체(swap)는 트래픽을, 저장소에 이미 있는 이전 아티팩트로 가리키는 것입니다.

교체가 유효한 이유는 분명합니다.

이전 아티팩트는 이미 테스트를 통과한 것입니다. 돌아가는 것으로 확인된 버전입니다.

교체는 단일 조작입니다. 트래픽을 뒤집거나, 프로세스를 다시 올리거나.

실패 모드가 단순합니다. 교체 자체가 안 되면, 예전 것이 여전히 도는 것입니다.

그래서 설계의 질문은 이것으로 좁혀집니다. 교체를 깨끗하게 하려면 무엇이 필요하나.

4.2 깨끗한 교체를 위한 세 가지

첫째, 새 코드가 옛 코드가 읽을 수 없는 데이터를 쓰면 안 됩니다.

새 버전이 응답 JSON에 필드를 추가하면, 옛 버전은 모를 필드를 무시하니 문제없습니다. 하지만 새 버전이 DB에 새 형식으로 데이터를 쓰고, 옛 버전이 그 형식을 못 읽는다면, 롤백한 순간 서비스가 깨집니다. 롤백할 곳은 있는데, 가면 곧바로 터지는 경우입니다.

예를 들어, 새 버전이 주문을 JSON으로 저장하면서 discount_type 필드를 추가한다고 해 봅시다. 옛 버전은 읽을 때 모를 필드를 무시하므로, 문제없습니다. 하지만 새 버전이 amount 필드의 형식을 정수에서 문자열로 바꾼다면, 정수로 읽던 옛 버전은 깨집니다. 전자와 후자의 차이는 "추가"와 "기존의 의미를 바꾸기"입니다. 롤백 설계는 전자를 허용하고 후자를 금합니다.

실용적인 규칙은 "쓰기는 옛 형식으로, 읽기는 새 형식으로"입니다. 한동안은 옛것과 새것이 모두 이해할 수 있는 형식으로 쓰고, 형식 마이그레이션은 다음 버전에서 천천히 합니다. 2장의 expand/contract와 같은 정신입니다.

둘째, 설정은 바깥에 있어야 합니다.

새 버전이 새로운 환경 변수를 기대하는데, 프로덕션에 그 변수가 없으면, 롤백이 안 되는 상황이 만들어집니다. 새 설정을 추가하는 순간, 해당 환경에 값을 함께 두고 전환 기간 동안에는 옛 동작과 호환되는 기본값을 설정에 넣어 두세요. 그러면 옛것과 새것 모두, 같은 환경에서 뜹니다. 2장의 배포 순서와 같은 이야기입니다.

셋째, 기능 플래그를 씁니다.

기능 플래그는 배포를 다시 돌리지 않고 특정 기능을 켜고 끌 수 있게 하는 스위치입니다. 위험한 기능이라면, 코드는 올려 두되 플래그는 꺼 두었다가,

5퍼센트의 사용자만 켜고 지켜 보는 식으로 쓸 수 있습니다. 이상하면 플래그만 끄면 됩니다. 코드는 그대로, 동작은 이전 상태로. 아티팩트 전체를 바꾸는 것보다 빠르고 안전했던 다른 변경까지 같이 되돌리는 일도 없습니다.

다만 플래그는 만능이 아닙니다. 한동안 켜다 끄면, "코드는 있는데 아무도 쓰지 않는 기능"이 서비스 안에 남게 되고 이것이 쌓입니다. 플래그에는 만료일을 붙이고, 다음 릴리스에서 플래그와 함께 죽은 코드를 지워 주세요. 3개월 넘게 제자리에 있는 플래그는, 부채입니다.

4.3 롤백을 결정하는 기준

교체의 기계가 갖춰졌더라도, "지금 롤백할 때"를 판단하는 일은 쉽지 않습니다. 이유가 감정입니다. "방금 올린 건데, 아마 괜찮을 것이다. 조금만 더 지켜보자." 기다리는 사이, 작은 문제가 사고로 커집니다.

그래서 기준은 배포하기 전에 정해 둡니다.

에러율: 5xx 비율이 배포 전의 두 배로 오르면 당장 조사합니다. 빠른 설명이 없으면 롤백합니다.

핵심 경로: 스모크 테스트가 깨졌거나, 결제 성공률 같은 사업 지표가 떨어졌으면, 토론하지 말고 먼저 롤백합니다.

데이터: 데이터가 잘못되었을 가능성만 있어도, 먼저 롤백합니다. 데이터 정합성을 확인하는 데는 시간이 걸리고, 롤백은 몇 초면 되니까요.

"먼저 되돌리고 나중에 디버그한다"는 말에는 이런 뜻이 있습니다. 디버깅은 프로덕션이 안정된 상태에서 하는 것이 가치가 높다는 것. 깨진 환경은 생각하기 나쁜 곳입니다. 알려진 상태로 돌아가고 개발 환경에서 문제를 재현한 뒤에 조사합니다.

한 모양을 보여 줍니다. 오후 2시에 새 빌드를 배포했다고 하면, 2시 7분

스모크가 통과하고 2시 10분에 에러율이 올라오기 시작하고 2시 15분에 롤백 명령을 실행합니다. 2시 16분, 트래픽이 옛 아티팩트로 갑니다. 2시 20분, 옛 아티팩트의 스모크도 통과하며 "복구 확인"이 끝납니다. 결정에서 복구까지 약 10분. 이 속도를 만든 것은, 배포 전에 이미 준비돼 있던 롤백할 곳(이전 아티팩트)과 절차(한 줄 명령, 런북)였습니다.

그 반대도 기억해 두세요. 롤백은 끝이 아닙니다. 깨진 아티팩트는 저장소에 있고 로그는 있고 커밋은 있습니다. "왜 깨졌나"는 그 재료를 가지고 차분한 상태에서 답하는 질문입니다. 원하면 짧은 사후 분석을 한 번 합니다. "무엇이 일어났고 왜 기준이 더 빨리 잡지 못했고 무엇을 바꿀 것인가." 한 페이지면 충분합니다.

4.4 배포 후 보는 지표

"배포 후 몇 분을 본다"고 했습니다. 구체적으로 어떤 숫자를 보아야 할까요.

웹 서비스라면, 이 네 개를 두면 좋습니다.

5xx 비율: 배포 전 대비 두 배 이상 오르면 신호입니다.

P95 지연: 에러보다 먼저 오르는 경우가 많습니다. 작은 상승이 일찍 알려 줍니다.

백그라운드 작업 큐 길이: 워커가 깨지면 큐가 늘어납니다.

재시도 횟수: 외부 API 호출이 실패하면 재시도가 쌓입니다.

이 네 개를 처음부터 다 둘 필요는 없습니다. 5xx 비율과 큐 길이 두 개부터 시작하세요. 둘이 대부분의 문제를 잡아 줍니다. 사고가 날 때마다, "이 지표를 그때 있었다면"이라는 것을 하나씩 더하는 식입니다.

"배포 직후 10분간 이 숫자들을 본다"는 습관이 먼저입니다.

4.5 한 페이지짜리 롤백 런북

롤백도 파이프라인처럼, 한 분기에 읽을 수 있는 절차여야 합니다. 런북에 들어갈 내용은 이것입니다.

1. 현재 상태 확인 (어떤 아티팩트인가, 트래픽은 어디로 가나)
2. 롤백 명령 (한 줄)
3. 걸릴 예상 시간
4. 롤백 성공 확인 (스모크 테스트 재실행)
5. 누구에게 알릴 것

문서는 파이프라인 옆, 같은 리포지토리에 두고 배포 절차가 바뀔 때마다 함께 고칩니다. 6개월간 아무도 읽지 않은 런북은, 그 날 당장 실패할 런북입니다. 그래서 확인법이 하나 있습니다. 한 분기에 한 번, 프로덕션이 아닌 환경에서 롤백 절차를 실제로 돌려 보세요. "런북과 현실이 안 맞는다"는 것을 연습에서 발견하는 순간은, 싼 순간입니다.

4.6 롤백해도 안 되는 것: 이미 쓴 데이터

교체로 해결되지 않는 영역이 하나 있습니다. 새 코드가 돌고 있는 동안에 이미 쓰인 데이터입니다.

롤백으로 코드를 옛것으로 돌렸더라도, 새 코드에게서 나온 데이터는 그대로 남아 있습니다. 그래서 앞의 원칙 "쓰기는 옛 형식으로"가 중요한 것입니다. 옛 형식으로 쓴 데이터라면, 옛 코드도 그대로 읽을 수 있으니까요.

만약 이미 새 형식으로 데이터가 쓰였다면, 선택지는 두 가지입니다.

첫째, 백필의 역을 돌립니다. 새 형식의 데이터를 옛 형식으로 다시 쓰는 잡을 한 번 실행합니다. 데이터가 많지 않다면, 이것도 배포와 같은 수준의 조작입니다.

둘째, 읽기 레이어를 넓힙니다. 옛 코드에서 "읽을 때"만 두 형식을 다 이해하도록 패치를 합니다. 쓰기는 그대로 옛 형식입니다.

둘 다, "데이터의 모양"에 대한 판단입니다. 그리고 이 판단은 배포 전에 해야 합니다. 새 형식으로 쓸 수 있는 데이터가 어떤 것인지, 그리고 그것이 롤백된 세계에서 어떻게 살아남을 것인지. 이 질문을 배포 전에 못 던졌다면, 롤백은 코드의 문제만 해결해 주고 데이터의 문제를 다음 날 아침에 남깁니다.

그래서 4장의 핵심 원칙을 다시 쓰면, "코드를 교체하는 것은 쉽다. 데이터가 코드를 따라와야 한다"는 것입니다.

4.7 롤백 이후의 재배포

롤백은 같은 루프의 한 바퀴입니다. 문제를 고치고 다시 빌드하고 다시 배포합니다. 이때 주의할 것이 두 가지 있습니다.

첫째, 고친 빌드는 "새 아티팩트"입니다. 1.4.3을 롤백했다고 해서, 1.4.2를 다시 올리는 것이 아닙니다. 고친 것은 1.4.4로, 저장소에 새 이름으로 push합니다. 롤백의 목적지가 변해서는 안 됩니다.

둘째, 재배포는 원래 루프를 그대로 갑니다. 테스트, 빌드, push, 배포, 스모크. 롤백을 한 번 했다는 이유로 단계를 건너뛰면, 건너뛴 단계를 다시 사람의 기억에 맡기는 것입니다.

한 가지 더. 롤백과 재배포 사이의 시간 동안, 기준을 점검해 보세요. "왜 기준이 더 빨리 잡아주지 못했는가"에 답이 없다면, 다음 배포 전에 기준을 한 줄이라도 고쳐 둡니다. 에러율의 임계값을 낮춘다든가, 스모크에 핵심 경로를 하나 더 넣는다든가.

4.8 배포 전 체크리스트

매 배포 전에 돌리는 목록. 30초면 끝날 만큼 짧게 유지합니다.

- 이전 아티팩트가 저장소에 있는가 (롤백할 곳이 있는가)
- 스모크 테스트를 프로덕션 밖에서 최소 한 번 돌렸는가
- 새 버전이 요구하는 설정과 시크릿이 전부 준비되었는가
- 지금 시간이 트래픽의 낮은 때인가 (피크, 유지보수 시간 피하기)
- 런북이 열려 있는가

상식처럼 보이지만 서두르는 배포에서는 이것들을 하나씩 건너뜁니다. 그래서 체크리스트여야 합니다.

4.9 작고 잦은 배포: 롤백을 가볍게 하는 형태

롤백을 더 쉽게 만드는 원칙이 하나 더 있습니다. 배포를 작게, 자주 하세요.

배포 하나가 200개 변경을 한꺼번에 실으면, 롤백도 200개를 한꺼번에 되돌리고 문제 원인을 찾기도 어렵습니다. 배포 하나가 10개 변경을 실으면, 롤백은 작고 원인 찾기는 쉬워집니다. "어디서 깨졌나"를 묻는 질문에, 후보가 10개인 것과 200개인 것은 전혀 다른 문제입니다.

그래서 "배포 빈도" 자체가 지표가 됩니다. 프로덕션을 알려진 상태로 옮기는 횟수입니다. 자주 옮기는 팀은, 빨리 되돌리는 팀입니다. 같은 근육입니다.

4.10 배포의 루프

네 장을 함께 돌려 보세요.

빌드는 신원이 있는 아티팩트가 되어 머신을 떠납니다. 환경은 코드가 서명하는 계약이고 설정과 데이터 마이그레이션은 그 계약이 가장 잘

깨지는 두 곳입니다. 파이프라인은 배포의 판단을 실행 가능한 순서로 옮기고 아티팩트 push와 스모크 테스트가 그 순서의 두 핵심입니다. 그리고 롤백은 교체이며, 깨끗한 교체를 위한 조건은 바깥의 설정, 앞으로 호환되는 데이터, 기능 플래그입니다.

이것은 소규모 팀이 매일 반복하는 루프입니다.

push -> test, build -> 아티팩트 push -> deploy -> smoke -> observe -> 문제 없음: 머지, 알림 -> 문제 있음: 롤백(교체), 런북, 이후 디버그

배포가 무서운 이유는 기술이 어려워서가 아니라, “깨지는 순간”에 대비가 없기 때문입니다. 이 책의 논리는 그 순간을 미리, 설계로 준비하라는 것뿐입니다. 롤백할 곳이 항상 준비되어 있다면, 무언가를 보내는 순간은 도박이 아니라, 연습해 온 루프의 다음 바퀴가 됩니다.