1인 개발자를 위한 AI 네이티브 CI/CD

테스트·리뷰·배포를 AI가 전자동화하는 시대의 개발 워크플로우

ThakiCloud (Hyojung Han)

1인 개발자를 위한 AI 네이티브 CI/CD

테스트·리뷰·배포를 AI가 전자동화하는 시대의 개발 워크플로우

ThakiCloud (Hyojung Han)

Contents

1 왜 지금 AI 네이티브 CI/CD인가

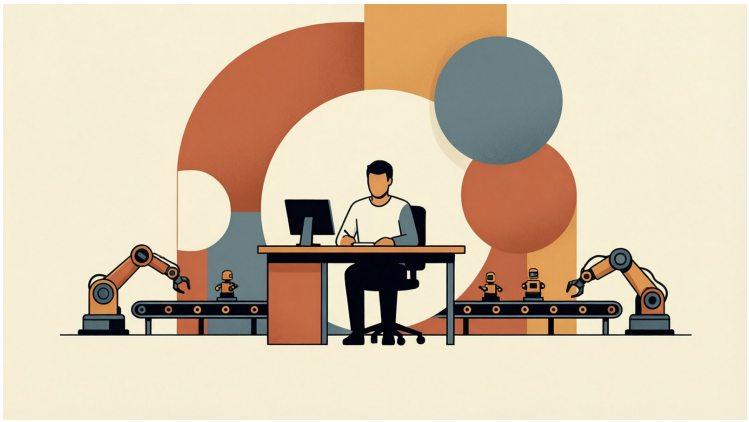

Figure 1.1: 왜 지금 AI 네이티브 CI/CD인가

1.1 전형적인 1인 개발자의 하루

오전 9시. 커피를 내리며 새 기능을 한 줄 완성한다. Git에 커밋하고 브랜치를 머지한다. Jenkins 파이프라인이 돌아가고 있다. 12분이 흐른다. 빨간 불이 들어오면 로그를 뒤져 에러를 잡고 다시 커밋한다. 오후 3시. 배포 버튼을 누르려는데 테스트가 또 깨졌다. 원인은 모듈 간 결합도 변경. 다시 파이프라인 앞부터 시작한다.

전통 CI/CD는 **사람이 코드를 확인하면 파이프라인이 돌아가고, 사람이 로그를 읽으면 문제가 발견되는 구조**다. 파이프라인 자체는 자동이지만 검증은 사람이 수동으로 한다. 2010년경 CI/CD가 처음 자리 잡을 때는 이 구조로 충분했다. 코드베이스가 작았고 배포 빈도도 하루 한 번이면 넉넉했다.

2026년 기준 SaaS 서비스는 하루에 수십 번 배포한다. 한 번 배포할

때마다 개발자가 15분씩 확인하면 하루 12시간이 CI/CD 관리에 나간다. 1인 개발자에게 이 시간은 제품 개발 시간을 곧바로 깎아먹는다.

1.2 AI 에이전트가 검증하는 시대

AI 에이전트 기술이 발전하면서 코드를 검증하는 행위 자체를 자동화할 수 있게 됐다. 컴파일러와 테스트 러너가 exit code로 성공과 실패를 판정하듯, AI 에이전트도 코드를 분석해 문제를 찾아내고 자동으로 수정까지 해준다.

가장 큰 차이는 검증 범위에 있다. 전통 CI/CD는 **입력이 올바른지**만 확인한다. 테스트가 통과하면 배포 후보가 된다. AI 네이티브 CI/CD는 **코드가 원하는 상태에 도달했는지**까지 판단한다. 테스트가 통과해도 코드가 의도한 기능을 구현했는지는 AI가 따로 감사한다.

예를 들어 REST API의 응답 형식이 바뀌었을 때, 전통 파이프라인은 새 형식에 맞춘 테스트만 통과시키면 그만이다. AI 네이티브 파이프라인은 API 문서와 실제 구현이 일치하는지까지 확인한다. 응답의 필드 이름이 문서와 다르면 테스트가 녹색이어도 경고를 낸다.

1.3 자동화 비율이라는 경쟁력

1인 개발자의 병목은 **검증 속도**다. 코드를 한 줄 바꾸는 데 1시간, 그 한 줄을 검증하는 데 3시간이 걸리면 하루에 세 번 할 수 있던 배포가 한 번으로 줄어든다. 자동화 비율이 높은 개발자는 같은 시간에 더 많은 피드백 루프를 돌린다.

피드백 루프가 빠를수록 버그는 작아진다. 코드를 작성한 직후 5분 안에 버그를 알면 고치는 데도 5분이면 충분하다. 같은 버그를 3일 뒤에 알면 맥락을 재구성하는 데만 1시간이 걸린다. AI 네이티브 CI/CD는 이 피드백 루프를 사람보다 빠르고 일관되게 실행한다.

전형적인 테스트 자동화 수준에서 60%의 테스트 커버리지를 갖출 때 버그 발견 비용은 코드 작성 비용의 약 3배다. AI가 코드 변경의 영향을 분석해 변경 부분만 집중 테스트하면 같은 커버리지에서 검증 비용을 40% 줄일 수 있다[추정].

1.4 이 책이 다루는 네 가지 축

AI 네이티브 CI/CD는 네 가지 축으로 구성된다. **Loop**는 AI 에이전트가 코드를 검토하고 피드백을 생성하는 반복 구조를 말한다. 에이전트에 명령을 내리는 시스템 프롬프트와 도구 정의가 **Harness**다. **Evals**은 에이전트 출력을 검증하는 외부 기준이며, 실패 원인을 소급해 기록하는 체계가 **Tracing**이다.

이 네 가지 축을 어떻게 설계하느냐에 따라 AI 파이프라인의 품질이 갈린다. 다음 장부터 축마다 하나씩 다룬다.

2 AI 네이티브 CI/CD의 4가지 축

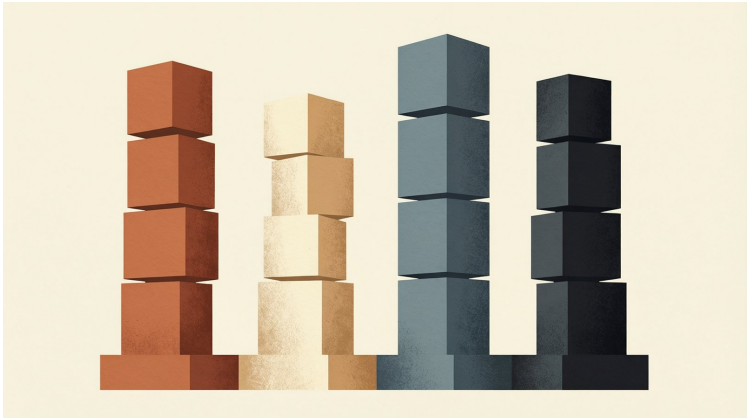

Figure 2.1: AI 네이티브 CI/CD의 4가지 축

2.1 축1: Loop, 관찰에서 실행까지

Loop는 AI 에이전트가 코드를 분석하고 변경을 제안하고 그 변경이 실제로 동작하는지 확인하는 **반복 구조**다. 전통 CI/CD는 파이프라인을 한 번 실행하고 끝나지만 Loop는 다르다. AI 네이티브 CI/CD의 루프는 다음 흐름으로 돌아간다.

개발자가 코드 변경을 커밋하면 AI 에이전트가 그 변경 사항을 분석한다. 이어서 테스트 러너와 linter를 실행해 출력을 해석하고, 문제가 보이면 코드를 고쳐 다시 테스트한다. 모든 검증을 통과해야 비로소 배포 후보가 된다.

이 루프의 핵심은 **컴파일러와 테스트 러너가 reward signal로 작동한다는 것**이다. 에이전트가 "이 코드가 맞다"고 주장하지 않는다. 외부 도구가 exit code를 돌려주면 에이전트는 그 신호로 다음 행동을 정한다.

예를 들어 Python으로 작성한 `calculate_discount` 함수에 할인율 계산 버그가 있다고 하자. AI 에이전트는 이 파일을 분석하고 테스트를 실행한다. 테스트가 실패하면 stderr를 읽어 버그 위치를 짚어내고, 코드를 고쳐 다시 테스트한다. 이 과정은 테스트가 통과할 때까지 반복된다.

루프의 종료 조건은 **외부 도구가 판정**한다. 에이전트 스스로 "완료됐다"고 말해서는 안 되고, `make test-short`가 exit code 0을 반환해야 끝난다.

2.2 축2: Harness, 명령 체계 설계

Harness는 AI 에이전트에게 **무엇을 해야 하는지, 어떻게 실행해야 하는지**를 알려주는 명령 체계다. 시스템 프롬프트, 도구 정의, 출력 형식 규칙이 여기에 포함된다.

Harness 설계에서 가장 흔한 실수는 **에이전트에게 너무 많은 자유를 주는 것**이다. "코드를 개선하라"는 명령에는 너무 많은 해석의 여지가 있다. 어느 정도로 개선해야 하는지, 무엇을 기준으로 삼아야 하는지 에이전트가 스스로 판단해야 한다. 이 판단이 사람마다 다르면 파이프라인의 출력이 불안정해진다.

좋은 Harness는 **행동의 범위를 좁히고 검증의 기준을 명확히** 한다. 예를 들어 "PR이 올라오면 코드 변경을 분석하고 보안 취약점이 있으면 `SECURITY_FINDINGS` 배열에 추가하라"는 명령이다. 에이전트는 보안 스캔 도구를 실행하고, 취약점이 발견되면 정해진 형식으로 결과를 반환한다. 자유 판단은 최소화된다.

도구 정의에서도 같은 원칙이 적용된다. 각 도구가 어떤 입력을 받고 어떤 출력을 반환하는지 **명확하게 기술**해야 한다. 출력이 변수라면 에이전트가 결과를 해석하는 데 에너지를 쓰게 된다. 출력이 고정된 스키마면 에이전트는 결과를 파싱하는 데 에너지를 아끼고 판단에 집중한다.

2.3 축3: Evals, 검증을 검증하는 기준

Evals는 AI 에이전트의 출력이 실제로 품질 기준에 부합하는지를 **외부 기준**으로 확인하는 단계다. 에이전트가 자기 결과를 스스로 판단하면 피드백 루프가 닫히지 않는다. 제3의 기준이 필요하다.

가장 단순한 Eval은 **테스트 실행 결과**다. Go 언어라면 `go test./...`의 exit code가 0이면 통과다. 이 검증은 에이전트와 무관하게 외부 도구가 수행하므로 신뢰할 수 있다.

더 복잡한 Eval은 **LLM-as-Judge** 패턴이다. 한 에이전트가 코드를 작성하면 다른 에이전트가 그 코드를 평가하는데, 이때 판정 기준을 명확히 해야 한다. "이 코드가 읽기 쉬운가?"라는 질문은 평가자마다 기준이 다르지만, "이 함수의 Cyclomatic Complexity가 10 이하인가?"는 수치로 판정할 수 있다.

실용적인 접근은 **2단계 Eval**이다. 1단계에서는 결정론적 도구(linter, 테스트, 타입 체커)를 돌리고, 2단계에서는 테스트 커버리지나 코드 복잡도 같은 수치 지표가 임계치를 넘었을 때만 LLM-as-Judge를 실행한다. 그래야 정말 필요할 때만 비용을 쓴다.

2.4 축4: Tracing, 실패의 고고학

Tracing은 AI 에이전트가 작업하는 동안 **각 단계의 입력과 출력, 소요 시간을 기록**하는 체계다. 실패가 발생했을 때 루프가 왜 실패했는지 소급할 수 있어야 한다.

전통 CI/CD에서는 Jenkins 로그가 Tracing에 해당한다. 파이프라인의 각 단계가 어떤 명령을 실행했고 어떤 출력을 반환했는지 로그에 남아 있다. AI 네이티브 CI/CD에서는 에이전트의 **사고 사슬까지 기록**해야 한다.

구체적으로는 에이전트가 코드 변경을 분석한 시각과 내용, 실행한 도구

목록과 각 도구의 입력과 출력, 테스트 결과와 그 해석 방식, 최종 결정과 근거까지 기록해야 한다.

이 기록이 있으면 문제가 발생했을 때 **루프가 어떤 단계에서 잘못되었는지**를 특정할 수 있다. 테스트가 실패했는데 에이전트가 그 실패를 무시했다면 Harness 설계 문제다. 테스트가 잘못된 결과를 반환했다면 Eval 기준 문제다. Tracing이 없으면 원인을 찾는 데 시간이 오래 걸린다.

2.5 4축이 서로 맞물리는 구조

Loop가 에이전트를 구동하고, Harness가 에이전트의 명령 체계를 정의하고, Evals가 결과를 검증하고, Tracing이 실패를 기록한다. 이 네 축이 모두 제대로 작동해야 AI 네이티브 CI/CD가 완성된다.

한 축이 빠져도 시스템이 돌아가지 않는다. Loop만 있으면 에이전트가 무한히 반복한다. Harness만 있으면 명령을 내릴 수 있지만 실행되지 않는다. Evals가 없으면 잘못된 결과도 배포 후보가 된다. Tracing이 없으면 실패의 원인을 알 수 없다.

다음 장부터는 각 축을 **구체적인 구현 예시**와 함께 다룬다.

3 테스트 자동화의 새로운 정의

Figure 3.1: 테스트 자동화의 새로운 정의

3.1 TDD에서 AI-Driven Testing으로

전통 TDD의 흐름은 간단했다. RED(실패하는 테스트를 먼저 작성), GREEN(테스트를 통과시키는 최소 코드 작성), REFACTOR(코드를 개선) 반복이다. 이 사이클이 개발자들에게 테스트와 구현을 동시에 작성하는 습관을 들였다.

AI-Driven Testing은 이 흐름을 확장한다. 개발자가 비즈니스 로직만 작성하면 AI 에이전트가 그 로직의 **경계 조건, 에러 처리, 역케이스**를 분석해 자동으로 테스트를 생성한다. 개발자는 생성된 테스트를 검토하고 문제가 있으면 수정 지시만 내린다.

구체적인 예를 보자. Python 함수 `transfer_funds(sender, receiver, amount)`가 있다. 개발자는 기본 로직만 구현했다. AI 에이전트가 이 함수를 분석해서 다음과 같은 테스트 케이스를 자동 생성한다.

잔액이 부족하면 `InsufficientFundsError`가 나와야 하고, 송금 금액이 0 이하면 `ValueError`가 나와야 한다. 존재하지 않는 수신자를 지정하면 `AccountNotFoundError`가 발생해야 하며, 송금자와 수신자가 같을 때의 동작도 확인 대상이다.

이 자동 생성은 **프로퍼티 기반 테스팅(Property-Based Testing)**과 연결된다. 일반적인 예시 기반 테스트가 "이 입력에 이 출력이 맞다"를 확인하는 반면, 프로퍼티 기반 테스팅은 "이 클래스의 모든 입력에서 이 속성이 성립한다"를 확인한다. AI 에이전트가 함수의 프로퍼티를 추론하고, 그 프로퍼티를 만족하는 무작위 입력을 생성해 테스트를 실행한다.

3.2 구간 커버러지 분석과 실패 포인트 감지

AI 에이전트가 테스트 커버러지를 분석할 때 단순한 라인 커버러지보다 **구간 커버러지**를 활용하면 분석이 더 정밀해진다.

구간 커버러지는 코드 실행 경로가 얼마나 자주 지나가는지를 본다. if-else 분기가 여러 개 있으면 각 분기의 실행 빈도가 기록된다. 한 분기가 100번 실행되고 다른 분기가 1번만 실행되면, 1번 실행된 분기가 잠재적 버그 숨김 지역이 된다.

AI 에이전트가 이런 패턴을 감지하면 **집중 테스트할 영역**을 추천한다. 예를 들어 "이 API 핸들러의 에러 처리 분기가 테스트되지 않았습니다. 404 응답을 반환하는 로직이 실제로 동작하는지 확인해 주세요"라는 메시지를 개발자에게 전달한다.

실패 포인트 감지에서는 실행 빈도와 결합해 **변경 영향 범위**도 분석한다. 한 모듈을 수정했을 때 그 모듈에 의존하는 다른 모듈들의 실행 경로가 어떻게 변하는지 추적한다. 의존성이 복잡한 모놀리식 코드베이스에서는 이 분석이 회귀 버그를 찾는 데 바로 쓰인다.

3.3 단위 테스트 줄이기의 전략

AI 네이티브 테스트 자동화에서 자주 나오는 질문이 "단위 테스트를 줄이면 품질이 떨어지지 않느냐"다. 답은 줄이는 게 아니라 재배치하는 데 있다.

전통적인 테스트 전략은 피라미드 형태를 따른다. 밑단에 단위 테스트가 수백 개, 중단에 통합 테스트가 수십 개, 정상에 E2E 테스트가 수 개 놓인다. 이 구조에서 단위 테스트는 수가 가장 많고 실행도 가장 빠르다.

AI 네이티브 구조에서는 **단위 테스트를 AI 생성으로 자동화**하고, 개발자의 시간을 **통합 테스트와 E2E 테스트 설계**에 집중한다. 단위 테스트의 수는 줄어들지만 AI가 생성한 테스트가 기존 단위 테스트보다 더 광범위한 입력을 커버한다. 개발자는 "이 기능이 올바르게 동작하는가"를 검증하는 시나리오 테스트에 에너지를 쓴다.

실제로 Google의 테스트 문화를 보면 단위 테스트와 통합 테스트의 비율이 프로젝트 성격에 따라 유동적으로 변한다. 핵심 비즈니스 로직이 담긴 모듈은 단위 테스트 비율을 높게 유지하고, UI와 결합된 모듈은 시나리오 테스트 위주로 바꾸는 편이 효율적이다.

3.4 검증 품질을 높이는 실제 파이프라인

GitHub Actions 기반 Python 프로젝트를 예로 들면, AI 네이티브 테스트 파이프라인은 다음과 같은 형태를 띤다.

1단계에서 개발자가 PR을 생성한다. 이 시점에 AI 에이전트가 코드 변경을 분석하고 변경 영향을 받는 모듈을 특정한다. 2단계에서 AI 에이전트가 해당 모듈에 **무작위 입력 기반 테스트**를 생성하고 실행한다. 생성된 테스트가 실패하면 에이전트가 원인을 분석하고, 코드 수정이 필요한 경우 자동으로 수정한다.

3단계에서 기존 테스트 스위트를 실행한다. 이 단계에서 linter와 타입

체커도 함께 실행된다. 4단계에서 테스트 결과를 AI 에이전트가 종합하고 PR 코멘트로 개발자에게 보고한다. 커버러지 부족 영역이 있으면 그 영역을 집중 테스트하라는 권고도 함께 전달한다.

실패 시에는 AI 에이전트가 자동으로 에러 로그를 분석하고 가장 가능성 높은 원인을 특정해 제안한다. 개발자는 제시된 원인을 검토하고, 맞으면 승인, 틀리면 조정을 지시한다. 이 사이클은 전통 파이프라인보다 **피드백 속도를 3배 이상** 높인다는 평가도 있다[추정].

4 AI 코드 리뷰 워크플로우

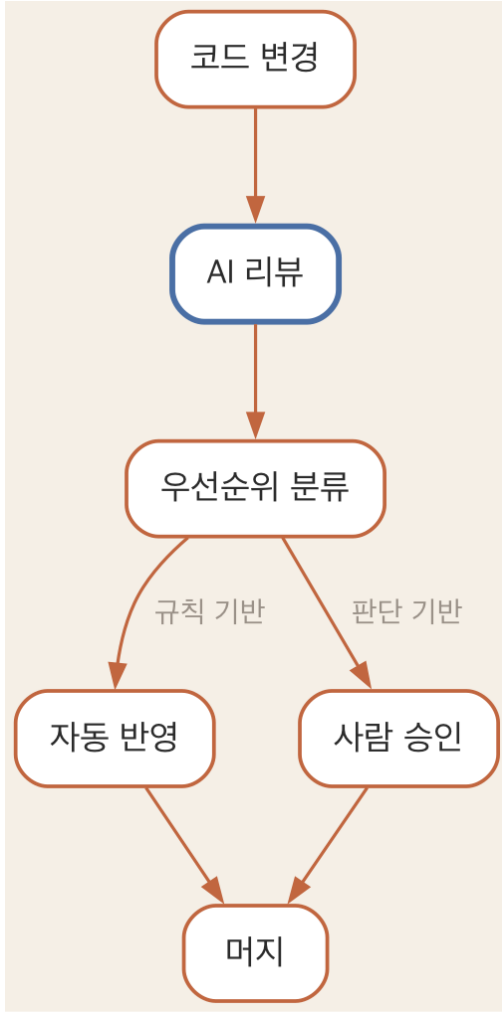

Figure 4.1: AI 코드 리뷰 워크플로우

4.1 AI 리뷰어의 역할과 한계

AI 코드 리뷰는 사람이 하는 리뷰를 완전히 대체하지 않는다. AI가 잘하는 영역과 사람이 잘하는 영역이 나뉘어 있을 뿐이다.

AI는 문법과 스타일 위반, 알려진 안티패턴, 타입 불일치와 타입 안전성 위반을 잘 찾아낸다. 테스트 커버리지 공백을 분석하고 반복되는 코드 패턴을 짚어내는 일도 AI에게 맡기면 된다.

반면 제품 요구사항과 구현이 실제로 맞는지, 아키텍처 결정이 타당한지는 사람만 판단할 수 있다. 비즈니스 로직의 타당성 검증이나 팀 Coding Convention과의 일치 여부도 마찬가지로 사람의 몫이다.

AI 코드 리뷰를 설계할 때 핵심은 여기에 있다. **AI가 할 수 있는 검증은 자동화하고, AI가 할 수 없는 판단은 사람이 내릴 수 있도록 정보를 정리해 제공하는 것**이다.

4.2 프로덕트 코드에 특화된 커스텀 리뷰 루프

AI 코드 리뷰의 효과를 극대화하려면 프로젝트의 **프로덕트 맥락을 Harness에 반영**해야 한다. 맥락이 없는 AI는 "이 함수가 너무 깁니다" 수준의 뻔한 피드백만 내놓는다.

프로덕트 맥락을 Harness에 반영하는 방법은 세 가지다.

먼저 **도메인 용어 사전**을 구축한다. 프로젝트에서 쓰는 고유 용어와 그 의미를 Harness에 등록하는 작업이다. "사용자가 결제를 완료한 상태"를 `payment_status = 'paid'`로 표현한다면 이 정보를 Harness에 알려준다. 그러면 AI가 변수명 `ps`를 봤을 때 payment_status의 약자임을 연결할 수 있다.

다음으로 **중요한 함수 목록**을 정의한다. 결제 처리, 인증, 데이터 삭제처럼

보안과 직결되는 함수를 Harness에 명시하고, 이 함수들이 변경될 때마다 보안 리뷰 에이전트가 실행되도록 한다.

마지막으로 **커스텀 lint 규칙**을 추가한다. 프로젝트 특유의 코딩 규칙을 ESLint나 golangci-lint의 커스텀 규칙으로 등록하는 것이다. "API 응답의 에러 필드는 반드시 `error.code`와 `error.message` 두 필드를 모두 포함해야 한다" 같은 규칙은 범용 linter에는 없지만 프로젝트에는 필수적이다.

이 세 단계를 거치면 AI 리뷰어의 피드백은 **일반적인 수준에서 프로젝트에 특화된 수준**으로 바뀐다.

4.3 보안과 접근성 자동 감사

보안 리뷰는 AI 코드 리뷰의 가장 효과적인 활용 사례 중 하나다. OWASP Top 10에 해당하는 패턴을 AI 에이전트가 자동으로 감지한다.

대표적인 감지 패턴은 이렇다. SQL 문자열에 사용자 입력이 그대로 이어붙는 SQL 인젝션, HTML 응답에 사용자 입력이 이스케이프 없이 들어가는 XSS, 인증 체크를 건너뛰는 분기, 로그에 비밀번호나 API 키가 그대로 찍히는 민감 데이터 노출 같은 것들이다.

AI가 보안을 감사할 때는 **가양성(False Positive) 처리**에 주의해야 한다. AI 보안 스캔은 보수적으로 설계되는 경우가 많아 실제로는 안전한 코드에서도 경고를 띄운다. 이 경고를 일일이 사람이 검토하면 리뷰 시간만 늘어나 오히려 비효율적이다. 그래서 AI 보안 감사 결과를 **우선순위별로 분류**하는 후처리 단계가 필요하다. 높음/중간/낮음으로 나눠 높음 우선순위만 사람이 검토하고 낮음 우선순위는 일괄 승인하는 Workflow가 효과적이다.

접근성(Accessibility) 감사도 비슷하다. 이미지 ALT 텍스트 누락, 키보드 내비게이션 불가, 색상 대비율 부족 같은 대표적인 접근성 위반을 AI 에이전트가 자동으로 감지한다. WCAG 2.1 기준을 Harness에 명시해두면

AI가 웹 앱의 접근성을 체계적으로 감사할 수 있다.

4.4 리뷰 결과를 코드에 자동 반영하는 파이프라인

AI가 찾아낸 문제를 개발자에게 보고만 하고 끝나면 리뷰의 일관성이 떨어지고 같은 실수가 반복되기 쉽다. 더 나은 Workflow는 **AI가 발견한 문제를 자동으로 코드에 반영**하는 것이다.

자동 반영이 가능한 영역은 명확하다. 포맷팅 문제(linter가 잡아내는 스타일 위반)는 자동으로 고칠 수 있고, 반복되는 코드 패턴을 추출해 함수로 분리하는 단순한 리팩토링도 에이전트가 처리할 수 있다. 인터페이스 변경에 따른 테스트 스텁 업데이트도 자동화에 적합하다.

반면 수동 검토가 필요한 영역도 뚜렷하다. 보안 관련 변경은 항상 사람이 승인해야 하고, 비즈니스 로직 변경은 개발자의 의도를 파악해야 하므로 자동화에 맞지 않는다. 아키텍처 결정과 관련된 변경은 팀 내 논의를 거쳐야 한다.

실용적인 구분 기준은 간단하다. **규칙 기반**으로 판단할 수 있는 것은 자동화하고, **판단 기반**인 것은 사람이 검토하고 승인한다. 자동화 범위를 넓히면 속도가 빨라지고, 수동 검토 범위를 지키면 품질이 보장된다.

자동 반영 Workflow의 구체적인 예를 보자. 개발자가 API 응답 형식을 변경하면 AI 에이전트가 이 변경을 감지하고 영향받는 클라이언트 코드를 분석한다. 응답 필드 이름이 `user_name`에서 `display_name`으로 바뀌었다면 클라이언트에서 해당 필드를 참조하는 코드를 자동으로 고친다. 수정 내역은 PR 코멘트에 남기고 개발자에게 승인을 요청하며, 개발자가 승인하면 머지한다.

이 Workflow가 작동하려면 변경 영향 범위를 정확히 분석하는 것이 핵심이다. AI 에이전트가 영향 범위를 잘못 판단하면 클라이언트 코드가 그대로 망가진다. 그래서 자동 반영 전에는 **영향 범위 검증 단계**를 반드시

포함해야 한다.

5 1인 개발 워크플로우 설계

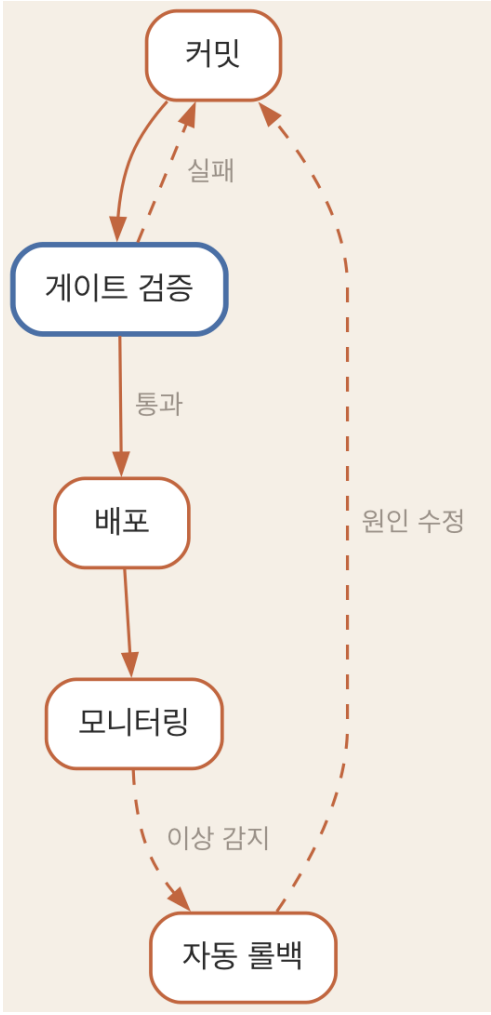

Figure 5.1: 1인 개발 워크플로우 설계

5.1 GitHub에서 바로 배포까지 한 번에

1인 개발에서 CI/CD 파이프라인을 단순화하는 핵심 원칙은 **검증과 배포의 경계를 명확히** 하는 것이다. 검증은 로컬에서 끝내고, 배포는 파이프라인에서 시작한다.

GitHub 기반의 가장 단순한 파이프라인을 살펴보자. 개발자가 로컬에서 `git commit`하면 GitHub Actions가 자동으로 실행되어, 테스트를 먼저 돌리고 이어서 AI 에이전트가 코드 변경을 분석한 뒤 문제가 없으면 자동으로 배포한다.

이 워크플로우의 핵심은 **gate-first 설계**다. 파이프라인을 구성할 때는 어떤 조건이 충족되어야 배포 후보가 되는지를 먼저 정의해야 한다. 이 조건을 만족하지 못하면 파이프라인은 다음 단계로 진행하지 않는다.

gate-first 설계의 구체적인 예시를 살펴보자. 테스트 커버리지가 70퍼센트 이상이어야 하고 lint 오류가 0개여야 하며, 보안 스캔에서 높음 우선순위 취약점이 나오지 않아야 하고 빌드도 성공해야 한다. 이 네 가지 조건을 모두 만족해야 배포 단계로 진입하며, 하나라도 어긋나면 파이프라인이 실패하고 개발자에게 통보된다.

5.2 Branch 전략과 병합 자동화의 균형

1인 개발에서는 trunk-based development가 가장 효율적이다. `main` 브랜치 하나에서 바로 개발하고 변경 사항을 검증하는 즉시 배포한다. feature 브랜치를 쓰면 병합 작업 자체가 비용이 된다.

trunk-based development를 선택하되 **보호 규칙**은 설정해둔다. 직접 푸시가 금지된 `main` 브랜치를 보호하고, 모든 변경은 PR을 거치도록 한다. PR에서는 AI 에이전트가 자동으로 검증하고, 검증이 끝나면 바로 머지한다.

PR 규모 관리도 자동화 대상이다. AI 에이전트는 PR의 변경 파일 수가

일정 수준을 넘으면 경고하는데, 1인 개발에서는 너무 큰 PR이 리뷰 품질을 떨어뜨리기 때문이다. 변경 파일이 10개를 넘으면 자동으로 분할을 권고한다.

5.3 배포 직전 자동 검사 목록

배포 전 반드시 확인해야 하는 항목을 **검사 목록(Checklist)**으로 정의하고, 이 목록을 AI 에이전트가 자동으로 검증하게 한다. 사람이 일일이 확인하면 누락이 생기기 쉽지만, 자동화하면 그 누락이 줄어든다.

배포 직전 자동 검사 목록의 예시를 살펴보자.

데이터베이스 마이그레이션이 필요하면 롤백 스크립트가 존재하는지 검증하고, 환경 변수가 바뀌었다면 새 값이 배포 환경에 실제로 반영되어 있는지 확인한다. API 문서를 업데이트해야 하는 상황이라면 문서가 실제 API와 일치하는지 살펴보며, 의존성이 바뀌었을 때는 취약점 스캔을 돌린다.

이 검사 목록은 프로젝트마다 다르다. 자사 프로젝트의 특성에 맞게 검사 목록을 정의하고, 이를 Harness에 등록한다. AI 에이전트가 이 목록을 순회하며 각 항목을 검증하고, 불만족 항목이 있으면 배포를 차단한다.

5.4 실패 시 자동 롤백과 알림

배포 후 문제가 감지되면 **자동 롤백**이 실행되어야 한다. 1인 개발자는 모니터링 화면을 계속 지켜볼 수 없기 때문이다.

자동 롤백의 트리거 조건도 먼저 정해두어야 한다. HTTP 500 에러 비율이 평소보다 3배 이상 늘거나, 특정 API 엔드포인트의 응답 시간이 평소의 5배를 넘거나, 결제 실패율 급증 같은 비즈니스 로직 기준을 벗어나면 롤백이 발동된다.

트리거 조건이 감지되면 AI 에이전트가 자동으로 이전 배포 버전으로 롤백하고, 그 뒤에는 **Slack이나 이메일로 개발자에게 통보**한다. 통보 내용에는 무엇이 문제였는지, 언제 롤백되었는지, 롤백 후 서비스 상태가 어떤지가 담긴다.

롤백 후 개발자는 문제를 해결하고 나서 다시 배포한다. 이때 롤백의 원인이었던 문제를 먼저 고치고, 그 수정 내용을 테스트에 반영해야 한다. 수정된 코드를 다시 배포하면 같은 원인으로는 다시 실패하지 않는다.

5.5 1인 개발자를 위한 툴 체인 정리

이 책에서 다룬 내용을 실무에 적용하려면 적절한 툴 체인이 필요하다. 1인 개발자의 예산과 시간 제약에 맞는 툴 체인을 아래와 같이 제안한다.

목적	추천 도구	비용
코드 저장소	GitHub	무료
CI/CD 파이프라인	GitHub Actions	무료(시간 제약)
AI 코드 리뷰	GitHub Copilot, CodeRabbit	유료
테스트 자동화	pytest, unittest	무료
보안 스캐닝	Snyk, Dependabot	무료 제한
배포	Vercel, Netlify, Railway	무료 제한

AI 네이티브 CI/CD의 핵심은 도구의 조합에 있다. 각 도구가 담당하는 역할이 명확하면 파이프라인이 비효율적으로 돌아가지 않는다. 역할을 정의한 다음에는 **각 도구의 출력을 검증하는 기준**을 구체적으로 세워야 하는데, 도구만 도입하고 검증 기준을 정의하지 않으면 AI 파이프라인은 사람을 대신할 수 없다.

이 책의 내용을 자신의 프로젝트에 적용하려면 2장에서 다룬 4가지 축을 먼저 설계하는 것을 권한다. Loop와 Harness를 먼저 구성하고 Evals와

Tracing은 점진적으로 추가하면 된다. 한 번에 모든 것을 자동화하려 하면 실패하기 마련이다. 핵심 기능 하나를 자동화해서 성과를 확인한 뒤 범위를 넓혀 나가는 것이 지속 가능한 접근법이다.