AI로 만들었는데 서버가 죽었습니다

바이브 코더를 위한 스케일링 생존 가이드

AGENT CODING 시대의 소프트웨어 엔지니어가 알아야 할 이야기들

목차

PART 1: 지금은 무엇을 만들 수 있는 시대인가

1장 소프트웨어의 형식은 어떻게 바뀌고 있는가

2장 AI는 흔히 무엇을 자동으로 만들어 주는가

3장 앱, AI 앱, 에이전트, 플랫폼은 어디에서 갈라지는가

PART 2: 내가 지금 만든 것은 정확히 무엇인가

4장 화면, 앱, 서비스는 무엇이 다른가

5장 프론트엔드, 백엔드, 저장 계층은 어떻게 나뉘는가

6장 요청 하나가 실제로 어떻게 흐르는가

PART 3: 프론트엔드

7장 React, Vite, Next.js는 각각 무엇인가

8장 웹 프론트엔드는 어떻게 돌아가는가

9장 SSR과 CSR은 누가 먼저 화면을 그리느냐의 차이이다

10장 SEO는 왜 렌더링 방식과 연결되는가

11장 첫 화면 속도는 결국 무엇을 먼저 보여주느냐의 문제이다

12장 AI 앱과 에이전트 프론트엔드는 무엇이 달라지는가

PART 4: 백엔드 1 – 앱 시대의 백엔드

13장 서버는 정확히 무슨 일을 하는가

14장 API와 웹훅은 무엇인가

15장 오래 걸리는 일은 요청과 분리해야 한다

16장 외부 API는 내 서버 밖의 병목이다

PART 5: 백엔드 2 – AI와 에이전트 시대의 백엔드

17장 AI 앱의 백엔드는 무엇이 다른가

18장 툴, 함수 호출, MCP는 무엇이 다른가

19장 에이전트 스킬은 무엇이며 어디에 놓이는가

20장 멀티 에이전트, 에이전트 플랫폼, 에이전틱 인터넷은 어떻게 이어지는가

PART 6: 저장소

21장 느린 서비스의 뒤편에서 벌어지는 일

22장 캐시는 속도를 높이는 도구이기 전에, 같은 질문을 덜 하게 만드는 도구이다

23장 쓰기가 몰리면 성능 문제가 아니라 값의 정확성이 무너진다

PART 7: 배포와 인프라

24장 서버 구조를 바라보는 가장 쉬운 기준

25장 VPS, EC2, GCE, Azure VM은 어떻게 대응되는가

26장 서버리스는 왜 많은 웹앱의 기본값이 되는가

27장 컨테이너는 언제 필요한가

28장 엣지는 입구에서 끝내는 판단에 가깝다

29장 실제 서비스에서는 구조를 어떻게 조합하는가

PART 8: 비용과 운영

30장 "어제까지는 잘 되던 앱"이 무너지는 방식

31장 느려지는 것과 멈추는 것은 다른 사건이다

32장 청구서는 서비스의 또 다른 로그이다

33장 비용을 낮춘다는 것은 요청을 덜 만들고, 같은 일을 덜 반복하게 만드는 일이다

PART 9: 성장 단계별 판단

34장 유저 0~100명 — 먼저 존재하는 서비스를 만들어야 한다

35장 유저 100~10,000명 — 첫 번째 병목은 이 시기에 드러난다

36장 유저 10,000~100,000명 — 구조를 다시 묻기 시작하는 구간

부록

A 응급 처치 플로우차트

B AI에게 구조 문제를 물을 때의 기준

C 무료 티어를 바라보는 태도

0장. 바이브 코딩 이후의 개발

개발의 출발선이 달라졌다

불과 몇 년 전만 해도 앱 하나를 만들기 위해서는 긴 준비 기간이 필요했다. 프로그래밍 언어 문법을 익히고, 프레임워크를 배우고, 데이터베이스를 붙이고, 배포 과정까지 익힌 뒤에야 비로소 작은 서비스를 세상에 내놓을 수 있었다. 무엇인가를 만든다는 일 자체가 높은 진입 장벽을 전제로 하던 시기였다.

지금은 그 풍경이 크게 달라졌다. 누군가는 주말 동안 AI에게 설명을 건네며 사진 업로드 앱을 만들고, 누군가는 퇴근 후 몇 시간 만에 결제 기능이 달린 서비스를 공개한다. 예전 같으면 팀이 필요했을 일을 이제는 혼자서도 시작할 수 있게 되었다. 코드를 직접 한 줄씩 모두 작성하지 않아도, 기능을 비교적 분명하게 설명하기만 하면 AI가 상당 부분을 대신 구성해 주기 때문이다.

이 변화는 단순한 생산성 향상에 그치지 않는다. 이전에는 시작조차 하기 어려웠던 사람들이 이제는 직접 서비스를 만들고 배포할 수 있게 되었다. 작은 생산성 도구를 만들든, 첫 창업 아이디어를 시험하든, 단순한 취미 프로젝트를 세상 밖으로 꺼내든, 출발 자체는 확실히 쉬워졌다. 만드는 일의 문턱이 내려간 것이다.

그러나 여기서 새로운 문제가 시작된다. 만드는 일은 쉬워졌지만, 버티게 만드는 일은 여전히 어렵다. 유저가 다섯 명일 때는 완벽하던 앱이 백 명이 되면 느려지고, 천 명이 되면 가끔 오류를 내고, 만 명이 되면 어느 날 아무 일도 없었다는 듯 멈춰 버린다. 그 순간 많은 바이브 코더는 같은 질문 앞에 서게 된다. 코드는 돌아가는데 서비스는 왜 버티지 못하는가. 이 책은 바로 그 질문에서 출발한다.

작동하는 것과 버티는 것은 다르다

AI는 놀라울 정도로 많은 일을 대신한다. 화면을 만들고, API를 만들고, 인증을 붙이고, 배포 설정까지 어느 정도 제안한다. 처음 접하는 사람에게는 거의 마법처럼 보일 정도이다. 하지만 AI가 주로 만들어 주는 것은 대체로 "작동하는 것"이다. 반면 실제 사용자가 몰려도 멀쩡한 구조, 비용이 통제되는 구조, 장애가 났을 때 복구 가능한 구조는 별도의 이해를 요구한다. 이 둘은 비슷해 보이지만 실제로는 전혀 다른 층위의 문제이다.

작동하는 앱을 만드는 일은 기능 중심의 문제이다. 버튼을 누르면 무슨 일이 일어나는지, 데이터를 어떻게 저장하고 보여 주는지가 핵심이다. 반면 버티는 앱을 만드는 일은 구조와 운영의 문제에 가깝다.

동시에 많은 요청이 들어올 때 무엇이 먼저 막히는지, 어떤 데이터 접근이 병목이 되는지, 어느 지점에서 비용이 폭증하는지, 장애가 발생했을 때 무엇부터 확인해야 하는지가 중요해진다.

이 간극을 처음 만나는 순간, 바이브 코더는 자주 당황한다. 무엇을 모르는지조차 분명하지 않은 상태이기 때문이다. 에러 메시지는 낯설고, 대시보드의 숫자는 많고, 플랫폼마다 쓰는 용어도 제각각이다. 검색을 하면 더 복잡한 말들이 쏟아지고, AI에게 물으면 지나치게 일반적인 조언이 돌아올 때도 많다. 결국 핵심은 하나이다. 기능을 만드는 법은 배웠지만, 그 기능이 현실의 트래픽과 비용과 장애를 만났을 때 어떻게 변하는지는 충분히 설명받지 못한 채 달려왔다는 점이다.

문제의 핵심도 바로 그 지점에 있다. 이미 빠르게 앱을 만들 수 있게 된 사람에게 필요한 것은 더 어려운 기술의 이름이 아니라, 그 앱이 왜 느려지고, 왜 깨지고, 왜 비싸지고, 왜 멈추는지를 설명할 수 있는 언어이다. 만들기의 시대 다음에 오는 것은 버티기의 시대이다.

바이브 코더에게 비어 있는 자리

오늘날의 바이브 코더는 혼자서도 작은 팀처럼 움직인다. 아이디어를 정하고, 화면을 만들고, 데이터베이스를 붙이고, 인증을 구성하고, 배포까지 마친다. 겉으로 보면 1인 개발자이지만 실제로는 기획자, 디자이너, 프론트엔드 개발자, 백엔드 개발자의 역할을 동시에 조금씩 수행하고 있는 셈이다. AI 도구는 이 과정을 압축해 준다.

그런데 이 팀에는 자주 빠져 있는 역할이 하나 있다. 서비스가 실제로 버티는 구조를 이해하는 사람, 다시 말해 인프라와 운영을 보는 사람이다. 이것이 바이브 코더가 특별히 취약해지는 이유이다. 기능을 완성하는 속도는 빠른데, 그 기능이 실제 사용자와 만나면서 어떤 종류의 병목과 장애와 과금 구조를 만들어 내는지에 대한 감각은 뒤늦게 따라오기 때문이다.

이 상황은 비유하자면 성능 좋은 차를 먼저 손에 넣은 뒤에 운전 감각을 뒤늦게 익히는 것과 비슷하다. 시동을 걸고 직선 도로를 달리는 일은 금방 배울 수 있다. 그러나 고속도로에서 차가 많아지고, 비가 오고, 급정거해야 하는 상황이 오면 그때부터는 전혀 다른 감각이 필요하다. 더 빠른 차가 필요한 것이 아니라, 차를 다루는 이해가 필요해지는 것이다.

필요한 것은 바로 그 감각을 언어로 풀어내는 작업이다.

이 책의 범위

이 책은 세상의 모든 컴퓨터 과학을 다루지 않는다. 쿠버네티스 운영 전반을 다루는 책도 아니고, 거대한 분산 시스템 설계 전체를 설명하는 책도 아니며, 대규모 엔터프라이즈 조직 운영을 위한 지침서도 아니다. 대신 훨씬 더 현실적인 범위를 본다. Vercel, Firebase, Supabase, Next.js 같은 도구를 활용해 서비스를 만들고 있는 사람, 아직 팀이 크지 않고, AI의 도움을 적극적으로 받으며 빠르게 기능을 구현하는 사람을 주요 독자로 상정한다.

이들이 실제로 가장 자주 만나는 문제는 대체로 비슷하다. 앱이 갑자기 느려진다. 어느 날은 아예 열리지 않는다. 데이터베이스가 예상보다 빨리 한계에 닿는다. 무료 플랜인 줄 알았는데 과금이 시작된다. 이미지가 늘자 비용이 치솟는다. 실시간 기능을 붙였더니 구조가 생각보다 훨씬 무거워진다. 배포는 했지만, 그 뒤로 무엇을 보고 어디를 점검해야 하는지 모르겠다.

중요한 점은, 쉬운 말로 설명한다고 해서 내용을 가볍게 만들지는 않는다는 사실이다. 이 책은 초심자도 읽을 수 있어야 하지만, 동시에 실제 문제를 푸는 데 충분한 밀도를 가져야 한다. 처음 읽을 때는 "왜 이런 일이 벌어지는지"가 이해되어야 하고, 두 번째 읽을 때는 "지금 내 서비스는 어느 지점에 있는지"를 스스로 판단할 수 있어야 한다. 설명은 친절해야 하지만, 판단은 흐리지 않아야 한다.

이 책이 다루는 범위는 분명하지만, 문제의 성격은 매우 현실적이다. 기능을 만들 수 있게 된 뒤에야 비로소 드러나는 병목, 비용, 구조, 장애 대응의 문제를 중심에 둔다. 따라서 여기서 필요한 것은 거대한 이론의 완결성보다, 실제 서비스가 흔들릴 때 그 이유를 설명할 수 있는 감각이다.

PART 1

지금은 무엇을 만들 수 있는 시대인가

웹 앱에서 AI 에이전트 플랫폼까지, 지금 시대의 소프트웨어 형식을 먼저 읽기

소프트웨어를 만드는 출발선은 이미 예전과 다르다. 한때는 화면을 만드는 일과 서비스를 운영하는 일이 서로 다른 세계처럼 느껴졌지만, 지금은 둘 사이의 간격이 훨씬 좁아졌다. AI에게 요구를 적절히 설명하면 로그인 화면이 생기고, 데이터가 저장되고, 배포 설정까지 이어지는 경우가 많다. 그 결과 초심자는 아주 오랜 시간 개념만 배우고 나서야 무엇인가를 만드는 대신, 무언가를 만든 뒤에야 자신이 만든 것이 정확히 어떤 성격의 산물인지 되묻는 위치에 서게 된다.

이 변화는 단순히 개발이 빨라졌다는 말로는 충분히 설명되지 않는다. 소프트웨어의 형태 자체가 넓어졌다. 웹 앱과 모바일 앱만이 있던 시대를 지나, AI 기능이 붙은 앱과 대화형 AI 앱이 등장했고, 이제는 도구를 사용하는 에이전트와 여러 에이전트를 조율하는 시스템, 그리고 그런 실행 단위를 만들고 운영하는 플랫폼까지 하나의 흐름 안에 놓인다. 같은 “앱”이라는 단어로 묶기에는 책임의 범위가 너무 달라졌다. 어떤 것은 화면을 잘 보여 주는 일이 핵심이고, 어떤 것은 모델을 잘 호출하는 일이 핵심이며, 어떤 것은 여러 실행 단위를 안정적으로 운영하는 일이 핵심이다.

바이브 코딩으로 들어온 사람에게 먼저 필요한 것은 이 지형을 눈에 익히는 일이다. 이름을 외우는 것이 아니라, 어떤 결과물이 어느 층위에 속하는지 구분하는 능력이다. 그 능력이 있어야 이후에 등장하는 프론트엔드, 백엔드, 저장소, 인프라라는 단어들도 추상적인 장식이 아니라 실제 구조로 읽힌다. 지금 시대의 소프트웨어는 더 많은 것을 만들 수 있게 되었지만, 그만큼 무엇을 만들고 있는지를 정확히 부르는 능력이 더 중요해졌다.

Chapter 1: 소프트웨어의 형식은 어떻게 바뀌고 있는가

가장 익숙한 형식은 웹 앱이다. 브라우저에서 열리고, 계정을 만들고, 데이터를 저장하고, 페이지나 화면이 바뀌는 서비스가 여기에 속한다. 노션, 피그마, 전자상거래 서비스, 관리 도구처럼 현대의 많은 서비스가 여전히 이 범주 안에 있다. 겉으로는 화면의 모음처럼 보이지만, 실제로는 사용자의 행동을 받아 상태를 바꾸고, 저장하고, 다시 보여 주는 순환 구조를 갖고 있다. 웹 앱은 단순히 웹페이지가 아니라, 브라우저 위에서 돌아가는 응용 프로그램이다.

모바일 앱은 같은 계열에 속하지만, 사용자가 만나는 접점이 다르다. 휴대전화의 운영체제 안에서 동작하고, 터치와 스와이프, 푸시 알림, 권한 요청, 백그라운드 실행 같은 요소가 자연스럽게 포함된다. 같은 기능을 제공하더라도 사용자 경험의 감각이 달라지고, 기술적 제약도 달라진다. 따라서 웹 앱과 모바일 앱은 비슷한 말을 가리키는 것이 아니라, 같은 서비스가 서로 다른 환경에서 살아가는 방식이라고 보는 편이 정확하다.

AI가 들어오면 소프트웨어의 형식은 한 번 더 넓어진다. 처음에는 기존 앱 안에 요약, 추천, 번역, 생성 같은 기능이 붙는다. 이 단계에서는 아직 웹 앱이나 모바일 앱이라는 이름이 맞다. 다만 핵심 가치의 일부가 모델 호출에 의존하기 시작한다. 사용자는 메뉴를 눌러 기능을 찾는 대신 문장을 입력하고, 시스템은 그 문장을 해석해 결과를 돌려준다. 그러다 보면 화면의 중심이 목록과 버튼에서 대화와 응답으로 이동한다. 이때부터 대화형 AI 앱이라는 표현이 자연스러워진다.

대화형 AI 앱은 인터페이스의 중심도 바꾼다. 화면이 예뻐도 사용자가 오래 머무는 곳은 입력창이고, 중요한 순간은 버튼 클릭이 아니라 응답이 생성되는 동안이다. 그래서 이런 앱은 단순한 정보 표시보다 상태 표현, 스트리밍 응답, 중간 진행 표시가 더 중요해진다. 결과물 자체보다 결과가 만들어지는 과정이 사용자 경험의 핵심이 되는 순간이다. 이것은 전통적인 폼 기반 앱과는 다른 감각이다.

에이전트는 여기서 한 걸음 더 나아간 형식이다. 단순한 답변 생성이 아니라 목표를 받아 여러 단계를 이어 붙이고, 도구를 호출하고, 때로는 중간 상태를 유지하면서 작업을 진행하는 실행 주체이다. 사용자가 문서를 정리해 달라고 요청하면, 에이전트는 한 번 답하고 끝나는 것이 아니라 파일을 읽고, 필요한 정보를 찾고, 내용을 정리하고, 다시 검토하는 식으로 움직인다. 사용자에게는 하나의 요청이지만 내부에서는 여러 행동이 연속된다. 이것이 일반 챗봇과 에이전트를 가르는 핵심이다.

멀티 에이전트 시스템은 실행 주체를 여러 개로 나누어 협업시키는 구조이다. 조사하는 에이전트, 요약하는 에이전트, 검토하는 에이전트, 최종 결정을 돕는 에이전트가 각자 다른 역할을 맡을 수 있다. 한 사람이 모든 일을 처리하는 대신 역할을 분리해 안정성을 높이는 것과 같은 이치이다. 문제가 복잡해질수록 한 덩어리의 똑똑함보다 분업된 실행이 더 견고해질 수 있다.

플랫폼은 그 위에 있는 층이다. 플랫폼은 개별 에이전트를 한 번 실행하는 수준을 넘어, 에이전트를 만들고 연결하고 상태를 추적하고, 승인과 권한을 관리하고, 실패를 관찰하고, 비용과 성능을 운영하는 환경이다. 이 단계에 이르면 소프트웨어는 단일한 앱이 아니라 여러 실행 단위를 다루는 운영 체계에 가까워진다. 이름이 화려해서 플랫폼이 되는 것이 아니라, 반복 가능한 실행과 통제를 가능하게 할 때 플랫폼이 된다.

결국 오늘날의 소프트웨어는 `웹 앱 → AI 기능이 붙은 앱 → AI 앱 → 에이전트 → 멀티 에이전트 시스템 → 에이전트 플랫폼`으로 이어지는 연속선 위에 놓인다. 각각은 완전히 끊어진 별개의 세계가 아니라, 기능과 책임이 하나씩 더해지는 방향으로 확장된다. 이 흐름을 먼저 잡아 두면 이후에 나오는 기술 이름들이 서로 다른 유행어가 아니라, 같은 시대 안의 서로 다른 층으로 읽힌다.

Chapter 2: AI는 흔히 무엇을 자동으로 만들어 주는가

AI가 자동으로 만들어 주는 것은 생각보다 많다. 화면 컴포넌트, 버튼과 폼, 라우팅, 간단한 API 뼈대, 인증 흐름의 기본 구조, 데이터베이스 연결, 배포 설정까지 한 번에 조립되는 경우도 드물지 않다. 그래서 처음 결과물을 마주한 사람은 종종 자신이 이미 서비스 전체를 다 만든 것처럼 느낀다. 그 감각은 이해할 만하다. 실제로 눈앞에는 로그인도 되고 저장도 되는 무언가가 있기 때문이다.

문제는 AI가 잘하는 일이 주로 형태를 빠르게 조립하는 데 있다는 점이다. 화면을 그럴듯하게 만들고, 파일 구조를 정리하고, 기본 기능이 이어지게 하는 일은 매우 강하다. 그러나 구조를 오래 버티게 만드는 일은 다른 문제이다. 어떤 요청이 병목이 될지, 어디에서 비용이 급증할지, 어떤 기능을 서버리스에 두고 어떤 기능을 분리해야 할지, 데이터베이스 읽기와 쓰기를 어떻게 줄여야 할지는 별도의 이해가 필요하다. AI가 자동으로 만든 결과물은 출발점을 빠르게 만들어 주지만, 그 출발점이 곧바로 결승선은 아니다.

많은 사람이 여기서 가장 쉽게 빠지는 착각은 바깥모양과 완성도를 같은 것으로 보는 일이다. 로그인 화면이 있다고 해서 인증 체계가 튼튼한 것은 아니고, API가 있다고 해서 백엔드의 경계가 잘 나뉜 것도 아니며, 배포가 끝났다고 해서 운영 준비가 완료된 것도 아니다. AI가 만든 화면은 종종 아주 그럴듯하지만, 그 아래의 구조는 아직 임시적이다. 그래서 무엇이 이미 자동으로 생겼는지와 무엇이 아직 비어 있는지를 분리해서 보는 습관이 필요하다.

AI 시대의 개발자에게 필요한 첫 번째 능력은 바로 그 구분이다. 기본형이 만들어졌는지, 확장 가능한 구조가 준비되었는지, 나중에 손댈 부분이 어디인지 읽어내는 일이다. 이 능력이 없으면 모든 문제가 갑작스러운 사고처럼 보인다. 반대로 이 능력이 있으면, AI가 빠르게 만든 뼈대를 어디까지 신뢰하고 어디부터 직접 구조를 세워야 하는지 판단할 수 있다.

AI가 흔히 제공하는 것은 또 하나의 층위에서 보면 기본형이다. 폼, CRUD, 기본 API, 기본 인증, 기본 배포는 쉽게 나온다. 그러나 실제 서비스는 그 기본형 위에 예외 처리, 권한 분기, 데이터 모델의 성장, 장애 대응, 비용 제어 같은 층이 더해져야 한다. 결국 AI가 만들어 주는 것은 결과물의 중심이 아니라 출발점인 경우가 많다. 이 사실을 인정해야 이후의 보강이 자연스러워진다.

Chapter 3: 앱, AI 앱, 에이전트, 플랫폼은 어디에서 갈라지는가

이 구분은 유행어를 정리하려는 작업이 아니다. 내가 무엇을 만들고 있는지에 따라 프론트엔드의 모습도, 백엔드의 책임도, 저장소의 종류도, 비용 구조도 달라지기 때문이다. 같은 화면이라도 앱인지, AI 앱인지, 에이전트인지에 따라 내부 설계는 전혀 다를 수 있다. 따라서 이 경계를 분명히 하는 일은 선택 사항이 아니라 기본이다.

전통적인 앱은 대체로 사용자가 직접 조작하는 화면과 명확한 기능 단위로 이루어진다. 사용자가 목록을 보고, 버튼을 누르고, 폼을 제출하면 서버가 응답한다. 여기에 저장과 인증이 붙으면 사람들은 그것을 웹 서비스라고 부르기 시작한다. 핵심은 사용자의 조작이 비교적 명확하고, 시스템의 역할도 비교적 고정되어 있다는 점이다. 앱은 정해진 기능을 안정적으로 수행하는 데 강하다.

AI 앱은 여기에 모델 호출이 중심 기능으로 들어간 상태이다. 사용자가 던진 입력을 해석하고, 생성하거나 요약하거나 추천하는 능력이 서비스 가치의 핵심이 된다. 하지만 모든 AI 앱이 곧바로 에이전트는 아니다. 많은 서비스는 사실상 질문을 받고 답을 돌려주는 구조에 머문다. 이 경우 AI는 강력한 기능이지만, 여전히 하나의 기능 층으로 붙어 있는 셈이다. 사용자가 느끼는 혁신은 크지만 내부의 책임 구조는 아직 전통적인 앱의 연장선일 수 있다.

에이전트가 되려면 목표와 행동이 생겨야 한다. 한 번의 응답으로 끝나는 것이 아니라, 여러 단계에 걸친 실행과 도구 사용, 상태 추적, 때로는 승인 흐름이 필요해진다. 사용자가 문서를 정리해 달라고 요청했을 때, 시스템이 파일을 찾고 내용을 읽고 필요한 정보를 추출하고 다시 검토하는 식으로 움직여야 한다면 그것은 이미 에이전트적 구조이다. 이때 중요한 것은 결과 문장만이 아니라 그 결과에 도달하는 과정 전체이다.

플랫폼은 다시 한 단계 위이다. 플랫폼은 개별 결과물보다 그 결과물을 계속 만들고 운영하는 시스템에 가깝다. 에이전트를 추가하고, 권한을 분리하고, 어떤 툴을 누구에게 허용할지 정하고, 실행 로그를 보고, 실패를 복구하고, 팀 차원에서 통제하는 문제가 중요해진다. 에이전트가 하나의 작업 단위라면, 플랫폼은 그런 작업 단위를 지속적으로 다루는 운영 환경이다. 이 시점에서 소프트웨어는 기능 묶음이 아니라 관리 가능한 생태계가 된다.

이 책이 이 경계를 강조하는 이유는 단순하다. 모든 것을 앱이라고 부르면 AI 앱의 고유한 백엔드가 보이지 않는다. 모든 것을 에이전트라고 부르면, 실제로는 단순한 모델 호출 래퍼에 불과한 서비스도 과장되게 이해하게 된다. 지금 무엇을 만들고 있는지 정확히 불러낼 수 있어야 이후 파트에서 그 구조를 해부할 수 있다. 이름은 겉치레가 아니라 설계의 출발점이다.

PART 2

내가 지금 만든 것은 정확히 무엇인가

화면, 앱, 서비스, 저장 계층을 구분하고 요청 하나가 실제로 어디를 거치는지 이해하기

기술 이름보다 먼저 필요한 것은 결과물의 층위를 구분하는 능력이다. 같은 프로젝트 안에 화면 코드와 서버 코드와 저장 코드가 한꺼번에 들어 있을 수 있기 때문에, 처음 만드는 사람은 자신이 정확히 무엇을 만든 것인지 자주 놓친다. 브라우저에서 보이는 화면을 만든 것인지, 그 화면이 서버와 대화하는 앱을 만든 것인지, 인증과 저장과 외부 연동까지 갖춘 서비스를 만든 것인지가 흐릿하면 이후의 모든 설명도 추상적으로 들린다.

개발 초반에는 결과물이 종종 하나의 덩어리처럼 보인다. AI가 만들어 준 파일 구조도 그렇고, 직접 붙인 기능도 그렇다. 화면에서 API를 부르고, API가 DB를 만지고, 그 결과가 다시 화면으로 올라오면, 눈앞에는 이미 작동하는 무언가가 있다. 하지만 작동한다는 사실과 구조를 이해한다는 사실은 다르다. 이 차이를 분명히 보아야, 지금 가진 것이 데모인지 앱인지 서비스인지 판단할 수 있다.

화면은 보여 주는 일에 집중하고, 앱은 사용자의 입력을 받아 내부 상태를 바꾸며, 서비스는 거기에 저장과 인증과 배포와 운영을 더한다. 같은 프로젝트라도 어느 층까지 갖추었는지에 따라 성격이 달라진다. 겉으로는 똑같이 보이는 버튼 하나도, 단순히 디자인 요소인지 실제로 서버에 데이터를 남기는 진입점인지에 따라 의미가 완전히 달라진다. 따라서 무엇을 만들었는지 묻는 일은 사소한 분류가 아니라, 앞으로 어떤 책임을 추가해야 하는지 정하는 일이다.

Chapter 4: 화면, 앱, 서비스는 무엇이 다른가

화면은 사용자가 보는 결과이다. 버튼이 있고, 입력창이 있고, 목록이 있고, 상태가 바뀌면 다시 그려진다. 그러나 화면만으로는 서비스가 되지 않는다. 화면이 서버와 연결되지 않고, 데이터가 저장되지 않고, 다른 사용자와 상태를 공유하지 않는다면 그것은 여전히 데모나 인터페이스에 가깝다. 화면은 보이는 층이고, 서비스는 그 보이는 층 뒤에 축적과 규칙과 운영을 포함한 더 넓은 구조를 가진다.

앱은 화면보다 한 단계 더 넓은 개념이다. 사용자의 입력을 받고, 그 입력에 따라 내부 상태가 바뀌고, 때로는 서버와 통신한다. 하지만 앱이라고 해서 반드시 서비스인 것은 아니다. 로컬에서만 동작하는 생산성 도구도 앱일 수 있고, 브라우저 안에서만 돌아가는 싱글 페이지 앱도 앱일 수 있다. 앱은 사용 가능한 기능의 단위에 가깝고, 서비스는 그 기능이 여러 사람과 여러 상황 속에서 지속되도록 만든 구조이다.

서비스는 다른 사용자와 데이터, 인증, 저장, 배포, 운영을 함께 전제로 한다. 로그인과 데이터 공유, 외부 결제, 알림, 관리 화면 같은 것이 들어오면 단순한 앱을 넘어서는 순간이 온다. 누가 무엇을 볼 수 있는지, 어떤 정보가 남아야 하는지, 문제가 생겼을 때 어디에서 끊어야 하는지까지 포함되어야 서비스가 된다. 따라서 서비스는 단지 앱보다 더 큰 말이 아니라, 책임의 범위가 다른 말이다.

이 차이가 중요한 이유는 운영의 문제도 여기서 갈라지기 때문이다. 화면은 잘 보이면 끝날 수 있지만, 서비스는 데이터를 잃지 않아야 하고, 응답이 너무 느려지지 않아야 하고, 누가 무엇을 볼 수 있는지도 관리해야 한다. 디자인이 아무리 좋고 기능이 아무리 많아도, 그 뒤의 구조가 허술하면 서비스라고 부르기 어렵다. AI가 화면과 간단한 기능을 한꺼번에 조립해 주더라도, 그것이 서비스로서 준비된 상태인지까지는 따로 판단해야 한다.

Chapter 5: 프론트엔드, 백엔드, 저장 계층은 어떻게 나뉘는가

입문자 설명에서는 흔히 `프론트엔드 / 백엔드 / 데이터베이스`를 같은 선 위에 놓는다. 이해를 돕는 데는 쓸모가 있지만, 조금 더 정확하게 말하면 데이터베이스는 대개 백엔드가 사용하는 핵심 저장 계층에 가깝다. 따라서 구조를 볼 때는 먼저 `사용자와 직접 만나는 층`과 `요청을 받아 처리하는 층`을 나누고, 그다음 `무엇이 값을 기억하는가`를 별도로 보는 편이 자연스럽다. 이렇게 보면 DB를 백엔드와 완전히 같은 급으로 놓는 설명이 왜 엄밀하지 않은지도 함께 이해된다.

프론트엔드는 사용자가 직접 보는 화면과 상호작용의 층이다. 버튼을 누르고, 내용을 입력하고, 목록을 보고, 상태 변화를 체감하는 곳이 여기이다. 프론트엔드는 보여 주는 방식만 다루는 것이 아니라, 사용자의 의도를 잘 받는 방식도 다룬다. 어떤 입력을 먼저 받을지, 어떤 오류를 먼저 보여 줄지, 결과가 지연될 때 무엇을 표시할지 같은 것이 모두 프론트엔드의 일이다. 사용자는 이 층을 통해 시스템 전체를 경험한다.

백엔드는 사용자의 요청을 받아 규칙을 적용하고, 인증을 확인하고, 계산을 하고, 저장소와 외부 서비스를 다루는 층이다. 백엔드는 눈에 보이지 않지만, 실제로는 서비스의 판단이 집중되는 곳이다. 누가 접근할 수 있는지, 무엇을 저장할지, 어떤 조건에서 실패로 돌려줄지, 외부 서비스가 느릴 때 어떻게 처리할지 같은 결정을 맡는다. 그래서 백엔드는 단순한 데이터 전달 통로가 아니라, 서비스의 행동을 정의하는 곳이다.

저장 계층은 이 백엔드가 읽고 쓰는 바탕이다. 관계형 데이터베이스, 문서형 데이터베이스, 캐시, 파일 저장소, 검색 인덱스가 모두 이 범주 안에 들어갈 수 있다. 중요한 것은 저장소가 하나의 형태만 뜻하지 않는다는 점이다. 어떤 데이터는 오래 남아야 하고, 어떤 데이터는 빠르게 읽혀야 하며, 어떤 데이터는 검색을 쉽게 해야 하고, 어떤 데이터는 잠시만 필요하다. 백엔드는 이 서로 다른 성격의 저장 수단을 조합해서 서비스의 기억 방식을 만든다.

이 구분을 정확히 해야 하는 이유는 책임이 서로 다르기 때문이다. 프론트엔드는 보여 주는 방식과 상호작용을 다루고, 백엔드는 판단과 처리와 연결을 다루며, 저장 계층은 상태를 유지한다. AI가 만들어 준 코드가 한 프로젝트 안에 섞여 보여도, `누가 보여 주는가`, `누가 판단하는가`, `누가 기억하는가`를 기준으로 나누면 훨씬 덜 흐려진다. 구조를 나눌 수 있어야 구조를 고칠 수도 있다.

Chapter 6: 요청 하나가 실제로 어떻게 흐르는가

사용자가 버튼을 누르는 순간부터 생각해 보자. 프론트엔드는 입력값을 모아 서버에 요청을 보낸다. 백엔드는 이 요청을 받아 사용자가 누구인지 확인하고, 요청이 허용되는지 판단하고, 필요한 데이터를 읽거나 쓰고, 외부 API를 부를 수도 있다. 그다음 결과를 프론트엔드에 돌려주면, 프론트엔드는 화면을 갱신한다. 사용자는 대개 이 과정을 하나의 자연스러운 반응으로 느끼지만, 내부에서는 여러 층이 순서대로 움직이고 있다.

이 흐름을 머릿속에 그려 두면 앱을 보는 눈이 달라진다. 화면에서 버튼이 한 번 눌렸을 뿐이지만, 실제로는 여러 단계의 책임이 겹쳐 있다. 입력 검증이 프론트엔드에서 일어날 수도 있고 백엔드에서 다시 검증될 수도 있으며, 저장 계층은 데이터를 남기고 외부 서비스는 별도의 응답 시간을 가질 수 있다. 사용자는 하나의 행동으로 느끼지만, 시스템은 여러 구성요소가 동시다발적으로 협력한 결과를 돌려준다.

문제는 AI가 이 여러 층을 한 번에 만들어 주는 경우가 많다는 점이다. 그래서 초심자는 화면 파일 안에 서버 호출 코드가 있고, 서버 코드 안에 저장소 호출이 있고, 인증 로직까지 함께 붙어 있는 모습을 보며 이 모든 것이 하나의 덩어리처럼 느낀다. 하지만 실제 구조를 이해하려면 요청의 흐름을 직선이 아니라 층위로 읽어야 한다. 브라우저가 시작하고, 서버가 판단하고, 저장 계층이 값을 돌려주며, 외부 서비스가 개입할 수 있다는 흐름을 이해하는 것이 중요하다.

이 흐름이 눈에 익으면 서비스처럼 보이지만 아직 서비스라고 부르기 어려운 상태도 구분할 수 있다. 화면은 있지만 저장이 없는 경우, 저장은 되지만 인증과 권한이 엉성한 경우, AI 응답은 나오지만 실제로는 한 번의 모델 호출만 감싼 구조인 경우를 더 정확하게 볼 수 있다. 무엇을 만들었는지 분명하게 이름 붙이는 일은 결코 사소하지 않다. 바로 그 이름이 다음에 무엇을 보강해야 하는지를 알려 주기 때문이다.

PART 3

프론트엔드

React, Vite, Next.js, SSR, CSR, SEO, 그리고 AI 시대의 사용자 인터페이스

프론트엔드는 사용자가 처음 만나는 층이기 때문에 가장 친숙해 보이지만, 실제로는 많은 오해가 쌓이는 곳이기도 하다. React가 무엇인지, Next.js가 무엇인지, Vite는 왜 필요한지, SSR과 CSR은 무슨 차이인지, SEO가 왜 렌더링 방식과 연결되는지 같은 질문이 모두 이 파트에 모여 있다. 문제는 이 개념들이 대개 서로 다른 층위에 속하는데도 처음 배울 때는 한 덩어리의 용어처럼 느껴진다는 데 있다.

프론트엔드를 제대로 이해한다는 것은 예쁜 화면을 만드는 기술을 익히는 데서 끝나지 않는다. 사용자가 무엇을 먼저 보게 되는지, 브라우저와 서버가 화면을 어떻게 나누어 맡는지, 검색 엔진과 첫 화면 속도가 왜 구조의 영향을 받는지, AI 앱과 에이전트 UI에서는 왜 대화와 실행 상태가 중요해지는지까지 함께 이해해야 한다. 이 파트는 바로 그 지형을 정리하는 데 목적이 있다.

Chapter 7: React, Vite, Next.js는 각각 무엇인가

이제 도구 이름을 올려놓으면 조금 덜 헷갈린다. React, Vite, Next.js는 모두 프론트엔드 생태계에서 자주 등장하지만, 서로 같은 역할을 하지 않는다. React는 화면을 만들기 위한 라이브러리이다. Vite는 그런 프론트엔드 앱을 빠르게 개발하고 빌드하기 위한 도구이다. Next.js는 React 앱을 실제 웹 서비스 구조 안에 놓기 위한 프레임워크이다.

React부터 보면 가장 단순하다. React는 화면을 컴포넌트 단위로 나누어 만들게 해 준다. 버튼, 카드, 폼, 목록, 모달처럼 반복되는 UI 조각을 독립된 단위로 만들고, 상태가 바뀌면 필요한 부분만 다시 렌더링하게 만드는 것이 핵심이다. 다시 말해 React는 "사용자에게 보이는 화면을 어떻게 구성하고 갱신할 것인가"를 다루는 도구이다. 이 점에서 React는 프론트엔드 층에 직접 놓인다.

Vite는 React와 같은 층의 도구가 아니다. Vite는 빠르게 개발 서버를 띄우고, 프론트엔드 자산을 번들링하는 데 초점이 있는 개발 도구이다. React 프로젝트를 시작할 때 Vite를 쓰면 개발 서버가 빠르게 뜨고, 바뀐 화면을 즉시 확인하기 쉽다. 그래서 "Vite가 React보다 더 큰 프레임워크인가"라고 오해하기 쉬운데, 실제로는 React 앱을 만드는 과정을 빠르고 가볍게 만들어 주는 작업 도구에 가깝다. Vite는 특히 브라우저 중심의 단일 페이지 앱, 곧 CSR 중심 구조와 잘 맞는다.

Next.js는 여기서 한 단계 더 나아간다. React 화면을 실제 웹 서비스로 만들기 위해 필요한 여러 층을 함께 제공한다. 파일 기반 라우팅, 서버에서 데이터를 가져오는 흐름, 페이지를 미리 만들거나 서버에서 렌더링하는 방식, 이미지 최적화, API 라우트, 배포 친화적 구조 같은 것들이 여기에 속한다. 즉 React가 화면 자체를 구성하는 도구라면, Next.js는 그 화면이 서버와도 연결되고 서비스로도 작동하게 만드는 프레임워크라고 볼 수 있다.

조금 더 직관적으로 말하면, React는 가구를 만드는 기술에 가깝다. Vite는 그 가구를 빨리 조립하고 시험해 볼 수 있게 해 주는 작업실 도구에 가깝다. Next.js는 그 가구를 실제 집 안에 배치하고, 문과 주소를 붙이고, 필요한 경우 주방과 창고까지 연결하는 시스템에 가깝다. React만으로도 화면은 만들 수 있다. Vite를 쓰면 그 화면을 빠르게 개발할 수 있다. 그러나 실제 서비스에서는 경로, 데이터 요청, 서버 렌더링, 배포, 최적화 문제가 곧바로 따라오기 때문에, 많은 경우 Next.js 같은 프레임워크가 필요해진다.

핵심은 이 셋을 경쟁 관계로 보는 것이 아니라, 구조 안에서 어디에 놓이는지를 이해하는 일이다. React는 프론트엔드 화면을 구성하는 핵심이고, Vite는 그런 프론트엔드 앱을 개발하는 도구이며, Next.js는 프론트엔드와 서버를 함께 다루는 서비스 프레임워크이다. 그래서 "React 프로젝트를 Vite로 시작한다"는 말과 "React 기반 서비스로 Next.js를 쓴다"는 말은 서로 다른 층위를 가리킨다.

AI에게 이렇게 물어보세요

> "프론트엔드, 백엔드, 데이터베이스 구조 위에 React, Vite, Next.js가 각각 어디에 놓이는지 설명해줘. 누가 화면을 만들고, 누가 개발을 돕고, 누가 서비스 구조를 맡는지 구분해서 알려줘."
>
> "React, Vite, Next.js가 각각 무엇인지 헷갈린다. 화면 라이브러리, 개발 도구, 서비스 프레임워크라는 관점에서 차이를 설명해줘."

Chapter 8: 웹 프론트엔드는 어떻게 돌아가는가

웹 프론트엔드는 브라우저 안에서 살아 움직이는 프로그램이다. 사용자가 링크를 누르고, 폼을 채우고, 상태가 바뀌면 화면의 일부가 다시 그려진다. 예전에는 페이지를 이동할 때마다 서버가 새로운 HTML을 통째로 돌려주는 구조가 기본이었다면, 지금의 웹 프론트엔드는 브라우저 안에서 라우팅과 상태 갱신과 부분 렌더링을 더 많이 맡는다. React가 널리 퍼진 이유도 이 흐름을 컴포넌트와 상태라는 단위로 다루기 쉽게 만들어 주었기 때문이다.

여기서 중요한 것은 웹 프론트엔드가 단지 그림을 그리는 층이 아니라, 사용자 경험을 조직하는 층이라는 점이다. 어떤 상태를 브라우저가 기억할지, 언제 서버에 요청을 보낼지, 목록이 길어질 때 어떻게 렌더링할지, 입력 중인 폼을 어떻게 안정적으로 유지할지가 모두 프론트엔드의 책임 안에 들어온다. 그래서 화면이 예쁘게 보이는 것과 프론트엔드가 잘 설계된 것은 같은 말이 아니다. 보이지 않는 상호작용의 리듬과 상태 관리가 훨씬 큰 비중을 차지한다.

모바일 앱 프론트엔드도 이와 닮았지만 완전히 같지는 않다. 푸시 알림, 권한 요청, 백그라운드 동작, 운영체제와의 결합이 더 강하다. 반면 웹 프론트엔드는 검색 엔진, 링크 공유, 주소 체계, 초기 로딩 속도와 더 깊게 묶여 있다. 언뜻 차이가 작아 보여도, 바로 그 차이 때문에 웹에서는 SSR과 CSR, SEO 같은 주제가 자연스럽게 프론트엔드 파트 안으로 들어온다.

Chapter 9: SSR과 CSR은 누가 먼저 화면을 그리느냐의 차이이다

SSR과 CSR은 프론트엔드에서 가장 많이 등장하면서도 가장 자주 오해되는 개념이다. 초심자에게 이 둘은 복잡한 약어처럼 보이지만, 실제 차이는 비교적 단순하다. 화면을 누가 먼저 완성하느냐의 차이라고 생각하면 된다.

CSR, 즉 `Client-Side Rendering`은 브라우저가 화면을 주로 완성하는 방식이다. 서버는 기본 골격이나 JavaScript 파일을 보내고, 실제 화면 내용은 브라우저가 그 코드를 실행하면서 채워 넣는다. 이 방식은 한 번 앱이 올라온 뒤 인터랙션이 풍부하고 화면 전환이 잦은 서비스에 잘 맞는다. 대시보드, 관리 도구, 로그인 후 사용하는 복잡한 앱이 여기에 속하기 쉽다.

반면 SSR, 즉 `Server-Side Rendering`은 서버가 먼저 HTML을 만들어 보내는 방식이다. 사용자의 요청이 들어오면 서버가 데이터까지 반영된 HTML을 생성해 브라우저로 보내고, 브라우저는 이를 먼저 보여 준다. 사용자는 화면의 내용이 더 빨리 보이는 것처럼 느낄 수 있고, 검색 엔진도 페이지 내용을 더 쉽게 읽을 수 있다.

이 차이를 식당에 비유하면 이해가 쉽다. CSR은 손님이 자리에 앉은 뒤 식탁 위 재료를 가지고 직접 마지막 조립을 하는 방식에 가깝다. SSR은 주방에서 이미 어느 정도 완성된 요리를 만들어 내보내는 방식에 가깝다. 어느 쪽이 절대적으로 더 좋은 것은 아니다. 무엇을 얼마나 빨리 보여주어야 하는지, 그리고 그 페이지가 누구에게 어떤 방식으로 소비되는지에 따라 적합성이 달라진다.

많은 현대 웹앱은 SSR과 CSR 중 하나만 순수하게 택하지 않는다. 서버에서 먼저 HTML을 보내고, 그 뒤 브라우저가 React를 붙여 상호작용을 가능하게 만드는 혼합형 구조가 흔하다. Next.js는 바로 이 혼합형 구성을 자연스럽게 다루는 프레임워크이다.

AI에게 이렇게 물어보세요

> "이 페이지가 SSR에 더 맞는지 CSR에 더 맞는지 판단해줘. 검색 노출, 첫 화면 표시 속도, 사용자 상호작용의 비중을 기준으로 설명해줘."

Chapter 10: SEO는 왜 렌더링 방식과 연결되는가

SEO를 검색 엔진용 마케팅 기법 정도로만 이해하면 프론트엔드 구조와의 관계를 놓치기 쉽다. 그러나 SEO의 중요한 한 축은 검색 엔진이 그 페이지를 얼마나 쉽게 읽고 이해할 수 있느냐에 있다. 여기서 SSR과 CSR의 차이가 직접적인 의미를 가지게 된다.

검색 엔진은 점점 더 JavaScript를 이해할 수 있게 되었지만, 여전히 가장 안정적인 방식은 읽을 수 있는 HTML이 먼저 제공되는 구조이다. 페이지 제목, 본문, 링크, 설명이 처음부터 HTML 안에 담겨 있으면 검색 엔진은 그 페이지의 내용을 더 분명하게 파악할 수 있다. 반대로 HTML은 비어 있고, 실제 내용이 JavaScript 실행 이후에야 늦게 채워지는 구조라면 읽기 과정이 더 복잡해진다.

이 때문에 콘텐츠 자체가 검색 유입과 연결되는 페이지는 SSR이나 정적 생성이 훨씬 유리한 경우가 많다. 블로그 글, 상품 설명, 문서 페이지, 랜딩 페이지, 카테고리 페이지가 대표적이다. 이런 페이지는 사용자뿐 아니라 검색 엔진도 내용을 읽어야 하기 때문이다. 반면 로그인 뒤에만 보이는 관리자 화면이나 내부 대시보드는 검색 노출이 핵심이 아니므로 CSR 중심이어도 문제가 되지 않는다.

SEO는 단지 검색 엔진에 잘 보이는 문구를 넣는 일이 아니다. 페이지를 어떤 방식으로 만들어 전달하는가와도 깊이 연결된다. 메타 태그와 제목도 중요하지만, 검색 엔진이 실제 내용을 언제 어떤 형태로 읽을 수 있느냐가 훨씬 근본적인 문제이다.

AI에게 이렇게 물어보세요

> "이 페이지가 SEO가 중요한 페이지인지 먼저 판단해줘. 중요하다면 SSR이나 정적 생성이 왜 유리한지, 메타데이터와 본문 구조는 어떻게 잡아야 하는지도 함께 설명해줘."

Chapter 11: 첫 화면 속도는 결국 무엇을 먼저 보여주느냐의 문제이다

사용자가 느끼는 속도는 절대 시간만으로 결정되지 않는다. 아무것도 보이지 않는 2초는 매우 길게 느껴지고, 기본 구조가 먼저 잡힌 채 세부 내용이 뒤따르는 2초는 상대적으로 짧게 느껴진다. 따라서 첫 화면 성능의 핵심은 얼마나 빨리 모든 것을 끝내는가보다, 얼마나 빨리 안심할 만한 첫 신호를 보여주는가에 있다.

프론트엔드가 무거워지는 가장 흔한 이유는 첫 화면에 필요하지 않은 것까지 한꺼번에 싣기 때문이다. 차트 라이브러리, 뒤쪽 탭 컴포넌트, 큰 이미지, 여러 위젯, 과한 스크립트가 모두 같은 우선순위로 내려오면 브라우저는 시작부터 과한 짐을 지게 된다. 그래서 프론트엔드 최적화는 전체 코드 양을 줄이는 일만이 아니라, 첫 순간에 필요한 것을 엄격하게 가리는 일이다.

Next.js의 동적 로딩, 이미지 최적화, 서버 컴포넌트와 클라이언트 컴포넌트의 구분 같은 기능이 중요한 이유도 여기에 있다. 사용자가 지금 당장 볼 필요가 없는 것, 브라우저에서만 필요한 것, 늦게 도착해도 되는 것을 구분할 수 있을 때 첫 화면은 훨씬 가벼워진다.

AI에게 이렇게 물어보세요

> "첫 화면이 느린 이유를 전체 코드 양이 아니라, 처음에 꼭 필요한 자원과 나중에 불러와도 되는 자원을 기준으로 나누어 분석해줘."

Chapter 12: AI 앱과 에이전트 프론트엔드는 무엇이 달라지는가

전통적인 웹앱 프론트엔드는 대개 목록, 폼, 검색, 필터, 상세 화면의 조합으로 이해할 수 있다. 반면 AI 앱과 에이전트 프론트엔드는 사용자가 단순히 조작만 하는 것이 아니라, 시스템의 중간 상태를 기다리고 해석하는 경험을 함께 가진다. 답이 한 번에 완성되어 도착하지 않고 스트리밍으로 흘러오거나, 지금 어떤 단계를 수행 중인지, 어떤 도구를 쓰는지, 사용자의 승인이 필요한지 같은 정보가 중요해진다.

그래서 AI 앱의 UI는 채팅창처럼 보이기 쉬우며, 에이전트 UI는 관제 화면처럼 변하기 쉽다. 사용자는 단지 결과만 보는 것이 아니라 과정도 본다. 지금 검색 중인지, 파일을 읽는 중인지, 실패했는지, 다시 시도했는지가 중요한 정보가 되기 때문이다. 이는 기존의 앱 UI와 꽤 다른 감각이다. 예쁜 카드보다 신뢰할 수 있는 진행 상태가 더 중요해질 때가 많다.

실시간과 긴 목록 문제도 여기서 다시 나타난다. 채팅은 실시간성이 중요할 수 있지만, 모든 에이전트 상태가 1초마다 갱신되어야 하는 것은 아니다. 도구 실행 이력이나 작업 목록이 길어질수록, 그 자체가 또 하나의 프론트엔드 병목이 된다. 따라서 AI 앱과 에이전트 프론트엔드를 설계할 때도 결국 같은 원칙이 돌아온다. 지금 꼭 보여줘야 하는 것과 나중에 보여줘도 되는 것을 분리하고, 사용자가 기다리는 동안 의미 있는 상태를 먼저 제공하는 것이다.

Part 3 마무리

프론트엔드는 가장 눈에 잘 띄는 층이지만, 동시에 가장 많은 구조적 판단이 숨어 있는 층이기도 하다. React, Vite, Next.js는 서로 다른 역할을 맡고 있으며, SSR과 CSR은 누가 먼저 화면을 완성하느냐의 차이이고, SEO와 첫 화면 속도는 결국 전달 방식의 문제이다. 여기에 AI 앱과 에이전트 UI까지 들어오면, 프론트엔드는 더 이상 화면 배치만의 문제가 아니라 상태와 실행을 보여 주는 층으로 확장된다.

PART 4

백엔드 1 – 앱 시대의 백엔드

서버, API, 웹훅, 외부 연동, 오래 걸리는 작업 분리처럼 전통적인 앱 백엔드의 핵심

앱 시대의 백엔드는 기본적으로 사용자 요청을 받아 판단하고 처리하는 층이다. 사용자가 화면에서 어떤 행동을 하면, 백엔드는 그 요청이 허용되는지 확인하고, 필요한 데이터를 읽고 쓰고, 외부 서비스와 통신하며, 결과를 다시 돌려준다. 겉보기에는 단순한 왕복처럼 보이지만, 실제로는 인증과 권한, API 경계, 외부 연동, 긴 작업 분리 같은 여러 판단이 여기서 이루어진다.

이 파트는 아직 AI 에이전트 백엔드로 넘어가기 전, 전통적인 앱 백엔드가 어떤 책임을 갖는지부터 정리한다. 서버가 정확히 무슨 일을 하는지, API와 웹훅은 어떤 차이가 있는지, 요청 안에 너무 많은 일을 욱여 넣으면 왜 구조가 흔들리는지, 외부 API는 왜 내 서버 밖의 병목이 되는지를 차례로 살핀다.

Chapter 13: 서버는 정확히 무슨 일을 하는가

서버는 단순히 코드를 올려 두는 장소가 아니다. 사용자의 요청을 받아 규칙을 적용하고, 데이터를 읽고 쓰고, 외부 시스템과 연결하며, 어느 정보를 누구에게 보여줄 수 있는지 결정하는 판단의 층이다. 사용자가 브라우저에서 폼을 제출하면 서버는 그 내용을 받아 유효한지 확인하고, 저장소에 기록하고, 필요한 경우 다른 시스템과도 연결한 뒤 결과를 돌려준다. 이 흐름이 바로 백엔드의 기본 동작이다.

이때 중요한 것은 서버가 `무엇을 보여줄지`보다 `무엇을 판단할지`를 맡는다는 점이다. 같은 버튼 클릭이라도 로그인 여부에 따라 허용 여부가 달라질 수 있고, 같은 데이터라도 권한에 따라 반환 범위가 달라질 수 있다. 따라서 백엔드는 단순한 데이터 통로가 아니라 규칙을 적용하는 문지기에 가깝다.

초심자는 서버를 종종 데이터베이스와 혼동하거나, 프론트엔드와 겹쳐 생각한다. 그러나 서버의 핵심은 저장 자체가 아니라 판단과 조율이다. 무엇을 저장할지, 어떤 외부 서비스와 언제 통신할지, 실패했을 때 어디서 재시도할지, 어떤 정보는 지금 당장 돌려주고 어떤 정보는 나중에 처리할지를 정하는 역할이 모두 여기에 속한다.

이 차이를 실감하는 가장 쉬운 예는 인증이다. 화면에서는 로그인 버튼과 입력창만 보인다. 데이터베이스에는 사용자 정보가 저장되어 있다. 그러나 실제로 "이 사용자가 누구인지", "이 요청이 허용되는지", "이 계정이 이 데이터를 볼 권한이 있는지"를 판단하는 일은 서버의 몫이다. 화면이 직접 그 결론을 내려서는 안 되고, 저장소가 혼자 그 결론을 설명하지도 않는다. 백엔드는 바로 이런 규칙의 층을 맡는다.

또 하나 자주 놓치는 지점은 서버가 단지 내부 데이터만 다루지 않는다는 점이다. 결제, 이메일, 메시징, 지도, AI 모델 호출 같은 외부 서비스는 이제 현대 앱의 일부가 되었다. 따라서 백엔드는 내부 규칙을 지키는 관리자이면서 동시에 바깥 세계와 연결되는 통역기이기도 하다. 사용자가 버튼 하나를 누른 뒤 실제로는 여러 시스템이 연쇄적으로 움직이는 이유가 여기에 있다.

그래서 백엔드를 이해할 때는 "서버에 코드가 있다" 정도로 끝내지 않는 편이 좋다. 더 정확한 표현은 이렇다. 백엔드는 사용자 요청을 해석하고, 서비스의 규칙을 적용하고, 저장 계층과 외부 시스템을 조율해 결과를 돌려주는 층이다. 이 관점이 있어야 API도, 웹훅도, 큐도, 외부 의존성 문제도 모두 하나의 그림으로 이어진다.

Chapter 14: API와 웹훅은 무엇인가

API는 프로그램끼리 대화하는 약속이다. 프론트엔드가 백엔드에게 데이터를 달라고 요청하고, 백엔드가 정해진 형식으로 응답하는 것이 가장 익숙한 API의 모습이다. 중요한 것은 API가 단지 URL 목록이 아니라, 프론트엔드와 백엔드 사이의 책임 분리를 드러낸다는 점이다. 프론트엔드는 무엇을 원한다고 말하고, 백엔드는 그 요청을 받아 판단한 뒤 결과를 돌려준다.

웹훅은 이 대화의 방향이 다르다. 보통 API는 내가 상대에게 요청하는 구조이지만, 웹훅은 어떤 사건이 발생했을 때 상대가 나에게 알려주는 구조에 가깝다. 결제가 완료되었을 때 결제 서비스가 내 서버를 호출하거나, GitHub가 이벤트 발생 사실을 내 서버로 보내는 방식이 대표적이다. 따라서 앱 시대의 백엔드는 사용자의 브라우저뿐 아니라 외부 서비스와도 API와 웹훅을 통해 끊임없이 대화한다.

이 구분은 나중에 AI 백엔드를 이해할 때도 중요하다. API는 여전히 핵심 통신 수단으로 남지만, 그것만으로는 에이전트의 상태와 툴 실행을 설명하기 어려워진다. 그래서 전통적인 앱 시대 백엔드에서는 먼저 API와 웹훅이라는 기본 대화법을 확실히 이해하는 편이 좋다.

초심자에게는 API가 종종 마법처럼 보인다. 프론트엔드에서 `fetch` 한 줄을 쓰면 어디선가 데이터가 돌아오기 때문이다. 그러나 그 안에서는 꽤 많은 일이 일어난다. 브라우저가 요청을 만들고, 서버는 그 요청의 형식과 인증 상태를 확인하고, 필요한 데이터를 읽거나 계산하고, 오류가 있으면 그 이유를 정리해 돌려주어야 한다. 즉 API는 단순한 출입문이 아니라, 서비스의 규칙을 바깥에 드러내는 표면이다.

이 때문에 좋은 API를 만든다는 것은 단지 엔드포인트를 많이 만드는 일이 아니다. 어떤 요청은 어디까지 허용할지, 어떤 입력이 잘못되었을 때 어떤 형식으로 알려줄지, 성공과 실패를 어떤 의미로 나눌지를 정하는 일이다. 겉보기에는 기술적 선택처럼 보여도, 실제로는 서비스가 자기 질서를 외부에 설명하는 방식에 가깝다.

웹훅도 비슷하다. 결제 서비스가 "결제가 끝났다"고 알려줄 때, 그것은 단순한 알림이 아니라 `이제 당신 쪽 상태를 바꾸어도 된다`는 사건 신호이다. 따라서 웹훅을 다룰 때는 속도보다 정확성이 중요해진다. 같은 이벤트가 중복으로 와도 안전해야 하고, 잠시 늦게 와도 전체 흐름이 틀어지지 않아야 한다. 이 점에서 웹훅은 API의 변형이라기보다, 백엔드가 이벤트를 받아들이는 방식에 가깝다.

앱 시대의 백엔드가 성숙해진다는 것은 결국 이 두 언어를 구분해서 다루게 된다는 뜻이다. 사용자가 직접 요청하는 대화는 API로, 외부 시스템이 사건을 통보하는 대화는 웹훅으로 이해하면 구조가 훨씬 선명해진다.

Chapter 15: 오래 걸리는 일은 요청과 분리해야 한다

서버를 불안정하게 만드는 가장 흔한 구조 중 하나는, 한 번의 요청 안에 너무 많은 일을 몰아넣는 것이다. 사용자가 버튼을 누르면 서버는 데이터를 저장하고, 재고를 확인하고, 결제를 호출하고, 이메일을 보내고, 이미지도 처리하고, 외부 서비스에도 알림을 보내는 식이다. 처음에는 단순해 보인다. 한 함수 안에 모든 흐름이 모여 있으니 이해하기 쉬워 보이기 때문이다.

그러나 실제 운영에서는 이 단순함이 곧 병목이 된다. 외부 API 하나라도 느려지면 전체 요청이 길어진다. 이미지 처리나 PDF 생성처럼 시간이 오래 걸리는 작업이 끼어 있으면, 가벼운 요청까지 함께 지연된다. 서버리스 환경이라면 시간 제한에 걸릴 가능성도 커진다. 무엇보다 사용자는 "요청이 접수되었는가"를 알고 싶은데, 시스템은 "모든 후속 처리까지 끝나야 응답한다"는 구조를 취하고 있게 된다.

이 문제를 풀기 위한 가장 기본적인 사고는 접수와 처리를 나누는 것이다. 사용자가 요청을 보냈을 때 서버가 가장 먼저 해야 할 일은 요청을 안정적으로 받아두는 것이다. 그 뒤에 오래 걸리거나 재시도가 가능한 처리, 외부 의존성이 많은 처리는 별도의 흐름에서 수행하는 편이 낫다. 이를 위해 쓰는 대표적인 구조가 큐이다.

큐는 기술적으로 복잡해 보이지만, 개념은 매우 단순하다. 지금 당장 다 끝내지 말고, 먼저 줄을 세워 안정적으로 처리하자는 방식이다. 식당에서 주문을 받는 사람과 요리를 하는 사람이 나누어져 있는 이유와 같다. 주문을 받는 사람이 직접 주방까지 뛰어들어 모든 요리를 끝내고 돌아오면, 카운터는 곧 막히게 된다. 반면 주문은 먼저 접수하고, 실제 조리는 뒤에서 순서대로 처리하면 흐름이 훨씬 안정된다.

초심자에게 큐가 중요한 이유는 성능만이 아니다. 실패를 다루는 방식까지 바꾸기 때문이다. 이메일 발송이나 외부 웹훅 전송 같은 작업은 한 번 실패했다고 해서 즉시 서비스 전체를 실패로 만들 필요가 없는 경우가 많다. 잠시 후 다시 시도하면 되는 일은, 사용자 요청과 분리되어 있는 편이 훨씬 안정적이다. 큐는 바로 이런 성격의 작업을 다루기 좋다.

따라서 어떤 작업이 요청 안에 남아 있어야 하고, 어떤 작업이 뒤로 밀려나도 되는지를 구분하는 감각이 필요하다. 로그인처럼 즉시 결과를 보여주어야 하는 작업은 요청 안에서 끝나야 한다. 반대로 메일 발송, 리포트 생성, 이미지 후처리, 외부 동기화처럼 시간이 길고 재시도가 가능한 작업은 큐로 빼는

편이 자연스럽다. 구조를 이렇게 나누는 순간부터, 서버는 단순히 빠른 기계가 아니라 흐름을 조정하는 시스템이 된다.

AI에게 이렇게 물어보세요

> "이 API가 요청 안에서 너무 많은 일을 처리하고 있는지 판단해줘. 어떤 단계는 즉시 응답 전에 끝나야 하고, 어떤 단계는 큐로 분리해도 되는지 이유와 함께 설명해줘."
>
> "주문 처리, 이메일 발송, 외부 API 호출이 한 함수 안에 들어 있다. 접수와 처리를 분리하는 구조로 다시 설계해줘."

Chapter 16: 외부 API는 내 서버 밖의 병목이다

많은 서비스는 이제 단독으로 동작하지 않는다. AI API, 결제 API, 이메일 API, 지도 API, 메시징 API 같은 외부 의존성 위에 서 있다. 이들은 매우 유용하지만, 동시에 내 서비스의 속도와 안정성을 일정 부분 외부에 맡긴다는 뜻이기도 하다. 내 코드가 아무리 단정해도, 외부 API가 느리거나 응답을 주지 않으면 사용자는 결국 느린 내 서비스를 만나게 된다.

초심자가 외부 API를 붙일 때 가장 자주 놓치는 지점은 대기 시간이다. 외부 호출은 단지 함수 한 줄이 아니다. 요청을 보내고, 네트워크를 타고, 상대 서버가 처리하고, 결과가 돌아오기까지 전체 시간이 필요하다. 이 시간이 길어질수록 내 함수는 "아무 일도 하지 않는 듯 보이지만 사실은 기다리는 상태"로 오래 붙들린다. 서버리스 환경에서는 이 대기 시간도 결국 함수 실행 시간과 비용에 반영된다.

따라서 외부 API를 다룰 때는 기능 연결 자체보다 실패와 지연을 어떻게 흡수할지가 중요하다. 답을 아주 길게 기다려야 하는가, 사용자는 중간 상태를 볼 수 있는가, 같은 요청을 매번 다시 보내고 있는가, 외부 서비스가 일시적으로 느릴 때 내 서비스는 어떤 태도를 취하는가를 먼저 생각해야 한다.

여기서 핵심이 되는 방식이 세 가지 있다. 첫째, 사용자가 기다리는 시간을 그대로 방치하지 않고 스트리밍이나 중간 상태 표시로 흡수하는 것이다. 둘째, 외부 API가 잠시 흔들릴 때 즉시 포기하거나 무한 재시도하지 않고, 적절한 간격을 두고 다시 시도하는 것이다. 셋째, 같은 질문과 같은 요청에 대해 매번 외부 API를 새로 부르지 않도록 캐시나 결과 저장을 활용하는 것이다.

이 세 가지는 서로 다른 기술처럼 보이지만, 사실 같은 원리 위에 있다. 외부 API를 "항상 즉시, 항상 완벽하게 응답해 주는 내부 함수"처럼 다루지 않는다는 원리이다. 외부 서비스는 느릴 수 있고, 실패할 수 있고, 비용도 들 수 있다. 이 전제를 받아들이는 순간부터 구조가 달라진다.

여기서 특히 중요한 것은 외부 API 지연을 내 서버의 실패와 동일시하지 않는 것이다. 외부 API가 잠시 느린 것은 흔한 일이다. 그때 사용자가 아무 설명도 없이 하염없이 기다리게 만들 것인지, "처리 중"이라는 상태를 먼저 보여주고 나중에 완료를 알릴 것인지, 혹은 같은 요청이면 이전 결과를 재사용할 것인지가 서비스 품질을 가른다.

AI에게 이렇게 물어보세요

> "이 외부 API 호출이 내 서비스 전체를 느리게 만들고 있다. 어떤 부분은 즉시 기다리고, 어떤 부분은 나중 처리로 넘기고, 어떤 부분은 캐시해야 하는지 구조적으로 설명해줘."
>
> "AI API 응답이 느릴 때 사용자 경험을 망치지 않으려면 스트리밍, 재시도, 캐시를 각각 언제 써야 하는지 비교해줘."

Part 4 마무리

서버를 안정적으로 운영한다는 것은 단순히 더 빠른 장비를 쓰는 일이 아니다. 서버가 어디에서 시간을 잃는지, 무엇을 한 요청 안에 과도하게 몰아넣고 있는지, 외부 의존성이 흔들릴 때 구조가 어떤 태도를 취하는지를 이해하는 일이다.

서버리스의 한계를 아는 것은 서버리스를 버리기 위함이 아니라, 그 장점을 무리 없이 오래 활용하기 위함이다. 오래 걸리는 작업을 큐로 분리하는 것은 복잡함을 늘리기 위함이 아니라, 요청의 흐름을 안정시키기 위함이다. 외부 API를 신중하게 다루는 것은 남의 문제를 대신 떠안지 않기 위함이다.

이 관점이 자리 잡으면 서버는 단지 코드가 실행되는 장소가 아니라, 요청과 시간과 실패를 다루는 구조로 보이기 시작한다.

PART 5

백엔드 2 – AI와 에이전트 시대의 백엔드

모델 호출, MCP, 툴, 스킬, 멀티 에이전트, 플랫폼까지 AI 백엔드의 새로운 층위

앱 시대의 백엔드는 주로 요청을 받아 처리하고, 필요한 데이터를 읽고 쓰고, 결과를 돌려주는 층이었다. 그 구조만으로도 충분히 복잡했지만, AI와 에이전트가 본격적으로 들어오면서 백엔드는 한 단계 더 많은 책임을 떠안게 되었다. 이제 백엔드는 단순히 정답을 돌려주는 곳이 아니라, 모델을 호출하고, 스트리밍 응답을 흘려보내고, 외부 도구를 붙이고, 실행 상태를 유지하고, 때로는 하나의 목표를 여러 단계로 분해해 움직이는 조정자가 되었다.

이 변화는 단순한 유행이 아니다. 기존 백엔드 위에 새로운 층이 얹히면서 생긴 구조적 변화이다. API는 여전히 필요하고, 저장소도 여전히 필요하며, 인증과 비용과 운영도 여전히 중요하다. 다만 그 위에 모델, 툴, 함수 호출, MCP, 스킬, 에이전트, 멀티 에이전트, 플랫폼이라는 새로운 이름들이 더해졌다. 이 파트의 목적은 그 이름들을 따로 떼어 암기하게 만드는 데 있지 않다. 각각이 어떤 책임을 갖고, 어디에서 만나며, 왜 서로를 대체하지 못하는지를 또렷하게 이해하게 만드는 데 있다.

Chapter 17: AI 앱의 백엔드는 무엇이 다른가

일반적인 앱 백엔드는 사용자의 요청을 받아 데이터를 읽고 쓰고, 권한을 확인하고, 외부 서비스와 통신한 뒤 결과를 반환한다. 구조 자체는 분명하다. 입력이 있고, 판단이 있고, 저장이 있고, 응답이 있다. AI 앱의 백엔드는 여기에 하나의 거대한 생성 계층을 더 얹은 형태이다. 사용자의 말을 바로 규칙으로 처리하는 것이 아니라, 모델에게 문맥을 주고 해석과 생성을 맡긴 뒤, 그 결과를 다시 서비스의 흐름에 맞게 정리해야 한다.

이 차이는 단순히 `모델 API를 한 번 더 호출한다` 는 수준으로 끝나지 않는다. 모델은 일반적인 함수보다 예측 가능성이 낮고, 응답의 길이와 시간도 일정하지 않다. 같은 질문이라도 매번 조금씩 다른 결과가 나오며, 입력이 조금만 길어져도 토큰 비용과 지연 시간이 빠르게 늘어난다. 따라서 AI 앱 백엔드는 출력 자체보다, 그 출력이 언제, 어떤 방식으로, 어느 정도 신뢰도로 제공되는지를 관리하는 쪽에 더 가깝다.

예를 들어 사용자가 긴 글을 요약해 달라고 요청했다고 하자. 전통적인 백엔드라면 작업이 끝난 뒤 결과를 한 번에 돌려주는 방식이 자연스럽다. 그러나 AI 앱에서는 길이가 긴 생성이므로 중간 진행 상황을 보여주는 편이 훨씬 낫다. 사용자는 모델이 생각하고 있다는 감각을 원하고, 시스템은 그 사이에 실패를 감지하고 재시도할 여지도 가져야 한다. 이 때문에 AI 앱 백엔드는 응답을 단순한 결과값으로 보는 것이 아니라, 스트림, 상태, 재시도, 캐시, 관찰 가능성까지 포함한 실행 과정으로 다루게 된다.

또 하나 중요한 점은, AI 앱에서는 입력이 곧바로 출력으로 이어지지 않는다는 사실이다. 같은 사용자의 질문이라도 프롬프트 템플릿, 시스템 메시지, 대화 기록, 검색 결과, 사용자 권한, 비용 정책에 따라 전혀 다른 실행 경로를 갖게 된다. 그래서 AI 앱 백엔드는 모델을 붙이는 곳이 아니라, 모델이 흔들리지 않도록 경계 조건을 만들어 주는 곳이라고 보는 편이 정확하다. 좋은 AI 앱 백엔드는 똑똑한 모델보다 먼저, 모델이 어디서 멈추고 어디서 도구를 쓰며 어디서 사람의 확인을 기다려야 하는지를 정의한다.

실무에서는 이 지점에서 처음 혼란이 생긴다. 많은 사람이 AI 기능을 넣으면 백엔드가 단순해질 것이라고 기대하지만, 실제로는 반대이다. 프롬프트 관리, 토큰 예산, 응답 지연, 부분 실패, 로그 보관, 입

력 검증, 안전장치, 사용자 피드백까지 모두 백엔드의 책임이 된다. 즉 AI 앱의 백엔드는 계산을 대신해 주는 층이 아니라, 생성형 불확실성을 제품의 질서로 바꾸는 층이다.

Chapter 18: 툴, 함수 호출, MCP는 무엇이 다른가

AI와 에이전트 이야기를 시작하면 가장 먼저 용어가 엉킨다. 함수 호출, 툴 사용, MCP가 모두 비슷한 말처럼 들리기 때문이다. 그러나 이 셋은 같은 층의 개념이 아니다. 차이를 분명히 해 두어야 이후의 에이전트 구조가 흔들리지 않는다.

함수 호출은 모델이 구조화된 출력을 만들어 특정 작업을 요청하는 방식이다. 모델은 자연어를 그대로 길게 늘어놓는 대신, 어떤 함수를 어떤 인자로 실행하면 되는지를 보여 주는 형식의 출력을 만든다. 실제 실행은 모델이 하지 않는다. 실행은 여전히 백엔드의 몫이다. 이 구조가 중요한 이유는, 모델이 말로만 지시하는 것이 아니라 백엔드가 기계적으로 확인할 수 있는 형태로 의도를 전달할 수 있기 때문이다.

툴은 그보다 넓은 말이다. 검색, 파일 읽기, 데이터 조회, 일정 생성, 결제 확인, 외부 API 호출처럼 에이전트가 사용할 수 있는 외부 능력 전체를 툴이라고 부를 수 있다. 함수 호출은 툴을 부르는 한 방식이고, 툴은 함수보다 더 넓은 실행 자원이다. 따라서 툴을 이해한다는 것은 단순히 함수 시그니처를 보는 일이 아니라, 에이전트가 무엇을 할 수 있는지, 어디까지 직접 움직일 수 있는지를 보는 일이다.

MCP는 여기서 또 다른 층이다. MCP는 `Model Context Protocol`의 약자로, 에이전트가 도구와 맥락에 접속하는 방식을 표준화하려는 인터페이스에 가깝다. API가 서비스와 서비스 사이의 대화 규칙이라면, MCP는 에이전트와 도구 사이의 연결 규칙이라고 이해하면 된다. 이 차이는 중요하다. API는 시스템끼리 데이터를 주고받는 데 초점이 있고, MCP는 모델이 도구를 발견하고 호출하고 문맥을 주고받는 과정을 더 일관되게 다루는 데 초점이 있다.

따라서 `API와 MCP는 무엇이 다른가`라는 질문에는 이렇게 답할 수 있다. API는 여전히 서비스의 기본 계약이다. 백엔드끼리 통신할 때도 API를 쓰고, 프론트엔드가 백엔드를 부를 때도 API를 쓴다. MCP는 그 위에서 에이전트가 툴을 발견하고 사용할 수 있게 해 주는 연결 규격이다. API가 도로망이라면, MCP는 그 위에서 에이전트가 목적지와 중간 정류장을 더 잘 이해하게 해 주는 표준 장치에 가깝다. 둘은 경쟁하는 개념이 아니라, 서로 다른 문제를 푸는 개념이다.

이 구분이 흐려지면 시스템 설계도 같이 흐려진다. 어떤 사람은 툴이면 다 MCP라고 부르고, 어떤 사람은 API를 그대로 에이전트 프로토콜처럼 사용하려고 한다. 그러나 실제로는 API가 데이터와 기능

을 노출하는 방식이고, MCP는 그 노출된 능력을 에이전트가 쓰기 좋게 묶는 방식이다. 이 둘을 같은 말처럼 쓰지 않아야, 이후에 멀티 에이전트나 플랫폼으로 확장할 때 구조가 무너지지 않는다.

Chapter 19: 에이전트 스킬은 무엇이며 어디에 놓이는가

스킬은 도구의 목록이 아니다. 스킬은 도구를 어떤 순서와 조건으로 사용해야 하는지를 포함한 실행 능력이다. 같은 검색 툴과 같은 파일 읽기 툴이 있어도, 어떤 에이전트는 문서를 잘 요약하고 어떤 에이전트는 맥락을 놓친다. 차이는 툴 유무가 아니라, 툴을 조합하는 방식에 있다. 그 조합이 바로 스킬이다.

이 개념을 오해하면 시스템은 금방 어수선해진다. 많은 초기 구현이 `툴만 많이 붙이면 에이전트가 알아서 잘할 것`이라고 기대한다. 그러나 실제로는 그렇지 않다. 어떤 문제에서 어떤 툴을 먼저 쓰고, 중간에 결과가 모호하면 다시 확인하고, 권한이 필요한 단계에서는 사람의 승인을 받고, 실패하면 어느 시점까지 되돌아갈지까지 정해 주어야 한다. 이러한 절차가 스킬이다. 스킬은 에이전트가 할 수 있는 일을 늘리는 것이 아니라, 할 수 있는 일을 안정적으로 재현하게 만드는 장치이다.

예를 들어 `문서 조사 스킬`은 단순히 검색 툴 하나를 호출하는 것이 아니다. 질문을 분해하고, 필요한 자료를 찾고, 신뢰도를 비교하고, 상충하는 정보를 걸러내고, 마지막에 결과를 정리하는 흐름 전체를 포함한다. `코드 리뷰 스킬`도 마찬가지이다. 코드 파일을 읽고, 변경 맥락을 파악하고, 위험한 부분을 찾고, 테스트 관점에서 보완점을 제시하는 일련의 판단이 함께 들어간다. 스킬이란 결국 `무엇을 할 수 있는가`보다 `무엇을 잘하게 만들 것인가`를 설계하는 층이다.

이 때문에 스킬은 백엔드에서 따로 관리되는 경우가 많다. 모델 자체가 스킬을 내장하는 것이 아니라, 백엔드가 스킬을 정의하고, 그 스킬에 필요한 툴과 프롬프트와 검증 규칙을 묶는다. 즉 스킬은 모델의 재능이 아니라 시스템의 습관이다. 좋은 시스템은 모델이 우연히 잘하는 순간에 기대지 않고, 원하는 행동을 반복 가능하게 만든다. 스킬은 바로 그 반복 가능성을 담당한다.

또한 스킬은 사용자 경험과도 연결된다. 사용자는 모델이 내부에서 어떤 툴을 몇 번 호출했는지보다, 결과가 예측 가능하고 일관되게 나오는지를 더 중요하게 느낀다. 따라서 스킬은 눈에 보이지 않지만 매우 실용적인 계층이다. 에이전트가 똑똑해 보이는가보다, 같은 유형의 문제를 계속 안정적으로 해결하는가가 더 중요하다. 스킬은 그 안정성을 만든다.

Chapter 20: 멀티 에이전트, 에이전트 플랫폼, 에이전틱 인터넷은 어떻게 이어지는가

멀티 에이전트는 여러 에이전트가 역할을 나누어 일하는 구조이다. 조사하는 에이전트, 계획을 세우는 에이전트, 실행하는 에이전트, 검토하는 에이전트를 따로 둘 수 있다. 겉으로 보면 팀을 흉내 내는 구조처럼 보이지만, 실제 의미는 분리와 조율이다. 하나의 에이전트 안에 모든 책임을 욱여 넣으면 상태가 복잡해지고, 실패 원인을 찾기 어려워지며, 비용도 통제하기 힘들어진다. 역할을 나누면 복잡성이 사라지는 것은 아니지만, 최소한 어디에서 문제가 생겼는지는 보이게 된다.

다만 멀티 에이전트는 자동으로 더 좋은 구조가 되지는 않는다. 에이전트 수가 늘수록 메시지 흐름은 길어지고, 상태 공유는 어려워지고, 지연 시간은 늘어난다. 어떤 에이전트가 어떤 입력을 받고, 어떤 에이전트가 결과를 검증하며, 실패했을 때 누가 되돌아가고, 어떤 단계에서 사람의 확인이 필요한지를 명확히 설계해야 한다. 멀티 에이전트는 화려함을 더하기 위한 장치가 아니라, 복잡한 일을 분해하기 위한 장치이다.

이 구조를 실제 서비스로 돌리려면 플랫폼이 필요하다. 에이전트 플랫폼은 개별 에이전트를 한번 실행해 보는 도구가 아니라, 에이전트를 만들고, 버전 관리하고, 툴과 권한을 연결하고, 실행 로그를 보고, 상태를 추적하고, 실패를 복구하고, 여러 팀원이 함께 운영할 수 있게 해 주는 층이다. 즉 플랫폼은 에이전트의 실행 환경이면서 동시에 관제실이다.

플랫폼을 이해하는 방식도 중요하다. 플랫폼은 단순한 UI가 아니다. 플랫폼은 에이전트를 관리 가능한 소프트웨어로 바꾸는 운영 계층이다. 에이전트가 무엇을 했는지 나중에 추적할 수 있어야 하고, 어떤 툴을 썼는지 확인할 수 있어야 하며, 어느 단계에서 비용이 늘었는지도 보여야 한다. 이 수준의 관리가 없다면 에이전트는 데모에는 강하지만 운영에는 약한 구조로 남는다.

하지만 플랫폼보다 더 바깥에는 또 하나의 층이 생길 수 있다. 그것이 에이전틱 인터넷이다. 플랫폼이 특정 조직이나 생태계 안에서 에이전트를 만들고 운영하는 기반이라면, 에이전틱 인터넷은 서로 다른 사람과 서로 다른 업체가 가진 에이전트들이 공통 규약 위에서 직접 소통하고 거래하는 더 큰 네트워크를 가리킨다. 다시 말해 플랫폼은 에이전트를 굴리는 기반이고, 에이전틱 인터넷은 그런 기반들 위에서 형성되는 연결 질서이다.

이 지점을 생각하면 그림이 달라진다. 각 개인이 자신의 에이전트를 갖고, 각 업체도 자기 서비스의 에이전트를 갖고, 이들이 서로 메시지를 주고받으며 거래와 예약과 협상을 수행하는 상황이 가능해진다. 사용자의 휴대전화 안에 있는 개인 에이전트가 피자집의 주문 에이전트와 대화하고, 결제 에이전트와 확인을 주고받고, 배달 에이전트와 상태를 맞추는 식의 세계이다. 이것은 단순한 멀티 에이전트나 하나의 플랫폼 안의 자동화보다 더 큰 범위의 발상이다. 말하자면 한 서비스 내부의 분업을 넘어, 에이전트끼리 연결된 사회적 네트워크에 가까워진다.

그래서 에이전틱 인터넷은 지금의 인터넷과도 비슷하면서 다르다. 지금의 인터넷이 사람과 서비스가 브라우저와 앱을 통해 연결되는 구조라면, 에이전틱 인터넷은 사람의 에이전트와 서비스의 에이전트가 공통 규약 위에서 직접 소통하는 구조를 가리킨다. 사용자가 일일이 앱을 열고 버튼을 누르지 않아도, 자신의 에이전트가 필요한 서비스를 찾아가고, 조건을 확인하고, 예약을 넣고, 결제를 확인하고, 결과를 다시 가져올 수 있는 세계이다. 사람이 모든 화면을 직접 통과하는 인터넷에서, 에이전트가 상당수의 상호작용을 대신 수행하는 인터넷으로 무게중심이 이동하는 셈이다.

이 구조가 성립하려면 단순히 에이전트가 많아지는 것만으로는 부족하다. 누가 누구인지 증명하는 신원 체계가 필요하고, 어떤 행동을 대신할 수 있는지에 대한 권한 위임이 필요하며, 메시지 형식과 상태 전달 방식이 맞아야 하고, 결제나 계약처럼 책임이 따르는 행위에는 기록과 검증이 뒤따라야 한다. 사람끼리 인터넷에서 거래할 때도 로그인, 결제, 영수증, 약관이 필요한 것처럼, 에이전트끼리 거래하는 인터넷에도 신뢰와 검증의 질서가 필요하다. 그래서 에이전틱 인터넷은 단순히 멋진 미래상이 아니라, 프로토콜과 권한과 신뢰의 문제를 함께 품은 개념이다.

개인 에이전트와 서비스 에이전트를 구분해 보는 것도 도움이 된다. 개인 에이전트는 사용자의 일정과 취향과 우선순위를 알고, 사용자를 대신해 결정을 보조하거나 대행하는 주체이다. 반면 서비스 에이전트는 피자집, 병원, 쇼핑몰, 항공사처럼 특정 조직의 정책과 재고와 가격과 예약 가능 시간을 대변한다. 개인 에이전트가 "오늘 저녁 여덟 시 전에 배달 가능한 피자를 찾아 달라"고 요청하면, 서비스 에이전트는 현재 재고와 배달 가능 시간과 가격 조건을 바탕으로 응답한다. 이 상호작용이 여러 서비스와 표준화된 방식으로 반복된다면, 그것은 더 이상 개별 앱 호출의 합이 아니라 에이전트 네트워크의 초기 형태라고 볼 수 있다.

이런 세계에서는 사용자 경험의 단위도 달라진다. 오늘날에는 사용자가 여러 앱을 직접 오가며 비교하고 입력하고 확인한다. 그러나 에이전틱 인터넷에서는 사용자가 의도를 한 번 표현하고, 자신의 에이전트가 여러 서비스 에이전트와 대화한 뒤 결과를 가져오는 쪽으로 바뀔 수 있다. 화면의 수가 줄어드

는 대신, 신뢰할 수 있는 대리 실행이 중요해진다. 사용자는 예쁜 UI보다 내 에이전트가 무엇을 누구와 약속했는지, 어디까지 권한을 넘겼는지, 문제가 생기면 어떻게 되돌릴 수 있는지를 더 중요하게 여기게 된다.

따라서 이 장에서 말하는 세 개념은 위계를 가진 하나의 선 위에 놓인다. 멀티 에이전트는 한 시스템 안에서 일을 어떻게 나눌지의 문제이다. 에이전트 플랫폼은 그렇게 나뉜 실행을 어떻게 만들고 관찰하고 통제할지의 문제이다. 에이전틱 인터넷은 그보다 더 바깥에서, 사람의 에이전트와 업체의 에이전트와 여러 플랫폼이 네트워크처럼 연결되는 문제이다. 처음에는 하나의 에이전트와 몇 개의 툴만으로도 충분할 수 있다. 그러나 스킬이 쌓이고, 역할이 나뉘고, 플랫폼이 생기고, 서로 다른 주체의 에이전트가 연결되기 시작하면, 멀티 에이전트와 플랫폼과 에이전틱 인터넷은 같은 시대의 서로 다른 층으로 이어지게 된다.

PART 6

저장소

DB, 캐시, 동시 쓰기와 같은 저장 계층의 병목을 읽는 법

서비스가 느려지면 많은 사람은 가장 먼저 서버 사양을 떠올린다. CPU가 부족한가, 메모리가 모자란가, 더 큰 인스턴스로 옮겨야 하나 같은 생각이 먼저 든다. 그러나 실제로는 서버보다 데이터베이스가 먼저 지치는 경우가 훨씬 많다. 앱이 하는 일의 대부분이 결국 데이터를 읽고 쓰는 일로 수렴하기 때문이다. 로그인도 데이터를 읽는 일이고, 목록 페이지도 데이터를 읽는 일이며, 좋아요나 댓글도 데이터를 쓰는 일이다. 사용자가 체감하는 거의 모든 핵심 기능 뒤에는 데이터베이스와의 왕복이 놓여 있다.

초심자에게 데이터베이스는 종종 막연한 공포의 대상이다. SQL 문법도 낯설고, 인덱스나 캐시 같은 말은 어렵게 들린다. 그러나 서비스 운영의 관점에서 보면 데이터베이스 문제는 의외로 일정한 패턴으로 반복된다. 읽는 방식이 비효율적이거나, 같은 데이터를 지나치게 많이 다시 읽거나, 동시에 많은 쓰기가 몰릴 때 순서가 꼬이거나 하는 문제들이다. 겉으로는 복잡해 보여도 뿌리는 대체로 그 안에 있다.

이 파트는 데이터베이스를 이론으로 배우기보다는, 서비스가 실제로 왜 느려지는지를 이해하는 쪽에 초점을 둔다. 무엇이 잘못된 조회 습관인지, 캐시는 정확히 무엇을 줄이는 도구인지, 쓰기가 몰릴 때 왜 숫자가 틀어지는지, 그리고 그런 문제를 어떻게 생각해야 하는지를 차근차근 설명할 것이다.

Chapter 21: 느린 서비스의 뒤편에서 벌어지는 일

데이터가 적을 때는 거의 모든 코드가 그럴듯해 보인다

앱을 막 만들었을 때는 데이터가 거의 없다. 게시글이 몇 개 안 되고, 사용자는 손에 꼽히고, 댓글도 얼마 없기 마련이다. 이 시기에는 쿼리가 다소 비효율적이어도 쉽게 티가 나지 않는다. 테이블 전체를 읽어도 금방 끝나고, 목록을 불러온 뒤 항목마다 다시 조회를 해도 체감상 큰 문제가 없다. 개발자는 "문제없다"는 판단을 내리기 쉽고, 실제로도 테스트 단계에서는 잘 돌아간다.

문제는 그 판단이 코드의 질을 보장하는 것은 아니라는 점이다. 데이터가 적어서 문제가 드러나지 않았을 뿐일 수 있다. 서비스가 성장해 데이터가 쌓이면, 초기에는 아무렇지 않던 구조가 갑자기 무거워진다. 게시글 수가 수십 개일 때는 괜찮던 쿼리가 수만 개가 되면 부담이 되고, 수십만 개가 되면 병목이 된다. 많은 장애가 "갑자기 생긴 것"처럼 보이는 이유가 여기에 있다. 실제로는 오래전부터 존재하던 비효율이 데이터 규모가 커지면서 비로소 눈에 띄기 시작한 것이다.

따라서 느려지는 서비스 앞에서 가장 먼저 가져야 할 태도는 "어제까지는 잘 됐는데"라는 감정적 반응보다 "이 구조가 지금 데이터 규모에서도 여전히 합리적인가"라는 질문이다. 그 질문이 생기는 순간부터, 데이터베이스는 두려운 블랙박스가 아니라 구조를 점검할 수 있는 대상이 된다.

N+1 문제는 초심자가 가장 먼저 맞닥뜨리는 병목이다

바이브 코딩으로 앱을 만들다 보면 아주 자주 등장하는 패턴이 있다. 목록을 먼저 읽고, 그 목록의 각 항목에 대해 다시 데이터를 읽는 구조이다. 예를 들어 게시글 목록을 가져온 다음 각 게시글의 댓글 수를 다시 세거나, 주문 목록을 가져온 뒤 각 주문의 사용자 정보를 따로 읽거나, Firestore에서 게시글 문서를 읽은 뒤 작성자 문서를 다시 하나씩 불러오는 식이다.

이 구조는 처음 보면 자연스럽다. 인간이 머릿속으로 생각하는 순서와 비슷하기 때문이다. 먼저 게시글을 가져오고, 그다음 각 게시글에 필요한 정보를 보충한다. 그러나 데이터베이스 입장에서는 이것이 대단히 비효율적일 수 있다. 목록을 한 번 읽고 끝날 일을, 결과의 개수만큼 반복해서 왕복하게 만들기 때문이다. 이를 흔히 N+1 문제라고 부른다. 결과가 N개일 때, 기본 조회 1번에 더해 N번의 추가 조회가 붙는 구조라는 뜻이다.

이 문제가 위험한 이유는 쿼리 한 번 한 번이 길지 않아도 전체 왕복 횟수가 급격히 늘어난다는 데 있다. 데이터베이스는 메모장 파일이 아니다. 질문을 보내고, 실행하고, 응답을 받아야 한다. 이 왕복이 스무 번, 쉰 번, 백 번으로 늘어나면, 개별 쿼리가 빠르더라도 전체 응답은 무거워질 수밖에 없다.

초심자에게 이 문제를 가장 쉽게 설명하는 방법은 택배 비유이다. 상자 스무 개를 한 번에 건네주면 될 일을, 상자 하나 줄 때마다 택배 기사를 다시 부르는 식이다. 개별 행동은 작아 보여도, 전체 시스템은 훨씬 큰 부담을 짊어진다.

데이터베이스가 잘하는 일은 애플리케이션이 대신하지 않는 편이 낫다

N+1 문제를 해결할 때 핵심은 쿼리를 화려하게 만드는 데 있지 않다. 데이터베이스가 본래 잘하는 일을 데이터베이스에게 맡기는 데 있다. 관계를 묶고, 집계를 하고, 조건에 맞춰 정렬하고, 필요한 결과를 한 번에 가져오는 일은 데이터베이스의 본업이다. 애플리케이션이 목록을 하나씩 돌며 정보를 짜깁기하는 구조는 읽기에는 직관적일 수 있어도, 규모가 커질수록 비효율을 감추기 어렵다.

그래서 목록과 관련 데이터가 함께 필요할 때는 조인, 집계 쿼리, 배치 조회 같은 방식을 먼저 떠올리는 편이 낫다. Firestore처럼 관계형 데이터베이스가 아닌 환경이라도 사고방식은 비슷하다. 필요한 식별자를 먼저 모으고, 가능한 한 묶어서 읽고, 반복 왕복을 줄이는 쪽으로 구조를 조정해야 한다.

초심자에게는 문법보다 원칙이 중요하다. 루프 안에 데이터베이스 호출이 보이면, 그 구조가 정말 불가피한지 먼저 의심하는 습관이 필요하다. 많은 경우 문제의 실마리는 바로 거기에 있다.

느린 쿼리는 감으로 찾는 것이 아니라 로그와 실행 계획으로 읽는다

서비스가 느려질 때 가장 많이 하는 실수 중 하나는 원인을 추측으로 단정하는 것이다. 서버가 약한 것 같고, 네트워크가 느린 것 같고, 외부 API가 문제인 것 같다는 식의 감은 출발점이 될 수는 있어도 답이 되지는 않는다. 데이터베이스 문제는 특히 더 그렇다. 겉으로는 페이지 전체가 느려 보여도, 실제 원인은 특정 쿼리 하나일 수 있다.

그래서 로그와 측정 도구가 필요하다. Supabase나 PostgreSQL 계열에서는 느린 쿼리 로그와 `EXPLAIN ANALYZE`가 핵심 도구가 된다. 어떤 쿼리가 얼마나 자주 실행되고, 평균 시간이 얼마나 걸리며, 데이터베이스가 실제로 그 쿼리를 어떤 방식으로 처리하는지를 볼 수 있기 때문이다. Firestore 계열에서는 같은 형태의 SQL 로그는 없지만, 사용량 그래프와 인덱스 경고, 읽기와 쓰기 수의 변화를 통해 구조적 문제를 포착할 수 있다.

중요한 것은 화려한 도구 이름보다 읽는 관점이다. 어떤 쿼리가 느린가, 어떤 쿼리가 너무 자주 실행되는가, 데이터베이스가 전체를 훑고 있는가, 아니면 필요한 경로로 바로 접근하고 있는가를 보는 것이 핵심이다. 측정 없이 최적화에 들어가면, 종종 시간을 많이 쓰고도 문제를 비껴가게 된다.

인덱스는 데이터베이스에 미리 깔아두는 길이다

인덱스는 초심자에게 가장 많이 오해되는 개념 중 하나이다. 어떤 사람은 만능처럼 여기고, 어떤 사람은 너무 어렵게 생각한다. 실제로는 훨씬 단순하다. 인덱스는 데이터베이스가 자주 찾는 기준에 맞춰 미리 길을 내두는 장치이다. 책 뒤의 색인이나 사전의 가나다 정렬을 떠올리면 된다. 원하는 대상을 찾을 때 처음부터 끝까지 훑지 않도록, 찾기 쉬운 질서를 따로 만들어 두는 것이다.

문제는 인덱스가 자동으로 "모든 경우"에 최적이 되지는 않는다는 점이다. 어떤 컬럼으로 자주 검색하는지, 어떤 정렬을 반복하는지, 어떤 조합의 조건이 자주 붙는지에 따라 필요한 인덱스가 달라진다. 다시 말해 인덱스는 데이터의 양보다 조회 습관과 더 깊게 연결된다.

이 지점에서 초심자가 자주 빠지는 오해가 있다. 느리면 인덱스를 많이 만들면 된다고 생각하는 것이다. 그러나 인덱스는 읽기를 빠르게 하는 대신 쓰기에 비용을 더한다. 데이터를 추가하거나 수정할 때마다 인덱스도 함께 갱신해야 하기 때문이다. 책 뒤에 색인이 많을수록 찾기는 쉬워지지만, 본문이 바뀔 때 수정할 것도 많아지는 것과 같다.

따라서 원칙은 단순하다. 실제로 자주 검색하고, 실제로 느린 쿼리를 일으키는 컬럼에 맞춰 인덱스를 만든다. 미리 무작정 깔아두는 것은 좋은 전략이 아니다.

실행 계획을 읽는다는 것은 데이터베이스의 속마음을 읽는 일이다

`EXPLAIN ANALYZE` 같은 도구가 중요한 이유는, 데이터베이스가 쿼리를 실제로 어떻게 처리하는지 보여주기 때문이다. 겉으로는 같은 `SELECT` 문처럼 보여도, 어떤 경우에는 전체 테이블을 처음부터 끝까지 훑고 있고, 어떤 경우에는 인덱스를 타고 바로 접근한다. 초심자에게는 이 차이가 추상적으로 느껴질 수 있지만, 바로 이 차이가 응답 시간을 몇 배, 때로는 몇백 배까지 벌린다.

따라서 실행 계획은 데이터베이스 문제를 추측으로 다루지 않게 해준다. 서비스가 느린 이유가 "왠지 데이터가 많아서"가 아니라, 인덱스가 없는 상태에서 전체를 훑고 있기 때문이라는 식으로 구체화되는 것이다. 이 구체화가 이루어져야 비로소 정확한 수정을 할 수 있다.

AI에게 이렇게 물어보세요

> "이 쿼리가 왜 느린지 감으로가 아니라 실행 계획 기준으로 설명해줘. `EXPLAIN ANALYZE` 결과를 함께 붙일 테니, 어떤 부분이 병목이고 왜 그런지 초심자도 이해할 수 있게 풀어줘."
>
> "이 코드에 N+1 문제가 있는지 판단해줘. 루프 안의 DB 호출이 왜 문제인지 설명하고, 관계형 DB 또는 Firestore에 맞는 방식으로 묶어서 읽는 구조를 제안해줘."

Chapter 22: 캐시는 속도를 높이는 도구이기 전에, 같은 질문을 덜 하게 만드는 도구이다

캐시를 이해하는 가장 쉬운 방식

캐시는 종종 과장되거나, 반대로 지나치게 기술적인 용어로 설명된다. 그러나 원리는 단순하다. 자주 쓰는 데이터를 원본 저장소에서 매번 다시 가져오지 않고, 가까운 곳에 잠시 두었다가 재사용하는 것이다. 냉장고 안의 반찬을 식사 때마다 하나씩 꺼냈다 넣기보다, 먹을 만큼만 식탁 위에 올려두는 것에 가깝다.

이 비유에서 원본 저장소는 데이터베이스이고, 식탁은 캐시이다. 중요한 것은 캐시가 원본을 없애는 기술이 아니라는 점이다. 원본은 그대로 두고, 반복해서 필요한 데이터를 조금 더 가까운 곳에 두는 방식일 뿐이다. 그래서 캐시는 빠른 데이터처럼 보이지만, 실은 반복 조회를 줄이는 기술이라고 이해하는 편이 더 정확하다.

왜 같은 데이터를 반복해서 읽게 되는가

서비스에는 놀랄 만큼 많은 "반복 조회"가 존재한다. 메인 페이지의 인기 게시글, 자주 바뀌지 않는 공지사항, 공통 네비게이션에 필요한 카테고리 목록, 프로필의 기본 정보, 앱 전체 설정값 같은 것들이 그렇다. 이런 데이터는 많은 사용자에게 거의 동일하게 보인다. 그런데 캐시가 없으면 사용자 천 명이 페이지를 열 때 데이터베이스도 천 번 같은 질문을 받는다.

처음에는 별문제가 없어 보인다. 사용자 수가 적고 데이터 양도 작기 때문이다. 그러나 어느 순간부터 이 반복 조회는 속도와 비용에 동시에 영향을 미친다. 관계형 데이터베이스에서는 같은 쿼리가 지나치게 반복되어 부하가 생기고, Firestore에서는 읽기 수가 곧 비용으로 이어지며, 서버리스에서는 같은 작업을 요청마다 다시 수행하느라 함수 실행 시간과 호출 수가 늘어난다.

이 때문에 캐시는 "나중에 여유가 생기면 넣는 최적화"가 아니라, 반복 조회가 구조적으로 많은 서비스에서 생각보다 초기에 검토해야 하는 설계 요소가 된다.

캐시는 Redis부터 떠올릴 필요가 없다

캐시를 말하면 많은 사람이 곧바로 Redis를 떠올린다. 물론 Redis는 매우 유용한 도구이다. 하지만 초심자에게는 종종 너무 이른 선택지일 수 있다. 캐시에서 더 중요한 것은 "전용 캐시 서버를 쓰는가"보다 "반복 조회를 줄이고 있는가"이다.

실제로는 페이지 자체를 잠시 저장해 두는 방식만으로도 큰 효과를 보는 경우가 많다. Next.js의 ISR처럼 일정 시간 동안 페이지 결과를 재사용하는 구조는, 적은 코드로 매우 큰 비용 절감을 가져오기도 한다. 경우에 따라서는 특정 데이터만 함수 단위로 캐시해도 충분하다. 아주 작은 서비스라면 단순한 인메모리 캐시로도 필요한 만큼의 효과를 얻을 수 있다.

중요한 것은 단계이다. 페이지 단위 캐시로 해결할 수 있는 문제에 굳이 복잡한 공유 캐시 시스템을 먼저 도입할 필요는 없다. 초심자일수록 "가장 단순한 캐시부터 시작해도 되는가"를 먼저 물어야 한다.

Firestore에서 캐시는 특히 비용 문제와 직접 연결된다

Firestore는 읽기 수가 비용과 직접 연결되는 구조이기 때문에, 반복 조회의 부담이 특히 명확하게 드러난다. 게시글을 읽고, 각 게시글의 작성자를 다시 읽고, 댓글 수를 또 따로 읽고, 좋아요 수를 또 불러오는 식의 구조는 느려질 뿐 아니라 청구서에도 즉각 반영된다.

이 지점에서 중요한 개념이 데이터 중복 저장, 즉 필요한 정보를 일부 함께 저장하는 설계이다. 관계형 데이터베이스를 처음 배울 때는 중복을 줄이는 방향을 먼저 배우지만, 실제 서비스 운영에서는 읽기 성능과 비용을 줄이기 위해 일부 중복을 의도적으로 허용할 때가 많다. 예를 들어 게시글 문서 안에 작성자 이름이나 댓글 수, 좋아요 수 같은 값을 함께 넣어 두면, 읽기 횟수는 크게 줄어든다. 대신 그 값을 갱신하는 쓰기 로직은 조금 더 복잡해진다.

이것은 나쁜 설계가 아니라, 읽기를 단순하게 만들기 위한 현실적 선택일 수 있다. 많은 초심자가 여기에서 혼란을 겪지만, 서비스 운영에서는 "읽기를 위해 쓰기를 조금 더 복잡하게 만든다"는 전략이 충분히 합리적일 때가 많다.

모든 것을 캐시하면 안 되는 이유

캐시가 강력하다고 해서 아무 데이터나 캐시할 수 있는 것은 아니다. 인기 글 목록, 공지사항, 카테고리 목록, 자주 바뀌지 않는 프로필 정보는 캐시해도 큰 문제가 없을 수 있다. 반면 잔액, 결제 금액, 실시간 재고, 권한 정보처럼 틀리면 즉시 사고가 나는 데이터는 매우 신중해야 한다.

판단 기준은 간단하다. 조금 오래된 정보를 보여줘도 괜찮은가, 아니면 틀린 정보가 곧 신뢰 손상이나 금전 문제로 이어지는가를 물으면 된다. 캐시는 속도를 위한 도구이기 전에, "조금 늦어도 괜찮은 정보"를 다루는 도구이다.

캐시 시간은 짧게 시작하는 편이 낫다

초심자는 캐시 시간을 지나치게 짧게 잡아 효과를 거의 못 보거나, 반대로 지나치게 길게 잡아 오래된 정보를 너무 오래 보여주는 실수를 한다. 가장 안전한 방법은 짧게 시작해 점진적으로 늘리는 것이다. 예를 들어 메인 페이지 목록은 수십 초, 공지사항은 몇 분, 설정값은 그보다 조금 더 길게 두는 식의 접근이 보통 무난하다.

캐시는 정답을 맞히는 기술이 아니라, 틀려도 안전한 범위 안에서 시스템의 부담을 줄이는 기술이다. 이 감각을 가지면 복잡한 캐시 시스템 없이도 많은 문제를 충분히 완화할 수 있다.

AI에게 이렇게 물어보세요

> "이 페이지는 많은 사용자에게 거의 같은 데이터를 보여준다. ISR, 데이터 캐시, 단순 인메모리 캐시 중 무엇이 가장 적절한지 이유와 함께 설명해줘."
>
> "Firestore 읽기 수가 많이 나온다. 어떤 정보를 문서 안에 함께 저장하면 읽기 비용을 줄일 수 있는지, 그리고 그 대가로 어떤 쓰기 복잡도가 생기는지도 설명해줘."

Chapter 23: 쓰기가 몰리면 성능 문제가 아니라 값의 정확성이 무너진다

읽기 문제와 쓰기 문제는 다르게 다루어야 한다

읽기 문제는 대체로 느림으로 나타난다. 응답이 늦어지고, 페이지가 무거워지고, 사용자가 답답함을 느낀다. 물론 충분히 심각한 문제이지만, 읽기가 느리다고 해서 값 자체가 틀어지는 것은 아니다. 반면 쓰기 문제는 성능 문제와 함께 정합성 문제를 낳는다. 좋아요 수가 실제보다 적게 올라가거나, 재고가 음수가 되거나, 같은 포인트가 중복 적립되는 식의 현상이 생긴다.

초심자가 이 차이를 놓치면 "가끔 숫자가 이상하다"는 현상을 단순한 버그처럼 받아들이게 된다. 하지만 많은 경우 그것은 경쟁 조건, 즉 동시에 여러 요청이 같은 데이터를 건드리면서 발생하는 구조 문제이다.

카운터가 가장 먼저 흔들리는 이유

좋아요 수나 조회수 같은 카운터는 처음에는 매우 단순해 보인다. 현재 값을 읽고, 하나 더한 뒤, 다시 저장하면 될 것처럼 느껴진다. 그러나 동시에 여러 요청이 들어오면 이야기가 달라진다. 두 명의 사용자가 같은 값을 읽고, 각각 하나를 더해 저장하면 실제로는 두 번 증가해야 할 수치가 한 번만 반영될 수 있다. 둘 다 논리적으로는 맞는 일을 했지만, 순서가 겹치면서 결과가 틀어지는 것이다.

이것이 경쟁 조건이다. 혼자 실행될 때는 멀쩡한 코드가, 동시에 실행될 때만 값을 틀리게 만든다. 초심자에게는 특히 낯선 종류의 문제이다. 테스트 환경에서는 잘 돌아가고, 사용자가 적을 때도 잘 보이지 않기 때문이다. 그러나 실제 서비스에서는 인기 게시글, 이벤트 카운트, 실시간 반응 수치처럼 동시에 많은 사람이 건드리는 값이 생각보다 많다.

이 문제의 핵심은 애플리케이션이 값을 읽어서 계산하지 않게 만드는 데 있다. 데이터베이스 안에서 바로 증가시키는 원자적 연산을 쓰면, 중간에 다른 요청이 끼어들 여지를 줄일 수 있다. Firestore의 `FieldValue.increment`, SQL의 `SET count = count + 1` 같은 방식이 바로 그 예이다. 여기서 중요한 것은 문법이 아니라 철학이다. 읽고 계산해서 저장하는 구조를 피하고, 데이터베이스 내부의 안전한 증가 연산을 사용하는 것이다.

하나의 값에 너무 많은 쓰기가 몰리면 다른 종류의 병목이 생긴다

원자적 증가를 사용한다고 해서 모든 쓰기 문제가 끝나는 것은 아니다. 특정 문서나 행 하나에 초당 수십, 수백 번 이상의 쓰기가 집중되면, 그 한 지점 자체가 물리적 병목이 되기 시작한다. 실시간 좋아요 수, 조회수, 인기 투표, 이벤트 참여 카운트가 대표적이다.

이때는 구조를 바꾸어야 한다. 단 하나의 통장에 모든 입금을 몰아넣는 대신, 여러 개의 통장으로 나누어 적립한 뒤 나중에 합치는 것처럼 생각하면 된다. 흔히 분산 카운터라고 부르는 방식이 바로 이것이다. 값을 여러 샤드에 나누어 기록하고, 읽을 때 합산하는 구조이다. 구현은 조금 더 복잡해지지만, 하나의 지점에 쓰기가 몰려 생기는 압박을 분산할 수 있다.

초심자에게는 이 방식이 어렵게 들릴 수 있다. 그러나 원리는 단순하다. 한곳에 너무 많이 모이면 부딪히므로, 여러 곳에 나누어 쓴다는 것이다. 규모가 커질수록 이 단순한 원리가 점점 더 중요해진다.

오래 걸리는 쓰기는 아예 별도의 흐름으로 분리하는 편이 낫다

카운터와 달리, 어떤 쓰기 작업은 그 자체가 길고 복잡하다. 주문을 받고, 재고를 확인하고, 결제를 요청하고, 메일을 보내고, 알림을 보내는 일을 하나의 HTTP 요청 안에서 모두 처리한다고 가정해 보자. 이 구조는 처음에는 단순해 보인다. 그러나 외부 결제 API가 느리거나, 이메일 서비스가 지연되거나, 특정 단계가 재시도를 필요로 하면 전체 요청이 길어지고, 서버리스 환경이라면 시간 제한에 걸리기 쉽다.

이럴 때 필요한 것은 접수와 처리의 분리이다. 사용자는 요청을 접수시키고 빠르게 응답을 받아야 하고, 실제 무거운 처리나 재시도 가능한 처리는 뒤쪽의 별도 흐름에서 수행하는 편이 낫다. 이것이 큐가 필요한 이유이다.

큐는 기술적으로 복잡해 보이지만, 개념은 단순하다. 지금 당장 처리하기보다 줄을 세운 뒤 차례대로 안정적으로 처리하는 구조이다. 식당의 주문 접수와 주방의 조리를 분리하는 것과 같다. 주문을 받는 사람이 직접 요리까지 해버리면 카운터가 금방 막히지만, 주문은 접수만 하고 조리는 뒤에서 따로 처리하면 전체 흐름이 훨씬 안정된다.

쓰기 문제를 바라보는 순서

초심자에게 쓰기 문제는 무엇보다 "무엇이 틀어질 수 있는가"를 먼저 보는 것이 중요하다. 값이 정확해야 하는 기능인지, 동시에 많은 사람이 같은 값을 건드리는지, 실패했을 때 나중에 다시 처리해도 되는지, 요청 안에서 모두 끝낼 필요가 있는지를 판단하면 대응 방향이 정리된다. 카운터라면 원자적 연산을 먼저 떠올리고, 한 지점에 과도하게 몰린다면 분산 카운터를 검토하고, 오래 걸리는 작업이라면 큐를 검토하는 식이다.

이 순서는 기술의 난이도 순서이기 전에, 문제의 성격을 구분하는 순서이다. 읽기 문제와 쓰기 문제를 같은 방식으로 다루지 않는 것이 데이터베이스 운영의 가장 중요한 감각 중 하나이다.

AI에게 이렇게 물어보세요

> "좋아요 수가 가끔 실제보다 적게 올라간다. 이 현상이 경쟁 조건인지 판단해주고, 왜 그런 일이 생기는지 구조적으로 설명해줘."
>
> "단일 문서 또는 행에 쓰기가 너무 많이 몰릴 때 왜 분산 카운터가 필요한지, 구현 원리까지 포함해 초심자도 이해할 수 있게 설명해줘."
>
> "주문 처리 API가 재고 확인, 결제, 메일 발송을 한 번에 처리한다. 이 구조를 큐 기반으로 바꾸어야 하는 이유와, 접수와 처리의 흐름을 어떻게 분리해야 하는지 설명해줘."

Part 6 마무리

데이터베이스는 겉으로 보이지 않기 때문에 더 어렵게 느껴진다. 그러나 서비스가 느려지고 흔들리는 이유를 따라가다 보면, 결국 많은 문제가 읽기 습관과 쓰기 습관에서 비롯된다는 사실을 알게 된다. 데이터를 어떻게 읽고, 얼마나 자주 다시 읽고, 동시에 어떻게 쓰는지가 서비스의 속도와 비용, 정확성을 함께 결정한다.

이 파트에서 기억해야 할 핵심은 단순하다. 데이터가 적을 때는 많은 구조가 멀쩡해 보이지만, 그 상태가 좋은 구조를 의미하지는 않는다. 데이터베이스가 잘하는 일은 데이터베이스에게 맡겨야 하고, 같은 질문을 반복할수록 캐시와 구조 조정의 필요성이 커지며, 쓰기가 몰릴수록 값의 정확성을 지키는 방식으로 사고를 바꾸어야 한다.

데이터베이스를 어려운 기술로 보지 않고, 앱이 데이터와 대화하는 방식으로 보기 시작하면 구조가 훨씬 선명해진다. 그 순간부터 병목은 막연한 불운이 아니라 수정 가능한 설계 문제가 된다.

PART 7

배포와 인프라

VPS, EC2, GCE, Azure VM, 서버리스, 컨테이너, 엣지와 국내 클라우드 감각

앱을 만든 뒤 배포를 앞두고 가장 먼저 마주하는 난관은 코드가 아니라 구조이다. 기능을 만드는 동안에는 화면이 어떻게 보이는지, 버튼을 누르면 무엇이 일어나는지, 데이터가 잘 저장되는지를 중심으로 사고하게 된다. 그러나 배포 단계에 들어서면 질문의 축이 달라진다. 이 코드는 항상 켜져 있어야 하는가, 요청이 올 때만 잠깐 실행되면 충분한가, 오래 걸리는 작업을 버틸 수 있는가, 사용자가 몰릴 때 무엇이 가장 먼저 한계에 닿는가 같은 질문이 등장한다.

문제는 이 시점부터 설명이 갑자기 불친절해진다는 데 있다. 인터넷에는 EC2, Lambda, Cloud Run, Docker, Edge 같은 이름이 끝없이 등장하지만, 대부분의 설명은 이미 그 차이를 알고 있다는 전제를 깔고 있다. 처음 접하는 사람에게는 비슷해 보이는 이름이 서로 다른 방향을 가리키는 듯 보이고, 무엇을 기준으로 골라야 하는지도 분명하지 않다. 서버 구조를 고르는 일이 기술적 판단이 아니라 생소한 용어를 구별하는 시험처럼 느껴지는 이유가 여기에 있다.

하지만 실제로 서버 구조는 플랫폼 이름을 고르는 문제가 아니다. 내 앱이 어떤 방식으로 일하는지 이해하는 문제가 먼저이다. 계속 대기해야 하는 앱인지, 요청이 올 때만 잠깐 움직이면 되는 앱인지, 무거운 처리가 있는지, 아주 빠른 첫 반응이 중요한지를 먼저 파악하면 구조 선택은 훨씬 단순해진다.

먼저 큰 그림을 잡고, 각 방식이 실제로 어떤 성격을 가지는지 차례대로 살펴볼 필요가 있다. 그 위에서 초심자에게 왜 어떤 선택이 더 안전한지, 그리고 현실에서는 이 구조들을 어떻게 조합하게 되는지를 판단할 수 있다.

Chapter 24: 서버 구조를 바라보는 가장 쉬운 기준

서버는 장소가 아니라 방식이다

초심자에게 서버는 대개 코드를 올려두는 장소처럼 보인다. 물론 틀린 말은 아니다. 그러나 실제 차이는 코드가 어디에 올라가느냐보다 어떻게 실행되느냐에서 생긴다. 어떤 환경은 요청이 올 때만 코드를 불러와 잠깐 실행하고 끝낸다. 어떤 환경은 하루 종일 켜져 있으면서 누군가의 요청을 기다린다. 어떤 환경은 높은 자유를 주는 대신 관리 부담을 떠안기고, 어떤 환경은 자유를 조금 제한하는 대신 운영을 상당 부분 대신한다.

따라서 서버 구조를 처음 이해할 때는 서비스 이름보다 실행 방식의 차이를 먼저 보는 편이 낫다. 아주 거칠게 나누면, 직접 관리하는 컴퓨터 한 대를 빌리는 방식이 있고, 요청이 있을 때만 실행되는 방식이 있고, 앱을 하나의 실행 상자처럼 포장해 옮기는 방식이 있고, 사용자 가까운 곳에서 짧은 판단만 먼저 처리하는 방식이 있다. 여기에 배포와 운영을 상당 부분 대신해 주는 관리형 서비스까지 더하면, 오늘날 초심자가 마주치는 대부분의 선택지는 대체로 이 범주 안에 들어간다.

이 차이를 생활적인 비유로 바꾸면 이해가 빠르다. VPS나 VM은 작업실을 하나 통째로 얻는 것에 가깝다. 원하는 도구를 전부 들여놓을 수 있지만, 청소와 수리도 결국 직접 해야 한다. 서버리스는 필요할 때만 부르는 단기 인력과 비슷하다. 일이 없을 때는 비용 부담이 작지만, 오래 걸리거나 계속 대기해야 하는 일에는 잘 맞지 않는다. 컨테이너는 이삿짐 상자에 가깝고, 엣지는 건물 입구의 안내 데스크에 가깝다. PaaS는 관리인이 붙어 있는 사무실에 비유할 수 있다.

이 비유가 중요한 이유는, 초심자에게 서버 구조가 "어떤 것이 더 고급인가"의 문제가 아니라 "내 앱이 어떤 생활 패턴을 가지는가"의 문제라는 사실을 보여주기 때문이다.

가장 먼저 던져야 할 질문

서버 구조를 고를 때 처음부터 모든 요소를 동시에 따질 필요는 없다. 가장 먼저 던져야 할 질문은 하나이다. 내 앱의 주요 기능은 요청이 들어올 때만 잠깐 실행되면 충분한가, 아니면 계속 살아 있으면서 누군가의 연결이나 작업을 붙들고 있어야 하는가.

게시글 저장, 회원가입 처리, 결제 완료 알림 수신, 외부 AI API를 한 번 호출하고 결과를 반환하는 기능은 대개 짧은 요청-응답 흐름에 속한다. 이런 기능은 서버리스와 잘 맞는다. 반대로 실시간 채팅처럼 연결을 지속적으로 유지해야 하거나, 백그라운드에서 큐를 계속 감시하는 워커처럼 쉬지 않고 살아 있어야 하는 역할은 컨테이너나 VPS 쪽이 더 자연스럽다.

이 질문 하나만 명확해져도 선택의 절반은 정리된다. 플랫폼이 유명한지 아닌지보다, 내 기능이 짧은 호흡인지 긴 호흡인지가 훨씬 더 중요하기 때문이다.

초심자가 가장 많이 하는 오해

여기서 많은 사람이 "결국 무엇이 가장 좋은가"를 묻는다. 그러나 그 질문에는 정답이 없다. 더 정확한 질문은 "현재의 내 상황에서 가장 덜 무리하고, 가장 덜 위험한 선택은 무엇인가"이다. 초심자에게는 이 차이가 결정적이다.

예를 들어 VPS는 매우 강력하다. 거의 무엇이든 할 수 있고, 원하는 대로 설정할 수 있다. 반면 그만큼 직접 돌봐야 할 것도 많다. 서버리스는 제약이 존재하지만 시작하기는 훨씬 쉽고, 운영 부담도 낮다. 결국 중요한 것은 기술적으로 가장 강력한 구조가 아니라, 현재의 내가 감당할 수 있는 구조이다.

이 파트가 견지하는 기본 태도도 여기에 있다. 특별한 이유가 없다면 초심자는 가장 복잡한 것부터 시작할 필요가 없다. 쉬운 구조에서 출발하고, 실제로 한계를 만났을 때 필요한 부분만 따로 분리하는 편이 훨씬 현실적이다.

AI에게 이렇게 물어보세요

> "내 앱은 [앱 설명]이고, 요청이 들어오면 [주요 작업]을 수행한다. 계속 켜져 있어야 하는 기능이 있는지, 요청이 있을 때만 실행되면 되는지부터 구분해서 서버 구조를 추천해줘. 기술 이름보다 역할 중심으로 설명해줘."

Chapter 25: VPS, EC2, GCE, Azure VM은 어떻게 대응되는가

VPS를 어떻게 이해해야 하는가

VPS는 `Virtual Private Server`의 약자이다. 한국어로는 보통 가상 사설 서버, 또는 가상 전용 서버 정도로 옮긴다. 이름이 조금 어렵게 들리지만, 초심자에게 가장 쉬운 설명은 이것이다. 인터넷 어딘가에 리눅스 컴퓨터 한 대가 있고, 그 컴퓨터를 내가 원격으로 접속해 직접 관리하는 방식이라고 생각하면 거의 맞다. AWS EC2든, DigitalOcean Droplet이든, Vultr든 이름은 달라도 체감은 비슷하다. 터미널에 접속해 프로그램을 설치하고, 서버를 띄우고, 로그를 보고, 설정을 바꾼다.

여기서 EC2는 `Elastic Compute Cloud`의 약자이다. AWS가 제공하는 가상 서버 서비스 이름이라고 이해하면 된다. `Elastic`은 필요에 따라 자원을 늘리거나 줄일 수 있다는 뜻이고, `Compute`는 계산 자원, 곧 서버 성능을 뜻하며, `Cloud`는 그것이 클라우드 환경 위에서 제공된다는 뜻이다. 이름은 거창하지만, 실무 감각으로는 "AWS의 가상 서버"라고 이해하면 충분하다. 같은 자리에 놓고 볼 수 있는 구글 클라우드 서비스는 `Google Compute Engine`, 줄여서 `GCE`이다. 이름 그대로 구글 클라우드의 가상 서버 서비스라고 보면 된다. Azure에서는 같은 역할을 `Azure Virtual Machines`, 줄여서 `Azure VM`이 맡는다. 결국 AWS의 EC2, 구글의 GCE, Azure의 Virtual Machines는 서로 다른 회사가 제공하는 "직접 관리하는 가상 서버"라는 점에서 같은 부류의 선택지이다.

국내 클라우드도 같은 관점으로 보면 이해가 쉬워진다. NAVER Cloud의 `Server`, NHN Cloud의 `Instance`, KakaoCloud의 `Virtual Machine` 같은 상품은 모두 이 범주에 놓인다. 여기서 특히 헷갈리기 쉬운 이름이 NAVER Cloud와 NHN Cloud인데, 두 서비스는 현재는 별도 회사의 클라우드이다. 다만 역사적으로는 NHN이 과거 네이버와 한게임이 합쳐져 만들어진 회사였기 때문에 이름이 비슷하게 느껴질 수 있다. 지금 시점에서는 같은 회사의 다른 브랜드가 아니라, 서로 다른 사업자가 운영하는 별개의 클라우드 서비스로 이해하는 편이 정확하다.

이 구조가 매력적으로 보이는 이유는 분명하다. 자유도가 높기 때문이다. 원하는 프로그램을 거의 무엇이든 설치할 수 있고, 운영체제 수준의 설정도 만질 수 있고, 포트도 열 수 있고, 특수한 라이브러리

도 비교적 자유롭게 다룰 수 있다. 다른 플랫폼이 "그 기능은 지원하지 않는다"고 말할 때, VPS는 대개 "직접 해보라"고 답할 수 있다.

그러나 바로 그 자유가 책임으로 돌아온다. 직접 할 수 있다는 말은, 직접 책임져야 한다는 말이기도 하기 때문이다.

VPS의 진짜 비용

처음에는 높은 자유가 장점으로만 보인다. 원하는 것을 모두 할 수 있기 때문이다. 그러나 서비스를 실제로 운영하기 시작하면 다른 얼굴이 보인다. 디스크가 찼을 때 누가 정리할 것인지, 메모리가 점점 새는 것처럼 보일 때 누가 원인을 찾을 것인지, 보안 업데이트는 누가 챙길 것인지, 인증서는 누가 갱신할 것인지, 새벽에 서버가 멈추면 누가 일어나 확인할 것인지가 모두 질문으로 돌아온다.

조직 안에 인프라 담당자가 따로 있다면 이 일은 역할 분담으로 해결될 수 있다. 하지만 바이브 코더는 대부분 혼자이다. 낮에는 기능을 만들고, 밤에는 장애를 보고, 그 사이에 배포와 비용과 운영까지 떠안게 된다. VPS가 초심자에게 어려운 이유는 문법이 아니라 역할이 늘어난다는 데 있다. 어느 순간부터 개발자이자 운영자이자 보안 담당자가 되어 버리기 때문이다.

VPS가 필요한 경우

물론 VPS가 필요한 순간은 분명히 있다. GPU가 필요한 작업을 직접 돌려야 할 때, 장시간 살아 있어야 하는 실시간 서버를 운영해야 할 때, 특수한 시스템 프로그램을 설치해 사용해야 할 때, 브라우저를 통째로 띄우는 자동화 작업을 해야 할 때, 또는 백그라운드 프로세스를 장시간 돌려야 할 때는 VPS가 현실적인 선택이 되기도 한다.

중요한 것은 "정말 필요해서" 고르는가이다. 초심자가 VPS에 먼저 끌리는 가장 흔한 이유는 그것이 더 본격적이고 강력해 보이기 때문이다. 그러나 특별한 이유 없이 VPS를 먼저 선택하면 얻는 자유보다 감당해야 할 운영 부담이 더 커지기 쉽다.

초심자가 VPS에서 맞닥뜨리는 장면

많은 사람이 비슷한 경로를 밟는다. 처음에는 배포가 잘된다. AI의 도움을 받아 Nginx도 설치하고, 런타임도 깔고, 앱도 띄운다. 며칠 혹은 몇 주 동안은 꽤 잘 돌아간다. 그런데 시간이 지나면 로그 파일이

쌓이고, 메모리가 점점 늘고, 재부팅 후 앱이 자동으로 다시 켜지지 않거나, SSL 설정이 꼬이는 순간이 온다. 기능을 만드는 일보다 서버를 유지하는 일이 더 큰 비중을 차지하기 시작한다.

이때 분명해지는 사실이 있다. VPS는 "처음 올리기"보다 "계속 건강하게 유지하기"가 더 어렵다는 점이다. 한 번 배포하는 일보다 한 달 뒤에도, 두 달 뒤에도 멀쩡하게 돌아가도록 두는 일이 훨씬 까다롭다.

VPS를 고를 때 기억해야 할 기준

VPS를 선택해야 할 이유가 아주 분명하다면 괜찮다. 그러나 이유가 단지 "더 제대로 하는 것처럼 보여서"라면 다시 생각해 볼 필요가 있다. 초심자에게 좋은 기본값은 대개 가장 자유로운 구조가 아니라, 가장 덜 위험한 구조이다. 기능을 빠르게 출시해야 하고, 사용자가 아직 많지 않고, 서버 관리에 많은 시간을 쓰고 싶지 않다면 VPS는 출발점보다는 후순위의 선택지에 가깝다.

그럼에도 VPS를 쓰기로 했다면 처음부터 완벽하려 하기보다 기본 안전장치를 먼저 마련하는 편이 현실적이다. 자동 백업, 모니터링 알림, SSH 키 로그인, 비밀번호 로그인 차단, 앱 자동 재시작 설정, 로그 정리는 "있으면 좋은 것"이 아니라 초기에 반드시 챙겨야 할 기본이다. 이것은 멋을 위한 설정이 아니라, 혼자 운영할 때 스스로를 지키기 위한 최소한의 장치이다.

AI에게 이렇게 물어보세요

> "내 앱에서 [GPU / 실시간 연결 / 특수 프로그램 설치]가 필요해 보인다. 이 요구가 정말 VPS를 선택해야 하는 이유인지부터 판단해주고, 초심자가 감당해야 할 운영 부담까지 포함해 설명해 줘."
>
> "Ubuntu 서버에 [Node.js/Python] 앱을 배포하려고 한다. 복잡함을 최소화하되, 백업과 자동 재시작, SSH 키 로그인 같은 기본 안전장치가 포함된 절차를 단계별로 정리해줘."

Chapter 26: 서버리스는 왜 많은 웹앱의 기본값이 되는가

서버리스의 정확한 의미

서버리스라는 이름은 종종 오해를 부른다. 서버가 아예 없는 것처럼 들리기 때문이다. 그러나 실제로는 서버가 없어진 것이 아니다. 다만 그 서버를 내가 직접 관리하지 않아도 되는 쪽으로 무게중심이 옮겨간 것이다. 요청이 들어오면 코드를 실행하고, 일이 끝나면 쉬고, 확장과 기본 운영은 플랫폼이 상당 부분 대신한다.

이 차이는 초심자에게 매우 크게 작용한다. 내가 관리해야 할 대상이 "컴퓨터 한 대"가 아니라 "기능 하나"로 줄어들기 때문이다. 예를 들어 회원가입 API를 만든다고 할 때, 서버리스에서는 그 기능이 요청을 받으면 실행되고 끝나는 흐름에 집중하면 된다. 디스크 공간을 청소하고, 프로세스를 재시작하고, 운영체제 패치를 확인하는 문제는 전면에서 물러난다.

그래서 서버리스는 처음 서비스를 내놓는 사람에게 특히 잘 맞는다. 빠르게 만들 수 있고, 비교적 적은 설정으로 배포할 수 있으며, 운영 부담도 낮게 유지할 수 있기 때문이다.

왜 서버리스가 기본값이 되는가

서버리스가 널리 선택되는 이유는 기술적으로 멋져 보이기 때문이 아니다. 가장 적은 짐으로 시작할 수 있기 때문이다. 배포가 단순하고, 아무도 사용하지 않을 때는 비용 부담이 작고, 사용자가 갑자기 조금 몰려도 직접 서버를 늘릴 고민 없이 버틸 가능성이 높다. 무엇보다 "운영"이라는 별도의 세계와 일정한 거리를 둘 수 있다.

이 점은 특히 사이드 프로젝트에서 중요하다. 유저가 있는 날도 있고 거의 없는 날도 있는 서비스에, 하루 종일 켜져 있는 서버를 하나 들고 있는 것은 생각보다 비효율적일 수 있다. 반면 서버리스는 필요할 때 일하고 쉬기 때문에 초기의 불확실한 트래픽과 잘 맞는다. Next.js와 Vercel, Supabase, Firebase 같은 도구가 이 경험을 매우 쉽게 만들어 준 것도 큰 이유이다. 예전 같으면 서버를 세우고 라우팅과 프로세스 관리까지 따로 고민해야 했던 일을, 이제는 파일 하나와 배포 한 번으로 끝내는 경우도 적지 않다.

서버리스에 잘 맞는 기능

서버리스는 짧고 명확한 요청-응답 흐름에 강하다. 회원가입, 로그인, 폼 제출, 게시글 저장, 프로필 조회, 결제 웹훅 처리, 외부 AI API를 한 번 호출한 뒤 응답을 돌려주는 작업처럼 "요청 하나를 처리하고 끝나는" 기능에 특히 적합하다. 일반적인 웹앱의 핵심 기능 상당수가 여기에 속한다. 그래서 초심자가 만드는 많은 서비스는 서버리스만으로도 꽤 멀리 갈 수 있다.

이것은 임시방편이라는 뜻이 아니다. 서버리스는 초심자용 장난감이 아니라, 적합한 문제에 대해서는 매우 강력한 기본 구조이다. 다만 모든 문제를 같은 방식으로 해결할 수는 없다는 사실만 기억하면 된다.

서버리스의 한계

서버리스는 편리하지만 만능은 아니다. 오래 걸리는 작업에는 제약이 생기기 쉽고, 연결을 오랫동안 유지하는 구조와는 잘 맞지 않을 때가 많다. 요청이 지나치게 무겁거나, 실행 시간이 길거나, 특수한 환경 제약 때문에 별도 프로그램 설치가 필요하면 서버리스는 갑자기 답답해진다.

또 하나 기억해야 할 점은 첫 요청이 약간 느릴 수 있다는 사실이다. 한동안 호출되지 않다가 다시 실행될 때 준비 시간이 필요할 수 있기 때문이다. 이 현상은 이름을 외울 필요보다, 성격을 이해하는 편이 중요하다. 즉 "가끔 첫 호출이 평소보다 느릴 수 있다"는 정도의 감각만 있어도 충분하다.

그리고 편리함의 반대편에는 비용 문제가 있다. 서버리스는 트래픽이 늘어나면 잘 버틸 수 있지만, 그만큼 버그나 악성 요청이 생겼을 때도 열심히 일한다. 플랫폼이 친절하게 처리하는 만큼, 청구서도 따라 움직인다. 따라서 서버리스에서는 운영 부담이 줄어드는 대신, 비용 알림과 호출 제한 같은 안전장치를 더 의식적으로 챙겨야 한다.

한계를 만났을 때의 대응

많은 초심자는 서버리스에서 어떤 한계를 만나면 곧바로 "이제 전부 옮겨야 하는가"를 고민한다. 그러나 대개 그럴 필요는 없다. 보통은 서버리스가 특히 버거워하는 기능 하나만 따로 떼어 내면 된다. 예를 들어 메인 웹앱과 일반 API는 그대로 두고, PDF 생성이나 영상 처리 같은 무거운 작업만 컨테이너로 옮기는 식이다.

이 관점은 매우 중요하다. 구조 변경은 늘 전체를 갈아엎는 일처럼 보이지만, 실제로는 문제를 일으키는 부분만 따로 분리하는 쪽이 훨씬 흔하고 합리적이다.

AI에게 이렇게 물어보세요

> "내 앱은 대부분 짧은 요청-응답 구조인데, 일부 기능만 자꾸 시간 초과가 발생한다. 서버리스를 버려야 하는 상황인지, 아니면 문제 기능만 분리하면 되는지 초심자도 이해할 수 있도록 설명해 줘."
>
> "Vercel에 올린 앱에서 어떤 기능이 서버리스에 맞지 않는지 판단하고 싶다. 요청 길이, 연결 유지 필요 여부, 외부 라이브러리 제약을 기준으로 어디까지는 서버리스로 두고 어디부터는 분리해야 하는지 설명해줘."

Chapter 27: 컨테이너는 언제 필요한가

컨테이너를 어떻게 이해해야 하는가

컨테이너라는 단어는 초심자에게 기술 냄새가 강하게 느껴진다. 그러나 개념 자체는 의외로 생활적이다. 이삿짐을 생각하면 된다. 살림살이를 하나씩 손에 들고 옮기면 번거롭고 빠뜨리기 쉽기 때문에 상자에 담아 라벨을 붙인다. 컨테이너는 앱을 그런 식으로 다루는 방법이다. 코드만 담는 것이 아니라, 그 코드가 돌아가기 위한 런타임과 라이브러리, 실행 명령까지 함께 묶는다.

이렇게 하면 "내 컴퓨터에서는 되는데 서버에서는 왜 안 되는가"라는 문제가 줄어든다. 실행 환경을 하나의 상자처럼 포장해 두기 때문에, 어디서 열어도 비슷하게 돌아가게 만들기 쉬워지기 때문이다.

컨테이너의 위치

컨테이너는 모든 프로젝트의 출발점이라기보다, 서버리스와 VPS 사이에서 특별히 유용해지는 선택지이다. 서버리스는 편하지만 제약이 있고, VPS는 자유롭지만 관리 부담이 크다. 컨테이너는 그 둘 사이에서 "조금 더 자유롭되, 그렇다고 모든 것을 직접 관리하고 싶지는 않을 때" 힘을 발휘한다.

그래서 초심자에게 컨테이너를 가장 쉽게 설명하는 문장은 이것이다. 서버리스가 버거워하는 일을 따로 맡기는 데 좋은 구조라는 말이다.

컨테이너에 잘 맞는 기능

PDF 생성, 이미지와 영상 처리, 브라우저 자동화, 장시간 작업, 큐 워커처럼 계속 살아 있어야 하는 백그라운드 프로세스는 서버리스보다 컨테이너에 더 잘 맞는 경우가 많다. 이런 기능은 실행 시간이 길고, 필요한 라이브러리가 무겁고, 환경 제약도 더 많다. 서버리스에서 겨우 욱여 넣어 돌릴 수는 있어도, 언젠가 무리해지는 순간이 온다.

그때 컨테이너는 자연스러운 대피처가 된다. 메인 앱 전체를 옮길 필요 없이, 문제를 일으키는 기능만 따로 떼어 돌리기 좋기 때문이다. 일반적인 웹 페이지와 가벼운 API는 기존 구조에 두고, PDF 생성 API만 Cloud Run으로 분리하는 식이 대표적이다. 이렇게 하면 전체 구조는 크게 흔들지 않으면서도 서버리스의 약한 지점을 우회할 수 있다.

오늘날의 컨테이너 플랫폼

예전에는 컨테이너가 운영이 복잡한 기술로 받아들여졌다. 그러나 지금 초심자가 자주 만나는 Cloud Run, Fly.io, Railway 같은 서비스는 훨씬 다루기 쉽다. 컨테이너 이미지를 올리면 배포를 도와주고, 요청에 맞춰 늘리거나 줄여 주며, 비교적 간단한 설정으로 시작할 수 있게 해준다.

따라서 오늘날의 컨테이너는 "거대한 인프라 기술"이라기보다, 특정 기능을 조금 더 자유로운 환경으로 옮겨 두는 실용적 수단으로 이해하는 편이 적절하다.

Dockerfile의 역할

컨테이너를 이야기할 때 Dockerfile을 빼놓을 수 없다. 그러나 이것을 지나치게 어렵게 볼 필요는 없다. Dockerfile은 이 상자를 어떻게 만들지 적은 설명서이다. 어떤 바탕 이미지를 사용할지, 작업 폴더를 어디로 할지, 패키지 파일을 어떤 순서로 복사할지, 라이브러리를 어떻게 설치할지, 마지막에 어떤 명령으로 앱을 실행할지를 기록한다.

문법은 낯설 수 있다. 그러나 실제로 하고 있는 말은 단순하다. 어떤 환경 위에, 어떤 파일들을 넣고, 어떤 준비를 거친 뒤, 어떤 방식으로 실행해 달라는 지시이다. 초심자에게는 각 줄을 외우는 것보다, 이 파일이 앱 실행 준비 절차를 적은 문서라는 감각을 잡는 편이 더 중요하다.

컨테이너가 빛나는 순간

초심자가 컨테이너를 이해해야 하는 이유는 이것이 모든 앱의 기본값이어서가 아니다. 서버리스의 장점은 유지하고 싶지만, 특정 기능은 더 넓고 자유로운 환경이 필요해지는 순간이 오기 때문이다. 이때 컨테이너를 알고 있으면 구조를 훨씬 부드럽게 확장할 수 있다. 갑자기 전부 VPS로 옮길 필요도 없고, 애초에 서버리스에 무리하게 욱여 넣은 구조를 끝까지 붙들 필요도 없다.

요컨대 컨테이너는 출발점이라기보다 전환점에 가깝다. 서비스가 조금 자라면서 일부 기능만 더 넓은 공간을 필요로 할 때, 그 기능을 안전하게 옮겨 놓는 장소라고 이해하면 된다.

AI에게 이렇게 물어보세요

> "내 앱 전체는 그대로 두고 [PDF 생성 / 이미지 처리 / 백그라운드 워커]만 따로 빼고 싶다. 왜 이런 기능이 서버리스보다 컨테이너에 잘 맞는지부터 설명하고, 초심자 입장에서 가장 쉬운 배포 방법도 추천해줘."
>
> "Dockerfile을 처음 보는 사람도 이해할 수 있게, 내 [Node.js/Python] 앱용 Dockerfile을 작성하고 각 줄이 왜 필요한지도 함께 설명해줘."

Chapter 28: 엣지는 입구에서 끝내는 판단에 가깝다

엣지의 역할

엣지도 이름만 들으면 어렵게 느껴진다. 그러나 감각은 단순하다. 사용자가 앱에 요청을 보냈을 때, 그 요청이 멀리 있는 본 서버까지 가기 전에, 사용자 가까운 곳에서 아주 짧은 판단을 먼저 처리하는 구조라고 생각하면 된다. 로그인 여부 확인, 지역에 따른 경로 분기, 간단한 리다이렉트, 가벼운 A/B 테스트 같은 작업이 여기에 잘 맞는다.

건물에 비유하면 엣지는 입구의 안내 데스크이다. 안쪽 부서에서 본격적인 일을 처리하기 전에, 입구에서 신분을 확인하고, 어느 층으로 갈지 정하고, 잘못 들어온 사람은 돌려보낸다. 안내 데스크가 복잡한 회계 업무까지 하지는 않는다. 엣지도 마찬가지다. 빠른 첫 판단에는 강하지만, 무거운 계산이나 복잡한 비즈니스 로직 전체를 떠맡는 곳은 아니다.

왜 엣지가 필요한가

로그인 여부를 확인해 로그인하지 않은 사용자를 곧바로 로그인 페이지로 보내거나, 사용자의 지역과 언어에 따라 경로를 나누는 일은 짧고 빠르게 끝나야 한다. 굳이 모든 요청을 본 서버까지 보내지 않아도 된다. 이런 일을 앞단에서 처리하면 체감 속도가 좋아지고, 본 서버까지 도달하는 불필요한 요청도 줄어든다.

다만 초심자에게 중요한 것은 엣지를 과장해서 보지 않는 것이다. 엣지는 "더 빠른 서버"가 아니다. 정확히는 "입구에서 끝낼 수 있는 가벼운 판단을 미리 처리하는 방법"이다.

엣지에 적합하지 않은 일

엣지가 빠르다는 이유로 모든 일을 여기서 처리하면 좋을 것처럼 보일 수 있다. 그러나 엣지는 원래 그런 용도로 만들어진 환경이 아니다. 파일 시스템 접근, 무거운 계산, 길게 이어지는 작업, 복잡한 외부 서비스 호출은 엣지와 잘 맞지 않는다. 엣지의 장점은 가벼움에서 나오기 때문에, 그 성격에 맞는 일을 맡길 때 비로소 빛난다.

따라서 가장 좋은 원칙은 단순하다. 엣지에서는 빠르게 확인하고, 본격적인 처리는 서버리스나 컨테이너에서 수행한다. 이 한 줄만 기억해도 엣지를 무리하게 쓰는 실수를 크게 줄일 수 있다.

Next.js에서 엣지를 만나는 자리

Next.js를 사용하는 사람이라면 `middleware.ts`가 엣지와 가장 가까운 접점이 된다. 여기에서 로그인하지 않은 사용자를 리다이렉트하거나, 특정 경로를 보호하거나, 국가별 언어 경로를 나누는 식의 작업을 하게 된다. 즉, 요청이 앱 안으로 깊이 들어오기 전에 앞단에서 검사와 방향 결정을 수행하는 자리이다.

이 감각만 잡혀도 `middleware.ts`를 훨씬 덜 어렵게 볼 수 있다. 본격적인 비즈니스 로직을 쓰는 파일이 아니라, 입장 전에 간단한 판단을 처리하는 파일로 이해하면 된다.

AI에게 이렇게 물어보세요

> "엣지를 처음 접하는 사람 기준으로, Next.js `middleware.ts`가 정확히 어떤 역할을 하는지 설명해줘. 로그인 검사나 지역별 리다이렉트를 왜 여기서 처리하는지도 함께 설명해줘."
>
> "내 앱에서 로그인 여부 확인만 앞단에서 빠르게 하고 싶다. 어디까지를 엣지에서 처리하고, 어디부터는 서버리스에서 처리해야 하는지 나누어 설계해줘."

Chapter 29: 실제 서비스에서는 구조를 어떻게 조합하는가

하나만 고르는 문제는 아니다

지금까지는 각 구조를 따로 살펴보았다. 그러나 현실의 서비스는 대개 둘 이상을 함께 쓴다. 메인 웹앱은 서버리스에 두고, 무거운 PDF 생성만 컨테이너로 빼고, 로그인 확인 같은 간단한 작업은 엣지에서 먼저 처리하는 식이다. 처음에는 이런 조합이 더 복잡해 보일 수 있다. 하지만 실제로는 오히려 각 구조를 억지로 하나로 통일하는 것보다 더 자연스럽고 효율적인 경우가 많다.

중요한 것은 여기서도 "무엇이 더 고급한가"가 아니라 "무엇을 어디에 맡겨야 가장 무리 없이 돌아가는가"이다. 전체를 VPS로 옮기면 모든 문제가 해결될 것처럼 보일 수 있지만, 동시에 운영 부담도 크게 늘어난다. 반대로 모든 기능을 끝까지 서버리스에 욱여 넣으면 특정 작업에서 한계를 반복해 맞닥뜨리게 된다. 현실적인 답은 대개 그 중간 어딘가에 있다.

가장 안전한 출발점

초심자에게 가장 자주 권하게 되는 전략은 단순하다. 먼저 서버리스로 시작하고, 실제로 서버리스가 힘들어하는 기능이 나타났을 때 그 부분만 따로 떼어 내는 것이다. 이 전략이 좋은 이유는 처음부터 관리할 것이 지나치게 많아지지 않기 때문이다. 기능을 빠르게 출시할 수 있고, 운영 부담도 비교적 낮은 상태로 유지할 수 있다.

예를 들어 일반적인 블로그, 대시보드, 커뮤니티, 소규모 SaaS는 Vercel과 Supabase 같은 조합만으로도 꽤 오랫동안 버틸 수 있다. 여기에 로그인 확인을 조금 더 빠르게 하고 싶다면 엣지 미들웨어를 추가하면 된다. PDF 생성이나 이미지 변환처럼 유난히 무거운 기능이 생기면 그 부분만 Cloud Run 같은 컨테이너 환경으로 분리하면 된다. 실시간 연결이 꼭 필요해질 때만 별도의 장기 연결 서버를 고민하면 된다.

이런 점진적 분리가 중요한 이유는, 문제를 경험하기도 전에 모든 미래를 한꺼번에 설계하려 하면 현재의 개발 속도와 이해도가 함께 무너지기 쉽기 때문이다. 반대로 실제 문제를 만난 뒤 필요한 만큼만 구조를 늘리면, 왜 그 구조가 필요한지도 자연스럽게 이해하게 된다.

구조를 고를 때 스스로에게 던질 질문

복잡한 용어 대신 이런 질문이 더 유용하다. 내 앱의 대부분은 요청 하나를 짧게 처리하고 끝나는가. 어떤 기능은 오래 걸리는가. 사용자의 연결을 오랫동안 붙들고 있어야 하는가. 특수한 프로그램 설치가 꼭 필요한가. GPU 같은 특별한 자원이 필요한가. 로그인 확인이나 지역 분기처럼 아주 짧은 판단을 앞단에서 끝내고 싶은가.

이 질문에 대한 답이 구조를 거의 정해 준다. 대부분의 경우 첫 번째 질문에 "그렇다"라고 답하게 된다. 그때 기본값은 서버리스가 된다. 그리고 나머지 질문에 해당하는 소수의 기능만 나중에 따로 분리하면 된다. 이 흐름이 자연스러운 이유는 실제 서비스도 대개 처음부터 거대한 인프라로 출발하지 않기 때문이다.

비용까지 포함한 판단

서버 구조를 논할 때 사람들은 자주 월 서버비만 비교한다. 물론 중요하다. 그러나 초심자에게는 운영 시간까지 포함해 보아야 한다. VPS가 몇 달러 더 저렴해 보여도, 직접 장애를 보고 보안 업데이트를 하고 로그를 정리해야 한다면 그 비용은 숫자에 잘 드러나지 않을 뿐 실제로는 매우 크다. 반대로 서버리스나 관리형 서비스는 약간 더 비싸 보일 수 있어도, 그만큼 시간을 돌려준다.

혼자 서비스를 만드는 사람에게 가장 비싼 자원은 종종 서버비가 아니라 집중력이다. 기능을 만들 시간과 에너지를 어디에 쓸지 생각하면, 약간 더 비싸더라도 덜 피곤한 구조가 더 좋은 선택일 때가 많다.

이 파트의 결론

초심자에게 서버 구조는 겁먹을 만큼 복잡한 주제가 아니다. 플랫폼 이름을 모두 외울 필요도 없다. 먼저 내 앱이 짧은 요청 중심인지, 오래 걸리는 일이나 장기 연결이 있는지를 보면 된다. 대부분의 웹앱은 서버리스에서 무리 없이 출발할 수 있고, 서버리스가 버거워하는 기능만 컨테이너나 다른 구조로 따로 분리하면 된다. 엣지는 앞단에서 빠른 판단을 맡길 때 빛나고, VPS는 높은 자유도가 정말 필요할 때 선택하는 구조이다.

좋은 구조는 처음부터 완벽한 구조가 아니다. 지금의 나와 지금의 앱에 가장 잘 맞는 구조가 좋은 구조이다. 그리고 대부분의 바이브 코더에게 그 첫 문장은 여전히 같다. 서버리스로 시작하고, 실제 문제를

만났을 때 필요한 부분만 분리하면 된다. 이 감각을 가지고 다음 파트로 넘어가면, 인프라라는 주제가 조금 덜 막연하게 보이기 시작할 것이다.

AI에게 이렇게 물어보세요

> "내 앱의 기능 목록을 보고, 어떤 것은 서버리스에 남기고 어떤 것은 컨테이너나 다른 구조로 빼야 하는지 역할 기준으로 설명해줘. 기술 이름보다 왜 그렇게 나누는지가 먼저 이해되게 써줘."
>
> "지금은 기능 출시 속도가 중요하고 운영 부담은 최소화하고 싶다. 내 상황에서 가장 안전한 시작 구조를 추천하고, 나중에 어떤 신호가 보이면 구조를 분리해야 하는지도 함께 설명해줘."

PART 8

비용과 운영

청구서, 대시보드, 장애 징후, 운영 감각을 같은 문제로 보는 법

서비스를 실제로 운영하기 시작하면 기술 문제와 돈 문제는 분리된 두 개의 일이 아니라 같은 구조를 다른 각도에서 보는 일이라는 사실이 드러난다. 사용자는 느리다고 말하고, 대시보드는 응답 시간이 늘었다고 말하고, 청구서는 저장소와 트래픽과 호출 횟수가 늘었다고 말한다. 겉으로는 서로 다른 증상처럼 보이지만, 실제로는 같은 원인이 다른 숫자로 나타난 것인 경우가 많다. 운영을 배운다는 것은 이 숫자들을 각자 따로 읽는 일이 아니라, 서로 어떤 관계에 있는지 읽는 일이다.

초기에는 서비스가 잘 버티는 것처럼 보이기 쉽다. 사용자가 적을 때는 비효율이 눈에 띄지 않고, 외부 API가 조금 느려도 전체 흐름이 살아 있으며, 데이터가 적으니 목록 조회도 무겁지 않다. 그러나 서비스가 자라면 같은 설계가 더 이상 조용히 지나가지 않는다. 그때부터 필요한 것은 새로운 기술의 이름이 아니라, 문제가 언제부터 시작되었는지, 어떤 경로에서 가장 먼저 드러나는지, 무엇을 먼저 줄여야 하는지를 읽는 능력이다. 이 파트는 바로 그 능력을 다룬다.

Chapter 30: "어제까지는 잘 되던 앱"이 무너지는 방식

많은 서비스는 갑자기 무너지는 것처럼 보이지만, 실제로는 오래전부터 약해지고 있었던 경우가 많다. 다만 초반에는 사용량이 작아서 문제가 보이지 않았을 뿐이다. 목록을 가져온 뒤 각 항목마다 다시 상세 조회를 하는 구조도, 이미지를 원본 크기 그대로 내려 보내는 방식도, 요청마다 외부 API를 다시 호출하는 방식도, 소수 사용자에게는 그럭저럭 돌아간다. 하지만 사용자가 조금만 늘어나도 이 습관들은 응답 시간과 비용과 실패율을 동시에 밀어 올린다.

운영자가 먼저 알아야 하는 것은 “무슨 일이 터졌는가”보다 “터지기 전에 무엇이 반복되었는가”이다. 서비스가 무너질 때는 보통 하나의 큰 원인보다 여러 개의 작은 비효율이 겹쳐 있다. 페이지를 여러 번 다시 읽고, 같은 데이터를 여러 번 가공하고, 같은 결과를 저장하지 않은 채 매번 다시 계산하고, 같은 파일을 여러 번 전송하는 구조가 쌓이면 어느 순간 임계점을 넘는다. 그 임계점은 기능의 문제처럼 보이지만 사실은 구조의 문제이다.

초기 징후는 대개 소란스럽지 않다. 특정 페이지가 조금씩 느려지고, 같은 요청이 간헐적으로만 실패하고, 하루 중 특정 시간대에만 지연이 커진다. 이때 중요한 것은 지나치게 늦기 전에 구조를 읽는 일이다. 서비스가 흔들리기 시작할 때 가장 먼저 보는 숫자는 단순한 매출이 아니라 응답 시간, 오류율, 반복 호출 횟수, 외부 의존성의 실패율, 저장소 읽기 횟수 같은 운영 지표이다. 이 숫자들은 대체로 문제를 숨기지 않는다.

Chapter 31: 느려지는 것과 멈추는 것은 다른 사건이다

사용자는 둘 다 “안 된다”라고 표현하지만, 운영의 입장에서는 완전히 다른 사건이다. 느려짐은 대체로 구조 안에 누적된 낭비가 드러나는 과정이다. 요청 하나에 너무 많은 작업이 묶여 있거나, 같은 데이터를 여러 번 읽고 있거나, 브라우저와 서버와 저장소가 서로 불필요하게 여러 번 왕복하고 있을 가능성이 높다. 반대로 멈춤은 한계치 초과, 배포 오류, 자원 고갈, 외부 서비스 장애처럼 경계가 넘어진 사건인 경우가 많다.

이 차이를 구분하지 못하면 대응도 엉킨다. 느려짐에 대해 재시도만 반복하면 더 많은 요청이 쌓여 상황이 악화되고, 멈춤에 대해 병목 최적화만 고민하면 당장 복구해야 할 문제를 놓친다. 그래서 운영에서는 먼저 증상을 분리해야 한다. 화면은 뜨지만 느린가, 특정 기능만 느린가, 특정 시점부터 급격히 죽는가, 일부 사용자에게만 보이는가, 배포 직후에만 문제가 생기는가를 나누어 보아야 한다. 이 질문이 곧 진단의 출발점이다.

초기 경고 신호는 아주 작다. 평균 응답 시간이 조금씩 오르거나, 95퍼센타일 응답 시간이 평균보다 훨씬 빨리 나빠지거나, 특정 API만 재시도 횟수가 늘거나, 한 시간에 한두 번 나던 오류가 하루에 수십 번으로 바뀌는 식이다. 이런 변화를 무시하면 어느 날부터는 “느림”이 아니라 “정지”로 보이기 시작한다. 따라서 운영은 사건이 발생한 뒤의 대응만이 아니라, 사건이 사건으로 굳어지기 전에 숫자를 읽는 습관까지 포함한다.

Chapter 32: 청구서는 서비스의 또 다른 로그이다

청구서는 비용을 지불하라는 문서이기도 하지만, 동시에 서비스가 어떤 구조로 움직였는지를 기록한 요약본이기도 하다. 어떤 항목이 크게 늘었는지 보면 어떤 습관이 반복되었는지 추정할 수 있다. 저장 비용이 늘었다면 파일이나 로그나 원본 데이터를 지나치게 쌓고 있을 수 있고, 네트워크 전송량이 급증했다면 이미지, 동영상, 대용량 JSON 응답을 많이 보내고 있을 수 있으며, 함수 호출 수가 늘었다면 작은 작업들을 지나치게 잘게 쪼개서 계속 실행하고 있을 수 있다.

비용을 읽을 때 중요한 것은 가격표를 외우는 일이 아니다. 어떤 자원이 반복될수록 비싸지는지를 아는 일이다. 읽기 호출이 많을수록 비싸지는 서비스가 있고, 저장된 데이터가 많을수록 비싸지는 서비스가 있으며, 전송량이 늘수록 비싸지는 서비스가 있다. 여기에 AI 기능이 붙으면 토큰 수, 모델 호출 횟수, 스트리밍 시간, 문맥 길이까지 비용에 들어오기 시작한다. 따라서 청구서는 단순한 월말 정산표가 아니라, 서비스가 어디에서 과하게 숨을 쉬고 있는지를 보여 주는 지표이다.

이 관점이 생기면 비용을 줄이는 방식도 달라진다. 단순히 더 싼 상품으로 옮기는 것이 아니라, 같은 결과를 더 적은 왕복으로 만들고, 같은 질문에 같은 답을 다시 계산하지 않고, 지금 당장 필요하지 않은 처리를 뒤로 미루는 방향으로 구조를 바꾸게 된다. 비용의 핵심은 절약의 태도보다 낭비를 식별하는 능력이다.

Chapter 33: 비용을 낮춘다는 것은 요청을 덜 만들고, 같은 일을 덜 반복하게 만드는 일이다

비용 최적화는 종종 “더 싼 곳으로 옮기기”와 같은 구매 문제로 오해된다. 물론 플랫폼 선택은 중요하다. 그러나 실제로는 서비스 구조 자체가 비용의 대부분을 결정하는 경우가 많다. 같은 데이터를 매번 다시 읽지 않게 하고, 같은 계산을 매 요청마다 반복하지 않게 하고, 지금 당장 보여 주지 않아도 되는 내용을 미리 만들어 두고, 무거운 작업을 사용자 요청과 분리하면 비용과 성능이 동시에 좋아진다.

예를 들어 캐시는 단순히 빠르게 만드는 장치가 아니다. 반복되는 요청을 덜 비싸게 만드는 장치이다. 정적 생성은 화면을 미리 준비하는 일이지만, 동시에 서버가 매번 같은 일을 하지 않아도 되게 만드는 일이기도 하다. 큐와 백그라운드 작업은 응답 시간을 줄이는 장치이면서, 긴 작업 때문에 전체 요청이 묶여 버리는 상황을 막는 장치이기도 하다. 그래서 비용 최적화는 기능을 덜 제공하는 일이 아니라, 같은 기능을 더 적은 중복으로 제공하는 일이라고 보는 편이 맞다.

AI가 들어간 서비스에서는 이 능력이 더 중요해진다. 모델을 한 번 더 부르는 비용은 작아 보일 수 있지만, 사용자 수가 늘고 대화 길이가 길어지면 금방 커진다. 같은 문맥을 반복해서 보내지 않도록 설계하고, 긴 작업은 스트리밍과 분리하며, 자주 쓰는 결과를 재사용하는 구조가 필요하다. 결국 비용은 기술 선택의 문제가 아니라 설계 습관의 문제이다.

Part 8 마무리: 로그와 알림은 운영의 감각을 몸에 붙게 만든다

운영을 잘하는 사람은 모든 것을 외우는 사람이 아니라, 필요한 순간에 어디를 봐야 하는지를 아는 사람이다. 로그는 그때 가장 먼저 확인해야 하는 기록이고, 알림은 이상 징후를 가장 빨리 알려 주는 감시 장치이다. 하지만 로그와 알림이 있다고 해서 저절로 운영이 되는 것은 아니다. 무엇이 정상이고 무엇이 이상인지, 어느 정도의 지연은 허용할 수 있는지, 어떤 오류는 즉시 대응해야 하는지를 먼저 정해 두어야 한다.

좋은 로그는 과도하지 않으면서도 충분히 맥락이 있다. 요청이 언제 시작되었는지, 어디에서 실패했는지, 어떤 사용자 상태에서 발생했는지, 외부 서비스와의 상호작용은 어땠는지를 남겨야 한다. 알림은 너무 적어도 문제지만 너무 많아도 문제이다. 알림이 남발되면 결국 아무도 보지 않게 된다. 따라서 알림은 “지금 바로 봐야 하는 것”에만 걸어 두어야 한다.

초기 서비스에서는 로그가 개발 편의용으로만 보이기 쉽다. 그러나 규모가 조금만 커져도 로그는 디버깅 도구를 넘어 운영 기록이 된다. 무엇이 바뀌었고, 어디서 느려졌고, 어떤 조건에서 실패했는지를 다시 재구성할 수 있어야 하기 때문이다. 이 복기 가능성이 서비스의 생존력을 좌우한다.

PART 9

성장 단계별 판단

유저 수가 달라질 때 무엇을 직접 붙들고 무엇을 넘겨야 하는지

성장은 좋은 일이다. 그러나 성장에는 언제나 구조 변경이 따라온다. 사용자가 늘면 같은 코드가 같은 방식으로 버티지 못하고, 같은 운영 습관도 같은 효율을 내지 못한다. 그래서 성장 단계별 판단은 단순히 “언제 무엇을 업그레이드할까”를 정하는 일이 아니다. 지금의 규모에서 무엇이 충분하고, 무엇이 이미 과한지, 무엇은 지금 미루어도 되는지, 무엇은 지금 손대지 않으면 나중에 더 비싸지는지를 구분하는 일이다.

이 파트의 목적은 커지는 서비스를 무작정 복잡하게 만들지 않되, 너무 오래 단순함에만 기대지 않게 하는 데 있다. 적은 사용자에게는 과한 아키텍처가 방해가 되고, 많은 사용자에게는 단순함이 더 이상 미덕이 아니라 리스크가 된다. 따라서 좋은 판단은 정답 하나를 찾는 일이 아니라, 현재 단계에 맞는 분기점을 읽는 일에 가깝다.

Chapter 34: 유저 0~100명 – 먼저 존재하는 서비스를 만들어야 한다

이 구간에서 가장 중요한 것은 완성도가 아니라 존재감이다. 사용자가 거의 없고, 아이디어가 맞는지 틀리는지부터 확인해야 하는 시기에는 과한 최적화가 오히려 제품을 늦춘다. 여기서 필요한 것은 모든 경로를 완벽하게 만들겠다는 다짐이 아니라, 핵심 흐름이 실제로 동작하는지 빠르게 확인할 수 있는 최소한의 구조이다.

하지만 “빨리 만들기”가 “아무렇게나 만들기”를 뜻하는 것은 아니다. 비밀 키를 코드에 바로 넣지 않는 습관, 기본 로그를 남기는 습관, 배포를 한 번 되돌릴 수 있게 해 두는 습관, 비용 알림을 일찍 걸어 두는 습관은 이 시기에도 필요하다. 이 습관들은 나중에 대규모 운영을 돕기 전에, 초반의 치명적인 실수를 막아 준다.

이 단계에서는 무엇보다도 바꿀 수 있는 구조가 중요하다. 팀이 작고 요구사항도 자주 바뀌기 때문에, 지금 당장은 단순하더라도 나중에 경계를 세우기 쉬운 형태로 두는 편이 낫다. 그래야 실제 사용자의 반응을 보고 빠르게 고쳐 나갈 수 있다. 0~100명 구간의 목표는 완벽한 아키텍처가 아니라, 살아 있는 서비스의 초기 형태를 만드는 것이다.

Chapter 35: 유저 100~10,000명 – 첫 번째 병목은 이 시기에 드러난다

사용자가 본격적으로 들어오기 시작하면 서비스는 자기 몸의 약점을 숨길 수 없게 된다. 이전까지는 아무 문제 없어 보이던 구조가 응답 시간, 오류율, 반복 호출 수, 저장소 읽기 수, 외부 API 비용으로 드러난다. 이 구간의 핵심은 전체를 다시 설계하는 것이 아니라, 먼저 가장 큰 병목이 어디에 있는지를 정확히 보는 것이다.

많은 팀이 이 시기에 실수하는 지점은, 문제를 감으로만 판단하고 한 번에 너무 많은 것을 바꾸려는 것이다. 그러나 병목은 대개 한 군데에만 있지 않다. 데이터베이스가 느린 것처럼 보이지만 실제로는 불필요한 API 왕복이 더 큰 원인일 수 있고, 프론트엔드가 무거운 것처럼 보이지만 실제로는 서버가 과도한 데이터를 내려 보내는 것이 더 큰 원인일 수 있다. 따라서 증상을 기술 이름으로 바로 번역하기 전에 흐름을 분해해야 한다.

이 단계에서는 “무엇이 느린가”보다 “어디서부터 느려졌는가”를 봐야 한다. 동일한 기능이 일부 사용자에게만 느린지, 특정 시간대에만 느린지, 특정 데이터 조건에서만 느린지, 배포 이후에만 느린지에 따라 대응은 완전히 달라진다. 이때부터는 기능 개발 속도와 병목 제거 속도를 함께 관리해야 한다. 둘 중 하나만 빠르면 서비스는 오래 버티지 못한다.

Chapter 36: 유저 10,000~100,000명 – 구조를 다시 묻기 시작하는 구간

이 규모에 들어서면 서비스는 더 이상 단순한 기능 집합이 아니다. 운영, 비용, 신뢰성, 인수인계, 장애 대응이 모두 제품의 일부가 된다. 이 시기에는 지금까지 잘 돌아가던 구조를 한 번 더 질문해야 한다. 이 구조가 더 많은 사용자와 더 많은 요청과 더 많은 데이터에도 버틸 수 있는지, 아니면 일부 책임을 분리해야 하는지 다시 따져야 한다.

특히 이 단계에서는 무거운 작업이 본 요청 경로를 방해하는지 살펴야 한다. 이미지 처리, 파일 변환, 대규모 검색, 긴 AI 추론, 외부 API에 대한 느린 호출 같은 작업이 메인 흐름에 그대로 묶여 있으면 서비스는 점점 둔해진다. 또한 운영 지식이 사람 머릿속에만 남아 있으면 팀이 커질수록 리스크가 커진다. 문서화, 경보 기준, 재현 절차, 롤백 방법이 모두 실제 운영 자산이 된다.

이 시기에는 무엇을 직접 유지할지, 무엇을 플랫폼에 맡길지, 무엇을 표준화할지 판단해야 한다. 초반에는 손으로 관리하던 것들이 이 규모에서는 손으로 관리되면 안 된다. 그러나 모든 것을 바로 추상화하는 것도 해답은 아니다. 좋은 판단은 지금의 병목과 다음 단계의 병목을 함께 보는 데서 나온다. 이 구간의 목표는 더 큰 규모를 버티는 것이 아니라, 더 큰 규모를 감당할 수 있는 방향으로 미리 구조를 정렬하는 것이다.

성장 단계가 바뀌면 팀이 해야 할 일도 바뀐다. 초반에는 속도가 중요해서 많은 것을 직접 만들 수 있지만, 규모가 커질수록 직접 만드는 것 자체가 부담이 된다. 이때 중요한 기준은 “우리에게 경쟁력이 되는가”와 “운영 복잡도를 얼마나 늘리는가”이다. 인증, 결제, 배포, 관측, 저장소, AI 모델 호출 같은 영역은 직접 통제하고 싶은 마음이 들 수 있지만, 실제로는 유지 비용까지 함께 봐야 한다.

직접 만드는 것이 좋은 경우도 분명 있다. 제품의 핵심 차별점이 되는 로직, 특별한 도메인 규칙, 사용자 경험을 크게 좌우하는 흐름은 직접 설계해야 한다. 반대로 운영의 기본기이지만 차별점이 크지 않은 부분은 잘 관리되는 도구나 플랫폼에 맡기는 편이 낫다. 문제는 “우리만의 것”과 “이미 검증된 것”을 구분하지 못할 때 생긴다. 성장한 서비스는 모든 것을 소유하려고 할수록 느려진다.

이 판단은 기술 선호가 아니라 지속 가능성의 문제이다. 팀 규모, 장애 대응 능력, 인수인계 가능성, 비용 예측 가능성까지 함께 봐야 한다. 작은 팀에게는 단순함이 생존이며, 큰 팀에게는 표준화가 생존이다. 어느 단계에서 무엇이 생존 전략인지 읽는 일이 이 파트의 핵심이다.

처음에는 한 사람이 서비스 전체를 이해하고 움직일 수 있다. 그러나 일정 규모를 넘어서면 이해와 실행을 한 사람이 모두 맡는 구조는 병목이 된다. 이때 중요한 것은 사람 수를 늘리는 것 자체가 아니라, 혼자 할 수 있는 범위와 팀으로 나눠야 하는 범위를 구분하는 일이다. 한 사람이 끝까지 책임질 수 있는 부분은 빠르게 움직이게 두고, 여러 사람이 동시에 건드리면 깨지기 쉬운 부분은 역할과 절차를 분리해야 한다.

팀이 필요해지는 순간은 기능이 많아져서가 아니라, 지식이 흩어지기 시작할 때이다. 누가 봐도 이해할 수 있는 문서와 재현 가능한 운영 절차가 없다면, 사람 수가 늘수록 서비스는 오히려 불안정해진다. 따라서 팀이 커질수록 중요한 것은 코드의 양이 아니라 공유 가능한 판단의 양이다. 장애가 났을 때 누구나 같은 순서로 대응할 수 있고, 어떤 선택이 왜 내려졌는지 설명할 수 있어야 한다.

좋은 성장 판단은 “더 많은 사람을 붙이는 것”과 “더 잘 나누는 것”을 같이 본다. 모든 문제를 인력으로 해결하려고 하면 비용만 늘고, 모든 문제를 시스템으로만 해결하려고 하면 속도가 떨어진다. 결국 성장은 사람과 구조의 균형을 맞추는 일이다.

부록

부록 A: 응급 처치 플로우차트

장애가 났을 때 가장 먼저 해야 할 일은 수정이 아니라 범위 확인이다. 전체 사용자에게 공통으로 발생하는 문제인지, 특정 기능만 죽었는지, 특정 지역이나 브라우저에서만 보이는지, 최근 배포와 연관이 있는지부터 나누어 봐야 한다. 그다음에 로그, 대시보드, 최근 변경사항, 외부 서비스 상태를 함께 확인해야 한다. 장애 대응은 대담한 추측보다 정확한 순서가 더 중요하다.

부록 B: AI에게 구조 문제를 물을 때의 기준

AI는 운영 문제를 정리하는 데 강력한 도구이지만, 질문이 흐리면 답도 흐리다. 어떤 서비스인지, 어떤 경로에서 문제가 생겼는지, 최근 사용량이 어떠했는지, 어떤 지표가 흔들리는지, 무엇을 마지막으로 바꾸었는지를 함께 주어야 한다. 좋은 질문은 이미 문제를 반쯤 구조화한 질문이다. 구조화가 잘될수록 AI의 답도 실무적으로 쓸 수 있게 된다.

부록 C: 무료 티어를 바라보는 태도

무료 티어는 시작을 쉽게 해 주는 장치이지, 영원한 운영 모델이 아니다. 어떤 서비스는 무료 구간에서 오래 버틸 수 있지만, 어떤 서비스는 요청 패턴 때문에 금방 한계에 닿는다. 중요한 것은 무료를 유지하려는 집착이 아니라, 무료 구간이 내 서비스의 어떤 습관과 충돌하는지 보는 일이다. 무료는 목적이 아니라 출발점이다.