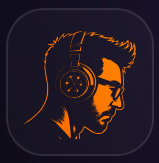

HANS17 ACADEMY · HANDS-ON MANUAL

실습 매뉴얼
설치부터 배포까지

계정 하나 없는 상태에서 시작해, 나만의 AI 에이전트를 실제 인터넷 주소로 배포하기까지. 이 한 권이면 됩니다.

부록 F	제로부터 배포까지 — 설치 · 계정 · 키 · Claude Code	
M5	개발 환경 + Claude Code 셋업	설정 · 스킬셋 · MCP · hooks + 서비스 9종
M6	Next.js 프로젝트 부트스트랩	App Router + TypeScript + Tailwind
M7	LLM API 통합 (Claude / OpenAI)	callLLM() 하나로 통합한다
M8	RAG 구축 — 임베딩·검색·환각 차단	in-memory 코사인 & 벡터 DB(pgvector)
M9	Vercel Blob 영속 저장소	DB 없이 KV 패턴으로 버틴다
M10	보고서 생성 에이전트	6단 표준 구조 + 결정적 조립
M11	STT / TTS 통합 (ElevenLabs)	내 목소리로 보고하는 에이전트
M12	메일 발송 + 공유 페이지	토큰 기반 공유 + 후속 Q&A
M13	배포 · 운영 · 확장	본인 도메인으로 production 배포

SETUP · 준비편

부록 F

제로부터 배포까지

계정 · 키 발급 & Claude Code로 서비스 만들기

부록 실습노트 — **제로부터 배포까지: 계정·키 발급 & Claude Code로 서비스 만들기** 이 문서 하나만 순서대로 따라 하면, 계정이 하나도 없는 상태에서 시작해 GitHub·Vercel·OpenAI·Supabase 계정과 키를 만들고, Claude Code로 코드를 생성해 실제 인터넷 주소로 **배포**까지 끝낼 수 있습니다. Hans17 Academy와 동일한 스택(Next.js 16 + TypeScript + Tailwind 4 · Vercel · Supabase · OpenAI 임베딩 · Claude · ElevenLabs)을 그대로 씁니다.
화면 명칭·메뉴 위치는 2025~2026년 기준입니다. 각 서비스가 UI를 자주 바꾸므로, 버튼 이름이 조금 다르면 **비슷한 뜻의 항목**을 찾으세요. 요금이 발생할 수 있는 지점에는 매번 경고를 달았습니다. 확실치 않아 원문 그대로 확인이 필요한 부분은 "(확인)"으로 표시했습니다.

0. 준비물과 전체 그림

먼저 최종적으로 우리가 채워 넣을 `.env.local` **키 목록**을 머리에 그려두면 길을 잃지 않습니다. 아래 키들이 이 실습의 "목표 수집물"입니다. 각 섹션이 하나씩 발급해 채워 줍니다.

```
# .env.local  (프로젝트 루트, git에 올리지 않음 — .gitignore에 포함)

# Claude (강사/에이전트 두뇌)
ANTHROPIC_API_KEY=sk-ant-...

# OpenAI (임베딩 전용: text-embedding-3-small)
OPENAI_API_KEY=sk-...

# Supabase (DB + 인증키 3종)
NEXT_PUBLIC_SUPABASE_URL=https://xxxx.supabase.co
NEXT_PUBLIC_SUPABASE_ANON_KEY=eyJ...        # 공개 가능(브라우저 노출 OK)
SUPABASE_SERVICE_ROLE_KEY=eyJ...            # 절대 공개 금지(서버 전용)

# ElevenLabs (음성 TTS/STT — 음성 기능 쓸 때만)
ELEVENLABS_API_KEY=...

# Vercel Blob (파일 업로드 저장 — 쓸 때만, 보통 Vercel이 자동 주입)
BLOB_READ_WRITE_TOKEN=vercel_blob_rw_...
```

준비물: 이메일 계정 하나, 휴대폰(문자 인증용), 신용/체크카드 1장(OpenAI 크레딧 결제 시 필요할 수 있음 — 소액), 그리고 Mac 또는 Windows PC.

용어 3개만 기억하세요. - **레포지토리(repo)**: 코드가 담기는 GitHub의 저장소. "프로젝트 폴더의 온라인 백업 + 협업 공간"입니다. - **환경변수(env)**: 코드에 직접 안 적고 따로 넣는 비밀값(키). 로컬은 `.env.local`, 배포 서버는

Vercel 대시보드에 넣습니다. - **배포(deploy)**: 내 코드가 인터넷 주소(https://...)로 실제 동작하게 올리는 것.

---

1. GitHub — 계정 생성·레포 생성·토큰 발급

GitHub는 코드 저장소이자, Vercel이 코드를 가져가는 출발점입니다. 여기 레포가 있어야 Vercel이 "이 코드 배포해줘"를 할 수 있습니다.

1-1. 회원가입 → 이메일 인증

1. 브라우저에서 **https://github.com** 접속 → 우측 상단 **Sign up** 클릭.
2. **Email**(실제 받을 수 있는 주소), **Password**(강력하게), **Username**(영문 소문자·숫자·하이픈, 나중에 URL에 노출됨 예: `github.com/내아이디`) 순서로 입력.
3. 사람 확인(퍼즐/캡차)을 통과합니다. 사이트가 요구하는 퍼즐은 **본인이 직접** 풉니다.
4. 입력한 이메일로 **8자리 인증 코드(launch code)** 가 옵니다. 코드를 입력하면 계정이 활성화됩니다. → 이메일 인증을 끝내지 않으면 레포 생성·푸시가 막히니 반드시 완료.
5. 요금 플랜은 **Free**로 충분합니다(공개·비공개 레포 무제한).

1-2. 새 레포지토리 생성 (New repository)

1. 로그인 후 좌측 상단 초록색 **New** 버튼(또는 우측 상단 **+** → **New repository**) 클릭.
2. 항목 입력: - **Repository name**: 예 `my-ai-agent` (소문자·하이픈 권장). - **Description**: 선택. "나의 첫 AI 에이전트" 등. - **Public / Private**: 학습용이면 **Private** 추천(코드가 남에게 안 보임). Vercel 무료 플랜은 Private도 배포 가능. - **Add a README file**: 체크하면 빈 레포가 아니라 파일 1개가 생겨 편함. (Claude Code로 로컬에서 시작할 거면 체크 안 해도 됨.) - **Add .gitignore**: 템플릿에서 **Node** 선택 → `node_modules`, `.env*` 등이 자동 제외됨. **중요**: 이게 있어야 `.env.local`(비밀키)이 실수로 업로드되지 않습니다.
3. **Create repository** 클릭. 생성된 주소는 `https://github.com/내아이디/my-ai-agent`.

> 주의: 절대 `.env.local`을 커밋하지 마세요. `.gitignore`에 `.env*`가 들어 있는지 반드시 확인. 키가 공개 레포에 올라가면 자동 스캐너가 몇 분 내 도용합니다.

1-3. Personal Access Token 발급 (범위: repo)

Claude Code나 로컬 git이 GitHub에 코드를 올릴 때 비밀번호 대신 **토큰**으로 인증합니다.

1. 우측 상단 프로필 사진 → **Settings**.
2. 좌측 맨 아래 **Developer settings** 클릭.
3. **Personal access tokens** → 두 종류가 있습니다. - **Tokens (classic)**: 간단. 범위 체크박스로 권한 지정. - **Fine-grained tokens**: 레포별 세밀 권한(권장·최신). 특정 레포에만 권한 부여 가능.
4. 초보자용 **classic** 기준: - **Generate new token (classic)** 클릭. - **Note**(용도 메모): `claude-code-push` 처럼 알아볼 이름. - **Expiration**(만료): 90일 권장(무기한은 보안상 비추천). - **Select scopes**에서 `repo` 전체 체크 (Private 레포 push·pull 전 권한). 이것만 있으면 코드 올리기·내리기 충분. - 맨 아래 **Generate token**.

5. 화면에 뜨는 `ghp_...` 토큰을 **즉시 복사**해 안전한 곳에 저장. → 이 화면을 벗어나면 **다시 볼 수 없습니다**. 잃어버리면 새로 발급.

> 용도: 로컬에서 `git push` 시 GitHub가 아이디/비밀번호를 물으면, 비밀번호 자리에 이 토큰을 붙여넣습니다(비밀번호로는 push 불가). Vercel이 GitHub와 연동될 때는 이 토큰이 아니라 GitHub 앱 권한(OAuth)을 쓰므로 토큰이 없어도 됩니다 — 즉 **Vercel 연동만 할 거면 토큰은 없어도 되고**, Claude Code가 터미널에서 직접 push하게 할 때 유용합니다.

---

2. Vercel — 계정·프로젝트·배포·환경변수·Blob·Supabase 통합

Vercel은 Next.js를 만든 회사의 배포 플랫폼입니다. GitHub 레포를 연결하면 **git push만 해도 자동 배포**됩니다.

2-1. GitHub로 로그인

1. **https://vercel.com** → **Sign Up**.
2. **Continue with GitHub** 선택(이게 핵심 — GitHub와 바로 연결됨). GitHub 인증 화면에서 **Authorize Vercel**.
3. 개인 학습용이면 팀 유형은 **Hobby(무료)** 선택. 무료로 배포·커스텀 도메인·환경변수 다 됩니다.

2-2. 프로젝트 Import & 배포

1. 대시보드에서 **Add New... → Project**.
2. **Import Git Repository** 목록에서 1장에서 만든 레포(`my-ai-agent`)를 찾아 **Import**. 목록에 없으면 **Adjust GitHub App Permissions**(또는 "Configure GitHub App")로 그 레포 접근 권한을 Vercel에 부여.
3. 설정 화면: - **Framework Preset**: `Next.js`로 자동 감지됨. - **Root Directory**: 레포 루트에 코드가 있으면 그대로. - **Build & Output Settings**: 기본값 유지(Next.js는 자동). Hans17처럼 커스텀 빌드 커맨드가 있다면 여기서 지정. - **Environment Variables**: 지금 넣어도 되고 나중에 넣어도 됨(다음 항목).
4. **Deploy** 클릭 → 1~3분 빌드 후 `https://my-ai-agent.vercel.app` 같은 주소가 생깁니다. 처음엔 코드가 비어 있어도 빈 페이지가 뜨면 성공.

2-3. 환경변수 넣기 (가장 중요)

로컬 `.env.local`의 키들을 **똑같이** Vercel에도 넣어야 배포본이 동작합니다(로컬 파일은 서버에 안 올라감).

1. 프로젝트 → 상단 **Settings** 탭 → 좌측 **Environment Variables**.
2. 각 키를 **Key / Value** 쌍으로 추가. 예: Key `ANTHROPIC_API_KEY`, Value `sk-ant-...`.
3. 적용 환경 선택: **Production / Preview / Development** — 보통 **셋 다 체크**.
4. `NEXT_PUBLIC_`로 시작하는 키는 브라우저에 노출되어도 되는 값(예: Supabase anon key, Supabase URL)이고, 접두어 없는 키(`SUPABASE_SERVICE_ROLE_KEY`, `ANTHROPIC_API_KEY`, `OPENAI_API_KEY`)는 **서버 전용 비밀**입니다. 이 규칙은 Next.js가 강제합니다 — `NEXT_PUBLIC_`이 없으면 브라우저 코드에서 읽을 수 없습니다.
5. 값을 추가/변경한 뒤에는 **재배포**해야 반영됩니다: **Deployments** 탭 → 최신 배포 우측 **⋯ → Redeploy**.

> 팁: `vercel env pull` 명령으로 Vercel에 넣은 값을 로컬 `.env.local`로 내려받을 수 있고, 반대로 Vercel

CLI로 올릴 수도 있습니다. Claude Code가 이 동기화를 도와줄 수 있습니다(5장).

2-4. Blob 토큰 — BLOB_READ_WRITE_TOKEN

Vercel Blob은 이미지·오디오 같은 파일을 저장하는 스토리지입니다(음성 파일 저장 등에 사용).

1. 프로젝트 → **Storage** 탭 → **Create Database**(또는 **Connect Store**) → **Blob** 선택 → 이름 짓고 생성.
2. 생성 후 **Connect to Project**를 하면 Vercel이 `BLOB_READ_WRITE_TOKEN`을 **환경변수로 자동 주입**합니다(직접 복붙 안 해도 됨).
3. 로컬에서도 쓰려면 `vercel env pull`로 이 토큰을 `.env.local`에 내려받습니다.

> 용도: 코드에서 `@vercel/blob` 패키지로 파일 업로드 시 이 토큰으로 인증. 파일 기능을 안 쓰면 생략 가능.

2-5. Marketplace에서 Supabase 통합 (선택)

Supabase를 4장처럼 직접 만들어도 되고, Vercel Marketplace로 연결하면 환경변수를 자동으로 꽂아줘 더 편합니다.

1. 프로젝트 → **Storage** 또는 상단 **Integrations**(팀 대시보드 → **Integrations → Browse Marketplace**).
2. **Supabase** 선택 → **Add Integration** → 프로젝트 선택.
3. 새 Supabase 프로젝트를 만들거나 기존 것을 연결하면, Vercel이 `NEXT_PUBLIC_SUPABASE_URL`, `NEXT_PUBLIC_SUPABASE_ANON_KEY`, `SUPABASE_SERVICE_ROLE_KEY` 등을 **자동으로 환경변수에 추가**합니다.
4. 이 방식을 쓰면 4장의 키 복붙 과정을 건너뛸 수 있습니다. (단, 키가 무엇인지 이해하려면 4장을 읽어 두세요.)

2-6. Deployment Protection (배포 보호)

배포한 주소를 아무나 못 보게 막는 기능. 개발 중 미완성 화면 노출을 막습니다.

1. 프로젝트 → **Settings** → **Deployment Protection**.
2. **Vercel Authentication**: 켜면 Vercel 로그인한 사람만 접근(팀 내부 검토용). Preview 배포에 특히 유용.
3. 일반 사용자에게 공개할 서비스라면 **Production은 보호 해제**(Disabled)로 두어야 방문자가 로그인 없이 접속합니다. → 라이브 서비스인데 접속이 막혀 있다면 이 설정을 먼저 확인.
4. **Password Protection**(유료 플랜): 비밀번호 아는 사람만 접근. 소규모 비공개 데모에 적합.

---

3. OpenAI — 계정·API 키 (임베딩 전용)

우리 스택에서 OpenAI는 **오직 임베딩**(`text-embedding-3-small`, 1536차원)에만 씁니다. 강의노트 텍스트를 숫자 벡터로 바꿔 RAG 검색에 쓰는 용도입니다. 대화 생성은 Claude가 담당하므로 GPT 대화 모델은 필요 없습니다.

3-1. 계정 생성

1. **https://platform.openai.com** 접속 → **Sign up**(구글/이메일 가입 가능).
2. 휴대폰 번호 인증(SMS 코드)이 요구될 수 있습니다.
3. 로그인하면 개발자용 **Platform** 대시보드가 나옵니다(챗봇 화면 chatgpt.com과 다름 — 반드시 platform.openai.com).

3-2. 결제/크레딧 설정

임베딩 API는 유료입니다(다만 매우 저렴 — text-embedding-3-small은 100만 토큰당 수 센트 수준(확인)). 사용하려면 크레딧 충전이 필요합니다.

1. 좌측/우측 상단 **Settings(톱니) → Billing**.
2. **Add payment details**로 카드 등록 → **Add credits**로 소액(예 $5) 선충전. → 크레딧 없이 키만 있으면 호출이 429/quota 오류로 실패합니다.
3. (권장) **Usage limits**에서 월 상한(hard limit)을 낮게 걸어 과금 사고 방지.

> 카드번호·결제정보는 **본인이 직접** OpenAI 사이트에 입력하세요. 저(조교)는 결제정보를 대신 입력하지 않습니다.

3-3. API 키 발급 → OPENAI_API_KEY

1. 좌측 메뉴 **API keys**(또는 Settings → API keys).
2. **Create new secret key** 클릭.
3. **Name**: `hans17-embedding` 등 용도 메모. **Project** 선택(기본 프로젝트 OK). 권한은 기본(All) 또는 임베딩만 필요하면 제한 가능.
4. 생성된 `sk-...` 키를 **즉시 복사**. → 이후 다시 볼 수 없음. 잃으면 재발급.
5. `.env.local`에 넣기: `bash OPENAI_API_KEY=sk-...`
6. Vercel에도 동일하게 `OPENAI_API_KEY` 추가(2-3 참고).

> 용도 요약: 코드의 임베딩 생성 스크립트(예 `npm run embed`)가 이 키로 OpenAI를 호출해 `content/*.md`를 벡터로 만들어 `embeddings.json`에 저장 → 런타임에 코사인 유사도로 관련 문단을 찾음(벡터DB 불필요).

---

4. Supabase — 계정·프로젝트·API 키·테이블·RLS

Supabase는 PostgreSQL 데이터베이스 + 인증 + 스토리지를 얹은 백엔드입니다. 대화 로그·FAQ·수집 데이터 저장에 씁니다.

4-1. 계정 & New project

1. **https://supabase.com** → **Start your project** → **Continue with GitHub** 로그인.
2. **New organization**(팀 이름·플랜=Free) 생성 → **New project**.
3. 항목: - **Name**: `my-ai-agent`. - **Database Password**: 강력하게 설정하고 **따로 저장**(DB 직접접속·마이그레이션에 필요, 분실 시 재설정). - **Region**: 사용자와 가까운 곳(한국이면 **Northeast Asia (Seoul/Tokyo)** 권장 — 지연 감소). - **Plan**: Free.
4. **Create new project** → 1~2분 프로비저닝.

4-2. API 키 3종 확보 → .env.local

프로젝트 생성 후 좌측 하단 **Settings(톱니) → API**(또는 **Project Settings → API / API Keys**)로 이동: - **Project URL** → `NEXT_PUBLIC_SUPABASE_URL` (`https://xxxx.supabase.co`) - **anon public** 키 → `NEXT_PUBLIC_SUPABASE_ANON_KEY` (브라우저 노출 OK, RLS로 보호되는 공개 키) - **service_role** 키 →

`SUPABASE_SERVICE_ROLE_KEY` (**RLS를 우회하는 관리자 키 — 절대 클라이언트/공개 레포에 노출 금지, 서버 코드에서만**)

```
NEXT_PUBLIC_SUPABASE_URL=https://xxxx.supabase.co
NEXT_PUBLIC_SUPABASE_ANON_KEY=eyJhbGciOi...
SUPABASE_SERVICE_ROLE_KEY=eyJhbGciOi...
```

> 참고: Supabase가 2025년 들어 키 체계를 새로운 **publishable / secret 키**로 개편 중입니다(확인). 화면에 `anon/service_role` 대신 `publishable/secret`으로 보이면, publishable ≈ anon(공개), secret ≈ service_role(비밀)로 대응해 같은 환경변수에 넣으면 됩니다. Vercel Marketplace 통합(2-5)을 쓰면 이 값들이 자동 주입됩니다.

Vercel 대시보드에도 위 3개를 동일하게 추가(2-3).

4-3. 테이블 만들기

두 가지 방법: 1. **Table Editor**(좌측 메뉴) → **New table** → 이름·컬럼 지정(GUI). 예: `chat_logs` (id, created_at, question text, answer text, ...). 2. **SQL Editor**(좌측 메뉴) → SQL 직접 실행: `sql create table chat_logs ( id bigint generated always as identity primary key, created_at timestamptz default now(), question text, answer text );` Claude Code에게 "이런 테이블 만들어줘"라고 하면 이 SQL을 대신 작성/실행(MCP 연결 시)해 줍니다(5장).

4-4. RLS (Row Level Security) — 반드시 이해

RLS는 "행 단위 접근 제어"입니다. **anon 키로 오는 요청**은 RLS 정책이 허용한 행만 접근합니다.

- 새 테이블은 기본적으로 RLS가 **켜져 있고 정책이 없어** anon 키로는 아무것도 못 읽/씁니다(안전한 기본값).
- 공개적으로 읽기만 허용하려면 정책 추가: `sql alter table chat_logs enable row level security; create policy "public read" on chat_logs for select using (true);`
- **쓰기(insert)는 service_role 키를 쓰는 서버 코드로만** 하는 게 안전합니다. service_role 키는 RLS를 **우회**하므로, 서버(예: Next.js API 라우트)에서만 이 키로 저장하세요. → 그래서 클라이언트엔 anon, 서버 저장엔 service_role로 역할을 나눕니다.
- **경고**: service_role 키가 브라우저 번들이나 GitHub에 노출되면 DB 전체가 무방비가 됩니다. `NEXT_PUBLIC_` 접두어를 절대 붙이지 마세요.

---

5. Claude Code — 가입 & 위 서비스 연결·셋팅

5-1. Claude Code란

Claude Code는 **터미널에서 동작하는 AI 코딩 에이전트**입니다. 자연어로 "이런 기능 만들어줘"라고 하면 파일을 직접 읽고/쓰고, 명령을 실행하고, git 커밋·배포까지 수행합니다. 앞 장에서 모은 키들을 이 도구가 코드에 연결해 줍니다.

5-2. 가입 & 설치

1. **Claude 계정**: **https://claude.ai** 가입. Claude Code 사용에는 유료 플랜(Pro/Max) 또는 Anthropic Console의 API 크레딧이 필요합니다(확인 — 플랜별 제공 범위는 공식 문서에서 확인).
2. **설치**(Node.js 18+ 필요): `bash npm install -g @anthropic-ai/claude-code` 또는 공식 안내의 설치 스크립트 사용.
3. 프로젝트 폴더에서 실행: `bash cd my-ai-agent claude` 최초 실행 시 브라우저로 **로그인/인증**을 진행합니다(계정 연결). 안내에 따라 승인.

5-3. .env.local에 키 모으기

프로젝트 루트에 `.env.local`을 만들고 1~4장에서 모은 키를 전부 넣습니다(0장의 목록 참고). Claude Code에게 이렇게 시켜도 됩니다:

> "루트에 `.env.local` 만들고 이 키들 넣어줘. 그리고 `.gitignore`에 `.env*` 있는지 확인해줘."

Claude Code는 키 **값** 자체를 코드에 하드코딩하지 않고 `process.env.KEY`로 참조하도록 작성합니다. 값은 여러분이 붙여넣습니다.

5-4. MCP로 Supabase / Vercel 연결

MCP(Model Context Protocol)는 Claude Code가 외부 서비스를 "도구"로 직접 다루게 해주는 연결 규격입니다. 연결하면 Claude가 브라우저 없이 DB에 테이블을 만들거나 배포 상태를 조회할 수 있습니다.

- **Supabase MCP**: 연결하면 Claude Code가 SQL 실행, 테이블/마이그레이션 생성, 로그 조회를 직접 수행. 등록 예: `bash claude mcp add supabase --scope project -- npx -y @supabase/mcp-server-supabase@latest` (연결에는 Supabase 액세스 토큰/프로젝트 ref가 필요할 수 있음 — 안내에 따라 입력. 정확한 패키지명·인자는 Supabase 공식 MCP 문서에서 확인.)
- **Vercel MCP**: 배포 목록·빌드 로그·환경변수·배포 보호를 Claude가 조회·조작. Vercel의 MCP 서버/커넥터를 등록하고 OAuth로 인증(확인 — 최신 등록 방식은 Vercel 문서 확인). 인증은 대화형 세션에서 `/mcp` 또는 커넥터 설정으로 진행합니다.
- 연결된 MCP는 세션 안에서 `/mcp`로 상태를 확인할 수 있습니다.

> 주의: MCP 인증(OAuth)은 대화형(interactive) 세션에서만 됩니다. 처음엔 브라우저로 권한 승인 창이 뜹니다 — 본인이 직접 승인.

5-5. CLAUDE.md — 프로젝트 규칙 메모

루트에 `CLAUDE.md`를 두면 Claude Code가 **매번 자동으로 읽는** 프로젝트 설명서가 됩니다. 스택·명령어·규칙을 적어두면 헛짓을 줄입니다. 예:

```
# My AI Agent — 컨텍스트
- 스택: Next.js 16 + TS + Tailwind 4, Vercel 배포, Supabase, OpenAI 임베딩, Claude.
- 로컬 실행: `npm run dev`. 임베딩 재생성: `npm run embed`.
- 배포: `vercel --prod`.
- 규칙: 비밀키는 .env.local에만. service_role 키는 서버 코드에서만 사용.
```

`/init` 명령을 쓰면 Claude Code가 코드베이스를 훑어 CLAUDE.md 초안을 만들어 줍니다.

5-6. 권한/설정

- Claude Code는 파일 수정·명령 실행 전에 **권한을 물어봅니다**. 안전한 명령은 "허용"으로, 위험한 명령(삭제 등)은 매번 확인.
- 자주 쓰는 안전한 명령은 `.claude/settings.json`의 allowlist에 넣어 확인 프롬프트를 줄일 수 있습니다.
- 비밀키·설정 파일은 신뢰할 수 있는 본인 지시로만 변경하세요.

---

6. Claude Code와 연결된 상태에서 서비스 만드는 법

이제 실전 루프입니다. **자연어 요구 → 코드 생성 → 로컬 검증 → 브라우저 확인 → 커밋 → 배포**를 반복합니다.

6-1. 기본 루프

1. **요구를 자연어로**: 예)

   > "질문을 입력하면 Claude가 답하고, 질문·답변을 Supabase `chat_logs`에 저장하는 채팅 페이지를 만들어줘. 답변 생성은 서버 API 라우트에서 ANTHROPIC_API_KEY로 호출하고, 저장은 service_role 키로 서버에서만 해줘."

2. **코드 생성**: Claude Code가 `app/`에 페이지·API 라우트, 필요한 패키지 설치(`npm install`)까지 수행. 진행 중 권한을 물으면 확인.
3. **로컬 검증**: `bash npm run dev` `http://localhost:3000`이 뜹니다. 에러가 나면 그 로그를 그대로 Claude에게 붙여넣고 "이 에러 고쳐줘".
4. **브라우저 확인**: 실제로 질문을 넣어 답이 오는지, Supabase **Table Editor**에 행이 쌓이는지 눈으로 확인.
5. **커밋**:

   > "지금까지 변경 커밋해줘. 메시지는 'feat: 채팅+로그 저장'." Claude가 `git add/commit`을 수행(비밀키 파일은 .gitignore로 제외됨).

6. **배포**: `bash vercel --prod` 또는 GitHub에 push하면 Vercel이 자동 배포. 배포 후 `.vercel.app` 주소에서 최종 확인. → 로컬은 되는데 배포본이 안 되면 **99%가 Vercel 환경변수 누락**(2-3). 그 키를 추가하고 Redeploy.

6-2. 프롬프트 팁

- **역할과 제약을 함께**: "무엇을" 뿐 아니라 "어디서(서버/클라이언트), 어떤 키로, 어떤 제약으로"를 명시. 예: "service_role 키는 클라이언트에 노출하지 말고 API 라우트에서만."
- **작게 쪼개기**: 한 번에 거대한 기능 대신 "먼저 채팅 UI만 → 다음에 저장 → 다음에 RAG" 순서로. 검증 지점이 잦을수록 디버깅이 쉽습니다.
- **에러는 원문 그대로**: 터미널·브라우저 콘솔 에러를 통째로 붙여넣으세요. 요약하지 말고.
- **검증을 시켜라**: "npm run dev로 띄워서 실제로 동작하는지 확인하고, 안 되면 고칠 때까지 반복해줘."
- **되돌리기**: 마음에 안 들면 "방금 변경 되돌려줘(git으로)". 그래서 자주 커밋하는 게 안전망.

- **비용 주의**: 임베딩 재생성(`npm run embed`)은 OpenAI 과금이 있으니 콘텐츠가 바뀐 뒤에만.

6-3. RAG 파이프라인을 붙일 때 (이 스택의 핵심 흐름)

1. 답변에 참고시킬 지식 문서를 `content/*.md`로 작성.
2. `npm run embed` — OpenAI `text-embedding-3-small`로 각 문단을 벡터화해 `embeddings.json` 생성.
3. 사용자의 질문도 같은 모델로 임베딩 → **코사인 유사도**로 가장 가까운 문단 몇 개를 뽑음(벡터DB 없이 인메모리 계산).
4. 뽑힌 문단을 Claude 프롬프트에 컨텍스트로 넣어 답변 생성. "주어진 근거에 없으면 모른다고 답하라"는 지시로 **환각을 차단**.
5. 배포 시 `content/*.md`와 `embeddings.json`을 함께 커밋해야 서버에서 같은 답을 냅니다.

---

7. 서비스 기획에 필요한 것 — Hans17 서비스의 특장점 요약

좋은 AI 서비스는 "모델이 좋아서"가 아니라 **문제·사용자·차별화가 뾰족해서** 좋습니다. 기획 뼈대와, Hans17 서비스가 그 뼈대를 어떻게 채웠는지(그리고 왜 좋은지)를 나란히 봅니다.

7-1. 기획 3요소

1. **문제 정의**: 누구의 어떤 불편을, 지금은 어떻게(불편하게) 해결하나? — 한 문장으로.
2. **사용자**: 대상이 구체적일수록 기능이 선명해집니다("여행자" 대신 "혼자 처음 일본 밤거리를 다니는 20~30대 한국인").
3. **차별화**: 남들과 다른 한 가지. 기술이 아니라 **사용자가 체감하는 결과**로 표현.

7-2. Hans17의 실제 특장점 — "왜 좋은가"로

- **벡터DB 없는 하이브리드 코사인 RAG** — 별도 벡터 데이터베이스(Pinecone 등)를 두지 않고 임베딩을 파일로 갖고 인메모리 코사인 + 키워드를 섞어 검색합니다. **왜 좋은가**: 인프라가 1개 줄어 배포·운영·비용이 단순해지고, 강의 규모의 데이터에선 지연도 짧습니다. 키워드를 섞어 고유명사·숫자 같은 "의미 임베딩이 놓치는 것"까지 잡아 정확도가 올라갑니다.
- **환각 차단** — "근거 문단에 없으면 지어내지 말고 모른다고 답하라"를 구조로 강제합니다. **왜 좋은가**: 교육·상담처럼 **틀린 답의 대가가 큰** 도메인에서 신뢰를 지킵니다. 그럴듯한 거짓말보다 "모른다"가 더 안전합니다.
- **웹검색 + 리플렉션(2차 자기검증)** — Anthropic 서버사이드 웹검색으로 최신 정보를 가져온 뒤, 답을 한 번 더 스스로 검토(reflection)합니다. **왜 좋은가**: 지식 컷오프 이후의 최신성 문제를 메우고, 2차 검증으로 초안의 오류·과장을 걸러 품질을 끌어올립니다.
- **클론 음성 TTS/STT** — ElevenLabs로 복원한 실제 목소리로 답을 읽어주고(TTS), 사용자의 말을 받아씁니다(STT). **왜 좋은가**: 텍스트만인 봇 대비 **정체성·몰입·접근성**(화면 못 보는 상황, 이동 중)이 확 올라가 서비스가 기억에 남습니다.
- **GPS + 날씨 + 기념일 컨텍스트** — 위치(역지오코딩), 실시간 날씨, 그 나라의 공휴일을 자동으로 읽어 답에 반영합니다. **왜 좋은가**: 같은 질문이라도 **"지금·여기·오늘"에 맞는** 답을 주므로 조언이 실제로 쓸모 있어집니다(비 오면 실내 추천, 축일이면 붐빔 경고).

- **현지어 자동 전환** — 사용자가 위치한 나라의 언어로 서비스 전체가 동작합니다. **왜 좋은가**: 여행지에서 언어 장벽을 없애 "번역 앱 따로 켜기"를 없앱니다 — 하나의 흐름 안에서 통역·말걸기까지 이어집니다.
- **대화세트 사전 추론** — 상황별 대화 시나리오(단계·상대 관점 추론 포함)를 미리 설계해 둡니다. **왜 좋은가**: 실시간 생성만 의존할 때의 들쭉날쭉함을 없애고, 검증된 흐름으로 **첫 마디부터 자연스럽게** 이어가게 합니다.
- **DB 누적 심층분석** — 수집·분석 결과를 Supabase에 계속 쌓습니다. **왜 좋은가**: 쓸수록 데이터가 축적되어 분석이 깊어지고, 개인화·재방문 가치가 커지는 **복리형 자산**이 됩니다.

7-3. 관통하는 원칙 — "사이트가 곧 교재"

Hans17 Academy는 강의에서 가르친 기술 선택(RAG·환각차단·리플렉션 등)을 **사이트 코드가 그대로 구현**합니다. 여러분의 서비스도 "설명한 대로 실제로 동작하는가"를 기준으로 만들면, 데모가 곧 신뢰가 됩니다.

7-4. 기획 → 이 문서로 실행하는 순서(요약)

1. 문제·사용자·차별화 한 문장씩 정의(7-1).
2. GitHub 레포(1장) → Vercel 배포 뼈대(2장).
3. 필요한 키만 발급: 대화=Claude, 검색용 벡터화=OpenAI(3장), 저장=Supabase(4장), 음성=ElevenLabs(선택).
4. Claude Code 연결(5장)로 자연어 개발 루프(6장) 시작.
5. 작게 배포하고 실제 사용자에게 보여주며 차별화 한 가지를 계속 날카롭게.

> 마지막 점검: 배포본이 동작하려면 (a) Vercel 환경변수에 모든 키가 있고, (b) service_role 같은 비밀키가 GitHub에 노출되지 않았고, (c) Supabase RLS 정책이 의도대로 설정됐는지 — 이 셋을 항상 확인하세요.

PRACTICE · 실습편

M5

개발 환경 + Claude Code 셋업

설정 · 스킬셋 · MCP · hooks + 서비스 9종

이 강의의 실습은 **손으로 타이핑하는 코딩이 아니라, Claude Code에 프롬프트로 지시해 만드는 방식**입니다. (지금 보는 이 사이트도 그렇게 만들어졌어요.) 그래서 가장 먼저 할 일은 **Claude Code를 제대로 세팅**하는 것 — 설정, 스킬셋, MCP, hooks. 그다음 9개 서비스 키를 한 파일로 모읍니다.

로컬 환경

- **Node.js 24 LTS** (Mac/Win)
- **Git** + 터미널
- **Claude Code 설치**: `npm i -g @anthropic-ai/claude-code` → 프로젝트 폴더에서 `claude` 실행 (VS Code/JetBrains 확장, 데스크탑·웹 앱으로도 가능)

Claude Code 설정 — settings.json · 권한 · 모델

프로젝트 루트 `.claude/settings.json`(프로젝트별) 또는 `~/.claude/settings.json`(전역):

```
{
  "permissions": {
    "allow": ["Bash(npm run *)", "Bash(vercel *)", "Read", "Edit", "Write"],
    "deny": ["Bash(rm -rf *)"]
  },
  "model": "claude-opus-4-8"
}
```

- **permissions.allow**: 자주 쓰는 명령을 넣어두면 매번 승인 팝업 없이 흐름이 끊기지 않습니다. 위험한 건 `deny`.
- **model**: 작업 난이도에 맞게 (대화 중 `/model`로도 전환).
- `/config`로 테마·모델 등 간단 설정.

팁: 승인 피로가 심하면 `/permissions`로 자주 쓰는 읽기 전용 명령을 allow에 추가하세요. 단, 파괴적 명령은 절대 와일드카드로 열지 말 것.

스킬셋(Skills) — 반복 작업을 명령으로 박제

Skill = "이런 요청이 오면 이렇게 처리하라"를 담은 폴더. 자연어로 부르거나 슬래시 명령(`/이름`)으로 호출합니다.

- 위치: `.claude/skills/<이름>/SKILL.md`

- 형식:

```
---
name: deploy-check
description: 배포 전 빌드·린트·점검. "배포 전 점검" 같은 요청에 사용.
---
1. `npm run build` 실행 후 에러 확인
2. 린트·타입 오류 정리
3. 핵심 페이지·API 200 확인
```

- **description이 트리거**입니다 — Claude가 이 설명을 보고 자동 호출 여부를 판단하니, "언제 쓰는지"를 명확히 적으세요.
- 호출: `/deploy-check` 또는 그냥 "배포 전 점검해줘".
- **마켓플레이스 스킬**도 설치해 씁니다 — 예: `vercel` (배포·env·로그), `anthropic-skills` (pdf·docx·pptx·xlsx 생성) 등. (이 강의의 평가서 PDF도 스킬로 만들었어요.)

MCP 서버 — 외부 도구를 Claude 손에 쥐어주기

MCP(Model Context Protocol)로 Vercel·Supabase·ElevenLabs·GitHub 등을 Claude가 *직접* 조작하게 합니다.

```
// .mcp.json (프로젝트) — 예시
{ "mcpServers": {
  "vercel":   { "command": "npx", "args": ["-y", "@vercel/mcp"] },
  "supabase": { "command": "npx", "args": ["-y", "@supabase/mcp-server-supabase"] }
} }
```

- 연결되면 Claude가 배포·로그·DB·도메인을 코드 없이 호출합니다. (이 강의도 Vercel·Supabase·ElevenLabs MCP를 그대로 씁니다.)
- 커넥터(원클릭 연결)로 붙이는 방법도 있습니다.
- 도구가 너무 많아지면 **ToolSearch**로 필요한 것만 그때그때 로드 — 컨텍스트 절약.

Hooks — 자동화 (Claude가 아니라 하네스가 실행)

특정 시점에 명령을 자동 실행합니다. "저장할 때마다 X" 같은 반복은 메모리/지시가 아니라 **hooks**로.

```
// .claude/settings.json
{ "hooks": {
  "PostToolUse": [
    { "matcher": "Edit|Write", "hooks": [{ "type": "command", "command": "npx
prettier --write $CLAUDE_FILE" }] }
  ]
} }
```

- 예: 편집 후 자동 포맷, 커밋 전 점검, 작업 종료 알림.

외부 서비스 9종 + `.env.local`

#	서비스	용도	무료 한도
1	GitHub	소스·자동배포	무료
2	Vercel	호스팅·Blob·Functions	무료(+Blob 1GB)
3	Anthropic	Claude API	$5 크레딧
4	OpenAI	임베딩·GPT	$5 크레딧
5	ElevenLabs	STT·TTS	월 10k자
6	Resend	메일	월 3,000건
7	Brave Search	웹검색(선택)	월 2,000건
8	Vercel AI Gateway	LLM 라우팅(권장)	무료
9	Sentry	모니터링(선택)	월 5,000건

발급한 키는 `.env.local` **한 파일**로 모읍니다. (절대 커밋 금지 — `.gitignore` 확인. 템플릿은 부록 B.)

핵심 정리

- 실습 도구는 **Claude Code** — 손코딩 대신 **프롬프트로 지시**해서 만든다.
- 설정은 `.claude/settings.json`(**권한·모델**), 반복 작업은 **Skills**(`.claude/skills/*/SKILL.md`), 외부 도구는 **MCP**, 자동화는 **Hooks**.
- **권한 allow**를 잘 넣으면 승인 피로가 줄고 작업이 빨라진다. 파괴적 명령은 deny.
- Skill의 **description**이 자동 호출 트리거 — "언제 쓰는지"를 분명히.
- 키 9종은 `.env.local` 한 파일에, 커밋 금지.

PRACTICE · 실습편

M6

Next.js 프로젝트 부트스트랩

App Router + TypeScript + Tailwind

이 모듈은 강의 전체의 뼈대가 되는 Next.js 프로젝트를 직접 손으로 만들어 보는 시간입니다. 이후 모든 모듈(RAG, Agent, 음성, 보고서)은 여기서 잡은 구조 위에 쌓이므로, 지금 설계를 제대로 이해해 두는 것이 핵심입니다.

프로젝트 생성

터미널에서 아래 명령 한 줄로 시작합니다.

```
npx create-next-app@latest hans17-ai \
  --typescript \
  --tailwind \
  --eslint \
  --app \
  --src-dir \
  --import-alias "@/*"
```

`--app` 플래그가 App Router를 활성화합니다. Next.js 13 이후 기본값이지만, 명시적으로 지정하는 습관을 들이세요.

디렉터리 구조

생성 직후 `src/` 아래를 아래처럼 정리합니다. 강의 전체에서 이 구조를 일관되게 유지합니다.

```
src/
  app/
    (auth)/
      login/
        page.tsx        ← 로그인 페이지
    (protected)/
      dashboard/
        page.tsx        ← 보호된 메인 페이지
    api/
      chat/
        route.ts        ← Route Handler
    layout.tsx
    page.tsx
  components/           ← 재사용 UI 컴포넌트
  lib/
    llm.ts              ← callLLM() 통합 래퍼
    auth.ts             ← 세션 유틸
  public/
middleware.ts           ← 인증 보호 (app/ 바깥)
```

`(auth)`, `(protected)` 같은 괄호 폴더는 Route Group으로, URL 경로에는 포함되지 않고 레이아웃만 구분합니다.

Server vs Client Components

App Router의 가장 중요한 개념입니다. 파일 상단에 아무 선언도 없으면 **Server Component**가 기본값입니다.

구분	선언	할 수 있는 것	할 수 없는 것
Server Component	(없음)	DB/API 직접 호출, 환경변수 사용	useState, useEffect, 브라우저 API
Client Component	`"use client"`	인터랙션, 훅, 이벤트 핸들러	서버 전용 모듈 import

> **실수 방지**: `process.env.ANTHROPIC_API_KEY` 같은 비밀 환경변수는 Server Component나 Route Handler에서만 사용하세요. `"use client"` 파일에서 참조하면 브라우저에 노출됩니다. 브라우저에 공개해도 되는 값만 `NEXT_PUBLIC_` 접두사를 붙입니다.

Route Handler (route.ts)

API 엔드포인트는 `app/api/*/route.ts` 파일로 정의합니다. Express의 라우터와 동일한 역할이지만, Edge/Node.js 런타임 선택이 가능합니다.

```ts
// src/app/api/chat/route.ts
import { NextRequest, NextResponse } from "next/server";

export const runtime = "nodejs"; // 기본값; Edge로 바꾸면 경량 실행

export async function POST(req: NextRequest) {
  const { message } = await req.json();

  // 이후 모듈에서 callLLM()을 여기에 연결합니다
  return NextResponse.json({ reply: `echo: ${message}` });
}
```

`GET`, `POST`, `PUT`, `DELETE` 함수를 같은 파일에 나란히 export하면 HTTP 메서드가 자동으로 라우팅됩니다.

환경변수

`.env.local`에 키를 선언하고, `process.env`로 읽습니다.

```
# .env.local
ANTHROPIC_API_KEY=sk-ant-...
OPENAI_API_KEY=sk-...
ELEVENLABS_API_KEY=...
RESEND_API_KEY=re_...
SESSION_SECRET=최소32자이상의랜덤문자열
```

> `.env.local`은 **절대 Git에 커밋하지 마세요.** `.gitignore`에 이미 포함되어 있지만 첫 커밋 전에 반드시 확인하세요.

middleware.ts — 인증 보호

`src/` 바깥, 프로젝트 루트에 위치합니다. 요청이 페이지/API에 도달하기 전에 실행됩니다.

```ts
// middleware.ts
import { NextRequest, NextResponse } from "next/server";

const PUBLIC_PATHS = ["/login", "/api/auth"];

export function middleware(req: NextRequest) {
  const { pathname } = req.nextUrl;
  const isPublic = PUBLIC_PATHS.some((p) => pathname.startsWith(p));

  const token = req.cookies.get("session")?.value;

  if (!isPublic && !token) {
    return NextResponse.redirect(new URL("/login", req.url));
  }
  return NextResponse.next();
}

export const config = {
  // _next/static 등 정적 파일은 미들웨어에서 제외
  matcher: ["/((?!_next/static|_next/image|favicon.ico).*)"],
};
```

세션 토큰 검증 로직(`auth.ts`)은 M7에서 확장하지만, 미들웨어 패턴 자체는 지금 완성해 두세요.

Dev 서버 + Hot Reload

```bash
npm run dev
```

`http://localhost:3000`에서 즉시 확인 가능합니다. 파일을 저장하면 브라우저가 자동으로 갱신됩니다(Fast Refresh). Server Component를 수정해도 전체 리로드 없이 서버만 다시 렌더링됩니다.

실습 (Lab)

목표

로그인 페이지(`/login`)와 미들웨어로 보호된 대시보드(`/dashboard`)를 완성합니다.

단계별 안내

Step 1 — 프로젝트 생성 위 `create-next-app` 명령을 실행하고 디렉터리 구조를 위 표대로 만드세요.

Step 2 — 로그인 페이지 구현 `src/app/(auth)/login/page.tsx`를 Client Component로 작성합니다. 이메일/비밀번호 폼을 만들고, 제출 시 `POST /api/auth/login`을 호출합니다. 성공하면 쿠키에 `session` 토큰을 심고 `/dashboard`로 이동합니다.

Step 3 — 보호된 대시보드 구현 `src/app/(protected)/dashboard/page.tsx`를 Server Component로 작성합니다. `cookies()` 유틸로 세션을 읽어 사용자 이름을 표시합니다.

Step 4 — 미들웨어 연결 위 `middleware.ts` 코드를 그대로 붙여넣고, 로그인 없이 `/dashboard`에 접근하면 `/login`으로 리다이렉트되는지 확인합니다.

Step 5 — 동작 확인 1. `npm run dev` 실행 2. `http://localhost:3000/dashboard` 직접 접근 → `/login` 리다이렉트 확인 3. 로그인 후 대시보드 진입 확인 4. 브라우저 쿠키 삭제 후 `/dashboard` 재접근 → 다시 리다이렉트 확인

핵심 정리

- App Router에서 모든 컴포넌트는 **Server Component가 기본값**이며, 인터랙션이 필요한 경우에만 `"use client"`를 선언한다.
- API 엔드포인트는 `app/api/*/route.ts`에 HTTP 메서드별 함수를 export해서 정의한다.
- 비밀 환경변수(`ANTHROPIC_API_KEY` 등)는 Server Component·Route Handler에서만 사용하고, `NEXT_PUBLIC_` 없이는 브라우저에 노출되지 않는다.
- `middleware.ts`는 요청 최전선에서 실행되므로 인증 보호·리다이렉트 로직을 집중 관리하기에 최적의 위치다.
- `(auth)`, `(protected)` Route Group으로 URL 구조를 오염시키지 않고 레이아웃과 접근 제어를 분리할 수 있다.
- 이 모듈에서 잡은 디렉터리 구조(`app/api/`, `lib/`, `components/`)는 이후 RAG·Agent·음성 모듈이 의존하는 공통 뼈대다.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "Next.js 16 App Router + TypeScript + Tailwind 4로 프로젝트를 부트스트랩해줘. `app/ components/ lib/ public/` 구조로 잡아줘." "로그인 페이지와, `middleware.ts`로 비로그인 사용자를 막은 보호된 메인 페이지를 만들어줘."

팁 · 마음에 안 들면 전체를 다시 쓰지 말고 *차이만* 지시하세요 — "헤더를 sticky 글래스로 바꿔줘"처럼.

PRACTICE · 실습편

M7

LLM API 통합 (Claude / OpenAI)

callLLM() 하나로 통합한다

LLM은 AI Agent의 두뇌입니다. 어떤 프로바이더를 쓰든 호출 코드가 흩어지면 나중에 모델을 바꿀 때 전체 코드를 뜯어고쳐야 합니다. 이 모듈에서는 Claude와 OpenAI를 **단 하나의** `callLLM()` **함수**로 감싸고, Vercel AI Gateway를 통해 비용·레이턴시를 한눈에 관리하는 방법을 익힙니다.

패키지 설치

```
npm install @anthropic-ai/sdk openai
```

두 SDK를 모두 설치합니다. `@anthropic-ai/sdk`는 Claude 전용, `openai`는 GPT 계열뿐 아니라 Vercel AI Gateway의 OpenAI 호환 엔드포인트에도 사용합니다.

lib/llm.ts — callLLM() 통합 함수

`lib/llm.ts` 파일 하나에 두 프로바이더를 추상화합니다. 호출부는 `provider/model` 문자열만 바꾸면 됩니다.

```ts
// lib/llm.ts
import Anthropic from "@anthropic-ai/sdk";
import OpenAI from "openai";

export interface LLMMessage {
  role: "system" | "user" | "assistant";
  content: string;
}

export interface LLMOptions {
  model: string;          // "anthropic/claude-sonnet-4" | "openai/gpt-4o" 등
  messages: LLMMessage[];
  temperature?: number;
  maxTokens?: number;
  jsonMode?: boolean;
}

export async function callLLM(options: LLMOptions): Promise<string> {
  const { model, messages, temperature = 0.7, maxTokens = 1024, jsonMode = false } =
options;
  const [provider] = model.split("/");

  if (provider === "anthropic") {
    const client = new Anthropic({ apiKey: process.env.ANTHROPIC_API_KEY });
    const system = messages.find((m) => m.role === "system")?.content ?? "";
    const userMessages = messages
      .filter((m) => m.role !== "system")
      .map((m) => ({ role: m.role as "user" | "assistant", content: m.content }));

    const response = await client.messages.create({
      model: model.replace("anthropic/", ""),
      system,
      messages: userMessages,
      temperature,
      max_tokens: maxTokens,
    });
    return (response.content[0] as Anthropic.TextBlock).text;
  }

  // OpenAI / Vercel AI Gateway (OpenAI-compatible)
  const baseURL = process.env.VERCEL_AI_GATEWAY_URL; // 게이트웨이 사용 시
  const client = new OpenAI({ apiKey: process.env.OPENAI_API_KEY, ...(baseURL ? {
baseURL } : {}) });
  const response = await client.chat.completions.create({
    model: model.replace("openai/", ""),
    messages: messages as OpenAI.ChatCompletionMessageParam[],
    temperature,
    max_tokens: maxTokens,
    ...(jsonMode ? { response_format: { type: "json_object" } } : {}),
  });
```

```
  return response.choices[0].message.content ?? "";
}
```

> **핵심 설계 원칙**: 호출부는 `model` 문자열만 다르고, 나머지 코드는 동일합니다. 나중에 Gemini나 다른 프로바이더를 추가하고 싶다면 이 파일만 수정하면 됩니다.

System + User 프롬프트 작성 원칙

- **System 프롬프트**: 역할·어조·출력 형식 등 불변 지시사항. Claude에서는 별도 `system` 필드, OpenAI에서는 `role: "system"` 메시지.
- **User 프롬프트**: 실제 요청. 동적 값(이름, 문서 내용 등)은 여기에 삽입.
- 보고서·분석처럼 결정론적 결과가 필요할 때는 `temperature: 0.2`를 명시하세요.

JSON 강제 + extractJSON() 강건 파서

LLM이 JSON을 반환해야 할 때, OpenAI는 `jsonMode: true`로 강제할 수 있지만 Claude는 텍스트 안에 JSON을 감쌀 때가 있습니다. 두 경우 모두 안전하게 파싱하는 유틸을 만들어 둡니다.

```
// lib/llm.ts 에 추가
export function extractJSON<T = unknown>(raw: string): T {
  // 마크다운 코드펜스 제거
  const stripped = raw.replace(/^```(?:json)?\n?/i, "").replace(/\n?```$/i,
"").trim();
  try {
    return JSON.parse(stripped) as T;
  } catch {
    // JSON 블록만 추출 시도
    const match = stripped.match(/\{[\s\S]*\}|\[[\s\S]*\]/);
    if (match) return JSON.parse(match[0]) as T;
    throw new Error(`JSON 파싱 실패: ${raw.slice(0, 200)}`);
  }
}
```

사용 예시:

```
const raw = await callLLM({
  model: "anthropic/claude-sonnet-4",
  messages: [
    { role: "system", content: "반드시 JSON 객체로만 응답하세요. {summary: string, tags:
string[]}" },
    { role: "user", content: `다음 글을 6단으로 요약해줘:\n\n${article}` },
  ],
  temperature: 0.2,
  jsonMode: true,
});
const result = extractJSON<{ summary: string; tags: string[] }>(raw);
```

Vercel AI Gateway

Vercel AI Gateway는 Claude·OpenAI·Gemini 등 여러 프로바이더 호출을 단일 엔드포인트로 프록시합니다. 대시보드에서 **토큰 소비량·비용·레이턴시**를 모델별로 비교할 수 있습니다.

설정 항목	값 예시
`VERCEL_AI_GATEWAY_URL`	`https://gateway.ai.vercel.app/v1`
모델 문자열	`"anthropic/claude-sonnet-4"`
인증	Vercel 프로젝트 토큰 자동 주입

`callLLM()`에서 `baseURL`을 게이트웨이 URL로 넘기면 OpenAI SDK가 그대로 Claude를 포함한 모든 모델을 호출합니다. Claude를 게이트웨이 경유로 쓸 때는 `provider/model` 문자열을 OpenAI 호환 경로로 라우팅하므로, `anthropic/` 분기 대신 `openai/` 분기를 타도록 모델 문자열을 조정하거나 게이트웨이 전용 분기를 추가하세요.

> **비용 모니터링 팁**: Vercel 대시보드 → AI Gateway → Usage에서 모델별 일일 토큰을 확인하세요. 프로덕션 배포 전에 `max_tokens`를 용도에 맞게 제한해 두면 비용 스파이크를 예방할 수 있습니다.

실습 (Lab)

목표

Claude로 텍스트를 받아 6단 JSON 요약을 반환하는 API 라우트와, 간단한 채팅 UI를 만듭니다.

1단계 — 환경 변수 설정

`.env.local`에 추가합니다.

```
ANTHROPIC_API_KEY=sk-ant-...
OPENAI_API_KEY=sk-...
```

2단계 — API 라우트 작성

`app/api/chat/route.ts` 파일을 생성합니다.

```ts
import { NextRequest, NextResponse } from "next/server";
import { callLLM, extractJSON } from "@/lib/llm";

export async function POST(req: NextRequest) {
  const { text } = await req.json();
  if (!text) return NextResponse.json({ error: "text required" }, { status: 400 });

  const raw = await callLLM({
    model: "anthropic/claude-sonnet-4",
    messages: [
      {
        role: "system",
        content:
          "당신은 글 요약 전문가입니다. 반드시 JSON으로만 응답하세요.\n" +
          '형식: { "summary": "6단 요약 문장", "keywords": ["키워드1", "키워드2"] }',
      },
      { role: "user", content: `다음 글을 6단으로 요약하세요:\n\n${text}` },
    ],
    temperature: 0.2,
    jsonMode: true,
  });

  const result = extractJSON<{ summary: string; keywords: string[] }>(raw);
  return NextResponse.json(result);
}
```

3단계 — 간단한 채팅 UI 연결

`app/page.tsx`에서 `fetch("/api/chat", { method: "POST", body: JSON.stringify({ text }) })`를 호출하고, 응답의 `summary`와 `keywords`를 화면에 표시합니다. Tailwind로 `max-w-2xl mx-auto p-6` 컨테이너와 `<textarea>`, `<button>` 하나면 충분합니다.

4단계 — 동작 확인

`npm run dev` 후 텍스트를 입력하면 JSON 요약이 반환되는지 확인합니다. 응답이 정상이면 `model`을 `"openai/gpt-4o"`로 바꿔 동일하게 동작하는지도 검증하세요.

핵심 정리

- `callLLM()`은 `provider/model` 문자열로 Claude·OpenAI를 단일 인터페이스로 감싸며, 나머지 코드는 프로바이더에 무관합니다.
- `extractJSON()`은 마크다운 펜스와 불완전한 래핑을 제거한 뒤 파싱하므로 Claude와 OpenAI 모두 안전하게 처리합니다.
- 결정론적 결과(보고서·분류)가 필요할 때는 `temperature: 0.2`를 명시하세요.
- Vercel AI Gateway의 `provider/model` 문자열 하나로 모델을 전환하고, 대시보드에서 비용·레이턴시를 모니터링합니다.

- `jsonMode: true`는 OpenAI에서 `response_format: { type: "json_object" }`로 매핑되고, Claude에서는 System 프롬프트로 JSON 출력을 유도합니다.
- `max_tokens`를 용도에 맞게 제한해 두면 비용 스파이크와 응답 지연을 동시에 방지할 수 있습니다.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "`lib/llm.ts`에 `callLLM()`을 만들어줘 — Claude(@anthropic-ai/sdk) 우선, 키 없으면 OpenAI 폴백. system·messages·temperature·maxTokens 받게." "`app/api/chat/route.ts`에 그걸 쓰는 채팅 API와 간단한 UI를 붙여줘." "응답을 JSON 스키마로 강제하고, 깨진 JSON도 살리는 `extractJSON()` 헬퍼를 추가해줘."

팁 · "Vercel AI Gateway의 `\"anthropic/claude-sonnet-4\"` 문자열로 바꿔줘" 한 줄로 게이트웨이 전환.

PRACTICE · 실습편

M8

RAG 구축 — 임베딩·검색·환각 차단

in-memory 코사인 & 벡터 DB(pgvector)

RAG는 **선택이 아니라 필수**입니다. LLM은 학습 시점 이후를 모르고(Knowledge Cutoff), 모르는 걸 그럴듯하게 지어내니까(Hallucination) — 내 데이터를 근거로 박아 넣는 RAG 없이는 production 서비스가 성립하지 않습니다. 그리고 RAG에서 진짜 갈리는 결정은 딱 하나, **"벡터를 어디에 저장하고 어떻게 검색하느냐"** 입니다. 이 모듈은 두 가지 방식을 **모두** 구현하고, 언제 무엇을 쓸지까지 정리합니다.

RAG 파이프라인 — 백엔드와 무관한 공통 골격

저장 방식이 무엇이든 흐름은 같습니다.

```
문서 → 청킹(600자+overlap) → 임베딩(text-embedding-3-small, 1536d)
     → 저장(① 메모리/Blob  또는  ② 벡터 DB)
질문 → 임베딩 → 코사인 유사도 검색(Top-K) → 근거를 프롬프트에 주입 → LLM 답변 + 출처 인용
```

바뀌는 건 가운데 **저장·검색** 한 칸뿐입니다. 나머지(청킹·임베딩·근거 주입·환각 차단)는 완전히 동일하니, 한 번 만들면 두 방식에 그대로 재사용합니다.

```
// 두 방식이 공유하는 임베딩 헬퍼 (lib/embed.ts)
export async function embed(text: string): Promise<number[]> {
  const r = await fetch("https://api.openai.com/v1/embeddings", {
    method: "POST",
    headers: { "Content-Type": "application/json", Authorization: `Bearer
${process.env.OPENAI_API_KEY}` },
    body: JSON.stringify({ model: "text-embedding-3-small", input: text.slice(0,
8000) }),
  });
  return (await r.json()).data[0].embedding; // number[1536]
}
```

임계값 함정: `text-embedding-3-small` 은 한국어에서 코사인 유사도가 낮게 나옵니다. 관련 있는 매치도 0.2~0.4 수준이에요. threshold를 0.3 이상으로 잡으면 정답까지 다 걸러집니다. **0.15~0.2** 로 시작하세요. (이 강의 사이트도 이 값을 씁니다.)

방식 ① 벡터 DB 없이 — in-memory 코사인

임베딩을 JSON 파일(또는 Vercel Blob)에 저장하고, 요청 시 메모리에 올려 코사인을 직접 계산합니다. 인프라가 0이고, 핵심 코드는 15줄입니다.

```
// 코사인 유사도 — 라이브러리 없이 직접
export function cosine(a: number[], b: number[]): number {
  let dot = 0, na = 0, nb = 0;
  for (let i = 0; i < a.length; i++) { dot += a[i]*b[i]; na += a[i]*a[i]; nb +=
b[i]*b[i]; }
  return dot / (Math.sqrt(na) * Math.sqrt(nb) + 1e-8);
}

// 검색 — 빌드 때 만들어 둔 embeddings.json 을 메모리에서 정렬
import EMB from "@/content/embeddings.json"; // [{ text, embedding }]
export async function searchTopK(query: string, k = 5, threshold = 0.2) {
  const q = await embed(query);
  return EMB.map(r ⇒ ({ ...r, score: cosine(q, r.embedding) }))
    .filter(h ⇒ h.score ⩾ threshold)
    .sort((a, b) ⇒ b.score - a.score)
    .slice(0, k);
}
```

언제 쓰나 — 데이터가 **고정·소규모(대략 1만 청크 미만)** 일 때. 예: 이 사이트의 강사 Agent는 강의 노트 13개(146청크)를 이 방식으로 검색합니다. 별도 서비스·비용·운영 부담이 전혀 없고, Cold start 포함 수백 ms 안에 끝납니다.

방식 ② 벡터 DB — Supabase pgvector

데이터가 **크거나, 계속 쌓이거나, 영속·공유**가 필요하면 진짜 벡터 DB를 씁니다. Postgres 확장인 **pgvector**를 Supabase에서 켜면, SQL 한 번으로 벡터 검색 인프라가 생깁니다.

```
-- 1) 확장 + 테이블 + 인덱스
create extension if not exists vector;

create table documents (
  id bigint generated always as identity primary key,
  content text,
  embedding vector(1536),
  created_at timestamptz default now()
);
create index on documents using hnsw (embedding vector_cosine_ops); -- HNSW: 학습 불필요, 정확/빠름

-- 2) 코사인 검색 RPC (<=> 는 코사인 거리, 1 - 거리 = 유사도)
create or replace function match_documents(query_embedding vector(1536), match_count int, match_threshold float)
returns table (id bigint, content text, similarity float)
language sql stable as $$
  select id, content, 1 - (embedding <=> query_embedding) as similarity
  from documents
  where 1 - (embedding <=> query_embedding) >= match_threshold
  order by embedding <=> query_embedding
  limit match_count;
$$;
```

```
// 3) 앱에서 저장 / 검색 (supabase-js)
import { createClient } from "@supabase/supabase-js";
const sb = createClient(process.env.NEXT_PUBLIC_SUPABASE_URL!,
process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!);

// 저장: 임베딩은 '[0.1,0.2,...]' 텍스트 형태로 insert
await sb.from("documents").insert({ content, embedding: `[${vec.join(",")}]` });

// 검색: match_documents RPC 호출
const { data } = await sb.rpc("match_documents", {
  query_embedding: await embed(question), match_count: 5, match_threshold: 0.15,
});
```

언제 쓰나 — 데이터가 **수만~수백만 청크**, 또는 사용자가 계속 업로드해 양이 가변적일 때. 영속 저장·동시 접근·메타데이터 필터(카테고리·태그·날짜)가 필요할 때. 이 사이트의 **RAG Lab(`/lab`)** 이 정확히 이 방식입니다 — 학습자가 올린 문서를 pgvector에 넣고 검색해요.

케이스 스터디 — GHOSTSHIN (신해철 고스트스테이션)

GHOSTSHIN은 방식 ②가 **왜 필요한지**를 보여주는 실제 사례입니다.

- **규모**: 신해철의 고스트스테이션 라디오 **403개 방송 대본 ≈ 700만 자.** 청킹하면 수만~수십만 벡터. → 메모리에 다 올리는 건 비현실적, **벡터 DB가 필수.**
- **스택**: Supabase **pgvector** + `match_rag_documents` RPC. OpenAI 임베딩으로 인덱싱.

- **메타데이터 필터**: 단순 유사도만이 아니라 **카테고리 코드 · 방송 회차 · 태그(방송대본/Q&A) · priority(신해철 본인이 강조한 핵심 발언)** 로 필터링. RPC가 벡터 검색 결과와 우선 반영 자료를 합쳐 반환합니다.
- **페르소나 RAG**: 검색된 실제 발언을 근거로 "마왕(신해철)" 톤 답변을 생성 — 환각 없이 *그 사람이 실제로 한 말*에 뿌리내린 대화. (방송용 존댓말·청취자 호칭까지 RAG로 일관성 유지)

> **교훈**: 700만 자 코퍼스에 in-memory 코사인을 쓰면 메모리·레이턴시가 폭발합니다. 반대로 강의 노트 146청크에 pgvector를 붙이면 과한 인프라죠. **규모가 방식을 결정합니다.**

결정 가이드 — 둘 중 무엇을?

기준	방식 ① in-memory	방식 ② 벡터 DB(pgvector)
청크 규모	~1만 미만	1만 ~ 수백만+
데이터 갱신	고정 / 드묾	사용자 업로드 등 가변
인프라 · 비용	0 (파일 / Blob)	Supabase (무료 티어~)
영속 · 동시접근	약함	강함
메타데이터 필터	직접 구현	SQL `where` 로 자유
대표 사례	이 사이트 강사 Agent	RAG Lab · GHOSTSHIN

> **마이그레이션 팁**: 청크 스키마(`{ content, embedding, metadata }`)를 처음부터 동일하게 유지하면, ①로 시작했다가 데이터가 커질 때 ②로 옮기는 게 거의 복붙입니다.

환각 차단 — 두 방식 공통

저장 방식과 무관하게, production RAG의 신뢰도는 이 4가지에서 나옵니다.

1. **score 임계값**: 위 threshold(0.15~0.2)로 약한 매치를 버린다.
2. **출처 인용 의무화**: 시스템 프롬프트에 "근거를 `[1] [2]` 로 인용하라" 강제.
3. **"찾을 수 없으면 솔직히"**: 검색 결과가 비면 지어내지 말고 *"문서에서 답을 찾지 못했어요"*라고 답하게 한다.
4. **근거 밖 답변 금지**: "[검색 결과] 안의 내용만 사용하라"를 시스템 프롬프트 최상단에.

```
[규칙] 아래 [검색 결과]에 근거해서만 답한다. 문장 끝에 [n] 으로 출처를 인용한다.
검색 결과가 비었으면 "문서에서 답을 찾지 못했어요"라고만 답한다. 절대 지어내지 않는다.
```

평가 — Faithfulness 자체 측정

Ground Truth 질문 5건을 만들어, 답변이 (a) 검색된 근거에 충실한가(Faithfulness), (b) 질문과 관련 있는가(Relevance), (c) 정답 청크가 Top-K에 들어왔는가(Recall@K)를 사람이 채점합니다. 임계값·청크 사이즈·overlap을 바꿔가며 이 점수로 튜닝하세요.

실습 (Lab)

이 사이트가 **두 방식을 동시에 시연**합니다 — 직접 비교해 보세요.

1. **방식 ① (in-memory)** — 우측 하단 **강사 Agent**. 강의 노트 146청크를 메모리 코사인으로 검색해 답합니다.
2. **방식 ② (pgvector)** — `/lab` **RAG Lab**. 본인 노트/문서를 붙여넣어 Supabase pgvector에 임베딩 저장 → 질문 → 출처 인용 답변.
3. 같은 질문을 두 곳에 던져보고 score·출처·속도를 비교하면 "규모가 방식을 결정한다"가 몸으로 이해됩니다.

핵심 정리

- RAG는 필수. 진짜 결정은 **저장·검색 백엔드(① in-memory ② 벡터 DB)** 하나뿐이고, 나머지 파이프라인은 공유한다.
- **① in-memory 코사인**: ~1만 청크 미만·고정 데이터에 최적. 인프라 0, 15줄.
- **② Supabase pgvector**: 대규모·가변·영속에 필수. `vector` 확장 + HNSW 인덱스 + `match_documents` RPC.
- **GHOSTSHIN(700만 자, 403방송)** 은 ②가 필요한 규모의 실제 사례 — 메타데이터 필터·priority까지 활용.
- **규모가 방식을 결정한다.** 스키마를 통일해 두면 ①→② 마이그레이션은 거의 복붙.
- 한국어 임베딩은 유사도가 낮으니 **threshold 0.15~0.2**. 환각 차단 4종은 두 방식 공통.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "`lib/embed.ts` 에 text-embedding-3-small 임베딩 + 코사인 유사도(직접 구현)를 만들어줘." "노트를 600자 청킹·임베딩해 `content/embeddings.json` 으로 저장하는 스크립트와, 메모리 코사인 검색 `searchTopK()` 를 만들어줘. 한국어라 임계값 0.15~0.2." "대규모용으로 Supabase pgvector 버전도 — vector 확장·HNSW 인덱스·match 함수까지."

팁 · "검색 결과 없으면 '못 찾음'이라 답하고 출처 `[n]` 을 인용하게 시스템 프롬프트에 강제해줘" → 환각 차단.

PRACTICE · 실습편

M9

Vercel Blob 영속 저장소

DB 없이 KV 패턴으로 버틴다

지금까지 만든 AI Agent는 요청이 끝나면 데이터가 사라집니다. 사용자 파일, STT 원본 오디오, 프로젝트 메타데이터를 **영속적으로** 저장하려면 Vercel Blob이 필요합니다. 이 모듈에서는 Blob을 KV처럼 추상화하는 wrapper를 직접 만들고, 대용량 파일을 서버를 우회해 안전하게 올리는 방법까지 익힙니다.

@vercel/blob 설치 + Storage Dashboard

```
pnpm add @vercel/blob
```

Vercel Dashboard → Storage → Create Database → Blob을 생성하면 `BLOB_READ_WRITE_TOKEN` 환경변수가 자동으로 프로젝트에 연결됩니다. `.env.local`에도 동일한 값을 복사해 넣어야 로컬에서 동작합니다.

> **팁**: `vercel env pull .env.local` 한 줄로 모든 환경변수를 로컬에 내려받을 수 있습니다. 매번 복사할 필요가 없습니다.

Blob은 S3 호환 객체 스토리지입니다. 키는 경로처럼 생긴 문자열(`projects/abc/meta.json`)이고 값은 바이너리 또는 텍스트입니다. KV 데이터베이스가 아니기 때문에, JSON을 직렬화하고 역직렬화하는 얇은 wrapper가 필요합니다.

KV Wrapper 구현

아래 네 함수(`kvPutText`, `kvGetJson`, `kvList`, `kvDelete`)가 이 강의 전체에서 사용하는 표준 인터페이스입니다.

```ts
// lib/blob-kv.ts
import { put, get, list, del } from "@vercel/blob";

const TOKEN = process.env.BLOB_READ_WRITE_TOKEN!;

/** JSON 객체를 Blob에 저장 */
export async function kvPutText(key: string, value: unknown): Promise<string> {
  const body = JSON.stringify(value);
  const { url } = await put(key, body, {
    access: "public",
    token: TOKEN,
    contentType: "application/json",
    addRandomSuffix: false, // 키를 덮어써야 하므로 반드시 false
  });
  return url;
}

/** Blob에서 JSON 역직렬화 */
export async function kvGetJson<T = unknown>(key: string): Promise<T | null> {
  try {
    const res = await get(key, { token: TOKEN });
    if (!res) return null;
    const text = await res.text();
    return JSON.parse(text) as T;
  } catch {
    return null;
  }
}

/** prefix로 시작하는 키 목록 */
export async function kvList(prefix: string) {
  const { blobs } = await list({ prefix, token: TOKEN });
  return blobs;
}

/** 키 삭제 */
export async function kvDelete(url: string) {
  await del(url, { token: TOKEN });
}
```

주의: `addRandomSuffix: false`를 빠뜨리면 같은 키에 업데이트할 때마다 새 URL이 생겨 데이터가 중복 적재됩니다. 반드시 명시하세요.

Prefix 격리 전략

멀티테넌트 서비스에서는 데이터를 테넌트별로 격리해야 합니다. 이 강의는 경로 기반 prefix로 격리합니다.

용도	Blob 키 예시
프로젝트 메타데이터	`projects/<projectId>/meta.json`
STT 원본 오디오	`projects/<projectId>/audio/<uploadId>.webm`
RAG 청크 인덱스	`projects/<projectId>/rag/chunks.json`
보고서	`projects/<projectId>/reports/<date>.md`

`kvList("projects/abc123/")` 한 번으로 특정 프로젝트의 모든 파일을 열거할 수 있습니다. 서로 다른 프로젝트의 데이터는 prefix가 달라 자연스럽게 분리됩니다.

Direct Client Upload (1 GB 서버 우회)

Next.js API Route는 기본 요청 바디 한도가 4 MB입니다. 대용량 오디오나 PDF를 올릴 때 서버를 경유하면 이 한도에 걸립니다. Vercel Blob의 **client upload** 패턴은 서버가 토큰만 발급하고 클라이언트가 Blob에 직접 업로드합니다.

```
// app/api/upload-token/route.ts  ← 토큰 발급 엔드포인트
import { handleUpload, type HandleUploadBody } from "@vercel/blob/client";
import { NextRequest, NextResponse } from "next/server";

export async function POST(request: NextRequest) {
  const body = (await request.json()) as HandleUploadBody;

  try {
    const jsonResponse = await handleUpload({
      body,
      request,
      onBeforeGenerateToken: async (pathname) => ({
        allowedContentTypes: ["audio/webm", "audio/mp4", "application/pdf"],
        tokenPayload: JSON.stringify({ pathname }),
        maximumSizeInBytes: 1_000_000_000, // 1 GB
      }),
      onUploadCompleted: async ({ blob, tokenPayload }) => {
        // 업로드 완료 후 메타데이터를 KV에 기록
        const { pathname } = JSON.parse(tokenPayload ?? "{}");
        const metaKey = pathname.replace(/\.[^.]+$/, "_meta.json");
        await kvPutText(metaKey, { url: blob.url, uploadedAt: new
Date().toISOString() });
      },
    });
    return NextResponse.json(jsonResponse);
  } catch (error) {
    return NextResponse.json({ error: (error as Error).message }, { status: 400 });
  }
}
```

클라이언트에서는 `@vercel/blob/client`의 `upload()` 함수를 호출합니다. 파일이 서버 메모리를 전혀 거치지 않고 Blob에 직접 전송됩니다.

자동 삭제: STT 완료 후 원본 오디오 제거

원본 오디오는 STT 처리 직후 삭제하는 것이 좋은 관행입니다. 비용 절감과 개인정보 최소화를 동시에 달성합니다.

```
// STT 처리 후 호출
async function transcribeAndClean(audioUrl: string): Promise<string> {
  // ElevenLabs Scribe v2 STT 호출 (강의 M8 참고)
  const transcript = await callScribeV2(audioUrl);

  // 원본 오디오 즉시 삭제
  await kvDelete(audioUrl);

  return transcript;
}
```

`kvDelete`는 Blob URL을 직접 받으므로, 업로드 완료 콜백에서 받은 `blob.url`을 그대로 넘기면 됩니다.

실습 (Lab)

목표: PDF 또는 오디오 파일을 direct upload로 올리고, 메타데이터를 KV에 영구 저장한 뒤 목록을 조회합니다.

단계별 안내

1. **환경 준비** - `vercel env pull .env.local`로 `BLOB_READ_WRITE_TOKEN` 확인 - `pnpm add @vercel/blob` 설치
2. **KV wrapper 작성** - `lib/blob-kv.ts`에 위의 `kvPutText`, `kvGetJson`, `kvList`, `kvDelete` 복사
3. **업로드 토큰 API 작성** - `app/api/upload-token/route.ts` 생성 - `onUploadCompleted`에서 `projects/<projectId>/audio/<filename>_meta.json` 키로 메타데이터 저장
4. **클라이언트 업로드 UI 작성** - `<input type="file" accept="audio/*,application/pdf" />` - `@vercel/blob/client`의 `upload(pathname, file, { handleUploadUrl: "/api/upload-token" })` 호출
5. **목록 조회 API 작성** - `GET /api/files?projectId=xxx` → `kvList("projects/xxx/")` 결과 반환
6. **자동 삭제 검증** - 업로드된 오디오 URL로 `transcribeAndClean()` 호출 후 Vercel Dashboard에서 파일이 사라졌는지 확인

> **체크포인트**: 업로드 완료 후 `kvList`가 메타데이터 JSON을 반환하면 성공입니다. Blob 원본 오디오는 삭제되고 메타데이터만 남아 있어야 합니다.

핵심 정리

- `@vercel/blob`은 S3 호환 객체 스토리지이며, JSON 직렬화 wrapper(`kvPutText` / `kvGetJson` / `kvList` / `kvDelete`)로 KV처럼 사용한다.
- `addRandomSuffix: false` 옵션을 반드시 설정해야 같은 키 덮어쓰기(upsert)가 정상 동작한다.
- Prefix 경로(`projects/<id>/...`)로 테넌트 또는 프로젝트 단위의 데이터 격리를 구현한다.
- Direct client upload는 서버를 우회하므로 4 MB 한도 없이 최대 1 GB까지 업로드 가능하다.
- STT 완료 직후 원본 오디오를 `kvDelete`로 삭제하면 비용 절감과 개인정보 최소화를 동시에 달성한다.
- 모든 Blob 작업은 `BLOB_READ_WRITE_TOKEN` 하나로 인증되며, 이 토큰은 절대 클라이언트 코드에 노출해선 안 된다.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "`@vercel/blob`로 KV처럼 쓰는 wrapper(`kvPutText·kvGetJson·kvList·kvDelete`)를 만들어줘. `projects/<id>/` prefix로 격리." "1GB 파일도 되게 클라이언트 direct upload로 해줘."

팁 · "STT 끝나면 원본 음성 자동 삭제"처럼 후처리도 한 문장으로 지시.

PRACTICE · 실습편

M10

보고서 생성 에이전트

6단 표준 구조 + 결정적 조립

LLM이 보고서를 "그냥 써주길" 기대하면 매번 다른 구조, 다른 어투, 다른 섹션 순서가 나옵니다. M10에서는 **LLM은 데이터에서 의미를 추출하는 역할만** 맡기고, 실제 문서 구조는 TypeScript 헬퍼가 결정론적으로 조립하는 패턴을 익힙니다.

왜 "결정적 조립"인가

LLM의 창의성은 보고서에서 오히려 노이즈입니다. 같은 회의록을 5번 넣어도 섹션 이름이 바뀌거나 항목 순서가 달라지면 자동화 파이프라인이 깨집니다. 해법은 **역할 분리**입니다.

역할	담당	특성
의미 추출	Claude (temperature 0.2)	확률적이지만 제약으로 수렴
문서 조립	TypeScript 헬퍼 함수	완전 결정적
저장	Vercel Blob	KV처럼 사용

temperature를 0.2로 고정하는 것만으로는 부족합니다. **JSON Schema를 시스템프롬프트에 명시**해 LLM 출력 자체를 구조화해야 합니다.

6단 보고서 구조

강의 전체에서 쓰는 표준 보고서 구조는 다음과 같습니다.

1. **주요사항(Highlights)** — 오늘의 핵심 성과 3개 이내
2. **일정(Schedule)** — 완료/예정 항목 목록
3. **요약(Summary)** — 전체 맥락 1~2문장
4. **블로킹(Blockers)** — 진행을 막는 이슈
5. **액션(Actions)** — 담당자·기한 포함 할 일
6. **Flow** — 오늘의 컨디션/집중도 지표 (1~5 점수)

이 순서는 코드에서 하드코딩됩니다. LLM이 순서를 바꿀 여지가 없습니다.

시스템프롬프트 설계

```
const REPORT_SYSTEM_PROMPT = `
당신은 일간 활동 보고서 추출기입니다.
사용자의 원문(회의록·메모·로그)에서 구조화 데이터를 추출하여
반드시 아래 JSON Schema를 따르는 JSON 객체만 반환하세요.
설명, 마크다운 펜스, 여분의 텍스트는 절대 포함하지 마세요.

Schema:
{
  "highlights": string[],      // 최대 3개
  "schedule": { "done": string[], "upcoming": string[] },
  "summary": string,           // 1~2문장
  "blockers": string[],        // 없으면 빈 배열
  "actions": { "task": string, "owner": string, "due": string }[],
  "flow": number               // 1(최저)~5(최고) 정수
}
`;
```

중요: `반드시 JSON만 반환` 지시를 해도 LLM이 가끔 마크다운 펜스를 붙입니다. 파싱 전에 `text.replace(/^```json\n?/, '').replace(/\n?```$/, '')` 로 방어하세요.

LLM 호출 — callLLM()과 Vercel AI Gateway

강의 전체에서 통일된 `callLLM()` 래퍼를 사용합니다. 보고서 추출은 `claude-sonnet-4-5`를 쓰되 Vercel AI Gateway의 `provider/model` 문자열로 라우팅합니다.

```ts
// lib/report-agent.ts
import { callLLM } from '@/lib/llm';

export interface ReportData {
  highlights: string[];
  schedule: { done: string[]; upcoming: string[] };
  summary: string;
  blockers: string[];
  actions: { task: string; owner: string; due: string }[];
  flow: number;
}

export async function extractReportData(rawText: string): Promise<ReportData> {
  const raw = await callLLM({
    model: 'anthropic/claude-sonnet-4-5', // Vercel AI Gateway 형식
    system: REPORT_SYSTEM_PROMPT,
    messages: [{ role: 'user', content: rawText }],
    temperature: 0.2,
    max_tokens: 1024,
  });

  const cleaned = raw.trim()
    .replace(/^```json\n?/, '')
    .replace(/\n?```$/, '');

  return JSON.parse(cleaned) as ReportData;
}
```

결정적 Markdown 조립

추출된 데이터를 받아 Markdown을 조립하는 함수는 **순수 함수(pure function)** 로 작성합니다. LLM을 호출하지 않으며, 같은 입력에는 항상 같은 출력이 나옵니다.

```
// lib/report-builder.ts
import type { ReportData } from './report-agent';

export function buildMarkdownReport(data: ReportData, date: string): string {
  const flowBar = '■'.repeat(data.flow) + '□'.repeat(5 - data.flow);

  const sections: string[] = [
    `## 📋 일간 보고서 — ${date}`,
    `### 🌟 주요사항\n${data.highlights.map(h => `- ${h}`).join('\n')}`,
    `### 📅 일정\n**완료**\n${data.schedule.done.map(d => `- [x]
${d}`).join('\n')}\n\n**예정**\n${data.schedule.upcoming.map(u => `- [ ]
${u}`).join('\n')}`,
    `### 📝 요약\n${data.summary}`,
    `### 🚧 블로킹\n${data.blockers.length ? data.blockers.map(b => `- ⚠️
${b}`).join('\n') : '- 없음'}`,
    `### ✅ 액션 아이템\n| 할 일 | 담당 | 기한 |\n|---|---|---|\n${data.actions.map(a => `|
${a.task} | ${a.owner} | ${a.due} |`).join('\n')}`,
    `### 💡 Flow\n\`${flowBar}\` ${data.flow}/5`,
  ];

  return sections.join('\n\n');
}
```

이 헬퍼가 6단 구조의 순서, 이모지, 표 형식을 **모두 소유**합니다. LLM 출력의 변동성은 `extractReportData`의 JSON 경계에서 차단됩니다.

일관성 측정 — 5회 반복 테스트

temperature 0.2가 실제로 일관성을 보장하는지 확인하려면 같은 입력으로 5회 호출해 필드별 분산을 측정합니다.

```
// scripts/consistency-check.ts
async function measureConsistency(rawText: string, runs = 5) {
  const results = await Promise.all(
    Array.from({ length: runs }, () => extractReportData(rawText))
  );

  const flowValues = results.map(r => r.flow);
  const highlightCounts = results.map(r => r.highlights.length);

  console.table({
    'flow 분산': Math.max(...flowValues) - Math.min(...flowValues),
    'highlights 개수 분산': Math.max(...highlightCounts) - Math.min(...highlightCounts),
    'blockers 없음 비율': results.filter(r => r.blockers.length === 0).length / runs,
  });
}
```

팁: temperature 0.2에서도 `flow` 점수가 ±1 흔들릴 수 있습니다. 수용 범위를 정해두고 테스트를 자동화하면 모델 버전 업그레이드 시 회귀를 잡을 수 있습니다.

실습 (Lab)

목표: 하루치 활동 메모를 붙여넣으면 표준 6단 보고서 Markdown을 자동 생성하고 Vercel Blob에 저장하는 API 라우트를 만든다.

단계별 안내

Step 1 — 프로젝트 준비

```
app/
  api/
    report/
      route.ts    ← POST: 원문 수신 → 추출 → 조립 → Blob 저장
lib/
  report-agent.ts
  report-builder.ts
  llm.ts          ← 기존 callLLM() 재사용
```

Step 2 — 환경변수 설정 - `ANTHROPIC_API_KEY` (또는 Vercel AI Gateway 토큰) - `BLOB_READ_WRITE_TOKEN` (Vercel Blob)

Step 3 — API 라우트 구현 `app/api/report/route.ts`에서 `{ rawText, date, userId }`를 받아 `extractReportData` → `buildMarkdownReport` → `put(blob)` 순서로 연결합니다. Blob 키는 `reports/${userId}/${date}.md` 패턴으로 고정해 날짜별 덮어쓰기가 되도록 합니다.

Step 4 — 5회 일관성 테스트 `scripts/consistency-check.ts`를 실행해 `flow` 분산이 1 이내, `highlights` 개수 분산이 0인지 확인합니다. 실패하면 시스템프롬프트의 제약 문구를 강화하세요.

Step 5 — (선택) 간단한 UI `app/report/page.tsx`에 textarea + 버튼을 두고 API를 호출한 뒤 결과 Markdown을 `react-markdown`으로 렌더링합니다.

핵심 정리

- **LLM은 추출만, 조립은 코드가** — 역할을 분리해야 보고서 구조가 일관된다.
- **temperature 0.2 + JSON Schema** 두 가지가 함께 있어야 LLM 출력을 예측 가능하게 만든다.
- **결정적 Markdown 헬퍼**는 순수 함수로 작성해 단위 테스트가 가능하게 한다.
- **Vercel AI Gateway** `provider/model` **문자열**로 Claude와 OpenAI를 동일 인터페이스로 교체할 수 있다.
- **5회 반복 일관성 테스트**를 CI에 포함시키면 모델 업그레이드 시 회귀를 자동 감지할 수 있다.
- **Vercel Blob을 KV처럼** 사용할 때 키 패턴을 규칙적으로 설계해야 조회·덮어쓰기가 예측 가능하다.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "6단 구조(주요사항·일정·요약·블로킹·액션·Flow) 보고서 생성기를 만들어줘. LLM은 추출만(JSON), Markdown 조립은 결정적 헬퍼가, temperature 0.2." "같은 입력으로 5번 호출해 일관성을 확인하는 테스트도 만들어줘."

팁 · 페르소나·제약을 시스템 프롬프트에 명시하라고 함께 지시.

PRACTICE · 실습편

M11

STT / TTS 통합 (ElevenLabs)

내 목소리로 보고하는 에이전트

음성 인터페이스는 AI Agent 서비스의 접근성을 한 단계 끌어올립니다. 이 모듈에서는 ElevenLabs의 Scribe v2 STT와 eleven_v3 TTS를 조합해 **음성 → 텍스트 → 내 목소리 음성**으로 돌아오는 풀 파이프라인을 직접 구현합니다.

ElevenLabs Scribe v2 STT

Scribe v2는 ElevenLabs가 제공하는 REST 기반 STT 모델로, 화자분리(diarization)와 단어 수준 타임스탬프를 지원합니다. SDK 없이 `fetch`로 직접 호출합니다.

```
// app/api/stt/route.ts
export async function POST(req: Request) {
  const formData = await req.formData();
  const audioFile = formData.get("audio") as File;

  const body = new FormData();
  body.append("audio", audioFile);
  body.append("model_id", "scribe_v2");
  body.append("diarize", "true");          // 화자분리 활성화
  body.append("timestamps_granularity", "word"); // 단어별 타임스탬프

  const res = await fetch("https://api.elevenlabs.io/v1/speech-to-text", {
    method: "POST",
    headers: { "xi-api-key": process.env.ELEVENLABS_API_KEY! },
    body,
  });

  if (!res.ok) throw new Error(`STT failed: ${res.status}`);
  const data = await res.json();

  // data.words: [{ text, start, end, speaker_id }]
  return Response.json({ transcript: data.text, words: data.words });
}
```

응답의 `words` 배열에는 `speaker_id`(화자 구분)와 `start` / `end`(초 단위 타임스탬프)가 함께 담깁니다. 회의록 자동 생성이나 다자 대화 분석에 바로 활용할 수 있습니다.

cloud_storage_url 패턴 — 서버 우회

오디오 파일이 크면(예: 30분 녹음) 서버를 경유하는 것은 1GB 제한과 Cold Start 타임아웃 위험을 함께 안고 갑니다. ElevenLabs는 `audio_url` 파라미터로 Vercel Blob의 public URL을 직접 넘기는 패턴을 지원합니다.

```
// 클라이언트가 먼저 Vercel Blob에 업로드 → URL만 API로 전달
const blobUrl = await uploadToBlob(audioFile); // Vercel Blob 퍼블릭 URL

const body = new FormData();
body.append("model_id", "scribe_v2");
body.append("audio_url", blobUrl);  // 파일 대신 URL
body.append("diarize", "true");

const res = await fetch("https://api.elevenlabs.io/v1/speech-to-text", {
  method: "POST",
  headers: { "xi-api-key": process.env.ELEVENLABS_API_KEY! },
  body,
});
```

이 패턴을 쓰면 Next.js API Route는 파일 바이트를 전혀 다루지 않아도 됩니다. **파일 → Blob → URL → ElevenLabs** 흐름이 핵심입니다.

> **주의**: Vercel Blob URL은 기본적으로 공개입니다. 민감한 음성 데이터라면 처리 완료 후 즉시 삭제(`del`)하거나 signed URL 패턴을 검토하세요.

eleven_v3 TTS와 음성 클로닝

TTS 엔드포인트는 `/v1/text-to-speech/{voice_id}` 입니다. `optimize_streaming_latency=3`은 지연 시간과 품질의 균형점으로, 실시간 응답이 필요한 Agent 서비스에서 권장 설정입니다.

음성 클로닝 절차: ElevenLabs 대시보드(또는 Instant Voice Clone API)에 30초 이상의 깨끗한 샘플을 올리면 `voice_id`가 발급됩니다. 이 ID를 환경변수(`MY_VOICE_ID`)로 관리합니다.

```
// app/api/tts/route.ts
export async function POST(req: Request) {
  const { text } = await req.json();

  const res = await fetch(
    `https://api.elevenlabs.io/v1/text-to-speech/${process.env.MY_VOICE_ID}` +
      `?optimize_streaming_latency=3`,
    {
      method: "POST",
      headers: {
        "xi-api-key": process.env.ELEVENLABS_API_KEY!,
        "Content-Type": "application/json",
      },
      body: JSON.stringify({
        text,
        model_id: "eleven_v3",
        voice_settings: { stability: 0.5, similarity_boost: 0.75 },
      }),
    }
  );

  if (!res.ok) throw new Error(`TTS failed: ${res.status}`);

  // 오디오 스트림을 그대로 클라이언트로 전달
  return new Response(res.body, {
    headers: { "Content-Type": "audio/mpeg" },
  });
}
```

클라이언트에서는 응답 Blob을 `HTMLAudioElement`에 물려 자동재생합니다.

```
// components/AudioPlayer.tsx (일부)
const playTTS = async (text: string) => {
  const res = await fetch("/api/tts", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ text }),
  });
  const blob = await res.blob();
  const url = URL.createObjectURL(blob);
  const audio = new Audio(url);
  audio.play();                          // 사용자 제스처 후 호출 필요
};
```

브라우저 정책: `audio.play()`는 사용자 인터랙션(버튼 클릭 등) 이후에만 허용됩니다. 페이지 로드 직후 자동재생은 대부분의 브라우저에서 차단됩니다.

모델 비교

항목	Scribe v2 (STT)	eleven_v3 (TTS)
방식	REST POST (multipart or URL)	REST POST → 오디오 스트림
지연	파일 길이에 비례	`latency=3` 기준 ~400ms
주요 기능	화자분리, 단어 타임스탬프	감정 표현, 클로닝 지원
요금 단위	오디오 분(minute)	문자 수(character)

실습 (Lab)

목표: 음성 메모를 녹음하고, STT로 텍스트화한 뒤, 본인 보이스로 요약 음성을 재생하는 엔드-투-엔드 흐름을 완성합니다.

단계 1 — 음성 클로닝 준비

1. ElevenLabs 대시보드 → "Voices" → "Add a new voice" → "Instant Voice Cloning" 선택.
2. 30초 이상 잡음 없이 본인 목소리로 녹음한 파일을 업로드.
3. 생성된 `voice_id`를 복사해 `.env.local`에 `MY_VOICE_ID=...`로 저장.

단계 2 — 녹음 UI 구현

`MediaRecorder API`로 브라우저 마이크 입력을 Blob으로 수집합니다. 녹음 완료 시 `/api/stt`로 POST합니다.

단계 3 — STT 라우트 연결

위 `app/api/stt/route.ts` 코드를 프로젝트에 추가합니다. 파일이 1MB 초과이면 먼저 Vercel Blob에 업로드하고 `cloud_storage_url` 패턴으로 전환합니다.

단계 4 — 요약 → TTS 재생

STT 결과 텍스트를 `callLLM()`으로 넘겨 3줄 요약을 생성(`temperature: 0.2`)한 뒤, `/api/tts`로 보내 오디오를 받아 `HTMLAudioElement`로 재생합니다.

단계 5 — 화자분리 결과 표시

`words` 배열을 파싱해 `speaker_id`별로 색을 달리한 트랜스크립트 UI를 렌더링합니다(예: `speaker_0` → 파란색, `speaker_1` → 주황색).

검증 기준: 녹음 버튼 클릭 → 말하기 → 중지 → 텍스트 표시 → "요약 재생" 클릭 → 본인 목소리로 요약 재생. 이 흐름이 에러 없이 완료되면 실습 성공입니다.

핵심 정리

- Scribe v2 STT는 SDK 없이 `fetch`로 직접 호출하며, `diarize=true`로 화자분리, `timestamps_granularity=word`로 단어별 타임스탬프를 얻는다.
- 대용량 오디오는 Vercel Blob에 먼저 올린 뒤 `audio_url`로 넘기는 **cloud_storage_url 패턴**으로 서버 부하를 완전히 우회한다.

- eleven_v3 TTS는 `optimize_streaming_latency=3`으로 호출해 응답 스트림을 `Content-Type: audio/mpeg`로 클라이언트에 바로 전달한다.
- 음성 클로닝은 30초 샘플만으로 가능하며, 발급된 `voice_id`를 환경변수로 관리한다.
- `HTMLAudioElement.play()`는 반드시 사용자 인터랙션 이후에 호출해야 브라우저 자동재생 정책을 통과한다.
- STT 결과를 `callLLM()`에 넘길 때는 `temperature: 0.2`를 유지해 결정적 요약을 생성한다.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "ElevenLabs Scribe로 STT 라우트(`app/api/stt`), eleven 모델로 TTS 라우트(`app/api/tts`)를 만들어줘. 보이스 ID는 env로." "마이크 녹음 → STT → 자동 전송 → 답변 자동 음성재생되는 위젯으로 만들어줘." "30초 샘플로 음성 클로닝해 voice ID를 보관하는 흐름도."

팁 · "`optimize_streaming_latency=3`으로 지연 줄여줘".

PRACTICE · 실습편

M12

메일 발송 + 공유 페이지

토큰 기반 공유 + 후속 Q&A

보고서가 화면에 뜨는 것과, 받은 사람이 그 보고서를 바탕으로 후속 질문까지 할 수 있는 것은 UX 품질의 차이가 크다. 이 모듈에서는 Resend로 HTML 메일을 발송하고, 128비트 공유 토큰 기반의 비로그인 공유 페이지에서 후속 Q&A가 가능한 흐름을 완성한다.

Resend SDK로 메일 발송하기

이 강의는 메일 발송에 **Resend**를 사용한다. nodemailer는 SMTP 자격증명 관리가 번거롭고 Vercel Edge에서 동작이 불안정하기 때문이다.

항목	nodemailer	Resend SDK
인증 방식	SMTP 자격증명	API Key (env var 1개)
Vercel 호환	불안정 (Node.js 전용)	Edge/Node 모두 지원
HTML 조립	수동	`html` + `text` 필드로 분리
발송 확인	직접 구현	응답 객체에 `id` 포함

설치 후 가장 기본적인 발송 코드는 아래와 같다.

```ts
// lib/mail.ts
import { Resend } from "resend";

const resend = new Resend(process.env.RESEND_API_KEY!);

export async function sendReportMail({
  to,
  subject,
  htmlBody,
  textBody,
}: {
  to: string;
  subject: string;
  htmlBody: string;
  textBody: string;
}) {
  const { data, error } = await resend.emails.send({
    from: "Hans17 Academy <noreply@aicrmeet.com>",
    to,
    subject,
    html: htmlBody,
    text: textBody, // 멀티파트: HTML 미지원 클라이언트용
  });
  if (error) throw new Error(`Resend error: ${error.message}`);
  return data?.id;
}
```

`html`과 `text`를 동시에 넣으면 Resend가 자동으로 `multipart/alternative`로 패키징한다. 스팸 필터 통과율을 높이기 위해 `text` 필드는 절대 생략하지 않는다.

HTML 본문 + 자동 footer 조립

보고서 본문은 이미 Markdown으로 생성되어 있다. 이것을 HTML로 변환한 뒤, **공유 URL**과 **후속 질의 URL**이 담긴 footer를 덧붙인다.

```
// lib/buildReportHtml.ts
import { marked } from "marked";

export function buildReportHtml({
  markdownBody,
  shareToken,
  baseUrl,
}: {
  markdownBody: string;
  shareToken: string;
  baseUrl: string;
}) {
  const htmlBody = marked.parse(markdownBody);   // Markdown → HTML
  const shareUrl = `${baseUrl}/share/${shareToken}`;
  const askUrl   = `${baseUrl}/share/${shareToken}/ask`;

  const footer = `
    <hr style="margin:32px 0;border:none;border-top:1px solid #e5e7eb"/>
    <p style="font-size:13px;color:#6b7280">
      이 보고서를 공유하려면:
      <a href="${shareUrl}">${shareUrl}</a><br/>
      후속 질문하기:
      <a href="${askUrl}">${askUrl}</a>
    </p>`;

  return `<!DOCTYPE html><html><body>
    <div style="max-width:720px;margin:auto;font-family:sans-serif">
      ${htmlBody}
      ${footer}
    </div>
  </body></html>`;
}
```

`text` 버전도 같은 URL을 plain text로 추가하면 된다.

공유 토큰 생성 (128비트)

공유 링크는 **예측 불가능한 토큰**이 핵심이다. `Math.random()` 은 절대 사용하지 않는다. Web Crypto API의 `crypto.getRandomValues` 로 16바이트(128비트) 난수를 생성한다.

```
// lib/shareToken.ts
export function generateShareToken(): string {
  const bytes = new Uint8Array(16);
  crypto.getRandomValues(bytes);                    // Web Crypto — Node/Edge 모두 지원
  return Array.from(bytes)
    .map((b) => b.toString(16).padStart(2, "0"))
    .join(""); // 32자 hex 문자열
}
```

생성된 토큰은 Vercel Blob에 보고서 메타데이터와 함께 저장한다. 키 형식은 `reports/{token}.json`을 권장한다.

> **보안 주의**: 토큰을 URL에 노출하더라도 128비트 무작위성 덕분에 브루트포스는 현실적으로 불가능하다. 단, 토큰을 DB에 저장할 때 만료 시각(`expiresAt`)을 반드시 포함하고, 공유 페이지 서버 코드에서 만료 여부를 먼저 확인한다.

비로그인 공유 페이지 + 후속 Q&A

`/share/[token]/page.tsx`는 로그인 미들웨어를 거치지 않는다. `token`으로 Blob에서 보고서를 읽어 렌더링만 하면 된다.

`/share/[token]/ask/route.ts`는 POST를 받아 원본 보고서를 컨텍스트로 삼아 Claude에 질의한다. 이때 Agent dispatcher를 따로 태우지 않고, **단일 LLM 호출**로 간단하게 처리한다. 원본 보고서가 이미 충분한 컨텍스트이기 때문이다.

실습 (Lab)

목표: 보고서를 메일로 발송하고, 수신자가 공유 링크에서 후속 질의까지 완료한다.

Step 1 — 환경 변수 세팅

```
RESEND_API_KEY=re_xxxx
NEXT_PUBLIC_BASE_URL=https://your-domain.vercel.app
```

Step 2 — 토큰 생성 및 Blob 저장

`generateShareToken()`으로 토큰을 만들고, `reports/{token}.json`에 `{ reportMarkdown, createdAt, expiresAt }` 형태로 `put()`한다.

Step 3 — 메일 발송 API Route 작성

`app/api/report/send/route.ts`를 POST로 작성. 요청 바디에서 `to`, `reportId`를 받아 Blob에서 Markdown을 꺼낸 뒤 `buildReportHtml()`과 `sendReportMail()`을 순서대로 호출한다.

Step 4 — 공유 페이지 라우트 추가

`app/share/[token]/page.tsx` : Blob에서 보고서 로드 → 만료 검사 → Markdown 렌더링.
`app/share/[token]/ask/route.ts` : POST body의 `question`과 보고서 Markdown을 system prompt에 넣어 Claude 호출. streaming 응답 권장.

Step 5 — 수동 E2E 테스트

1. 로컬에서 Send API를 `curl`로 호출해 메일 수신 확인.
2. 메일 내 공유 링크를 **시크릿 모드** 브라우저(비로그인 상태)에서 열어 보고서가 뜨는지 확인.
3. Ask 폼에 질문 입력 → 스트리밍 응답이 정상 출력되는지 확인.
4. `expiresAt`을 과거 시각으로 바꿔 Blob을 수동 수정한 뒤, 만료 처리가 올바르게 동작하는지 검증.

> **팁**: Resend 무료 플랜은 하루 100통 제한이 있다. 개발 중에는 실제 메일 발송 대신 `console.log(htmlBody)`로

확인하고, 최종 E2E 때만 실발송하면 발송 한도를 아낄 수 있다.

핵심 정리

- Resend SDK는 `html` + `text` 멀티파트를 한 번의 호출로 처리하며, Vercel Edge에서도 안정적으로 동작한다.
- 메일 footer에 `shareUrl`과 `askUrl`을 자동 삽입해 수신자가 추가 행동을 바로 취할 수 있게 한다.
- 공유 토큰은 반드시 `crypto.getRandomValues(16 bytes)`로 생성해 128비트 예측 불가능성을 확보한다.
- 보고서는 Vercel Blob에 `reports/{token}.json`으로 저장하고 `expiresAt` 필드를 포함한다.
- `/share/[token]`은 로그인 없이 접근 가능하며, `/share/[token]/ask`는 원본 보고서를 컨텍스트로 단일 LLM 호출로 후속 Q&A를 처리한다.
- 만료 토큰 접근 시 404 또는 안내 페이지로 명확히 처리해야 보안 사고를 예방할 수 있다.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "Resend로 HTML+text 멀티파트 메일 발송을 만들고, footer에 share·ask URL을 자동 합성해줘."
> "`crypto.getRandomValues(16바이트)` 토큰으로 비로그인 공유 페이지(`/share/[token]`)와 후속 Q&A 라우트를 만들어줘."

팁 · 파일을 *첨부*하려면 서버 발송(Resend), 본문·링크만이면 `mailto`.

PRACTICE · 실습편

M13

배포 · 운영 · 확장

본인 도메인으로 production 배포

코드를 로컬에서 완벽하게 동작시켰다면, 이제 그것을 세상에 내보내야 합니다. M13은 Vercel CLI를 중심으로 배포 파이프라인 전체—환경 변수 동기, 커스텀 도메인, 로그 관측, AI Gateway 폴백, 드레인 연동—를 한 흐름으로 익히는 모듈입니다.

Vercel CLI 설치 및 프로젝트 연결

```
npm i -g vercel@latest
vercel login          # GitHub/GitLab OAuth 또는 이메일
vercel link           # 현재 디렉터리를 Vercel 프로젝트에 연결
```

`vercel link`를 실행하면 `.vercel/project.json`이 생성됩니다. 팀 프로젝트라면 `--scope <team-slug>` 옵션을 함께 넘기세요. 이 파일은 `.gitignore`에 추가하지 않아도 되지만, 민감 정보는 없으니 커밋해도 무방합니다.

환경 변수 동기: `vercel env pull`

Vercel 대시보드에 등록된 환경 변수를 로컬 `.env.local`로 내려받습니다.

```
vercel env pull .env.local
```

이 명령은 **Development** 환경의 변수를 가져옵니다. CI/CD에서 Preview·Production 변수가 필요하다면 `--environment preview` 또는 `--environment production` 플래그를 사용하세요.

> **중요**: `ANTHROPIC_API_KEY`, `OPENAI_API_KEY`, `ELEVENLABS_API_KEY` 같은 시크릿은 반드시 Vercel 대시보드에서 **Encrypted** 타입으로 등록하고, `.env.local`은 `.gitignore`에 추가해 두세요. `vercel env pull` 결과물을 커밋하는 실수가 가장 흔한 보안 사고입니다.

강의의 `callLLM()` 통합 함수는 `VERCEL_AI_GATEWAY_URL` 같은 게이트웨이 엔드포인트도 환경 변수로 관리합니다.

```
// lib/llm.ts — callLLM() 핵심 발췌
export async function callLLM(
  model: string,        // "anthropic/claude-sonnet-4-5" | "openai/gpt-4o" 형식
  messages: CoreMessage[],
  options?: { temperature?: number }
) {
  const baseURL = process.env.VERCEL_AI_GATEWAY_URL;    // 게이트웨이 엔드포인트
  const apiKey  = process.env.AI_GATEWAY_TOKEN;

  const response = await fetch(`${baseURL}/${model}`, {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      Authorization: `Bearer ${apiKey}`,
    },
    body: JSON.stringify({
      messages,
      temperature: options?.temperature ?? 0.7,
    }),
  });

  if (!response.ok) throw new Error(`LLM call failed: ${response.statusText}`);
  return response.json();
}
```

`"provider/model"` 문자열 한 줄만 바꾸면 Anthropic ↔ OpenAI를 전환할 수 있는 구조입니다.

Production 배포: `vercel --prod`

```
vercel --prod
```

Git을 통한 자동 배포(Vercel ↔ GitHub 연동)도 가능하지만, 첫 실습에서는 CLI 흐름을 직접 체험하는 것이 좋습니다. 배포 URL은 `https://<project>.vercel.app` 형식으로 즉시 발급됩니다.

> Preview 배포(`vercel` 명령만 실행)는 PR 단위 검토에, `--prod` 는 실제 트래픽에 붙는 슬롯입니다. 두 환경의 환경 변수가 다를 수 있으므로 대시보드에서 반드시 확인하세요.

커스텀 도메인 연결

대시보드 **Settings → Domains**에서 도메인을 추가하면 Vercel이 CNAME/A 레코드를 안내합니다. DNS TTL이 전파될 때까지(보통 수 분~1시간) 기다린 뒤 `vercel domains inspect <yourdomain.com>` 으로 상태를 확인합니다.

레코드 타입	호스트	값
CNAME	`www`	`cname.vercel-dns.com`
A	`@`	`76.76.21.21`

HTTPS 인증서는 Vercel이 자동으로 발급·갱신합니다. 별도 설정은 불필요합니다.

Functions Logs와 AI Gateway 관측

배포 후 런타임 오류나 LLM 응답을 추적하려면 **Functions Logs**를 활용합니다.

```
vercel logs <deployment-url> --follow
```

AI Gateway를 경유한 요청은 대시보드 **AI Gateway → Observability** 탭에서 모델별 토큰 사용량, 레이턴시, 오류율을 확인할 수 있습니다. Fallback 설정은 대시보드 또는 `vercel.json`에서 지정합니다.

```
// vercel.json — AI Gateway fallback 예시
{
  "aiGateway": {
    "routes": [
      {
        "path": "anthropic/claude-sonnet-4-5",
        "fallback": ["openai/gpt-4o"]
      }
    ]
  }
}
```

Claude가 429(Rate Limit)나 5xx를 반환하면 자동으로 GPT-4o로 폴백됩니다. `callLLM()`은 동일 인터페이스이므로 애플리케이션 코드 변경 없이 이 혜택을 누립니다.

Drains: Sentry / Datadog 연동

Vercel **Log Drains**는 Functions·Edge의 모든 로그를 외부 서비스로 스트리밍합니다.

- **Sentry**: 대시보드 **Integrations → Sentry** 설치 후 `SENTRY_DSN` 환경 변수 등록. 오류는 자동으로 이슈화됩니다.
- **Datadog**: **Integrations → Datadog** 설치 또는 HTTP Drain(Endpoint URL + API Key)으로 수동 연결.

Log Drain은 **Settings → Log Drains**에서 추가하며, JSON 또는 NDJSON 포맷을 선택할 수 있습니다.

실습 (Lab)

이 Lab의 목표는 본인 도메인으로 AI Agent 서비스를 Production 배포하고, URL과 화면을 팀에 공유하는 것입니다.

Step 1. CLI 설치 및 프로젝트 연결

```
npm i -g vercel@latest && vercel login && vercel link
```

Step 2. 환경 변수 확인

```
vercel env pull .env.local
# ANTHROPIC_API_KEY, OPENAI_API_KEY, AI_GATEWAY_TOKEN 등 누락 없는지 확인
```

Step 3. Production 배포

```
vercel --prod
# 출력된 배포 URL을 메모
```

Step 4. 커스텀 도메인 연결 - Vercel 대시보드 → Settings → Domains → 본인 도메인 입력 - DNS 레코드를 도메인 레지스트라에 적용 - `vercel domains inspect <yourdomain.com>` 으로 연결 확인

Step 5. 로그 확인 및 AI Gateway 관측

```
vercel logs https://<yourdomain.com> --follow
```

- 대시보드 AI Gateway 탭에서 토큰 사용량 스크린샷 촬영

Step 6. URL 공유 및 시연 - Slack/채널에 `https://<yourdomain.com>` 공유 - 에이전트 기능(RAG 질의, 음성 입력, 보고서 생성 중 택 1) 실시간 시연

> 시연 중 에러가 발생하면 `vercel logs --follow` 를 열어두고 함께 디버깅하세요. 배포 환경에서만 재현되는 버그는 환경 변수 누락이 원인인 경우가 80%입니다.

핵심 정리

- `vercel link` → `vercel env pull` → `vercel --prod` 세 단계가 기본 배포 흐름이다.
- 시크릿은 Vercel 대시보드에 Encrypted로 등록하고 절대 커밋하지 않는다.
- AI Gateway의 `"provider/model"` 라우팅과 Fallback 설정으로 `callLLM()` 은 코드 변경 없이 멀티 모델 폴백을 지원한다.
- Functions Logs(`vercel logs --follow`)와 AI Gateway Observability로 프로덕션 트래픽을 실시간 추적한다.
- Log Drains(Sentry/Datadog)을 연결해 오류 알림과 장기 메트릭을 외부 관측 도구로 분리한다.
- 운영 체크리스트: 환경 변수 암호화, Rate Limit 모니터링, 월별 AI 비용 상한선(Vercel Spend Management) 설정.

🤖 Claude Code 프롬프트 (이렇게 시키면 됩니다)

이 모듈은 손코딩이 아니라 **Claude Code에 프롬프트로 지시**해서 만듭니다.

> "Vercel에 배포해줘 — `vercel link` 로 연결하고, `.env.local` 키를 `vercel env` 로 올린 뒤 `vercel --prod` ." "커스텀 도메인을 이 프로젝트에 연결하고 www는 apex로 308 리다이렉트해줘." "배포 후 라이브 URL로 페

> 이지·API가 200인지 확인해줘."

팁 · **Vercel MCP**를 연결하면 Claude가 배포·로그·도메인을 *직접* 처리합니다.