GILLILAB · TECH

# ChatGPT·Claude 로 쓰고 코드로 찍는 전자책

AI 원고 집필부터 EPUB·PDF 조판 자동화까지

gillilab

TECH SERIES

# Contents

**1 주제 하나에서 전자책 두 판형까지** **1**
1.1 수작업 조판이 무너지는 세 지점 . . . 1
1.2 이 책의 원칙: diagram-as-code . . . 2
1.3 만들 물건 . . . 2
1.4 도구 선택 . . . 4
1.5 누구를 위한 책인가 . . . 4
1.6 이 책을 읽는 법 . . . 5
1.7 이 책이 다루지 않는 것 . . . 5

**2 도구 체인 준비** **6**
2.1 무엇을 왜 까는가 . . . 6
2.2 폰트: 이름 하나가 전부를 정한다 . . . 7
2.3 점검 스크립트 . . . 9
2.4 프로젝트 뼈대 . . . 10
2.5 환경 변수 . . . 11
2.6 점검 . . . 11

# 1

# 주제 하나에서 전자책 두 판형까지

머릿속에 책 한 권 분량의 주제가 있다. 몇 년째 다뤄 온 기술이든, 반복해서 설명해 온 노하우든, 강의로 여러 번 풀어 본 내용이든. 그런데 그것이 책이 되지 못하는 이유는 대개 두 곳에서 막히기 때문이다.

**앞에서는 원고가 안 나온다.** 목차까지는 어떻게 잡는데, 12 장짜리 초고를 끝까지 쓰는 데 몇 달이 걸리고 대개 4 장쯤에서 멈춘다. **뒤에서는 파일이 안 나온다.** 원고를 다 썼다 해도 그 마크다운 더미를 팔 수 있는 EPUB·PDF 로 바꾸는 구간이 남아 있고, 여기서 그림과 판형과 한글 조판이 차례로 무너진다.

이 책은 그 양쪽을 모두 자동화한다. **주제 한 줄에서 시작해 AI 와 함께 목차를 잡고 초고를 쓰고, 그 원고 폴더를 입력으로 받아 EPUB 과 PDF 를 동시에 뽑아내는 로컬 파이프라인**을 실제로 동작하는 코드와 함께 처음부터 끝까지 만든다. 지금 읽고 있는 이 책 자체가 그 파이프라인으로 만들어졌다.

## 1.1 수작업 조판이 무너지는 세 지점

직접 해 보면 순서대로 무너진다.

**첫째, 그림이다.** 기술서에는 구조도와 흐름도가 들어간다. 이걸 그림판이나 Figma 로 그려서 PNG 로 붙이면, 원고를 고칠 때마다 그림을 다시 열어야 한다. 3 장의 다이어그램에서 상자 하나의 이름을 바꾸는 데 도구를 켜고 편집하고 내보내고 파일을 갈아끼우는 네 단계가 든다. 개정판을 낼 때쯤이면 원본 파일이 어디 갔는지도 모른다.

**둘째, 두 판형이다.** EPUB 은 글자 크기가 독자 손에서 바뀌는 리플로우 판형이고, PDF 는 페이지가 고정된 판형이다. 요구가 정반대다. EPUB 에서 폭 100% 로 예쁘게 걸린 다이어그램이 PDF 에서는 페이지를 넘겨 버리고, PDF 에서 벡터로 선명한 그림이 구형 킨들에서는 아예 안 뜬다. 두 판형을 각각 손으로 맞추면 작업량이 두 배가 아니라, **고칠 때마다 두 배**가 된다.

**셋째, 한글이다.** 한글 전자책은 영문 전자책과 실패하는 지점이 다르다. 폰트를 임베드하지 않으면 리더마다 다른 글꼴로 뜬다. 코드 블록의 긴 줄은 종이에서 잘린 채 인쇄된다—가로 스크롤이 없는 매체에서는 복구 불가능한 결함이다. 표는 폭이 제

각각으로 놓이고, LaTeX 은 한글 조판을 위해 별도의 설정을 요구한다.

세 가지 모두"한 번 잘 맞춰 놓으면 되는"문제처럼 보인다. 실제로는 원고를 고칠 때마다 다시 무너지는 문제다. 그래서 답은 조판을 잘하는 것이 아니라, **조판을 코드로 만드는 것**이다.

## 1.2 이 책의 원칙: diagram-as-code

이 파이프라인의 설계 원칙은 하나다.

> **핵심**
>
> 그림을 이미지로 **생성**하지 않고, 텍스트 코드로 **다룬다**.

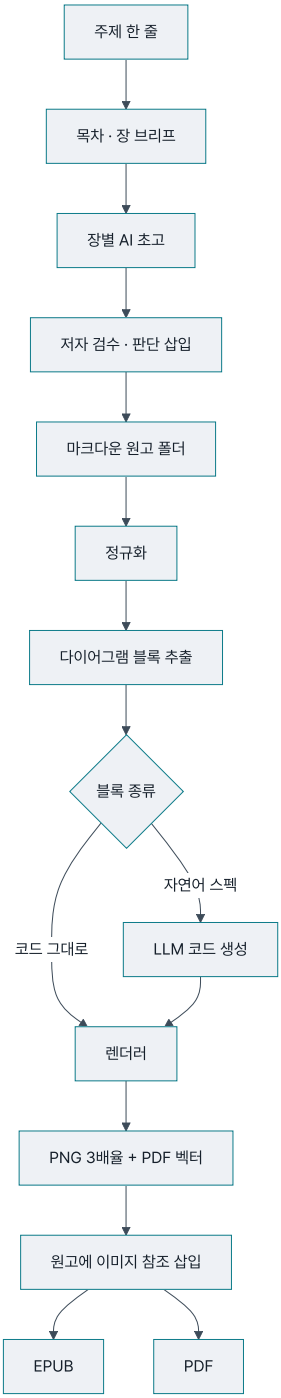
원고 안에 Mermaid·D2·Vega-Lite 코드를 그대로 써 두고, 빌드 시점에 렌더러가 이미지로 바꾼다. 이 결정 하나에서 나머지가 따라온다.

- **원고와 함께 버전 관리된다.** 다이어그램의 변경 이력이 `git diff` 에 텍스트로 남는다.
- **재현된다.** 원고를 고치면 그림도 같이 다시 만들어진다. 낡은 PNG 가 남아 있을 자리가 없다.
- **벡터로 인쇄된다.** PDF 에 벡터로 들어가므로 확대해도 깨지지 않는다.
- **한글 폰트를 정확히 지정할 수 있다.** 렌더 시점에 폰트를 주입하므로 글꼴이 리더에 따라 흔들리지 않는다.

여기에 한 겹을 더 얹는다. 다이어그램 코드를 **직접 쓰지 않아도 되게** 하는 것이다. 자연어로"오케스트레이터가 렌더러에 코드를 보내고, 실패하면 에러를 LLM 에 넘겨 고친다"라고 쓰면 LLM 이 Mermaid 코드를 만들어 준다. 그리고 그 코드가 문법 오류로 렌더에 실패하면, 파서 에러를 다시 LLM 에 넣어 **스스로 고치게** 한다. 이 자기교정 루프가 이 파이프라인에서 가장 재미있는 부분이고, 10 장에서 통째로 다룬다.

## 1.3 만들 물건

전체 흐름은 이렇게 생겼다. 왼쪽 절반이 AI 와 함께 원고를 만드는 구간이고, 오른쪽 절반이 그 원고를 파일로 굽는 구간이다.




오른쪽 절반은 명령 하나로 돈다.

```
uv run python -m pipeline.run --chapters manuscripts/my-book --title " 책 제목"
```

산출물은 `build/` 아래에 떨어진다. EPUB 은 폰트가 임베드되고 표지가 붙은 완성본이고, PDF 는 속표지·장 오프너·러닝 헤더·목차가 들어간 조판본이다. 둘 다 손댈 곳이 없는 상태로 나온다.

## 1.4 도구 선택

이 파이프라인은 **전부 로컬에서 돈다.** Docker 도, 워크플로 엔진도, SaaS 조판 서비스도 쓰지 않는다. 파이썬 오케스트레이터가 로컬 CLI 들을 부르는 구조다.

| 역할 | 도구 | 왜 이걸 골랐나 |
|---|---|---|
| 조판 | `pandoc` | 마크다운에서 EPUB·LaTeX·HTML 로 가는 가장 짧은 경로 |
| PDF 엔진 | `xelatex` | 한글 조판에 필요한 `xeCJK` 를 쓸 수 있는 사실상 유일한 선택 |
| 다이어그램 | `mmdc`·`d2` | 텍스트 문법이 안정적이고 CLI 로 렌더된다 |
| 차트 | `vl-convert` | 브라우저 없이 Vega-Lite 를 벡터로 렌더한다 |
| SVG 변환 | `rsvg-convert` | D2 의 SVG 를 PNG·PDF 로 바꾼다 |
| 검증 | `epubcheck` | EPUB 표준 위반을 잡는다 |

로컬을 고집하는 이유는 취향이 아니다. 조판은 **되돌아가는 작업**이다. 표 하나를 고치고 다시 빌드하는 일을 하루에 수십 번 한다. 원격 빌드로는 이 반복 속도가 나오지 않는다. 여기서 만든 흐름을 나중에 CI 나 워크플로 엔진으로 옮기는 건 어렵지 않지만, 순서는 반드시 로컬이 먼저다.

## 1.5 누구를 위한 책인가

**IT 를 어느 정도 아는 사람**이면 충분하다. 구체적으로는 이렇다.

| | 내용 |
|---|---|
| **필요한 것** | 터미널에서 명령을 실행할 수 있다. 마크다운으로 글을 써 봤다. |
| **있으면 좋은 것** | 파이썬 코드를 읽을 수 있다 (쓸 필요는 없다). |
| **필요 없는 것** | LaTeX, EPUB 내부 구조, 조판 용어, 디자인 경험 |

본문의 코드는 전부 **그대로 복사해 쓸 수 있는 완성된 형태**다. 원리가 궁금하지 않다면 붙여 넣고 넘어가도 책은 나온다. 새 용어는 처음 나올 때 한 문장으로 설명하고 지나간다.

## 1.6 이 책을 읽는 법

**3 장까지 읽으면 이미 전자책 파일이 손에 있다.** 코드 없이 도구만으로 만드는 투박한 버전이고, 거기서부터 하나씩 제대로 만든다.

| 부 | 장 | 무엇을 얻나 |
|---|---|---|
| 준비 | 1~3 장 | 도구 설치, 그리고 **첫 EPUB·PDF** |
| 원고 | 4~6 장 | AI 로 목차·초고를 만들고 규약에 맞춰 정리한다 |
| 그림 | 7~12 장 | 다이어그램 자동 생성·자기교정, 이미지 조달 판단 |
| 조판 | 13~15 장 | 디자인 토큰, EPUB·PDF 조판과 한글 함정 |
| 마무리 | 16~19 장 | 여러 권 배치, 미리보기·목차, 검증 절차 |

읽는 방식은 둘 중 하나를 고르면 된다.

- **처음부터 순서대로** —각 장이 앞 장의 결과 위에 쌓인다. 다 읽으면 빈 폴더에서 완성된 파이프라인까지 간다.
- **3 장까지 읽고 필요할 때 펴 보기** —조판 함정 장 (14~15 장) 과 검증 장 (18 장) 은 문제가 생겼을 때 찾아보는 참조 문서로도 쓰인다.

**참고**

코드는 파이썬 3.13 기준이고 패키지 관리는 `uv` 를 쓴다. 설명은 macOS 환경을 전제하지만, 도구 설치 명령만 바꾸면 리눅스에서도 그대로 동작한다.

## 1.7 이 책이 다루지 않는 것

- **글을 잘 쓰는 법.** AI 와 함께 원고를 **만드는 절차**는 다루지만 (4~5 장), 문장력이나 서술 기법 자체는 범위 밖이다.
- **일러스트레이션과 브랜딩.** 코드로 표지를 만들고 (13 장) AI 이미지 생성 선택지를 정리하지만 (12 장), 그래픽 디자인 자체는 다루지 않는다.
- **유통과 판매.** 만든 파일을 어디에 올려 팔지는 플랫폼마다 다르다. 다만 어디에 올리든 필요한 미리보기와 목차 데이터는 17 장에서 만든다.

**주의**

자동화는 검수를 대체하지 않는다. 18 장에서 만드는 검증 절차—EPUB 표준 검사, PDF 페이지 이미지 확인, 리더 렌더 재현—는 선택 사항이 아니다. 파이프라인이 조용히 잘못된 결과를 낼 수 있는 지점들이 있고, 그게 어디인지는 각 장에서 표시해 둔다.

준비가 됐으면 도구부터 깔자.

2

# 도구 체인 준비

파이프라인은 파이썬 코드가 로컬 CLI 들을 부르는 구조다. 그래서 첫 작업은 그 CLI 들을 깔고, **깔렸는지 기계가 확인해 주는 장치**를 만드는 것이다. 두 번째가 더 중요하다. 이 파이프라인의 실패 대부분은 코드 버그가 아니라"그 도구가 이 머신에 없다"이기 때문이다.

## 2.1 무엇을 왜 까는가

| 도구 | 용도 | 설치 |
|---|---|---|
| `uv` | 파이썬 환경·실행 | `brew install uv` |
| `pandoc` | EPUB·PDF·HTML 조판 | `brew install pandoc` |
| `xelatex` | PDF 엔진 (한글) | `brew install --cask mactex-no-gui` |
| `mmdc` | Mermaid 렌더 | `npm i -g @mermaid-js/mermaid-cli` |
| `d2` | D2 렌더 | `brew install d2` |
| `rsvg-convert` | SVG → PNG·PDF | `brew install librsvg` |
| `epubcheck` | EPUB 표준 검증 | `brew install epubcheck` |

각 도구가 파이프라인의 어느 자리에 붙는지 보면 목록이 덜 임의적으로 보인다.

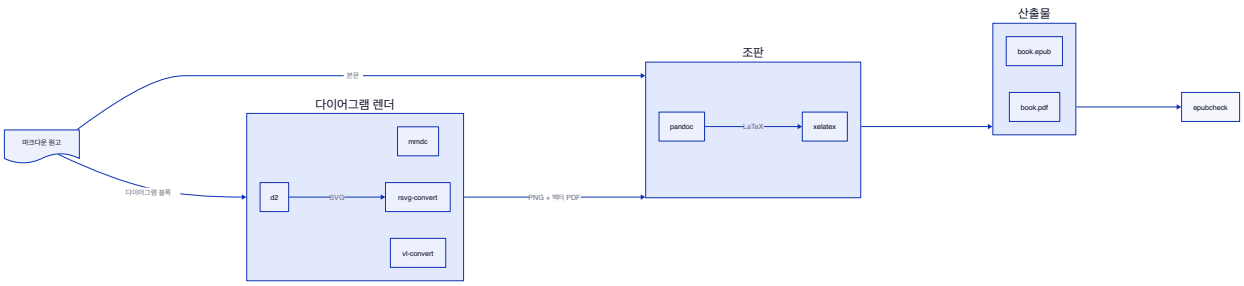



차트 렌더러 (`vl-convert-python`) 는 파이썬 패키지라 `uv sync` 로 따라 들어온다. 브라우저가 필요 없는 Vega-Lite 구현이라는 점이 중요하다—다른 선택지들은 헤드리스 크롬을 요구하고, 그러면 빌드가 느려지고 CI 로 옮기기도 어려워진다.

MacTeX 전체는 5GB 가 넘는다. `mactex-no-gui` 는 GUI 앱을 뺀 배포판이고, 우리가 쓰는 건 `xelatex` 바이너리와 한글 패키지뿐이다. TeX Live 를 쓴다면 최소 설치 후 필요한 패키지만 `tlmgr` 로 받아도 된다.

**주의**

`pdflatex` 가 아니라 **`xelatex`** 여야 한다. 한글 조판에 필요한 `xeCJK`, 그리고 뒤에서 표지에 쓸 `fontspec` 의 자간 조절 기능이 xelatex 전용이다. 엔진을 바꾸면 조판이 통째로 깨진다.

## 2.2 폰트: 이름 하나가 전부를 정한다

본문 폰트는 **Pretendard** 로 간다. 한글 기술서에서 화면과 종이 양쪽에 무리 없이 걸리는 무료 폰트이고, 웨이트가 충분히 나뉘어 있어 제목과 본문의 위계를 만들 수 있다.

```
brew install --cask font-pretendard
```

여기서 이 파이프라인 전체를 관통하는 제약이 하나 생긴다. **폰트를 정하는 곳은 단 하나여야 한다.** 폰트 이름은 EPUB 의 CSS, 표지 SVG, PDF 의 `mainfont` 와 `CJKmainfont`, Mermaid 테마, Vega-Lite 설정까지 여섯 군데로 퍼진다. 이걸 각자 하드코딩하면 어느 하나를 바꿨을 때 나머지 다섯이 조용히 어긋난다. 13 장에서 만드는 디자인 토큰이 이 문제의 답이다.

### 2.2.1 D2 만 TTF 를 요구한다

D2 에는 별도의 함정이 있다. D2 의 `--font-regular` 옵션은 **깨끗한 TTF 만 받는다.** OTF 도 TTC 도 거부하고, 거부할 때 나오는 메시지는 `expected .ttf` 뿐이라 원인을 알기 어렵다.

문제는 Homebrew 의 `font-pretendard` cask 가 `~/Library/Fonts` 에는 **OTF 만** 심는다는 점이다. 그대로 넘기면 실패한다. TTF 는 Caskroom 안에 남아 있으므로 그걸 프로젝트로 복사해 오는 스크립트를 만든다.

```
#!/usr/bin/env bash
# D2 한글용 TTF 준비. D2 는 '깨끗한 TTF'만 받는다 (OTF/TTC 거부).
set -euo pipefail
```

```
cd "$(dirname "$0")/.."

PRETENDARD_VER="v1.3.9"
CDN="https://cdn.jsdelivr.net/gh/orioncactus/pretendard@${PRETENDARD_VER}/packages/pr
  ↪ etendard/dist/public/static/alternative"

mkdir -p fonts

fetch_weight() {
  local weight="$1" dest="$2"
  [ -f "$dest" ] && { echo " 이미 있음: $dest"; return 0; }

  # Caskroom 은 버전 디렉터리가 바뀌므로 glob 으로 가장 최근 것을 고른다.
  local cask
  cask=$(find /opt/homebrew/Caskroom/font-pretendard
    ↪ /usr/local/Caskroom/font-pretendard \
         -path "*/public/static/alternative/Pretendard-${weight}.ttf" 2>/dev/null \
       | sort | tail -1 || true)
  if [ -n "$cask" ]; then
    cp "$cask" "$dest"; echo "Caskroom 에서 복사: $dest"; return 0
  fi

  curl -fSL -o "$dest" "${CDN}/Pretendard-${weight}.ttf"
}

fetch_weight Regular fonts/Pretendard.ttf
fetch_weight Bold    fonts/Pretendard-Bold.ttf
```

조달 순서가 세 단계인 게 핵심이다. 이미 있으면 건너뛰고, Caskroom 에 있으면 복사하고 (오프라인에서도 된다), 없으면 CDN 에서 개별 파일만 받는다. 배포 zip 은 55MB 라 쓰지 않는다.

**참고**

`find` 는 존재하지 않는 경로를 만나면 종료 코드 1 을 낸다. `set -e` 아래에서는 이것만으로 스크립트가 죽으므로 `|| true` 가 필요하다. Intel 맥과 Apple Silicon 의 Homebrew 접두사가 달라 두 경로를 모두 뒤지는 것도 같은 이유다.

## 2.3 점검 스크립트

도구가 하나라도 빠지면 파이프라인은 빌드 중간에 죽는다. 그것도 pandoc 스택 트레이스나 puppeteer 에러 같은, 원인과 거리가 먼 메시지로. 그래서 **빌드 전에 한 번에 확인하는** 스크립트를 만든다.

```
#!/usr/bin/env bash
# 로컬 도구 점검 —파이프라인이 돌 수 있는지 확인
set -u
ok=0; miss=0
check() {
  printf "  %-12s" "$1"
  if command -v "$1" >/dev/null 2>&1; then
    echo "OK ($($1 --version 2>&1 | head -1))"; ok=$((ok+1))
  else
    echo "MISSING — $2"; miss=$((miss+1))
  fi
}

echo "[필수]"
check uv      "brew install uv"
check mmdc    "npm i -g @mermaid-js/mermaid-cli"
check pandoc  "brew install pandoc"
check xelatex "brew install --cask mactex-no-gui"

echo "[선택 · D2 엔진]"
check d2           "brew install d2"
check rsvg-convert "brew install librsvg"
printf "  %-12s" "d2 한글 TTF"
[ -f fonts/Pretendard.ttf ] && echo "OK" || echo "MISSING (bash
  ↪ scripts/prepare_d2_font.sh)"

echo "[LaTeX 한글 패키지]"
for sty in kotex xeCJK; do
  printf "  %-12s" "$sty"
  kpsewhich "$sty.sty" >/dev/null 2>&1 && echo "OK" || echo "MISSING (tlmgr install
    ↪ $sty)"
done
```

```
echo "[한글 폰트]"
for name in "Pretendard"; do
  printf "  %-20s" "$name"
  fc-list 2>/dev/null | grep -qi "$name" && echo "OK" || echo "MISSING"
done

echo "---"; echo " 필수 OK=$ok  MISSING=$miss"
```

세 층으로 나눈 게 의도다.

- **실행 파일**은 `command -v` 로 본다. 있으면 버전까지 찍어 준다—버전 차이로 생기는 문제가 실제로 있다 (11 장의 D2 사례).
- **LaTeX 패키지**는 실행 파일이 아니라 `.sty` 파일이라 `kpsewhich` 로 찾아야 한다. xelatex 이 깔려 있어도 `kotex` 이 없으면 한글 PDF 는 안 나온다.
- **폰트**는 파일 경로가 아니라 **fontconfig 가 이름으로 해석할 수 있는지**를 본다. LaTeX 과 CSS 가 폰트를 파일이 아니라 이름으로 부르기 때문이다. `~/Library/Fonts` 에 파일이 있어도 fontconfig 캐시에 안 잡혔으면 조판에서 못 쓴다.

**팁**

점검 스크립트에 `set -e` 를 넣지 않는다. 이 스크립트의 목적은 **빠진 것을 전부 모아서 한 번에 보여 주는 것**이다. 첫 번째 누락에서 죽으면 설치를 한 번에 하나씩 하게 되고, 그게 가장 짜증나는 실패 방식이다. 같은 원칙이 뒤에서 원고 검증에도 그대로 적용된다.

## 2.4 프로젝트 뼈대

```
pipeline/           오케스트레이터 파이썬 패키지
  filters/          pandoc lua 필터
book/
  metadata.yaml     제목·저자·언어
  chapters/*.md     원고
manuscripts/        다권 배치용 원고 루트
assets/diagrams/    렌더된 다이어그램 캐시
build/              산출물
fonts/              D2 용 TTF
scripts/            doctor.sh · prepare_d2_font.sh
tokens.yaml         디자인 토큰
```

파이썬 환경은 `uv` 로 만든다.

```
uv init
uv add pyyaml openai vl-convert-python
uv add --dev pytest
```

의존성이 이게 전부다. 조판도 렌더도 전부 외부 CLI 가 하기 때문에, 파이썬 쪽은 YAML 을 읽고 LLM 을 부르고 서브프로세스를 돌리는 얇은 층이다. 이 얇음이 의도된 설계다—무거운 일은 각 분야에서 이미 검증된 도구에 맡기고, 우리 코드는 그것들을 **어떤 순서로 어떤 인자로** 부를지만 안다.

## 2.5 환경 변수

LLM 코드 생성 (9 장) 에 OpenAI 키가 필요하다. 저장소 루트의 `.env` 에 둔다.

```
OPENAI_API_KEY=sk-...
```

**주의**

키가 없어도 파이프라인은 돈다—자연어 스펙 블록만 건너뛰고 경고 노트로 대체한다. 이건 의도된 동작이다. 키 하나 때문에 원고 전체가 빌드되지 않으면 파이프라인으로서 쓸모가 없다. 이"한 부분이 실패해도 전체는 계속된다"는 원칙은 뒤에서 여러 번 다시 나온다.

## 2.6 점검

```
bash scripts/doctor.sh
```

전부 OK 가 뜨면 준비가 끝났다. 하나라도 MISSING 이면 옆에 적힌 설치 명령을 실행하고 다시 돌린다. 이 스크립트를 통과하지 못한 상태에서 빌드를 시도하면, 원인이 도구 누락인 실패를 코드 버그로 착각하며 시간을 쓰게 된다.

다음 장부터 실제 코드를 쓴다.

---