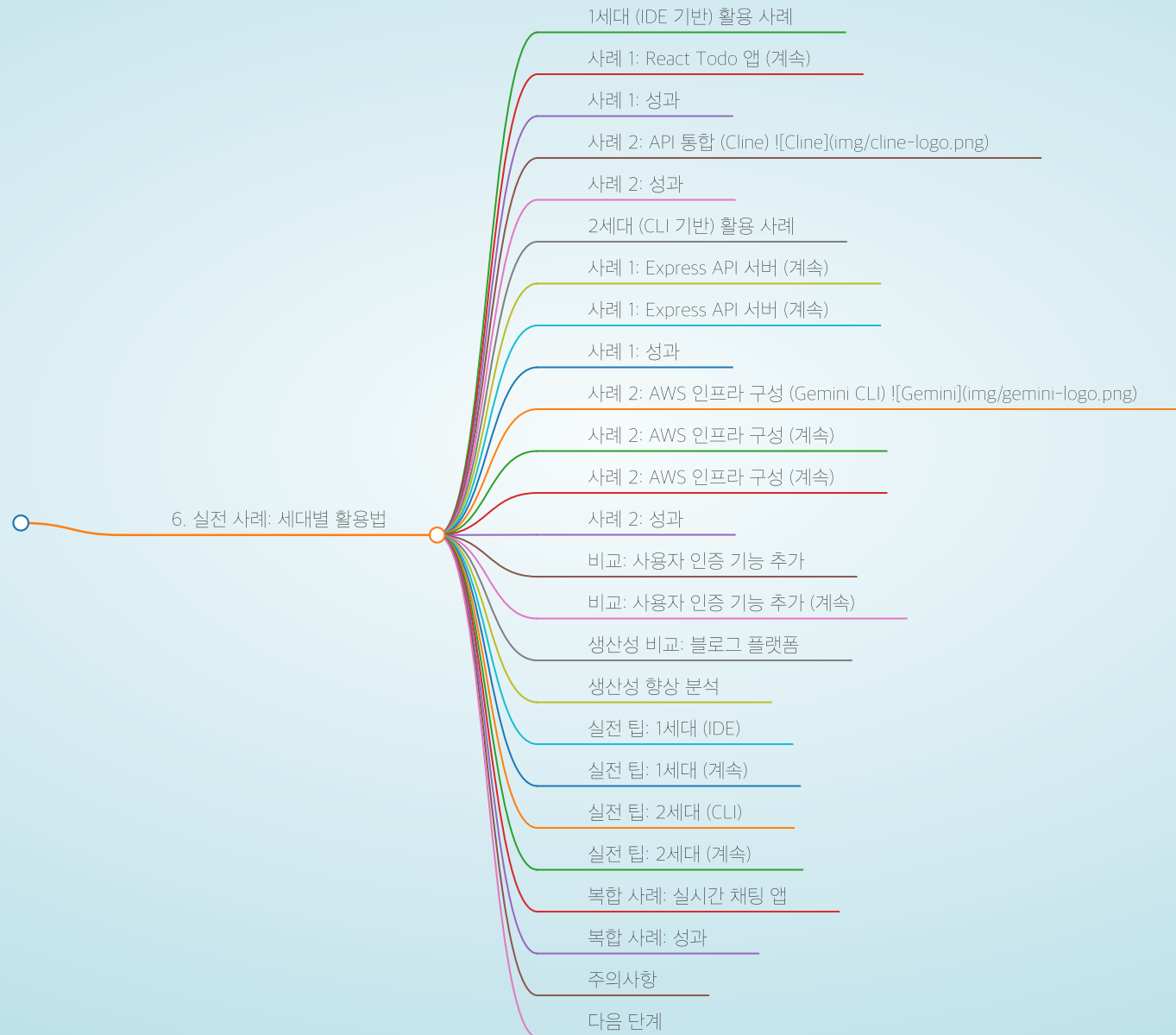


# Examples for MarkdownGraph



# 1세대 (IDE 기반) 활용 사례

6. 실전 사례: 세대별 활용법

## 사례 1: REACT TODO 앱 개발 (CURSOR)

시나리오: React와 TypeScript로 간단한 Todo 앱 만들기



# 사례 1: React Todo 앱 (계속)

6. 실전 사례: 세대별 활용법

## 작업 과정

1. Cursor에서 새 프로젝트 생성
2. Cmd+K 누르고 요청:  
"React와 TypeScript로 Todo 앱 만들어줘."
  - 할 일 추가, 삭제, 완료 표시
  - localStorage에 저장
  - Tailwind CSS로 스타일링"
3. AI가 자동 생성:
  - App.tsx, TodoItem.tsx
  - types/Todo.ts, hooks/useTodos.ts
4. diff 모드로 확인 후 승인
5. npm start로 즉시 실행

# 사례 1: 성과

항목	결과
소요 시간	약 5분
기존 방식	약 30-40분
생산성 향상	<b>6-8배</b>

## 장점

- 실시간 UI 프리뷰
- 컴포넌트 구조 시각적 확인
- 즉시 브라우저에서 테스트

# 사례 2: API 통합 (Cline) ![Cline](img/cline-logo.png)

6. 실전 사례: 세대별 활용법

시나리오: 기존 React 앱에 REST API 연동 p 가

## 작업 과정

1. Cline 채팅 패널에서 요청:  
"GitHub API를 사용해서 사용자 저장소 목록 가져오기."
  - axios 사용
  - React Query로 캐싱
  - 로딩 및 에러 상태 처리"
2. AI가 자동 작업:
  - api/github.ts 생성
  - hooks/useRepositories.ts 생성
  - RepositoryList.tsx 컴포넌트 생성
  - package.json에 의존성 추가 제안

# 사례 2: 성과

항목	결과
소요 시간	약 7분
기존 방식	약 45-60분
생산성 향상	<b>6-8배</b>

## 핵심 포인트

- API 호출 로직 자동 생성
- 에러 핸들링 자동 구현
- React Query 설정 완료

# 2세대 (CLI 기반) 활용 사례

6. 실전 사례: 세대별 활용법

사례 1: EXPRESS API 서버 (CLAUDE CODE) ! [CLAUDE]  
(IMG/CLAUDE-LOGO.PNG)

시나리오: Node.js Express로 RESTful API 서버 개발

# 사례 1: Express API 서버 (계속)<sup>6. 실전 사례: 세대별 활용법</sup>

## 작업 과정

```
$ claude
```

```
> Express로 블로그 API 서버 만들어줘.
```

- 게시글 CRUD
- 사용자 인증 (JWT)
- MongoDB 연동
- 입력 검증
- 에러 핸들링 미들웨어

[Claude가 작업 시작]

- ✓ server.js, routes/posts.js, routes/auth.js
- ✓ middleware/auth.js, models/Post.js, models/User.js
- ✓ .env.example, package.json, README.md

# 사례 1: Express API 서버 (계속)

6. 실전 사례: 세대별 활용법

> 테스트 코드도 작성해줘

- ✓ `__tests__/posts.test.js`
- ✓ `__tests__/auth.test.js`
- ✓ `jest.config.js`

> `npm install` 실행하고 서버 시작해줘

[`npm install` 실행]

[서버 시작됨]

Server running on port 3000

# 사례 1: 성과

항목	결과
소요 시간	약 10분
기존 방식	약 2-3시간
생산성 향상	<b>12-18배</b>

## 장점

- 멀티파일 자동 생성
- 보일러플레이트 자동 처리
- 테스트 코드까지 완성
- 즉시 실행 가능

# 사례 2: AWS 인프라 구성 (Gemini CLI)! [Gemini](img/gemini-logo.png)

시나리오: Terraform으로 AWS EC2 + RDS 인프라 구성

- cf) 2세대 Vibe Coding : AI 도구를 활용한 클라우드 인프라 자동화

# 사례 2: AWS 인프라 구성 (계속)

6. 실전 사례: 세대별 활용법

## 작업 과정

```
$ gemini
```

> AWS에 프로덕션 환경 인프라 구성해줘:

- VPC, 서브넷, 인터넷 게이트웨이
- EC2 인스턴스 (t3.medium, 2개)
- RDS PostgreSQL (Multi-AZ)
- 로드 밸런서, 보안 그룹 설정
- Terraform 코드로 작성

[Gemini가 Terraform 코드 생성]

- ✓ main.tf, variables.tf, outputs.tf
- ✓ vpc.tf, ec2.tf, rds.tf, security-groups.tf

# 사례 2: AWS 인프라 구성 (계속)

6. 실전 사례: 세대별 활용법

```
> terraform init하고 plan 보여줘
```

```
[terraform init 실행]
```

```
[terraform plan 실행 및 결과 표시]
```

```
Plan: 15 to add, 0 to change, 0 to destroy.
```

```
> apply해줘
```

```
[terraform apply 실행]
```

```
Apply complete! Resources: 15 added
```

# 사례 2: 성과

항목	결과
소요 시간	약 15분
기존 방식	약 3-4시간
생산성 향상	<b>12-16배</b>

## 핵심 포인트

- 복잡한 인프라 코드 자동 생성
- 베스트 프랙티스 자동 적용
- 즉시 배포 가능

# 비교: 사용자 인증 기능 추가

6. 실전 사례: 세대별 활용법

## 1세대 (Cursor) 접근

1. IDE에서 프로젝트 열기
2. Cmd+K: "JWT 기반 사용자 인증 추가"
3. 시각적으로 변경사항 확인
  - Login.tsx 수정 확인
  - auth.service.ts 생성 확인
4. Accept All
5. 브라우저에서 로그인 폼 확인

적합: 프A 트a 드 중심 작업, UI 피드S 필요

# 비교: 사용자 인증 기능 추가 (계속)

6. 실전 사례: 세대별 활용법

## 2세대 (Claude Code) 접근

```
$ claude
```

```
> 사용자 인증 기능 추가해줘:
```

- JWT 토큰 발급
- Refresh 토큰
- 비밀번호 해싱 (bcrypt)
- Rate limiting
- 로그인 시도 제한

```
[모든 파일 자동 수정 및 생성]
```

```
[테스트 코드 자동 생성]
```

```
[npm test 자동 실행]
```

적합: S a 드 전체 로직, 테스트 자동화

# 생산성 비교: 블로그 플랫폼

작업	전통	Cursor	Claude Code
프로젝트 설정	30분	5분	3분
데이터 모델	1시간	15분	10분
CRUD API	4시간	1시간	30분
프론트엔드 UI	6시간	2시간	-
인증	3시간	45분	20분
테스트	4시간	1시간	30분
배포 설정	2시간	1시간	15분
총합	20.5시간	6시간	1.8시간

## Cursor (1세대)

- 약 3.4배 빠I
- 시각적 피드S 장점
- UI 중심 작업에 o 적

## Claude Code (2세대)

- 약 11.3배 빠I
- 완전 자동화 가능
- S 존드/인프라에 o 적

# 실전 팁: 1세대 (IDE)

## 효율적인 프롬프트

❌ 나쁜 a : "컴시넌스 만들어줘"

✅ 좋은 a :

```
"사용자 프로필 카드 컴포넌트 만들어줘.  
- props: name, email, avatar, bio  
- Tailwind CSS 사용  
- 클릭 시 상세 페이지로 이동  
- 반응형 디자인"
```

# 실전 팁: 1세대 (계속)

## Composer 모드 활용

- 여러 파일을 한번에 수정
- @파일명으로 특정 파일 참조
- 프로젝트 전체 맥락 활용

## 단축키 마스터

- Cmd/Ctrl + K: 프롬프트 입력
- Cmd/Ctrl + L: 전체 프로젝트 개업
- Cmd/Ctrl + Shift + P: 명령 팔레트

# 실전 팁: 2세대 (CLI)

작은 단위로 분할

## 명확한 작업 단위

# 작은 단위로 분할

❌ "전체 애플리케이션 만들어줘"

✅ "먼저 데이터 모델 만들어줘"

✅ "이제 CRUD API 만들어줘"

✅ "테스트 코드 추가해줘"

# 실전 팁: 2세대 (계속)

AI 작업 후 바로 커밋

## Git 커밋 자주 하기

```
# AI 작업 후 바로 커밋
$ git add .
$ git commit -m "feat: Add user authentication"

# 문제 발생 시 쉽게 롤백
$ git reset --hard HEAD^
```

## 자동화 스크립트 활용

```
#!/bin/bash
claude << EOF
프로젝트 분석해서 README.md 업데이트해줘
테스트 실행해줘
커버리지 리포트 생성해줘
EOF
```

# 복합 사례: 실시간 채팅 앱

AI 작업 후 바로 커밋

## 1단계: 백엔드 (Claude Code) - 10분

- > Socket.IO 기반 채팅 서버 만들어줘:
  - 실시간 메시지 전송
  - 채팅방 관리
  - MongoDB에 메시지 저장

## 2단계: 프론트엔드 (Cursor) - 15분

Cmd+K: "Socket.IO 클라이언트로 채팅 UI 만들어줘"

## 3단계: 배포 (Claude Code) - 10분

- > Docker Compose로 컨테이너화해줘
- > AWS ECS 배포 설정해줘

# 복합 사례: 성과

AI 작업 후 바로 커밋

총 소요 시간라약 3대

전통 방식라약 10-1대시간

생산성 향상라17-25배

## 보안 체크리스트

- 수세 변경 Octi 장작방어 확인
- X수수방어 확인
- 인증구인가 로i 검증
- 민감 정보 하드t 단 여부
- A생번키 노출 여부

## 품질 체크리스트

- 에러 핸들링 적절성
- 테스트 커버특지
- t 드 가독성
- 서느 o 저장 여부

# 다음 단계

상세 도입 로드맵은 7장 참조

1. 작은 프로젝트부터 시작
2. 점가적 확대
3. 팀 협업 및 버스트 프랙티스 / s