DEVELOPER PORTFOLIO · 2026.09

박건우

Park Geon Woo

실시간 협업부터 AI 업무지원까지,
화면·API·상태로 연결한 개발 경험

ceh20002@naver.com · github.com/pigpgw

---

01 · 디지엠 유닛원 · 실무

CPPM

LG 전 계열사가 사용하는 공통 업무 플랫폼의 AI 업무지원 서비스

LG 전 계열사가 사용하는 공통 업무 플랫폼의 AI 업무지원 기능을 개발했습니다. 챗봇 화면부터 API, 데이터 조회, 그래프 생성, 장시간 작업 처리까지 전체 흐름을 맡았고, AWS Bedrock 기반 다중 에이전트 구조를 설계했습니다.

---

02 · 디지엠 유닛원 · 실무

프비티

프랜차이즈 본사·가맹점·고객을 연결하는 통합 솔루션

고객·본사·가맹점이 사용하는 화면을 개발하고 API와 연결했습니다. 영업관리·수발주 서비스의 분리, 계정·권한 흐름 분석, 리뷰 자동화와 AI 퍼블리싱 작업 방식 개선까지 맡았습니다.

---

03 · 크래프톤 정글 6기 · 최종 프로젝트

Code Sync

GitHub PR을 함께 보며 화상으로 리뷰하고 코드를 동시 편집하는 협업 서비스

프론트엔드 핵심 기능을 개발했습니다. GitHub PR 데이터를 리뷰 화면에 연결하고, 코드 동시 편집, 커서와 파일 위치 동기화, 노트·그림판 협업 등 Code Sync의 주요 기능을 구현했습니다.

디지엠 유닛원 · 실무

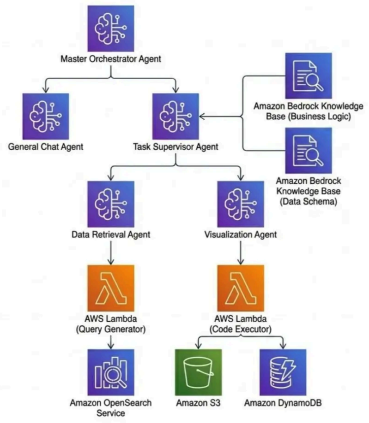
CPPM

AWS Bedrock 기반 다중 에이전트 설계·최적화

2025.10.01 - 2025.12.31 · 프론트엔드·풀스택 개발

LG 전 계열사가 사용하는 공통 업무 플랫폼에서 자연어로 업무 데이터를 조회하고 분석 결과를 그래프로 확인하는 챗봇을 개발했습니다. AWS Bedrock Agent POC를 검증한 뒤 일반 대화, 데이터 조회, 그래프 생성을 담당하는 에이전트를 나누고, Lambda·SQS·WebSocket 기반의 비동기 처리와 React 화면을 연결했습니다.

React · Next.js · TypeScript · Jest · Cypress · Express.js · MySQL · AWS Bedrock · Knowledge Base · OpenSearch · Lambda · SQS · DynamoDB · S3 · API Gateway WebSocket · CloudWatch · IAM



AWS Bedrock Agent 기반 다중 에이전트 아키텍처

CASE 01

다중 에이전트 아키텍처 설계 및 최적화

상황

초기 단일 Bedrock Agent에 일반 대화, 데이터 조회, 그래프 생성과 계열사별 업무 규칙이 집중되어 프롬프트 복잡도와 유지보수 비용이 증가했습니다. Trace·CloudWatch에서 검색 실패 뒤 기간 확장과 유사어 재검색이 반복되고, 검색 결과 없이 답변을 생성하는 흐름을 확인했습니다.

판단

AWS Bedrock Agent를 활용해 요청 분류와 작업 조율, 데이터 조회와 시각화의 책임을 나눴습니다. 업무 지식은 프롬프트에 모두 넣지 않고 필요한 시점에 Knowledge Base에서 조회하도록 구성했습니다.

구현

Master Agent가 일반 대화와 데이터 분석 요청을 분류하고, Supervisor Agent가 OpenSearch Agent와 Graph Agent의 작업을 조율하도록 설계했습니다.

일반 대화는 전용 Chat Agent로 보내 분석 경로를 거치지 않도록 했습니다. Master는 하위 Agent의 응답을 임의로 재해석하거나 다른 Agent로 재라우팅하지 않고 전달하도록 지침을 구성했습니다.

분석 유형은 Use Case Knowledge Base, 계열사별 필드·파생 지표·차트 규칙은 계열사별 Knowledge Base로 분리했습니다.

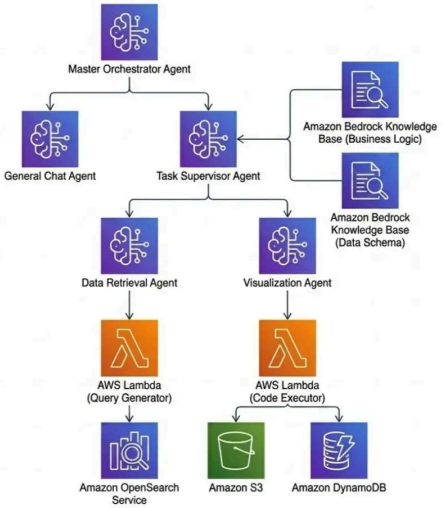
has_results=false 또는 hits.total.value=0이면 결과 없음으로 종료하고 후속 Graph Agent 호출을 중단하도록 했습니다. HTTP 500 오류는 재검색으로 해결하려 하지 않고 오류를 전달하도록 종료 규칙을 구성했습니다.

Graph Agent는 조회 결과와 차트 규칙을 받아 그래프 생성 코드를 만들고, Action Group의 Lambda 실행·S3 저장을 거쳐 이미지 URL을 반환하도록 연결했습니다.

결과

> Agent별 프롬프트·Action Group·Knowledge Base를 독립적으로 관리하고 불필요한 재검색과 후속 호출을 제어했습니다. Agent 역할과 프롬프트 최적화로 평균 응답 시간을 약 2분에서 1분~1분 30초로 단축했습니다.

응답시간은 당시 프로젝트의 개선 기록입니다. Trace UI의 대기 경험 개선과는 별도의 성과입니다.



Master·Chat·Supervisor·OpenSearch·Graph Agent와 Knowledge Base를 분리한 설계 구조

CASE 02

장시간 Agent 응답에 대한 사용자 대기 경험 개선

- 상황

OpenSearch 조회와 데이터 기반 그래프 생성은 System Prompt·Knowledge Base·Agent 협업·코드 생성·S3 업로드를 거치며 1분~2분 이상 걸렸습니다. 화면에는 로딩만 표시돼 사용자가 진행 중인지 오류인지 알기 어려웠습니다.

- 판단

Agent가 어떤 작업을 수행하는지 사용자에게 전달하고, 이벤트 수신·배포를 컴포넌트의 생명주기와 분리했습니다.

- 구현

  PreProcessingTrace와 OrchestrationTrace에서 입력 분류·Knowledge Base 조회·Action Group 호출 상태를 추출했습니다.

  Trace를 사용자 친화적인 처리 단계로 변환해 WebSocket으로 실시간 전달했습니다.

  싱글톤 AgentEventManager가 WebSocket 수신과 이벤트 배포를 담당하고 컴포넌트는 구독·해제하도록 구현했습니다.

- 결과

> 응답 속도를 억지로 줄이는 대신 처리 단계를 보여줘 체감 대기 시간을 줄였습니다. 이벤트 관리를 분리해 리스너 중복 등록을 방지하고 일관된 상태를 유지했습니다.

AgentEventManager 싱글톤과 컴포넌트 생명주기 분리

WebSocket 수신은 AgentEventManager 한 곳에서 담당하고, 화면 컴포넌트는 필요한 이벤트를 구독한 뒤 언마운트 시 해제하도록 구성한 구조 발췌입니다. 실제 사내 코드의 식별자와 전송 형식은 제거·축약했습니다.

typescript

```typescript
type AgentEvent = { type: string; message: string };
type Listener = (event: AgentEvent) => void;

class AgentEventManager {
  private static instance: AgentEventManager;
  private listeners = new Set<Listener>();
  private socket?: WebSocket;

  private constructor() {}

  static getInstance() {
    return (this.instance ??= new AgentEventManager());
  }

  connect(socket: WebSocket) {
    if (this.socket === socket) return;
    this.socket = socket;
    socket.addEventListener("message", this.handleMessage);
  }
```

```
  subscribe(listener: Listener) {
    this.listeners.add(listener);
    return () => this.listeners.delete(listener);
  }

  private handleMessage = (message: MessageEvent<string>) => {
    const event = JSON.parse(message.data) as AgentEvent;
    this.listeners.forEach((listener) => listener(event));
  };
}

const agentEventManager = AgentEventManager.getInstance();

// React 컴포넌트에서는 구독과 해제만 담당
useEffect(() => {
  const unsubscribe = agentEventManager.subscribe(setAgentEvent);
  return unsubscribe;
}, []);
```

구조 발췌 · 수신 계층과 React 구독 계층을 분리

CASE 03

API Gateway 타임아웃 해결 · SQS 기반 비동기 구조 분리

상황

Bedrock Agent의 데이터 조회·그래프 생성은 1~2분이 걸렸습니다. 요청 Lambda가 최종 결과를 기다리면서 API Gateway WebSocket 통합 요청이 먼저 타임아웃됐습니다.

판단

장시간 작업을 요청 수신과 분리했습니다. 연결 식별자를 작업과 함께 전달하고, 결과는 기존 WebSocket 연결로 돌려주는 구조를 구성했습니다.

구현

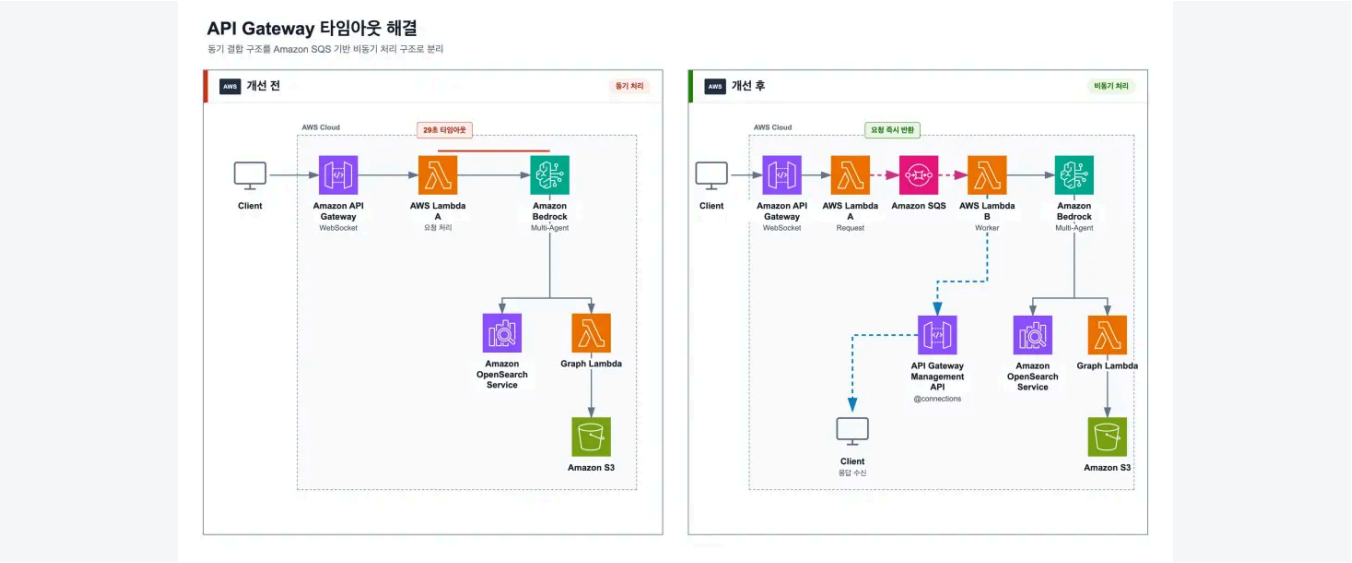
요청 Lambda가 connectionId와 요청 정보를 SQS에 적재한 뒤 즉시 반환하도록 구성했습니다.

Event Source Mapping으로 호출되는 Worker Lambda가 Bedrock·OpenSearch·그래프 생성 작업을 수행하도록 분리했습니다.

Worker가 API Gateway Management API의 PostToConnection으로 응답과 Trace를 전달하고, 화면에는 사용자 친화적인 처리 단계를 표시했습니다.

결과

> 요청 수신과 장시간 작업을 분리해 통합 타임아웃 문제를 해소하고, 사용자가 정상 진행 여부를 확인할 수 있는 흐름을 만들었습니다.



요청 수신 Lambda와 Worker Lambda를 SQS로 분리한 전후 구조

CASE 04

채팅 검색 및 탐색 경험 최적화

상황

화면에 로드되지 않은 이전 메시지는 브라우저 기본 검색으로 찾을 수 없어 채널과 대화를 직접 탐색해야 했습니다.

판단

검색 결과를 보여주는 데서 끝내지 않고 해당 채널·메시지로 이동하는 전체 흐름을 제안하고 구현했습니다.

구현

ERD·REST API와 React 검색 화면, 검색어 하이라이팅을 구현했습니다.

300ms debounce와 LastEvaluatedKey 기반 20개 단위 페이지네이션으로 요청과 응답 범위를 제한했습니다.

채널 이동 후 requestAnimationFrame에서 DOM을 조회해 해당 메시지 위치로 스크롤하도록 구현했습니다.

결과

> 이전 대화를 검색하고 해당 메시지의 맥락으로 돌아가는 탐색 흐름을 완성했습니다.

ARCHITECTURE & PROMPTS

역할 분리와 종료 조건을 프롬프트에 명시

Bedrock의 Agent 협업 기능을 사용하되, 어떤 요청을 누구에게 보내고 언제 멈출지는 직접 설계했습니다. 아래는 업무 기록에 남긴 Master·Supervisor 프롬프트의 일부입니다.

> Master는 일반 대화와 분석 요청을 분류하고, 일반 대화는 Chat Agent에 바로 전달하도록 했습니다.
> Supervisor는 Use Case KB에서 분석 방법을, 계열사별 KB에서 필드·파생 지표·차트 규칙을 찾아 OpenSearch·Graph Agent에 전달하도록 했습니다.
> 검색 결과가 없거나 서버 오류가 발생하면 후속 그래프 생성과 불필요한 재호출을 멈추도록 응답 처리 규칙을 작성했습니다.

Master: 요청 분류와 응답 전달

일반 대화가 분석 Agent까지 거치지 않도록 경로를 나눴습니다. 하위 Agent가 결과를 반환한 뒤 Master가 다시 판단해 다른 Agent를 호출하지 않도록 제한했습니다.

markdown

```markdown
## 라우팅 규칙

- **일반 대화** (인사, 질문, 시스템 기능 안내 등) → CppmChatAgent
- **데이터 검색/분석/그래프 생성** → CppmSupervisorAgent

## 응답 전달 (매우 중요 - 절대 규칙)

- 하위 에이전트(CppmChatAgent, CppmSupervisorAgent)의 응답을 **반드시 그대로** 사용자에게 전달합니다.
- 하위 에이전트의 응답이 만족스럽지 않아도 그대로 전달합니다. 절대 다른 에이전트로 다시 라우팅하지 않습니다.
```

Master Agent 프롬프트 · 원문 발췌

Supervisor: 검색 결과에 따른 종료

빈 결과와 서버 오류를 구분해 사용자에게 알리도록 했습니다. 특히 HTTP 500 응답을 받은 Supervisor가 검색을 다시 지시하지 않도록 종료 조건을 명시했습니다.

markdown

```markdown
CppmOpenSearchAgent 응답 처리 규칙:
- CppmOpenSearchAgent가 검색 결과를 반환했을 때:
  - has_results가 false이거나 hits.total.value가 0인 경우: 검색 결과가 없는 것이므로, 사용자에게 "검색 결과가 없습니다"라고 명확히 전달하고 FINISH로 종료합니다.
  - error 필드가 있는 경우:
    * 에러 메시지에 "500", "service unavailable", "Internal Server Error"가 포함된 경우: "일시적 오류가 발생했습니다. 잠시 후 다시 시도해주세요"라고 사용자에게 전달하고 FINISH로 종료합니다.
    * 그 외의 에러: 에러 메시지를 사용자에게 전달하고 FINISH로 종료합니다.
  - 정상적인 검색 결과가 있는 경우: 다음 단계(예: CppmGraphAgent)로 진행하거나 FINISH로 종료합니다.
```

Supervisor Agent 프롬프트 · 원문 발췌

디지엠 유닛원 · 실무

프비티

본사·가맹점·고객을 연결하는 프랜차이즈 통합 솔루션

2025.07 - 2026.01 · 프론트엔드·풀스택 개발

프비티는 프랜차이즈 본사가 멤버십, 수발주, 재고, 가맹점 관리, CRM 마케팅을 한곳에서 운영하는 솔루션입니다. 고객용 화면과 본사·가맹점 운영 화면을 개발했으며, API·계정·권한 의존성을 분석해 영업관리·수발주 서비스를 분리하고 Flask·PyQt6 기반 네이버 리뷰 자동화도 구현했습니다.

React · Next.js · TypeScript · styled-components · Spring Boot · MySQL · Docker · Flask

출처: 프비티 공식 서비스 소개 · 제품 소개 이미지이며 개인 구현 범위는 아래 기여 항목에 구분했습니다.

프비티 공식 서비스 소개 · https://www.fbiti.co.kr/

CASE 01

고객 프로모션과 운영 화면의 반응형 UI 개발

- 상황

고객은 모바일로 소식과 프로모션을 확인하고, 가맹점과 본사는 태블릿·데스크톱으로 운영 업무를 처리해야 했습니다.

- 판단

고객용 콘텐츠 화면과 본사·가맹점의 운영 화면을 각 사용 환경에 맞게 구현하고 API를 연동했습니다.

- 구현

  여러 브랜드의 고객 유입을 지원하는 CRM·마케팅 랜딩을 배너·캐러셀·공지·이벤트·프로모션 중심으로 리뉴얼했습니다.

  룰렛 프로모션 참여부터 결과 확인까지 이어지는 화면을 구현했습니다.

  고객 모바일·가맹점 태블릿·본사 데스크톱에 맞춰 반응형 UI를 구성하고 크로스 브라우징 이슈를 해결했습니다.

  프랜차이즈 운영 어드민의 기존 기능을 유지보수하고 신규 콘텐츠·공지사항 화면을 API와 연결했습니다.

- 결과

> 고객이 소식·이벤트를 확인하고 프로모션에 참여할 수 있는 화면과 본사·가맹점의 운영 화면을 제공했습니다.

CASE 02

영업관리·수발주 서비스와 계정관리 분리

- 상황

영업관리·수발주·계정관리 기능이 강하게 결합돼 각 도메인을 독립적으로 운영하기 어려웠습니다.

- 판단

코드를 옮기기 전에 의존성과 API 연결 범위를 분석하고 각 프로젝트에 필요한 계정·인증 흐름을 정리했습니다.

- 구현

  Next.js·TypeScript와 PHP·MariaDB 기반 서비스에서 36,238 LOC·300개 이상 파일·39개 API 규모의 영향 범위를 분석했습니다.

  hqID·brandID 참조 233곳, header.php 참조 24개 파일, EcAccessControl 사용 18개 파일을 추적해 데이터·인증·권한 의존성을 구분했습니다.

  영업관리 화면을 별도 저장소로 분리하고 API·모델·Repository와 전용 계정 흐름을 재구성했습니다.

  공유 테이블과 공통 인증 로직의 사용 범위를 서비스별로 재정의해 사용자 조회·권한 확인·API 접근 흐름을 분리했습니다.

- 결과

> 영업관리와 수발주를 별도 프로젝트로 분리하고 각 서비스에 필요한 계정관리 흐름을 구성했습니다.

파일·LOC·API 수는 분석 범위입니다. 개발량이나 생산성 지표로 해석하지 않습니다.

CASE 03

AI 퍼블리싱 작업에 공용 컴포넌트와 검수 기준 적용

상황

반복 퍼블리싱과 API 확정 후 재작업이 발생했고, 단발성 AI 코드 생성만으로 공용 컴포넌트와 접근성 기준을 일관되게 반영하기 어려웠습니다.

판단

AI가 디자인뿐 아니라 기존 컴포넌트와 프로젝트 규칙을 함께 참고하도록 작업 지침을 만들고, 구현 전 계획과 구현 결과를 사람이 검토하도록 했습니다.

구현

Figma 디자인과 가이드, 기존 공용 컴포넌트를 참조하는 frontend-publishing Skill을 구성했습니다.

시맨틱 마크업·ARIA·모바일 우선 반응형 기준과 검증 절차를 명시했습니다.

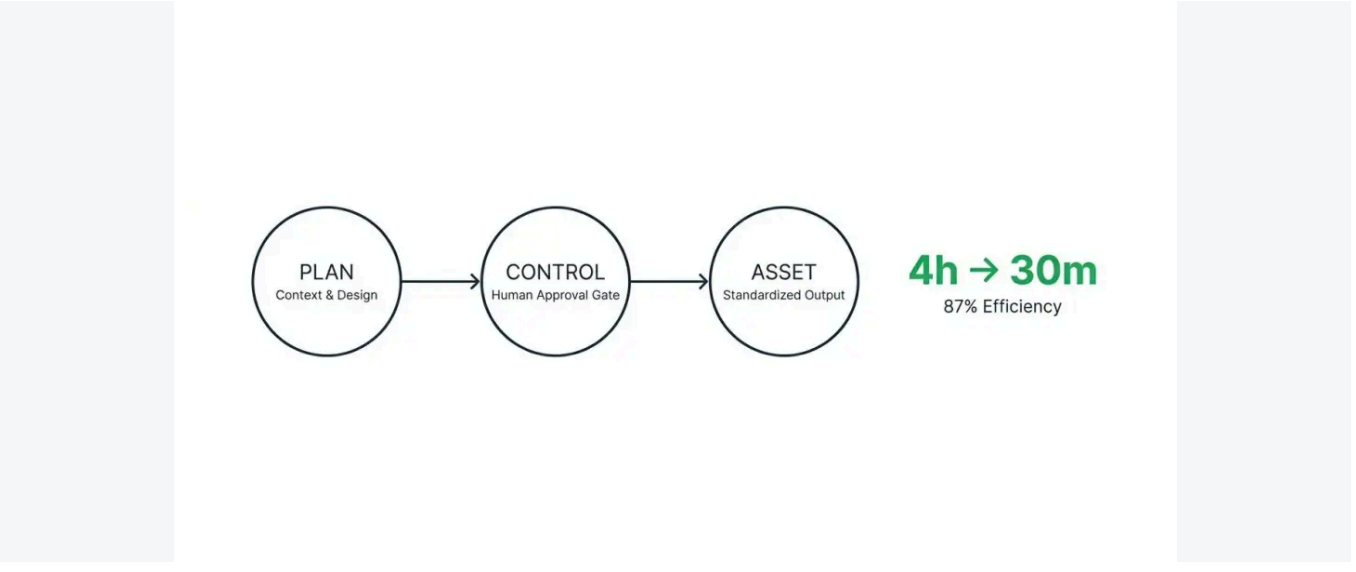
구현 전에 범위와 검증 항목을 정하고 결과를 사람이 검수하는 흐름을 적용했습니다.

Figma MCP로 디자인 정보를 읽도록 연결했습니다. Cursor Docs·MCP·Skills 활용 방법을 팀에 공유하고, 기존 이슈·PR·기술 문서를 작업 시 참고하도록 했습니다.

결과

> 반복 가능한 퍼블리싱 절차를 만들었습니다. 당시 작업 기록에서 반복 퍼블리싱 시간이 4시간 이상에서 약 30분으로 줄었습니다.

특정 반복 작업의 기록이며 전체 개발 생산성이나 모든 화면의 소요 시간을 의미하지 않습니다.



디자인·맥락 확인, 사람의 승인, 표준화된 산출물로 이어지는 퍼블리싱 절차

WORKFLOW & SKILL

frontend-publishing Skill에 담은 작업 기준

AI가 화면을 생성하는 데서 끝나지 않고, 기존 컴포넌트와 프로젝트 규칙을 따르도록 작업 지침을 구성했습니다. 구현 전 범위를 확인하고 결과를 검수하는 과정은 사람이 맡도록 했습니다.

입력: Figma 디자인·디자인 가이드·기존 공용 컴포넌트와 관련 기술 문서를 함께 참조하도록 했습니다.
구현: 공용 컴포넌트 재사용, 시맨틱 마크업·ARIA, 모바일 우선 반응형 기준을 명시했습니다.
검수: 구현 전에 계획과 검증 항목을 확인하고 승인된 범위에서 작업한 뒤 품질을 점검하도록 했습니다.

작업 계획부터 검수까지

아래는 업무 기록에 기재된 적용 기준을 정리한 요약입니다. 당시 Skill 원본 파일을 그대로 옮긴 코드는 아닙니다.

markdown

```
## 작업 전
- 기존 코드와 공용 컴포넌트 확인
- Figma 디자인과 프로젝트 디자인 가이드 참조
- 구현 범위와 검증 항목을 정하고 사람의 승인 확인

## 구현 기준
- 기존 공용 컴포넌트 재사용
- 시맨틱 마크업과 ARIA 적용
- 모바일 우선 반응형 구현

## 결과 검수
- 프로젝트 규칙과 공용 컴포넌트 사용 여부 확인
- 접근성과 반응형 구현 결과 확인
- 구현 결과를 사람이 검수
```

디지엠 유닛원 업무 기록 · Skill 적용 기준 요약

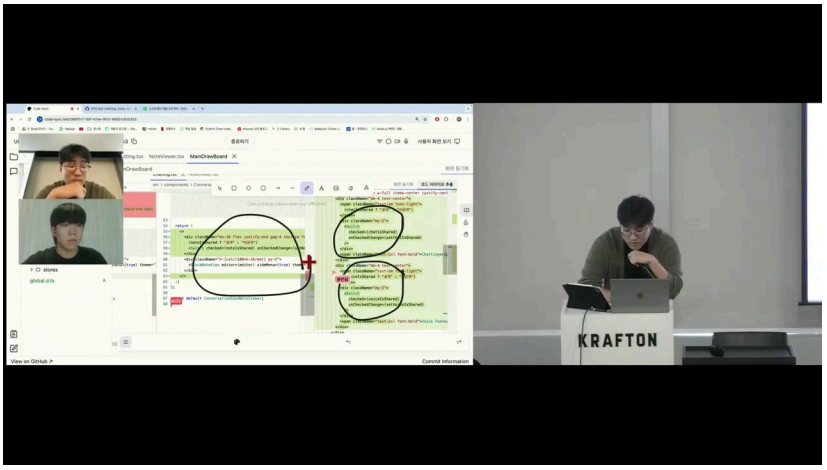
Code Sync

실시간 화상 코드리뷰 플랫폼

2024.10.11 - 2024.11.16 · 프론트엔드 핵심 기능 개발

크래프톤 정글 6기 최종 팀 프로젝트로, GitHub PR 리뷰에 화상 대화와 실시간 협업 기능을 더한 서비스입니다. 프론트엔드 핵심 기능을 맡아 PR 데이터 조회와 리뷰 화면, 코드 동시 편집, 파일·커서·화면 위치 동기화, Host와 Guest 사이의 데이터 공유 흐름을 구현했습니다.

React · TypeScript · Yjs · Monaco Editor · GitHub API



Code Sync 팀 발표 영상 · 아래 발표 영상 링크에서 확인할 수 있습니다

GitHub · https://github.com/pigpgw/code-sync-fe

발표 영상 · https://www.youtube.com/watch?v=yfaaHW_4KC8

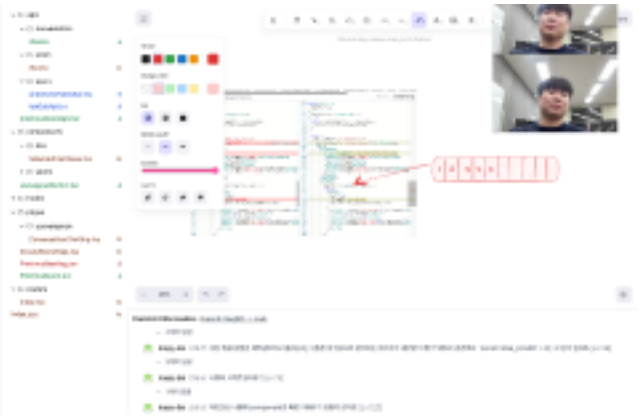
CODE SYNC · PRODUCT EVIDENCE

리뷰 화면과 협업 흐름

GitHub PR 리뷰, 동시 편집, 문서·화이트보드 협업을 연결한 실제 화면입니다.

PR 변경 파일 동시 편집

변경 전후 코드를 비교하고 같은 PR 맥락에서 리뷰하는 화면

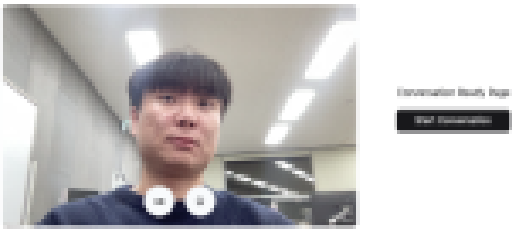


코드 위에 설명을 더하는 DrawBoard

에디터 화면과 드로잉을 함께 활용하는 회의 화면

회의 내용을 함께 기록하는 문서

화상 대화와 노트 협업을 연결한 화면



회의 입장 전 준비

카메라·마이크 상태를 확인하고 회의에 입장하는 서비스 화면

CASE 01

변경된 코드에서도 리뷰 맥락 유지하기

상황

리뷰 댓글은 과거 커밋의 파일을 가리키지만 GitHub REST API는 최신 변경 파일만 반환했습니다. 파일이 삭제되거나 경로가 바뀌면 댓글은 남아 있어도 연결할 파일을 찾지 못해 리뷰 맥락이 끊겼습니다.

판단

문제를 단순한 API 오류로 처리하지 않고 댓글과 파일의 기준 시점이 다르다는 데이터 모델의 차이로 정의했습니다. 추가 API 호출 없이 클라이언트에서 현재 파일 목록과 댓글의 경로·상태를 비교하도록 했습니다.

구현

최신 변경 파일 목록을 기준으로 댓글이 참조한 경로의 존재 여부를 판별하고, 목록에 없으면 Outdated 상태로 분류했습니다.

Outdated 댓글은 회색 배경과 안내 메시지로 구분해 사용자가 댓글이 사라진 것이 아니라 이전 코드에 남은 리뷰임을 이해하도록 했습니다.

PR 병합 뒤 head 브랜치가 삭제되는 경우에는 브랜치명에 의존하지 않고 head.sha와 base.sha를 기준으로 변경 내역을 조회하도록 분기했습니다.

결과

> 삭제·이동된 파일의 댓글도 리뷰 맥락을 유지하고, 병합 이후에도 변경 내역을 다시 확인할 수 있게 했습니다.

댓글과 최신 파일 목록의 상태 비교

최신 변경 파일 목록에 댓글의 경로가 없으면, 댓글을 삭제하지 않고 이전 코드에 남은 리뷰로 분류합니다.

typescript

```typescript
const currentPaths = new Set(
  changedFiles.map((file) => file.filename),
);

const commentsWithStatus = comments.map((comment) => ({
  ...comment,
  isOutdated: !currentPaths.has(comment.path),
}));
```

구현 로직 요약 · 실제 식별자와 API 전송 형식은 축약

CASE 02

PR 데이터는 한 번 조회하고 함께 사용하기

상황

참여자마다 PR 메타데이터, 변경 파일, 파일별 diff, 댓글을 다시 요청했습니다. 변경 파일이 10개인 경우 한 참여자당 13회 이상의 API 호출이 발생했고, 참여자가 늘수록 같은 요청이 반복됐습니다.

판단

각 사용자가 데이터를 요청하는 구조를 방 생성자의 최초 조회 결과를 공유하는 구조로 전환했습니다. 서버는 GitHub 데이터를 반복해서 중계하기보다 WebSocket 기반 상태 동기화에 집중하도록 책임을 나눴습니다.

구현

Host가 조회한 파일·댓글은 Y.Array(fileMetadata·commentMetadata), PR 정보는 Y.Map(prInfoMetadata)에 저장했습니다.

방 생성자만 GitHub API를 최초 1회 호출하고 PR 데이터를 Y.Doc에 저장했습니다. 이후 참여자는 Yjs 동기화로 데이터를 받아 별도 API 요청 없이 화면을 렌더링했습니다.

PR의 head.sha·base.sha를 함께 저장해 공유된 PR snapshot과 커밋 기준 변경 내역을 같은 리뷰 화면에서 사용했습니다.

결과

> Host가 조회한 PR 데이터를 Y.Doc으로 공유하고 Guest가 복원하도록 연결했습니다. GitHub API 호출을 44회에서 22회로 줄여 중복 호출과 Rate Limit 위험을 낮췄습니다.

연결 완료와 문서 동기화 완료 구분

소켓 연결과 공유 문서 동기화는 별개의 상태이므로, Yjs의 sync 이벤트를 기다린 뒤 Guest의 PR 데이터를 복원하도록 했습니다.

typescript

```typescript
export const waitForYdocSync = (provider: WebsocketProvider) => {
  if (provider.synced) {
    return Promise.resolve();
  }

  return new Promise<void>((resolve) => {
    const handleSync = (isSynced: boolean) => {
      if (!isSynced) return;

      provider.off("sync", handleSync);
      resolve();
    };

    provider.on("sync", handleSync);
  });
};
```

구현 코드 · yjs.ts

src/lib/yjs.ts · GitHub에서 원본 보기

CASE 03

같은 코드, 같은 위치에서 리뷰하기

- 상황

참여자가 서로 다른 파일과 위치를 보면 설명을 반복해야 했습니다. 파일 전환 뒤 이전 파일의 커서가 남는 상태도 정리해야 했습니다.

- 판단

동시 편집 내용의 동기화와 충돌 병합은 Yjs를 사용했습니다. 직접 구현한 부분은 파일별 Monaco 연결, 상대방 위치 표시, 화면 이동과 파일 전환 시 정리 로직입니다.

- 구현

  회의마다 Y.Doc을 생성하고 파일명을 키로 Y.Text를 구분했습니다. 선택한 파일의 Y.Text와 Monaco 모델을 MonacoBinding으로 연결하고, 파일을 바꾸면 기존 바인딩을 해제했습니다.

  Awareness에 현재 파일 경로를 저장하고 상대방의 위치를 화면에 표시했습니다. 화면 동기화 버튼을 누르면 상대방이 보는 파일·노트·그림판으로 이동하도록 구현했습니다.

  BlockNote와 Excalidraw의 협업 기능을 같은 Yjs 세션에 연동하고, 파일 전환 시 기존 커서 상태와 이벤트 리스너를 정리했습니다.

- 결과

> 리뷰 중 상대방이 설명하는 파일로 바로 이동하고, 같은 파일을 함께 편집할 수 있게 했습니다.

Awareness에 현재 파일 경로 저장

Awareness의 상태 공유 기능에 사용자 정보와 현재 파일 경로를 저장했습니다. 정리 함수에서는 이전 사용자 상태를 비웁니다.

typescript

```typescript
provider.awareness.setLocalStateField("user", {
  name: checkUser.data.name,
  color: "#ff6161",
  colorLight: "#30bced33",
  cursor: {
    current_file_path: selectedCommitFile.filename,
  },
});

return () => {
  provider?.awareness.setLocalStateField("user", null);
};
```

구현 코드 · MainFrame.tsx

src/components/Frame/MainFrame.tsx · GitHub에서 원본 보기

CASE 04

Route 단위 Code Splitting과 Lazy Loading

상황

여러 협업 기능과 무거운 편집·화상·문서 라이브러리를 사용하는 SPA가 처음 진입할 때, 사용하지 않는 화면 코드까지 함께 불러왔습니다. 첫 화면에 필요한 범위보다 초기 JavaScript가 커져 진입 비용이 늘어나는 구조였습니다.

판단

화면 단위로 코드를 나누고 필요한 경로에 진입할 때 불러오도록 해, 첫 화면과 이후 기능의 로딩 시점을 분리했습니다.

구현

로그인·회원가입·회의 생성·이전 회의·저장·공유 페이지를 React.lazy와 Suspense로 분리해 route 단위 청크로 나눴습니다.

초기 화면에 필요한 코드만 먼저 불러오고, 분리된 화면은 해당 경로에 진입할 때 로드되도록 구성했습니다.

초기 진입 JavaScript 번들의 변경 전후 크기를 비교해 최적화 결과를 확인했습니다.

결과

> 당시 기록 기준 초기 JS 번들을 1,820KB에서 99KB로 줄였습니다.

당시 기록한 초기 JS 청크 크기이며, 전체 파일 용량이나 현재 빌드 크기를 뜻하지 않습니다.

화면 단위로 코드 나누기

회의와 협업 화면을 초기 번들에서 분리하고, 해당 경로에 진입했을 때 필요한 코드를 불러오도록 구성했습니다.

typescript

```
const MeetingPage = lazy(() => import("./pages/MeetingPage"));
const SavedMeetingPage = lazy(
  () => import("./pages/SavedMeetingPage"),
);

<Suspense fallback={<PageSkeleton />}>
  <Routes>
    <Route path="/meeting" element={<MeetingPage />} />
    <Route path="/saved" element={<SavedMeetingPage />} />
  </Routes>
</Suspense>;
```

구현 로직 요약 · 실제 페이지명과 경로는 축약