한계 너머를 꿈꾸는

Engineering Portforlio

김정래 Node.js Developer

LLM기반 프로젝트 생성
자동 Feadback 시스템 구축

(주) 구름, Ide SQD

Fullstack Engineer

LLM 코드 생성 조건을 수정할 때

프롬프트 버전 조정

사용 가능한 모델 변경

요청 후 실행 결과 확인

하나의 유형을 재검증 할 때 마다 수 십 분의 시간을 소모

오류 결과에 대한 피드백을 어렵게 만듬

성공여부 검증 시스템

Web 기반 프로젝트 검증

프로세스가
올바르게 떴는가?

OS의 포트감지

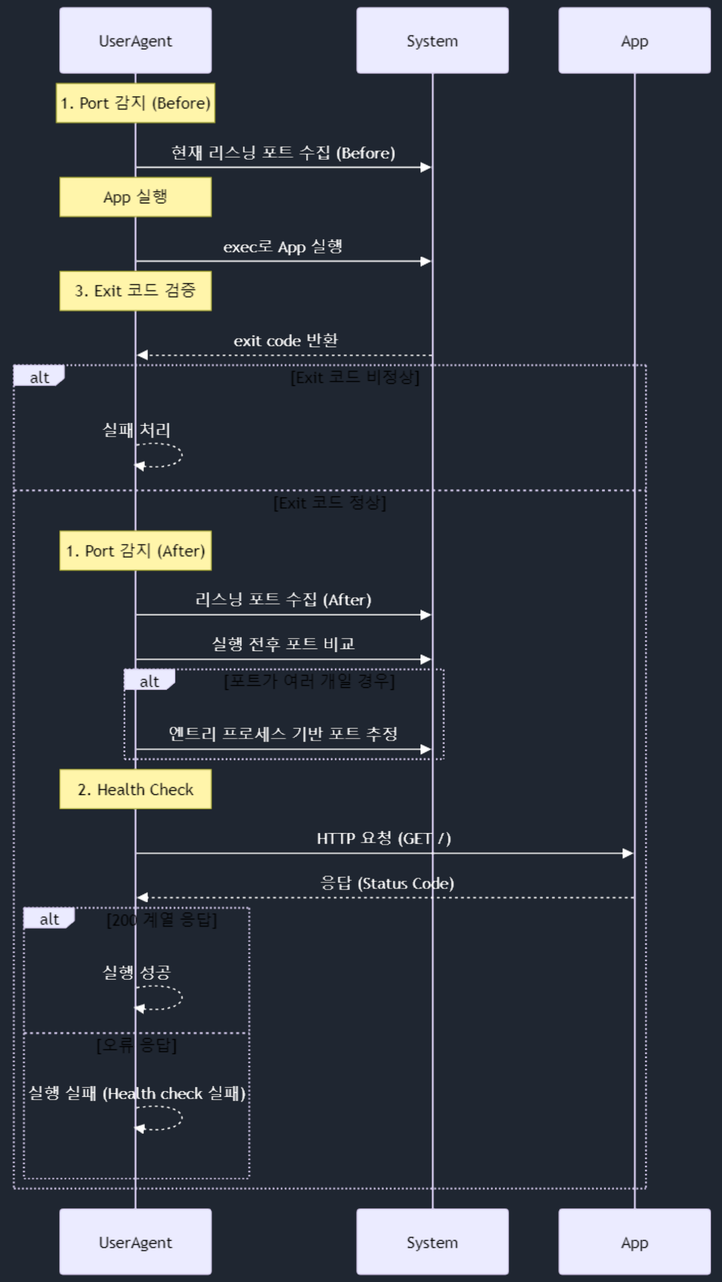
Exec 기반 Exit code 분류

/ 경로에 대한 네트워크 요청

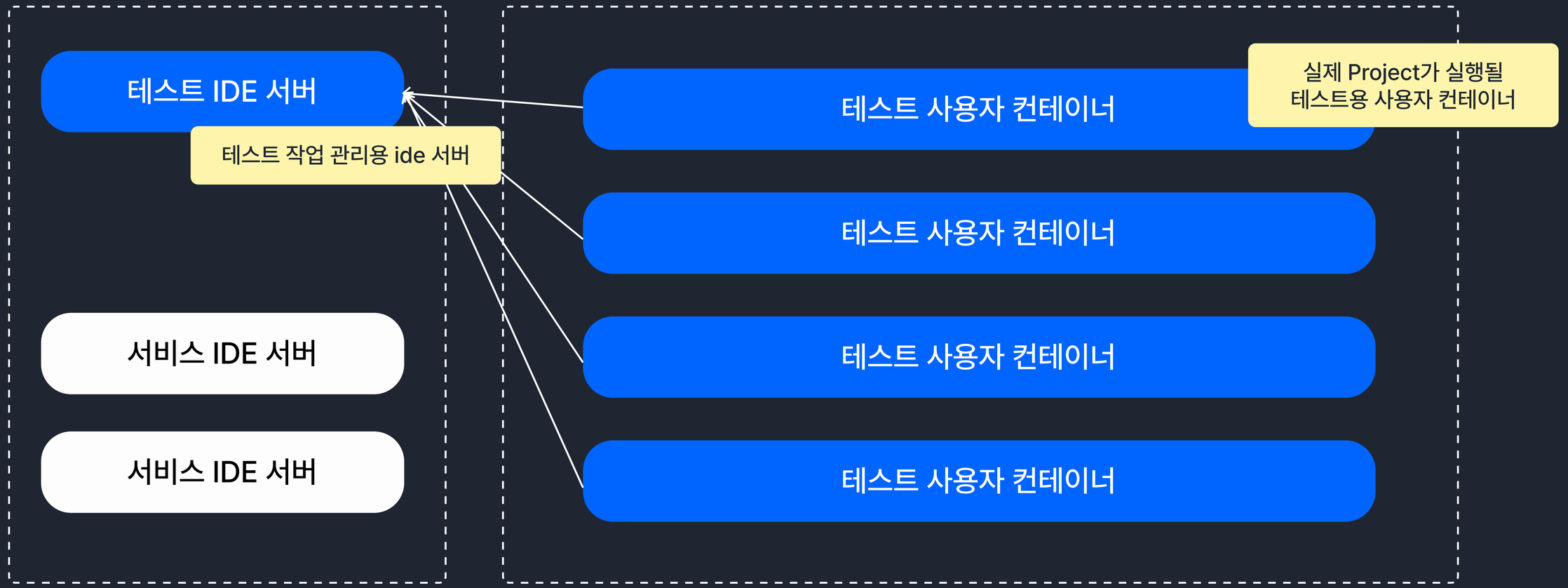
정규식 기반 터미널 로그 검증

health Check을
통과하는가?

OS 와의 시퀀스

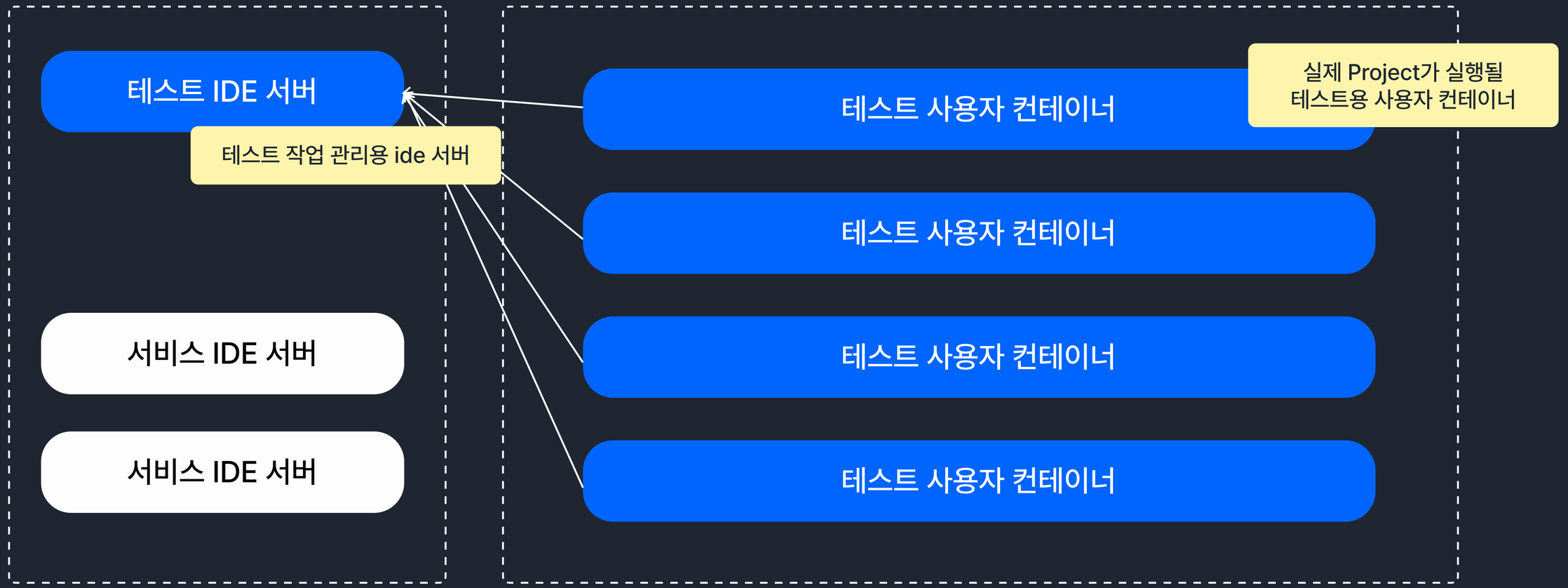

UserAgent
System
App
1. Port 감지 (Before)
현재 리스닝 포트 수집 (Before)
App 실행
exec로 App 실행
3. Exit 코드 검증
exit code 반환
alt
[Exit 코드 비정상]
실패 처리
[Exit 코드 정상]
1. Port 감지 (After)
리스닝 포트 수집 (After)
실행 전후 포트 비교
alt
[포트가 여러 개일 경우]
엔트리 프로세스 기반 포트 추정
2. Health Check
HTTP 요청 (GET /)
응답 (Status Code)
alt
[200 제일 응답]
실행 성공
[오류 응답]
실행 실패 (Health check 실패)
UserAgent
System
App

여러 케이스로 재시도 해보기

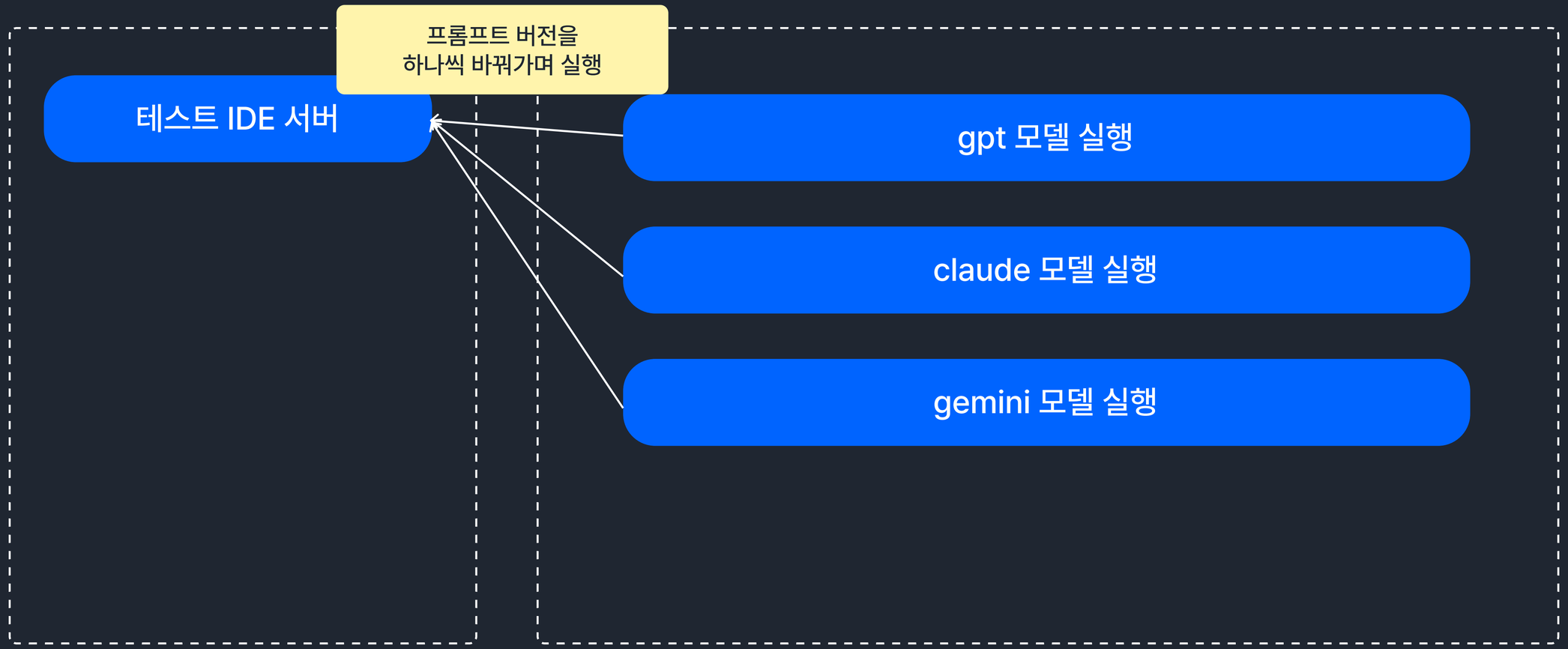


서버와 실행 컨테이너 영역이 구분된 k8s 클러스터

여러 케이스로 재시도 해보기



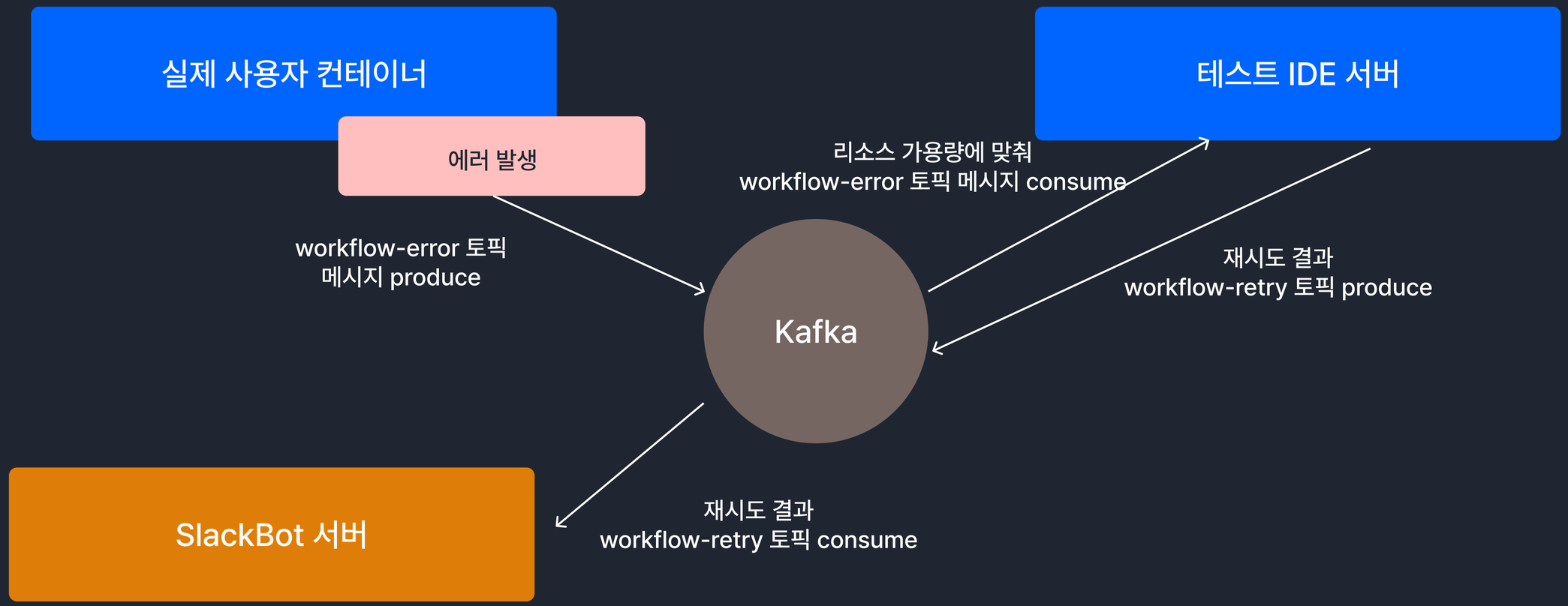
서버와 실행 컨테이너 영역이 구분된 k8s 클러스터



각 사용자 컨테이너를 모델별로 병렬적으로 실행하여 테스트

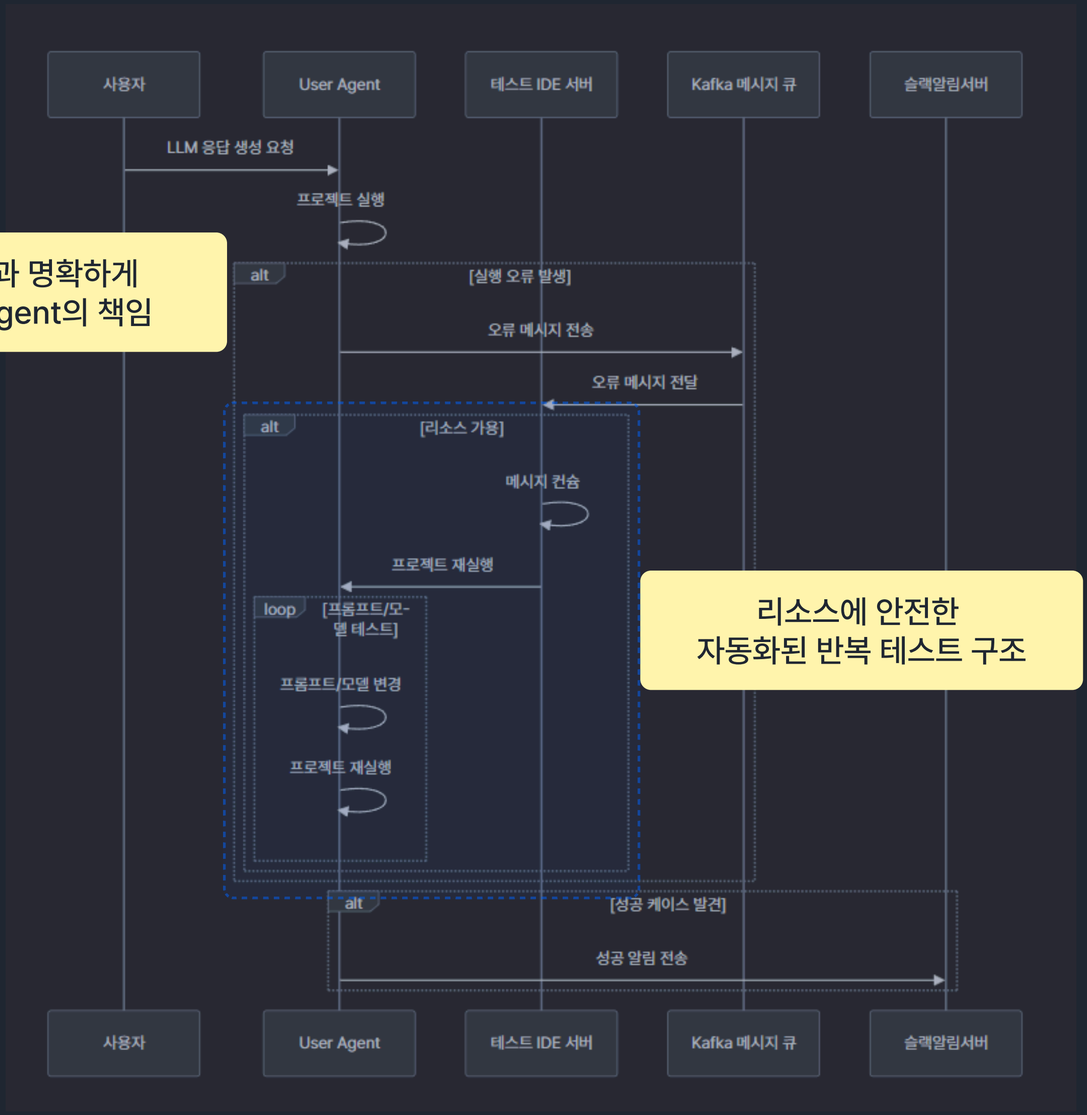
테스트에 사용할 수 있는 가용 리소스에 제한이 있음

Kafka 기반으로 리소스에 맞춰 작업을 컨슘



흐름

에러 분석과 명확하게
분리되는 agent의 책임



리소스에 안전한
자동화된 반복 테스트 구조

성과

오류 개당 처리 시간

기존 30분 ▶ 도입 후 6분

오류 해결 가능 조합 발견

74% 확률로 적절한 조합 발견

오류 케이스 재발생 확률

동일한 오류 발생 확률 41% 감소

로깅을 우아하게,
LLM 로깅 시스템 구축

(주) 구름, Ide SQD

Fullstack Engineer

로깅의 불편함

```
class WorkflowService implements LlmService {
    constructor(private readonly tokenLogger: Logger) {}

    async generate(userId: string, containerId: string, feature: LlmFeature) {
        const llm = new ChatOpenAI({
            model: "gpt-4o",
            callbacks: [
                {
                    handleLLMEnd: (output) => {
                        const tokens = output.generations
                            .flat()
                            .map((gen) => gen.text.trim());
                        this.tokenLogger.log({
                            tokens,
                            finger: {
                                userId,
                                containerId,
                                feature,
                            },
                        });
                    },
                },
            ],
        });

        // 생략
    }
}
```

로깅을 위해 관련 이벤트에
콜백을 항상 등록 시켜야한다

로깅에만 사용되는 인자를
항상 전달해주어야 한다.

응답 생성이, 로깅이라는 책임을 같이 가지게 된다.

컨텍스트로 사용자 정보를 넘기기

AsyncLocalStorage를 통해
컨텍스트를 관리할 수 있다.

```ts
import { AsyncLocalStorage } from "node:async_hooks";

export interface LlmContext {
    userId: string;
    containerId: string;
    feature: LlmFeature;
}

export const contextStorage = new AsyncLocalStorage<LlmContext>();

export function getContext(): LlmContext | undefined {
    return contextStorage.getStore();
}

export async function runWithContext<T>(
    context: LlmContext,
    fn: () => Promise<T>
): Promise<T> {
    return contextStorage.run(context, fn);
}
```

미들웨어에서 context.run
호출을 통해 시작을 할 수 있다

```ts
export function fingerMiddleware(req: Request, res: Response, next: NextFunction) {

  const { userId, containerId } = TokenParser.getPayload(req);
  const feature = getFeature(req);

  const context: LlmContext = { userId, containerId, feature };

  thContext(context, async () => {
    xt();
  });
}
```

미들웨어 등록시
컨텍스트를 기록하며 시작

기본 정보는
가지고 있던 곳에서 획득

한 흐름내에서 참조가능한 정보는 컨텍스트로 관리

팩토리를 통한 일반화

```
export class ChatModelFactory {
    static create(modelName: ModelName): ChatModel {
        const callbacks = [
            {
                handleLLMEnd: async (output: LLMResult) => {
                    const context = getContext();
                    if (!context) return;

                    const tokens = output.generations
                        .flat()
                        .map((gen) => gen.text.trim());

                    tokenLogger.log({
                        tokens,
                        finger: {
                            userId: context.userId,
                            containerId: context.containerId,
                            feature: context.feature,
                        },
                    });
                },
            },
        ];

        switch (modelName) {
            case ModelName.GPT_4:
            case ModelName.GPT_4O:
            case ModelName.GPT_3_5:
                return new ChatOpenAI({
                    modelName,
                    ...modelConfig,
                    callbacks,
                });

            // 생략
```

로깅 관련 정보를
context에서 가져올 수 있다.

종류별로 확장성있게
모델을 추가할 수 있다

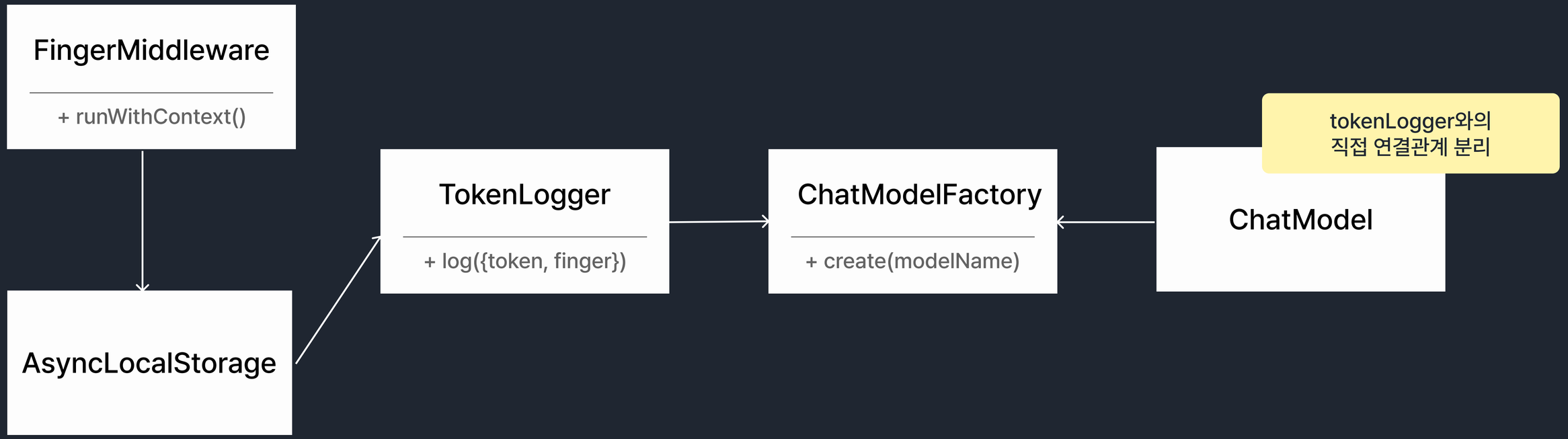
개선된 호출 부분

```
class WorkflowService implements LlmService {
    async generate() {
        const llm = ChatModelFactory.create(ModelName.GPT_4O);

        const response = await llm.call([
            { role: "system", content: "안녕하세요, 구름입니다." },
        ]);

        return response;
    }
}
```

더 이상 로깅을 비지니스 로직에 작성하지 않아도 된다.



참조 관계에 대한 구분을 명확히 책임을 나누도록 한다.