클라이언트에서 시작되는 실패를 “관측 가능”하게 만들기

토스 Learner's High 서버 2기 성장 일지 — 이승빈

이승빈 — Software Engineer

GitHub · https://github.com/sungbinlee/

Blog · https://tech.sungbinlee.dev/

LinkedIn · https://www.linkedin.com/in/sungbinlee0/

저는 Python을 중심으로, 여러 서비스가 맞물려 동작하는 MSA 환경에서 통합·운영 이슈를 해결해 온 소프트웨어 엔지니어입니다. 기능을 추가하는 것만큼 “서비스 간 연결이 왜 이렇게 동작하는지”를 끝까지 추적하는 데 관심이 많아, API 경계와 데이터 흐름, 장애 지점뿐 아니라 인증·인가(IAM)처럼 서비스 전체에 영향을 주는 기반 레이어까지 구조적으로 파악하는 것을 중요하게 생각합니다. 이런 맥락에서 시스템 구성과 통신 흐름을 먼저 다이어그램으로 정리해 팀이 같은 그림을 보게 만든 뒤, 구현과 개선으로 연결하는 방식으로 일해왔습니다.

이번 글은 러너스 하이에서 진행한 tk-incidentreporter 프로젝트 여정을 담았으며, 클라이언트에서 발생하는 실패를 운영 관점에서 관측 가능(Observability) 하게 만들기 위해 설계부터 MVP 구현 그리고 오픈소스화까지의 여정을 기록했습니다.

0. 들어가며 — 나는 어떤 사람이고, 어떤 방식으로 성장해 왔나

나는 컴퓨터 과학을 전공했다. 학부 시절 가장 재미있었던 과목은 **소프트웨어 엔지니어링**이었다. 해당 수업에서 2학기에 걸쳐 SRS(시스템 요구사항 명세서)를 작성하고, 다이어그램으로 구조를 설계한 뒤, Laravel로 실제 구현까지 엔드투엔드로 옮겨 보는 경험을 했다. 그 과정에서 나는 **"무작정 코드를 작성하는 것"**보다 먼저 **무엇을 만들지(요구사항)**를 정의하고, **어떤 구조로 만들지(설계)**를 정리한 다음 구현으로 가는 방식이 더 효율적이라는 걸 체감했다. 나는 그때 이후로, 기능을 만들든 시스템을 고치든 항상 같은 순서로 일을 시작했다. 먼저 목표를 한 문장으로 고정하고, 입력·출력을 정리한 뒤, 흐름을 다이어그램으로 그려 팀과 합의했다. 이렇게 의도를 먼저 공유해두면 구현이 흔들리지 않았고, 변경이 들어와도 영향 범위(사이드 이펙트)를 더 잘 예상할 수 있었다.

지금은 Python을 주로 사용하는 개발자로 일하고 있다. MSA 기반 시스템을 유지보수 하며, 동시에 클라이언트를 직접 만나 불확실한 요구사항을 정리하고 이를 바탕으로 설계·구현·납품까지 연결하는 일을 한다. 현장에서는 요구사항이 처음부터 완벽하게 정리되어 있는 경우가 거의 없었고, 말로만 공유된 요구는 구현 단계에서 쉽게 흔들린다. 그래서 나는 초반에 질문을 통해 기준을 구체화하고, 무엇을 할지와 무엇을 하지 않을지를 나누고, 최소 기능(MVP)의 범위를 먼저 잡는 방식으로 일을 진행해 왔다. 이번 러너스 하이 과정은 한 달 동안 하나의 주제에 몰입할 수 있는 기회였다. 하지만 그 시작점은 단순했다.

> ***"그래서… 한 달 동안 뭘 해야 하지?"***

이 글은 그 질문에서 출발해, 내가 한 달 동안 어떤 문제를 붙잡았고 어떤 흐름으로 정리하고 구현했는지를 기록한 **성장 일지**다.

목차

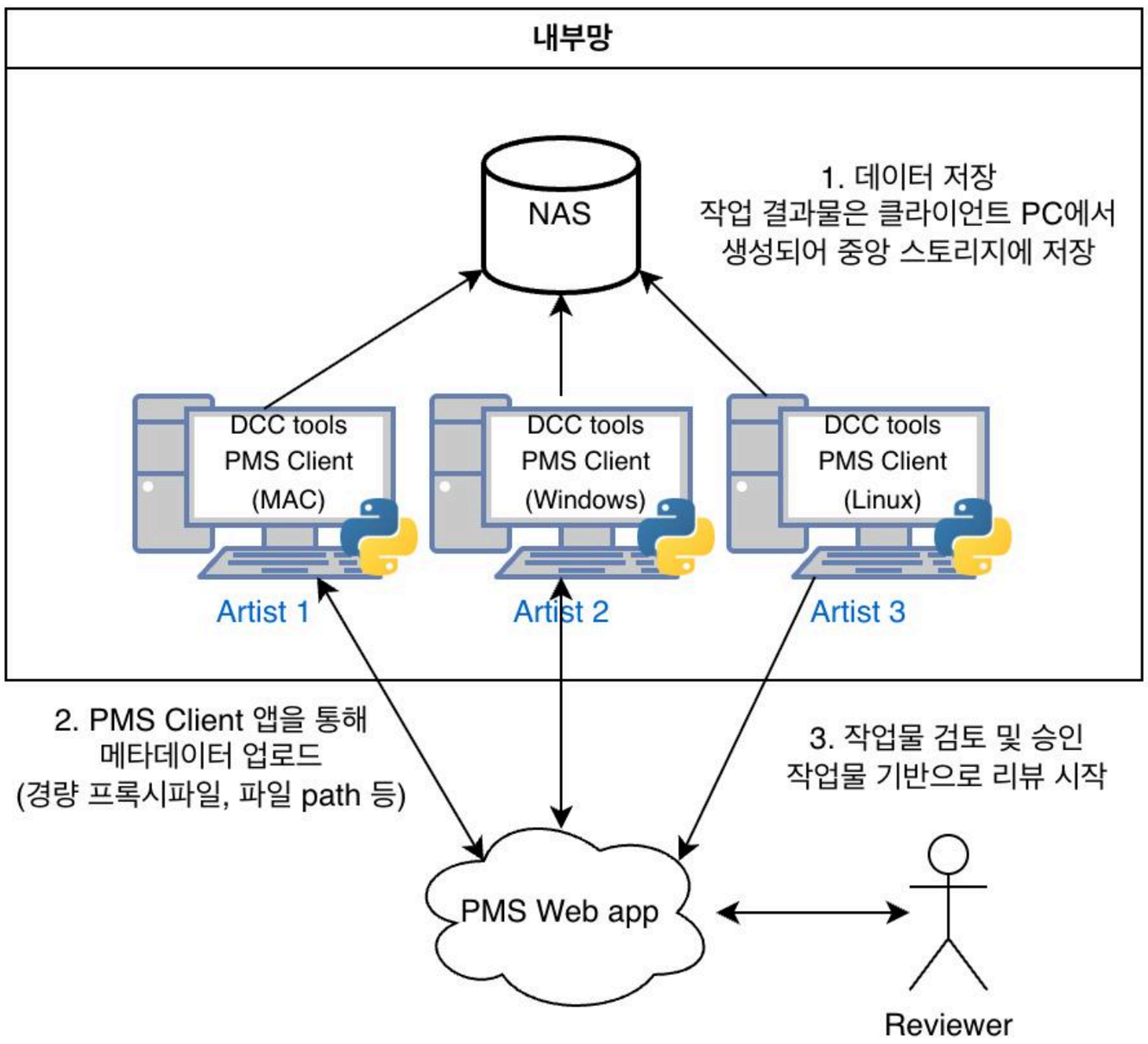
1장. 시작 — 한 달 동안 뭘 할 것인가

러너스 하이는 과제를 던져주지 않는다. “이거 해보세요” 같은 명확한 가이드도 없다. 그래서 시작은 늘 똑같았다. 무엇을 만들지(**What**)보다, 내가 왜 이걸 해야 하는지(**Why**)를 먼저 세워야 했다.
주제를 고를 때 ‘기술적으로 멋있어 보이는 것’을 먼저 떠올리면 한 달이 끝났을 때 남는 게 얇아질 가능성이 크다고 생각했다. 반대로, 실제 업무에서 **반복적으로 부딪혔던 문제**를 고르면 몰입의 핀이 더 잘 꽂힌다. 나는 아주 빨리 한 가지 장면으로 돌아갔다. 장애가 발생했을 때의 풍경이다. 특히 ‘**클라이언트 PC에서 시작되는 실패**’가 언제나 사람의 전달에 기대고 있다는 사실이 계속 불편했다. 그 불편함이 이번 한 달의 주제가 됐다.

> **1장 한줄 결론**
> 한 달의 주제는 “현장에서 반복되는 불편"에서 꺼내야 깊게 파고들 수 있다.

2장. 1주차 — 현실 관찰: 문제는 항상 클라이언트 PC 에서 시작된다

현재 재직 중인 회사에서는 PMS(Project Management System)를 기반으로 미디어 & 엔터테인먼트 제작 파이프라인을 주력으로 구축하고 있다. 아래는 제작 파이프라인 작업 공정을 최대한 단순화한 도식이다.



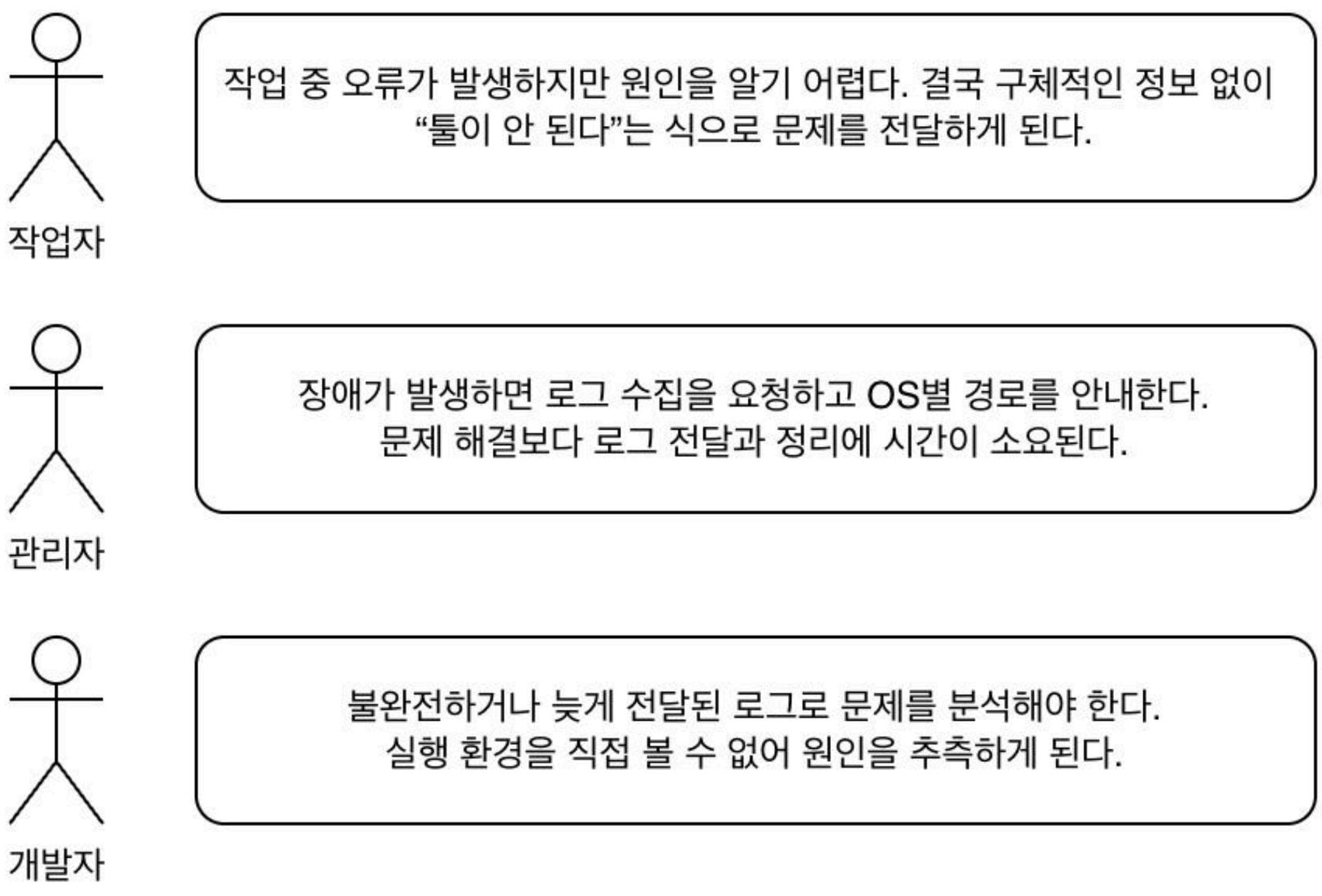
실제 작업은 작업자의 로컬 PC에서 이루어진다. DCC툴(Digital Content Creation Tool)을 통해 파일을 생성하고, 렌더링하고, 결과물을 만든다. 이렇게 생성된 작업 결과물은 클라우드로 직접 업로드되지 않고, 회사 내부의 중앙 스토리지(NAS)에 저장된다. 각 작업자 PC는 운영체제가 다르더라도 사전에 약속된 마운트 포인트를 통해 동일한 파일 경로 구조를 공유한다. PMS 서버로 전달되는 것은 파일 자체가 아니라, 어디에 어떤 결과물이 생성되었는지에 대한 정보(메타데이터) 와 리뷰용 미디어파일이다.

로그는 왜 항상 클라이언트 PC에 남는가?

위 구조를 이해하고 나면, 로그가 왜 서버가 아니라 클라이언트 PC에 남을 수밖에 없는지도 자연스럽다. 작업의 주체가 클라이언트 PC이기 때문이다. 연산, 파일 I/O, DCC 툴 연동 모두 로컬 프로세스에서 수행된다. 따라서 에러와 예외, 실패에 대한 정보 역시 각 작업자 PC의 로그 파일로 남게 된다. 문제는 이 로그들이 각 OS, 각 PC에 흩어져 있다는 점이다.

문제가 발생했을 때 실제로 벌어지는 일

이 환경에서 장애는 대부분 클라이언트 PC에서 시작되지만, 그 사실을 시스템적으로 인지할 방법은 없어서 결국 문제는 사람을 통해 전달된다.



- 작업자는 “툴이 안 돼요”만 남기기 쉽다. **(정보 부족)**
- 관리자는 로그 경로를 안내하고 회신을 기다린다. **(대기/전달 비용)**
- 개발자는 로그로만 상황을 재구성한다. **(추측 비용)**

이 흐름에서 가장 큰 문제는 구조 자체에 있다. 실행은 클라이언트에서 발생하지만, 관측은 시스템적으로 이루어지지 않고, 그 사이를 사람이 메우고 있다. 이 때문에 문제는 항상 늦게 발견되고, 대응까지의 시간이 걸린다.

이 패턴은 작년에 진행한 C사의 미디어 제작공정 파이프라인 구축 사업의 PoC 단계 에서도 이미 여러 번 반복되었고, **작업자 → 관리자 → 개발자**로 이어지는 사람 중심의 커뮤니케이션 구조 자체가 문제 해결의 병목이 되고 있었다.

이쯤에서 자연스럽게 이런 질문이 들었다.

서버에서는 너무 당연한 Observability를 왜 클라이언트 쪽에는 거의 적용하지 못하고 있을까?

로그는 존재한다. 다만 관측 가능하지 않은 상태로 흩어져 있을 뿐이다.

> **2장 한줄 결론**
>
> 실패는 클라이언트에서 시작되는데, 그 실패를 “시스템이” 바로 관측하지 못해서 사람이 병목이 되고 있었다.

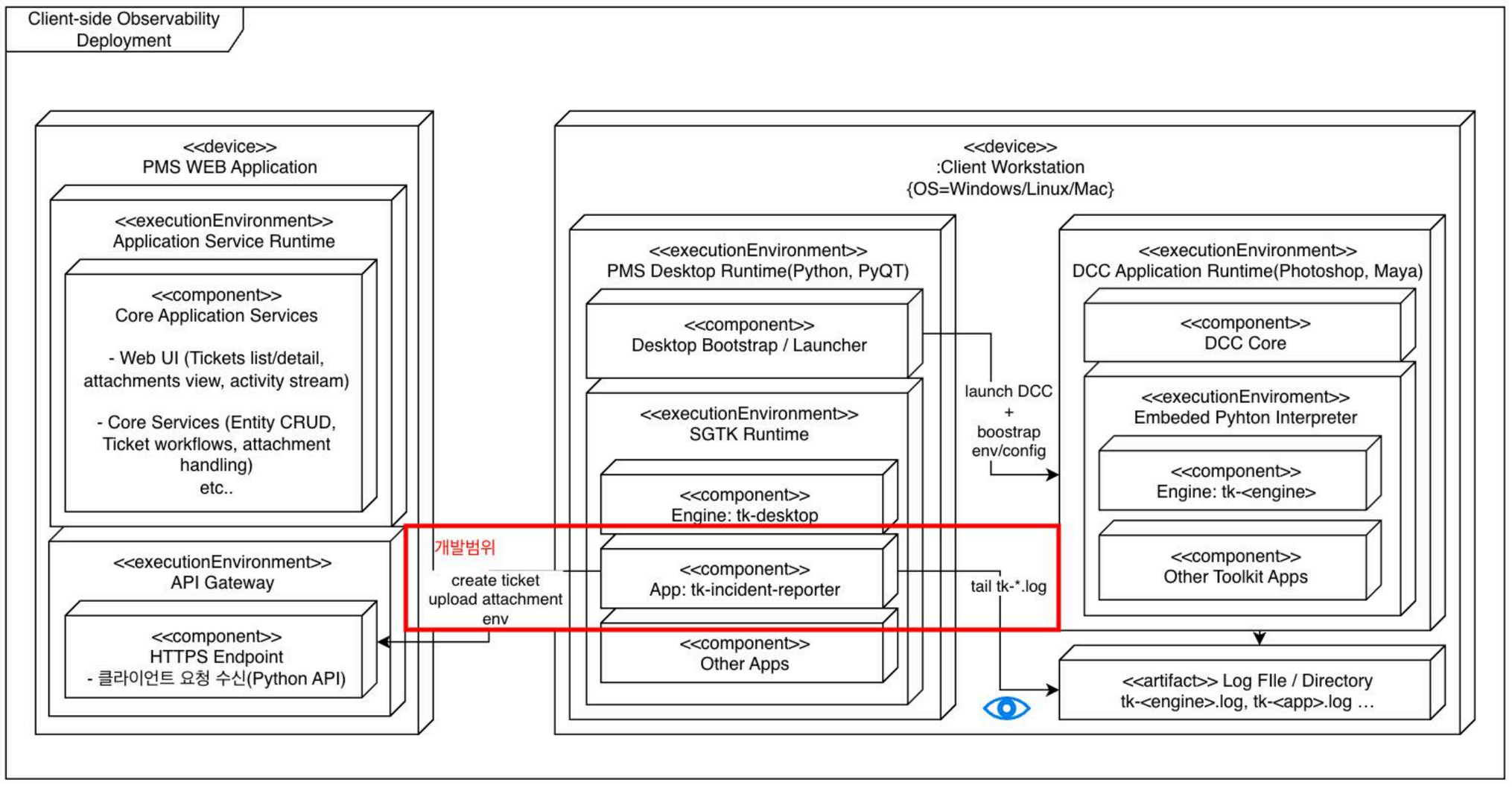
3장. 2주차 — 설계: 트리거 기반 로그 수집과 티켓 생성 흐름

아래 이미지는 배포 다이어그램으로, 클라이언트(engine/sgtk/DCC)와 중앙 PMS 관계, 그리고 내가 개발할 범위(tk-incidentreporter)를 보여준다.

3.1 클라이언트 애플리케이션 구조

현재 사용하는 PMS 클라이언트 애플리케이션은 **Python** 기반의 Mac / Windows / Linux 환경에서 동작하는 **크로스 플랫폼 네이티브 데스크탑 앱**이다.

이 앱은 단일 실행 파일이라기보다는, 각 DCC툴(Photoshop, Maya 등, 2D 3D디자인 툴)을 실행시켜주는 런쳐 같은 역할을 하며 부트스트랩 시 클라이언트 앱이 플러그인 형태로 탑재되어 동작한다. DCC 툴 내부에는 Python 인터프리터가 포함되어 있고, 상호작용도 이 Python 레이어를 통해 이루어진다. 이 구조 때문에 로그는 다음과 같은 특성을 가진다.



- 로그는 서버가 아니라 **클라이언트 PC 로컬 디스크**에 남는다.
- OS마다 경로는 다르지만, **특정 로그 디렉터리**에 모인다.
- 로그는 **엔진(실행환경) 단위**로 생성된다.

각 엔진은 독립적인 Python 실행 환경을 가지고 있고, 그 안에서 플러그인이 동작하며 로그를 남긴다.
그래서 로그 파일도 보통 tk-<engine>.log 혹은 tk-<app>.log 형태로 생성된다.
결과적으로 하나의 클라이언트 PC 안에는 여러 엔진/앱들의 로그 파일이 동시에 존재한다.

3.2 로그가 관측되지 않는 지점

로그 자체는 항상 남고 있었다. 문제는 로그가 존재하지 않는 것이 아니라, 문제가 발생했을 때 이 로그와 클라이언트 실행 환경 정보를 중앙에서 바로 관측하고, 운영 단위의 액션으로 연결할 수 있는 구조가 아니라는 점이었다.

- 작업 중 오류가 발생하면 로그는 남는다.
- 하지만 그 로그와 실행 환경 정보는 **작업자 PC 안에만 머문다.**
- 중앙에서는 문제 발생 자체를 **즉시 인지하기 어렵다.**

이 구조에서는 에러의 원인을 분석하기 이전에, **“어디서, 어떤 환경에서, 어떤 문제가 발생했는지”**를 파악하는 단계에서부터 지연이 발생한다. 그래서 단순한 로그 수집이 아니라, 수집과 동시에 운영 흐름의 시작점으로 보는 방향으로 구조를 잡아봤다.

3.3 로그 수집 및 티켓 생성 설계

기존 구상안: 로그를 일정 기준으로 모아서 전달

클라이언트에서 로그를 주기적으로 모아서 서버로 전달하는 방식이다. 구현은 단순하지만

- 대부분의 로그는 정상 동작 로그고
- 실제 문제가 없는 로그까지 함께 수집된다.
- PMS 서버 쪽 저장 용량과 운영 비용이 불필요하게 증가한다.

운영 관점에서 모든 로그를 상시 수집하는 구조는 목적 대비 과하다고 판단했다.

설계안 에러 발생을 트리거로 수집과 티켓 생성을 묶기

그래서 설계의 중심으로 잡은 방식은 에러 발생을 트리거로 로그 수집과 티켓 생성을 하나의 흐름으로 묶는 구조다.

이 설계에서는:

- 클라이언트에서 로그 파일을 **상시 참조**하고
- 에러 패턴이 감지되면
  - 문제가 발생한 클라이언트를 식별하고
  - 해당 로그 파일과 실행 환경 정보와 함께 **내부 티켓을 자동으로 생성**한다.

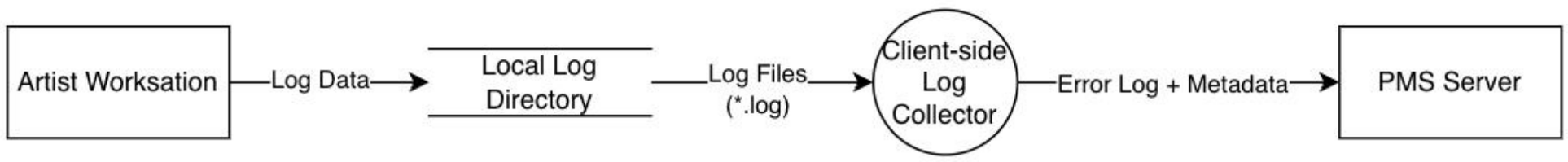
이렇게 하면:

- 로그 수집 자체가 운영 이벤트가 되고
- 별도의 수동 보고 없이도
- **장애 인지부터 대응까지 하나의 흐름**으로 이어진다.

관리자는 티켓을 통해, **“이 클라이언트에서 이 시점에 이 문제가 발생했다”**는 운영 단위의 명확한 상태 표현이 된다.

3.4 로그 데이터 흐름 정리

위 내용을 바탕으로 데이터 흐름도를 간략히 작성해보았다.



- 로그는 클라이언트 환경에서 생성되고
- 로컬 로그 디렉터리에 쌓이며
- 클라이언트 수집기가 이를 읽어
- 에러 로그와 실행 환경 정보가 중앙 시스템(티켓)으로 전달된다.

> **3장 한줄 결론**
>
> **"상시 수집"이 아니라 "에러 트리거"로, 관측과 운영 액션(티켓 생성)을 하나로 묶는 흐름을 선택 했다.**

4장. 3주차 — 구현(MVP): 엔드투엔드 검증

3주차에는 설계(Deployment Diagram)에서 정한 범위를 실제로 구현해 보고, 에러 감지 → 로그 묶음 → 티켓 생성까지의 최소 동작을 확인했다. 구현은 Desktop client(sgtk) 환경 전용으로 백그라운드 앱 tk-incidentreporter 형태로 만들었다.

레포지토리 링크: https://github.com/sungbinlee/tk-incidentreporter

> sgtk는 Desktop 개발을 위한 환경(프레임워크)이며 오픈소스로 공개되어 있다.

4.1 워크플로우 요약

- 멀티 OS(Windows/macOS/Linux)에서 공통 방식으로 지정 로그 경로 하위의 tk-*.log를 스캔하고 tail/follow 한다.
  - Windows: %APPDATA%\Shotgun\logs\
  - Linux: ~/.shotgun/logs/
  - Mac: ~/Library/Logs/Shotgun/
- 로그 레벨 중 CRITICAL/ERROR를 감지하면, 엔진 단위로 로그를 묶고 실행 환경 정보를 수집한 뒤 PMS에 티켓을 자동 생성한다.

4.2 레포지토리 핵심 파일/모듈

```
tk-incidentreporter/
├─ app.py, info.yml              # 진입점 및 매니페스트
└─ python
   └─ tk_incident
      ├─ bootstrap.py            # 설정 로드 및 초기화
      ├─ agent.py                # 워커 시작·관리
      ├─ matcher.py              # 로그 패턴 매칭(에러 감지)
      ├─ singleton.py            # 싱글톤
      ├─ tail_worker.py          # 로그 파일 tail 및 롤오버 처리
      ├─ uploader.py             # 티켓 생성·첨부 업로드·중복 제어
      └─ utils.py                # 공용 유틸 함수
```

4.3 개발하면서 고려했던 점들

1) 싱글톤 (중복 실행 방지)

같은 머신에서 Desktop 앱이 여러 개 띄워질 수도 있고, 개발 중 에이전트를 여러 번 실행해 버리는 상황이 발생했다. 그래서 에이전트가 한 프로세스만 동작하도록 싱글톤을 넣었다.

처음엔 파일락을 생각했지만, Desktop 앱이 Qt 기반이어서 구현은 Qt의 로컬 IPC인 QLocalServer를 사용했다. 로컬 소켓을 이용하면 플랫폼에 관계없이 간단하고 안전하게 이름(name)으로 락을 잡을 수 있다.

```python
# singleton.py (요약본) — QLocalServer 기반 싱글톤 락 (중복 실행 방지)

class SingletonLock:
    def __init__(self, name, parent=None):
        self.name, self.parent, self._server = name, parent, None

    def acquire(self, timeout_ms=150):
        sock = QLocalSocket(self.parent)
        sock.connectToServer(self.name)
        if sock.waitForConnected(timeout_ms):
            return False  # already running
```

```
        server = QLocalServer(self.parent)
        if not server.listen(self.name):
            try:
                QLocalServer.removeServer(self.name)  # stale cleanup
            except Exception:
                pass
            if not server.listen(self.name):
                return False

        self._server = server
        return True

    def release(self):
        if self._server:
            try:
                name = self._server.serverName()
                self._server.close()
                QLocalServer.removeServer(name)
            except Exception:
                pass
            self._server = None
```

2) 로그 tail + 롤오버 처리

로그 시스템은 파일 rotate나 truncate가 발생할 수 있으므로, 각 파일에 대해 바이트 오프셋을 유지하고 inode와 파일 크기를 비교해 안전하게 follow 하도록 했다.

```
# tail.py (요약본) — rotate/truncate

class TailWorker(QThread):
    signal = Signal(dict)  # {path, line, pos, ts}

    def __init__(self, folder, patterns=("tk-*.log",), poll=0.5):
        self.folder, self.patterns, self.poll = Path(folder), patterns, poll
        self.state = {}  # path -> {"pos": int, "inode": any}

    def _register(self, p):
        st = p.stat()
        self.state.setdefault(str(p), {"pos": st.st_size, "inode": st.st_ino})

    def _tick(self):
        for pat in self.patterns:
            for p in self.folder.glob(pat):
                self._register(p)
```

```
        for path, s in list(self.state.items()):
            p, st = Path(path), Path(path).stat()

            if s["inode"] != st.st_ino:          # rotate
                s["pos"], s["inode"] = 0, st.st_ino
            if st.st_size < s["pos"]:             # truncate
                s["pos"] = 0

            with open(path, "r", errors="ignore") as f:
                f.seek(s["pos"])
                for line in f:
                    s["pos"] = f.tell()
                    self.signal.emit({"path": path, "line": line.rstrip(), "pos":
s["pos"], "ts": time.time()})

    def run(self):
        while not self.isInterruptionRequested():
            self._tick()
            time.sleep(self.poll)
```

3) Uploader: 티켓 폭주 방지(중복 방지 · 쓰로틀링)

개발중에 가장 골치였던 건 동일 오류가 반복 발생할 때 티켓이 폭주하는 문제였다. 클라이언트는 같은 머신에서 앱이 계속 실행되기 때문에, 버그가 수정된 버전이 배포되기 전까지 동일 오류가 반복 발생하며 티켓이 폭주할 수 있다. 그래서 Uploader는 “무조건 남기는 것”이 아니라 “폭주를 막는 것”을 우선으로 설계했다.

- 감지 기준: 로그에서 ERROR|CRITICAL 라인을 정규식으로 감지
- 폭주 원인: 동일 머신에서 같은 오류가 짧은 시간 안에 반복 발생(배포 전까지 지속)
- 해결 방식: 티켓 제목을 시그니처로 고정해 중복을 묶음
  - 시그니처 형식: "<user_login> - <ErrorName or short message>"
  - 동작: 티켓 생성 전에 PMS에서 동일 제목 티켓을 조회 → 존재하면 새 티켓 생성하지 않음
- 효과: **동일 사용자·동일 오류는 하나의 티켓으로만 유지되어 중복 폭주를 방지**

업로드 플로우를 요약하면 다음과 같다.

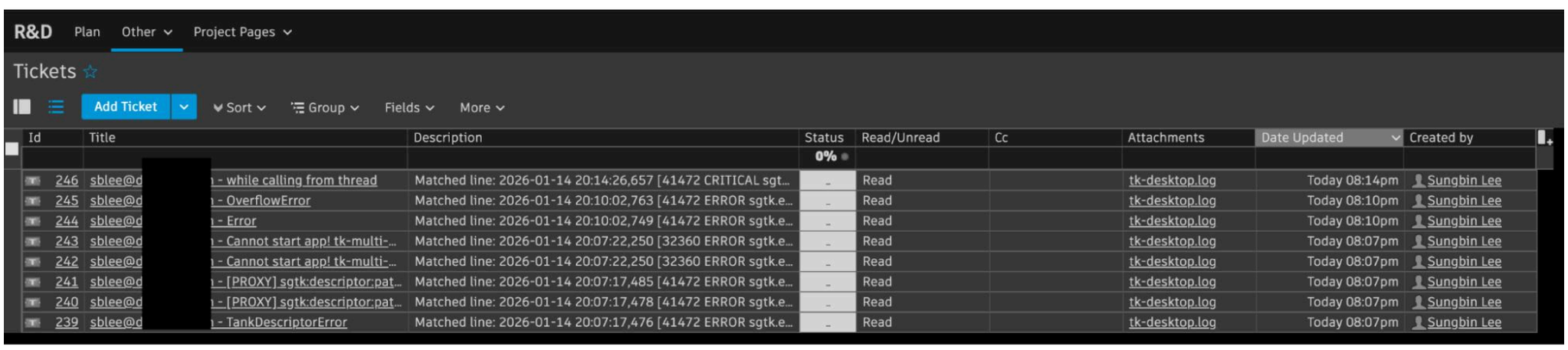
```python
# uploader.py (요약본) — upload_log 플로우(중복 방지 → 생성 → 첨부)

def upload_log(self, log_path, pos, trigger, snippet=None):
    # 1) title 시그니처 생성: "<user_login> - <ErrorName|short message>"
    title = self._make_title_signature(trigger.get("matched_line", ""))

    # 2) 동일 title 티켓 존재 여부 조회 → 있으면 생성 스킵(폭주 방지)
    if self._find_ticket_by_title(title):
        return True

    # 3) 새 티켓 생성 (project + title + description)
    created = self._sg.create(self.ticket_entity_type, {
        "project": {"type": "Project", "id": self.project_id},
        "title": title,
        "description": self._build_description(log_path, pos, trigger, title),
    })

    # 4) 생성된 티켓에 로그 첨부 (retry/backoff는 내부 처리)
    return self._attach_log_to_ticket(created["id"], log_path)
```

4.5 테스트

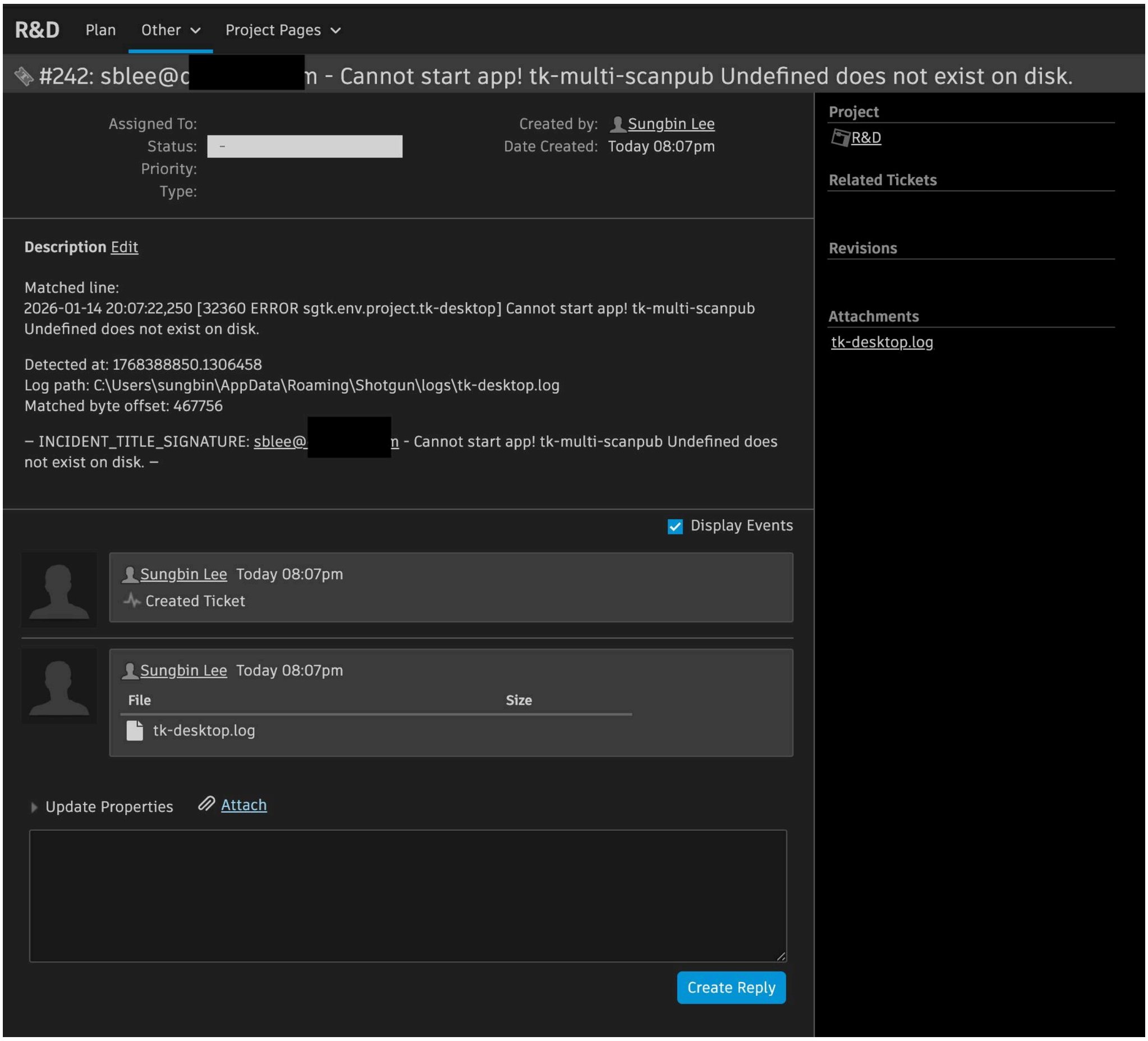
- Desktop에서 tk-incidentreporter 앱을 로드한 뒤, tk-desktop.log / tk-maya.log에 ERROR 라인을 추가 → Agent가 감지하고 티켓을 생성함을 확인했다.
- 동일 에러를 여러 번 발생시키는 시나리오에서 동일 제목의 티켓이 한 건만 생성된다.
- 로그 파일 rotate / truncate 상황에서 재열거·재오픈하여 새 로그를 추적하는 것도 확인했다.

실제 PMS에 올라간 모습(현장 검증)

구현 후 가장 체감된 변화는 별도의 로그 요청 없이도 문제가 발생하면 자동으로 티켓이 생성되어 운영자가 바로 확인할 수 있게 된 점이다. 테스트 환경에서 로그를 트리거로 여러 건의 티켓이 PMS에 자동 생성되는 것을 확인했다. 각 티켓에는 감지된 매치 라인과 탐지 시각, 원본 로그 파일(tk-desktop.log)이 첨부되어 있어, 별도 로그 요청이나 전달 과정 없이도 즉시 원인 확인과 우선순위 판단이 가능했다.



스크린샷1 — 티켓 목록: 동일한 사용자명-에러명 패턴으로 생성된 티켓들이 목록에 쌓여 있다.



스크린샷2 — 티켓 본문: 더 자세한 내용을 확인할 수 있다.

이렇게 되면 별도 로그 요청 없이도 바로 로그를 열어 원인 추적을 시작할 수 있다.

- **운영적 효과:** 작업자가 버그리포트를 하고, 담당자가 **“로그 어디 있나요?”**를 묻고 회신을 기다리는 과정이 사라졌다. 덕분에 **문제 인지부터 대응 착수까지의 시간이 단축**되고, 로그 수집·정리로 인한 운영 부담이 눈에 띄게 줄 것으로 기대된다.

- **자동 생성된 티켓 패턴:** 여러 건의 티켓이 동일한 사용자명-에러명 패턴으로 생성되는 것을 확인했고, 현재 설계는 이 패턴을 이용해 **중복 티켓 폭주를 방지**하고 있다.
- **감지 → 시그니처 생성(중복 회피) → 수집/첨부 → 티켓 생성/기록**
  이 흐름이 안정적으로 돌아가면, 이후에는 세부적인 운영 정책 수립과 품질 개선에 집중할 수 있을 것으로 기대된다.

4장 한줄 결론

MVP로 "감지→수집→티켓" 엔드투엔드를 확인했고, 운영 관점에서 가장 중요한 문제는 티켓 폭주였고 이를 중복 방지(시그니처)로 제어했다.

5장. 4주차 — 오픈소스화, 커뮤니티 공유

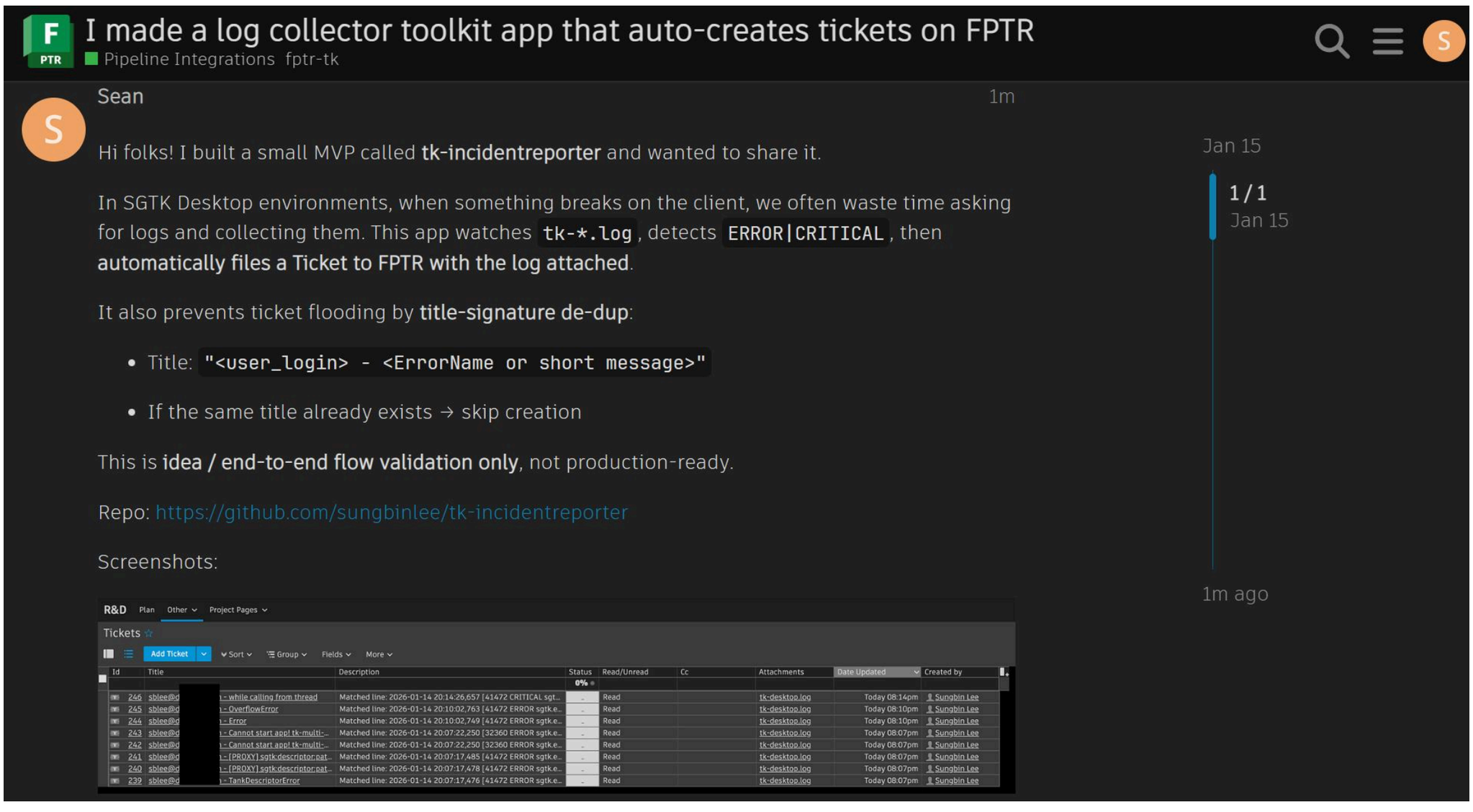
4주차 에는 결과물을 **오픈소스로 공개하고 커뮤니티에 공유**하는 데 초점을 맞췄다.

이 결정을 한 이유는 단순히 "**오픈소스가 좋다**"가 아니었다.

- 기반 생태계가 오픈소스이고, 업무를 하면서 커뮤니티에 공개된 코드들이 많이 도움이 되었다. 나 또한 가능한 선에서 **생태계에 계속 기여**해 왔다.
- 현실적으로 봤을 때 지금 상태로는 당장 프로덕션에 적용하기 어려웠다.
- 그래서 더더욱 **"내 환경 밖"에서 의견이 필요**했다. 기능을 더 키우기 전에 방향을 확인하고 싶었다.

커뮤니티에는 이렇게 공유했다.

"저느 평소에 이게 불편했어요. 장애가 나면 결국 로그를 모으고 전달하고 정리하는 데 시간이 다 쓰이더라고요. 그래서 자동으로 수집해서 티켓까지 남기는 도구를 한 번 만들어봤습니다."



https://community.shotgridsoftware.com/t/i-made-a-log-collector-toolkit-app-that-auto-creates-tickets-on-fptr/20614

공개 범위는 현실적으로 조절했다. 아직 프로덕션에 적용하기엔 리스크가 크고 다른 사람도 읽고 개선할 수 있는 최소 단위로 만드는 게 목표였다.

- 코어: 로그 스캔/추적, 에러 감지, 엔진 단위 묶음 생성, 실행 환경 정보 수집
- 연동: 티켓 생성/첨부 업로드는 최소 동작 위주로, 교체 가능한 인터페이스로 분리
- 문서: 설치/설정, 기능, 기여 가이드

공개하면서 가장 신경 쓴 부분은 **"내 환경에서만 돌아가는 코드"**가 되지 않게 하는 것이었다. 그래서 README에 설정 및 구성 예시 그리고 제한 사항을 최대한 명시했다.

> **5장 한줄 결론**
>
> **완성보다 먼저 오픈소스로 공유하고, 커뮤니티의 피드백으로 다음 방향을 배우고 다듬고 싶었다.**

6장. 맺음말 — 아쉬움과 지속할 약속

이번 한 달이 계획대로 흘러가지는 않았다. 최대한 시간을 내고 싶었지만, 12월 말 보안 취약점(몽고 블리드) 이슈가 터지면서 내가 맡고 있는 시스템도 영향권에 들어갔다. 사이드 이펙트를 파악하고 업그레이드 계획을 세우고, 배포 스크립트 작성하는 등 긴급하게 조치가 이루어져야 해서 한 주 정도는 러너스 하이에 제대로 시간을 쓰지 못했다.

원래는 오픈소스 공개에 그치지 않고, 가능하면 프로덕션에 일부라도 적용해 운영 관점의 피드백까지 성장 일지에 담고 싶었는데 그 단계까지는 못 간 게 아쉽다.

그럼에도 한달이라는 기간 동안 남은 건 분명하다.

- **현장에서 반복되는 불편을 '느낌'이 아니라 '구조'로 관찰했고**
- **불편했던 점(병목)을 설명 가능한 형태로 문제 정의했으며**
- **최소 기능으로 엔드투엔드 흐름을 구현해 검증했다.**
- **그리고 공개 가능한 형태로 정리해 밖으로 내놓았다.**

나는 앞으로도 문제를 '**느낌**'이 아니라 '**정의**'로 바꾸고 '**해결하는 사람**'으로 성장해 나가고 싶다.

더 보면 좋은 글

- 1주 차 회고: https://tech.sungbinlee.dev/journey/토스-러너스하이2기-1주차
- 2주 차 회고: https://tech.sungbinlee.dev/journey/토스-러너스하이2기-2주차
- 3주 차 회고: https://tech.sungbinlee.dev/journey/토스-러너스하이2기-3주차
- 4주 차 회고: https://tech.sungbinlee.dev/journey/토스-러너스하이2기-4주차
- 레포지토리 링크: https://github.com/sungbinlee/tk-incidentreporter
- 오픈소스 커뮤니티 글: https://community.shotgridsoftware.com/t/i-made-a-log-collector-toolkit-app-that-auto-creates-tickets-on-fptr/20614

본 성장 일지는 위 글들을 바탕으로 작성되었습니다.