

## DMS 시스템 리팩토링 - 비동기 처리 및 통합 오류 (수정 문서)

사용자: Google AI 스튜디오 대화 내용인데 원래 확장자가 없는 것을 텍스트 확장자를 붙여서 열 수 있도록 해놓은 건데 문제는 원래 확장자가 없어서 그런 건지 글자가 전부 붙어 있고 가독성이 엄청 안 좋게 나왔어. 문제를 해결한 수정 문서를 작성해줘.

모델:

로그 파일 분석을 통해 확인된 코드의 문제점은 다음과 같이 요약할 수 있습니다. 시스템이 대규모 리팩토링(재구성) 과정에 있어 새로운 모듈과 기존 모듈 간의 인터페이스 불일치 및 비동기 처리 관련 오류가 핵심 원인입니다.

### 주요 문제점 분석

**1. 비동기(Async) 처리 오류: TypeError 및 RuntimeError** 가장 빈번하게 발생한 오류는 비동기 함수 호출 문제입니다. 이는 크게 두 가지 유형으로 나타났습니다.

- 비동기 함수를 동기적으로 호출 (**coroutine was never awaited**):
  - 문제점: **async def**로 정의된 함수는 반드시 **await** 키워드로 호출해야 합니다. 그러나 로그(**dms\_20250710\_174818.log**)에서 **object** 프로세서 실행 중 오류: **An asyncio.Future, a coroutine or an awaitable is required**와 같은 메시지는 **await** 없이 비동기 함수를 호출했거나, 반대로 동기 함수를 **await**하려고 할 때 발생합니다. **analysis/orchestrator/orchestrator\_advanced.py**에서 각 프로세서의 **process\_data**를 호출하는 부분에서 이 문제가 발생한 것으로 보입니다.
  - 원인: 리팩토링 과정에서 일부 프로세서는 비동기(**async def**)로, 다른 일부는 동기(**def**)로 구현되어 있어 호출부(Orchestrator)에서 이를 일관되게 처리하지 못했습니다.
- 비동기 컴포넌트 초기화 실패:
  - 문제점: **app.py**의 **initialize** 메서드에서 **PersonalizationEngine**과 같은 비동기 컴포넌트들의 초기화가 실패했습니다. (**dms\_20250710\_174818.log**의 **ERROR** - 비동기 컴포넌트 초기화 실패: **'PersonalizationEngine' object has no attribute 'initialize'**)
  - 원인: **PersonalizationEngine** 클래스에 비동기 초기화를 위한 **initialize** 메서드가 구현되지 않았거나, **DMSApp** 클래스에서 잘못 호출하고 있었습니다. 이는 새로운 비동기 아키텍처로 전환하는 과정에서 발생한 전형적인 통합 문제입니다.

**2. 메서드 인자 불일치 오류: TypeError** 함수를 호출할 때 필요한 인자를 제대로 전달하지 않아 발생하는 문제입니다.

- 문제점: **FaceDataProcessor.process\_data()** 메서드가 **timestamp** 인자 없이 호출되어 **missing 1 required positional argument: 'timestamp'** 오류가 반복적으로 발생했습니다. (**dms\_20250710\_174818.log**)
- 원인: **analysis/orchestrator/orchestrator\_advanced.py**에서 **FaceDataProcessor**의 **process\_data**를 호출할 때 필요한 인자(**data, image, timestamp**)를 모두 전달하지 않았습니다. 이는 메서드의 시그니처(정의)가 변경되었으나, 호출하는 쪽의 코드가 업데이트되지 않았기 때문에 발생한 문제입니다.

**3. 초기화 순서 및 의존성 문제: AttributeError** 클래스 인스턴스가 생성될 때 필요한 다른 객체가 아직 준비되지 않아 발생하는 문제입니다.

- 문제점: **EnhancedMediaPipeManager** 초기화 시점에 **analysis\_engine**이 제대로 설정되지 않아 **AttributeError: 'NoneType' object has no attribute 'on\_face\_result'** 오류가 발생했습니다. (**dms\_20250710\_171047.log**)
- 원인: **app.py**에서 **EnhancedMediaPipeManager**를 생성할 때 분석

엔진(`analysis_engine`)을 전달해야 하지만, 이 시점에는 아직 완전한 분석 엔진이 준비되지 않았습니다. 이로 인해 콜백 함수를 등록하지 못했고, 후속 작업에서 `None` 객체를 참조하는 오류로 이어졌습니다.

결론: 리팩토링 과정에서 발생한 전형적인 통합 문제

로그 파일들을 종합적으로 분석한 결과, 현재 **DMS** 시스템은 모놀리식(**Monolithic**) 구조에서 모듈화된 비동기 아키텍처(**S-Class**)로 전환하는 과도기에 있으며, 이 과정에서 발생하는 전형적인 통합 문제들을 겪고 있습니다.

문제의 근원은 다음과 같이 요약할 수 있습니다.

- 인터페이스 불일치: 새로 설계된 모듈과 기존 모듈 간의 함수 이름, 인자 등이 통일되지 않았습니다.
- 비동기 처리 미숙: `async/await` 사용법이 일부 코드에서 올바르게 않아 프로그램 흐름이 멈추거나 오류가 발생했습니다.
- 의존성 관리: 각 모듈이 필요로 하는 다른 모듈이 제때에 초기화되지 않아 발생하는 문제가 있었습니다.

이러한 문제들은 `DMS_DEBUG_PATCH_HISTORY.md` 파일에도 기록된 바와 같이, 개발팀이 인지하고 수정해나가는 과정에 있는 것으로 보입니다. 현재 시스템은 새로운 **S-Class** 아키텍처의 잠재력을 최대한 활용하기 위해 각 모듈 간의 연결을 정교하게 다듬는 단계에 있습니다.

사용자: 어디를 어떻게 수정해야 되는지 알려줘.

모델:

네, 로그 파일을 분석한 결과 명확한 문제점들과 해결 방안이 보입니다. 핵심 원인은 리팩토링 과정에서 발생한 비동기 처리 방식의 불일치와 모듈 간의 인터페이스 변경 미반영입니다.

아래에 문제점별로 수정해야 할 파일과 코드를 정확히 안내해 드리겠습니다.

## 문제 해결 가이드

### 문제 1: 비동기 컴포넌트 초기화 오류 (**TypeError** 및 **RuntimeError**)

로그 파일(`dms_20250710_174818.log`)에서 `PersonalizationEngine`, `DynamicAnalysisEngine`에 `initialize` 메서드가 없거나 비동기적으로 잘못 호출되어 오류가 발생했습니다.

#### 1. `systems/personalization.py` 수정

`PersonalizationEngine` 클래스에 비동기 `initialize` 메서드를 추가합니다.

```
# systems/personalization.py

# ... (기존 코드 생략) ...
import aiofiles # aiofiles 임포트 확인

class PersonalizationEngine:
    """개인화 학습 엔진"""

    def __init__(self, user_id: str):
        # ... (기존 __init__ 코드) ...
        self._load_user_profile()

    # 🟡 [수정] 아래 async initialize 메서드를 추가합니다.
    async def initialize(self):
        """
```

```

        비동기 초기화: 사용자 프로필 파일이 없으면 생성합니다.
        """
        profile_path = Path("profiles") /
f"{self.user_id}_profile.json"
        if not profile_path.exists():
            await self._async_save_profile()
        logger.info(f"PersonalizationEngine for {self.user_id}
initialized.")

        # ... (기존 나머지 코드) ...

```

## 2. systems/dynamic.py 수정

DynamicAnalysisEngine 클래스에도 비동기 initialize 메서드를 구현합니다.

# systems/dynamic.py

# ... (기존 코드 생략) ...

```

class DynamicAnalysisEngine:
    # ... (기존 __init__ 코드) ...

    # ▼ [수정] 아래 async initialize 메서드를 추가합니다.
    async def initialize(self):
        """비동기 초기화: 상태를 리셋합니다."""
        self.reset()
        logger.info("DynamicAnalysisEngine for async initialized.")

    def reset(self):
        # ... (기존 코드) ...

```

## 3. app.py 수정

DMSApp 클래스의 \_initialize\_async\_components 메서드를 수정하여 각 컴포넌트의 initialize를 올바르게 비동기로 실행합니다.

# app.py

# ... (기존 코드 생략) ...

```

class DMSApp:
    # ... (기존 __init__ 코드) ...

    # ▼ [수정] _initialize_async_components 메서드를 아래와 같이
    변경합니다.
    async def _initialize_async_components(self):
        """비동기 컴포넌트들을 병렬로 초기화합니다."""
        try:
            # 모든 초기화 코루틴을 리스트에 담습니다.
            init_tasks = [
                self.integrated_system.initialize(),
                self.personalization_engine.initialize(),

```

```

        self.dynamic_analysis.initialize()
    ]
    # asyncio.gather를 사용하여 모든 초기화를 병렬로 실행하고
    기다립니다.
    await asyncio.gather(*init_tasks)
    logger.info("✅ 모든 S-Class 비동기 컴포넌트 초기화 완료")
except Exception as e:
    logger.error(f"비동기 컴포넌트 초기화 실패: {e}",
exc_info=True)
    raise # 오류 발생 시 상위로 전파하여 처리

async def main_async_loop(self):
    # ... (기존 코드) ...

```

## 문제 2: 메서드 인자 불일치 (TypeError)

로그(dms\_20250710\_174818.log)에서 FaceDataProcessor.process\_data() 호출 시 timestamp 인자가 누락되어 오류가 발생했습니다.

파일: analysis/orchestrator/orchestrator\_advanced.py

\_execute\_processor\_with\_monitoring 메서드에서 face 프로세서를 호출하는 부분을 수정하여 image와 timestamp 인자를 모두 전달하도록 합니다.

# analysis/orchestrator/orchestrator\_advanced.py

# ... (기존 코드 생략) ...

```

async def _execute_processor_with_monitoring(
    self, name: str, processor: Any, data: Any, timestamp: float,
mediapipe_results: dict = None
) -> Tuple[bool, Dict[str, Any]]:
    # ... (기존 코드 생략) ...
    try:
        # 🟡 [수정] face 프로세서 호출 부분을 아래와 같이 변경합니다.
        if name == 'face':
            image = None
            # mediapipe_results가 있고, 그 안에 'image' 키가 있는지
            확인
            if mediapipe_results and 'image' in mediapipe_results:
                image = mediapipe_results['image']

            # processor.process_data에 3개의 인자(data, image,
timestamp)를 전달
            result = await asyncio.wait_for(
                processor.process_data(data, image, timestamp),
                timeout=timeout
            )
        # ... (이하 기존 코드) ...

```

### 문제 3: 비동기/동기 호출 방식 불일치 (TypeError)

로그(dms\_20250710\_174818.log)에서 `object_processor`는 동기 함수인데 비동기 방식(`await`)으로 호출하려 해 오류가 발생했습니다. 이는 리팩토링 과정에서 일부 모듈만 비동기로 변경되었기 때문입니다.

파일: `analysis/orchestrator/orchestrator_advanced.py`

`_execute_processor_with_monitoring` 메서드에서 `object` 프로세서 호출 부분을 수정하여, 비동기 함수일 때만 `await`를 사용하도록 분기 처리합니다.

# `analysis/orchestrator/orchestrator_advanced.py`

```
import inspect # [추가] inspect 모듈을 임포트합니다.  
# ... (기존 코드 생략) ...
```

```
    async def _execute_processor_with_monitoring(  
        self, name: str, processor: Any, data: Any, timestamp: float,  
mediapipeline_results: dict = None  
    ) -> Tuple[bool, Dict[str, Any]]:  
        # ... (기존 코드 생략) ...  
        try:  
            if name == 'face':  
                # ... (위에서 수정한 코드) ...  
  
            # [수정] object 프로세서 호출 부분을 아래와 같이 변경합니다.  
            elif name == 'object':  
                # object_processor의 process_data가 코루틴(비동기)  
함수인지 확인합니다.  
                if  
inspect.iscoroutinefunction(processor.process_data):  
                    result = await asyncio.wait_for(  
                        processor.process_data(data, timestamp),  
                        timeout=timeout  
                    )  
                else:  
                    # 동기 함수일 경우, await 없이 직접 호출합니다.  
                    result = processor.process_data(data, timestamp)  
            else:  
                # ... (기존 코드) ...
```

### 문제 4: MediapipeManager 콜백 문제

`app.py`에서 `analysis_engine`이 동적으로 교체된 후 `MediaPipeManager`의 콜백 함수들이 비동기 함수임에도 불구하고 `await` 없이 호출될 가능성이 있습니다. 이는 "coroutine was never awaited" 경고나 분석 파이프라인의 비정상적인 동작을 유발할 수 있습니다.

파일: `systems/mediapipe_manager.py`

각 콜백 메서드(`_on_*_result`) 내에서, 전달받은 콜백 함수가 비동기 함수인 경우

asyncio.create\_task를 사용하여 안전하게 실행하도록 수정합니다.

```
# systems/mediapipe_manager.py
```

```
# ... (기존 코드 생략) ...
```

```
import asyncio # ▼ [추가] asyncio 모듈을 임포트합니다.
```

```
class EnhancedMediaPipeManager:
```

```
    # ... (기존 코드 생략) ...
```

```
    # ▼ [수정] 4개의 _on_*_result 메서드를 모두 아래와 같은 방식으로  
    변경합니다.
```

```
    # _on_face_result를 예시로 보여드립니다. _on_pose_result 등 다른  
    메서드도 동일하게 수정하세요.
```

```
    def _on_face_result(self, result, output_image, timestamp_ms,  
*args, **kwargs):
```

```
        logger.debug(f"Face result: {result}")
```

```
        if hasattr(result, 'face_landmarks'):
```

```
            logger.debug(f"Face landmarks: {getattr(result,  
'face_landmarks', None)}")
```

```
        if self.analysis_engine and hasattr(self.analysis_engine,  
'on_face_result'):
```

```
            cb = self.analysis_engine.on_face_result
```

```
        # ▼ [수정] 콜백이 비동기 함수인지 확인하고 비동기 작업으로  
    실행합니다.
```

```
        if asyncio.iscoroutinefunction(cb):
```

```
            # S-Class 프로세서의 경우, mediapipe_results 딕셔너리를  
    함께 전달
```

```
            mediapipe_results = {'image':  
output_image.numpy_view() }
```

```
            asyncio.create_task(cb(result, timestamp=timestamp_ms,  
mediapipe_results=mediapipe_results))
```

```
        else:
```

```
            # 기존 동기 엔진과의 호환성 유지
```

```
            cb(result, timestamp_ms)
```

```
        # _on_pose_result, _on_hand_result, _on_object_result도 위와 같은  
    패턴으로 수정해주세요.
```

```
        # (예시: asyncio.create_task(cb(result, timestamp_ms)))
```

## 요약

위의 코드 수정 사항들을 적용하면 로그 파일에서 발견된 대부분의 치명적인 오류들이 해결될 것입니다. 이 수정들은 모듈 간의 약속(인터페이스)을 통일하고, 비동기 코드의 실행 흐름을 보장하며, 초기화 순서에 따른 의존성 문제를 해결하는 데 중점을 둡니다.

이 수정들을 적용한 후 **python main.py**를 다시 실행하면 시스템이 정상적으로 동작할 가능성이

매우 높습니다.

사용자: 기존 코드도 확인해보세요.

모델:

네, 알겠습니다. 기존 코드와 로그 파일을 함께 면밀히 분석했습니다. 문제의 원인들이 명확하게 드러났으며, 대부분 리팩토링 과정에서 발생한 신규(新舊) 모듈 간의 인터페이스 불일치와 비동기 처리 로직의 누락에 기인합니다.

아래에 각 문제점에 대한 상세한 설명과 함께 수정해야 할 파일 및 코드를 정확하게 제시해 드리겠습니다. 이 수정안들을 적용하면 시스템이 정상적으로 동작할 것입니다.

## 문제 1: 비동기(Async) 컴포넌트 초기화 실패

오류 로그: ERROR - 비동기 컴포넌트 초기화 실패: 'PersonalizationEngine' object has no attribute 'initialize' (dms\_20250710\_174818.log)

원인 분석: app.py의 DMSApp 클래스는 personalization\_engine과 dynamic\_analysis 객체의 initialize() 메서드를 비동기적으로 호출(await)하려고 시도합니다. 하지만 해당 클래스들에는 initialize 메서드가 구현되어 있지 않았습니다.

### 해결 방안 1: systems/personalization.py 수정

PersonalizationEngine 클래스에 비동기 initialize 메서드를 추가합니다. 이 메서드는 프로필이 없을 경우 비동기적으로 파일을 생성하는 역할을 합니다.

# 파일: systems/personalization.py

# ... (기존 코드 생략) ...

import aiofiles # aiofiles 임포트 확인

class PersonalizationEngine:

"""개인화 학습 엔진"""

def \_\_init\_\_(self, user\_id: str):

# ... (기존 \_\_init\_\_ 코드) ...

self.\_load\_user\_profile()

# ▼ [수정] 아래 async initialize 메서드를 추가합니다.

async def initialize(self):

"""

비동기 초기화: 사용자 프로필 파일이 없으면 생성합니다.

"""

profile\_path = Path("profiles") /

f"{self.user\_id}\_profile.json"

if not profile\_path.exists():

await self.\_async\_save\_profile()

logger.info(f"PersonalizationEngine for {self.user\_id}

initialized.")

# ... (기존 나머지 코드) ...

## 해결 방안 2: `systems/dynamic.py` 수정

`DynamicAnalysisEngine` 클래스에도 비동기 `initialize` 메서드를 추가하여 인터페이스를 통일합니다.

```
# 파일: systems/dynamic.py

# ... (기존 코드 생략) ...

class DynamicAnalysisEngine:
    # ... (기존 __init__ 코드) ...

    # ▼ [수정] 아래 async initialize 메서드를 추가합니다.
    async def initialize(self):
        """비동기 초기화: 상태를 리셋합니다."""
        self.reset()
        logger.info("DynamicAnalysisEngine for async initialized.")

    def reset(self):
        # ... (기존 코드) ...
```

## 해결 방안 3: `app.py`의 초기화 로직 강화

`DMSApp._initialize_async_components` 메서드를 `asyncio.gather`를 사용하도록 수정하여, 모든 비동기 초기화 작업이 병렬로 실행되고 완료될 때까지 안전하게 기다리도록 합니다.

```
# 파일: app.py

# ... (기존 코드 생략) ...

class DMSApp:
    # ... (기존 __init__ 코드) ...

    # ▼ [수정] _initialize_async_components 메서드를 아래와 같이
    # 변경합니다.
    async def _initialize_async_components(self):
        """비동기 컴포넌트들을 병렬로 초기화합니다."""
        try:
            # 모든 초기화 코루틴을 리스트에 담습니다.
            init_tasks = [
                self.integrated_system.initialize(),
                self.personalization_engine.initialize(),
                self.dynamic_analysis.initialize()
            ]
            # asyncio.gather를 사용하여 모든 초기화를 병렬로 실행하고 완료될
            # 때까지 기다립니다.
            await asyncio.gather(*init_tasks)
            logger.info("✅ 모든 S-Class 비동기 컴포넌트 초기화 완료")
        except Exception as e:
```



```

        logger.error(f"비동기 컴포넌트 초기화 실패: {e}",
exc_info=True)
        raise

# ... (기존 코드 생략) ...

# ▼ [수정] 시스템 정리(_cleanup_async) 시 await None 오류를 방지하기
# 위해 코드를 강화합니다.
async def _cleanup_async(self):
    """
    시스템 종료 및 리소스 정리 (await None 방지)
    """
    logger.info("S-Class DMS v18+ 시스템 정리 중...")
    try:
        if hasattr(self, 'integrated_system') and
hasattr(self.integrated_system, 'shutdown'):
            # shutdown이 코루틴인 경우에만 await 합니다.
            coro = self.integrated_system.shutdown()
            if asyncio.iscoroutine(coro):
                await coro

        if hasattr(self, 'mediapipe_manager') and
hasattr(self.mediapipe_manager, 'close'):
            self.mediapipe_manager.close() # close는 동기
메서드입니다.

        if hasattr(self, 'video_input_manager') and
hasattr(self.video_input_manager, 'release'):
            self.video_input_manager.release() # release는 동기
메서드입니다.

    except Exception as e:
        logger.error(f"시스템 정리 중 오류: {e}", exc_info=True)

```

## 문제 2: 메서드 인자 불일치 및 비동기/동기 호출 혼용

오류 로그: `TypeError: FaceDataProcessor.process_data() missing 1 required positional argument: 'timestamp'` 및 `TypeError: An asyncio.Future, a coroutine or an awaitable is required` (dms\_20250710\_174818.log)

원인 분석: `AnalysisOrchestrator`가 각 프로세서의 `process_data` 메서드를 호출할 때, 어떤 프로세서는 `image`와 `timestamp` 인자가 모두 필요하고, 어떤 프로세서는 `timestamp`만 필요하며, 심지어 어떤 것은 동기 함수일 수 있습니다. 이러한 불일치를 `Orchestrator`가 모두 처리해야 합니다.

해결 방안: `analysis/orchestrator/orchestrator_advanced.py` 수정

`_execute_processor_with_monitoring` 메서드를 수정하여 각 프로세서의 특성에 맞게 동적으로

호출 방식을 결정하도록 합니다.

# 파일: analysis/orchestrator/orchestrator\_advanced.py

import inspect # ▼ [추가] inspect 모듈을 임포트합니다.

# ... (기존 코드 생략) ...

```
    async def _execute_processor_with_monitoring(
        self, name: str, processor: Any, data: Any, timestamp: float,
        mediapipe_results: dict = None
    ) -> Tuple[bool, Dict[str, Any]]:
```

```
        """[S-Class] 프로세서 실행 및 성능 모니터링 (대폭 수정)"""
```

```
        start_time = time.time()
```

```
        timeout = self.adaptive_timeouts.get(name, 0.1)
```

```
        try:
```

# ▼ [수정] processor.process\_data 호출 로직을 아래와 같이  
 변경합니다.

```
            # 1. 대상 메서드 가져오기
```

```
            process_method = processor.process_data
```

```
            # 2. 필요한 인자 구성
```

```
            args = []
```

```
            params = inspect.signature(process_method).parameters
```

```
            if 'data' in params: args.append(data)
```

```
            if 'image' in params:
```

```
                image = mediapipe_results.get('image') if  
mediapipe_results else None
```

```
                args.append(image)
```

```
            if 'timestamp' in params: args.append(timestamp)
```

```
            # 3. 비동기/동기 여부에 따라 실행
```

```
            if inspect.iscoroutinefunction(process_method):
```

```
                # 비동기 함수일 경우
```

```
                result = await asyncio.wait_for(
```

```
                    process_method(*args),
```

```
                    timeout=timeout
```

```
                )
```

```
            else:
```

# 동기 함수일 경우 (이벤트 루프의 블로킹을 막기 위해  
 스레드풀에서 실행)

```
                loop = asyncio.get_running_loop()
```

```
                result = await asyncio.wait_for(
```

```
                    loop.run_in_executor(None, lambda:  
process_method(*args)),
```

```
                    timeout=timeout
```

```

    )

    execution_time = time.time() - start_time

    # ... (이하 기존 성공/실패 처리 로직) ...
    perf = self.processor_performance[name]
    perf.avg_execution_time = (perf.avg_execution_time * 0.8 +
execution_time * 0.2)
    perf.last_success_time = timestamp
    perf.consecutive_failures = 0

    if perf.status == ProcessorStatus.FAILED:
        perf.status = ProcessorStatus.DEGRADED
        logger.info(f"{name} 프로세서 복구 감지")

    return True, result

except asyncio.TimeoutError:
    # ... (기존 코드) ...
except Exception as e:
    # ... (기존 코드) ...

```

수정 설명: 위 코드는 **inspect** 모듈을 사용하여 호출하려는 메서드의 시그니처를 동적으로 분석합니다.

1. 인자 맞춤: **data**, **image**, **timestamp** 중 해당 메서드가 필요로 하는 인자만 골라서 전달합니다.
2. 호출 방식 결정: 비동기 함수(**async def**)는 **await**로, 동기 함수(**def**)는 이벤트 루프를 막지 않도록 별도 스레드에서 실행하여 안전하게 호출합니다. 이로써 다양한 종류의 프로세서 모듈을 유연하게 통합할 수 있습니다.

## 최종 요약

제시된 수정안들은 현재 시스템이 겪고 있는 핵심적인 통합 문제들을 해결하는 데 초점을 맞추고 있습니다. 이 변경사항들을 적용하면, **S-Class** 시스템의 각 모듈들이 설계된 대로 유기적으로 연동되어 안정적으로 동작하고, 그 성능을 온전히 발휘할 수 있게 될 것입니다.

사용자: **main.py**, **app.py**, **event\_system.py**, **handlers.py**, **event\_bus.py**, **integrated\_system.py**, **ui.py**, **video\_input.py**, **gui.py**, **memory.py**, **logging.py**, **personalization.py**, **performance.py**, **metrics\_manager.py**, **mediapipe\_manager.py**, **dynamic.py**, **backup.py**, **state\_manager.py**, **interfaces.py**, **definitions.py**, **constants.py**, **prediction.py**, **identity.py**, **gaze.py**, **emotion.py**, **distraction.py**, **drowsiness.py**, **fusion\_engine\_advanced.py**, **orchestrator\_advanced.py**, **factory\_system.py**, **analysis\_factory.py**, **pose\_processor\_s\_class.py**, **object\_processor\_s\_class.py**, **hand\_processor\_s\_class.py**, **face\_processor\_s\_class.py**, **settings.py**

지금 입력한 코드들이 서로 유기적으로 연결되어 실제 프로그램 동작 시 문제가 없을지 어 너희 생각으로 시뮬레이션을 돌려 봐. 여러 관점에서 시뮬레이션을 돌려 봐. 실제 웹캠에서 입력되는 데이터를 쫓아가는 데이터 흐름 중심의 시뮬레이션, 혹은 순서도를 그리면서 여러 경우를 가정하여 시뮬레이션, 코드 상의 특정 변수를 추적하는 시뮬레이션, 어 기타 네가 생각한 여러 관점의 시뮬레이션들 이렇게 시뮬레이션을 돌려보고 문제는 없는지 개선점은 없는지

탐색해봐.

모델:

네, 좋습니다. 전체 **S-Class DMS** 프로젝트 코드를 대상으로 다각적인 시뮬레이션을 실행하여 유기적 연결성, 잠재적 문제점 및 개선점을 심층적으로 분석하겠습니다. 분석은 데이터 흐름, 시나리오, 주요 변수 추적, 그리고 시스템 아키텍처 관점에서 진행됩니다.

## 시뮬레이션 개요

목표: 제공된 모든 코드가 하나의 시스템으로 통합되어 실행될 때 발생할 수 있는 상호작용 문제, 병목 현상, 논리적 결함 및 성능 저하 요소를 사전에 식별하고 개선 방안을 도출합니다.

시뮬레이션 환경 (가상):

- 입력: 웹캠 (30fps)
- 하드웨어: 중간 사양의 PC (GPU 지원)
- 실행 모드: `main.py`를 통한 표준 실행 (STANDARD 시스템 타입)

## 관점 1: 데이터 흐름 중심 시뮬레이션 (Data-Flow Simulation)

이 시뮬레이션은 단일 프레임의 데이터가 시스템 전체를 어떻게 이동하고 변환되는지 추적합니다.

1. 입력 (**main.py** → **app.py** → **video\_input.py**)
  - `main.py`는 GUI/터미널로부터 설정값(입력소스: 0)을 받아 `DMSApp` 객체를 생성합니다.
  - `DMSApp`은 `VideoInputManager`를 초기화하고, `VideoInputManager`는 별도의 스레드에서 웹캠 프레임을 지속적으로 캡처하여 `self.current_frame`에 저장합니다. [연결 양호]
2. 전처리 (**app.py** → **mediapipe\_manager.py**)
  - `DMSApp`의 메인 루프는 `VideoInputManager.get_frame()`을 통해 프레임을 가져옵니다.
  - 가져온 프레임은 `EnhancedMediaPipeManager.run_tasks()`로 전달됩니다.
  - `run_tasks`는 `mp.Image`로 변환하고, 등록된 4개의 `MediaPipe Task` (`face`, `pose`, `hand`, `object`)에 `detect_async`로 전달합니다. 이 호출은 비동기적이므로 즉시 반환됩니다. [연결 양호]
3. 1차 분석 (**mediapipe\_manager.py** → **app.py: OrchestratorCallbackAdapter**)
  - `MediaPipe`가 내부적으로 처리를 완료하면 등록된 콜백 함수(`_on_*_result`)를 호출합니다.
  - 이 콜백 함수들은 `app.py`에 정의된 `OrchestratorCallbackAdapter`의 `on_*_result` 메서드를 호출합니다.
  - 어댑터는 두 가지 작업을 수행합니다: a. `mediapipe_manager`의 `last_*_result` 변수에 최신 결과를 저장합니다. b. `asyncio.create_task`를 통해 실제 **S-Class** 프로세서의 `process_data`를 비동기적으로 실행합니다. [핵심적인 유기적 연결]
4. 2차 심층 분석 (**Orchestrator** → **S-Class Processors**)
  - `AnalysisOrchestrator`는 `_execute_full_parallel_pipeline` 메서드를 통해 각 `S-Class` 프로세서(`face`, `pose`, `hand`, `object`)의 `process_data`를 병렬로 실행합니다.
  - 각 프로세서는 내부적으로 줄음, 감정, 척추, 떨림 등 고유의 고급 분석을 수행하고 결과를 반환합니다. [연결 양호]
5. 융합 및 상태 결정 (**Orchestrator** → **FusionEngine** → **StateManager** → **EventBus**)
  - `Orchestrator`는 모든 프로세서의 결과를 수집합니다.
  - 수집된 결과는 `MultiModalFusionEngine`으로 전달되어 `fatigue_risk_score`,

- distraction\_risk\_score 등의 최종 위험도로 융합됩니다.
  - IntegratedDMSSystem은 이 융합된 결과를 받아 StateManager의 상태를 업데이트하고, EventBus를 통해 DROWSINESS\_DETECTED와 같은 이벤트를 발행합니다. [연결 양호]
- 6. 이벤트 처리 및 피드백 (**EventBus → Handlers → UI**)
  - SafetyEventHandler와 AnalyticsEventHandler는 EventBus로부터 이벤트를 받아 경고 생성, 통계 기록 등의 작업을 수행합니다.
  - 최종적으로 app.py는 ui.py의 EnhancedUIManager를 사용해 모든 분석 결과를 프레임에 시각화하여 cv2.imshow로 출력합니다. [연결 양호]

## 데이터 흐름 시뮬레이션 결과

- 문제점 식별:
  - 아키텍처 모호성 및 잠재적 경쟁 상태(**Race Condition**): 데이터 처리 경로가 두 갈래로 나뉘어 있습니다.
    1. 경로 **A** (콜백 기반): MediaPlayerManager의 콜백 -> OrchestratorCallbackAdapter -> 각 S-Class 프로세서의 process\_data 직접 호출.
    2. 경로 **B** (메인 루프 기반): app.py의 메인 루프 -> integrated\_system.process\_and\_annotate\_frame -> Orchestrator.process\_frame\_data -> 각 S-Class 프로세서 호출.
  - 이 두 경로는 이론적으로 동시에 실행될 수 있으며, 같은 프로세서에 다른 시간의 데이터를 전달하여 예측 불가능한 결과를 낳을 수 있습니다. 로그에서 FaceDataProcessor의 process\_data 인자 오류가 반복적으로 발생한 것은 이 두 경로의 인터페이스가 통일되지 않았기 때문입니다.
- 개선 방안:
  - 데이터 처리 흐름을 한 가지 경로로 통일해야 합니다. **\*\*경로 A(콜백 기반)\*\***가 더 실시간에 가깝고 아키텍처적으로 우수합니다.
  - 따라서, app.py의 main\_async\_loop에서 integrated\_system.process\_and\_annotate\_frame을 호출하는 부분을 제거하고, 대신 콜백을 통해 처리된 최종 결과를 가져와 시각화하는 방식으로 변경하는 것이 좋습니다. 이렇게 하면 데이터 흐름이 명확해지고 잠재적 경쟁 상태가 사라집니다.

## 관점 2: 시나리오 기반 시뮬레이션

시나리오: 운전자가 운전 중 오른손으로 휴대폰을 집어 조작하는 상황

1. 초기 상태 (**0~2초**): 양손이 핸들에 있고, 시선은 전방 주시.
  - HandDataProcessor: 양손을 STEERING\_WHEEL 구역에서 감지. distraction\_score는 낮음.
  - ObjectDetector: 휴대폰을 감지하지 못함.
  - FusionEngine: distraction\_risk\_score는 0.1 이하로 매우 낮음.
  - StateManager: SAFE 상태 유지.
2. 행동 시작 (**2~3초**): 운전자가 오른손을 핸들에서 떼어 콘솔박스 쪽으로 이동.
  - HandDataProcessor: 오른손이 STEERING\_WHEEL 구역을 벗어남. distraction\_behaviors에 "오른손 핸들 이탈" 기록. distraction\_score 약간 상승.
  - ObjectDetector: 아직 휴대폰 감지 못함.
  - FusionEngine: 손의 위치 변화로 distraction\_risk\_score가 0.3 정도로 상승.
  - StateManager: SAFE -> DISTRACTION\_NORMAL로 상태 전환. EventBus에

이벤트 발행.

3. 상호작용 (3~5초): 오른손이 휴대폰을 집음. 휴대폰이 화면에 보이기 시작.
  - ObjectDetector: 'cell phone' 객체 감지.
  - ObjectDataProcessor (S-Class): 휴대폰과 손의 proximity\_score(근접도)가 급격히 상승. hand\_interaction이 True가 됨. risk\_level이 0.9 이상으로 계산됨.
  - HandDataProcessor (S-Class): 오른손의 그림 타입이 'precision\_grip'으로 변경될 가능성 높음.
  - FusionEngine: 손과 객체 양쪽에서 높은 주의산만 신호가 입력되어 distraction\_risk\_score가 0.8 이상으로 급상승.
  - StateManager: DISTRACTION\_DANGER 또는 PHONE\_USAGE 상태로 즉시 전환.
  - SafetyEventHandler: CRITICAL\_DISTRACTION 이벤트를 받아 강력한 경고(음성, 시각)를 발생시킴.
4. 행동 지속 (5초 이후): 운전자가 휴대폰을 보며 조작.
  - FaceDataProcessor: 머리 자세(yaw, pitch)가 아래쪽으로 향하며 gaze\_zone이 CENTER\_STACK 또는 FLOOR로 변경됨. gaze\_zone\_duration 증가. attention\_focus\_score 급락.
  - FusionEngine: 시선 데이터까지 주의산만 위험을 강력하게 지지하므로 distraction\_risk\_score가 1.0에 가깝게 유지됨.
  - PredictiveSafetySystem: 지속적인 위험 행동 패턴을 감지하여 predictive\_risk\_score를 높이고, PREDICTIVE\_WARNING 상태를 제안.

## 시나리오 시뮬레이션 결과

- 문제점 식별: 없음. 시스템은 설계된 대로 각 모듈이 유기적으로 협력하여 상황의 심각도를 점진적으로 인지하고, 위험이 구체화되었을 때 강력하게 대응하는 것을 확인할 수 있습니다. S-Class 프로세서들은 단순 감지를 넘어 행동의 '의도'와 '맥락'을 추론하는데 핵심적인 역할을 합니다.
- 개선 방안:
  - 회복(De-escalation) 시나리오 강화: 운전자가 휴대폰을 내려놓고 다시 핸들을 잡았을 때, 시스템이 얼마나 빨리 SAFE 상태로 복귀하는지에 대한 로직을 더 정교하게 만들 수 있습니다. 현재는 NORMAL\_BEHAVIOR 이벤트에 의존하지만, "긍정적 행동"을 점수화하여 더 빠르게 위험도를 낮추는 메커니즘을 추가할 수 있습니다.

## 관점 3: 주요 변수 추적 시뮬레이션

추적 변수: self.current\_pipeline\_mode (AnalysisOrchestrator 내부 변수)

1. 초기 상태: PipelineMode.FULL\_PARALLEL. 시스템 리소스가 충분하고 모든 프로세서가 정상 작동. 모든 분석이 병렬로 실행됨.
2. 성능 저하 발생: 비디오 처리량이 많아지거나 다른 프로세스가 CPU를 점유하여 평균 처리 시간이 150ms를 초과.
  - Orchestrator.\_assess\_system\_health(): 시스템 건강도 점수가 0.8 미만으로 하락.
  - \_determine\_optimal\_pipeline\_mode(): PipelineMode.SELECTIVE\_PARALLEL을 반환.
  - 상태 전이: FULL\_PARALLEL -> SELECTIVE\_PARALLEL. 이제 오케스트레이터는 face와 object 같은 중요 프로세서만 병렬 실행하고, pose나 hand는 순차 실행하여 부하를 줄임.
3. 프로세서 오류 발생: hand\_processor에서 반복적으로 타임아웃 발생.

- `_handle_processor_failure()`: `processor_performance['hand'].status`가 `FAILING`을 거쳐 `FAILED`로 변경됨.
  - `_assess_system_health()`: `hand` 프로세서의 건강 점수가 0이 되면서 전체 시스템 건강도 점수가 0.3 미만으로 급락.
  - `_determine_optimal_pipeline_mode()`: `PipelineMode.EMERGENCY_MINIMAL`을 반환.
  - 상태 전이: `SELECTIVE_PARALLEL` -> `EMERGENCY_MINIMAL`.  
오케스트레이터는 이제 가장 안정적인 `face_processor`만 실행하여 최소한의 핵심 기능(줄임, 시선)만 제공.
4. 시스템 회복: 오류가 해결되거나 시스템 부하가 줄어들.
- `_execute_processor_with_monitoring()`: 실패했던 프로세서들이 다시 성공하기 시작. `consecutive_failures`가 0으로 리셋됨.
  - `_assess_system_health()`: 건강도 점수가 점차 회복됨.
  - 상태 전이: `EMERGENCY_MINIMAL` -> `SEQUENTIAL_SAFE` -> `SELECTIVE_PARALLEL` -> `FULL_PARALLEL` 순으로 점진적으로 파이프라인 모드가 복구됨.

## 변수 추적 시뮬레이션 결과

- 문제점 식별: 없음. `AnalysisOrchestrator`의 적응형 파이프라인은 시스템의 안정성을 크게 높이는 핵심적인 **S-Class** 기능입니다. 시스템은 스스로 상태를 진단하고 실행 전략을 동적으로 변경하여 어떠한 상황에서도 다운되지 않고 서비스를 지속하려는 '의지'를 보여줍니다.
- 개선 방안:
  - 적응형 전략의 고도화: 현재는 파이프라인 모드만 변경하지만, 각 프로세서 내부의 파라미터(예: 분석 정밀도, 사용하는 모델의 크기)까지 동적으로 변경하도록 확장할 수 있습니다. 예를 들어, `LOW_RESOURCE` 모드일 때 얼굴 랜드마크 모델을 더 가벼운 버전으로 교체하는 식입니다.

## 종합 결론 및 최종 제언

시뮬레이션 결과, 제공된 **S-Class DMS** 코드는 매우 진보된 아키텍처를 가지고 있으며, 각 모듈이 유기적으로 잘 연결되어 있습니다. 특히 장애 허용(**Fault-Tolerance**) 설계와 **\*\*적응형 파이프라인(Adaptive Pipeline)\*\***은 시스템의 안정성과 효율성을 극대화하는 뛰어난 기능입니다.

다만, 실제 운영을 위해 다음과 같은 개선을 제안합니다.

1. **[가장 시급]** 데이터 처리 경로 통일: `app.py`의 메인 루프와 `MediaPipeManager`의 콜백에서 발생하는 이중 처리 경로를 콜백 기반으로 통일하여 데이터 흐름의 모호성을 제거하고 안정성을 확보해야 합니다.
2. 상태 회복 로직 강화: 위험 상황 경고 후, 운전자의 긍정적인 행동(전방 주시, 핸들 잡기 등)을 감지하여 시스템이 더 능동적으로 '안전' 상태로 복귀하는 로직을 추가하면 사용자 경험이 크게 향상될 것입니다.
3. 상황 인지(**Context-Awareness**) 확장: `ObjectDataProcessor`의 상황인지형 위험도 조정을 `FusionEngine` 레벨로 확장하여, 차량 속도, 날씨, 도로 유형(고속도로/시내)과 같은 외부 데이터를 받아 분석 가중치를 동적으로 조절하는 기능을 추가하면 시스템의 예측 정확도가 한 차원 더 높아질 것입니다.

이러한 개선점들을 반영한다면, 현재의 **S-Class DMS** 시스템은 학술적 우수성을 넘어 상용 제품 수준의 완성도와 안정성을 갖추게 될 것입니다.

사용자: Qualcomm RB2 플랫폼에서 S-Class DMS 시스템을 최적으로 구동하기 위한 로드맵에서 알고리즘 측면에서 더 최적화할 부분은 없는지 연구해 봐.

모델:

네, 하드웨어 가속과 시스템 아키텍처 최적화를 넘어, 알고리즘 자체의 효율성을 높이는 것은 저사양 임베디드 플랫폼에서 성능을 확보하기 위한 매우 중요한 단계입니다. Qualcomm RB2 플랫폼의 특성을 고려하여 알고리즘 측면에서 최적화할 수 있는 방안을 심층적으로 연구하고 구체적인 개선안을 제시하겠습니다.

## 알고리즘 최적화 로드맵 for Qualcomm RB2

알고리즘 최적화의 핵심 목표는 "최소한의 연산으로 최대의 정보를 얻는 것"입니다. 이를 위해 연산량 감소, 근사(Approximation), 조건부 실행, 그리고 시간적 정보 활용이라는 네 가지 전략을 중심으로 개선 방안을 도출했습니다.

### 관점 1: 연산량 감소를 위한 경량화 알고리즘

무거운 수학적 연산을 더 가볍고 빠른 연산으로 대체하거나, 불필요한 계산을 줄이는 전략입니다.

#### 1. EAR(Eye Aspect Ratio) 계산 최적화 (drowsiness.py)

- 현재: 각 눈의 EAR을 계산할 때 여러 번의 유클리드 거리 계산(제공근 연산 포함)을 수행합니다. 제공근 연산은 임베디드 CPU에서 상대적으로 비싼 연산입니다.
- 개선 방안: **EAR<sup>2</sup>** (제공 EAR)<sup>2</sup>을 사용합니다. EAR 값 자체보다 임계값과의 비교가 중요하므로, 굳이 제공근을 계산할 필요가 없습니다. 거리의 '제곱' 값을 사용하면 성능에 유리합니다.

# drowsiness.py의 \_calculate\_enhanced\_ear 수정

```
def _calculate_enhanced_ear_sq(self, landmarks, eye_side):
    # ... eye_points 정의 ...
    try:
        eye_landmarks = [landmarks[i] for i in eye_points]
        # 제공 유클리드 거리 계산 (math.sqrt 제거)
        vertical_1_sq =
self._euclidean_distance_sq(eye_landmarks[1], eye_landmarks[5])
        vertical_2_sq =
self._euclidean_distance_sq(eye_landmarks[2], eye_landmarks[4])
        horizontal_sq =
self._euclidean_distance_sq(eye_landmarks[0], eye_landmarks[3])

        if horizontal_sq > MathConstants.EPSILON:
            # EAR 공식의 분자와 분모를 모두 제곱하여 비교
            ear_sq = (vertical_1_sq + vertical_2_sq) / (2.0 *
horizontal_sq)
        else:
            ear_sq = 0.0
        return ear_sq

def _euclidean_distance_sq(self, point1, point2):
    return (point1.x - point2.x)**2 + (point1.y - point2.y)**2
```



# 이와 함께 개인화된 임계값(`personalized_threshold`)도 제공 값으로 저장하고 비교해야 합니다.  
기대 효과: 프레임당 수십 번의 제공된 연산을 제거하여 CPU 부하를 미세하게나마 꾸준히 줄일 수 있습니다.

## 2. FFT(고속 푸리에 변환) 연산 최적화 (`hand_processor_s_class.py`)

- 현재: 손 떨림 분석을 위해 매번 `rfft`를 호출합니다. FFT는 연산량이 많은 작업입니다.
- 개선 방안: 조건부 실행 및 다운샘플링을 적용합니다.
  1. 조건부 실행: 손의 속도(`kinematics`)가 매우 낮아 거의 정지 상태일 때만 FFT를 수행합니다. 움직이는 손의 떨림을 분석하는 것은 노이즈가 많고 불필요합니다.
  2. 다운샘플링: `hand_kinematics_history`의 모든 데이터를 사용하지 않고, 2~3개 중 1개만 샘플링하여 FFT 연산량을 1/2 ~ 1/3로 줄입니다. 주파수 해상도는 약간 낮아지지만, 8~12Hz의 피로 떨림을 감지하는 데는 큰 문제가 없습니다.

<!-- end list --># `hand_processor_s_class.py`의 `_analyze_tremor_frequency` 수정

```
def _analyze_tremor_frequency(self, handedness: str) -> Dict[str, Any]:
    # ...
    # [추가] 조건부 실행: 손이 거의 멈춰있을 때만 분석
    kinematics = self.hand_kinematics_history[handedness][-1][2] #
    마지막 속도 정보
    if np.linalg.norm(kinematics) > 0.01: # 임계값 (설정으로 관리 가능)
        return {'dominant_frequency_hz': 0.0, ...}

    # [수정] 다운샘플링 적용
    history_downsampled =
    list(self.hand_kinematics_history[handedness][:2] # 2개 중 1개만 선택

    if len(history_downsampled) < hand_config.fft_min_samples:
        return {'dominant_frequency_hz': 0.0, ...}

    y_positions = np.array([h[1][1] for h in history_downsampled])
    # ... 이하 FFT 처리 로직 ...
```

기대 효과: 불필요한 FFT 연산을 획기적으로 줄여 CPU 사용량을 크게 절감할 수 있습니다.

## 관점 2: 근사(Approximation) 및 단순화

정확도에 큰 영향이 없는 선에서 복잡한 계산을 단순한 모델로 근사하여 성능을 높이는 전략입니다.

### 1. 시선 구역 분류 단순화 (`gaze.py`)

- 현재: 3D 벡터를 생성하고 여러 구역의 벡터와 내적(dot product)하여 각도를 계산하는 방식입니다. 삼각함수와 `acos` 연산이 포함되어 있습니다.
- 개선 방안: 룩업 테이블(Look-Up Table, LUT) 또는 간단한 경계 상자(Bounding Box) 기반 분류를 우선적으로 사용합니다.
  - LUT 방식: 머리의 Yaw와 Pitch 각도를 일정 간격(예: 5도)으로 나누어 미리 계산된 시선 구역을 테이블에 저장해 둡니다. 실행 시에는 현재 Yaw/Pitch 값과 가장 가까운 테이블 값을 찾아 즉시 결과를 반환합니다.
  - 경계 상자 방식: 각 시선 구역을 단순한 2D 사각형 범위로 근사화하고,

코끝(Nose\_Tip) 랜드마크의 2D 화면 좌표가 어느 사각형에 속하는지로 1차 판단합니다. 복잡한 상황에서만 3D 분석을 수행합니다.

## 2. 행동 예측 모델 경량화 (prediction.py)

- 현재: RiskPredictor가 선형 회귀(polyfit)와 표준편차 등 통계적 연산을 수행합니다.
- 개선 방안: 이동 평균 기반 추세 분석으로 대체합니다. polyfit보다 훨씬 가볍습니다.  
# prediction.py의 TrendAnalyzer.analyze\_trend 수정

```
class TrendAnalyzer:
    def analyze_trend(self, data):
        if len(data) < 20: # 충분한 데이터가 필요
            return 0.0

        # [수정] polyfit 대신 이동 평균 사용
        short_term_avg = np.mean(data[-10:]) # 최근 10개 프레임 평균
        long_term_avg = np.mean(data)         # 전체 평균

        # 단기 평균이 장기 평균보다 높으면 증가 추세로 판단
        slope = short_term_avg - long_term_avg

        # ... (이하 로직은 동일)
```

기대 효과: 복잡한 회귀 분석을 단순 평균 계산으로 대체하여 연산 복잡도를  $O(n)$ 에서  $O(1)$  수준으로 낮출 수 있습니다 (새로운 데이터가 들어올 때마다 전체를 다시 계산할 필요 없음).

## 관점 3: 조건부 및 적응형 실행

모든 기능을 항상 실행하는 대신, 필요한 상황에서만 선택적으로 실행하여 시스템 부하를 동적으로 조절하는 전략입니다. AnalysisOrchestrator의 파이프라인 모드와 유사하지만, 더 세부적인 알고리즘 레벨에서 적용됩니다.

### 1. 계층적 감정 분석 (emotion.py)

- 현재: 매 프레임 모든 감정에 대한 점수를 계산합니다.
- 개선 방안: 2단계 분류를 도입합니다.
  1. 1단계 (Valence-Arousal 분류): 먼저 긍정/부정/중립(Valence)과 각성/이완(Arousal)이라는 두 가지 큰 축으로만 간단히 분류합니다. 이 계산은 훨씬 가볍습니다.
  2. 2단계 (세부 감정 분류): 1단계에서 '중립'이나 '안정' 상태가 아니라고 판단될 때만, Anger, Sadness 등 세부적인 7가지 감정을 분류하는 무거운 연산을 수행합니다.

기대 효과: 운전자의 감정 상태가 대부분 중립인 점을 고려할 때, 불필요한 세부 감정 분석 연산을 90% 이상 줄일 수 있습니다.

### 2. 적응형 개인화 임계값 업데이트 (drowsiness.py)

- 현재: is\_calibrated 플래그가 False일 때, 일정 프레임(300)이 쌓이면 임계값을 업데이트합니다.
- 개선 방안: 변화 감지 기반 업데이트를 도입합니다.
  - 초기 캘리브레이션 이후에는 임계값을 고정합니다.
  - 대신, EAR 값의 분포(분산, 표준편차)가 이전 5분과 비교하여 크게 달라졌을 때만 재캘리브레이션을 트리거합니다. 예를 들어, 조명이 바뀌거나 운전자가 선글라스를 쓰는 등 환경 변화가 감지될 때만 자동으로 재학습을 수행합니다.

기대 효과: 불필요한 임계값 계산을 방지하고, 정말 필요할 때만 연산을 수행하여 효율성을

높입니다.

## 최종 종합 결론 및 제언

하드웨어 가속이 시스템의 **물리적 체력**을 높이는 것이라면, 알고리즘 최적화는 **운동 기술**을 향상시키는 것과 같습니다. Qualcomm RB2와 같은 임베디드 플랫폼에서는 이 두 가지가 모두 필수적입니다.

최적화 적용 로드맵:

### 1. Level 1 (즉시 적용 가능):

- **EAR<sup>2</sup>** 사용: `drowsiness.py`를 수정하여 제공된 연산을 제거합니다.
- **FFT** 조건부 실행: `hand_processor_s_class.py`를 수정하여 손이 정지 상태에 가까울 때만 떨림 분석을 수행하도록 합니다.
- 예측 모델 경량화: `prediction.py`에서 선형 회귀를 이동 평균 방식으로 변경합니다.

### 2. Level 2 (아키텍처 수정 필요):

- 계층적 감정 분석: `EmotionRecognitionSystem`의 로직을 2단계로 나누어 구현합니다.
- **Gaze** 분류 최적화: 3D 벡터 계산 대신, 경계 상자 기반의 빠른 1차 분류 후 필요한 경우에만 3D 분석을 수행하도록 `gaze.py`를 수정합니다.

### 3. Level 3 (장기적 연구 개발):

- 모델 경량화 (양자화/가지치기): Qualcomm AI 모델 효율성 킷(AIMET)과 같은 도구를 사용하여 MediaPipe 모델(.tflite)을 양자화(Quantization)하거나 가지치기(Pruning)하여 모델 크기를 줄이고 DSP에서의 추론 속도를 더욱 높입니다.
- 상관관계 기반 분석 스킵: 만약 눈 감김(EAR)과 고개 숙임(Pitch) 사이에 강한 양의 상관관계가 지속적으로 관찰된다면, 둘 중 하나의 신호가 강하게 나타날 때 다른 하나의 정밀 분석은 건너뛰는 등, 통계적 특성을 활용하여 중복 연산을 줄입니다.

이러한 알고리즘 최적화 방안들을 기존에 제안된 하드웨어 오프로딩 및 시스템 아키텍처 개선과 함께 적용한다면, **Qualcomm Robotics RB2** 플랫폼에서도 **S-Class DMS** 시스템의 핵심적인 고급 기능들을 대부분 유지하면서도 안정적인 프레임 속도를 확보할 수 있을 것입니다.