ChatGPT

아스트레우스 연결 프레임워크 – 통합 구현 예시 및 가이드

본 가이드는 **아스트레우스 연결 프레임워크**(Astraeus Link Framework, ALF)의 핵심 모듈들을 프로토타입 수준으로 구현하고, 전체 코드를 GitHub에 올릴 수 있는 형태로 구성하는 방법을 설명합니다. 여기에는 Python 기반 백엔드와 React 기반 프론트엔드 예시, 카프카(Kafka) 연동을 통한 로그 관리(ψ-log), 단위 테스트, Dockerfile 및 문서화(README 등)가 모두 포함됩니다. 또한 ΔS/ΔR 등의 지표 모니터링과 Self-Check 루프(자기 점검 루프)를 통한 자동 조정 흐름도 설명합니다. 아래에서 각 구성 요소별 핵심 개념과 구현 방식, 디렉터리 구조, 실행 방법을 단계적으로 살펴보겠습니다.

🗺️ 핵심 모듈 및 아키텍처 개요

아스트레우스 연결 프레임워크는 **5대 축(Axis)**을 따라 모듈화되어 있으며, 각 축은 조직/개인의 주요 약점을 보완하는 기능을 담당합니다 [1] [2]. 다음은 각 모듈과 역할에 대한 요약입니다:

- **T-Axis: ΔS Gradient Scheduler** – 지식 엔트로피 변화(ΔS)의 **기울기**를 모니터링하여 실행 지연을 방지하는 스케줄러 [3]. 지식베이스(예: 최근 커밋 로그)의 엔트로피가 목표보다 **천천히 감소**하면 경고(alert)를 발생시켜 작업을 **타임박싱**합니다 [4].
- **Ψ-Axis: Impact-Penalty 가중치 계산기** – 성과 지표(CHI, ECR 등)에 **Impact Score(IS)**와 **Penalty Factor(PF)**를 부여하여, 특정 지표에 대한 과도한 집착을 완화합니다 [5]. 모든 지표 값을 `조정값 = 원본 × IS ÷ PF`로 재계산하여 균형을 유지하고, 조정된 값이 기준 구간을 벗어나면 경고 표시 및 **후행 지표(Lagging Indicator)** 개선 워크플로를 자동 생성합니다 [5].
- **C-Axis: Cross-Frame Sentinel** – 현재 프레임(work frame) 밖의 **이질적 정보** 유입을 조기에 탐지합니다 [6]. 입력 데이터를 벡터화하여 현재 프레임의 기준 벡터와 **코사인 유사도**를 계산하고, 유사도가 일정 임계값 θ보다 낮으면(예: 0.35 미만) 해당 입력에 `frame_divergence=true` 태그를 붙여 로그에 기록합니다 [6]. 이렇게 검출된 이탈(frame divergence) 이벤트들은 분기별 **Frame-Audit 미팅** 아젠다에 자동 추가됩니다 [7].
- **E-Axis: ΔR Elasticity Controller** – 상황 변화에 따라 **위험 허용폭(리스크 버퍼)**을 자동 조정합니다 [8]. 실시간으로 주요 지표의 변동성(예: ΔCHI, ΔECR)을 계산하여 변동성이 **감소(σ↓)**하면 리스크 허용 레벨(ΔR)을 한 단계 늘리고, 변동성이 **증가(σ↑)**하면 ΔR을 한 단계 줄입니다 (레벨 L1–L5 범위) [9]. ΔR이 충분히 높을 때(L3 이상)만 **Adaptive Sandbox** 실험모드를 허용하고, 리스크 완화 상태에서 성공한 실험은 지식 엔트로피 감소량(ΔS_reduction)에 비례하는 **보상**을 부여합니다 (reward = ΔS_reduction × β) [10].
- **S-Axis: Multi-Layer Viewport** – 동일한 콘텐츠를 **전문가/실무자/이해관계자** 등 서로 다른 **시각(layer)**으로 동시에 보여주는 다계층 뷰포트입니다 [11]. Markdown 문서의 front-matter 메타데이터에 `audience_level` 필드를 추가하여 내용의 대상 수준을 명시하면, 프론트엔드 렌더러가 해당 문서를 3계층으로 렌더링합니다 [12]. 또한 문서 내 전문용어 비율이 30%를 넘으면 TL;DR 요약 블록을 자동으로 삽입하여 이해를 돕습니다 [13].

위 모듈들은 **ψ-log**라는 중앙 로그 스트림과 **Self-Check 루프(자가 판단 루프)**를 통해 상호 연결됩니다. 예를 들어, T-Axis 스케줄러가 ΔS 목표 기울 대비 편차를 감지하면 **ψ-log에 경고 이벤트를 기록**하고(Self-Judgment 루프 트리거) [14], 이 이벤트는 Slack 알림 등을 통해 운영자에게 전파됨과 동시에 내부 알고리즘 파라미터(self-model)를 자동 조정합니다 [15]. C-Axis Sentinel이 frame_divergence를 true로 설정한 로그들을 모아 정기 Frame-Audit 미팅 자료로 활용하는 것도 이러한 루프의 일부입니다 [14]. 전체적으로 ψ-log 스트림에는 ΔS_target, impact_score, penalty_factor, frame_divergence 등의 필드가 포함되며, Kafka **psi-events** 토픽으로 수집됩니다 [16]. 모니터링 시스템은 ψ-log로부터 ΔS, ΔR, CHI, ECR 등 **핵심 지표 스트림을 실시간 시각화**하고 이상 징후 발생 시 경보를 발생시킵니다 [17].

프로젝트 구조 및 구성 요소

GitHub에 올릴 수 있도록, 프로젝트 코드는 백엔드와 프론트엔드가 구분된 **모듈화된 디렉터리 구조**로 구성됩니다. 각 주요 기능은 별도 파일/모듈로 분리하여 향후 확장이 쉽도록 했습니다. 예시 디렉터리 구조는 다음과 같습니다:

```
AstraeusLinkFramework/                # ALF 프로젝트 루트 디렉터리
├── backend/                          # Python 백엔드 관련 코드
│   ├── app.py                        # FastAPI 앱 및 백그라운드 작업 스케줄러
│   ├── scheduler.py                  # ΔS Gradient Scheduler (T-axis 구현)
│   ├── penalty.py                    # Impact-Penalty 가중치 모듈 (Ψ-axis 구현)
│   ├── sentinel.py                   # Cross-Frame Sentinel (C-axis 구현)
│   ├── elasticity.py                 # ΔR Elasticity Controller (E-axis 구현)
│   ├── resource_control.py           # 기타 리소스 제어 유틸 (Light-Prompt Bypass
등)
│   └── __init__.py                   # 패키지 초기화 (필요시)
├── frontend/                         # React 프론트엔드 코드
│   ├── public/                       # 정적 파일 (예: index.html, favicon 등)
│   ├── src/                          # 소스 코드
│   │   ├── App.jsx                   # 주요 React 앱 컴포넌트
│   │   ├── MultiLayerViewport.jsx    # 다계층 뷰포트 구현 컴포넌트 (S-axis)
│   │   └── ... 기타 컴포넌트/헬퍼 ...
│   ├── package.json                  # 프론트엔드 의존성 및 스크립트
│   └── ... (webpack 설정 등) ...
├── schema/
│   └── psi_event.avsc                # Kafka ψ-log 이벤트 Avro 스키마 정의 [18]
├── tests/                            # Pytest 기반 백엔드 단위 테스트 코드
│   ├── test_scheduler.py             # ΔS Scheduler 테스트
│   ├── test_penalty.py               # 가중치 조정 모듈 테스트
│   ├── test_sentinel.py              # Sentinel 기능 테스트
│   ├── test_elasticity.py            # Elasticity Controller 테스트
│   ├── test_resource_control.py      # 리소스 제어 유틸 테스트
│   └── conftest.py                   # 테스트 공용 설정 (있다면)
├── Dockerfile                        # 백엔드용 Docker 이미지 빌드 파일
├── docker-compose.yml                # 전체 서비스(Kafka, 백엔드, 프론트 포함) 구성
(옵션)
├── requirements.txt                  # 백엔드 Python 의존성 목록
└── README.md                         # 사용법 및 설정에 대한 문서
```

위 구조에서 **backend** 디렉터리가 주요 Python 서비스를 담고 있으며, **frontend** 디렉터리는 React로 작성된 간단한 뷰포트 앱을 포함합니다. Kafka 연동을 위해 Avro 스키마(`psi_event.avsc`)를 명시적으로 포함했고, **tests** 디렉터리에 각 모듈별 단위 테스트 코드를 작성하였습니다. 추가로 Docker로 손쉽게 배포할 수 있도록 Dockerfile과 docker-compose.yml도 제공됩니다. README.md에는 프로젝트 소개, 설정 방법, 실행 방법, 예시 등이 문서화됩니다.

각 구성 요소별로 구현 세부사항과 코드를 살펴보면 다음과 같습니다.

ΔS Gradient Scheduler (T-Axis) – scheduler.py

ΔS Gradient Scheduler는 시간 경과에 따른 지식 엔트로피(Shannon Entropy)의 감소량 ΔS를 추적하여, **프로젝트 진행 속도가 느려지는 것을 감지**하면 자동으로 경고를 발생시키는 모듈입니다 [3]. 구현 측면에서, 이전 시점의 지식베이스와 현재 시점의 지식베이스(예: 최근 커밋 로그의 토큰 분포)를 받아 두 시점 간 엔트로피 차이(ΔS)를 계산하는 함수와, ΔS 감소 **목표 기울기(grad)** 대비 실제 ΔS가 벗어나는지를 판단하는 로직으로 구성됩니다.

- **엔트로피 계산**: Shannon 엔트로피 $H(K) = -\Sigma\ p_i \log_2 p_i$ 로 정의되며, 주어진 토큰 분포에서 쉽게 계산할 수 있습니다 [19]. 아래 Python 함수는 토큰 리스트를 받아 엔트로피를 계산하고, ΔS 값을 반환합니다:

```
# scheduler.py (일부 발췌)
from collections import Counter
import math

def entropy(counter: Counter) -> float:
    total = sum(counter.values())
    return -sum((n/total) * math.log2(n/total) for n in counter.values())

def delta_s(prev_tokens: list[str], curr_tokens: list[str]) -> float:
  """두 시점의 토큰 리스트로부터 ΔS (엔트로피 감소량, bits)를 계산"""
    return entropy(Counter(prev_tokens)) - entropy(Counter(curr_tokens))
```

(위 pseudo-code는 설계서의 파이썬 예시를 참고하여 구현 [20])

- **스케줄러 로직**: ΔS_target (목표 ΔS 기울기)을 단계별로 설정해 두고, 실제 측정된 ΔS가 이 목표보다 너무 낮아지는 경우 **경사 이탈**로 간주합니다 [4]. 예를 들어 Day0에 ΔS_target=+0.08, Day5에 +0.02로 감소하는 식으로 목표를 줄여갈 수 있습니다 [4]. ΔS가 목표 대비 일정 임계치(ΔS_grad_threshold)보다 벗어나면, ψ-log에 `"msg": "ΔS out-of-gradient"` 이벤트를 기록하고 `alert_level=2`로 승격합니다 [21] [22]. 이때 백엔드에서는 Slack 웹훅 등을 통해 운영 담당자(Slack `#alf-ops` 채널)에 자동 알림을 보낼 수 있습니다 [23]. 동시에 Self-Check 루프를 통해 **24시간 내 핵심 결정 2개 이외에는 모두 보류**하도록 시스템 모드를 전환합니다 [24] (실행 과부하를 방지하고 핵심 결정에 집중).

스케줄러는 일정 시간 간격(예: 3시간마다)으로 ΔS를 측정하여 ψ-log에 기록하며 [21], 구현상 `app.py`에서 백그라운드 태스크로 주기적인 `scheduler.check_progress()`를 돌리거나 별도 스레드/프로세스로 실행할 수 있습니다. ΔS_grad_threshold 등의 파라미터는 **설정 파일**(예: `alf-config.yaml`)이나 환경변수로 조정 가능하게 해두는 것이 좋습니다.

Ψ-Axis Impact-Penalty 가중치 모듈 – penalty.py

Impact-Penalty 모듈은 **성과 지표(metric)**들의 **중요도와 페널티**를 반영하여 지표의 왜곡된 최적화를 방지합니다 [5]. 예컨대 모델 성능 지표인 CHI나 운영 지표인 ECR 등에 대해, 조직 전략상 중요도가 높은 지표에는 **Impact Score(IS)** > 1.0을 주고, 과도하게 치중하기 쉬운 지표에는 **Penalty Factor(PF)** > 1.0을 주어 조정합니다 [25]. 구현은 단순히 **조정값 = Raw × IS ÷ PF** 계산으로 이루어지며 [26], 각 지표별 IS와 PF는 설정으로 관리합니다 (예: CHI는 IS=1.0, PF=1.0 기본; 사용자 NPS는 IS=1.2, PF=0.8 등 [25]).

- **지표 조정 함수**: 다음 함수는 주어진 원시 지표값을 IS와 PF로 보정하여 반환합니다. 이 값을 이용해 지표가 정상 범위 안에 있는지 판단합니다.

```python
# penalty.py
METRIC_PARAMS = {
    "CHI": {"IS": 1.0, "PF": 1.0},
    "ECR": {"IS": 1.0, "PF": 1.0},
    "NPS": {"IS": 1.2, "PF": 0.8},   # 예: 고객 추천 지표에 가중치 적용
    # ... 기타 지표 ...
}

def adjust_metric(metric_name: str, raw_value: float) -> float:
  """지표의 원시값을 Impact/Penalty 가중치를 반영하여 조정된 값으로 반환"""
    params = METRIC_PARAMS.get(metric_name, {"IS": 1.0, "PF": 1.0})
    return raw_value * params["IS"] / params["PF"]
```

- **모니터링 및 워크플로**: 조정된 지표값이 미리 정의한 **레퍼런스 밴드(목표 범위)**를 벗어나면, ψ-log에 경고 이벤트를 남기고 해당 지표에 대한 **개선 워크플로**를 자동 생성합니다 [27]. 예를 들어 CHI 지표의 조정값이 너무 낮아지면, 시스템은 자동으로 "CHI 지표 개선을 위한 액션 아이템" 태스크를 생성하거나 담당자에게 알림을 보낼 수 있습니다. 이러한 기능은 거버넌스 룰 엔진 또는 간단한 **if-else 로직**으로 구현할 수 있으며, 중요 지표에 **노란색/빨간색 경고 표시**를 프론트엔드 대시보드에 표시하여 직관적으로 파악할 수도 있습니다 [28].

C-Axis Cross-Frame Sentinel – sentinel.py

Cross-Frame Sentinel은 **현재 설정된 문제 프레임을 벗어나는 입력**(데이터/이벤트)을 탐지하여, 조직이 특정 관점에 갇히는 현상(프레임 고착)을 방지하는 역할을 합니다 [29]. 구현상 이 모듈은 입력 데이터를 임베딩 벡터로 변환한 후, **기준 프레임 벡터**(예: 현재 프로젝트의 핵심 주제나 문맥을 대표하는 벡터)와의 코사인 유사도를 계산합니다 [6]. 유사도가 낮아서 임계값 θ 이하로 떨어지면 해당 이벤트를 `frame_divergence=true`로 로그에 남깁니다 [6].

- **벡터 유사도 계산**: 간단한 구현으로, 텍스트 입력을 받아 TF-IDF나 Word2Vec/임베딩 등을 통해 벡터화하고 코사인 유사도를 구할 수 있습니다. 아래 예시는 두 벡터 사이의 코사인 유사도를 계산하는 함수와 임계값 비교를 보여줍니다:

```python
# sentinel.py (벡터 유사도 계산 예시)
import numpy as np

def cosine_similarity(vec1: np.ndarray, vec2: np.ndarray) -> float:
    return float(np.dot(vec1, vec2) / (np.linalg.norm(vec1) * np.linalg.norm(vec2)))

FRAME_THRESHOLD = 0.35   # 유사도 임계값 θ

def is_frame_divergent(input_vec: np.ndarray, base_vec: np.ndarray) -> bool:
  """입력 벡터가 프레임 기준 벡터와 거리가 큰지 판단 (True=이탈)"""
    sim = cosine_similarity(input_vec, base_vec)
    return sim < FRAME_THRESHOLD
```

- **Sentinel 동작**: 위 함수에서 `True`가 반환되면, 이 입력은 프레임과의 편차가 큰 것으로 간주되어 ψ-log에 `frame_divergence: true` 속성과 함께 기록됩니다 [6]. 예를 들어 event_type을 `"frame_divergence"`로 하여 어떤 데이터가 이탈로 탐지되었는지 남길 수 있습니다. 프레임 이탈 이벤트들은 분기마다 열리는 **Frame-Audit 미팅** 때 아젠다로 활용됩니다 [7]. 본 프로토타입에서는 Sentinel이 남긴 로그를 모아 주기적으로 Markdown 리포트를 생성하는 스크립트를 작성할 수도 있습니다. (설계서 예시: 2주치 Sentinel 로그를 수집해 Mermaid 시퀀스 다이어그램으로 자동 생성 [30]).

🔍 참고: 실제 환경에서는 입력 데이터 유형에 따라 벡터화 방법을 다르게 적용해야 합니다 (텍스트, 이미지 등). 프로토타입에서는 예를 들어 **문서 임베딩**(문장 임베딩)으로 간단히 구현하거나, 임계값 θ 역시 `alf-config.yaml`에서 조정 가능하도록 설계합니다.

E-Axis ΔR Elasticity Controller – elasticity.py

ΔR Elasticity Controller는 변화하는 환경 속에서 **위험 허용 범위(리스크 버퍼)**를 유연하게 조절하는 모듈입니다 [8]. 이는 시스템의 **안정성 지표(CHI)**와 **위험/비용 지표(ECR)** 등의 변동성을 모니터링하여, **변동성이 낮아지면 더 많은 위험을 감수(L+1)**하고 변동성이 높아지면 보수적으로 변환(L-1)하는 식으로 ΔR 레벨(L1~L5)을 조절합니다 [9].

- **변동성 계산**: 구현에서는 일반적으로 **표준편차(σ)**나 **EWMA(Exponentially Weighted Moving Average)**로 최근 변동성을 계산할 수 있습니다. 예를 들어 일정 기간의 CHI, ECR 값을 모아 표준편차를 구하거나, EWMA로 추세를 파악합니다. 아래 함수는 단순히 CHI와 ECR 값 목록을 받아 결합 변동성을 구한 후 ΔR 레벨을 조정하는 예시입니다:

```python
# elasticity.py
import numpy as np

MAX_RISK_LEVEL = 5
MIN_RISK_LEVEL = 1

# 임계값 (예시값): 변동성 0.05 미만이면 완화, 0.10 초과면 경직
LOW_VOL_THRESHOLD = 0.05
HIGH_VOL_THRESHOLD = 0.10

def update_risk_level(current_level: int, chi_series: list[float], ecr_series: list[float]) -> int:
    """최근 CHI/ECR 시계열로부터 변동성을 계산하고 ΔR 레벨을 상향/하향 조정"""
    combined = np.array(chi_series + ecr_series)
    volatility = np.std(combined)
    new_level = current_level
    if volatility < LOW_VOL_THRESHOLD and current_level < MAX_RISK_LEVEL:
        new_level += 1  # 변동성 감소 → 리스크 허용 수준 상향
    elif volatility > HIGH_VOL_THRESHOLD and current_level > MIN_RISK_LEVEL:
        new_level -= 1  # 변동성 증가 → 리스크 허용 수준 축소
    return new_level
```

- **ΔR 레벨 적용**: 계산된 ΔR 레벨은 전역적으로 관리되며, **Adaptive Sandbox**와 같은 기능에서 활용됩니다. 예를 들어 ΔR 레벨이 3 이상인 경우에만 실험용 샌드박스 기능(잠재적으로 위험한 변경을 시험하는 모드)을 **활성화**하도록 시스템이 동작합니다 [9]. 또한 ΔR이 완화된 상태(L4–L5 등)에서 실행된 실험이 성공하여 ΔS 감소에 기여하면, 해당 성과를 보상으로 기록합니다 [31]. 보상 계산은 `reward = ΔS_reduction × β` 공식에 따라 이뤄지며 [31], 이 값도 ψ-log에 이벤트로 남겨 향후 자기학습에 활용됩니다.

Elasticity Controller 모듈 역시 백엔드 서비스로서 주기적으로 동작하거나, 새로운 CHI/ECR 값이 들어올 때 이벤트 기반으로 `update_risk_level`을 호출하도록 구현할 수 있습니다. ΔR 레벨 변화 이벤트는 `event_type: "risk_level_change"` 형태로 Kafka에 기록하고, 대시보드에 현재 레벨(L1–L5)을 표시합니다.

S-Axis Multi-Layer Viewport – React 프론트엔드

Multi-Layer Viewport는 하나의 문서를 서로 다른 전문 수준의 **3개 레이어**로 동시에 보여주는 프론트엔드 컴포넌트입니다 [11]. 예를 들어 **Expert(전문가)**, **Practitioner(실무자)**, **Stakeholder(비전문 이해관계자)** 레벨로 나눈 설명

을 탭(Tab) 등으로 전환하며 볼 수 있게 합니다 [32]. 이러한 동시 출력 기능을 통해, 복잡한 기술 문서라도 다양한 독자가 자신의 수준에 맞게 이해할 수 있습니다 [11].

프론트엔드는 **React**로 구현하며, 간단한 상태 관리로 현재 선택된 레이어를 보여줍니다. 문서 콘텐츠는 Markdown이나 HTML로 관리하고, 각 레벨별 내용을 구분하기 위해 Markdown의 **Front-matter**에 `audience_level` 메타데이터 필드를 사용합니다 [12]. 또한 각 레벨별 섹션을 문법적으로 구분하거나, 별도 파일로 분리한 후 프론트엔드에서 병합해 보여줄 수도 있습니다.

- **Front-matter 예시**: Markdown 문서 상단에 YAML 형식으로 대상 독자 수준을 표기합니다. 예:

```
---
title: "Quantum Corridor Simulation"
audience_level: "Expert"
---
이 문서에는 양자 역학 기반 Corridor 모델에 대한 상세 내용이 포함됩니다...
```

유사하게 Practitioner와 Stakeholder 용 문서를 준비하거나, 한 문서 내에 구획별로 `:::expert`, `:::practitioner` 블록을 둘 수도 있습니다. 프로토타입에서는 편의상 별도 파일이나 데이터 구조로 각 수준의 콘텐츠를 관리합니다.

- **React 컴포넌트 구현**: `MultiLayerViewport.jsx`는 세 가지 레벨의 콘텐츠를 받아 탭 전환 UI로 표시합니다. 아래는 핵심 로직의 개념적 구현입니다:

```
// MultiLayerViewport.jsx (요약 구현)
import React, { useState } from 'react';

function MultiLayerViewport({ expertContent, practitionerContent, stakeholderContent }) {
 const [level, setLevel] = useState("Stakeholder");
 return (
  <div className="viewport-container">
   {/* 레벨 전환 버튼들 */}
   <div className="level-tabs">
    {["Expert", "Practitioner", "Stakeholder"].map(l => (
     <button key={l} onClick={() => setLevel(l)}
          className={level === l ? "active" : ""}>
      {l}
     </button>
    ))}
   </div>
   {/* 선택된 레벨의 콘텐츠 표시 */}
   <div className="content">
    {level === "Expert" && <div>{expertContent}</div>}
    {level === "Practitioner" && <div>{practitionerContent}</div>}
    {level === "Stakeholder" && <div>{stakeholderContent}</div>}
   </div>
  </div>
 );
}
```

```
export default MultiLayerViewport;
```

이 컴포넌트는 상위 `App.jsx`에서 세 종류의 콘텐츠를 prop으로 받아 렌더링합니다. 실제 구현에서는 Markdown 파서나 HTML 렌더를 써서 front-matter의 `audience_level`을 검사하고 세 가지 버전의 내용을 구성할 수 있습니다. (예: **mermaid**나 **YAML parser**를 활용하여 정적 사이트 생성기 비슷하게 콘텐츠 준비 [33]). 또한, **전문용어 비율**을 계산하여 30%를 넘으면 해당 섹션 상단에 자동으로 "**요약 (TL;DR)**" 박스를 삽입합니다 [13]. 이 요약은 사전에 작성된 것을 넣거나, 중요 문장을 추려서 보여줄 수 있습니다.

> 참고: 이 다계층 뷰포트 아이디어는 문서의 각 단락에 `audience_level` 태그를 달아 구성할 수도 있습니다. 프로토타입에서는 간단히 구현하기 위해 레벨별 별도 콘텐츠를 정의했지만, 추후에는 하나의 문서에서 레벨에 따라 가려지는 방식을 고려할 수 있습니다. 또한, 프론트엔드에서 전문용어 비율을 계산하려면 사전에 정의된 **용어 사전**이나 NLP 기법을 사용하여 문장을 분석해야 하지만, 여기서는 설정된 비율 임계치만으로 간략히 다룹니다.

ψ-log 기록 및 Kafka 연동

앞서 설명된 모든 모듈의 이벤트와 지표들은 **ψ-log**라는 중앙 로그 시스템으로 모입니다. 본 구현에서는 Kafka 메시지 브로커를 사용하여 ψ-log 이벤트를 관리하며, 이벤트 스키마는 **Avro**로 정의합니다. Avro 스키마 파일 `schema/psi_event.avsc`에는 최소한 다음과 같은 필드들이 포함됩니다 [18]:

```
{
 "namespace": "alf.logs",
 "type": "record",
 "name": "PsiEvent",
 "fields": [
  {"name": "timestamp",  "type": "long"},   // 이벤트 발생 시간 (epoch ms)
  {"name": "event_type", "type": "string"}, // 이벤트 종류 (예: "delta_s_alert", "frame_divergence" 등)
  {"name": "payload",    "type": "string"}  // 기타 데이터(JSON 직렬화 문자열 등)
 ]
}
```

이 기본 스키마 외에, 설계서에서는 **새로 확장된 필드 4종**을 예시로 들었습니다 [16]: `ΔS_target`, `impact_score`, `penalty_factor`, `frame_divergence`. 이러한 값들은 payload 내에 JSON 형태로 포함시키거나, 필요하면 Avro 필드로 직접 추가할 수도 있습니다. 예를 들어 ΔS 스케줄러의 경고 이벤트는 JSON payload에 다음과 같이 기록합니다:

```
{
 "time": "2025-07-21T02:14:11Z",
 "msg": "ΔS out-of-gradient",
 "chi": 0.92,
 "ecr": 0.95,
 "delta_s": 0.055,
 "ΔS_target": 0.040,
 "impact_score": 1.0,
 "penalty_factor": 1.0,
 "frame_divergence": false,
```

```
  "alert_level": 2
}
```

(ψ-log 레코드 예시 [22] [34])

Kafka 연동을 위해 백엔드에서는 **프로듀서(producer)**와 **컨슈머(consumer)**를 설정합니다. 예를 들어 Python의 `kafka-python` 라이브러리를 사용하여, 각 모듈에서 이벤트가 발생할 때 Kafka Producer를 통해 `psi-events` 토픽에 Avro 메시지를 보냅니다. 초기 개발 단계에서는 **더미(dummy) 프로듀서** 스크립트를 만들어, 주기적으로 테스트 이벤트를 보내도록 할 수 있습니다:

```
# kafka_producer_demo.py (Kafka 더미 이벤트 송신 예시)
from kafka import KafkaProducer
import json, time
producer = KafkaProducer(bootstrap_servers=["localhost:9092"],
                         value_serializer=lambda v: json.dumps(v).encode('utf-8'))
event = {
    "timestamp": int(time.time()*1000),
    "event_type": "poc_progress",
    "payload": json.dumps({"message": "PoC step1 complete"})
}
producer.send("psi-events", event)
producer.flush()
```

운영 시에는 각 모듈이 직접 Kafka Producer를 사용하지만, 개발 단계에서는 위와 같은 스크립트로 토픽 동작을 검증합니다. **컨슈머**는 Elasticity Controller나 Scheduler 같은 모듈이 될 수 있습니다. 예를 들어 ΔR Elasticity Controller는 실시간 CHI/ECR 값을 받기 위해 해당 값이 포함된 ψ-log 이벤트를 **구독**(consume)하고, 값을 추출하여 `update_risk_level`을 호출하도록 설계합니다 [35] [36].

Kafka 및 ψ-log를 통해 시스템 전반의 이벤트가 중앙 관리되므로, 이를 활용한 **Self-Check 루프**가 구현 가능합니다. Self-Check(또는 Self-Judgment) 루프란 **MAPE-K** 제어 루프의 개념을 차용하여, ψ-log 스트림을 모니터링(Monitor) → 이상 패턴을 분석(Analyze) → 파라미터 자동 조정(Plan & Execute) → 지식베이스 업데이트(Knowledge) 과정을 거치는 자기 개선 알고리즘입니다 [37] [38]. ALF에서는 ψ-log를 **Meta Loop**에 입력하여, 내부 매개변수(예: ΔS_grad_threshold, θ 등)를 지속 조정하는 자기모델(Self-Model) 기법을 적용합니다 [15].

마지막으로, ψ-log 시스템은 대시보드 및 경보 체계와 연계됩니다. Grafana나 Kibana와 같은 툴을 사용하여 **ΔCHI, ΔECR, ΔS, Latency 등의 스트림을 실시간 표시**하고 이상치 발생 시 경고를 띄울 수 있습니다 [39]. Slack Webhook을 이용하여 중요한 이벤트(alert_level 2 이상)가 발생하면 팀 채널에 알림을 보내는 기능도 구현됩니다 [23]. 예를 들어 scheduler에서 ΔS 경고 발생 시:

```
# app.py 혹은 scheduler.py 내부
if ds < ds_target - DS_GRAD_TOLERANCE:
    log_event("delta_s_alert", { "delta_s": ds, "target": ds_target, "alert_level": 2 })
    notify_slack(channel="#alf-ops", message=f"ΔS 이탈 경고: {ds:.3f}/{ds_target:.3f}")
```

위 `notify_slack` 함수는 `requests` 등을 사용해 Slack Incoming Webhook URL로 POST 요청을 보내는 형태로 구현합니다. (Slack 알림 연동은 설계 로드맵 5주차에도 포함된 항목입니다 [40].)

리소스 제어 유틸리티 – resource_control.py (부가 기능)

프로토타입 코드에는 참고 문서에 언급된 **Light-Prompt Bypass**와 **ψ-MEM Compaction** 기능도 예시적으로 포함되었습니다 [41]. 이 모듈은 대화형 AI 에이전트 시나리오에서 프롬프트 길이와 위험도를 판단하여 **간단한 요청은 신속 처리**하고 복잡한 요청만 풀 파이프라인을 거치도록 하는 한편, 장기 메모리(ψ-MEM)을 주기적으로 정리하여 시스템이 **경량 유지**되도록 돕습니다 [42].

- Light-Prompt Bypass: 짧고(low-risk)한 질문은 바로 응답하고 복잡한 질문만 전체 파이프라인(예: 검색, 분석 등)을 수행합니다. 구현은 `is_light_prompt(question: str) -> bool` 함수로, 질문 길이가 일정 임계값 미만이고 사전에 정의한 **위험 키워드**가 없으면 True를 반환합니다 [43] [44]. 예를 들어 길이 50자 미만이고 "실험", "검증" 등의 단어가 없으면 LIGHT_BYPASS로 간주합니다.
- ψ-MEM Compaction: 에이전트의 메모리 저장소(예: 대화 내역, 벡터 임베딩 캐시 등)가 너무 커지지 않도록 **LRU(Least Recently Used) 전략**과 **엔트로피 기반 삭제**를 적용합니다 [45] [46]. 즉, 메모리가 `MAX_MEMORY_ITEMS` 개수를 초과하면 가장 오래된 항목부터 삭제하고, 남은 항목 중에서도 정보 엔트로피가 낮은(중복되거나 의미없는) 벡터는 제거합니다 [47] [46]. `compact_memory(store)` 함수는 이러한 로직을 수행하여 제거된 항목 수를 리턴합니다.

리소스 제어의 자세한 구현은 `src/resource_control.py`에 있으며, 이는 본 프로젝트의 `backend/resource_control.py`에 통합되었습니다. 이 모듈은 자체적으로 Pytest 단위 테스트도 포함하고 있어, 프로젝트 전반의 **테스트 코드 작성을 위한 좋은 참고 예시**가 됩니다 [48] [49]. 예컨대 `test_compact_memory_lru_and_entropy()`에서는 의도적으로 `MAX_MEMORY_ITEMS`보다 많은 항목과 한 개의 **저엔트로피 항목**을 삽입한 뒤 compact_memory를 실행해, 메모리 크기가 제한 이내로 떨어졌는지와 최소 하나 이상의 항목이 삭제되었는지를 검증합니다 [50] [51].

테스트 및 검증

본 프로젝트는 **pytest**를 사용하여 각 모듈의 기능을 검증하는 단위 테스트를 제공합니다. `tests/` 디렉터리 아래에 모듈별로 대응되는 테스트 파일이 있으며, pytest를 실행하면 모든 테스트가 구동됩니다. 예를 들어:

- `test_scheduler.py` – ΔS 계산 함수(`delta_s`)에 대해 이전/현재 토큰 집합을 넣어 예상한 ΔS 값이 나오는지 검사하고, 스케줄러의 경고 발생 조건이 제대로 동작하는지 시뮬레이션합니다.
- `test_penalty.py` – 여러 지표에 대해 `adjust_metric` 함수를 테스트하여, 설정된 IS/PF대로 값이 조정되는지, 그리고 임계 범위 초과 시 올바른 플래그/이벤트가 발생하는지 확인합니다.
- `test_sentinel.py` – `is_frame_divergent` 함수에 유사도 높은 벡터(살짝 변형된 base_vec)와 유사도 낮은 벡터를 넣어, 각각 False/True를 반환하는지 확인합니다. 벡터 유사도 계산의 정확성도 함께 테스트합니다.
- `test_elasticity.py` – 여러 시나리오의 CHI/ECR 시계열에 대해 `update_risk_level` 결과를 검증합니다. 변동성이 극히 낮은 데이터로 ΔR이 상승하는지, 높은 변동성으로 ΔR이 감소하는지 등을 확인합니다.
- `test_resource_control.py` – Light-Prompt Bypass와 메모리 콤팩션 기능 테스트 (이 코드는 이미 모듈에 포함된 예시를 활용) [48] [49]. 예를 들어 `test_is_light_prompt`는 짧은 질문, 위험 키워드 포함 질문, 매우 긴 질문에 대해 True/False 반환을 체크합니다 [52].

단위 테스트의 목표 커버리지는 80% 이상으로 설정되어 있으며 [53] [54], CI 파이프라인 (예: GitHub Actions)에서 테스트 통과 및 코드 커버리지 검증을 수행하도록 구성할 수 있습니다. 이는 GitHub에 코드를 올린 후 지속적인 코드 품질 관리에 도움을 줍니다.

GitHub 업로드 및 실행 절차

이제 전체 코드를 GitHub에 **배포 가능한 형태로 구성**했으므로, 실제로 사용자가 프로젝트를 받아 실행할 수 있는 방법을 정리하겠습니다.

1. 초기 설정 및 저장소 구성

- **저장소 생성**: GitHub에 새 repository를 만들고, 위에 소개한 디렉터리 구조에 따라 파일들을 배치합니다. 주요 파일(`app.py`, `Dockerfile`, `README.md` 등)이 최상위에 위치하도록 하고, 불필요한 파일은 포함하지 않습니다.
- **의존성 명시**: `backend/requirements.txt`에는 FastAPI, kafka-python, numpy, pyyaml, requests 등 백엔드에서 사용하는 모든 Python 패키지를 기록합니다. `frontend/package.json`에는 React, React-DOM 등의 의존성이 자동 포함되며, 필요에 따라 mermaid-js 등 추가 라이브러리를 포함할 수 있습니다.
- **README 문서화**: README.md에는 프로젝트의 개요, 핵심 기능 설명, 설치 및 실행 방법, 예시 등이 적절히 포함되어야 합니다. 본 가이드의 많은 부분은 README에도 요약하여 활용할 수 있습니다.

2. 실행 환경 준비

- **필수 구성요소**: Docker를 사용하는 경우, Docker와 Docker Compose가 설치된 환경이 필요합니다. 그렇지 않다면 로컬에 Python (3.9+ 권장)과 Node.js(16+ 권장), 그리고 Kafka (로컬 또는 원격 클러스터) 접근이 가능해야 합니다.
- **Kafka 설정**: 로컬에서 테스트하려면 Docker Compose를 통해 **Kafka 브로커와 ZooKeeper**를 띄울 수 있습니다. 예를 들어 `docker-compose.yml`에 confluentinc/cp-kafka 이미지 등을 포함해 `psi-events` 토픽 생성을 스クリ̗プト로 자동화해두면 편리합니다 [55] [56]. Avro 스키마 등록은 Schema Registry를 사용하거나, 애플리케이션 레벨에서 Avro 인코딩을 수행합니다.
- **환경변수**: Slack Webhook URL, Kafka 브로커 주소 등 민감한 설정은 `.env` 파일이나 환경변수로 관리합니다. README에 환경설정 방법을 명시하고, `.env.example` 파일을 제공하면 사용자가 쉽게 따라할 수 있습니다.

3. 빌드 및 실행

- **백엔드 실행**:
- **로컬 실행**: `pip install -r requirements.txt`로 의존성을 설치한 후, 예를 들어 FastAPI의 uvicorn 서버를 사용할 경우 `uvicorn backend.app:app --reload` 명령으로 개발 서버를 시작합니다. 백엔드 서비스가 기동되면, 콘솔 로그로 Kafka 연결 시도, Scheduler 시작 메시지 등이 출력됩니다. Kafka 브로커가 가동 중이라면 psi-log 구독/발행이 이 시점부터 활성화됩니다.
- **Docker 실행**: `docker build -t alf-backend .` 명령으로 이미지를 빌드하고, `docker run -d -p 8000:8000 alf-backend`로 컨테이너를 띄웁니다. Docker Compose를 사용했다면 `docker-compose up -d`로 한꺼번에 Kafka, 백엔드, 프론트엔드를 올릴 수도 있습니다.
- **프론트엔드 실행**:
- **로컬 실행**: `cd frontend` 후 `npm install` 또는 `yarn install`로 패키지를 설치하고, `npm start`로 React 개발 서버를 띄웁니다. 브라우저에서 `http://localhost:3000`으로 접속하면 Multi-Layer Viewport 예시 페이지가 열립니다. 이 페이지는 내부적으로 백엔드 API를 호출하지는 않지만(정적 예시 내용 표시), 필요에 따라 백엔드의 특정 엔드포인트(예: 최신 ΔS 값, 현재 ΔR 레벨 등)를 호출하도록 구현할 수 있습니다.
- **Docker 실행**: 프론트엔드용 Dockerfile을 별도 작성했거나, 멀티스테이지 빌드로 프론트엔드를 정적 번들로 만든 경우 Nginx 등을 통해 서빙합니다. 간단한 경우 `npm run build`로 정적 파일을 생성하고, Docker 이미지 내에서 이를 제공하도록 설정합니다.
- **Kafka 더미 프로듀서 실행 (테스트용)**: 카프카 토픽과 백엔드 컨슈머가 모두 준비되었다면, `python kafka_producer_demo.py`를 실행하여 테스트 이벤트를 몇 개 발생시킵니다. 그러면 백엔드 로그나 Kafka 소비자에서 해당 이벤트를 감지한 것을 확인할 수 있습니다.

4. 동작 확인 및 추가 조치

- **모듈 동작 확인**: 백엔드 로그를 통해 각 모듈이 의도대로 동작하는지 확인합니다. 예를 들어, 백엔드 기동 후 3시간 간격으로 `"ΔS out-of-gradient"` 경고가 발생하지 않으면 정상 (ΔS가 목표 이내), 일부러 ΔS_target 값을 낮춰서 경고를 발생시켜 볼 수 있습니다. 발생 시 Slack 채널 메시지 수신 여부도 확인합니다 [23].
- **프론트엔드 확인**: 브라우저에서 3개의 레벨 버튼을 클릭해보며 콘텐츠가 잘 전환되는지, TL;DR 블록이 필요한 경우 표시되는지 확인합니다. 만약 전문용어 비율 검증을 구현했다면 전문가 콘텐츠에 일부러 어려운 용어를 넣고 30% 넘게 한 뒤 TL;DR 출력 여부를 테스트합니다.
- **통합 시나리오 테스트**: 가능하다면 E2E 테스트나 통합 테스트 스크립트를 작성하여, 예를 들어 **ΔS 경고 발생 → ψ-log 기록 → Slack 알림 → 프론트엔드 대시보드 표시** 같은 흐름이 하나의 시나리오로 잘 이뤄지는지 검증합니다 [57] [58]. 이는 수동으로 로그를 추적하거나 pytest의 통합 테스트로 구성할 수 있습니다.

5. GitHub 푸시 및 관리

- 로컬에서 일련의 기능을 모두 확인했다면 Git 커밋을 만들고 GitHub에 코드를 푸시합니다. 저장소의 About/Description에 프로젝트 요약을 적고, 주요 releases 또는 tags를 활용해 버전을 관리합니다.
- 향후 확장을 위해 이슈 트래커나 프로젝트 보드를 만들어 개선할 점(예: θ 값 튜닝, UI 개선, 실서비스 연계 등)을 관리하면 좋습니다. 첫 **Frame-Audit 미팅**에서는 Sentinel 로그를 검토하여 임계값 θ와 ΔS_grad_threshold 등을 조정하고, 이를 `alf-config.yaml`에 반영한 후 커밋합니다 [59] [60].

결론 및 향후 확장

이상으로 **아스트레우스 연결 프레임워크 통합 코드**의 예시 구현과 사용 방법을 살펴보았습니다. 이번 프로토타입은 문서에 제시된 개념을 가능한 한 실제 코드와 시스템 아키텍처에 투영한 것으로, 핵심 취지는 다음과 같습니다:

- 조직의 약점을 **계량 가능한 파라미터**로 변환하고 자동화 루프로 편입시킴으로써, 약점을 지속적으로 모니터링하고 학습/보정할 수 있게 되었습니다 [61]. 이제 ΔS나 ΔR의 변화만 추적해도 실행 지연, 지표 편향, 프레임 고착 등의 문제를 실시간 감시·완화할 수 있습니다 [62].
- 본 구현은 **프로토타입 수준**이므로 간소화된 로직과 하드코딩된 값들이 존재합니다. 추후 실제 적용 단계에서는 각 모듈의 알고리즘을 고도화하고, 성능 최적화(FPGA 오프로드 등 고급 기법 적용 가능 [63]), 보안(예: Kafka ACL 및 오류 처리 [64]) 등을 보완해야 합니다.
- 프레임워크는 **모듈화된 구조**를 갖춰 향후 확장이 용이합니다. 예를 들어 새로운 축(Axis)을 추가하거나, 기존 모듈에 서브 컴포넌트(예: Knowledge Integration Layer, PCL 등)를 붙이는 것이 가능합니다 [65] [66]. 또한 지표 및 로그 구조도 스키마 버전 관리로 유연하게 확장될 수 있습니다 [67].

끝으로, 사용자는 본 가이드와 예시 코드를 참고하여 GitHub 상에서 동일한 구조의 프로젝트를 생성하고, 자신의 환경에 맞게 수정/실행할 수 있습니다. 초기 설정과 실행을 완료했다면, 실제 데이터와 연동하거나 팀원들과 함께 프레임워크를 실험해보면서 ALF의 효과를 검증해 볼 수 있을 것입니다. **아스트레우스 연결 프레임워크**를 활용한 자동화된 약점 보완 기법이 조직의 실행 속도와 혁신 민첩성을 높이는 좋은 기반이 되기를 기대합니다 .

참고 자료: 본 구현 가이드는 제공된 설계서와 모듈 코드 자료를 기반으로 작성되었습니다. 자세한 알고리즘 근거나 개념 설명은 "아스트레우스 연결 – 약점 보강 모듈 설계서" [3] [5] 및 통합 프레임워크 문서 [14]를 참조하십시오. 또한 일부 코드 구현과 테스트 예시는 제공된 `src_resource_control.py` 모듈 [45] [46]를 활용했습니다.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [16] [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28] [29] [30] [31] [32] [33] [34] [35] [36] [40] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [64] [67] 개인-올림-아스트레우스 연결 – 약점 보강 모듈 설계서-001.pdf
file://file-5LQPi5bin2fNQb934fhbkh

15 17 37 38 39 63 65 66 올림-002-아스트레우스 연결 통합 프레임워크-009.pdf
file://file-JFS9oWAgQNtnhgFDxB5tRe

41 42 43 44 45 46 47 48 49 50 51 52 src_resource_control.py
file://file-FsunTCwqCnEQExzJXUEdPs