댕동여지도 서비스

반려견의 산책을 위한 동기를 사용자에게 부여하는 서비스
반려견을 산책하며 땅따먹기 게임을 같이 진행하도록 하는 게이미피케이션 요소,
산책 도중 돌발 미션으로 반려견의 영상을 찍어서 제출 (앉아, 손, 엎드려, 등등) 하면
성공여부 도출,
헬스케어 기능에서는 반려견의 보행 영상을 찍어서 객체인식, 키포인트, 키포인트를
직선으로 연결 등을 해서 해당 반려견의 슬개골 위험, 좌우 보행 균형, 무릎 관절 가동성,
보행 안정성, 보행 리듬 등의 분석을 해서 사용자에게 해당 항목들을 위험 신호 (점수로
표현된?) 를 표시,
산책을 마친 후 반려견의 영상을 찍어서 올리면 표정을 촬영해 AI 로 감정을 분석하는 기능,
수의사 상담 진료 말뭉치 데이터를 학습시킨 AI 상담 챗봇 기능 ,
개인 랭킹, 지역별 랭킹, 기여도 등을 표시,
캘린더에 산책 일지를 저장하는 기능,
땅따먹기 게임을 팀배틀로 할 수 있게하는 기능,
등이 지금 있는데

헬스케어
표정분석
돌발미션
챗봇

배포 계획

---

v1
1. 산책
2. 돌발 미션
3. 표정 분석
4. 마이페이지
    1. 최근 업적 제외
5. 반려견 정보
6. 사용자 정보
7. 로그인

v2
1. 랭킹
    1. 개인 랭킹
    2. 지역 랭킹
2. 발자국
3. 헬스케어 및 챗봇
    1. 챗봇 대화 내용 기억 X

v3
1. 챗봇 고도화
    1. 이전 대화 내용 기억
2. AI 모델의 응답 정확도 고도화

3. 마이페이지 > 최근 업적
4. 산책하기 고도화
5. 다른 페이지 이동 가능하도록 고도화
    1. 산책 중 다른 페이지 이동
    2. 표정 분석 중 다른 페이지 이동
    3. AI추천경로 분석 중 다른 페이지 이동
    4. 헬스케어 분석 중 다른 페이지 이동

v4
1. AI추천경로
2. 리캡
3. 산책 > 팀 배틀
4. 랭킹 > 랜덤 랭킹

돌발미션 설계

yeonjung77 edited this page 2 hours ago · 40 revisions
‼ 돌발 미션
📌 개요
산책 중 촬영된 반려견 돌발미션 영상을 기반으로
상용 멀티모달 AI 기반으로 미션 수행 성공 여부를 판단하는
AI 모델 API의 설계 및 성능 최적화 전략을 정의한다.

단계 1
📌 모델 API 설계

본 API는 산책 중 촬영되어 DB에 저장된 반려견 영상을 입력으로 받아, 산책 완료 시점에 상용 멀티모달 AI API를 호출하여 각 영상에 대해 지정된 미션 수행 여부를 의미 이해 기반으로 분석하고, 서비스 레벨 결과로 제공한다.

분석 상태 관리와 분석 결과 제공을 분리된 API 역할로 제공한다.

돌발미션 영상은 산책 중 DB에 저장되고, 미션 성공/실패 분석은 산책 종료 시점에 일괄 수행되어 제공된다.

missions_type : 앉아(SIT), 엎드려(DOWN), 손(PAW), 돌아(TURN), 점프(JUMP)

1-1. API 설계 개요
분석 시점 분리
산책 중: 돌발미션 영상 촬영 및 DB 저장
산책 종료 시점: 저장된 모든 돌발미션 영상을 대상으로 분석
처리 방식 분리
동기처리 : 산책 완료 시점에 분석 요청 후 모든 미션이 완료될 때까지 대기
비동기 (Job) 확장 : 분석 요청과 상태 조회, 결과 조회를 분리
역할 분리 원칙
Job 상태 관리 API : 분석 진행 상태만 제공
결과 접근 API : 분석 완료 후 저장된 JSON 결과 제공
상용 멀티모달 AI API 연동
입력: 영상 URL + missions_type(텍스트)
출력: 미션 수행 성공 여부(success/failure), 신뢰도, 요약 설명

1-2. Endpoint 목록 및 기능 설명
동기
Method Endpoint Description
POST /api/mission/judge 산책 종료 후 돌발미션을 동기적으로 일괄 분석
/api/mission/judge 설명
산책 종료 시점에 호출되는 동기 분석 API
DB에 저장된 돌발미션 영상 목록과 mission_type을 입력으로 받음
각 미션 영상에 대해 상용 멀티모달 AI API를 즉시 호출
모든 미션 분석이 완료될 때까지 요청을 유지하며 대기

분석 결과를 즉시 응답으로 반환하는 단일 요청–단일 응답 구조

비동기

구분	Method	Endpoint	Description
돌발미션 Job 생성	POST	/api/missions/judge/jobs	돌발미션 분석 Job 생성
Job 상태 조회	GET	/api/missions/judge/jobs/status/{job_id}	Job 진행 상태 조회
Job 결과 조회	GET	/api/missions/judge/jobs/result/{job_id}	완료된 분석 결과(JSON) 조회

/api/missions/judge/jobs 설명
산책 종료 시점에 호출
DB에 저장된 돌발미션 영상 목록과 missions_type을 입력으로 받음
내부적으로 각 미션 영상에 대해 상용 멀티모달 API를 개별 호출
즉시 분석 결과를 반환하지 않고 job_id를 반환
※ Job 상태 조회: /api/missions/judge/jobs/status/{job_id}
※ Job 결과 조회: /api/missions/judge/jobs/result/{job_id}

1-3. 입력/출력 형식 명세
📌 동기

※ 동기 / 비동기 API는 동일한 Request Schema를 사용하며, 처리 방식만 다르다.

Post /api/missions/judge/jobs

Request

Field	Type	Description
analysis_id	string	산책 분석 ID
walk_id	string	산책 세션 ID
missions	array	돌발미션 목록
missions.mission_id	string	돌발미션 ID
missions.mission_type	enum	SIT, DOWN, PAW, TURN, JUMP
missions.instruction	string	사용자에게 제시된 미션 지시 문구
missions.video_url	string	해당 미션 수행 영상 URL

```
{
  "analysis_id": "string",
  "walk_id": "string",
  "missions": [
    {
      "mission_id": "string",
      "mission_type": "enum",
      "instruction": "string",
      "video_url": "string"
    }
  ]
}
```

Response

Field	Type	Description
analysis_id	string	분석 ID

walk_id string 산책 세션 ID
analyzed_at string (ISO8601) 분석 완료 시각
result object 돌발미션 분석 결과

```
{
  "analysis_id": "string",
  "walk_id": "string",
  "analyzed_at": "string",
  "result": {
    "missions": [
      {
        "mission_id": "string",
        "mission_type": "enum",
        "success": boolean,
        "confidence": float,
        "summary": "string"
      }
    ]
  }
}
```

📌 비동기
돌발미션 Job 생성
POST /api/missions/judge/jobs

Response - Job 접수
Field Type Description
analysis_id string 분석 ID
job_id string 비동기 작업 ID
status string queued
submitted_at string (ISO8601) 작업 접수 시각

```
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "string",
  "submitted_at": "string"
}
```

Job 상태 조회
GET /api/missions/judge/jobs/status/{job_id}

※ Job 상태 값은 다음 enum 중 하나를 가진다.
- queued → processing → finished
- failed (비정상 종료)

Response - 상태 전용

```
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "processing",
```

```
  "progress": 66
}
```

Job 상태 조회 - 실패
존재하지 않는 Job ID

```
{
  "status": "failed",
  "error": {
    "code": "INVALID_JOB_ID",
    "message": "존재하지 않는 Job ID입니다."
  }
}
```

Job 결과 조회
GET /api/missions/judge/jobs/result/{job_id}

status = finished 인 경우에만 유효
분석 완료 후 저장된 결과를 조회하여 반환

Response - 성공

```
{
  "analysis_id": "string",
  "job_id": "string",
  "analyzed_at": "string",
  "result": {
    "missions": [
      {
        "missions_id": "string",
        "missions_type": "enum",
        "success": true,
        "confidence": 0.92,
        "summary": "string"
      }
    ]
  }
}
```

Response - 미완료

```
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "processing",
  "message": "Analysis is not finished yet."
}
```

Job 결과 조회 - 실패
status = failed 인 경우

```
{
```

```
  "analysis_id": "string",
  "job_id": "string",
  "status": "failed",
  "error": {
    "code": "JOB_EXECUTION_FAILED",
    "message": "돌발미션 분석 Job이 실패했습니다."
  }
}
```


존재하지 않는 Job


```
{
  "analysis_id": "string",
  "status": "failed",
  "error": {
    "code": "INVALID_JOB_ID",
    "message": "존재하지 않는 Job ID입니다."
  }
}
```


1-4. 응답 오류 명세

공통 오류 응답 포맷

Field	Type	Description
analysis_id	string	분석 요청 ID
status	string	"failed"
error.code	string	오류 유형 식별 코드
error.message	string	오류 상세 설명


```
{
  "analysis_id": "string",
  "status": "failed",
  "error": {
    "code": "ERROR_CODE",
    "message": "string"
  }
}
```


오류 코드 예시

MISSION_VIDEO_INVALID
MISSION_INFERENCE_FAILED
MULTIMODAL_API_TIMEOUT
JOB_EXECUTION_FAILED
JOB_NOT_FINISHED
RESULT_NOT_AVAILABLE
INVALID_JOB_ID

1-5. 역할 및 연동 관계

역할

돌발미션 모델 API는 산책 중 촬영되어 저장된 돌발미션을 대상으로 상용 멀티모달 API를 활용해 각 미션 수행 여부를 영상의미 이해 기반으로 판단하는 AI 추론 전용 컴포넌트이다.

규칙 기반 포즈 판정이 아닌 영상 전체 맥락과 텍스트 지시어(mission_type)를 함께 고려해 고수준 판단 결과를 제공한다.

역할 분리
① FastAPI (AI Domain)

돌발미션 분석 Job 실행
상용 멀티모달 API 호출
미션별 결과 JSON 생성
결과를 DB / Object Storage에 저장
② Backend API

Job 생성 요청
Job 상태 조회 (/jobs/status/{job_id})
finished 상태 확인 후 결과 조회 API 호출 (/jobs/result/{job_id})
Client 응답 구성

연동 관계
호출 주체: 메인 Backend API 서버
호출 시점: 사용자가 산책 완료 버튼을 누른 직후
입력 데이터:
DB에 저장된 돌발미션 영상 URL
mission_type 및 미션 메타데이터
출력 데이터:
미션별 성공 여부
판단 신뢰도
사용자 노출용 요약 설명

동작 흐름 (동기)
산책 중 촬영된 각 돌발미션 영상은 즉시 DB/스토리지에 저장
사용자가 산책 완료 버튼 클릭
Backend API 서버가 /api/missions/judge 호출
모델 API가 요청 처리 흐름 내에서 각 미션 영상에 대해 상용멀티모달 API를 동기적으로 호출
모든 돌발미션 영상 분석이 완료될 때까지 대기
분석 결과를 한 번에 Backend API 서버로 응답
Backend가 산책 완료 페이지에 필요한 돌발미션 결과를 즉시 Client에 전달

동작 흐름 (비동기)
산책 중 돌발미션 영상 DB/스토리지 저장
산책 완료 시 Backend → /api/missions/judge/jobs 호출
FastAPI에서 Job 등록 (queued)
Worker가 분석 수행 (processing)
분석 완료 후 결과 JSON 저장 (finished)
Backend → /api/mission/judge/jobs/status/{job_id} 폴링
status = finished 확인
Backend → /api/mission/judge/jobs/result/{job_id} 호출

저장된 JSON 기반 결과 수신
Client에 산책 완료 결과 일괄 제공

데이터 귀속 및 저장
돌발미션 분석 결과는 산책 단위 분석 결과의 하위 도메인 데이터로 관리된다.
모든 돌발미션 결과는 analysis_id , walk_id를 기준으로 귀속된다.

단계 2
📌 모델 추론 성능 최적화
2-1. 범용 멀티모달 LLM 기반 영상 이해 API 후보 선정
돌발미션에서 AI의 역할은 단순한 객체 검출이 아닌 영상 속 반려견의 행동을 의미적으로 이해하고 특정 미션을 실제로 수행했는지 성공/실패 여부를 판단하는 것이 핵심이다.

행동 인식 모델을 자체적으로 구축할 경우, 대규모 행동 라벨 영상 데이터를 확보해야 하며, 시계열 모델링과 미션별 평가 기준 정의가 필요하며, 이는 초기 개발 비용과 리스크가 크다.

따라서 정확도, 개발 속도, 운영 안정성을 동시에 확보할 수 있는 외부 AI API를 활용하고자 한다.

선정 기준

단순 객체 인식이 아닌 영상 속 행동의 의미를 이해할 수 있다.
프롬프트로 판정 기준을 명시적으로 제어할 수 있다.
미션 성공 여부에 대한 결정형 결과를 도출할 수 있다.
주요 후보

Google Gemini(video understanding)
멀티모달 LLM 기반으로 영상과 자연어 지시를 함께 이해 가능
프롬프트를 통해 돌발미션 성공여부 판정 기준 명시 가능
Google 생태계 기반으로 안정성 확보 가능
TwelveLabs
영상 전용 Video Understanding API로 행동 인식에 특화
행동 분석 정확도 중심의 비교 대상 API로 활용 가능

2-2. 성능 측정 지표
구분 지표명 측정 방법 필요성
판정 정확도 미션 판정 정확률 (Accuracy) 정답과 일치한 영상 수 / 전체 영상 수
미션 성공/실패 판정 성능 확인
응답 품질 판정 명확성 결과가 YES/NO로 명확히 도출되는지 서비스 로직 연동 가능성
판정 근거 설명 가능 여부 자연어 설명 제공 여부 사용자 피드백 및 디버깅
시간 성능 응답 지연 시간 요청 → 결과까지 소요 시간(초) 산책 중 사용자 경험
안정성 API 호출 성공률 정상 응답 수 / 전체 요청 수 서비스 장애 리스크

2-3. 실험 환경
테스트 영상 :

생성형 AI(SORA)를 활용하여 미션 타입별 정답 영상 생성(5초)
테스트 프롬프트 템플릿(Twelvelabs)

You are a professional dog training judge.

Your task is to determine whether the dog SUCCESSFULLY completed the given training mission.
You must judge ONLY the specified mission type.
Do NOT compare with or infer other missions.

==========
MISSION DEFINITIONS
==========

[SIT]
Success Criteria:
- The dog's hips clearly touch the ground.
- The front legs support the body.
- The sitting posture is maintained continuously for at least 1.0 second.
Fail Criteria:
- The hips do not touch the ground, OR
- The posture is held for less than 1.0 second.

[DOWN]
Success Criteria:
- The dog's chest or belly touches the ground.
- The front legs are bent and resting.
- The posture is maintained continuously for at least 1.0 second.
Fail Criteria:
- The chest does not touch the ground, OR
- The posture is held for less than 1.0 second.

[PAW]
Success Criteria:
- One front paw is lifted off the ground.
- The paw is placed on a visible human hand.
- Contact is maintained continuously for at least 0.5 seconds.
Fail Criteria:
- The paw is lifted without contacting a human hand, OR
- Contact is unclear or too brief.

[SPIN]
Success Criteria:
- The dog rotates its body at least 180 degrees.
- The rotation is continuous and intentional.
Fail Criteria:
- Only partial or accidental turning.

[JUMP]
Success Criteria:
- All four paws leave the ground at the same time.
- A clear airborne moment is visible.
Fail Criteria:
- Only front paws leave the ground, OR
- The airborne moment is unclear.

==========
JUDGMENT RULES
==========
- Use ONLY visual information from the video.
- Judge ONLY the mission specified below.
- If the posture or motion cannot be confidently verified, return UNCERTAIN.
- Do NOT mention or evaluate other missions.
- Do NOT explain training quality or intent.

Return ONLY raw JSON in the following format.
Do NOT use markdown. Do NOT include extra text.

```
{
  "mission": "<MISSION>",
  "verdict": "SUCCESS | FAIL | UNCERTAIN",
  "hold_duration_sec": number | null,
  "evidence": [
    {
      "time_sec": number,
      "description": string
    }
  ],
  "confidence": number
}
```

==========
CURRENT_MISSION: SIT
==========

2-4. 외부 AI API 성능 비교 결과

구분	지표명	Google Gemini	TwelveLabs
판정 정확도	미션 판정 정확률 (Accuracy)	100%	100%
응답 품질	판정 명확성	높음 (SUCCESS/FAIL 명시, JSON 고정 포맷)	높음 (SUCCESS 명시, 구조화 JSON)
	판정 근거 설명 가능 여부	가능 (evidence + 시간 정보 제공)	가능 (evidence + 시간 정보 제공)
시간 성능	응답 지연 시간	10.97 sec(평균)	웹 기준 ~1–2 sec (관찰)* 동일한 영상을 업로드했을 때, 용량 제한 이슈로 웹에서 측정
안정성	API 호출 성공률	100%	100% (웹 콘솔 성공)

2-5. 외부 AI API 예상 비용
Google Gemini
토큰 기반 과금 : 영상 길이 × 프롬프트 길이 × 결과 길이 모두 비용에 영향
Gemini 2.5 Flash-Lite
입력(영상 포함): 약 $0.10 / 1M tokens
출력(판단 결과): 약 $0.40 / 1M tokens
TwelveLabs
분 단위 과금 : 영상 길이를 기준으로 비용이 정비례
Video Indexing / Analysis
약 $0.042 / 분(video)
인프라 비용: 약 $0.0015 / 분

구분 Google Gemini TwelveLabs
과금방식 토큰 기반 과금 분 단위 과금
단가 입력 : $0.10 / 1,000,000 토큰출력 : $0.40 / 1,000,000 토큰 $0.042 per minute(영상 indexing/analysis)$0.015 per minute (인프라)
영상 1개당 예상 비용 $0.0002 ~ $0.00025 $0.0036
비고 영상 1개당영상 입력 : 1,400 토큰프롬프트 : 600 토큰출력 : 100 토큰총 약 2,100 토큰 비용 고정

단계 3
📌 서비스 아키텍처 모듈화
3-1. 모듈화 적용 후 전체 아키텍처 개요
목표 구조

돌발미션 AI는 비동기 Job 기반 분석을 기본 처리 방식으로 한다.
영상 분석 요청은 Job 단위로 큐잉되며, Worker가 백그라운드에서 처리한다.
미션별 판정 결과는 Job 완료 이벤트 단위로 반환된다.
AI 도메인은 미션 판정(추론) 책임만 수행하고, 저장/응답 통합은 외부에서 처리한다.
구성요소 목록

AI Domain
AI Orchestrator
돌발미션 Job 수신 및 큐잉
Worker 실행 제어 및 상태 이벤트 관리
Message Queue
산책 종료 시점에 생성된 돌발미션 분석 Job 버퍼
Mission Worker
미션 영상 단위 분석 파이프라인 실행
상용 멀티모달 AI 호출
External Multimodal AI
입력: 영상 URL + 미션 지시어
출력: 성공/실패, 신뢰도, 요약 설명

3-2. 모듈별 책임(domain)과 분리 이유
AI Orchestrator

돌발미션 분석 Job 생성 및 관리
Job을 Queue에 enqueue
Worker 실행 상태(queued / processing / completed / failed) 추적
Worker 결과를 수신하여 상위 시스템으로 전달
분리 이유
미션 판정 로직과 Job 운영(큐, 재시도, 타임아웃)을 분리
미션 수 증가·트래픽 증가 시에도 안정적인 운영 가능
Message Queue

돌발미션 분석 Job을 Worker에게 전달하는 비동기 버퍼
산책 종료 시점 요청 폭주를 흡수
분리 이유
미션 개수(3~5개)에 따른 외부 AI 호출 지연을 동기 처리에서 분리
Job 단위 병렬 처리 가능
Mission Worker

돌발미션 영상 URL 기반 영상 로딩
미션 유형(SIT, DOWN, PAW, TURN, JUMP)과 영상 의미 결합
상용 멀티모달 AI 호출
미션별 성공 여부 판단 결과 생성
분리 이유
외부 멀티모달 API 호출 비용·지연을 격리
Worker 단위 확장으로 처리량 조절 가능

3-3. 모듈 간 인터페이스 설계(계약)
Client ↔ Backend (Public REST)
Job 생성 POST /api/mission/judge/jobs

Request : JSON
산책 종료 시점에 호출
산책 중 DB/스토리지에 저장된 돌발미션 영상 목록 전달
Response : { "job_id": "string", "status": "queued" }
상태 조회 GET /api/mission/judge/jobs/{job_id}

Response: { job_id, status, progress{stage, percent, message} }
Backend가 Job 상태 캐시를 조회 후 반환
결과 조회

GET /api/mission/judge/jobs/{job_id}/result

Response: 돌발미션 분석 결과 JSON
Job 완료 이후에만 호출 가능

Backend ↔ AI Orchestrator (Internal REST)
대용량 파일 전송 금지

영상은 video_url 참조 전달

돌발미션 분석은 Job 단위로 위임

Job 생성 요청 POST /internal/ai/mission/jobs

Request(예시) :

```
{
  "job_id": "job_mission_001",
  "job_type": "MISSION_JUDGE",
  "missions": [
    {
      "mission_id": "mission_01",
      "mission_type": "SIT",
      "video_url": "https://cdn.service.com/missions/sit_01.mp4"
    },
    {
      "mission_id": "mission_02",
      "mission_type": "PAW",
      "video_url": "https://cdn.service.com/missions/paw_02.mp4"
    }
  ],
  "callback_url": "https://backend/internal/mission/jobs/job_mission_001/events",
  "meta": {
    "analysis_id": "analysis_001",
    "walk_id": "walk_001",
    "requested_at": "2026-01-09T13:00:00+09:00"
  }
}
```

Response(예시) :

```
{
  "accepted": true,
  "job_id": "job_mission_001"
}
```

Orchestrator → Backend(Callback: 상태 / 결과 전달)

돌발미션 분석 Job의 상태/진행/완료/실패 이벤트를 단일 콜백 엔드포인트로 통합한다.
POST /internal/mission/jobs/{job_id}/events
event_type: STATUS | PROGRESS | COMPLETE | FAILED
예시 1) queued

```
{
  "event_type": "STATUS",
  "status": "queued",
  "progress": {
    "stage": "QUEUED",
    "percent": 0,
    "message": "돌발미션 분석 대기열에 등록되었습니다."
  }
}
```

예시 2) progress

```
{
  "event_type": "PROGRESS",
  "status": "processing",
  "progress": {
    "stage": "MISSION_INFERENCE",
    "percent": 66,
    "message": "상용 멀티모달 AI로 미션을 판정 중입니다."
  }
}
```

예시 3) complete

```
{
  "event_type": "PROGRESS",
  "status": "processing",
  "progress": {
    "stage": "MISSION_INFERENCE",
    "percent": 66,
    "message": "상용 멀티모달 AI로 미션을 판정 중입니다."
  }
}
```

예시 4) failed

```
{
  "event_type": "FAILED",
  "status": "failed",
  "progress": {
    "stage": "MISSION_INFERENCE",
    "percent": 40,
    "message": "미션 분석 중 오류가 발생했습니다."
  },
  "error": {
    "code": "MULTIMODAL_API_TIMEOUT",
    "message": "외부 AI 호출 시간이 초과되었습니다."
  }
}
```

Backend 처리 규칙

STATUS / PROGRESS: Job 상태 갱신

COMPLETE: 돌발미션 결과 저장 + Job 완료 처리

FAILED: 실패 상태 기록 및 오류 로그 저장

보안 / 신뢰성

콜백 인증: 서비스 토큰 또는 HMAC

콜백 실패 시 Orchestrator 재시도

Backend는 job_id + event_type 기준 멱등 처리

Orchestrator ↔ Queue (Job 메시지)

영상 데이터 직접 전달 금지

Worker는 video_url로 영상 참조

{

```
  "job_id": "job_mission_001",
  "job_type": "MISSION_JUDGE",
  "missions": [
    {
      "mission_id": "mission_01",
      "mission_type": "SIT",
      "video_url": "https://.../sit.mp4"
    }
  ]
}
```

Worker → Orchestrator (결과 보고)
미션 단위 결과를 묶어 Job 결과로 반환

```
{
  "job_id": "job_mission_001",
  "status": "completed",
  "result": {
    "missions": [
      {
        "mission_id": "mission_01",
        "success": true,
        "confidence": 0.92,
        "summary": "지시에 맞춰 안정적으로 앉았습니다."
      }
    ]
  }
}
```

3-4. 데이터 흐름 (비동기 Job 시나리오)
산책 중 돌발미션 영상이 저장됨
산책 종료 시점에 돌발미션 분석 Job 생성
Orchestrator가 Job을 Queue에 등록
Worker가 Job 수신
각 미션 영상에 대해 상용 멀티모달 AI 호출
미션별 성공/실패 판단
Job 완료 이벤트로 결과 전달

3-5. 독립 개발/배포/스케일링 고려사항
독립 배포 단위
AI Orchestrator
Mission Judge Worker
스케일링 전략
Worker 수를 Job 큐 길이 기준으로 확장
미션 개수 증가 시 Worker 병렬성으로 대응
멱등성 / 중복 처리
job_id 기준 결과 처리
동일 영상 + 동일 미션 중복 요청 시 재분석 방지 가능

3-6. 모듈화로 기대되는 효과와 장점
외부 멀티모달 API 지연 격리
산책 종료 트래픽 스파이크 흡수
미션 수 증가에도 안정적인 처리
AI 기능 단독 진화 가능 (프롬프트/모델 교체)

3-7. 서비스 시나리오 부합 근거
사례 1) 미션 개수 증가(3 → 5개)

Job 단위 처리로 응답 구조 유지
Worker 확장으로 처리 시간 증가 완화
사례 2) 외부 멀티모달 API 지연

동기 요청 차단 없이 Job 큐에서 흡수
타임아웃/재시도 정책을 Worker 레벨에서 통제
사례 3) 미션 판정 기준 변경

Worker 내부 프롬프트/판정 로직만 수정
서비스 전체 구조 영향 없음

돌발미션

단계 4
📌 멀티스텝 AI/파이프라인 구현 검토

돌발미션 영상분석을 상용 멀티모달 AI API에 전달하여 미션 성공/실패 여부를 판정하는 멀티스텝 파이프라인을 정의한다.

4-1. 멀티스텝 파이프라인 구성단계(Stage)
돌발미션 분석 Job은 산책 1회 단위 Job이며, Job 내부에서 mission 단위 파이프라인이 반복 실행된다.

Stage 구성
VALIDATING
FETCHING_VIDEO
MISSION_INFERENCE
RESULT_PARSING
FINALIZING

4-2. 멀티스텝으로 나눈 근거
운영 단계 분리 기술적으로는 단일 추론 문제지만, 서비스 관점에서는 잘못된 입력 요청/다운로드 실패/외부 멀티모달 API 타임아웃/파싱 실패 등을 하나의 실패를 운영이 불가 진행율 판단 진행율 판단을 위해 의미 있는 stage 경계가 필요
비용 절약과 장애 격리를 위한 조기 종료 잘못된 영상이나 정책 위반 요청을 외부 멀티 모달 API까지 보내는 것은 불필요한 비용 VALIDATING / FETCHING 단계에서 실패하면 즉시 종료하여 비용과 지연을 절약

4-3. 진행률(Percent) 보고
진행률은 Job rlwns 0~100%로 정의 mission 단위로 누적 계산

MISSION_INFERENCE에서 실제 체감 지연과 병목이 일어나기 때문에 가장 큰 비중을 둠

Stage Percent
VALIDATING 0 ~ 10
FETCHING_VIDEO 10 ~ 25
MISSION_INFERENCE 25 ~ 85
RESULT_PARSING 85 ~ 95
FINALIZING 95 ~ 100

이벤트 (progress) 송신 위치(콜백 사용 위치)
이벤트 송신 주체
Mission Worker → AI Orchestrator
AI Orchestrator → Backend (/events 콜백)
(Worker는 Backend를 직접 호출하지 않는다.)
이벤트 송신 규칙
stage 시작 시: PROGRESS

stage 종료 후 다음 stage 진입 시 percent 갱신

최종:

성공 → COMPLETE
실패 → FAILED
예시)

```
{
  "event_type": "PROGRESS",
  "status": "processing",
  "progress": {
    "stage": "MISSION_INFERENCE",
    "percent": 60,
    "message": "상용 멀티모달 AI로 돌발미션을 판정 중입니다."
  }
}
```

4-4. Stage 설계
공통 정책

max_duration_sec = 5
target_fps = 12
max_resolution = 720p
Worker는 대용량 전달 금지: video_url로 다운로드, 결과는 URL 참조로 저장
Backend 저장 규칙: Redis=status/progress, MySQL=result
Stage별 상세

Stage 역할 입력 출력 병목/리스크 실패 조건 → 에러
VALIDATING 요청/정책 검증, 조기 종료 job payload(video_url, mission_type) validated payload 거의 없음 정책 위반 → VIDEO_TOO_LONG(400), INVALID_MISSION_TYPE(400)
FETCHING_VIDEO 영상 다운로드/접근성 확인 video_url local_path 또는 stream 네트워크 지연, 만료 URL 다운로드 실패 → VIDEO_FETCH_FAILED(502/504)
MISSION_INFERENCE 상용 멀티모달 API 호출 video_url, mission_type, prompt raw AI response 외부 API 지연/타임아웃 타임아웃/에러 → MULTIMODAL_API_TIMEOUT, MISSION_INFERENCE_FAILED
RESULT_PARSING AI 응답 파싱/검증 raw response structured result JSON 불안정 파싱 실패 → MISSION_RESULT_PARSE_FAILED
FINALIZING 결과 조립, 이벤트 송신 mission result final result JSON 거의 없음 콜백 실패 → Orchestrator 재시도

4-5. 실패 기준
Mission 실패
VALIDATING 실패
FETCHING_VIDEO 실패
MISSION_INFERENCE 실패
RESULT_PARSING 실패
하나라도 발생하면 해당 mission은 FAILED

(실패 시에도 Job 자체는 계속 진행해 다른 mission은 정상 분석)

4-6. Worker Pipeline Runner 설계
구성 요소
PipelineRunner : stage 순서 관리, 공통 타이밍/로깅/예외 처리
VideoFetcher : video_url 다운로드
MultimodalClient : 상용 멀티모달 AI API 호출
ResultParser : JSON 파싱 및 스키마 검증
이벤트 송신 규칙
Worker → Orchestrator 내부 이벤트
Orchestrator → Backend 콜백 송신
Worker는 PROGRESS / COMPLETE / FAILED만 생성
파이프라인 러너 의사 코드

```
class PipelineRunner:
    def run(self, mission_ctx):
        try:
            emit_progress("VALIDATING", 5, "입력 정보를 확인하고 있습니다.")
            validate(mission_ctx)

            emit_progress("FETCHING_VIDEO", 15, "영상을 불러오고 있습니다.")
            fetch_video(mission_ctx.video_url)

            emit_progress("MISSION_INFERENCE", 60, "미션 수행 여부를 판정 중입니다.")
            raw = call_multimodal_api(
```

```
            video_url=mission_ctx.video_url,
            mission_type=mission_ctx.mission_type
        )

        emit_progress("RESULT_PARSING", 90, "분석 결과를 정리하고 있습니다.")
        result = parse_result(raw)

        emit_progress("FINALIZING", 99, "결과를 저장하고 있습니다.")
        emit_complete(result)

    except KnownError as e:
        emit_failed(e.stage, e.percent, e.code, e.message)
    except Exception as e:
        emit_failed("FINALIZING", 99, "INTERNAL_ERROR", "처리 중 오류가
발생했습니다.")
```

4-7. stage 별 구현 포인트
validating
Backend 에서 이미 mp4만 받더라도,

Worker 레벨에서 한 번 더 정책을 검증해 조기 종료한다.

잘못된 요청은 외부 멀티모달 API 호출 없이 즉시 실패
비용 절약 + 장애 전파 차단 목적

```
def _run_validating(self):
    p = self.job.policy

    if p.max_duration_sec > 5:
        raise KnownError(
            stage="VALIDATING",
            percent=2,
            code="VIDEO_TOO_LONG",
            message="돌발미션 영상 길이가 정책을 초과했습니다.",
            details={"max_duration_sec": 5}
        )

    if self.job.mission.mission_type not in {"SIT", "DOWN", "PAW", "TURN", "JUMP"}:
        raise KnownError(
            stage="VALIDATING",
            percent=3,
            code="INVALID_MISSION_TYPE",
            message="지원하지 않는 미션 유형입니다.",
            details={"mission_type": self.job.mission.mission_type}
        )

    return p
```

fetching_video

```
video_url 기반 접근성 검증
접근 가능한 URL인지 확인
def _run_fetching_video(self, video_url: str):
    try:
        # 전체 다운로드 대신 접근성 체크
        self.video_client.check_access(video_url)
        return {"video_url": video_url}
    except Exception as e:
        raise KnownError(
            stage="FETCHING_VIDEO",
            percent=15,
            code="VIDEO_FETCH_FAILED",
            message="돌발미션 영상을 불러올 수 없습니다.",
            details={"video_url": video_url, "reason": repr(e)}
        )

mission_inference
상용 멀티모달 AI API 호출
video_url + mission_type + 고정 프롬프트
def _run_mission_inference(self, video_url: str, mission_type: str):
    try:
        response = self.multimodal_client.judge_mission(
            video_url=video_url,
            mission_type=mission_type,
            timeout_sec=self.job.policy.inference_timeout_sec,
        )
        return response

    except TimeoutError:
        raise KnownError(
            stage="MISSION_INFERENCE",
            percent=70,
            code="MULTIMODAL_API_TIMEOUT",
            message="미션 판정 시간이 초과되었습니다."
        )
    except Exception as e:
        raise KnownError(
            stage="MISSION_INFERENCE",
            percent=70,
            code="MISSION_INFERENCE_FAILED",
            message="미션 판정 중 오류가 발생했습니다.",
            details={"exception": repr(e)}
        )

result_parsing
JSON 구조/필수 필드 검증
def _run_result_parsing(self, raw_response: dict):
    try:
```

```
        success = bool(raw_response["success"])
        confidence = float(raw_response["confidence"])
        summary = str(raw_response["summary"])
    except Exception as e:
        raise KnownError(
            stage="RESULT_PARSING",
            percent=90,
            code="MISSION_RESULT_PARSE_FAILED",
            message="미션 판정 결과를 해석할 수 없습니다.",
            details={"raw_response": raw_response, "error": repr(e)}
        )

    return {
        "success": success,
        "confidence": confidence,
        "summary": summary
    }
```

finalizing
구조를 단순화하되, processing / model_version 기록은 유지

```
from datetime import datetime, timezone

def iso_now():
    return datetime.now(timezone.utc).isoformat()

def now_ms():
    return int(datetime.now(timezone.utc).timestamp() * 1000)

def _finalize(
    self,
    mission_result: dict,
    started_ms: int,
):
    finished_ms = now_ms()
    analysis_time_ms = max(0, finished_ms - started_ms)

    job_id = self.job.job_id
    analysis_id = self.job.analysis_id or job_id

    final = {
        "analysis_id": analysis_id,
        "job_id": job_id,
        "analyze_at": iso_now(),
        "processing": {
            "analysis_time_ms": analysis_time_ms,
            "video_duration_sec": self.job.policy.max_duration_sec
        },
        "result": {
```

```
            "mission_id": self.job.mission.mission_id,
            "mission_type": self.job.mission.mission_type,
            "success": mission_result["success"],
            "confidence": mission_result["confidence"],
            "summary": mission_result["summary"]
        },
        "model_version": self._compose_model_version()
    }
    return final
def _compose_model_version(self):
    pipeline_ver = getattr(self.job, "pipeline_version", "mission_pipeline_v1")
    model_ver = getattr(self.multimodal_client, "model_name", "external_multimodal")
    return f"{pipeline_ver}:{model_ver}"
```

4-8. 멀티스텝 파이프라인 다이어그램
돌발미션 파이프라인 (1)

단계 5
📌 데이터/컨텍스트 보강 설계

돌발미션 기능은 촬영된 짧은 영상이 특정 미션을 수행했는지를 일관된 기준으로 판정하는 결정형 추론 서비스이다. 따라서 기준을 명시적으로 제공하는 '경량 컨텍스트 보강(RAG-lite)' 형태가 적합하다.

5-1. 컨텍스트 보강이 필요한 근거
미션 해석의 모호성
판정 기준의 일관성 유지

5-2. RAG-lite 적용 전략
검색 대상 문서가 없으며 자유 텍스트 생성이 목적이 아니므로 RAG는 부적합하다.

영상 생성이나 보강이 아닌 판정 문제로 Visual/Video RAG는 비용이나 지연을 증가시킬 수 있다.

미션 종류가 명확한 enum이며, 판정 기준이 사전에 정의 가능하다.

모델의 판정 해석 기준만 보강하기 위해 Rule/Reference 기반 RAG를 도입한다.

미션 유형(mission_type)에 대응되는 성공 기준, 실패 기준을 모델 입력 컨텍스트로 동적 삽입하여 판정의 일관성을 강화한다.

5-3. 전체 데이터 흐름
[Mission Video + mission_type]
↓
[Mission Reference Store 조회]
(해당 미션의 성공/실패 기준)
↓

[Inference Prompt 구성]
(영상 + 판정 기준 포함)
↓
[상용 멀티모달 AI API 호출]
↓
[success / confidence / summary]

5-4. Mission Reference Store 설계
데이터 성격
정적(reference) 데이터
미션 유형별 “판정 기준 컨텍스트” 저장, 학습 데이터가 아닌 판정 기준 데이터
운영 중 수정 가능 (프롬프트 재배포 불필요)
외부 검색이 불필요하고, 프롬프트 일관성 확보에 효과적
저장 형태
JSON 파일
mission_type 기준 Key-Value 조회
예시 스키마

```
{
  "mission_type": "PAW",
  "mission_name": "손",
  "success_criteria": [
    "반려견의 앞발이 사람 손 위에 명확히 올라가 있을 것",
    "앞발이 최소 1초 이상 유지될 것"
  ],
  "failure_cases": [
    "앞발을 잠깐 들었으나 손과 접촉하지 않음",
    "동작이 프레임 밖에서 발생함"
  ],
  "confidence_guideline": {
    "high": "기준을 명확히 충족",
    "medium": "시도는 있으나 일부 기준 미충족",
    "low": "동작이 불명확하거나 실패"
  },
  "capture_hint": "사람 손이 영상 프레임 안에 보이도록 촬영"
}
```

5-5. 모델 입력(프롬프트) 증강
컨텍스트 보강 입력 예시)

```
[미션 지시어]
- 미션: PAW (손)

[성공 기준]
- 앞발이 사람 손 위에 올라가 있어야 함
- 최소 1초 이상 유지

[실패 기준 예시]
- 앞발을 잠깐 들었으나 접촉 없음
```

[판정 요청]
위 기준을 바탕으로 영상에서 미션 수행 여부를 판단하고,
success / confidence / summary를 반환해줘.

5-6. 모델 통합 방식
프롬프트 구성 원칙
역할 명확화: “판정자” 역할 고정
기준 명시: 성공/실패 조건을 bullet로 제공
출력 형식 고정: JSON Schema 강제
추측 금지: 보이지 않으면 실패 처리

5-7. 프롬프트 템플릿 예시
You are an AI judge that determines whether a dog successfully performed
a specific training mission based on a short video.

[Mission Information]
- Mission Type: {{MISSION_TYPE}}
- Mission Name: {{MISSION_NAME}}

[Success Criteria]
{{SUCCESS_CRITERIA_LIST}}

[Failure Examples]
{{FAILURE_CASES_LIST}}

[Confidence Guideline]
{{CONFIDENCE_GUIDELINE}}

[Instruction]
Based on the video and the criteria above:
1. Decide whether the mission was successfully performed.
2. Assign a confidence score between 0.0 and 1.0.
3. Provide a short, user-friendly explanation.

If the action is unclear or partially visible, mark it as failure.

[Output Format]
Return ONLY the following JSON format:

```
{
  "success": boolean,
  "confidence": number,
  "summary": string
}
```

5-8. 도입 효과 및 검증 계획
도입 전

판정 기준이 암묵적
confidence 변동성 큼
도입 후

미션별 판정 기준 고정
confidence 분포 안정화
결과 재현성 향상
검증 방법

동일 영상 재판정 consistency 비교
미션별 confidence 수치에 비해 설명이 과하거나 부족하지 않은지 확인

단계 6
📌 표준화된 도구 통합 및 외부 API 활용 설계

6-1. 설계 목적
돌발미션 기능은 반려견의 행동 의미를 이해하여

성공/실패 여부를 판단하는 영상 의미 이해(Video Understanding)가 핵심이므로,

자체 모델 구축이 아닌 상용 멀티모달 AI API 활용 전략의 타당성을 중점적으로 검토한다.

외부 시스템 및 상용 서비스와의 상호작용
구분 설명
영상 분석 API 돌발미션 영상에서 반려견 행동을 의미적으로 이해하여 성공/실패 판단
객체/행동 이해 SIT, PAW, TURN, JUMP 등 훈련 동작의 의미 판별
스토리지 촬영 영상은 S3 기반 Object Storage에 저장 후 URL 전달
외부 상용 AI API Gemini 또는 TwelveLabs 기반 멀티모달 영상 분석

6-2. MCP 도입 여부 결정
MCP 도입 여부: 도입하지 않음

본 기능에서는 MCP(Model Context Protocol)를 도입하지 않는다.

돌발미션 기능에서는 모델이 판단만 수행하고, 어떤 API를 호출할지는 시스템에서 정한다.

따라서 MCP와 같은 복잡한 도구 제어 구조가 필요하지 않기 때문에 기존 REST 기반 API 호출 구조가 가장 합리적이라고 판단하였다.

근거
모델 주도 도구 선택 구조가 아님
돌발미션은 “지정된 영상 → 지정된 API 호출” 구조
모델이 스스로 여러 도구 중 하나를 선택할 필요가 없음
외부 API 종류가 제한적
Gemini 또는 TwelveLabs 중 1개 선택
다수 도구를 동적으로 연결·관리할 필요가 없음
판단 결과가 단순

성공 / 실패 + 신뢰도
복합 Reasoning + Tool Chaining 구조가 아님

6-3. 상용 API 사용 전략

돌발미션은 고수준 행동 의미 이해(Video Semantic Understanding)가 핵심이며, 이를 위해 상용 멀티모달 API 사용이 합리적이라고 판단하였다.

자체 모델 구축의 한계
항목 한계
데이터 반려견 행동별 대규모 고품질 영상 데이터 부족
학습 비용 GPU 인프라 구축 및 학습 비용 부담 큼
일반화 다양한 견종·촬영 각도·환경 대응 어려움
개발 기간 MVP/과제 일정 내 구현 난이도 높음

상용 API 사용 근거
영상 의미 이해 성능

TwelveLabs / Gemini는 대규모 사전학습 기반
“행동의 의미” 수준에서 판단 가능
인프라 비용 절감

GPU 서버 상시 운영 불필요

사용량 기반 과금 → 트래픽 변동에 유리

(국내 서비스 초기 런칭 패턴에 적합)

빠른 구현 및 안정성

API 호출만으로 즉시 서비스 연동 가능
SLA 및 운영 안정성 확보
서비스 품질 우선 전략

사용자 체감 정확도가 가장 중요
모델 소유보다 “판단 결과 신뢰도”가 핵심 가치

6-4. TwelveLabs vs Gemini 비교 및 사용 전략
항목 TwelveLabs Gemini
전문성 Video Understanding 특화 범용 멀티모달
행동 의미 분석 영상 중심 강점 언어 기반 의미 판단 강점
프롬프트 유연성 제한적 높음
추론 설명력 중간 높음
비용 구조 영상 분석 단가 기반 토큰/요청 기반
적합 시나리오 대량 영상 분석 돌발미션 성공/실패 판단

6-5. API 실패 대응 및 안정성 확보 전략
API 실패 및 Fallback 전략

외부 API 호출 실패 시 재시도를 수행한다.
돌발미션 분석을 위해 외부 API를 호출한다.
타임아웃 또는 에러가 발생하면 즉시 실패 처리하지 않는다.
정해진 횟수(2회)까지 재시도한다.
재시도 사이에는 짧은 대기 시간(2초)을 둔다.
반복 실패 시 해당 미션은 실패 처리하되 시스템 오류로 구분하여 기록한다.
사용자 화면에서는:
“이번 미션은 분석에 실패했습니다” 또는 “실패 처리”
내부 시스템에서는:
failure_type = SYSTEM_ERROR
reason = EXTERNAL_API_TIMEOUT 와 같이 기록

6-6. 확장성 고려
상용 API 호출 로직은 전용 Client 모듈로 분리한다.
“분석 흐름을 관리하는 코드”와 “외부 API를 실제로 호출하는 코드”를 분리한다
MissionOrchestrator
→ VisionApiClient.analyze(video_url)
→ GeminiClient
→ TwelveLabsClient
Client 모듈 : 외부 API와의 통신 전담

API 엔드포인트 관리
인증 키 처리
요청/응답 포맷 변환
타임아웃, 재시도, 에러 처리
Orchestrator

어떤 미션을 분석할지
결과를 어떻게 저장할지
향후 새로운 영상 분석 API 추가 시 Orchestrator 수정 없이 확장 가능하도록 설계한다.

모델 주도형 도구 선택이 필요해질 경우 MCP 도입을 검토할 수 있는 구조적 여지를 유지한다.

표정분석 설계

yeonjung77 edited this page 2 hours ago · 50 revisions
🐶 반려견 표정 감정 분석
📌 개요
산책 후 촬영된 반려견 표정 영상을 기반으로
대표 프레임 추출 → 얼굴 감정 분석 → 사용자 친화 설명 제공을 수행하는
AI 모델 API의 설계 및 성능 최적화 전략을 정의한다.

단계 1
📌 모델 API
본 API는 산책 후 촬영된 반려견 표정 영상을 입력으로 받아, 얼굴 탐지 기반 대표 프레임을 선택하고 해당 얼굴 영역을 감정 분류하여 사용자 친화적 설명을 제공한다.
비동기 처리 시 분석 상태 관리와 분석 결과 제공을 분리된 API 역할로 제공한다.

1-1. API 설계 개요
동기처리
단일 요청–응답으로 얼굴 탐지 및 감정 분석 결과를 즉시 반환
비동기 (Job) 확장
얼굴 탐지 및 감정 분석 작업을 백그라운드 Job으로 분리
Job 상태 조회 API와 분석 결과 조회 API를 분리하여 제공
역할 분리 원칙
Job 상태 관리 API: 분석 진행 상태만 제공
결과 접근 API: 분석 완료 후 저장된 JSON 결과 제공

1-2. Endpoint 목록 및 기능 설명
동기
구분 Method Endpoint Description
얼굴 탐지 POST /api/face/detect 영상에서 반려견 얼굴을 탐지하고 대표 프레임 1장을 선택하여 URL을 반환
감정 분석 실행 POST /api/face/emotion-inference 대표 프레임의 얼굴 영역을 입력받아 감정 분석 결과 JSON 반환
/api/face/detect 설명

영상에서 일정 간격으로 프레임을 추출하고 YOLO 이용해 반려견 얼굴 Object Detection
프레임 점수 계산 후 점수가 가장 높은 1장을 대표 프레임으로 선택(얼굴 검출 confidence, bounding box 크기, 얼굴 중심 정렬 여부, 블러 정도, 명암 등)
대표 프레임을 스토리지에 저장하고 URL 반환
/api/face/emotion-inference 설명

입력받은 대표 프레임에 대해 YOLO 강아지 얼굴 Object Detection 결과 기준 얼굴 영역 crop
감정 분류 모델 inference
감정 확률, 예측 감정, 사용자 친화적 설명 생성해 JSON으로 반환

비동기
구분 Method Endpoint Description

얼굴 탐지 Job 생성 POST /api/face/detect/jobs 얼굴 탐지 및 대표 프레임 선택 Job 생성
얼굴 탐지 Job 상태 조회 GET /api/face/detect/jobs/status/{job_id} 얼굴 탐지 Job 진행 상태 조회
얼굴 탐지 Job 결과 조회 GET /api/face/detect/jobs/result/{job_id} 완료된 얼굴 탐지 결과(JSON) 조회
감정 분석 Job 생성 POST /api/face/emotion-inference/jobs 감정 분석 Job 생성
감정 분석 Job 상태 조회 GET /api/face/emotion-inference/jobs/status/{job_id}
감정 분석 Job 진행 상태 조회
감정 분석 Job 결과 조회 GET /api/face/emotion-inference/jobs/result/{job_id}
완료된 감정 분석 결과(JSON) 조회
/api/face/detect/jobs 설명

/api/face/detect/jobs

얼굴 탐지 작업을 비동기 Job으로 생성
영상에서 YOLO 기반 반려견 얼굴 Object Detection 수행
최적의 대표 프레임 1장을 선택
대표 프레임 및 얼굴 Crop 이미지를 스토리지에 저장
즉시 결과를 반환하지 않고 job_id만 반환

/api/face/detect/jobs/status/{job_id}

지정된 job_id에 대한 얼굴 탐지 Job 진행 상태만 조회
Job 상태는 queued, processing, finished, failed 중 하나
분석 결과(JSON)는 포함하지 않음

/api/face/detect/jobs/result/{job_id}

status = finished 인 경우에만 유효
저장된 얼굴 탐지 결과 JSON을 조회하여 반환
/api/face/emotion-inference/jobs 설명

/api/face/emotion-inference/jobs

감정 분석 추론을 비동기 Job으로 생성
입력받은 대표 프레임을 기준으로 얼굴 영역 crop
반려견 감정 추론 수행
감정 확률 및 사용자 친화적 설명 생성
즉시 결과를 반환하지 않고 job_id 반환

/api/face/emotion-inference/jobs/status/{job_id}

지정된 job_id에 대한 감정 분석 Job 상태만 조회
Job 상태는 queued, processing, finished, failed

/api/face/emotion-inference/jobs/result/{job_id}

status = finished 인 경우에만 유효

저장된 감정 분석 결과 JSON을 조회하여 반환

1-3. 입력/출력 형식 명세
📌 얼굴 탐지 - 동기

/api/face/detect

Request
Field Type Description
analysis_id string 분석 ID
video_url string 업로드된 영상 접근 URL
{
"analysis_id": "string"
"video_url": "string"
}
Response
동기 방식에서는 업로드 시점과 분석 시점이 거의 동일하므로 captured_at은 필수로 받지 않으며,

서버가 생성한 analyze_at을 기준으로 분석 결과를 기록한다.

→ Top-level

Field Type Description
analysis_id string 분석 결과 식별자(서버 생성)
analyze_at string 분석 시각
→ processing

Field Type Description
analysis_time_ms int 분석 소요 시간
frames_sampled int 분석 프레임 수
fps_used int 내부 샘플링 FPS
→ result

Field Type Description
face.frame_id string 선택된 대표 프레임 ID
face.frame_score float 대표 프레임 점수
face.frame_url string 대표 프레임 url
face.face_crop_url string 얼굴 Crop 이미지 url
{
"analysis_id": "string",
"analyze_at": "string",
"processing": {
"analysis_time_ms": int,
"frames_sampled": int,
"fps_used": int
},
"result": {

```
            "face" : {
              "frame_id": "string",
              "frame_score": float,
              "frame_url": "string",
              "face_crop_url": "string"
              }
  }
}
```

📌 얼굴 탐지 – 비동기
비동기 - Job 생성
POST /api/face/detect/jobs

비동기 방식에서는 얼굴 탐지 및 대표 프레임 선택을 백그라운드 Job으로 처리하며, 초기 요청에서는 결과를 반환하지 않고 job_id만 반환한다.

Request

```
{
  "analysis_id": "string",
  "video_url": "string"
}
```

Response - Job 접수

Field	Type	Description
analysis_id	string	분석 ID
job_id	string	얼굴탐지 Job ID
status	string	queued
submitted_at	string (ISO8601)	작업 접수 시각

```
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "queued",
  "submitted_at": "string"
}
```

비동기 - Job 상태 조회
GET /api/face/detect/jobs/status/{job_id}

Field	Type	Description
analysis_id	string	분석 ID
job_id	string	Job ID
status	enum	queued | processing | finished | failed
progress	int	처리 진행률 (0~100)

비동기 - Job 결과 조회
GET /api/face/detect/jobs/result/{job_id}

status = finished 인 경우에만 유효

저장된 얼굴 탐지 결과 JSON을 조회하여 반환

Request - 성공


```
{
  "analysis_id": "string",
  "job_id": "string",
  "analyze_at": "string",
  "processing": {
    "analysis_time_ms": 320,
    "frames_sampled": 45,
    "fps_used": 5
  },
  "result": {
    "face": {
      "frame_id": "string",
      "frame_score": 0.91,
      "frame_url": "string",
      "face_crop_url": "string"
    }
  }
}
```


Response - 실패


```
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "failed",
  "error": {
    "code": "FACE_NOT_DETECTED",
    "message": "영상에서 반려견 얼굴을 검출하지 못했습니다."
  }
}
```


📌 감정 분석 – 동기

/api/face/emotion-inference

Request

Field	Type	Description
analysis_id	string	분석 ID
frame_id	string	대표 프레임 ID
frame_url	string	대표 프레임 이미지 URL


```
{
  "analysis_id": "string",
  "frame_id": "string",
  "frame_url": "string"
```

}
Response
→ Top-level

Field Type Description
analysis_id string 분석 결과 식별자
analyze_at string (ISO8601) 감정 분석 시각
→ processing

Field Type Description
analysis_time_ms int 감정 추론 소요 시간
face_detected boolean 얼굴 검출 성공 여부
→ result

Field Type Description
emotion.predicted_emotion enum happy, sad, relaxed, angry
emotion.confidence float 예측 감정 신뢰도
emotion.summary string 사용자 친화 요약 문장
emotion.emotion_probabilities object 감정별 예측 확률

```
{
  "analysis_id": "string",
  "analyze_at": "string",
  "processing": {
    "analysis_time_ms": int,
    "face_detected": "boolean"
  },
  "result": {
            "emotion" : {
              "predicted_emotion": "enum",
              "confidence": float,
              "summary": "string",
              "emotion_probabilities": {
                "happy": float,
                "relaxed": float,
                "sad": float,
                "angry": float
              }
            }
  }
}
```

📌 감정 분석 – 비동기
비동기 - Job 생성
POST /api/face/emotion-inference/jobs

Request
{

```
  "analysis_id": "string",
  "frame_id": "string",
  "frame_url": "string"
}
```
→ Response - Job 접수

```
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "queued",
  "submitted_at": "string"
}
```

비동기 - Job 상태 조회
GET /api/face/emotion-inference/jobs/status/{job_id}

```
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "processing",
  "progress": 80
}
```

비동기 - Job 결과 조회
GET /api/face/emotion-inference/jobs/result/{job_id}

Response - 성공
```
{
  "analysis_id": "string",
  "job_id": "string",
  "analyze_at": "string",
  "processing": {
    "analysis_time_ms": 180,
    "face_detected": true
  },
  "result": {
    "emotion": {
      "predicted_emotion": "happy",
      "confidence": 0.91,
      "summary": "반려견이 매우 긍정적이고 즐거운 상태입니다.",
      "emotion_probabilities": {
        "happy": 0.72,
        "relaxed": 0.18,
        "sad": 0.06,
        "angry": 0.04
      }
    }
  }
```

}
Response - 실패
{
  "analysis_id": "string",
  "job_id": "string",
  "status": "failed",
  "error": {
    "code": "EMOTION_INFERENCE_FAILED",
    "message": "감정 분석 모델 추론 중 오류가 발생했습니다."
  }
}

1-4. 응답 오류 명세
공통 오류 응답 포맷
Field Type Description
analysis_id string 분석 요청 ID
status string 항상 "failed"
error.code string 오류 유형 식별 코드
error.message string 오류 상세 설명
{
  "analysis_id": "string",
  "status": "failed",
  "error": {
    "code": "ERROR_CODE",
    "message": "string"
  }
}
오류 코드 예시
INVALID_REQUEST
INVALID_MEDIA_URL
UNSUPPORTED_MEDIA_FORMAT
FACE_NOT_DETECTED
LOW_QUALITY_FACE
EMOTION_INFERENCE_FAILED
INVALID_FACE_INPUT
JOB_NOT_FOUND
JOB_ALREADY_COMPLETED
MODEL_SERVICE_UNAVAILABLE

1-5. 역할 및 연동 관계
역할
AI 모델 서빙 전용 컴포넌트로서, 백엔드/프론트엔드에 영상처리와 모델 추론 로직이 섞이지 않도록 분리된 형태로 제공된다.

역할 분리
① FastAPI (AI Domain)

얼굴 탐지 / 감정 분석 Job 실행
모델 추론 및 후처리 수행
결과 JSON 생성
결과를 DB / Object Storage에 저장
② Backend API

Job 생성 요청
Job 상태 조회 (/jobs/status/{job_id})
finished 상태 확인 후 결과 조회 API 호출 (/jobs/result/{job_id})
Client 응답 구성

연동관계
호출 주체 : 메인 Backend API 서버
호출 시점 : 사용자가 “표정분석”을 실행해 영상을 업로드하는 시점

동작흐름(동기)
Client에서 반려견 표정 영상 촬영 후 업로드
Backend API 서버가 영상 저장(또는 임시 저장) 후 video_url 확보
Backend가 반려견 표정 감정 분석 API 호출
모델 API가 동기적으로 얼굴 탐지 및 감정 추론 수행
분석 결과(JSON)를 Backend로 반환
Backend가 결과를 분석 레코드로 저장 후 Client에 응답
Client는 감정 결과 요약 화면을 표시

동작흐름(비동기)
Client에서 반려견 표정 영상 업로드
Backend가 영상 저장 후 video_url 확보
Backend → /api/face/detect/jobs 호출
FastAPI에서 얼굴 탐지 Job 등록 (queued)
Worker가 얼굴 탐지 수행 (processing)
결과 JSON 저장 후 상태 finished
Backend가 /detect/jobs/status/{job_id} 폴링
상태 finished 확인
Backend → /detect/jobs/result/{job_id} 호출
대표 프레임 확보
Backend → /emotion-inference/jobs 호출
동일한 방식으로 감정 분석 Job 처리
최종 감정 분석 결과를 Client에 전달

데이터 귀속 및 저장
분석 결과는 analysis_id와 analyze_at를 기준으로 저장
대표 프레임 이미지 및 얼굴 Crop 이미지는 오브젝트 스토리지에 저장하고, DB에는 해당 리소스의 URL만 저장

단계 2
📌 모델 추론 성능 최적화
2-1. AI 파이프라인 개요

사용자가 촬영한 5초 분량의 반려견 산책 영상을 입력받아, 두 단계의 딥러닝 모델을 거쳐 최종 반려견 산책 만족도를 산출하는 비동기 추론 파이프라인을 구축한다.

1단계 : 얼굴 탐지(Detection)
목적: 영상 내 유효한 반려견 얼굴 영역(Bounding Box) 확보
프로세스: → ffmpeg를 이용한 I-Frame(키프레임) 추출 → 모델 추론 → 최적 영역 Crop
I-Frame (Intra Frame) : 주변 프레임에 의존하지 않고 단독으로 완전한 이미지를 복원할 수 있는 프레임 : 영상의 대표 스냅샷 역할 → Key Frame으로 활용

2단계 : 표정 분류(Classification)
목적: Crop된 얼굴 이미지 기반 4종 표정(happy / sad / angry / relaxed) 분류
프로세스: → 다중 프레임 표정 추론 → 탐지 신뢰도를 반영한 가중치 앙상블 → 최종 표정 결과 도출

2-2. 데이터 구조 및 실험 전제 조건
입력 데이터 : 사용자가 촬영한 5초 분량 영상(MOV, MP4 등)
전처리 : ffmpeg로 추출한 키프레임만을 분석 대상으로 하여 연산 효율성 확보

2-3. 모델 후보군 및 선정 근거
얼굴 탐지 단계 실시간성과 안정성을 중시하여 YOLO 계열 중심으로 비교
비교 후보군 (Backbone) 선정 이유
YOLOv5nu 경량 구조로 빠른 추론 속도 확보
YOLOv5s 정확도–속도 균형 비교용
YOLOv8n 최신 구조 기반 소형 객체 탐지 성능
YOLOv8s 탐지 정확도 상한 비교용
SSD MobileNet V2 YOLO 외 대안, 경량·안정성 비교용

표정 분류 단계 정확도 대비 모델 경량성과 도메인 적합성을 기준으로 구성
비교 후보군 (Backbone) 선정 이유
ResNet50 안정적인 CNN 기반 베이스라인
MobileNetV3 초경량 모델로 지연 최소화
EfficientNet-B0 정확도 대비 모델 효율 우수

2-4. 성능 지표 정의 및 목표 수치
2-4-1. 모델 단위 지표
얼굴 탐지(Detection)
지표명 정의 목표치
mAP@0.5 실제 얼굴 위치(GT)와 예측 BBox 간 IoU ≥ 0.5 기준 평균 정밀도 ≥ 0.85
Inference Latency (ms) 단일 프레임 기준 얼굴 탐지 모델 추론 시간 ≤ 30 ms
Valid Crop Rate(image-level) 전체 이미지 중,confidence ≥ 0.6 기준으로 표정 분류에 사용할 수 있는 얼굴 crop을 1개 이상 생성할 수 있었던 이미지의 비율 ≥ 95%

표정 분류(Classification)
지표명 정의 목표치
F1-Score (macro) 4개 감정 클래스에 대해 동일 가중치로 계산한 F1 평균 ≥ 0.80
Accuracy 전체 예측 중 정답 비율 참고 지표
Confusion Matrix 감정 간 오분류 패턴 시각적 분석 정성 분석
Inference Latency (ms) 단일 얼굴 Crop 기준 추론 시간 ≤ 20 ms

Size (MB)　　모델 가중치 파일 크기　　≤ 25 MB

📌 용어 정리

Precision: 얼굴이라고 예측한 것 중 실제 얼굴 비율
Recall: 실제 얼굴 중 모델이 탐지한 비율
AP(Average Precision): Precision–Recall 곡선을 하나의 점수로 요약
mAP(mean Average Precision): 클래스별 AP의 평균
mAP@0.5: IoU ≥ 0.5일 때만 정답으로 인정하여 계산한 mAP
GT (Ground Truth): 사람이 직접 라벨링한 실제 얼굴 위치 박스
IoU: 예측 박스와 GT 박스의 겹침 비율
F1-Score: Precision과 Recall의 균형 점수
F1-Score (macro): 클래스별 F1을 동일 가중치로 평균

시스템 단위 지표(E2E)
지표　기준　설명
E2E Latency (p50 / p95)　　≤ 3.0s / ≤ 5.0s 영상 업로드 완료 → 최종 결과 반환
분석 성공률　≥ 95%　얼굴 탐지 및 감정 결과 생성 성공 비율
Job 실패율　　< 3%　시스템 오류(I/O, 메모리, 타임아웃 등)
Valid Crop Rate (video-level) ≥ 70% 키프레임 중 conf ≥ 0.6으로 표정 분류에 실제 사용된 프레임 비율

📌 용어 정리

p50 : 50%의 요청이 끝나는 시간(일반적인 사용자 경험)
p95 : 95%의 요청이 끝나는 시간(최악의 사용자 경험)
📌 Valid Crop Rate 정의

image-level
단위: 이미지 1장
정의: conf ≥ 0.6 얼굴 bbox ≥ 1 → valid
video-level
단위: 영상 1개
분모: 해당 영상에서 추출된 키프레임 수(≤ Nmax)
분자: conf ≥ 0.6으로 얼굴 crop 생성 가능한 프레임 수

2-5. 주요 로직 및 예외 처리
키프레임 추출 시 N-max 정책

영상 전체 프레임을 처리하는 대신, 연산 효율을 위해 ffmpeg를 사용하여
I-Frame(키프레임)만 추출
제한 정책 : 최대 8개(Nmax = 8)로 제한하여 분석 시간을 일정하게 유지
실패 처리 로직 (Fail-safe)

얼굴 미검출
모든 프레임에서 얼굴 탐지 실패
사용자에게 재촬영 가이드 제공("반려견의 얼굴이 정면으로 보이게 다시 촬영해주세요")
저화질/흔들림

전체 프레임의 평균 신뢰도가 < 0.4
분석 결과는 제공하되, 신뢰도 낮음 메시지 함께 안내

2-6. 실험 환경
얼굴 탐지 모델
모델 학습 방식
YOLOv5n Dog face 데이터 fine-tuning
YOLOv8n Dog face 데이터 fine-tuning
SSD MobileNet V2 (SSDLite) COCO pre-trained
Dataset : Kaggle Dog Face Detection(YOLO format) (강아지 얼굴 bounding box GT 포함)
Input :
YOLO: 640 × 640
SSD: 320 × 320
Hardware :
GPU: NVIDIA H100
Framework :
Python: 3.10
PyTorch: 2.9.1 (CUDA 12.8)
Ultralytics YOLO: 8.4.3
Torchvision (SSD MobileNet)
Training Setup(YOLO)
Epochs: 50
Batch Size: 16
Optimizer: YOLO default (SGD/AdamW)

표정 분류 모델 :
Dataset: Kaggle Dog Emotion (4-class: happy/sad/angry/relaxed)
Split: train/val/test = 70/15/15
Input Resolution: 224×224
Optimizer: AdamW
LR: 3e-4
Epochs: 15
Batch size: 32
Loss: CrossEntropyLoss (불균형 시 class weight 적용)
Metric: F1-macro, Accuracy(ref), Confusion Matrix

E2E
Dataset : 실제 사용자 영상 데이터가 확보되지 않은 상황에서, 5초 길이의 합성 영상 생성
생성 방식:
Dog Emotion 이미지 데이터 활용
영상마다 이미지 전환 빈도를 다르게 설정하여, 키프레임(I-Frame) 개수가 영상별로 상이하도록 구성
Hardware: GPU 서버 (CUDA 지원)
Framework: PyTorch, Ultralytics YOLO

2-7. 성능 비교 결과
2-7-1. 얼굴 탐지 모델 성능 비교
모델명 mAP@0.5 Inference Latency (ms) Valid Crop Rate(image-level) 비고

YOLOv5n(fine-tuned) 0.995 0.2ms 83.74% 채택
YOLOv8n(fine-tuned) 0.994 0.2ms 81.45%
SSD MobileNet V2(COCO pretrained) COCO benchmark 기준 약 0.85 18.85 ms 95.22%
YOLOv5n이 YOLO8n 대비 mAP@0.5와 Inference Latecy이 유사하면서, 더 높은 Valid Crop Rate를 보임
YOLO n 모델에서 실사용에 문제 없는 성능을 확인하여 s 계열 모델은 성능 대비 연산 비용 증가가 계산되어 추가 실험을 생략
SSD MobileNet V2는 얼굴을 찾기 위해 여러 크기의 특징 맵을 각각 따로 처리하는 구조이기 때문에, 한 번에 처리하는 YOLO보다 모델을 한 번 통과하는 데 시간이 더 오래 걸리는 것으로 예상됨

2-7-2. 표정 분류 모델 성능 비교
모델명 Accuracy (ref) F1-Score (macro) Inference Latency (ms) Size (MB) 비고
ResNet50 0.7867 0.7866 3.29 91M
MobileNetV3-Large 0.8467 0.8468 3.31 17M
EfficientNet-B0 0.8700 0.8710 4.16 16M 채택
ResNet50은 성능과 모델 크기가 목표 수치에 충족하지 못함
MobileNetV3-Large는 낮은 Inference Latency과 모델 크기 측면에서 우수하나, F1-macro에서 EfficientNet-B0 대비 소폭 낮음
EfficientNet-B0는 세 모델 중 가장 높은 F1-macro(0.8710)를 기록, 모델 크와와 Inference Latency도 목표수치를 충분히 만족

2-7-3. 시스템 단위 성능 비교(Pipeline-level)
비교 Case 후보군

Case	얼굴탐지	표정분류	Ensemble 방식	Frame Policy (Nmax)	비고
A1	YOLOv5n	EfficientNet-B0	Weighted (conf_det)	8	Baseline
A2	YOLOv8n	EfficientNet-B0	Weighted (conf_det)	8	Detector 변경
A3	YOLOv5n	MobileNetV3-Large	Weighted (conf_det)	8	Classifier 변경
B1	YOLOv5n	EfficientNet-B0	Weighted (conf_det)	4	Frame 정책 변경

시스템 단위 성능 비교 결과 (Pipeline-level)

Case	E2E Latency p50	E2E Latency p95	분석 성공률	Job 실패율	Valid Crop Rate (video-level)(mean)
A1	0.98s	1.31s	100%	0%	0.536
A2	1.12s	1.40s	100%	0%	0.536
A3	0.96s	1.29s	100%	0%	0.536
B1	0.67s	1.04s	100%	0%	0.517

2-8. 예상 주요 병목 요소 및 최적화 전략
주요 병목
Key Frame 순차 추론
얼굴 미검출로 인한 재시도
프레임 수 증가에 따른 CPU I/O 부하
최적화 전략
Batch Inference (Key Frame 병렬 처리)
Nmax 제한으로 연산 상한 고정

단계 3
📌 서비스 아키텍쳐 모듈화
3-1. 모듈화 적용 후 전체 아키텍처 개요
목표 구조

표정분석 AI는 비동기 Job 기반 분석을 기본 처리 방식으로 한다.
영상 분석 요청은 Job 단위로 큐잉되며, Worker가 백그라운드에서 처리한다.
얼굴 탐지 결과(대표 프레임) 및 감정 분석 결과는 Job 완료 이벤트 단위로 반환된다.
AI 도메인은 영상 처리·추론 책임만 수행하고, 저장/응답 통합은 외부에서 처리한다.
구성요소 목록

AI Domain
AI Orchestrator
얼굴 탐지 / 감정 분석 Job 수신 및 큐잉
Worker 실행 제어 및 상태 이벤트 관리
Message Queue
표정분석 Job(얼굴 탐지, 감정 분석) 버퍼
Face Analysis Worker
영상 기반 얼굴 탐지 및 대표 프레임 선택
대표 프레임 기반 감정 분석 파이프라인 실행
Emotion Inference Model
입력: 얼굴 crop 이미지
출력: 감정 분류 결과, 확률, 요약 설명

3-2. 모듈별 책임(domain)과 분리 이유
AI Orchestrator

표정분석 Job 생성 및 관리
Job을 Queue에 enqueue
Worker 실행 상태(queued / processing / succeeded / failed) 추적
Worker 결과를 수신하여 상위 시스템으로 전달
분리 이유
영상 분석 로직과 Job 운영(큐잉, 재시도, 타임아웃)을 분리
트래픽 증가 시에도 안정적인 처리 구조 유지
Message Queue

얼굴 탐지/감정 분석 Job을 Worker에게 전달하는 비동기 버퍼
사용자 요청 폭주 시 처리량 완충
분리 이유
영상 처리 및 모델 추론 지연을 동기 요청 경로에서 제거
Job 단위 병렬 처리 가능
Face Analysis Worker

영상 URL 기반 프레임 샘플링
YOLO 기반 반려견 얼굴 탐지
대표 프레임 1장 선택 및 얼굴 crop 생성
감정 분석 모델 추론 수행

사용자 친화 요약 문장 생성
분리 이유
GPU/CPU 연산 비용이 큰 작업을 독립적으로 운영
Worker 단위 확장으로 처리량 조절 가능

3-3. 모듈 간 인터페이스 설계(계약)
Client ↔ Backend (Public REST)
얼굴 탐지 Job 생성

POST /api/face/detect/jobs

Request: JSON

표정 영상 업로드 후 호출
영상은 사전에 저장되어 video_url로 전달
Response:

{ "job_id": "string", "status": "queued" }

얼굴 탐지 Job 상태 조회

GET /api/face/detect/jobs/{job_id}

Response:

{ job_id, status, progress }

감정 분석 Job 생성

POST /api/face/emotion-inference/jobs

Request: 대표 프레임 ID 및 URL

Response:

{ "job_id": "string", "status": "queued" }

감정 분석 Job 결과 조회

GET /api/face/emotion-inference/jobs/{job_id}/result

Backend ↔ AI Orchestrator (Internal REST)
대용량 파일 전송 금지

영상/이미지는 URL 참조 방식

얼굴 탐지/감정 분석은 Job 단위로 위임

Job 생성 요청

POST /internal/ai/face/jobs

Request(예시) :

```
{
  "job_id": "job_face_001",
  "job_type": "FACE_ANALYSIS",
  "video_url": "https://cdn.service.com/videos/face_walk.mp4",
  "callback_url": "https://backend/internal/face/jobs/job_face_001/events",
  "meta": {
    "analysis_id": "analysis_001",
    "requested_at": "2026-01-09T13:00:00+09:00"
  }
}
```

Response(예시) :

```
{
  "accepted": true,
  "job_id": "job_face_001"
}
```

Orchestrator → Backend (Callback: 상태 / 결과 전달)

단일 콜백 엔드포인트 사용
POST /internal/face/jobs/{job_id}/events
event_type: STATUS | PROGRESS | COMPLETE | FAILED
예시 1) queued

```
{
  "event_type": "STATUS",
  "status": "queued",
  "progress": {
    "stage": "QUEUED",
    "percent": 0,
    "message": "표정 분석 대기열에 등록되었습니다."
  }
}
```

예시 2) progress

```
{
  "event_type": "PROGRESS",
  "status": "processing",
  "progress": {
    "stage": "FACE_DETECTION",
    "percent": 40,
    "message": "반려견 얼굴을 탐지하고 있습니다."
  }
```

```
}
```
예시 3) complete
```
{
  "event_type": "COMPLETE",
  "status": "succeeded",
  "result": {
    "emotion": {
      "predicted_emotion": "happy",
      "confidence": 0.82,
      "summary": "반려견이 편안하고 즐거운 상태로 보입니다."
    }
  }
}
```
예시 4) failed
```
{
  "event_type": "FAILED",
  "status": "failed",
  "error": {
    "code": "FACE_NOT_DETECTED",
    "message": "분석 가능한 얼굴을 찾지 못했습니다."
  }
}
```

Backend 처리 규칙
STATUS / PROGRESS: Job 상태 갱신
COMPLETE: 분석 결과 저장 + Job 완료 처리
FAILED: 실패 상태 기록 및 오류 로그 저장
보안 / 신뢰성
콜백 인증: 서비스 토큰 또는 HMAC
콜백 실패 시 Orchestrator 재시도
Backend는 job_id + event_type 기준 멱등 처리

Orchestrator ↔ Queue (Job 메시지)
```
{
  "job_id": "job_face_001",
  "job_type": "FACE_ANALYSIS",
  "video_url": "https://cdn.service.com/videos/face_walk.mp4"
}
```

Worker → Orchestrator (결과 보고)
```
{
  "job_id": "job_face_001",
  "status": "completed",
  "result": {
    "emotion": {
      "predicted_emotion": "happy",
      "confidence": 0.82,
      "summary": "편안하고 즐거운 표정입니다."
```

```
  }
 }
}
```

3-4. 데이터 흐름 (비동기 Job 시나리오)
산책 후 반려견 표정 영상 저장
표정분석 Job 생성 요청
Orchestrator가 Job을 Queue에 등록
Worker가 Job 수신
얼굴 탐지 → 대표 프레임 선택
감정 분석 모델 추론
Job 완료 이벤트로 결과 전달

3-5. 독립 개발/배포/스케일링 고려사항
독립 배포 단위
AI Orchestrator
Face Analysis Worker
스케일링 전략
Worker 수를 Job 큐 길이 기준으로 확장
동시 표정분석 요청 증가 시 Worker 병렬성으로 대응
멱등성 / 중복 처리
job_id 기준 결과 처리
동일 영상 재요청 시 재분석 여부 정책적으로 제어 가능

3-6. 모듈화로 기대되는 효과와 장점
영상 처리 및 추론 지연 격리
트래픽 스파이크 흡수
GPU 리소스 효율적 사용
얼굴 탐지/감정 모델 독립적 개선 가능

3-7. 서비스 시나리오 부합 근거
사례 1) 표정분석 요청 급증

Job 큐 기반 처리로 API 응답 안정성 유지
Worker 확장으로 처리량 대응
사례 2) 얼굴 탐지 모델 교체

Worker 내부 로직만 수정
서비스 인터페이스 영향 없음
사례 3) 비동기 UX 확장

진행률 기반 UX 제공
장시간 처리에도 사용자 이탈 최소화

표정분석

단계 4
📌 멀티스텝 AI/파이프라인 구현 검토

산책 후 촬영된 반려견 표정 영상을 입력으로 받아,
얼굴 탐지 기반 대표 프레임 선택 → 얼굴 Crop → 감정 분류 → 사용자 친화 요약 생성 과정을 멀티스텝 AI 파이프라인으로 정의하고,
비동기 Job 기반 서비스 아키텍처(3단계)와 정합되도록 Worker 내부 실행 구조를 구체화한다.

4-1. 멀티스텝 파이프라인 구성 단계(stage)
표정분석 분석 Job은 영상 1개 = Job 1개 구조이며,
Job 내부에서 아래 Stage가 순차적으로 실행된다.

Stage 구성
VALIDATING
FETCHING_VIDEO
FACE_DETECTION
FRAME_SELECTION
EMOTION_INFERENCE
NARRATION_GENERATOR
RESULT_PARSING
FINALIZING

4-2. 멀티스텝으로 나눈 근거
얼굴 탐지와 감정 추론 분리 단일 단계로 묶으면 실패 원인을 추적 불가
병목 구간 분리 및 진행률 UX 제공 FACE_DETECTION / FRAME_SELECTION: CPU + GPU 혼합 병목 EMOTION_INFERENCE: GPU 고정 비용 사용자에게 의미 있는 진행 상태 제공 필요
조기 실패를 통한 비용, 리소스 절약 얼굴이 탐지되지 않은 영상이나 품질이 너무 낮은 프레임을 감정 추론 모델까지 보내는 것은 불필요
진행률 (Percent) 보고
진행률은 Job 기준 **0~100%**로 정의

EMOTION_INFERENCE는 GPU 추론 구간이므로 실제 체감 지연을 반영해 가장 큰 비중을 둠

Stage Percent
VALIDATING 0 ~ 10
FETCHING_VIDEO 10 ~ 25
FACE_DETECTION 25 ~ 45
FRAME_SELECTION 45 ~ 60
EMOTION_INFERENCE 60 ~ 80
NARRATION_GENERATOR 80 ~ 88
RESULT_PARSING 88 ~ 95
FINALIZING 95 ~ 100

STAGE_PERCENT = { "VALIDATING": (0, 10), "FETCHING_VIDEO": (10, 25), "FACE_DETECTION": (25, 45), "FRAME_SELECTION": (45, 60),

"EMOTION_INFERENCE": (60, 80), "NARRATION_GENERATOR": (80, 88),
"RESULT_PARSING": (88, 95), "FINALIZING": (95, 100), }

이벤트 (progress) 송신 위치(콜백 사용 위치)
이벤트 송신 주체
Face Analysis Worker → AI Orchestrator
AI Orchestrator → Backend (/events 콜백)
이벤트 송신 규칙
stage 시작 시: PROGRESS

stage 종료 후 percent 갱신

최종:

성공 → COMPLETE
실패 → FAILED
예시)

```
{
  "event_type": "PROGRESS",
  "status": "processing",
  "progress": {
    "stage": "EMOTION_INFERENCE",
    "percent": 72,
    "message": "반려견의 감정을 분석하고 있습니다."
  }
}
```

4-4. Stage 설계
공통 정책

max_duration_sec = 5
target_fps = 5
max_resolution = 720p
Worker는 대용량 전달 금지 : 영상/이미지는 URL 참조
Backend 저장 규칙 : Redis=status/progress, MySQL=result
Stage별 상세

Stage 역할 입력 출력 병목/리스크 실패 조건 → 에러
VALIDATING 요청/정책 검증 video_url validated payload 거의 없음 VIDEO_TOO_LONG, INVALID_MEDIA_URL
FETCHING_VIDEO 영상 접근성 확인 video_url video_url 네트워크 INVALID_MEDIA_URL
FACE_DETECTION YOLO 얼굴 탐지 frames face bboxes GPU 병목 FACE_NOT_DETECTED
FRAME_SELECTION 대표 프레임 선택 face candidates best frame 품질 점수 계산 LOW_QUALITY_FACE

EMOTION_INFERENCE 감정 분류 추론 face crop raw inference GPU 병목 EMOTION_INFERENCE_FAILED
NARRATION_GENERATOR 감정 확률 기반 사용자 친화 나레이션 생성 predicted_emotion, confidence, emotion_probabilities narration text 정책 누락, 템플릿 오류 NARRATION_GENERATION_FAILED
RESULT_PARSING 결과 파싱/검증 raw output structured result JSON 불안정 INVALID_FACE_INPUT
FINALIZING 결과 조립/이벤트 result final JSON 거의 없음 콜백 실패(재시도)

4-5. 내부 테이터 형태 예시
FACE_DETECTION 결과

{ "face_detections": [{ "frame_id": "frame_003", "timestamp_ms": 600, "faces": [{ "bbox": [312, 180, 520, 420], "confidence": 0.91 }] }, { "frame_id": "frame_004", "timestamp_ms": 800, "faces": [{ "bbox": [305, 175, 515, 415], "confidence": 0.94 }] }] }

4-6. 실패 기준
FACE_DETECTION 실패

전체 분석 프레임 중 얼굴 탐지 성공 프레임 비율 < 60%
또는 모든 프레임에서 유효한 얼굴 bbox 미검출
→ FACE_NOT_DETECTED (422)

FACE_SIZE_INVALID 실패

탐지된 얼굴 bbox 면적이
프레임 대비 최소 비율 기준 미달 (예: < 5%)
또는 얼굴이 너무 멀어 표정 판별 불가능 수준
→ FACE_TOO_SMALL (422)

FACE_OCCLUDED 실패

얼굴 bbox는 존재하나,
얼굴 confidence 평균 < 임계값 (예: < 0.5)
또는 연속 프레임에서 얼굴 일부 이상 가림(귀/눈/주둥이 소실)
각도 문제(측면/후면)로 표정 핵심 특징점 식별 불가
→ LOW_QUALITY_FACE (422)

NARRATION_GENERATOR 실패

emotion_probabilities 누락
확률 합계 비정상
narration template 매핑 실패
→ NARRATION_GENERATION_FAILED (422)

4-7. Worker Pipeline Runner 설계

구성 요소
PipelineRunner : stage 순서 관리, 공통 타이밍/로깅/예외 처리
VideoFetcher : video_url 접근성 검증
FaceDetector (YOLO) : 반려견 얼굴 bbox 탐지
FrameSelector : 얼굴 품질 점수 기반 대표 프레임 선택
EmotionModel : 얼굴 crop 감정 분류
NarrationGenerator : 감정 확률 기반 사용자 친화 나레이션 생성
ResultParser : JSON 파싱 및 스키마 검증
이벤트 송신 규칙
Worker는 PROGRESS / COMPLETE / FAILED 이벤트만 생성
Orchestrator가 Backend 콜백 전담
Backend는 job_id + event_type 기준 멱등 처리
파이프라인 러너 의사 코드

```
class PipelineRunner:
    def run(self, ctx):
        try:
            emit_stage_start("VALIDATING", "입력 정보를 검증하고 있습니다.")
            validate(ctx)
            emit_stage_end("VALIDATING", "입력 검증이 완료되었습니다.")

            emit_stage_start("FETCHING_VIDEO", "영상을 불러오고 있습니다.")
            video_url = check_video_access(ctx.video_url)
            emit_stage_end("FETCHING_VIDEO", "영상 접근 확인이 완료되었습니다.")

            emit_stage_start("FACE_DETECTION", "반려견 얼굴을 탐지하고 있습니다.")
            detections = detect_faces(video_url, policy)
            emit_stage_end("FACE_DETECTION", "반려견 얼굴 탐지가 완료되었습니다.")

            emit_stage_start("FRAME_SELECTION", "대표 프레임을 선택하고 있습니다.")
            frame = select_best_frame(detections)
            emit_stage_end("FRAME_SELECTION", "대표 프레임 선택이 완료되었습니다.")

            emit_stage_start("EMOTION_INFERENCE", "감정을 분석하고 있습니다.")
            emotion_raw = infer_emotion(frame.face_crop)
            emit_stage_end("EMOTION_INFERENCE", "감정 분석이 완료되었습니다.")

            emit_stage_start("NARRATION_GENERATOR", "반려견의 마음을 말로 표현하고
있습니다.")
            narration = generate_narration(emotion_raw)
            emit_stage_end("NARRATION_GENERATOR", "나레이션 생성이 완료되었습니다.")

            emit_stage_start("RESULT_PARSING", "결과를 정리하고 있습니다.")
            result = parse_result(emotion_raw, narration)
            emit_stage_end("RESULT_PARSING", "결과 정리가 완료되었습니다.")

            emit_stage_start("FINALIZING", "결과를 저장하고 있습니다.")
            emit_complete(result)
```

```
        except KnownError as e:
            emit_failed(e.stage, e.percent, e.code, e.message)
```

4-8. stage별 구현 포인트

validating

Backend에서 mp4를 받고, Worker 레벨에서 한 번 더 정책을 검증해 조기 종료한다.

얼굴 탐지/감정 분석 이전 단계에서 실패를 차단

GPU 추론 비용 및 불필요한 영상 디코딩 방지 목적

```
def _run_validating(self):
    p = self.job.policy

    if p.max_duration_sec > 5:
        raise KnownError(
            stage="VALIDATING",
            percent=2,
            code="VIDEO_TOO_LONG",
            message="표정분석 영상 길이가 정책을 초과했습니다.",
            details={"max_duration_sec": 5}
        )

    if not self.job.video_url:
        raise KnownError(
            stage="VALIDATING",
            percent=3,
            code="INVALID_MEDIA_URL",
            message="video_url이 비어 있습니다."
        )

    return p
```

fetching_video

video_url 기반 접근성 검증

전체 다운로드 이전에 URL 접근 가능 여부만 확인

실제 디코딩은 이후 FACE_DETECTION 단계에서 수행

```
def _run_fetching_video(self, video_url: str):
    try:
        self.video_client.check_access(video_url)
        return {"video_url": video_url}
    except Exception as e:
        raise KnownError(
            stage="FETCHING_VIDEO",
            percent=15,
            code="INVALID_MEDIA_URL",
            message="표정분석 영상을 불러올 수 없습니다.",
            details={"video_url": video_url, "reason": repr(e)}
        )
```

face_detection
영상에서 일정 FPS로 프레임을 추출
YOLO 기반 반려견 얼굴 Object Detection
프레임별 얼굴 후보(bbox + confidence)를 수집

```
def _run_face_detection(self, video_url: str, policy):
    frames = self.video_client.sample_frames(
        video_url=video_url,
        fps=policy.target_fps,
        max_duration_sec=policy.max_duration_sec
    )

    detections = []
    for frame in frames:
        faces = self.face_detector.detect(frame.image)
        if faces:
            detections.append({
                "frame_id": frame.frame_id,
                "frame": frame,
                "faces": faces
            })

    if not detections:
        raise KnownError(
            stage="FACE_DETECTION",
            percent=40,
            code="FACE_NOT_DETECTED",
            message="분석 가능한 반려견 얼굴을 찾지 못했습니다."
        )

    return detections
```

실패 의미

얼굴 자체가 보이지 않음
각도/가림/거리 문제

frame_selection
얼굴 탐지 성공 프레임 중 대표 프레임 1장 선택
점수 기준 예시:
얼굴 confidence
bounding box 크기
얼굴 중심 정렬
블러/명암

```
def _run_frame_selection(self, detections):
    best = None
    best_score = -1.0

    for d in detections:
        face = max(d["faces"], key=lambda f: f.conf)
```

```
            score = self.frame_scorer.score(
                face_conf=face.conf,
                bbox=face.bbox,
                image=d["frame"].image
            )

            if score > best_score:
                best_score = score
                best = {
                    "frame": d["frame"],
                    "face": face,
                    "score": score
                }

        if not best:
            raise KnownError(
                stage="FRAME_SELECTION",
                percent=55,
                code="LOW_QUALITY_FACE",
                message="대표 프레임을 선택할 수 없습니다."
            )

        return best

    emotion_inference
    대표 프레임에서 얼굴 영역 crop
    감정 분류 모델 inference
    감정 확률 + 예측 감정 + 요약 문장 생성
    def _run_emotion_inference(self, selected_frame):
        try:
            face_crop = self.video_client.crop(
                image=selected_frame["frame"].image,
                bbox=selected_frame["face"].bbox
            )

            result = self.emotion_model.infer(face_crop)

            return {
                "predicted_emotion": result.label,
                "confidence": result.confidence,
                "emotion_probabilities": result.probabilities
            }

        except Exception as e:
            raise KnownError(
                stage="EMOTION_INFERENCE",
                percent=75,
                code="EMOTION_INFERENCE_FAILED",
```

```
            message="감정 분석 중 오류가 발생했습니다.",
            details={"exception": repr(e)}
        )

narration_generator
1인칭 반려견 시점
감정 확률 기반 강도 조절
confidence < 임계값 → 완곡 표현
def _run_narration_generator(self, emotion_raw: dict):
    try:
        emotion = emotion_raw["predicted_emotion"]
        conf = emotion_raw["confidence"]
        probs = emotion_raw["emotion_probabilities"]

        narration = self.narrator.generate(
            emotion=emotion,
            confidence=conf,
            probabilities=probs
        )

        return narration

    except Exception :
                    return "지금 내 기분을 말로 표현하기가 조금 어려워. 그래도 산책은
괜찮았어!"

result_parsing
감정 분석 결과의 필수 필드 검증
downstream(API 응답/저장)에서 신뢰 가능한 구조로 변환
def _run_result_parsing(self, emotion_raw: dict, narration: str):
    try:
        emotion = emotion_raw["predicted_emotion"]
        confidence = float(emotion_raw["confidence"])
        probs = emotion_raw["emotion_probabilities"]
    except Exception as e:
        raise KnownError(
            stage="RESULT_PARSING",
            percent=90,
            code="INVALID_FACE_INPUT",
            message="감정 분석 결과를 해석할 수 없습니다.",
            details={"raw": emotion_raw, "error": repr(e)}
        )

    return {
        "emotion": {
            "predicted_emotion": emotion,
            "confidence": confidence,
            "emotion_probabilities": probs
```

```
        },
        "narration": narration
    }

finalizing
분석 결과 JSON 조립
processing / model_version 기록 유지
Job 완료 이벤트 송신 준비
from datetime import datetime, timezone

def iso_now():
    return datetime.now(timezone.utc).isoformat()

def now_ms():
    return int(datetime.now(timezone.utc).timestamp() * 1000)

def _finalize(self, result: dict, started_ms: int):
    finished_ms = now_ms()
    analysis_time_ms = max(0, finished_ms - started_ms)

    job_id = self.job.job_id
    analysis_id = self.job.analysis_id or job_id

    final = {
        "analysis_id": analysis_id,
        "job_id": job_id,
        "analyze_at": iso_now(),
        "processing": {
            "analysis_time_ms": analysis_time_ms,
            "face_detected": True
        },
        "result": result,
                "model_version": self._compose_model_version()
    }
    return final
def _compose_model_version(self):
    pipeline_ver = getattr(self.job, "pipeline_version", "face_pipeline_v1")
    model_ver = getattr(self.emotion_model, "version", "emotion_model_v1")
    return f"{pipeline_ver}:{model_ver}"
```

4-9. 멀티스텝 파이프라인 다이어그램
표정분석 파이프라인

단계 5
📌 데이터/컨텍스트 보강 설계

표정분석 기능은 촬영된 영상에서 반려견의 현재 감정 상태를 일관된 기준으로 분류하는 결정형 추론 서비스이다. 분류 결과를 사용자에게 더 이해하기 쉬운 설명으로 전달하기 위해 모델 추론 결과에 경량 컨텍스트 보강(RAG-lite)을 적용한다. 설명 생성 품질을 안정화하기 위한 최소한의 컨텍스트 보강 구조만을 설계한다.

5-1. 컨텍스트 보강이 필요한 근거
감정 해석의 모호성 : 동일한 얼굴 표정이라도 촬영 각도, 조명, 프레임 선택, 반려견 개체 차이 등으로 해석이 달라질 수 있다. 모델 출력 확률만으로는 감정 결과에 대한 설명이 부족하다.
판정 기준의 일관성 유지 : 동일한 감정에 대해 설명 문구가 매번 달라지면 사용자 신뢰도가 저하될 수 있으므로 감정별 해석 기준을 명시적으로 고정할 필요가 있다.

5-2. RAG-lite 적용 전략
감정 분석은 Closed-set 분류 문제로, 외부 문서 검색을 통해 새로운 정보를 찾아야 할 필요가 없음

이미 분석된 영상 결과에 대한 해석 문제이므로 Visual/Video RAG는 비용이나 지연을 증가시킬 수 있다.

감정 유형은 명확한 enum 구조이며, 감정별 해석 기준은 사전에 정의 가능하다.

모델의 판정 해석 기준만 보강하기 위해 Rule / Reference 기반 RAG-lite를 적용한다.

감정 유형(emotion_type)에 대응되는 시각적 특징, 해석 가이드, 설명 톤을 모델 입력 컨텍스트로 동적 삽입하여 설명 결과의 일관성을 강화한다.
적용 범위

내부 규칙/설명 템플릿
LLM 프롬프트 증강

5-3. 전체 데이터 흐름
[Emotion Video]
↓
[Face Detection & Emotion Inference]
(emotion_type, confidence)
↓
[Emotion Reference Store 조회]
(해당 감정의 해석 가이드)
↓
[Explanation Prompt 구성]
(감정 결과 + 해석 기준 포함)
↓
[LLM 호출 (설명 생성)]
↓
[emotion / confidence / explanation]
검색 단계 없음
컨텍스트는 결과 이후(Post-Inference)에만 사용
모델 판단 자체에는 영향 없음

5-4. Emotion Reference Store 설계
데이터 성격
정적(reference) 데이터
감정 유형별 “해석 기준 컨텍스트” 저장, 학습 데이터가 아닌 설명 기준 데이터
운영 중 수정 가능 (프롬프트 재배포 불필요)
외부 검색이 불필요하고, 프롬프트 일관성 확보에 효과적
저장 형태
JSON 파일
emotion_type 기준 Key-Value 조회
예시 스키마

```
{
  "emotion_type": "STRESSED",
  "emotion_name": "긴장",
  "visual_cues": [
    "눈을 크게 뜨고 있음",
    "입을 다문 상태가 지속됨",
    "얼굴 근육이 긴장되어 있음"
  ],
  "interpretation_guideline": [
    "주변 환경에 대한 경계 또는 불안 신호일 수 있음",
    "즉각적인 부정 감정으로 단정하지 않음"
  ],
  "confidence_guideline": {
    "high": "여러 시각적 특징이 명확히 관찰됨",
    "medium": "일부 특징만 관찰됨",
    "low": "표정이 불명확하거나 프레임이 부족함"
  },
  "tone": "관찰 중심, 안심 유도"
}
```

5-5. 모델 입력(프롬프트) 증강
컨텍스트 보강 입력 예시)

```
[감정 분류 결과]
- 감정: STRESSED
- 신뢰도: 0.87

[시각적 특징 가이드]
- 눈을 크게 뜨고 있음
- 입을 다문 상태
- 얼굴 근육 긴장

[해석 가이드]
- 불안 또는 경계 상태일 수 있음
- 단정적인 표현은 피할 것

[설명 요청]
위 기준을 바탕으로,
```

보호자가 이해하기 쉬운 문장으로
현재 반려견의 감정 상태를 설명해줘.

5-6. 모델 통합 방식
프롬프트 구성 원칙
역할 명확화 : “감정 판정 설명자” 역할 고정
기준 명시 : 시각적 특징 + 해석 가이드 bullet 제공
출력 형식 고정 : JSON Schema 또는 명확한 필드 구조
추측 금지 : 보이지 않는 정보는 가정하지 않음

5-7. 프롬프트 템플릿 예시
You are an AI assistant that explains a dog's emotional state
based on a short facial video and predefined interpretation rules.

[Emotion Information]
- Emotion Type: {{EMOTION_TYPE}}
- Emotion Name: {{EMOTION_NAME}}
- Confidence: {{CONFIDENCE}}

[Visual Cues]
{{VISUAL_CUES_LIST}}

[Interpretation Guideline]
{{INTERPRETATION_GUIDELINE}}

[Tone Guideline]
{{TONE}}

[Instruction]
Using the information above:
1. Explain the dog's current emotional state in a calm, user-friendly manner.
2. Avoid definitive or alarming language.
3. Focus on observable facial cues.

[Output Format]
Return ONLY the following JSON format:

```
{
  "emotion": string,
  "confidence": number,
  "explanation": string
}
```

5-8. 도입 효과 및 검증 계획
도입 전

감정 설명이 매번 달라질 수 있음
confidence 대비 설명 품질 불균형

사용자 해석 혼란 가능성 존재
도입 후

감정별 해석 기준 고정
설명 문구의 일관성 향상
confidence 분포와 설명 간 정합성 개선
결과 재현성 및 사용자 신뢰도 향상
검증 방법

동일 영상 반복 분석 시 설명 일관성 비교
감정별 confidence 수치에 비해 설명이 과하거나 부족하지 않은지 확인

단계 6
📌 표준화된 도구 통합 및 외부 API 활용 설계
6-1. 설계 목적
표정분석 기능에서는 얼굴탐지, 감정 분류, 감정 설명 생성 세 부분에서 AI가 활용된다. 각 단계는 요구되는 정확도, 연산 비용, 데이터 통제 수준이 서로 다르므로 자체 모델과 외부 상용 API를 단계별로 선택적으로 활용하는 전략이 필요하다.

얼굴 탐지·감정 분류는 일관성·비용·통제 측면에서 자체 모델 활용
설명 생성은 자연어 품질 향상을 위해 외부 LLM API 활용
외부 도구/API 연동을 표준화하여 안정성과 확장성을 확보
• 외부 시스템 및 상용 서비스와의 상호작용 시나리오

구분 설명
영상 처리 업로드 영상 디코딩 및 프레임 샘플링
얼굴 탐지 모델 반려견 얼굴 위치 탐지 및 품질 필터링
감정 분류 모델 얼굴 이미지 기반 감정 분류
설명 생성 감정 결과를 자연어 설명으로 변환
스토리지 원본 영상·중간 산출물·결과를 S3에 저장
외부 상용 AI API 감정 설명 생성에 LLM API 활용

6-2. MCP 도입 여부 결정
MCP 도입 여부: 도입하지 않음

표정분석 기능에서는 MCP(Model Context Protocol)를 도입하지 않는다.

표정분석은 AI 파이프라인이 고정적이며, 모델이 상황에 따라 도구를 판단할 필요가 없다. 또한 외부 API 호출 여부와 대상은 시스템 설계 단계에서 결정되며 MCP가 제공하는 동적 구조가 필요하지 않다.

따라서 기존 REST 기반 API 호출 + Client 모듈 분리 구조가 더 합리적이라고 판단하였다.

근거
모델 주도 도구 선택 구조가 아님
얼굴 탐지/감정 분류는 항상 내부 모델 사용
외부 API는 설명 생성 단계에서만 호출
도구 호출 흐름이 단순

LLM API 1개 사용
복잡한 Tool Registry 관리 불필요

6-3. 얼굴 탐지 모델 전략
얼굴 탐지는 이후 감정 분류 품질을 결정하는 핵심 단계이므로 일관된 기준과 낮은 지연 시간이 중요하다.

자체 모델 사용 근거
항목 이유
실시간성 외부 API 대비 응답 지연 최소화
일관성 얼굴 크기/가림/각도 기준을 서비스 정책에 맞게 정의 가능
비용 영상 프레임 단위 호출 시 상용 API 비용 부담 큼
제어 실패 조건(미검출, 너무 작음 등)을 명확히 정의 가능
→ YOLO 계열 자체 모델 사용

6-4. 감정 분류 모델 전략
감정 분류는 얼굴 이미지 기반 분류 문제로, 라벨 정의와 confidence 해석의 일관성이 중요하다.

자체 모델 사용 근거
항목 이유
감정 라벨 통제 서비스에 맞는 감정 클래스 정의 가능
Confidence 해석 임계값/불확실성 기준을 정책적으로 설정
비용 안정성 요청 증가 시 API 비용 폭증 방지
데이터 축적 서비스 운영 데이터로 점진적 성능 개선 가능
→ ResNet 기반 분류 모델 또는 경량 CNN 사용

6-5. 상용 API 사용 전략
감정 결과를 사용자에게 전달할 때는 정확성보다 전달력, 설득력, 친절함이 중요하기 때문에 상용 LLM API를 활용하는 것이 합리적이다.

상용 LLM API 사용 근거
항목 상용 LLM 사용 이유
자연어 품질 다양한 표현과 문맥 있는 설명 가능
개발 속도 규칙 기반 설명 로직 구현 불필요
확장성 감정 유형 추가 시 즉시 대응 가능

Gemini vs OpenAI(LLM) 비교 및 사용 전략
항목 Gemini OpenAI
입력 확장성 이미지+텍스트 대응 텍스트 중심
설명 자연스러움 높음 매우 높음
프롬프트 제어 중간 높음
비용 구조 요청/토큰 기반 토큰 기반
적합 시나리오 감정+상황 설명 감정 요약·조언 생성

6-6. API 실패 대응 및 안정성 확보 전략

외부 API 실패 및 Fallback 전략

외부 API 실패는 설명 생성 단계에 한정된다.
외부 LLM API 호출 실패 시 즉시 전체 분석을 실패 처리하지 않는다.
처리 흐름
얼굴 탐지 및 감정 분류 결과 생성
설명 생성을 위해 외부 LLM API 호출
타임아웃 또는 오류 발생 시:
최대 2회 재시도
재시도 간 2초 대기
재시도 실패 시:
감정 결과는 정상 반환
설명은 기본 템플릿 문장으로 대체
실패 기록 방식

사용자 화면:
감정 결과 정상 표시
간단한 기본 설명 제공
내부 로그:
failure_type = SYSTEM_ERROR
reason = LLM_API_TIMEOUT 또는 LLM_API_ERROR

6-7. 확장성 고려
얼굴 탐지, 감정 분류, 설명 생성을 구분된 모듈로 설계한다.

외부 API 호출은 전용 Client 모듈에서만 수행한다.

EmotionAnalysisOrchestrator
 ├─ FaceDetectionService (Local Model)
 ├─ EmotionClassificationService (Local Model)
 └─ ExplanationClient.generate()
     ├─ GeminiClient
     └─ OpenAIClient
확장 전략

감정 분류 모델 교체 시 Orchestrator 수정 최소화
새로운 LLM 추가 시 Client 모듈만 확장

헬스 케어 기능

Jade Junghoo Lee edited this page 2 hours ago · 47 revisions

1. 모델 API 설계
본 헬스케어 모델 API는 반려견 보행 영상을 입력으로 받아, 반려견의 보행 패턴을 정량적으로 분석하고 이를 기반으로 아래 지표를 점수(0–100) 및 설명 형태로 제공한다.

슬개골 위험 신호
좌·우 보행 균형
관절 가동성(무릎 중심)
보행 안정성
보행 리듬(일관성)
해당 결과는 의료 진단이 아닌 조기 위험 신호 제공을 목적으로 하며, 장기 추세 분석 또는 “주의가 필요해 보입니다”와 같은 사용자 피드백에 활용된다.

1-1. API 설계 개요 (동기 + 비동기 동시 지원)
기본 - 비동기 Job
영상 기반 분석은 처리 시간이 길어질 수 있으므로, Job 기반 비동기를 기본으로 한다.

Client는 Backend만 호출한다.

AI 도메인(Orchestrator/Worker)은 DB/Redis에 직접 write 하지 않고, 상태/결과를 Backend로 콜백한다.

Backend는 Redis(MySQL) 저장의 단일 책임자(write owner) 로서 상태/결과를 저장하고 응답한다.

MVP - 동기 지원
MVP 단계 또는 UX 단순화를 위해 “업로드 후 로딩 화면에서 대기 → 결과 즉시 보기”가 필요할 수 있다.

동기 모드는 별도 파이프라인을 두지 않고, 동일 Job 파이프라인을 사용한다.

권장 구현: /api/healthcare/analyze는 내부적으로 Job 생성 후 완료까지 대기(timeout 내)하는 wrapper로 동작

timeout 초과 시 202 Accepted + job_id 반환(클라이언트는 상태/결과 조회로 전환)

1-2. 앤드포인트 목록 및 기능 설명
Public API(외부): Client → Backend(Spring)

Internal API(내부): Backend ↔ AI Orchestrator(FastAPI) / Orchestrator → Backend(콜백)

Public API (Client → Backend)
POST /api/healthcare/jobs
보행 분석 Job 생성(영상 업로드 포함 또는 video_url 참조 방식) 성공 시 job_id 반환(기본 status=queued)

GET /api/healthcare/jobs/{job_id}
Job 상태/진행률 조회 (Backend가 Redis에서 조회 후 반환)

GET /api/healthcare/jobs/{job_id}/result
최종 결과 조회 (Backend가 MySQL에서 조회 후 반환)

POST /api/healthcare/analyze
(옵션: 동기 wrapper) 동일 Job 파이프라인으로 처리하되, 완료까지 대기 후 결과를 즉시 반환 timeout 시 202 + job_id 반환(비동기 플로우로 전환)

GET /api/healthcare/health
헬스 체크(Backend 기준: 서비스 정상/의존성 상태)

사용자로부터 보행 영상 업로드(최대 N초 mp4)
영상에서 프레임 샘플링
반려견 탐지/추적(필요 시)
키포인트 추정(전신/하체 관절 중심)
프레임 간 키포인트 시계열로부터 지표 계산
지표를 점수화하고 간단한 설명을 생성해 JSON 반환
Internal API (Backend ↔ Orchestrator)
POST /internal/ai/healthcare/jobs
Backend가 Orchestrator에 Job 실행을 요청(큐잉/워커 실행은 Orchestrator 책임)

POST /internal/healthcare/jobs/{job_id}/events
(콜백 통합 권장) Orchestrator가 Backend로 상태/진행/완료/실패 이벤트를 전달 Backend는 이를 Redis/MySQL에 반영

내부 API는 외부 공개 금지이며, 서비스 간 인증(mTLS 또는 HMAC/서비스토큰) 적용을 전제로 한다.

/jobs로 분석 요청 등록 → job_id 반환
백엔드는 주기적으로 상태 조회 또는 결과 조회
완료 시 결과 저장/응답
1-3. 입력/출력 형식 명세
Public: Job 생성
Request 방식 1 — 업로드 포함 (multipart/form-data)

POST /api/healthcare/jobs
form fields

file: mp4 video (required)

dog_id: string (optional)

captured_at: string(ISO8601) (optional)

Backend는 file을 S3에 저장하고 video_url을 확보한 뒤, 내부적으로 Orchestrator에 Job을 생성한다.

Request 방식 2 — video_url 참조 (JSON)

```
{
  "video_url": "string",
  "dog_id": "string|null",
  "captured_at": "string(ISO8601)|null"
}
```

Response (공통)

```
{
  "job_id": "string",
  "status": "queued"
}
```

job_id: Backend가 생성(권장). 추적 단순화를 위해 analysis_id와 동일하게 사용해도 됨(아래 참조).

Public: Job 상태 조회

GET /api/healthcare/jobs/{job_id}

Response

```
{
 "job_id": "string",
 "status": "queued|processing|succeeded|failed",
 "progress": {
"stage":
"VALIDATING|FETCHING_VIDEO|SAMPLING_FRAMES|DOG_TRACKING|KEYPOINT_ESTIMATION|METRIC_SCORING|ARTIFACT_GENERATION|FINALIZING",
   "percent": 0,
   "message": "string"
 },
  "error": {
   "code": "STRING_CODE",
   "message": "string",
   "details": {}
 }
}
```

error는 status=failed일 때만 포함(권장)

Public: Job 결과 조회

GET /api/healthcare/jobs/{job_id}/result

Response (성공)

```
{
  "analysis_id": "string",
  "job_id": "string",
```

```
  "analyze_at": "string(ISO8601)",
  "processing": {
    "analysis_time_ms": 0,
    "video_duration_sec": 0.0,
    "frames_sampled": 0,
    "fps_used": 0
   },
   "result": {
    "overall_risk_level": "low|medium|high",
    "summary": "string"
  },
  "metrics": {
    "patella_risk_signal": {
      "level": "low|medium|high",
      "score": 0,
      "description": "string"
    },
    "gait_balance": { "score": 0, "description": "string" },
    "knee_mobility": { "score": 0, "description": "string" },
    "gait_stability": { "score": 0, "description": "string" },
    "gait_rhythm": { "score": 0, "description": "string" }
  },
  "artifacts": {
    "status": "pending|ready|failed",
    "keypoint_overlay_video_url": "string|null",
    "report_image_url": "string|null"
  },
  "model_version": "string"
}
```

analysis_id / job_id 관계(권장)

권장 1: analysis_id == job_id 로 단일 식별자 사용(단순/운영 편리)
권장 2: 분리하되 결과에 job_id를 포함(추적/디버깅 편리)
본 문서에서는 추적을 위해 job_id 포함을 기본으로 둔다.

Public: 동기 wrapper(옵션)
POST /api/healthcare/analyze
Request

(권장) multipart/form-data 업로드 또는 {video_url} JSON 둘 중 하나 지원(위 Job 생성과 동일)
Response

완료 시: 결과 JSON(= /result와 동일)
timeout 시: 아래 형태로 비동기 전환

```
{
  "job_id": "string",
  "status": "processing",
```

```
  "next": {
    "status_url": "/api/healthcare/jobs/{job_id}",
    "result_url": "/api/healthcare/jobs/{job_id}/result"
  }
}
```

Internal: Backend → Orchestrator Job 생성

POST /internal/ai/healthcare/jobs

Request

```
{
  "job_id": "string",
  "video_url": "string",
  "callback_url": "string",
  "policy": {
    "max_duration_sec": 10,
    "target_fps": 12,
    "max_resolution": 720
  },
  "meta": {
    "user_id": "string",
    "dog_id": "string|null",
    "requested_at": "string(ISO8601)",
    "captured_at": "string(ISO8601)|null"
  }
}
```

policy는 Backend가 보내도 되고, Orchestrator가 기본값을 적용해도 된다. (문서에는 “표준화 파라미터 결정은 Orchestrator 책임”을 권장)

Response

```
{ "accepted": true, "job_id": "string" }
```

Internal: Orchestrator → Backend 이벤트 콜백(통합)

POST /internal/healthcare/jobs/{job_id}/events

Event payload (공통)

```
{
  "event_type": "STATUS|PROGRESS|COMPLETE|FAILED",
  "status": "queued|processing|succeeded|failed",
  "progress": {
    "stage": "string",
    "percent": 0,
    "message": "string"
  },
  "result": {},
  "error": {}
}
```

STATUS/PROGRESS : progress 사용

COMPLETE : result 포함(= 최종 결과 JSON 구조)

FAILED : error 포함

Backend는 이벤트를 수신하면:

Redis에 status/progress 반영

COMPLETE면 MySQL에 결과 저장 + Redis succeeded

FAILED면 Redis failed + (선택) MySQL 실패 로그 저장

1-4. 응답 오류 명세
공통 오류

```
{
  "error": {
    "code": "STRING_CODE",
    "message": "string",
    "details": {}
  }
}
```

대표 오류

```
{
  "error": {
    "code": "INVALID_FILE_TYPE",
    "message": "mp4 형식의 영상을 업로드해 주세요.",
    "details": { "allowed": ["video/mp4"] }
  }
}
```

(400) INVALID_FILE_TYPE

```
{
  "error": {
    "code": "VIDEO_TOO_LONG",
    "message": "영상 길이가 너무 깁니다. 10초 이하로 업로드해 주세요.",
    "details": { "max_duration_sec": 10 }
  }
```

(400) VIDEO_TOO_LONG

```
{
  "error": {
    "code": "DOG_NOT_TRACKED",
    "message": "반려견을 안정적으로 추적하지 못했습니다. 촬영 각도/거리/조명을 바꿔 다시 촬영해 주세요.",
    "details": {}
  }
}
```

(422) DOG_NOT_TRACKED

```
{
  "error": {
    "code": "INSUFFICIENT_FRAMES",
    "message": "분석에 필요한 보행 장면이 부족합니다. 5~10초 정도 다시 촬영해 주세요.",
    "details": { "min_frames": 60 }
  }
}
```
(422) INSUFFICIENT_FRAMES

```
{
  "error": {
    "code": "INFERENCE_TIMEOUT",
    "message": "분석 시간이 초과되었습니다. 잠시 후 다시 시도해 주세요.",
    "details": { "timeout_ms": 15000 }
  }
}
```
(504) INFERENCE_TIMEOUT

1-5. 역할 및 연동 관계
역할
Backend(Spring): 인증/권한, 업로드(S3), 상태/결과 저장(Redis/MySQL), Client 응답 통합

AI Orchestrator(FastAPI): 큐잉/라우팅/워커 운영, 이벤트 콜백 송신

Worker(GPU/CPU): 영상 처리 파이프라인 수행(키포인트/지표/아티팩트)

연동 흐름(비동기 기본)
Client → Backend: POST /api/healthcare/jobs (업로드)
Backend → S3: 원본 영상 저장(video_url)
Backend → Orchestrator: POST /internal/ai/healthcare/jobs
Orchestrator → Queue: enqueue
Orchestrator → Backend: STATUS(queued) 콜백
Worker 처리 + progress 이벤트 → Orchestrator
Orchestrator → Backend: PROGRESS 콜백(선택)
완료 시 Orchestrator → Backend: COMPLETE 콜백(결과 포함)
Backend: MySQL 저장 + Redis succeeded
Client: GET /jobs/{id} / GET /result로 조회
연동 관계 (수정 예정)
동작 흐름 (동기) (수정 예정)
데이터 귀속 및 저장 (수정 예정)
아키텍쳐에서의 위치 (수정 예정)
분리 이유 (설계 근거) (수정 예정)
1-6. API 호출 예시 및 예시 응답
(비동기) Job 생성
POST /api/healthcare/jobs
```
{
  "video_url": "https://storage/.../gait.mp4",
```

```
  "dog_id": "d_456",
  "captured_at": "2026-01-06T10:29:30+09:00"
}
```

(비동기) Job 응답

```
{ "job_id": "job_3f2a9c8d1b", "status": "queued" }
```

상태 조회
```
GET /api/healthcare/jobs/job_3f2a9c8d1b
```

```
{
  "job_id": "job_3f2a9c8d1b",
  "status": "processing",
  "progress": {
    "stage": "KEYPOINT_ESTIMATION",
    "percent": 55,
    "message": "키포인트를 추정하고 있습니다."
  }
}
```

결과 조회
```
GET /api/healthcare/jobs/job_3f2a9c8d1b/result
```
응답 예시 (성공)

```
{
  "analysis_id": "gait_3f2a9c8d1b",
  "analyze_at": "2026-01-06T10:30:00+09:00",
  "processing": {
    "analysis_time_ms": 8200,
    "video_duration_sec": 8.3,
    "frames_sampled": 96,
    "fps_used": 12
  },
  "result": {
    "overall_risk_level": "medium",
    "summary": "좌우 보행 균형이 다소 불안정해 보입니다."
  },
  "metrics": {
    "patella_risk_signal": {
      "level": "medium",
      "score": 62,
      "description": "후지 지지 패턴의 좌우 차이가 관측됩니다. (...)"
    },
    "gait_balance": {
      "score": 68,
      "description": "좌우 보폭/지지시간 차이가 존재합니다. (...)"
    },
    "knee_mobility": {
      "score": 72,
      "description": "무릎 관절 가동 범위가 약간 제한적입니다. (...)"
    },
    "gait_stability": {
      "score": 75,
```

```
      "description": "전반적으로 안정적이나 간헐적 흔들림이 관측됩니다. (...)"
    },
    "gait_rhythm": {
      "score": 80,
      "description": "보행 리듬은 비교적 일정합니다. (...)"
    }
  },
  "artifacts": {
    "keypoint_overlay_video_url": "https://storage/.../gait_3f2a9c8d1b_overlay.mp4",
    "report_image_url": "https://storage/.../gait_3f2a9c8d1b_summary.png"
  },
  "model_version": "gait_analysis_v1"
}
```


호출 예시 (비동기)
Job 생성: POST /api/healthcare/jobs → job_id 수신
상태 조회: GET /api/healthcare/jobs/{job_id}
결과 조회: GET /api/healthcare/jobs/{job_id}/result (동기와 동일 결과 JSON)
2. 모델 추론 성능 최적화
2-1. 기존 모델 추론 성능 지표
측정 대상: POST /api/healthcare/analyze (동기), 1회 요청 기준

E2E 응답시간(p50 / p95): (예) 7.5s / 12.0s
GPU 사용률 / GPU 메모리: (예) 55% / 6.2GB
CPU 사용률: (예) 80% (영상 디코딩/전처리 구간에서 상승)
프레임 처리량: (예) 12fps 샘플링, 8초 영상 → 약 96프레임 처리
타임아웃/실패율: (예) p95에서 간헐적 504(대형 입력/네트워크 지연 시)
실제 수치는 4주차 상세 과제에서 측정 로그로 확정하고, 이번 단계에서는 “측정 항목”과 “목표 방향”을 확정한다.

2-2. 성능 병목 요소 및 원인
영상 I/O + 디코딩
원인: 원본 mp4 디코딩 비용, 고해상도/고fps 영상일수록 증가
증상: CPU 사용률 급등, 분석 시간 편차(p95)가 커짐
포즈(키포인트) 추정 모델 추론
원인: 모델 연산량 자체가 큼(GPU), 프레임 수에 선형으로 비례
증상: GPU 메모리/연산 병목, 프레임 수가 늘면 응답시간 급증
프레임별 후처리(추적/스무딩/각도 계산)
원인: 파이썬 루프 기반 구현 시 CPU 병목 발생 가능
증상: GPU는 놀고 CPU가 오래 걸리는 구간이 생김
(미정)아티팩트 생성(오버레이 영상/요약 이미지)
원인: 렌더링/인코딩 비용이 큼, 동기 요청에서 체감 지연 증가
증상: “분석 자체는 끝났는데 저장/인코딩 때문에 대기” 발생
2-3. 적용할 최적화 기법의 구체적 계획
이번 단계에서는 “바로 적용 가능한 저위험 최적화” 중심으로 계획한다.

A. 입력 제한 + 표준화(가장 효과 대비 난이도 낮음)
max_duration_sec를 서버 정책으로 강제(예: 10초)
해상도 리사이즈(예: 긴 변 720p 이하) 후 추론

샘플링 FPS 고정(예: 12fps) + 최소 프레임 수 미달 시 422로 조기 종료
기대효과: 프레임 수/연산량 상한을 고정하여 p95를 안정화

B. 영상 디코딩/전처리 최적화
ffmpeg 기반 디코딩 파이프라인 정리(불필요한 색공간 변환 제거)
디코딩과 추론을 스트리밍 처리(프레임을 한 번에 다 올리지 않고 순차 전달)
기대효과: CPU 병목 완화, 메모리 사용량 감소

C. 모델 추론 최적화(가능한 범위에서 단계적)
Torch compile / ONNX / TensorRT 중 1개를 선택해 적용(환경에 맞춰)
*FP16 / INT8(가능 시)**로 추론(정확도 영향 확인 후)
프레임 배치 처리(예: 4~8프레임 단위 micro-batch)로 GPU 효율 향상
기대효과: GPU 처리량 증가, 추론 시간 단축

D. 후처리 계산 최적화
각도/거리 계산을 numpy 벡터화, 반복 루프 제거
스무딩/필터링 연산 간소화(필요 최소만)
기대효과: CPU 구간 감소, 전체 E2E 감소

E. 아티팩트 생성 분리(동기 UX 유지하면서 지연 줄이기)
동기 응답에서는 점수/설명 우선 반환
오버레이 영상/요약 이미지는 옵션 기반 또는 후행 생성(비동기 워커)로 전환
1단계에서 이미 “향후 Job 기반 확장”을 언급했으므로 자연스럽게 연결 가능
기대효과: 체감 응답시간 대폭 감소(특히 p95)

2-4. 최적화 후 기대 성능 지표
동기 UX를 유지한다는 전제에서, 현실적인 목표치를 설정한다.

E2E 응답시간(p50 / p95)
Before: (예) 7.5s / 12.0s
After 목표: 4.5s / 8.0s
타임아웃(504) 비율
Before: (예) p95에서 간헐 발생
After 목표: 1% 미만
GPU 메모리 안정화
입력 표준화(10초/720p/12fps)로 메모리 상한 고정
처리량(동시성) - 단일 GPU 기준 동시 요청 시 queueing 최소화(추후 비동기 전환 시 더 개선)
2-5. (메모) 측정/검증 방법
동일한 조건(10초/720p/12fps, 동일 영상 세트)으로 전/후 비교
로그에 구간별 타임스탬프 기록
decode / pose_infer / postprocess / artifact
p50/p95, GPU/CPU, 실패율(422/504)을 함께 기록
3. 서비스 아키텍처 모듈화 (비동기 Job 기반)
아키텍처 다이어그램

기본 처리 방식: 비동기 Job 기반 (jobs / status / result)

호출 경계: Client는 Backend(Spring) 만 호출 (AI 도메인 직접 호출 없음)

저장소 접근 정책

MySQL/Redis write owner = Backend

AI Orchestrator/Worker는 DB/Redis 직접 write 금지

AI 도메인 역할: Orchestrator가 Queue + Worker 운영(라우팅/재시도/타임아웃/스케일링 기준), 결과/상태는 Backend로 콜백

대용량 전송 금지: 모듈 간 파일 전송 대신 참조 전달(video_url, artifact_urls) 원칙

3-1. 모듈화 적용 후 전체 아키텍처 개요
목표 구조(요약)
Backend는 서비스 도메인(인증/업로드/저장/응답) 에 집중한다.
AI 도메인은 Job 실행(큐잉 + 워커 + GPU 추론) 만 수행한다.
상태/결과는 콜백(callback) 기반으로 Backend에 전달되고, Backend가 Redis/MySQL에 저장한다.
Queue가 요청 폭주를 완충하고, Worker는 수평 확장 가능하며, AI 장애가 Backend로 전파되지 않도록 격리한다.
구성요소 목록
Client (App/Web)
Service Domain (예: AWS)
Backend API Server (Spring)
Redis: Job 상태/진행률 캐시 (Backend만 write/read)
MySQL: 분석 결과 영속 저장 (Backend만 write/read)
Object Storage (S3): 원본 영상, (선택) 오버레이/리포트 이미지
AI Domain (예: GCP 또는 별도 환경)
AI Orchestrator API (FastAPI): Job 라우팅/큐잉/콜백 전송(운영 로직)
Message Queue (SQS/RabbitMQ/Kafka 등 택1): 비동기 작업 버퍼
Healthcare Worker (Pipeline Runner): 영상 처리 파이프라인 실행
CV Inference (GPU): 탐지/추적, 키포인트 추정
Postprocess/Scoring (CPU): 시계열 계산/점수화/레벨링
3-2. 모듈별 책임(domain)과 분리 이유
Client(App/Web)
보행 영상 촬영 및 업로드 요청
Job 생성 요청 후 job_id 수신
상태 폴링(또는 SSE/WS)로 진행률 표시
완료 시 결과 조회 및 UI 렌더링
분리 이유

UI/UX 변경이 AI 처리 파이프라인에 영향을 주지 않도록 경계 고정
Backend API Server (Spring)
DB/Redis write owner
인증/권한(사용자/반려견 소유 관계)
영상 업로드 처리 → Object Storage(S3) 저장 → video_url 확보
AI Orchestrator에 Job 생성 요청(내부 호출)

Orchestrator로부터 콜백 수신(queued/progress/complete/failed)
Redis에 Job 상태/진행률 저장
MySQL에 결과 저장 및 Client 응답 포맷 통합
분리 이유

Backend가 직접 추론을 수행하면 지연/장애가 서비스 전체로 확산됨
서비스 도메인 로직(산책/반려견/기록)과 AI 운영 로직(큐/워커/타임아웃)을 분리해야 안정적 운영 가능
Redis (Job Status Cache)
Backend only
Job 상태/진행률 저장소 (queued/processing/succeeded/failed, stage/percent/message)
Client의 상태 조회 요청에 빠르게 응답하기 위한 캐시
분리 이유

조회 트래픽을 MySQL에 몰지 않고, UX(진행률) 응답을 경량화
write owner를 Backend로 고정해 책임/권한 혼선을 제거
MySQL (Service DB)
Backend only
분석 결과 영속 저장(analysis record)
사용자/반려견/분석 기록과 연계(추세 분석/히스토리/리캡 확장 대비)
분리 이유

서비스 데이터 일관성(권한/소유/감사 로그) 관점에서 Backend가 단일 진입점이 되는 것이 안전
Object Storage (S3)
원본 보행 영상 저장
(선택) 오버레이 영상 / 리포트 이미지 저장
DB에는 URL만 기록
분리 이유

대용량 파일을 DB나 서비스 간 직접 전달하지 않기 위함
AI 워커가 직접 다운로드하여 처리 가능(참조 전달)
AI Orchestrator API (FastAPI)
No direct DB/Redis
Backend로부터 내부 Job 생성 요청 수신
입력 정책 결정(표준화 파라미터: max 10초/720p/12fps 등)
Queue에 Job enqueue
Worker 실행/운영 로직(재시도/타임아웃/워커 라우팅)
Worker로부터 진행/완료/실패 이벤트 수신
Backend로 상태/결과를 콜백 전송
분리 이유

AI 운영 로직은 서비스 도메인과 성격이 다름(큐잉/워커/재시도/타임아웃)
Orchestrator만 독립 배포/롤백 가능해야 모델/최적화 반복이 쉬움
Message Queue (Async Job Buffer)
Job을 Worker에게 전달하는 버퍼
요청 폭주 시에도 API 서버는 빠르게 응답(queued)
Worker 확장으로 처리량을 조절

분리 이유

동기 처리의 병목(p95/timeout)을 완화
트래픽 스파이크에 대해 시스템 탄력성 확보
Healthcare Worker (Pipeline Runner) + GPU/CPU 컴포넌트
Worker 책임

S3에서 영상 다운로드
프레임 샘플링
반려견 탐지/추적(필요 시)
키포인트 추정(GPU)
시계열 지표 계산 및 점수화(CPU)
(선택) 오버레이/요약 이미지 생성 후 S3 업로드
진행/결과 이벤트를 Orchestrator에 전달
분리 이유

GPU/연산 비용이 큰 작업을 Backend/Orchestrator에서 분리
Worker만 독립 확장/교체 가능(모델 변경 영향 최소화)
3-3. 모듈 간 인터페이스 설계(계약)
Client ↔ Backend (Public REST)
Job 생성

POST /api/healthcare/jobs
Request: (권장) multipart/form-data 로 file 업로드
Backend가 S3 저장 후 내부적으로 video_url 확보
Response: { "job_id": "string", "status": "queued" }
상태 조회

GET /api/healthcare/jobs/{job_id}
Response: { job_id, status, progress{stage, percent, message} }
Backend가 Redis 조회 후 반환
결과 조회

GET /api/healthcare/jobs/{job_id}/result
Response: 1단계에서 정의한 결과 JSON과 동일
Backend가 MySQL 조회 후 반환
헬스체크

GET /api/healthcare/health
Response: 1단계에서 정의한 결과 JSON과 동일
주의: Client는 Orchestrator를 직접 호출하지 않는다.

Backend ↔ AI Orchestrator (Internal REST)
“대용량 파일 전송 금지”, 참조 전달 원칙
Worker는 file을 직접 S3에서 가져옴
Job 생성 요청

POST /internal/ai/healthcare/jobs

- Request 예시:

```json
{
  "job_id": "job_abc123",
  "video_url": "https://storage/.../gait.mp4",
  "callback_url": "https://backend/internal/healthcare/jobs/job_abc123/events",
  "meta": { "user_id": "u_123", "dog_id": "d_456", "requested_at":
"2026-01-09T13:00:00+09:00" }
}
```

- Response 예시:

```json
{ "accepted": true, "job_id": "job_abc123" }
```

job_id 생성 주체는 Backend로 고정하면(권장) “DB/Redis write owner” 정책과 함께 트레이싱이 단순해집니다.
Orchestrator → Backend (Callback: 상태/결과 전달)
다이어그램 기준으로 콜백을 1개 엔드포인트로 통합하는 방식을 권장합니다(문서/구현 단순화).

권장(통합 이벤트)

POST /internal/healthcare/jobs/{job_id}/events
event_type: STATUS | PROGRESS | COMPLETE | FAILED
예시 1) queued/status

{
  "event_type": "STATUS",
  "status": "queued",
  "progress": { "stage": "VALIDATING", "percent": 0, "message": "대기열에 등록되었습니다."
}
}
예시 2) progress

{
  "event_type": "PROGRESS",
  "status": "processing",
 "progress": { "stage": "KEYPOINT_ESTIMATION", "percent": 55, "message": "키포인트를 추정하고 있습니다." }
}
예시 3) complete

{

```
  "event_type": "COMPLETE",
  "status": "succeeded",
  "result": { /* 1단계 결과 JSON과 동일한 구조 */ }
}
```

예시 4) failed

```
{
  "event_type": "FAILED",
  "status": "failed",
  "progress": { "stage": "KEYPOINT_ESTIMATION", "percent": 55, "message": "처리 중 오류가 발생했습니다." },
  "error": { "code": "INFERENCE_TIMEOUT", "message": "분석 시간이 초과되었습니다.", "details": { "timeout_ms": 15000 } }
}
```

Backend 처리 규칙

이벤트 수신 시:
Redis에 status/progress 업데이트
COMPLETE면 MySQL 저장(analysis record) + Redis succeeded 갱신
FAILED면 Redis failed + (선택) MySQL에 실패 로그 저장
보안/신뢰성(필수 권장)

콜백은 mTLS 또는 HMAC 서명/서비스 토큰으로 인증
Orchestrator는 콜백 실패 시 재시도
Backend는 멱등성 보장(job_id + event_type + event_seq 기반 중복 무해 처리)
Orchestrator ↔ Queue (Job 메시지 포맷)
대용량 파일 전송 금지(참조 전달)

Worker는 video_urll로 S3에서 직접 다운로드

```
{
  "schema_version": "1.0",
  "job_id": "job_abc123",
  "job_type": "GAIT_ANALYSIS",
  "inputs": {
    "video_url": "https://storage/.../gait.mp4",
    "max_duration_sec": 10,
    "target_fps": 12,
    "max_resolution": 720
  },
  "meta": {
    "user_id": "u_123",
    "dog_id": "d_456",
    "requested_at": "2026-01-09T13:00:00+09:00"
  }
}
```

Worker → Orchestrator (진행/완료 보고)
Worker는 단계별 진행 이벤트와 최종 결과를 Orchestrator에 전달

Orchestrator가 Backend로 이벤트 콜백 전송

3-4. 데이터 흐름 (비동기 Job 시나리오)
헬스케어 보행 분석은 영상 기반 시계열 처리로 처리 시간이 길어질 수 있으므로, Job 기반 비동기 처리를 기본 시나리오로 설계한다. Backend는 서비스 도메인(인증/업로드/저장/응답 통합)에 집중하고, AI 연산은 Orchestrator + Worker 계층에서 수행한다.

전체 흐름 요약 (요청 → 진행률 → 결과)

```
<li>
  <strong>Client → Backend</strong>
  <ul>
    <li><code>POST /api/healthcare/jobs</code></li>
    <li>사용자가 보행 영상을 업로드하고 분석 실행 요청</li>
  </ul>
</li>

<li>
  <strong>Backend → Object Storage (S3)</strong>
  <ul>
    <li>원본 영상 저장 후 접근 가능한 <code>video_url</code> 확보</li>
  </ul>
</li>

<li>
  <strong>Backend → AI Orchestrator</strong>
  <ul>
    <li><code>POST /internal/ai/healthcare/jobs</code>로 Job 생성 요청</li>
    <li>(<code>job_id, video_url, callback_url</code> 전달)</li>
  </ul>
</li>

<li>
  <strong>AI Orchestrator → Queue</strong>
  <ul>
    <li>Message Queue에 Job 메시지 enqueue</li>
  </ul>
</li>

<li>
  <strong>Orchestrator → Backend</strong>
  <ul>
    <li>queued 이벤트 콜백</li>
  </ul>
</li>

<li>
```

Client → Backend (Polling)

<code>GET /api/healthcare/jobs/{job_id}</code>로 진행률/상태 조회
폴링 (Backend는 Redis 조회)

Worker

job consume → S3 download → GPU 추론/CPU 스코어링 → (선택) artifacts 업로드

Worker → Orchestrator

progress/complete 보고

Orchestrator → Backend

progress/complete 이벤트 콜백

Backend

MySQL 저장 + Redis succeeded 갱신

Client → Backend

<code>GET /api/healthcare/jobs/{job_id}/result</code> (Backend는 MySQL 조회)

3-5. 독립 개발/배포/스케일링 고려사항

비동기 Job 기반 모듈화의 핵심은 컴포넌트별 독립성이다.

각 모듈은 독립적으로 개발/배포/스케일링 가능해야 하며, 이를 위해 계약(스키마) 과 저장 원칙을 명확히 한다.

독립 배포 단위

Backend(Spring): 도메인/권한/저장/응답

Orchestrator(FastAPI): 큐잉/워커 운영/콜백

Worker(GPU/CPU): 추론 파이프라인

Infra: Queue, Redis, MySQL, S3

스케일링 전략
Worker: Queue depth, 처리시간(p95), GPU utilization 기반 수평 확장

Orchestrator: 경량, 수평 확장 가능(다중 인스턴스)

Backend: API 트래픽 기반 확장(저장소/DB 커넥션 풀 고려)

버전/호환성
Queue 메시지: schema_version

결과: model_version(필수), 필요 시 result_version(권장)

필드 추가는 허용, 삭제/의미 변경은 v2로 분리

멱등성/중복 처리
job_id를 단일 상관관계 키로 사용

콜백 중복 수신 시 Backend 업서트로 무해 처리

동일 영상 중복 요청 방지(선택): user_id + video_hash + job_type

3-6. 모듈화로 기대되는 효과와 장점
장애 격리: AI 워커 장애가 Backend 전체 장애로 번지지 않음

확장성: Queue로 완충 + Worker 수평 확장

개발 분리: Backend/AI가 병렬 개발 가능(계약만 맞추면 됨)

운영 용이성: 콜백 기반 상태/결과 이벤트로 관측/재시도/통계 수집 단순화

3-7. 서비스 시나리오 부합 근거 (구체 사례)
사례 1) 아티팩트 생성 때문에 완료 지연

대응: complete 이벤트는 “점수/요약” 먼저, artifacts는 후행 생성 가능
권장: 결과 JSON에 artifacts_status: pending|ready|failed 추가(UX/운영 개선)
사례 2) 특정 기간 분석 요청이 급증(트래픽 폭주)

대응: Backend/Orchestrator는 빠르게 queued 반환, Queue가 흡수
Worker만 확장하여 처리량 대응
사례 3) 키포인트 모델 교체/최적화(ONNX/TensorRT, FP16 등) 필요

대응: Worker만 독립 배포/롤백
model_version으로 결과 추적 및 비교 가능
아키텍처 다이어그램

------- 수정 중 -------

4. 멀티스텝 AI/파이프라인 구현 검토 (모델 유형별)
노션 링크

헬스케어 영상 분석을 단일 추론이 아니라 단계별 파이프라인으로 처리한다. (다단계 추론 파이프라인)

멀티스탭 파이프라인 구성 단계(stage)
Validating 정책 검사(길이/코덱/해상도), 입력 무결성 확인 실패가 나면 여기서 바로 400/422로 끝남(비용 절약)
Fetching_video video_url 기반 다운로드 네트워크/스토리지 이슈가 주된 실패 원인
Sampling_frames 디코딩 + 리사이즈 + fps 샘플링(표준화의 핵심) 프레임 수 상한 고정이 여기서 결정됨
Dog_tracking (현재 단계에서는 제외) 반려견 위치를 안정적으로 잡는 단계 탐지→트래킹의 핵심이 여기 들어감
Keypoint_estimation 가장 큰 병목(GPU) + 품질을 좌우
Metric_scoring 시계열 정제 + 피처 계산 + 점수화 (핵심 결과는 이 단계에서 산출완료) CPU 벡터화 여부가 성능에 영향을 줌
Artifact_generation (선택적) 오버레이/리포트 이미지 생성(인코딩 비용 큼) UX 지연을 줄이려면 후행 처리가 가능해야 함
Finalizing 결과 payload 조립, model_version 기록, COMPLETE 이벤트 송신 저장/콜백 일관성의 마지막 관문
멀티 스탭 필요 근거
병목 관리 키포인트 추정 (keypoint_estimation) 은 GPU 병목 (프레임 수에 선형)
디코딩/후처리 (sampling_frames / metric_scoring) 는 CPU 병목 (디코딩/벡터화 여부) → 단일 덩어리로 묶으면 p95/타임아웃 원인 추적이 불가능
실패를 사용자 UX 로 연결 탐지 (dog_tracking) 실패 → 각도/거리/조명 안내 프레임 (insufficient_frames) 부족 → 5~10초 재촬영 → stage 별 실패 정의가 있어야 422 메시지가 의미를 가짐
Artifact (overlay/report) 분리 점수/요약은 빠르게 반환하고, 오버레이/리포트 는 후행 생성 가능하도록 설계 → 멀티스텝 아니면 구조적으로 불가능
진행률 (Percent) 보고
어떤 단계가 사용자 체감에서 오래 걸리는지를 반영하고, 실제 병목(키포인트) 구간이 percent 상으로도 크게 보이게 해서 진행률이 거짓말처럼 느껴지지 않게 하는 장치

이벤트 (progress) 송신 위치 (콜백 사용 위치)
각 stage 시작 시 : PROGRESS(stage, percent_start, message)
stage 내부에서 오래 걸리면 중간 업데이트 1~2회
stage 완료 시 : 다음 stage 로 넘어가면서 percent 갱신
최종 : COMPLETE(result) 또는 FAILED(error)
stage 설계 (상세)
공통 정책

max_duration_sec = 10
target_fps = 12
max_resolution = 720p
Worker는 대용량 전달 금지: video_url로 다운로드, 결과는 URL 참조로 저장
Backend 저장 규칙: Redis=status/progress, MySQL=result
stage 별 상세
Stage 역할 입력 출력 병목/리스크 실패 조건 → 에러
Validating 요청/정책 검증, 조기 종료 job payload(video_url, policy) validated_policy, trace_meta 거의 없음 파일 포맷/정책 위반 → INVALID_FILE_TYPE(400), VIDEO_TOO_LONG(400)
Fetching_video 영상 다운로드 준비 video_url local_path 네트워크 지연, S3 권한/만료 다운로드 실패 → (추가 권장) VIDEO_FETCH_FAILED(502/504) (없으면 INTERNAL로 통일해도 됨)
Sampling_frames 디코딩+리사이즈+FPS 다운샘플링 video_stream/local_path, policy frames[](또는 generator), fps_used, duration_sec, frames_sampled CPU 디코딩, p95 변동 프레임 부족/보행 부족 → INSUFFICIENT_FRAMES(422)
Dog_tracking
(현재 단계에서는 제외) 반려견 bbox 확보(탐지+트래킹), 크롭 영역 안정화 frames[] bboxes_ts(frame_idx→bbox+conf), crop_frames[](또는 crop params) 탐지 실패/개가 너무 작음/가림 탐지 성공률 낮음 or 트래킹 붕괴 → DOG_NOT_TRACKED(422)
Keypoint_estimation 크롭 프레임에서 키포인트 추정(GPU) crop_frames[] keypoints_ts(frame_idx→(x,y,conf)*K), kp_valid_ratio GPU 병목, 프레임 수에 선형 kp conf 낮아 유효프레임 미달 → DOG_NOT_TRACKED(422) 또는 INSUFFICIENT_FRAMES(422) (둘 중 하나로 표준화 추천)
Metric_scoring시계열 정제/피처/점수화 keypoints_ts, fps_used metrics, overall_risk_level, summary, processing CPU 후처리(루프/벡터화) 보통 fail 대신 degraded 처리 권장. (추가 가능) METRIC_UNSTABLE(422)
Artifact_generation
(선택적) 오버레이/요약 이미지 생성 및 업로드 frames[] + bboxes_ts + keypoints_ts + metrics artifact_urls(overlay_video, report_image), artifacts.status 인코딩/렌더링 병목, UX 지연실패해도 전체 성공 유지 권장: artifacts.status=failed, 에러 로그만 남김
Finalizing 결과 조립, COMPLETE 이벤트 송신 metrics, processing, artifact_urls 최종 result JSON(= /result) 거의 없음 콜백 실패(재시도 필요) → Orchestrator 레벨에서 처리
내부 데이터 형태 예시
의사코드에 사용하기 위한 구조 (JSON 고정은 아님)

bboxes_ts

```
{
  "frame_001": {"x1":0, "y1":0, "x2":0, "y2":0, "conf":0.93, "track_id":"t1"},
  "frame_002": {"x1":0, "y1":0, "x2":0, "y2":0, "conf":0.90, "track_id":"t1"}
}
```

keypoints_ts

```
{
  "frame_001": {
    "keypoints": [
```

```
      {"name":"left_knee", "x":0.0, "y":0.0, "conf":0.88},
      {"name":"right_knee","x":0.0, "y":0.0, "conf":0.91}
    ],
    "frame_conf": 0.86
  }
}
```


processing


```
{
  "analysis_id": "string",
  "job_id": "string",
  "analyze_at": "string(ISO8601)",
  "processing": {
    "analysis_time_ms": 0,
    "video_duration_sec": 0.0,
    "frames_sampled": 0,
    "fps_used": 0
  },
  "result": { ... },
  "metrics": { ... },
  "artifacts": { ... },
  "model_version": "string"
}
```


실패 기준

DOG_TRACKING 실패

탐지 성공 프레임 비율 < 60%

또는 트레킹 유지 실패 연속 구간 > 1초

→ DOG_NOT_TRACKED(422)

KEYPOINT_ESTIMATION 실패

유효 프레임 수 < min_frames (예: 60 프레임)

또는 하체 관철 평균 conf < 0.5

→ INSUFFICIENT_FRAMES(422)

→ 아니면 DOG_NOT_TRACKED(422) 로 통일

추천: “프레임 자체가 부족/보행 부족”이면 INSUFFICIENT_FRAMES “개는 있는데 자세가 안 잡힘/가림/각도 문제”면 DOG_NOT_TRACKED

Worker Pipeline Runner 설계

구성 요소

PipelineRunner : stage 순서대로 실행, 공통 로깅/타이밍/예외처리

VideoIO : 다운로드/디코딩/샘플링(스트리밍 가능?)

(현재 단계에서 제외) DogDetector/Tracker : bbox 확보 및 안정화

KeypointModel : 크롭 프레임 keypoints 추정 (GPU)

Scorer : keypoints_ts → metrics/summary

(미정) ArtifactGenerator : overlay 영상/summary 이미지 생성

이벤트 송신 규칙

Worker 는 Orchestrator 에게 내부 이벤트를 보내고, Orchestrator 가 Backend /events 로 전달된다. (Worker 가 Backend 를 직접 치지 않는 설계도 가능은 한데 현재 문서에서는 Orchestrator 가 콜백 송신자이므로 그 구조를 유지)

이벤트 타입 사용원칙 STATUS : queued 등 초기 상태 (보통 Orchestrator 가 담당) Worker 는 실질적으로 PROGRESS, COMPLETE, FAILED 를 만든다.

파이브라인 러너 의사코드

```
class PipelineRunner:
    def __init__(self, job, clients):
        self.job = job
        self.cb = clients.callback_client   # to orchestrator
        self.video = clients.video_io
        self.detector = clients.dog_detector
        # (현재 단계에서 제외) self.tracker = clients.tracker
        self.kp = clients.keypoint_model
        self.scorer = clients.scorer
        self.art = clients.artifact_generator  # optional
        self.t = {}  # timing buckets

    def emit_progress(self, stage, percent, message):
        self.cb.send_event(
            job_id=self.job.job_id,
            event_type="PROGRESS",
            status="processing",
            progress={"stage": stage, "percent": percent, "message": message}
        )

    def emit_failed(self, stage, percent, code, message, details=None):
        self.cb.send_event(
            job_id=self.job.job_id,
            event_type="FAILED",
            status="failed",
            progress={"stage": stage, "percent": percent, "message": message},
            error={"code": code, "message": message, "details": details or {}}
        )

    def emit_complete(self, result_json):
        self.cb.send_event(
            job_id=self.job.job_id,
            event_type="COMPLETE",
            status="succeeded",
            result=result_json
        )

    def run(self):
        started = now_ms()
        try:
```

```
        # (1) VALIDATING
        self.emit_progress("VALIDATING", 1, "입력/정책을 검증하고 있습니다.")
        policy = self._run_validating()

        # (2) FETCHING_VIDEO
        self.emit_progress("FETCHING_VIDEO", 8, "영상을 불러오고 있습니다.")
        stream = self._run_fetching_video(policy)

        # (3) SAMPLING_FRAMES
        self.emit_progress("SAMPLING_FRAMES", 20, "프레임을 추출하고 있습니다.")
        frames, meta = self._run_sampling_frames(stream, policy)

        # (4) DOG_TRACKING (detect -> track + crop)
        # (현재 단계에서 제외) self.emit_progress("DOG_TRACKING", 35, "반려견을
인식/추적하고 있습니다.")
        # (현재 단계에서 제외) track = self._run_dog_tracking(frames, policy)

        # (5) KEYPOINT_ESTIMATION
        self.emit_progress("KEYPOINT_ESTIMATION", 50, "키포인트를 추정하고
있습니다.")
        keypoints_ts, kp_stats = self._run_keypoint_estimation(frames, track, policy)

        # (6) METRIC_SCORING
        self.emit_progress("METRIC_SCORING", 80, "지표를 계산하고 있습니다.")
        metrics_payload = self._run_metric_scoring(keypoints_ts, meta, kp_stats, policy)

        # (7) ARTIFACT_GENERATION (optional)
        artifacts_payload = {"status": "pending", "keypoint_overlay_video_url": None,
"report_image_url": None}
        if policy.generate_artifacts:
            self.emit_progress("ARTIFACT_GENERATION", 92, "리포트 영상을 생성하고
있습니다.")
            artifacts_payload = self._run_artifact_generation(frames, track, keypoints_ts,
metrics_payload, policy)
        else:
            # 점수 먼저 반환하고, 후행 생성 가능 설계
            pass

        # (8) FINALIZING
        self.emit_progress("FINALIZING", 99, "결과를 정리하고 있습니다.")
        result_json = self._finalize(metrics_payload, artifacts_payload, meta, started)
        self.emit_complete(result_json)

    except KnownError as e:
        # e has: stage, percent, code, message, details
        self.emit_failed(e.stage, e.percent, e.code, e.message, e.details)
    except TimeoutError as e:
```

```
            self.emit_failed("KEYPOINT_ESTIMATION", 70, "INFERENCE_TIMEOUT", "분석
시간이 초과되었습니다.", {"timeout_ms": self.job.timeout_ms})
        except Exception as e:
            self.emit_failed("FINALIZING", 99, "INTERNAL_ERROR", "처리 중 오류가
발생했습니다.", {"exception": repr(e)})
```

stage 별 구현 포인트

validating

여기서는 파일 타입은 이미 Backend 에서 mp4 만 받도록 했더라도 Worker 에서 한번 더 체크

정책 위반이면 즉시 실패 처리

```
def _run_validating(self):
    p = self.job.policy
    if p.max_duration_sec > 10:
        raise KnownError("VALIDATING", 2, "VIDEO_TOO_LONG", "영상 길이가 너무
깁니다.", {"max_duration_sec": 10})
    return p
```

sampling_frames

ffmpeg 로 표준화 (720p/12fps) 고정

프레임 수 미달이면 422

```
def _run_sampling_frames(self, stream, policy):
    frames, meta = self.video.decode_and_sample(stream, fps=policy.target_fps,
max_res=policy.max_resolution)
    if meta.frames_sampled < policy.min_frames:
        raise KnownError("SAMPLING_FRAMES", 28, "INSUFFICIENT_FRAMES",
                "분석에 필요한 보행 장면이 부족합니다.", {"min_frames":
policy.min_frames})
    return frames, meta
```

dog_tracking (현재 단계에서는 제외)

초반 N 프레임 탐지 → 트레킹 → 신뢰도 하락 시 재추적

성공률 기반으로 dog_not_tracked

```
def _run_dog_tracking(self, frames, policy):
    init_bboxes = self.detector.detect_first_n(frames, n=policy.detect_init_frames)
    if init_bboxes.success_ratio < policy.detect_success_ratio:
        raise KnownError("DOG_TRACKING", 40, "DOG_NOT_TRACKED",
                "반려견을 안정적으로 추적하지 못했습니다.", {})
    track = self.tracker.track(frames, init_bboxes, redetect_every=policy.redetect_every)
    if track.valid_ratio < policy.track_valid_ratio:
        raise KnownError("DOG_TRACKING", 44, "DOG_NOT_TRACKED",
                "반려견을 안정적으로 추적하지 못했습니다.", {})
    return track
```

keypoint_estimation

여기만 모델 후보 2개로 비교분석

유효 프레임 비율/평균 conf 를 state 로 남기기

```
def _run_keypoint_estimation(self, frames, track, policy):
    crop_iter = self.video.crop_generator(frames, track.bboxes_ts, pad=policy.crop_pad)
    keypoints_ts, stats = self.kp.infer_timeseries(crop_iter, batch=policy.micro_batch)
    if stats.valid_frames < policy.min_frames:
        # 촬영 문제면 DOG_NOT_TRACKED로도 가능. UX 메시지 차이를 위해 구분 추천
        raise KnownError("KEYPOINT_ESTIMATION", 70, "DOG_NOT_TRACKED",
                         "키포인트를 안정적으로 추정하지 못했습니다. 촬영 각도/거리/조명을 바꿔
다시 촬영해 주세요.", stats.to_dict())
    return keypoints_ts, stats
```

self.kp = KeypointModelA() vs KeypointModelB()

같은 frames/crop 입력에 대해

t_keypoint_ms
valid_frames
mean_conf
실패율
로그로 컴토/비교

metric_scoring
실패보다는 degraded 처리

일부 프레임만 유효해도 점수 산출은 하고 confidence/summary 에 반영

```
def _run_metric_scoring(self, keypoints_ts, meta, kp_stats, policy):
    return self.scorer.score(keypoints_ts, fps=meta.fps_used, stats=kp_stats)
```

artifact_generation (선택적)
실패해도 전체 결과는 succeeded 유지하고 artifacts.status 만 failed)

```
def _run_artifact_generation(...):
    try:
        overlay_url = self.art.make_overlay_video(...)
        report_url = self.art.make_report_image(...)
        return {"status":"ready", "keypoint_overlay_video_url": overlay_url, "report_image_url":
report_url}
    except Exception as e:
        return {"status":"failed", "keypoint_overlay_video_url": None, "report_image_url": None}
```

finalizing
processing.analysis_time_ms 는 stage 별 타이밍 합으로 계산가능

model_version 은 keypoint 모델/scorer 버전 포함

```
def _finalize(self, metrics_payload, artifacts_payload, video_meta, started_ms):
    """
    FINALIZING:
```

```
    - 최종 결과 JSON 조립
    - processing / artifacts / model_version 채움
    """

    finished_ms = now_ms()

    result_json = {
        "analysis_id": f"gait_{self.job.job_id}",
        "job_id": self.job.job_id,
        "analyze_at": iso_now(),

        "processing": {
            "analysis_time_ms": finished_ms - started_ms,
            "video_duration_sec": video_meta.video_duration_sec,
            "frames_sampled": video_meta.frames_sampled,
            "fps_used": video_meta.fps_used,
        },

        "result": {
            "overall_risk_level": metrics_payload.overall_risk_level,
            "summary": metrics_payload.summary,
        },

        "metrics": metrics_payload.metrics,

        "artifacts": {
            "status": artifacts_payload.get("status", "pending"),
            "keypoint_overlay_video_url": artifacts_payload.get("keypoint_overlay_video_url"),
            "report_image_url": artifacts_payload.get("report_image_url"),
        },

        "model_version": self.model_version,
    }

    return result_json
```

멀티스텝 파이프라인 다이어그램
아키텍처 다이어그램

VALIDATING: 길이/정책/형식 확인, 조기 실패로 비용 절약
FETCHING_VIDEO: S3에서 다운로드/스트리밍 준비
SAMPLING_FRAMES: 720p/12fps로 표준화 + 프레임 추출
DOG_TRACKING (현재 단계에서는 제외): 초반 탐지 → 트래킹 → 크롭 안정화
KEYPOINT_ESTIMATION: 크롭 프레임에서 키포인트 추정(GPU 병목, A/B 비교 대상)
METRIC_SCORING: 시계열 지표 계산 + 점수화 + 요약
ARTIFACT_GENERATION (선택적): 오버레이/요약이미지(실패해도 결과는 성공 가능)
FINALIZING: /result 스키마 조립 + COMPLETE 콜백
비교 분석 설계
최소 의사코드

```
def benchmark_keypoint_models(video_set, models):
    results = []
    for video in video_set:
        frames, meta = sampling_frames(video, fps=12, max_res=720)
        # (현재 단계에서 제외)track = dog_tracking(frames)  # detect->track->crop

        for model in models:  # [ModelA, ModelB]
            t0 = now_ms()
            keypoints_ts, stats = model.infer_timeseries(frames, track)
            t1 = now_ms()

            results.append({
                "video_id": video.video_id,
                "tag": video.tag,  # good/hard/fail
                "model": model.name,
                "t_keypoint_ms": t1 - t0,
                "valid_frames": stats.valid_frames,
                "valid_ratio": stats.valid_frames / meta.frames_sampled,
                "mean_kp_conf": stats.mean_conf,
                "status": "succeeded" if stats.valid_frames >= 60 else "failed",
                "fail_code": None if stats.valid_frames >= 60 else "DOG_NOT_TRACKED",
            })
    return results
```

측정 항목
성능 (속도)
t_keypoint_ms : 키포인트 추론 시간(ms)

t_total_ms : 파이프라인 전체 시간(ms)

안정성
valid_frames : 유효 프레임 수

valid_ratio : 유효 프레임 비율(valid_frames / frames_sampled)

mean_kp_conf : 주요 관절 평균 confidence

fail_code : 실패 사유가 되는 코드 (DOG_NOT_TRACKED/INSUFFICIENT_FRAMES)

정확도를 GT로 재기 어렵기 때문에, 유효프레임/신뢰도/실패율로 품질을 대체합니다.

서비스 시나리오 지표
처리시간(전체/키포인트 stage)

실패율(코드별)

GPU 메모리 피크 / 평균

처리 비용(초당 비용 * 평균 처리시간)

artifact 후행 처리 성공률 / 평균 추가 지연

지원/배포성 비교
설치 난이도

런타임 의존성

GPU 요구/속도

모델 파일 관리

운영 적합성(서빙/컨테이너/RunPod)

후보 모델 선정
실제 사용자 촬영 영상에 대한 안정성

실시간/비동기 서비스 파이프라인에의 적합성

GPU 환경(RunPod 등)에서의 배포·운영 용이성

실험 및 개선 반복이 가능한 엔지니어링 생산성

MideaPipe Pose 기반 키포인트
특징 Google에서 제공하는 경량 실시간 포즈 추정 라이브러리 CPU 환경에서도 동작 가능하며 모바일/웹 친화적 인체 관절 구조에 특화된 고정 keypoint 정의 사용

장점 설치 및 사용이 간단함 실시간 처리 성능이 우수함 모바일/엣지 환경에서는 매우 효율적

단점 인간 전용 모델로 설계되어 반려견 관절 구조와 맞지 않음 개의 다리, 무릎, 발목 등 보행 분석에 중요한 하체 관절이 정확히 표현되지 않음 keypoint 정의가 고정되어 있어 도메인 확장이 어려움

선택/비선택 이유 본 과제는 반려견 보행 분석이 핵심이며, 인간 관절 기준의 키포인트는 의미 있는 지표 산출로 연결되기 어렵다고 판단함 참고/비교 용도로만 검토

MMPose 계열 Animal Pose 모델
특징 OpenMMLab 기반의 연구용 포즈 추정 프레임워크 다양한 animal pose 연구 모델 존재 고정밀 실험 및 논문 재현에 적합

장점 정확도 중심의 연구 실험에 적합 다양한 백본/헤드 구조 선택 가능 GT 기반 정량 평가에 유리

단점 설정 및 의존성 구성이 매우 복잡 추론 파이프라인이 무겁고 지연이 큼 모델/데이터/설정 관리 비용이 높음 서비스 환경에서의 배포·운영 부담이 큼

선택/비선택 이유 본 과제는 연구 성능 경쟁이 아니라 실제 서비스 파이프라인에 통합 가능한 구조 설계가 목적 MMPose는 연구 비교용으로는 적합하나, RunPod 기반 GPU 워커 / 비동기 API 구조에는 과도하다고 판단

YOLOv8-Pose (Ultralytics) 기반 Dog/Animal Keypoints
특징 객체 탐지 + 키포인트 추정을 단일 모델에서 수행 Ultralytics 생태계 기반으로 학습/추론/배포가 단순 Dog-Pose 등 반려견 전용 데이터셋과 직접 연계 가능

장점 서비스 친화적인 구조 (단일 .pt 파일로 배포 가능, Python/CLI 모두 간단한 사용성) GPU 환경(RunPod, Colab)에서 안정적인 추론 성능 객체 탐지 + 키포인트가 결합되어 dog_tracking/crop 단계를 옵션으로 분리 설계 가능 학습/파인튜닝/재현이 용이하여 성능 개선 반복에 적합

단점 MMPose 계열 대비 절대적 연구 정확도는 낮을 수 있음 GT 기반 정밀 평가에는 한계 모델 크기/설정에 따라 GPU 메모리 사용량 증가 가능

선택 이유 본 과제의 목적(서비스 지향 멀티스텝 파이프라인 설계)에 가장 부합 실제 사용자 영상 기반의 실패율/안정성 지표를 중심으로 판단할 때, 학습/추론/배포/운영의 균형이 가장 우수함 RunPod 기반 GPU 워커, 비동기 분석 API, 단계별 실패 처리 구조에 자연스럽게 통합 가능 따라서 YOLOv8-Pose 기반 Dog Keypoint 모델을 최종 선택함

실험 데이터셋
(아직 비교실험은 시작하지 않음) 특정 기준으로 선정한 것이 아닌 무작위 반려견 보행 영상 10개

결과 정리
YOLOv8s 로 반려견 보행 영상을 보내도 키포인트찍는 작업을 할 수 없다. VOLOv8 은 COCO 사람의 관절 키포인트 데이터로만 학습된 모델이기 때문이다. 영상안의 개를 탐지하는 것은 가능하지만

그래서 dog-pose 데이터셋을 가지고 모델을 간단하게 fine-tuning 한 후 테스트를 진행하였다. Fine-tuning 을이게 진행하였다.

그 후 fine-tuning 된 모델을 가지고 샘플 영상 10개를 입력하여 키포인트 예측 작업을 진행하였다. 결과는 ___ 로 출력되었다.

하지만 아직 제대로 fine-tuning 을 진행한 것도 아니고 세세한 값들을 제대로 설정하지 않았기 때문에 키포인트가 완벽하게 찍히지는 않는다고 판단했다. 입력 영상의 조건, 추론 파이프라인 문제, 그리고 모델 자체(학습) 문제가 있을 것으로 판단하여 이 부분들을 최대한 해결하고 추후 다시 테스트를 진행할 것이다.

입력/촬영 조건: 측면이 아니라 사선/정면에 가까움, 다리(특히 발)가 풀/그림자/사람 다리에 가림, 개가 화면에서 작음(해상도 대비 픽셀이 부족), 모션 블러(흔들림) / 야간 / 역광 이 경우는 모델이 “맞힐 근거”가 프레임마다 달라져서 점프가 생길 수 있다.

추론 파이프라인 문제 (V1에서 자주 발생): 프레임마다 다른 스케일/위치로 들어가서(리사이즈만 하고, 크롭/정규화가 없음) 키포인트 좌표가 흔들림, 한 프레임에서

인스턴스 선택이 바뀜 (예: 개+사람이 같이 잡힐 때, pose 인스턴스가 매 프레임 바뀌면 점프처럼 보임), conf가 낮은 키포인트를 그대로 쓰면 튐이 크게 보임

모델 자체(학습) 문제: 데이터가 산책 “보행” 도메인과 다름 (실내/정면 위주로 학습됐다거나), epochs 부족 / augmentation 부족, keypoint head가 아직 안정적인 표현을 못 배움

해결책 고민

“키포인트 사용 규칙”을 바꾸기: 프레임별로 모든 키포인트를 믿지 말고, rame_conf < t_frame 이면 해당 프레임은 버림 (예: 0.40.5) 또는 키포인트별로 kp_conf < t_kp 는 마스킹 (예: 0.30.5) 또는 마스킹된 구간은 보간(interpolate) 하거나 “유효 프레임 부족”으로 처리처럼 gate를 걸기. 이걸 하면 시각적으로도 튐이 크게 줄고, metric도 안정될 수 있다.

시간축 스무딩(필수): 보행 분석은 원래 “시계열”이라, 후처리로 안정화하는 게 정석이다. 추천 옵션으로는 One Euro Filter (실시간/지연 적음), Savitzky–Golay (매끄러운 곡선), 간단히는 EMA(지수이동평균) 도 충분히 효과 있음 포인트는 좌표를 프레임별로 스무딩, conf 낮은 프레임은 업데이트 비중을 줄이거나 업데이트 자체를 막기 이다.

“한 마리만” 고정해서 따라가기: 지금 코드는 프레임마다 “가장 평균 conf가 큰 인스턴스”를 선택하고 있는데, 프레임에 따라 선택이 바뀌면 키포인트가 순간이동한다. 따라서 첫 프레임에서 선택한 인스턴스의 bbox 중심을 저장하고 다음 프레임에서는 이전 프레임 bbox 중심과 가장 가까운 인스턴스를 선택 (간단한 tracking 역할) 한다. 이거 하나만 해도 “중구난방 점프”가 눈에 띄게 줄어들 것으로 예상된다.

또한 dog_tracking/crop 의 필요성에 대해서 자세하게 공부해봐야할 것 같다. track/crop 이 품질 안정화 옵션으로써의 가치가 있을 수도 있다고 생각해봤다. 입력 프레임에 대상이 작거나 배경이 복잡하면 흔들리기 위워서 bbox 기반으로 개만 잘라서(padding 포함) pose 에 넣으면 스케일/배경 변동이 줄어서 키포인트가 안정될 수도 있을 것이라고 예상한다.

다음으로는 fine-tuning 을 더 해봐야할 것 같다. 하지만 이 부분은 비용과 시간 측면도 함께 고려를 해봐야할 것 같다.

5단계 - 데이터/컨텍스트 보강 설계 (RAG, RA-IS, VRAG 등)
헬스케어 기능은 반려견 보행 영상 → 키포인트 시계열 → 지표/점수 산출이 핵심이므로, 외부 지식 검색이 점수 정확도를 직접 올려주지는 않는다. 다만 고도화 단계에서는 사용자가 이해하기 쉬운 설명/가이드 품질을 높이거나, 서비스 운영 측면에서 정책/주의문구의 일관성을 확보하기 위해 “검색 기반 컨텍스트 보강”이 유의미해질 수 있다.

따라서 본 단계에서는 다음 원칙으로 설계한다.

MVP: RAG 미사용(설명은 룰/템플릿 기반 NLG로 생성)
고도화: LLM을 설명 확장에만 제한적으로 사용 가능 (이때 컨텍스트 보강이 필요하면 외부 웹이 아닌 내부 검수 문서 기반 ‘미니 RAG’를 도입한다.)
RAG 인프라 재사용: 챗봇 기능에서 RAG 인프라가 구축되면(공통 Retriever/인덱싱/모니터링), 헬스케어도 같은 인프라를 사용하되 헬스케어 전용 인덱스/문서로 분리한다.
5-1. Text/LLM 서비스 : RAG 적용 여부 및 구조

기본적으로 RAG 는 사용하지 않는 것으로 정한다. 하지만 고도화 단계에서 추가가 가능하다.

MVP : RAG 미사용
목표는 빠르고 안전하며 일관된 결과 설명을 제공한다.

방식은 metric_scoring 에서 산출된 결과를 기반으로 설명 블록 템플릿 조합으로 결과 요약, 관측 근거, 가능한 원인, 주의/권장 행동, 재촬영 가이드, 또는 과정 안전 문구들이 있다.

고도화 : LLM + Mini RAG
적용 범위는 점수 계산이 아니라 설명/가이드 생성 품질에 한정할 수 있다.

검색 소스는 촬영 가이드, 실패 코드별 대응 가이드, 지표 해석 가이드, 주의 문주/안전 정책 등이 있다.

출력은 기존 /result 스키마를 유지하면서 result.summary, metrics.*.description 을 expanded 버전으로 생성한다..

미니 RAG 데이터 흐름 (고도화)

입력(헬스케어 결과) metrics, explain_features, quality_flags, dog_meta(optional)
검색 쿼리 생성 예: “슬개골 위험 high + 후지 지지 비대칭 + 촬영 품질 양호/불량”
내부 KB 검색(top-k)
검색된 문구를 컨텍스트로 LLM에 주입(정책 프롬프트 포함)
설명 생성(진단 금지/안전 문구 강제)
5-2. 비전 서비스: Visual RAG/유사 기법 적용 검토
헬스케어의 비전 파이프라인은 키포인트 추정과 시계열 분석이 핵심이며, 현재 설계에서 검색 기반으로 비전 입력을 보강(Visual RAG)하는 필요성은 낮다.

(1) MVP: Visual RAG 미적용 이유: 키포인트 모델은 입력 영상 자체에서 추정되며, 외부 영상/이미지 검색이 추론 품질을 직접 개선하기 어렵고, 구현 복잡도 대비 효익이 작다.

(2) 고도화(선택): 품질 진단/가이드 목적의 제한적 시각 레퍼런스 사용 Visual RAG를 모델 추론 보강으로 쓰기보다는, 다음처럼 설명/UX 보강에 제한할 수 있다. 촬영 실패 유형(역광/가림/너무 멀다/사선)별 예시 프레임을 내부 DB에서 검색해 보여주기 사용자의 업로드 영상에서 추출한 대표 프레임과 유사한 “좋은 촬영 예시”를 매칭하여 안내

※ 이 경우도 외부 데이터가 아니라 내부 제작 가이드 이미지/영상 DB를 사용하며, “학습”이 아니라 “안내” 목적이다.

5-3. 음성 서비스: Speech RAG 적용 여부
본 헬스케어 기능 입력은 영상(mp4)이며 음성 기반 질의/응답을 전제하지 않는다. 따라서 Speech RAG는 본 기능 범위에서는 미적용으로 결정한다.

5-4. 이미지/영상 생성 서비스: RA-IS / Retrieval-Augmented Video Generation 적용 여부
헬스케어 기능은 생성형(이미지/영상 생성)이 아니라 분석형 서비스다. 오버레이 영상/리포트 이미지는 “생성”이라기보다 **분석 결과의 렌더링(시각화)**이므로 RA-IS/Video Generation은 범위 밖이다.

미적용 사유: 생성 모델을 통한 합성 품질 개선이 아니라, 키포인트/지표를 정확히 시각화하는 것이 목적

대체 고도화: 오버레이 렌더링 템플릿/가독성(색/두께/관절 강조) 개선, 요약 리포트 레이아웃 개선 등 UI/시각화 고도화로 처리

5-5. 기타 모델/전략: 재학습·파인튜닝 및 “컨텍스트 보강” 대안
헬스케어에서 RAG 대신 더 큰 효과를 내는 보강은 다음이다.

(1) 파인튜닝/데이터 보강(핵심)
dog-pose 기반 키포인트 모델을 실제 산책 도메인(측면 보행, 야외, 가림/그림자)로 보강
촬영 조건별(정면/사선/야간/역광) 실패 케이스를 수집해 재학습 전략 수립
(2) 품질 게이팅(컨텍스트 보강의 현실적 대체)
kp_valid_ratio, mean_conf, “한 마리 고정 추적” 등으로 신뢰도 플래그를 산출
신뢰도가 낮으면 결과를 “불확실/재촬영 권장”으로 명확히 표시 (잘못된 확신형 설명을 막는 것이 실제 품질을 크게 올림)
(3) 사용자 히스토리 Retrieval(리캡/추세) — RAG라기보다 DB 조회
같은 dog_id의 과거 분석 결과를 조회하여 변화량을 요약 (월/연 리캡과 자연스럽게 연결)
외부 문서 검색 없이도 개인화가 가능하고, 헬스케어와 매우 궁합이 좋음
리캡/추세 기능의 경우 V4 의 최후순위 구현 기능이다.
6단계 - 표준화된 도구 통합 및 외부 API 활용 설계 (MCP 등)
댕동여지도 서비스는 반려견 산책을 중심으로 한 게이미피케이션 서비스로,

돌발 미션, 표정 분석, 헬스케어 분석, 수의사 상담 챗봇 등 다양한 AI 기능을 포함한다.

이러한 AI 기능들은 영상·이미지·텍스트 분석 등 비교적 높은 연산 자원을 요구하며,

서비스 고도화 과정에서 외부 도구 또는 상용 API를 활용해야 할 가능성이 존재한다.

본 단계의 목적은 다음과 같다.

AI 모델과 외부 도구·서비스(API)를 어떤 기준으로, 어떤 방식으로 연결할 것인지를 정의
MCP(Model Context Protocol)와 같은 표준화된 도구 통합 방식의 도입 여부를 합리적으로 판단
초기 버전(v1v2)과 고도화 버전(v3v4)에서의 현실적인 적용 전략을 구분
본 서비스는 초기 단계에서의 안정성과 단순성을 우선하며,

표준 프로토콜(MCP)은 필수 요소가 아닌 고도화 단계에서 선택적으로 도입 가능한 옵션으로 설계한다.

6-1. 외부 도구·API 연동이 필요한 시나리오 분석
댕동여지도에서 외부 시스템 또는 상용 API 활용 가능성이 있는 시나리오는 다음과 같다.

GPU 자원 부족 또는 부하 분산 상황

영상/이미지 분석 요청이 급증하여 내부 GPU 리소스가 병목이 되는 경우
특정 분석 작업을 상용 비전 API로 임시 대체하거나 보조 처리

챗봇 고도화 단계

챗봇이 단순 질의응답을 넘어
반려견 정보

산책 기록

헬스케어 분석 결과

최근 활동 요약

등을 참고하여 응답해야 하는 경우

향후 기능 확장 가능성

이미지 생성, 음성 합성, 요약 생성 등 신규 AI 기능 추가
외부 데이터 소스 또는 분석 도구 연동 필요성 증가
단, v1~v2 범위의 기능(돌발 미션, 표정 분석, 헬스케어, stateless 챗봇)은

모두 자체 모델 및 내부 파이프라인으로 충분히 처리 가능하므로,

외부 API 사용은 필수 요건이 아니다.

6-2. MCP 도입 여부에 대한 판단
MCP(Model Context Protocol)는 AI 모델과 외부 도구·데이터 소스를

표준화된 인터페이스로 연결하기 위한 프로토콜이다.

그러나 본 서비스의 초기 단계(v1~v2)에서는 다음과 같은 이유로 MCP를 도입하지 않는다.

연동해야 할 도구의 수가 제한적이며 구조가 단순함
이미 Backend API 중심의 명확한 모듈 분리 구조를 보유
MCP 서버/클라이언트 운영으로 인한 복잡도 증가 대비 실질적 이점이 제한적
과제 및 MVP 범위 내에서 과도한 아키텍처 확장이 불필요
따라서 v1~v2에서는 MCP를 사용하지 않고,

기존 REST 기반 인터페이스와 내부 모듈 분리 방식으로 외부 호출을 관리한다.

6-3. MCP 미도입 시 외부 API 및 도구 통합 방식
MCP를 도입하지 않는 초기 단계에서는 다음과 같은 방식으로 외부 API 및 도구를
관리한다.

Backend 중심 통합 구조
모든 외부 API 호출은 Backend 전담 모듈에서 수행
AI 모델은 외부 API를 직접 호출하지 않음
공통 호출 정책
Timeout, Retry, Rate Limit 정책 적용
실패 시 graceful degradation (기본 메시지, 이전 결과 활용 등)

외부 API 응답 결과 캐싱 가능
모듈화 원칙
외부 API 호출 로직은 독립 모듈로 분리
내부 모델 교체/외부 API 전환 시 영향 최소화
이 구조는 초기 서비스에서 안정성, 디버깅 용이성, 운영 단순성을 확보하는 데 중점을 둔다.

6-4. MCP 도입을 고려하는 고도화 단계(v3~v4)
서비스가 고도화되어 다음 조건을 만족할 경우 MCP 도입을 검토한다.

챗봇이 내부 데이터를 도구 형태로 조회·조합·요약해야 하는 경우
헬스케어 결과 조회 도구
산책 기록 요약 도구
반려견 프로필 조회 도구
내부 API 수 증가로 인해:
권한 관리
도구 접근 제어
유지보수 복잡도가 증가하는 경우
이때 MCP는 다음 범위로 제한적 도입한다.

적용 대상: 챗봇 및 대화형 AI
비적용 대상: 헬스케어/돌발미션/표정 분석과 같은 핵심 비전 파이프라인
목적: 내부 데이터 접근을 표준화하고, 추후 도구 확장에 대비
MCP 도입 여부는 성능, 응답 지연, 비용, 운영 부담을 비교 평가한 후 결정한다.

6-5. 확장성·유지보수성·비용 관점 평가
본 설계는 다음과 같은 장점을 가진다.

초기 단계에서는 단순한 구조로 빠른 개발과 안정적인 운영 가능
MCP 도입을 강제하지 않음으로써 불필요한 복잡도 제거
고도화 단계에서 MCP 또는 유사한 표준 프로토콜을 유연하게 추가 가능
외부 API 활용 시 비용·지연·보안 문제를 단계적으로 검증 가능
이를 통해 댕동여지도 서비스는

현재 요구사항을 충족하면서도 장기 확장에 대비한 AI 통합 구조를 확보한다.

모델 선정 (비교 분석)
4단계에서 왜 YOLO 를 선택했는 지 서술함

v5, v9, 또는 v10 를 사용하지 않는 이유
YOLOv8, v11, 26은 현재 Pose Estimation(키포인트 검출) 분야에서 가장 최적화된 라인업이다. 다른 버전들이 제외된 이유는 다음과 같다.

YOLOv5: YOLO 시리즈의 표준이지만, Pose 전용 아키텍처가 v8만큼 현대적이지 않습니다. 특히 Decoupled Head(분류와 회귀를 분리하는 구조)가 없어 관절 위치를 정밀하게 잡는 능력이 최신 모델보다 떨어진다.
YOLOv9/v10: * v9는 정확도는 높지만 연산량이 많고 구조가 복잡하여 실시간 서비스(모바일/웹)용으로는 v11이나 26보다 비효율적일 수 있다.

v10은 NMS(후처리) 제거를 통한 속도 최적화에 특화되어 있으나, 공식적으로 Pose Estimation보다는 Object Detection(박스 검출) 성능 최적화에 집중된 모델이다.
헬스케어처럼 정밀한 관절 위치가 중요한 작업에는 v8 이상의 Pose 전용 모델이 더 유리하다.

모델별 성능 비교 분석

모델명	파라미터 (Params)	GFLOPs	Box mAP50-95	Pose mAP50-95	추론 속도 (Inference)	특징 및 비고
YOLOv8n	3.45M	9.9	0.889	0.341	0.5ms	가장 가벼우나 정확도 낮음
YOLO26n	3.27M	9.1	0.884	0.547	0.5ms	Nano 중 압도적 성능
YOLOv8s	11.86M	31.2	0.902	0.454	0.8ms	표준적인 성능
YOLO11s	10.15M	24.0	0.913	0.487	0.8ms	v8s 대비 효율적
YOLO26s	10.85M	25.9	0.894	0.598	0.9ms	Small 모델 중 최상위
YOLOv8m	26.74M	82.2	0.918	0.522	1.3ms	중간 체급, 무거운 편
YOLO11m	21.02M	72.0	0.919	0.514	1.4ms	v11 중간 체급
YOLO26m	22.15M	75.9	0.901	0.607	1.5ms	전체 모델 중 정밀도 1위

각 지표의 의미

지표명	의미 (설명)	헬스케어 서비스에서의 중요도
Box(P, R)	Precision(정밀도): 모델이 강아지라고 찾은 것 중 실제 강아지인 비율. Recall(재현율): 실제 강아지들을 놓치지 않고 찾아낸 비율.낮으면 반려견 자체를 못 찾거나 배경을 강아지로 오인함.	
mAP50	IoU(겹침 정도)가 50%만 넘어도 정답으로 인정했을 때의 평균 정확도.	모델의 기본적인 검출 능력을 보여주는 지표.
mAP50-95	IoU 임계값을 0.5부터 0.95까지 엄격하게 높여가며 계산한 평균치.	가장 중요. 이 수치가 높아야 강아지의 박스나 관절 위치가 미세하게 정확함.
Pose(P, R...)	박스가 아니라 관절(Keypoint)의 위치가 얼마나 정확한지를 나타내는 지표.	헬스케어 서비스에서 슬개골 탈구, 보행 분석 등을 할 때 핵심 지표.

모델 별 특징 및 장단점

모델	특징	장점	단점
YOLOv8	가장 검증된 표준 모델	에코시스템(자료)이 많고 배포가 매우 쉬움.	최신 모델(v11, 26) 대비 정확도-속도 밸런스가 소폭 낮음.
YOLOv11	v8의 직계 후속작	v8보다 적은 파라미터로 더 높은 성능을 냄. 효율성 극대화.	출시된 지 얼마 안 되어 레퍼런스가 v8보다 적음.
YOLO26	2025~2026 최신 모델 RLE(오차 추정) 기술 적용으로 관절 위치 정밀도가 압도적으로 높음.	모델 구조가 무거워질 수 있고, 특정 하드웨어 최적화가 필요할 수 있음.	

선정 순위 및 선택 이유

1순위: YOLO26s-pose (높은 정밀도)

이유: 로그상 Pose mAP50-95가 0.598로, v8s(0.454)나 v11s(0.487)를 압도한다.

서비스 적합성: 헬스케어 서비스는 관절 각도 1~2도 차이가 분석 결과를 바꿀 수 있다. YOLO26은 RLE(Residual Log-Likelihood Estimation)를 통해 키포인트의 불확실성을 계산하므로, 강아지가 움직이는 상황에서도 훨씬 정교하게 관절을 추적한다.

2순위: YOLOv11s-pose (높은 가성비/속도)

이유: 정확도(mAP 0.487)가 v8보다는 높으면서도, 파라미터 수가 적어 매우 가볍다.

서비스 적합성: 낙후한 환경에서는 v11s가 가장 쾌적한 사용자 경험을 제공합니다.

MideaPipe 와 YOLO 를 같이 사용하는 새로운 방식 고려 중

YOLO로 검출하고, MediaPipe로 정제하기

YOLO가 영상의 각 프레임에서 강아지 관절(Keypoints) 위치를 추론하면, 그 데이터는 원본이라 약간씩 흔들릴 수 있다. 이때 MediaPipe의 내부 알고리즘(예: One Euro Filter 등)을 활용해 관절 움직임을 부드럽게 보정한다.

효과: 사용자가 화면을 볼 때 관절 포인트가 파르르 떨리지 않고 매끄럽게 따라다닌다. (서비스 퀄리티 상승)
기술적 구현 방법: Mediapipe를 컨테이너로 활용
MediaPipe는 단순히 모델만 있는 게 아니라, 데이터를 실시간으로 처리하는 그래프(Graph) 구조를 가지고 있습니다.

YOLO 모델 수출: 학습시킨 .pt 파일을 TFLite 형식으로 변환합니다.
MediaPipe 이식: MediaPipe의 Object Detector나 커스텀 Task 기능을 이용해, 직접 만든 YOLO TFLite 모델을 MediaPipe 파이프라인 안에 태운다.
장점: 이렇게 하면 MediaPipe가 제공하는 안드로이드/iOS/웹 가속 기능을 그대로 쓰면서, 알맹이(모델)만 fine-tuning 한 고성능 YOLO26으로 바꿀 수 있다.

수의사 AI 챗봇 기능

Jade Junghoo Lee edited this page 10 hours ago · 10 revisions

모델 API 설계
본 수의사 AI 챗봇 모델 API는 사용자의 반려견 관련 질문(텍스트)과 필요 시 반려견 이미지(예: 피부/눈 등)를 입력으로 받아, 수의사 상담 말뭉치/지식 기반의 일반적 안내를 제공한다.

본 기능은 진단/진료/처방을 수행하지 않으며, 참고용 정보 제공 및 “병원 내원 권장/응급 판단 가이드(일반적 기준)” 수준의 답변을 목표로 한다.
결과는 사용자 Q&A 화면과 헬스케어 페이지 내 상담 버튼 섹션에 활용된다.
API 설계 개요
동기 처리
사용자는 질문을 입력하고 즉시 답변을 받는다(동기).

프론트/백엔드 연동을 단순화하기 위해 1단계에서는 동기 요청-응답 형태를 기본으로 한다.

고도화 계획
응답 지연이 길어질 경우 스트리밍(SSE/WebSocket) 또는 Job 기반 비동기로 확장 가능하도록 설계한다.

멀티모달(VLM) 사용 시, 이미지 업로드는 백엔드/스토리지에서 처리하고 Model API에는 image_url을 전달한다.

API Endpoint 목록 및 기능 설명
동기
Method Endpoint Desciption
POST /api/vet/chat 사용자 질문(text)과 선택 image URL 을 받아 상담 답변 생성
GET /api/vet/health Health check
사용자 질문 텍스트 수신
(선택) 이미지 URL 수신(피부/상처/눈 등)
안전/정책 필터링(의학적 진단·처방 방지, 응급 신호 시 권고 강화)
대화 히스토리(최근 N턴) 기반 맥락 구성
VLM(또는 LLM) 추론 수행
답변 + 주의 문구(디스클레이머) + 필요 시 “추가 질문(clarifying)” 반환
고도화(선택)
Method Endpoint Description
POST /api/vet/chat/stream 답변을 streaming 으로 전송 (SSE, etc..)
POST /api/vet/jobs 장문 분석/리포트 생성용 Job 생성
GET /api/vet/jobs/{job_id} Job 상태 조회
GET /api/vet/jobs/{job_id}/result Job 결과 조회
Input/Output 형식 명세
Request (JSON)

```
{
  "conversation_id": "enum",
  "message": {
    "role": "enum",
```

```
    "content": "string"
  },
  "image_url": "string",
  "history": [
    { "role": "enum", "content": "enum" },
    { "role": "enum", "content": "string" }
  ],
  "user_context": {
    "dog_age_years": int,
    "dog_weight_kg": int,
    "breed": "string"
  }
}
```

Field	Type	Required	Description
conversation_id	string	Yes	대화 세션 식별자(서버 생성 or 클라 생성)
message	object	Yes	이번 턴 사용자 메시지
message.role	enum	Yes	"user" 고정
message.content	string	Yes	사용자 질문 텍스트
image_url	string	No	반려견 관련 이미지(프리사인드 URL)
history	array	No	이전 대화 기록(최근 N턴)
user_context	object	No	(선택) 반려견 기본 정보/상황(나이, 체중 등)

Response (JSON)

```
{
  "conversation_id": "string",
  "answered_at": "string(ISO",
  "answer": {
    "content": "string"
  },
  "safety": {
    "disclaimer": "string",
    "urgent_signals": [
      "string",
      "string",
      "string"
    ],
    "urgent_recommendation": "string"
  },
  "followups": [
    "string",
    "string",
    "string"
  ],
  "processing": {
    "latency_ms": int,
    "model_used": "string"
  }
}
```

conversation_id: 대화 세션 식별자

answered_at: 응답 생성 시각(서버 생성)
answer.content: 사용자에게 보여줄 답변 본문
(페이지에 문구를 통일해서 박아놔도 된다.)safety.disclaimer: 진단/처방이 아님을 명시(필수 권장)
(필수아님)urgent_signals: 응급 판단에 도움 되는 일반적 경고 신호 목록
followups: 추가 질문(대화 품질 개선 및 안전성 강화)
processing.latency_ms: 처리 시간(성능 측정용)
processing.model_used: 모델 버전/이름
Error Response 명세
공통 Error schema

```
{
  "error": {
    "code": "STRING_CODE",
    "message": "사용자에게 표시될 메시지",
    "details": {}
  }
}
```

대표 Error schema

```
{
  "error": {
    "code": "INVALID_INPUT",
    "message": "질문 내용이 비어 있습니다.",
    "details": { "field": "message.content" }
  }
}
```

(400) INVALID_INPUT

```
{
  "error": {
    "code": "IMAGE_FETCH_FAILED",
    "message": "이미지를 불러오지 못했습니다. 다시 업로드해 주세요.",
    "details": {}
  }
}
```

(400) IMAGE_FETCH_FAILED

```
{
  "error": {
    "code": "LLM_TIMEOUT",
    "message": "답변 생성 시간이 초과되었습니다. 잠시 후 다시 시도해 주세요.",
    "details": { "timeout_ms": 15000 }
  }
}
```

(504) LLM_TIMEOUT

```
{
  "error": {
```

```
    "code": "RATE_LIMITED",
    "message": "요청이 많아 잠시 후 다시 시도해 주세요.",
    "details": {}
  }
}
```

(429) RATE_LIMITED

역할 및 연동 관계

역할

수의사 AI 챗봇 Model API는 상담 응답 생성 전용 컴포넌트로서, 메인 서비스(백엔드/프론트)에 LLM/VLM 호출 로직이 섞이지 않도록 분리한다.

본 API는 질문(텍스트)과 선택 이미지 입력을 받아,
상담 말뭉치 기반의 일반적 안내를 생성하고,
안전 디스클레이머 및 응급 신호 가이드를 함께 반환한다.

연동 관계

호출 주체: 메인 Backend API 서버 (클라이언트가 직접 모델 서버 호출하지 않음)

호출 시점: 사용자가 헬스케어 페이지에서 “수의사 상담 챗봇”을 열고 질문을 전송할 때

동작 흐름

Client(App)에서 질문 입력(선택 이미지 첨부)

Backend는 이미지가 있으면 스토리지 저장 후 image_url 발급

Backend가 Model API(/api/vet/chat) 호출

Model API가 답변/안전문구/추가질문 반환

Backend가 대화 로그를 저장하고 Client에 응답

Client는 챗 UI에 답변 표시

아키텍쳐에서의 위치

```
[Client(App)]
  │ (질문 전송 / 이미지 첨부)
  ▼
[Backend API Server]
  │ (인증/권한, 이미지 업로드 처리, 대화 로그 저장)
  ▼
[Vet Chatbot Model API]
  │ (VLM/LLM 추론 + 안전 가이드 생성)
  ▼
[DB] + [Object Storage]
(chat logs) + (uploaded images)
```

분리 이유 (설계 근거)

Model Reqeust/Prompt/안전 정책을 챗봇 API로 캡슐화해 서비스 코드 단순화

모델 업데이트(vLLM 적용, 모델 교체, 프롬프트 개선)가 서비스 로직과 분리되어 배포/유지보수 용이

API 호출 예시 및 예시 응답

호출 예시 (동기)

```
{
  "conversation_id": "conv_01HXYZ...",
  "message": {
    "role": "user",
    "content": "강아지가 귀를 계속 긁는데 진드기일까요?"
```

```
  },
  "image_url": "https://storage/.../dog_ear.jpg",
  "history": []
}
```

응답 예시

```
{
  "conversation_id": "conv_01HXYZ...",
  "answered_at": "2026-01-08T15:40:00+09:00",
  "answer": {
    "content": "귀를 지속적으로 긁는 경우 알레르기, 외이도염, 이물질, 진드기 등 다양한 원인이 있을 수 있습니다. 악취/분비물/붉어짐이 동반되면 병원 상담을 권장합니다. 집에서는 면봉 깊숙이 사용하지 말고, 겉부분만 부드럽게 확인해 주세요.",
    "tone": "informative"
  },
  "safety": {
    "disclaimer": "이 답변은 진단이 아닌 참고용 안내입니다. 증상이 심하거나 악화되면 수의사 진료를 받아 주세요.",
    "urgent_signals": ["심한 통증 반응", "피/고름 분비물", "고개를 한쪽으로 계속 기울임"],
    "urgent_recommendation": "위 신호가 있으면 빠른 시일 내 병원 내원을 권장합니다."
  },
  "followups": [
    "귀에서 냄새나 분비물이 보이나요?",
    "언제부터 증상이 시작됐나요?",
    "산책 후 풀숲에 자주 들어갔나요?"
  ],
  "processing": { "latency_ms": 2100, "model_used": "vet_vlm_v1" }
}
```

모델 추론 성능 최적화
기존 모델 추론 성능 지표
측정 대상: POST /api/vet/chat (동기), 1회 요청 기준

(텍스트만 / 텍스트+이미지 두 케이스로 나눠 측정)

텍스트-only 응답시간(p50 / p95): (예) 1.8s / 4.5s
텍스트+이미지(VLM) 응답시간(p50 / p95): (예) 4.5s / 9.0s
토큰 출력 속도(tokens/sec): (예) 20 tok/s
GPU 메모리 사용량: (예) 10–14GB(모델 크기/컨텍스트에 따라)
실패율: (예) 429(레이트리밋), 504(타임아웃) 간헐
이번 단계는 “측정 항목/목표/최적화 방향”을 확정하고, 다음주 상세 과제에서 실제 수치를 로그로 확정한다.

성능 병목 요소 및 원인
컨텍스트 길이(히스토리/프롬프트) 증가
원인: 히스토리/규칙/안전문구가 길수록 프리필(prefill) 비용 증가
증상: p95 급증, 특히 긴 대화에서 지연 증가
VLM 이미지 인코딩 비용
원인: 이미지 전처리/인코딩(비전 인코더) + 텍스트 생성 결합
증상: 텍스트-only 대비 지연이 크게 증가

모델 자체 크기 및 서빙 방식
원인: 미최적화 서빙(일반 torch), KV 캐시 비효율, 배치 처리 미흡
증상: GPU 메모리 압박, 동시 요청에서 처리량 저하
외부 API 사용 시 네트워크 지연/레이트 리밋
원인: 외부 LLM/VLM 호출은 네트워크 + rate limit에 민감
증상: 429/타임아웃 발생, 사용자 체감 불안정
적용할 최적화 기법의 구체적 계획
A. 프롬프트/히스토리 축소(가장 효과 대비 난이도 낮음)
히스토리 최근 N턴만 유지(예: 최근 6턴)
시스템 규칙/디스클레이머는 짧은 템플릿으로 고정
입력 길이 제한(너무 긴 질문은 요약 유도)
기대효과: prefill 시간 감소 → p95 개선

B. 응답 길이 제어 + 템플릿화
답변 최대 토큰 상한 설정(예: 300~500 tokens)

“응급 신호/주의 문구/추가 질문”을 구조화된 템플릿으로 생성 유도

(불필요한 장문 방지)

기대효과: 생성 시간 감소 + 비용 안정화

C. 캐싱(반복 질문/정형 답변)
FAQ류(예: 구토/설사/귀긁음 등 흔한 질문)는
의미 기반 캐시(간단 버전은 키워드/정규화 캐시로 시작)
같은 이미지 URL에 대한 비전 인코딩 결과를 단기 캐시(가능 시)
기대효과: 반복 요청에서 지연/비용 감소

D. 서빙 최적화(vLLM 등) / 양자화(가능 시)
자체 호스팅 시:
vLLM 적용(PagedAttention/KV 캐시 효율)으로 처리량 개선
FP16 기본 + 가능 시 INT8/4bit(정확도 영향 확인 후)
외부 API 사용 시:
모델 선택을 “빠른 모델(일반 상담)” vs “정확 모델(복잡/이미지)”로 분리
기대효과: 동시 요청 처리량 증가, GPU 메모리/지연 개선

E. 이미지 경로 최적화(VLM 케이스)
이미지 리사이즈/압축 정책(예: 1024px 제한)
이미지가 불필요한 질문은 텍스트-only 경로로 강제(라우팅)
예: “예방접종 일정”은 이미지 불필요 → LLM 라우팅
기대효과: VLM 호출 비율 감소 → 평균 지연/비용 감소

최적화 후 기대 성능 지표
텍스트-only p50 / p95
Before: (예) 1.8s / 4.5s
After 목표: 1.2s / 3.0s
텍스트+이미지(VLM) p50 / p95
Before: (예) 4.5s / 9.0s

After 목표: 3.0s / 6.5s
429/504 비율
After 목표: 1% 미만
처리량
vLLM 적용 시 동시 요청에서 throughput 개선(정량 수치는 다음주 측정으로 확정)
(메모) 측정/검증 방법
케이스를 분리해 전/후 비교
텍스트-only 30개 질문 세트
이미지 포함 30개 질문 세트(동일 해상도/용량)
로그 수집
prefill_time / generate_time / total_latency
tokens_in/out, 이미지 전처리 시간
p50/p95, 실패율(429/504) 비교
(추가메모) 모델 라우팅 규칙: Text-only vs VLM
노션에는 일단 추가 해놓음 해당 파트 노션 링크

서비스 아키텍쳐 모듈화
서비스 아키텍처 모듈화
아키텍처 다이어그램

챗봇 기능 설계 4주차 과제
3단계
3-1. 아키텍쳐 설계 원칙
호출 경계

Client(App/Web)는 Backend(Spring) 만 호출한다.
Client가 AI 도메인(Orchestrator/vLLM/RAG)을 직접 호출하지 않는다.
저장소 접근 정책 (권한/정합성)

MySQL/Redis write owner = Backend
AI 도메인(Orchestrator, Worker, vLLM)은 DB/Redis 직접 write 금지
AI 도메인은 결과를 Backend로 반환하고, Backend가 저장 및 응답을 통합한다.
대용량 전송 금지

이미지/파일 자체를 모듈 간에 직접 전달하지 않고, 참조(image_url) 만 전달한다.
Backend가 Object Storage에 저장하고 presigned URL을 발급한다.
확장성

기본은 동기 요청-응답
응답 지연이 길어지면 SSE(WebSocket) 스트리밍 또는 Job 기반 비동기로 확장 가능하도록 계약(스키마)과 모듈 경계를 유지한다.
3-2. 모듈화 적용 후 전체 아키텍쳐 개요
목표 구조(요약)
Backend는 서비스 도메인(인증/권한/로그/저장/클라이언트 응답)에 집중한다.
AI 도메인은 “답변 생성”에 필요한 추론 수행과 (선택) RAG 검색/컨텍스트 구성을 담당한다.
모델 변경(vLLM 적용, 모델 교체, 프롬프트 변경, RAG 적용/비적용)이 발생해도 Backend 코드는 최소 수정으로 유지한다.
구성 요소 목록

Service Domain (예: AWS)

Client (App/Web)
Backend API Server (Spring)
인증/권한
이미지 업로드 처리
대화 로그 저장
최종 응답 포맷 통합
MySQL (Service DB): 대화 로그/상담 기록 영속 저장 (Backend only)
Redis (Cache): 세션/레이트리밋/스트리밍 상태 등 캐시 (Backend only)
Object Storage (S3): 업로드 이미지 저장, DB에는 URL만 저장
AI Domain (예: GCP 또는 별도)

AI Orchestrator API (FastAPI): AI 요청 수신, 프롬프트/정책 처리, (선택)RAG 호출, vLLM 호출, 결과 반환
vLLM Inference Server (GPU): LLM/VLM 서빙 (챗봇 핵심 추론 엔진)
(선택) RAG Retriever/Vector DB: 문서 검색 및 컨텍스트 구성
(선택) Lightweight Worker (CPU): 문서 전처리/인덱싱, 로그 집계 등 운영 작업

3-2. 모듈별 책임과 분리 이유
Client
역할
질문 입력/응답 표시
(선택) 이미지 첨부
(선택) 스트리밍 UI(SSE/WS)로 토큰 단위 출력 표시
분리 이유
UX 변경이 AI 추론/인프라에 영향을 주지 않도록 경계 고정
Backend API Server (Spring)
역할
인증/권한(사용자/반려견 소유 관계, 접근 제어)
이미지 업로드 → S3 저장 → image_url 발급
AI Orchestrator에 내부 호출로 질문 전달
응답을 받아 대화 로그를 MySQL에 저장
결과를 Client로 반환 (응답 포맷 통합)
(선택) 레이트리밋/세션 캐시/스트리밍 세션 관리(Redis)
분리 이유
서비스 데이터 정합성과 보안을 위해 DB/Redis write를 Backend로 고정
프론트가 직접 AI에 붙지 않도록 하여 보안/관측/과금 관리 단순화
AI Orchestrator API (FastAPI)
역할
Backend로부터 AI 요청 수신(내부 API)
프롬프트 구성(대화 히스토리, 사용자 컨텍스트 반영)
안전 정책 적용(진단/처방 금지, 응급 권고 강화 등)
(선택) RAG 검색 호출하여 근거 컨텍스트 구성
vLLM 서버 호출하여 답변 생성
최종 결과를 Backend로 반환(동기) 또는 스트리밍 중계(선택)
분리 이유
모델/프롬프트/정책은 변경이 잦아 독립 배포 단위로 분리하는 것이 효율적

Backend가 모델 런타임(vLLM, 토크나이저, GPU) 세부를 몰라도 되도록 캡슐화
vLLM Inference Server (GPU)
역할
LLM/VLM 추론을 GPU에서 효율적으로 서빙
(선택) 스트리밍 출력 지원
분리 이유
GPU 리소스 운영은 서비스 도메인과 완전히 다른 성격
모델 스케일링/배치/스루풋 튜닝은 vLLM 레이어에서 집중 관리
(고도화 단계) RAG Retriever / Vecter DB
역할
사용자 질문 기반 top-k 문서 검색
(선택) rerank 후 컨텍스트 후보 압축
Orchestrator가 LLM에 넣을 “근거 컨텍스트” 생성
분리 이유
지식 베이스 업데이트/인덱싱은 추론과 별도의 운영 축
“근거 기반 답변”을 위해 검색 계층을 분리하면 품질 개선 반복이 쉬움
3-3. 모듈 간 인터페이스 계약 (설계)
Client ↔ Backend (Public REST)
A) 동기 상담
POST /api/vet/chat

Request (예시)
```
{
  "conversation_id": "string",
  "message": { "role": "user", "content": "string" },
  "image_url": "string|null",
  "history": [{ "role": "user|assistant", "content": "string" }],
  "user_context": { "dog_age_years": 0, "dog_weight_kg": 0, "breed": "string" }
}
```
Response (예시)
```
{
  "conversation_id": "string",
  "answered_at": "string(ISO8601)",
  "answer": { "content": "string" },
  "safety": {
    "disclaimer": "string",
    "urgent_signals": ["string"],
    "urgent_recommendation": "string"
  },
  "followups": ["string"],
  "citations": [
    {
      "doc_id": "string",
      "title": "string",
      "chunk_id": "string",
      "score": 0.0,
      "snippet": "string"
    }
```

```
  ],
  "processing": { "latency_ms": 0, "model_used": "string" }
}
```

citations는 RAG 사용 시 채워지고, 미사용 시 빈 배열 또는 null 처리 가능.

B) 스트리밍(선택)
POST /api/vet/chat/stream (SSE/WS 중 택1)

Response: 토큰 단위 chunk 스트림 + 마지막에 final JSON 메타 전송 (구현 방식은 Backend가 “스트림 프록시”를 하거나, Backend가 세션만 잡고 Orchestrator 스트림을 중계하는 형태 중 택1)

Backend ↔ AI Orchestrator (Internal REST)
A) 동기 상담 내부 호출
POST /internal/ai/vet/chat

Request (예시)

```
{
  "conversation_id": "string",
  "message": { "role": "user", "content": "string" },
  "image_url": "string|null",
  "history": [{ "role": "user|assistant", "content": "string" }],
  "user_context": { "dog_age_years": 0, "dog_weight_kg": 0, "breed": "string" },
  "options": {
    "use_rag": true,
    "max_tokens": 512,
    "temperature": 0.4
  }
}
```

Response (예시)

```
{
  "answer": { "content": "string" },
  "safety": {
    "disclaimer": "string",
    "urgent_signals": ["string"],
    "urgent_recommendation": "string"
  },
  "followups": ["string"],
  "citations": [
    { "doc_id": "string", "chunk_id": "string", "score": 0.0, "snippet": "string" }
  ],
  "processing": { "latency_ms": 0, "model_used": "string" }
}
```

B) 스트리밍 내부 호출(선택)
POST /internal/ai/vet/chat/stream

Orchestrator가 vLLM 스트림을 받아 Backend로 스트림 전달
Backend는 Client로 다시 SSE/WS 중계

3-4. 데이터 흐름
동기(기본) 흐름

Client → Backend: POST /api/vet/chat
(이미지 있으면) Backend → S3 업로드 후 image_url 확보
Backend → Orchestrator: POST /internal/ai/vet/chat
(선택) Orchestrator → Retriever/Vector DB: 검색 후 컨텍스트 구성
Orchestrator → vLLM(GPU): 답변 생성
Orchestrator → Backend: 답변/안전문구/추가질문/근거 반환
Backend → MySQL 저장 후 Client에 응답
스트리밍(선택) 흐름
3~6 단계가 스트리밍으로 동작
Backend는 “중계(Proxy)” 역할만 수행하고, 최종 메시지만 저장
Job 비동기(선택) 흐름
장문 리포트/요약 생성 등에서만 Job 형태 고려
헬스케어처럼 무조건 Job이 필수인 구조는 아니며, 기본은 동기/스트림이 자연스럽다.
3-5. 독립 개발/배포/스케일링 고려사항
독립 배포 단위
Backend(Spring): 인증/저장/응답 통합
Orchestrator(FastAPI): 프롬프트/정책/RAG/라우팅
vLLM(GPU): 모델 서빙 및 추론 효율 최적화
(선택) RAG(Vector DB/Indexer): 지식베이스 운영
스케일링 전략(초안)
vLLM(GPU): QPS/토큰 처리량/VRAM 기반 수평 확장(필요 시 멀티 GPU)
Orchestrator: CPU 기반 다중 인스턴스 수평 확장
Backend: API 트래픽 및 DB 커넥션 풀 기준 확장
RAG:
Vector DB는 매니지드 우선(운영 단순)
검색/재랭킹은 CPU로 시작, 필요 시 GPU reranker로 확장
버전/호환성
응답에는 model_used(모델 버전) 포함
RAG 사용 시 citations 구조는 “추가”만 허용하고, 의미 변경/삭제는 버전업(v2)으로 분리

3-6. 모듈화로 기대되는 효과와 장점
개발 병렬화: Backend/AI 팀이 API 계약만 맞추면 독립 구현 가능
유지보수 용이: 프롬프트/모델 교체(vLLM 포함)가 Backend 변경 없이 가능
운영 안정성: GPU 장애/추론 지연을 서비스 도메인과 격리
확장성: 스트리밍/Job/RAG 등 기능 확장을 AI 도메인 내부에서 점진 적용 가능
보안/정합성: DB/Redis write owner를 Backend로 고정해 권한/감사/정합성 관리 단순화

3-7. 서비스 시나리오 부합 근거
사례 1) 모델 교체/업데이트
vLLM 모델을 바꾸거나 프롬프트를 개선해도 Orchestrator/vLLM만 교체 배포 → Backend 영향 최소
사례 2) 응답 지연 증가
동기에서 체감 지연이 크면 /chat/stream로 전환해 UX 개선
API 계약은 유지하면서 전송 방식만 바꿀 수 있음
사례 3) 근거 기반 안내 필요(RAG 적용)

RAG를 도입해도 Backend는 citations를 그대로 전달/저장만 하면 됨
아키텍처 다이어그램

4단계 - 멀티스텝 AI/파이프라인 구현 검토 (모델 유형별)
멀티스텝 파이프라인 (stage) 정의
기본은 동기 /api/vet/chat이지만, 내부적으로는 아래 단계로 처리한다. (스트리밍은 동일 단계에서 “출력 전달 방식”만 달라짐)

Validating

필수 필드 검증(message.content 등), 길이 제한, conversation_id 유효성
rate limit / abuse 입력 차단(최소 수준)
Safety_screening (Pre)

“진단/처방 요청”, “약 용량 지시”, “응급 상황” 등 규칙 기반/경량 분류
응급 신호로 판단되면 답변 톤/권고를 강화하는 플래그 설정
Context_building

history 최근 N턴 컷(예: 6턴)
user_context(나이/체중/견종) 정규화
시스템 규칙(진단 금지/면책) 템플릿 주입
Image_preprocess (Optional)

image_url이 있으면 URL 접근 가능 여부 확인(백엔드가 presigned 발급했어도 Orchestrator에서 한 번 더 확인 가능)
리사이즈/압축 정책 적용(예: 1024px 제한) 또는 VLM 입력 규격 변환
Routing

텍스트-only vs VLM 경로 결정 예: 예방접종 일정/일반 질문은 텍스트-only로 강제
피부/눈/상처 상태 확인은 VLM 후보
(고도화) 라우팅 모델: 규칙 기반 → 경량 분류기 → LLM 라우터 순으로 발전 가능
Retrieval (Optional, 고도화)

RAG 사용 시: query 생성 → top-k 검색 → (선택) rerank → 컨텍스트 압축
MVP에서는 use_rag=false로 고정해도 되고, 인프라만 인터페이스 형태로 준비
LLM/VLM Inference

vLLM 또는 외부 API 호출
스트리밍이면 이 단계에서 토큰 스트림을 생성
Post_safety & Structuring

답변에서 진단/처방/단정 표현이 포함되었는지 후검사(규칙 기반)
답변을 “일반적 안내”로 완화, 불확실성 표현 강화
urgent_signals/urgent_recommendation 구성
followups(추가 질문) 생성(템플릿/모델)
(RAG 사용 시) citations 정리
Finalizing

latency_ms, model_used 기록
최종 JSON 스키마로 조립 후 Backend에 반환
Backend는 MySQL에 대화 로그 저장
멀티스텝이 필요한 이유
수의사 상담 챗봇은 “한 번 LLM 호출”로 끝낼 수도 있지만, 실제 서비스 품질/안전/성능을 생각하면 입력 검증 → 안전정책 → 컨텍스트 구성 → (선택)RAG → 모델 호출 → 후처리처럼 단계가 자연스럽게 분리된다.

안전/정책이 핵심 기능 요구사항: 진단/처방 금지, 응급 신호 시 권고 강화는 모델 호출 전/후에 명시적으로 제어가 필요
입력 형태가 다양: 텍스트-only vs 이미지 포함(VLM) → 라우팅/전처리가 필요
p95/timeout 관리: 히스토리 길이, RAG, VLM 인코딩이 지연을 키우므로 stage별로 병목을 분리해야 추적/개선이 가능
고도화 확장성: 스트리밍(/chat/stream)이나 Job(/jobs)로 확장하려면 단계가 이미 정의돼 있어야 한다

진행률/관측 설계
헬스케어처럼 퍼센트 진행률을 사용자에게 보여줄 필요는 낮지만(동기 응답), 내부 관측을 위해 stage 타이밍을 반드시 남긴다.

권장 로그 필드(Orchestrator 기준)

t_validating_ms
t_safety_pre_ms
t_context_ms
t_image_pre_ms(optional)
t_retrieval_ms(optional)
t_infer_ms
t_post_safety_ms
tokens_in / tokens_out
route = text_only | vlm
rag_used = true/false
error_code(timeout/429 등)
stage 설계 상세 표
Stage 역할 입력 출력 병목/리스크 실패 시 에러
Validating 필수값/길이/형식 검증 request JSON normalized_req 거의 없음 INVALID_INPUT(400)
Safety_screening(Pre) 응급/금지 요청 감지, 톤 플래그 message+history safety_flags 규칙 누락 가능 (보통 fail 없이 플래그 처리)
Context_building 히스토리 컷, 템플릿 주입 history, user_context prompt_pack 컨텍스트 과대 (보통 fail 없이 축소)
Image_preprocess 이미지 접근/규격화 image_url image_tensor/meta URL 만료, 파일 손상 IMAGE_FETCH_FAILED(400)
Routing text-only/VLM/RAG 결정 prompt_pack + image_meta route_plan 잘못 라우팅 시 비용↑ (fail 대신 기본값으로 fallback)
Retrieval(Optional) top-k 검색/압축 query citations + ctx 인덱스 품질/지연 (fail 시 rag_off fallback)

LLM/VLM Inference 답변 생성/스트림 prompt_pack raw_answer 가장 큰 병목 LLM_TIMEOUT(504), RATE_LIMITED(429)
Post_safety & Structuring 후검사/완화/필드 구성 raw_answer + flags final_answer 규칙 미흡 (fail 대신 안전 템플릿으로 교체)
Finalizing 응답 조립/메타 final_answer response JSON 거의 없음 INTERNAL_ERROR(500)
파이프라인 의사코드

```
def vet_chat_pipeline(req):
    t0 = now_ms()

    # 1) VALIDATING
    v = validate(req)  # raise INVALID_INPUT

    # 2) SAFETY (PRE)
    safety_flags = pre_safety_screen(v.message, v.history, v.user_context)

    # 3) CONTEXT BUILDING
    prompt_pack = build_context(
        message=v.message,
        history=cut_history(v.history, max_turns=6),
        user_context=normalize_user_context(v.user_context),
        safety_rules=SAFETY_TEMPLATE
    )

    # 4) IMAGE PREPROCESS (optional)
    image_blob = None
    if v.image_url:
        image_blob = fetch_and_resize(v.image_url, max_px=1024)  # raise IMAGE_FETCH_FAILED

    # 5) ROUTING
    route = choose_route(prompt_pack, image_blob, safety_flags)  # text_only / vlm
    use_rag = v.options.use_rag if hasattr(v, "options") else False

    # 6) RETRIEVAL (optional, 고도화)
    citations = []
    if use_rag:
        try:
            citations, rag_context = retrieve_and_compact(prompt_pack)
            prompt_pack = attach_rag_context(prompt_pack, rag_context)
        except Exception:
            # RAG 실패 시: RAG 없이 계속 진행 (품질만 약간 하락)
            use_rag = False

    # 7) INFERENCE
    try:
        raw_answer = call_model(prompt_pack, image_blob, route=route, max_tokens=512)
    except RateLimited:
```

```
        raise Error("RATE_LIMITED", 429)
    except Timeout:
        raise Error("LLM_TIMEOUT", 504)

    # 8) POST SAFETY + STRUCTURING
    safe_answer = post_safety_rewrite(raw_answer, safety_flags)
    followups = build_followups(v.message, safe_answer, safety_flags)

    resp = {
        "conversation_id": v.conversation_id,
        "answered_at": iso_now(),
        "answer": {"content": safe_answer},
        "safety": build_safety_payload(safety_flags),
        "followups": followups,
        "citations": citations if use_rag else [],
        "processing": {
            "latency_ms": now_ms() - t0,
            "model_used": current_model_version(route)
        }
    }
    return resp
```

멀티스탭 도입으로 기대되는 이점
안전성 강화: Pre/Post 안전 필터로 진단/처방 금지를 구조적으로 보장
성능/비용 안정화: Routing으로 불필요한 VLM 호출을 줄이고 p95 관리
고도화 용이성: RAG/스트리밍/Job 확장을 단계 추가/옵션 on/off 로 흡수
운영/디버깅 용이: stage별 latency로 병목을 정확히 집계(프롬프트 vs 인퍼런스 vs 이미지 전처리)

5단계 - 데이터/컨텍스트 보강 설계 (RAG, RA-IS, VRAG 등)
본 단계의 목표는 수의사 AI 챗봇의 응답 품질, 개인화 수준, 근거 일관성을 고도화하는 것이다. 이를 위해 단순한 “질문 → 답변 생성”을 넘어, 외부 데이터·과거 맥락·사용자/반려견 정보를 활용할 수 있는 확장 구조를 설계한다.

MVP 단계에서는 RAG를 사용하지 않는다.

AI-Hub 수의사 상담 말뭉치를 기반으로 학습된 LLM/VLM + 안전 정책 + 템플릿 응답으로 충분한 품질을 확보한다.
고도화 단계에서만 LLM 기반 컨텍스트 보강을 단계적으로 도입한다.\
학습(모델 파라미터)과 검색(Retrieval 기반 컨텍스트 주입)의 역할을 명확히 분리한다.
검색은 “항상”이 아니라 필요할 때만 조건부로 수행한다.
Backend는 데이터 소유자이며, AI 도메인은 검색·추론만 수행한다.
5-1. 학습 기반 접근 (Baseline)
AI-Hub 수의사 상담 말뭉치 학습
챗봇의 기본 추론 엔진은 AI-Hub에서 제공하는 수의사 상담 말뭉치 데이터를 활용하여 학습된다.

학습 목적

수의사 상담 특유의 말투 및 응답 구조 습득
진단/처방을 피하는 안전한 표현 패턴 학습
응급 신호 강조, 병원 내원 권고 방식의 일관성 확보
추가 질문(follow-up) 생성 능력 확보
학습 기반 접근의 장점

검색 없이도 안정적인 응답 품질 확보
지연 시간과 비용 예측이 용이
시스템 구조 단순화(MVP 적합)
한계

최신 정보 반영 한계
개인화(특정 반려견의 과거 기록/상태 반영) 제한
“왜 이런 답변을 했는지”에 대한 근거 설명 부족
→ 이러한 한계를 보완하기 위해 고도화 단계에서 Retrieval 기반 컨텍스트 보강을 도입한다.

5-2. 고도화 단계: Retrieval 기반 컨텍스트 보강 구조
고도화 단계에서는 RAG를 단일 개념으로 사용하지 않고, 검색 대상과 목적에 따라 3가지 유형으로 분리하여 설계한다.

Knowledge RAG (상담 지식/가이드 검색)
목적
일반적인 증상, 예방 관리, 응급 판단 기준에 대한 근거 일관성 강화
학습 데이터에 없는 표현 또는 모호한 질문에 대한 보완
검색 대상
수의사 상담 가이드 문서
일반적 증상/질환 설명 문서
응급 신호 체크리스트
(선택) AI-Hub 상담 말뭉치 요약/정제본
AI-Hub 원본 상담 대화는 “사례 기반 표현”이 포함되어 있으므로, 직접 처방/진단을 강화하지 않도록 검색 결과 필터링 정책을 적용한다.

활용 방식
Orchestrator가 사용자 질문 유형을 분류
“일반 기준 설명”이 필요한 경우에만 top-k 문서 검색
검색 결과를 요약/압축하여 LLM 프롬프트에 컨텍스트로 주입
필요 시 citations 필드로 근거 구조화 (UI 노출은 선택)
Conversation Memory RAG (이전 대화 맥락 검색)
목적
동일 사용자/반려견의 과거 대화 맥락 유지
반복 질문 또는 경과 질문에 대한 일관된 응답 제공
불필요한 재질문 감소
검색 대상
해당 conversation_id 또는 동일 반려견의 과거 대화 로그
대화 요약(summary) 및 핵심 사실(key facts)
권장 설계
최근 N턴은 프롬프트에 직접 포함
오래된 대화는 “요약/키팩트” 형태로 저장 후 검색

모델 답변보다는 사용자 발화 중심으로 memory 구성
주의점
과거 응답을 그대로 재사용하지 않도록 설계
이전 오판/오해가 누적되지 않도록 memory 우선순위 제한
User/Dog Context Retrieval (반려견·서비스 데이터 조회)
목적
개인화된 상담 응답 제공
사용자의 실제 상황(반려견 상태/생활 패턴)을 반영한 설명 생성
검색 대상
반려견 프로필(나이, 체중, 견종 등)
산책 기록 요약(최근 빈도/시간/변화)
헬스케어 분석 결과 요약(최근 리스크 수준, 특이사항)
구현 방식
Backend가 DB에서 조회 후 Orchestrator로 전달
또는 Orchestrator가 내부 API를 통해 Backend에 조회 요청
이 단계는 전통적인 Vector RAG가 아니라 DB 기반 컨텍스트 주입으로 설계한다.

5-3. Orchestrator 중심 라우팅 정책
고도화 단계에서 AI Orchestrator 는 다음과 같은 조건부 라우팅을 수행한다.

상황 적용 컨텍스트
일반 상담 질문 학습 모델만 사용
예방/응급 기준 설명 Knowledge RAG
“아까 말했잖아”, 경과 질문 Conversation Memory
“우리 반려견 기준으로” Dog Context Retrieval
복합 질문 필요 컨텍스트 조합
이를 통해 불필요한 검색/비용/지연을 최소화하면서 품질을 개선한다.

5-4. 호출 위치 및 아키텍쳐 연계
LLM(VLM)/RAG 호출은 AI Orchestrator(FastAPI) 에서 수행
Backend는 데이터 소유자이자 결과 통합자
외부 LLM API 사용 시에도 Orchestrator가 단일 호출 지점
실패 시 fallback 정책:
핵심 답변은 반환
설명/근거 영역에 오류 또는 제한 안내 문구 포함
5-5. 기대 효과
MVP에서는 구조 단순 + 안정적 UX 확보
고도화 단계에서:
응답 품질 및 개인화 수준 향상
동일 질문에 대한 일관성 확보
헬스케어/산책 기능과의 자연스러운 연계
RAG 인프라는 이후 챗봇 외 기능에도 재사용 가능(인덱스/정책 분리)
6단계 - 표준화된 도구 통합 및 외부 API 활용 설계 (MCP 등) - 모든 AI 기능 통합으로 작성
댕동여지도 서비스는 반려견 산책을 중심으로 한 게이미피케이션 서비스로,

돌발 미션, 표정 분석, 헬스케어 분석, 수의사 상담 챗봇 등 다양한 AI 기능을 포함한다.

이러한 AI 기능들은 영상·이미지·텍스트 분석 등 비교적 높은 연산 자원을 요구하며,

서비스 고도화 과정에서 외부 도구 또는 상용 API를 활용해야 할 가능성이 존재한다.

본 단계의 목적은 다음과 같다.

AI 모델과 외부 도구·서비스(API)를 어떤 기준으로, 어떤 방식으로 연결할 것인지를 정의
MCP(Model Context Protocol)와 같은 표준화된 도구 통합 방식의 도입 여부를 합리적으로 판단
초기 버전(v1v2)과 고도화 버전(v3v4)에서의 현실적인 적용 전략을 구분
본 서비스는 초기 단계에서의 안정성과 단순성을 우선하며,

표준 프로토콜(MCP)은 필수 요소가 아닌 고도화 단계에서 선택적으로 도입 가능한 옵션으로 설계한다.

6-1. 외부 도구·API 연동이 필요한 시나리오 분석
댕동여지도에서 외부 시스템 또는 상용 API 활용 가능성이 있는 시나리오는 다음과 같다.

GPU 자원 부족 또는 부하 분산 상황

영상/이미지 분석 요청이 급증하여 내부 GPU 리소스가 병목이 되는 경우
특정 분석 작업을 상용 비전 API로 임시 대체하거나 보조 처리
챗봇 고도화 단계

챗봇이 단순 질의응답을 넘어
반려견 정보

산책 기록

헬스케어 분석 결과

최근 활동 요약

등을 참고하여 응답해야 하는 경우

향후 기능 확장 가능성

이미지 생성, 음성 합성, 요약 생성 등 신규 AI 기능 추가
외부 데이터 소스 또는 분석 도구 연동 필요성 증가
단, v1~v2 범위의 기능(돌발 미션, 표정 분석, 헬스케어, stateless 챗봇)은

모두 자체 모델 및 내부 파이프라인으로 충분히 처리 가능하므로,

외부 API 사용은 필수 요건이 아니다.

6-2. MCP 도입 여부에 대한 판단
MCP(Model Context Protocol)는 AI 모델과 외부 도구·데이터 소스를

표준화된 인터페이스로 연결하기 위한 프로토콜이다.

그러나 본 서비스의 초기 단계(v1~v2)에서는 다음과 같은 이유로 MCP를 도입하지 않는다.

연동해야 할 도구의 수가 제한적이며 구조가 단순함
이미 Backend API 중심의 명확한 모듈 분리 구조를 보유
MCP 서버/클라이언트 운영으로 인한 복잡도 증가 대비 실질적 이점이 제한적
과제 및 MVP 범위 내에서 과도한 아키텍처 확장이 불필요
따라서 v1~v2에서는 MCP를 사용하지 않고,

기존 REST 기반 인터페이스와 내부 모듈 분리 방식으로 외부 호출을 관리한다.

6-3. MCP 미도입 시 외부 API 및 도구 통합 방식
MCP를 도입하지 않는 초기 단계에서는 다음과 같은 방식으로 외부 API 및 도구를 관리한다.

Backend 중심 통합 구조
모든 외부 API 호출은 Backend 전담 모듈에서 수행
AI 모델은 외부 API를 직접 호출하지 않음
공통 호출 정책
Timeout, Retry, Rate Limit 정책 적용
실패 시 graceful degradation (기본 메시지, 이전 결과 활용 등)
외부 API 응답 결과 캐싱 가능
모듈화 원칙
외부 API 호출 로직은 독립 모듈로 분리
내부 모델 교체/외부 API 전환 시 영향 최소화
이 구조는 초기 서비스에서 안정성, 디버깅 용이성, 운영 단순성을 확보하는 데 중점을 둔다.

6-4. MCP 도입을 고려하는 고도화 단계(v3~v4)
서비스가 고도화되어 다음 조건을 만족할 경우 MCP 도입을 검토한다.

챗봇이 내부 데이터를 도구 형태로 조회·조합·요약해야 하는 경우
헬스케어 결과 조회 도구
산책 기록 요약 도구
반려견 프로필 조회 도구
내부 API 수 증가로 인해:
권한 관리
도구 접근 제어
유지보수 복잡도가 증가하는 경우
이때 MCP는 다음 범위로 제한적 도입한다.

적용 대상: 챗봇 및 대화형 AI
비적용 대상: 헬스케어/돌발미션/표정 분석과 같은 핵심 비전 파이프라인
목적: 내부 데이터 접근을 표준화하고, 추후 도구 확장에 대비
MCP 도입 여부는 성능, 응답 지연, 비용, 운영 부담을 비교 평가한 후 결정한다.

6-5. 확장성·유지보수성·비용 관점 평가
본 설계는 다음과 같은 장점을 가진다.

초기 단계에서는 단순한 구조로 빠른 개발과 안정적인 운영 가능
MCP 도입을 강제하지 않음으로써 불필요한 복잡도 제거
고도화 단계에서 MCP 또는 유사한 표준 프로토콜을 유연하게 추가 가능
외부 API 활용 시 비용·지연·보안 문제를 단계적으로 검증 가능
이를 통해 댕동여지도 서비스는

현재 요구사항을 충족하면서도 장기 확장에 대비한 AI 통합 구조를 확보한다.