LLM 활용 소프트웨어

D-Day	마감
상태	완료
마감일	@2026/01/12
작업완료 여부	
작업 유형	

1. 시스템 개요

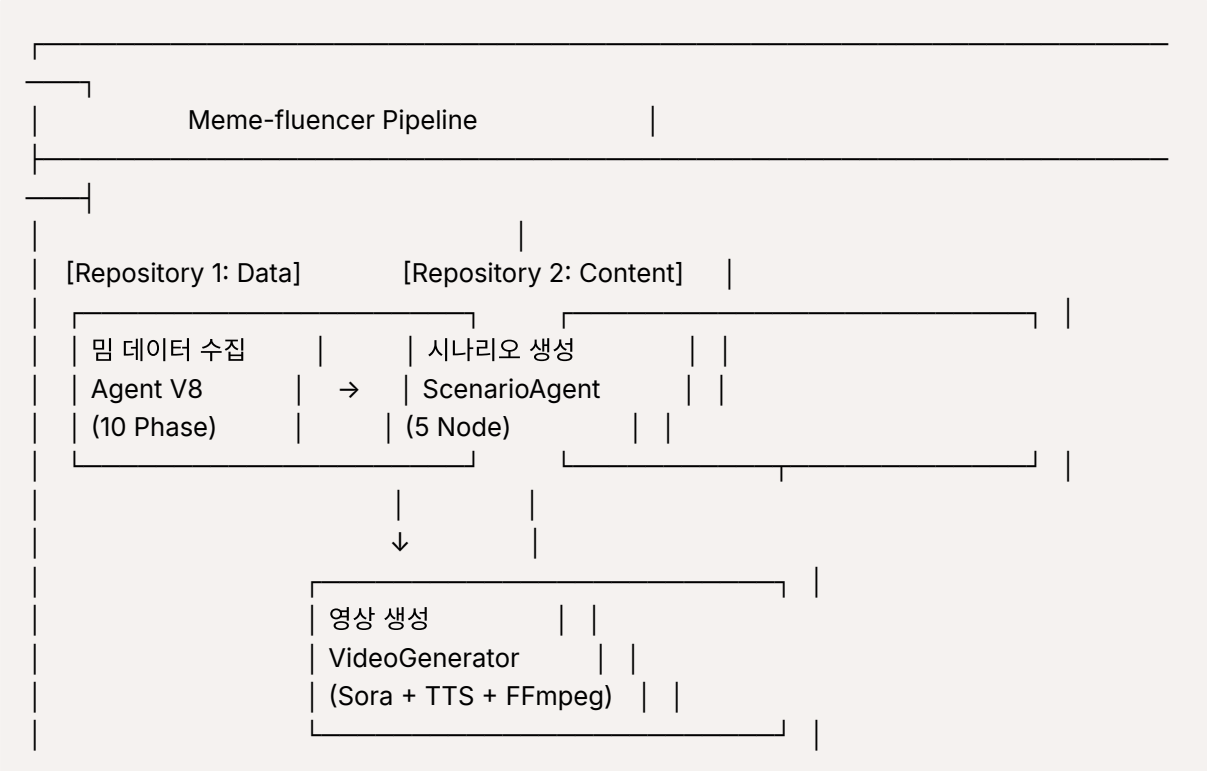
1.1 목적

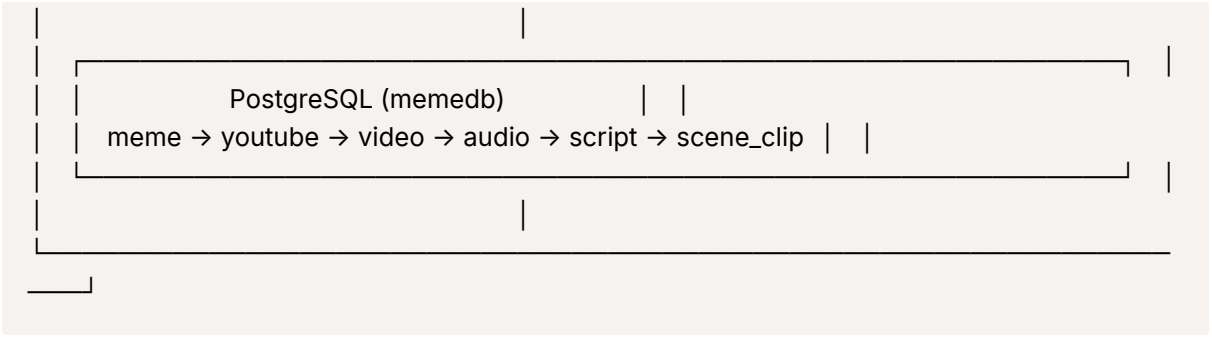
밈 데이터를 자동 수집하고, 캐릭터가 밈을 따라하는 숏폼 영상을 생성하는 End-to-End 파이프라인.

1.2 핵심 가치

단계	기존 방식	본 시스템
밈 조사	수동 검색 (30분+)	LLM 자동 수집 (5분)
시나리오 작성	직접 작성 (1시간+)	LLM 자동 생성 (5분)
영상 제작	촬영/편집 (수 시간)	AI 자동 생성 (10분)

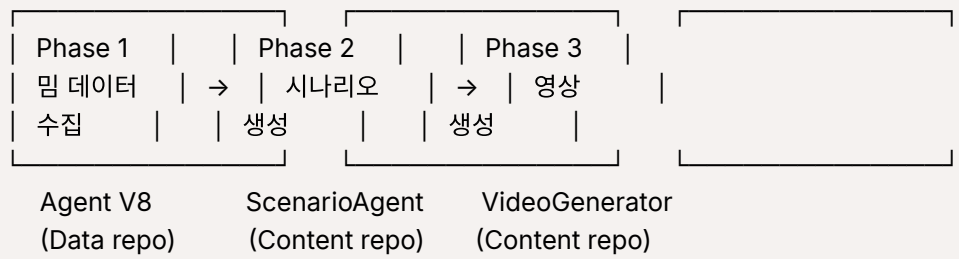
1.3 시스템 구성





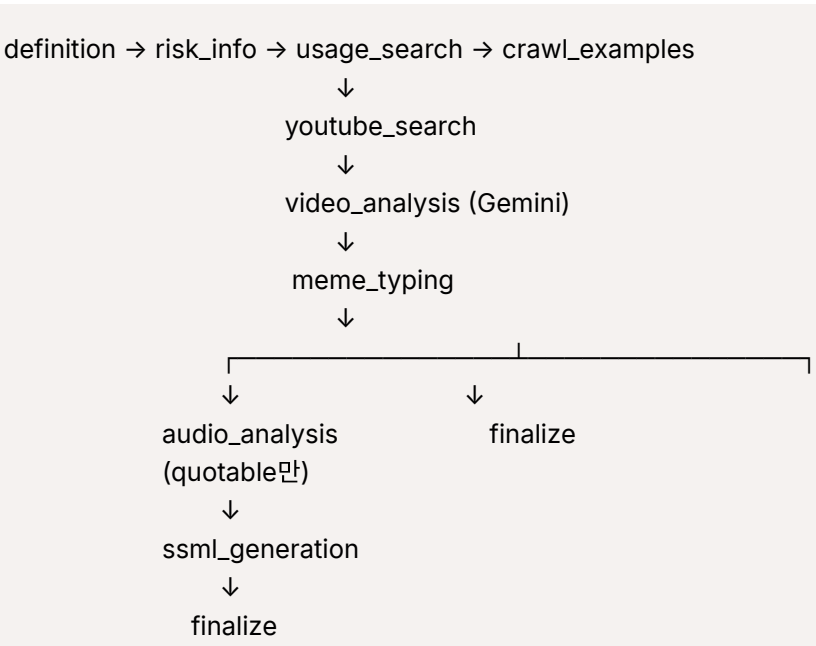
2. 전체 파이프라인 흐름

2.1 3단계 파이프라인



2.2 Phase 1: 밈 데이터 수집 (Agent V8)

10단계 LangGraph StateGraph 파이프라인



Phase	기능	LLM/API
1. definition	밈 정의/기원 수집	GPT-4o-mini + Serper
2. risk_info	논란/위험 정보 수집	GPT-4o-mini
3. usage_search	활용 예시 검색	Serper

Phase	기능	LLM/API
4. crawl_examples	URL 크롤링	-
5. youtube_search	YouTube 쇼츠 검색	YouTube API
6. video_analysis	영상 분석 + motion_prompt	**Gemini 2.0**
7. meme_typing	quotable/performable 분류	GPT-4o-mini
8. audio_analysis	오디오 구간 추출 + 운율 분석	librosa
9. ssml_generation	TTS용 SSML 생성	GPT-4o-mini
10. finalize	최종 JSON 생성	**GPT-4o**

2.3 Phase 2: 시나리오 생성 (ScenarioAgent)

5노드 LangGraph StateGraph 파이프라인

```
skeleton → dialogue → finalizer → [evaluate] → [save]
                                        ↓
                                (score < 60 && retry < 2)
                                        ↓
                                  ← skeleton (retry)
```

Node	기능	LLM(Test용)
skeleton	5단 구조 골격 생성	**GPT-4.1**
dialogue	캐릭터별 대사 생성	Fine-tuned EEVE
finalizer	placeholder 대사 삽입 + 정리	**GPT-4.1**
evaluate	품질 평가 (100점)	GPT-4o-mini
save	DB 저장	-

5단 씬 구조:

```
hook → setup → buildup → meme → punchline
(관심유도)  (상황설정)  (긴장고조)  (밈시전)  (마무리)
```

2.4 Phase 3: 영상 생성 (VideoGenerator)

7단계 순차 파이프라인

```
script 조회 → scene_clip 생성 → TTS 생성   →    Sora 영상 생성
                                        ↓
                                클립 합성 (FFmpeg)
                                        ↓
                                  최종 합성
                                        ↓
                                  DB 저장
```

단계	기능	API/Tool
TTS 생성	대사 → 음성	**OpenAI TTS**
Sora 영상	이미지 + 프롬프트 → 영상	**OpenAI Sora**

단계	기능	API/Tool
클립 합성	영상 + 오디오 합성	FFmpeg
최종 합성	클립 연결	FFmpeg

3. 코드 구조

3.1 Repository 구조

```
[Data Repository]
├── scripts/
│   ├── agent/                # Phase 1: 밈 데이터 수집
│   │   ├── meme_agent_v8.py      # 메인 Agent (LangGraph)
│   │   ├── state.py              # AgentV8State 정의
│   │   ├── tools_v8.py           # LangChain Tools (4개)
│   │   └── nodes/                # Phase별 노드 함수 (10개)
│   │
│   └── analysis/             # 분석 모듈
│       ├── video/analyzer.py     # Gemini 영상 분석
│       ├── audio/voice_analyzer.py # librosa 운율 분석
│       └── tts/ssml_generator.py # SSML 생성

[Content Repository]
├── backend/
│   ├── scenario/             # Phase 2: 시나리오 생성
│   │   ├── agent.py              # ScenarioAgent (LangGraph)
│   │   ├── graph.py              # StateGraph 구성
│   │   ├── nodes/                # 노드 함수 (5개)
│   │   └── generators/           # 생성기 (Skeleton, Dialogue, Finalizer)
│   │
│   ├── video/                # Phase 3: 영상 생성
│   │   ├── generator.py          # VideoGenerator (오케스트레이터)
│   │   ├── sora.py               # Sora API 클라이언트
│   │   ├── tts.py                # OpenAI TTS
│   │   └── composer.py           # FFmpeg 래퍼
│   │
│   └── db/                   # DB 연동
│       ├── repository.py         # ScenarioRepository
│       └── video_repository.py   # VideoRepository
```

4. 핵심 모듈 상세

4.1 Phase 1: MemeAgentV8 (LangGraph)

파일: `scripts/agent/meme_agent_v8.py`

LangGraph를 사용한 10단계 밈 데이터 수집 파이프라인.

```
class MemeAgentV8:
    """통합 밈 데이터 수집 에이전트 (v8)"""

    def _create_agent(self):
        """StateGraph 생성"""
        # LLM 초기화
        llm = ChatOpenAI(model="gpt-4o-mini", temperature=0)
        llm_with_tools = llm.bind_tools(ALL_TOOLS, parallel_tool_calls=True)
        llm_strong = ChatOpenAI(model="gpt-4o", temperature=0)

        # 노드 함수 정의
        def collect_node(state: AgentV8State) → AgentV8State:
            """도구 호출이 필요한 Phase 처리"""
            phase = state["phase"]

            if phase == "definition":
                return definition_node(state, llm_with_tools, llm)
            elif phase == "risk_info":
                return risk_info_node(state, llm_with_tools)
            # ... 생략

        def direct_node(state: AgentV8State) → AgentV8State:
            """도구 없이 직접 실행하는 Phase 처리"""
            phase = state["phase"]

            if phase == "youtube_search":
                return youtube_search_node(state)
            elif phase == "video_analysis":
                return video_analysis_node(state)
            elif phase == "meme_typing":
                return meme_typing_node(state, llm)
            # ... 생략

        # StateGraph 구성
        graph = StateGraph(AgentV8State)
        graph.add_node("collect", collect_node)
        graph.add_node("direct", direct_node)
        graph.add_node("finalize", finalize_node)
        # ... 엣지 연결
```

핵심 특징:

- `parallel_tool_calls=True`: 도구 병렬 호출로 속도 향상
- Phase별 노드 분리: 도구 호출 Phase / 직접 실행 Phase
- `needs_audio` 조건부 분기: quotable 밈만 오디오 분석 수행

4.2 Phase 1: Gemini 영상 분석

파일: `scripts/analysis/video/analyzer.py`

Gemini API로 YouTube 영상을 직접 시청하고 motion_prompt를 생성.

```
@retry_on_rate_limit(max_retries=3, initial_delay=60)
def generate_motion_prompt(
    video_id: str,
    meme_name: str,
    key_phrase: Optional[str] = None,
    model_name: str = "gemini-2.0-flash"
) → Dict[str, Any]:
    """
    Gemini로 영상 분석 + motion_prompt 생성

    Returns:
        {
            "time_stamp": {"start": 2.8, "end": 3.4, "detected_text": {...}},
            "video_generation": {
                "motion_prompt": "50-100 words English prompt...",
                "motion_sequence": [{"time": "0-2s", "action": "..."}],
                "body_parts": {"head": "...", "arms": "...", ...},
                "style_keywords": ["cute", "kawaii", ...],
                "negative_prompt": "static, blurry, ..."
            }
        }
    """
    client = get_client()

    # YouTube URL을 Gemini에 직접 전달
    video_url = f"https://www.youtube.com/watch?v={video_id}"

    prompt = f"""
    이 영상에서 "{meme_name}" 밈의 핵심 동작을 분석하세요.

    ## 출력 형식 (JSON)
    {{
        "time_stamp": {{"start": 시작초, "end": 종료초, "detected_text": {{...}}}},
        "video_generation": {{
            "motion_prompt": "50-100 words, detailed English motion description",
            "motion_sequence": [{{"time": "0-2s", "action": "..."}}],
            "body_parts": {{"head": "...", "face": "...", "arms": "...", ...}},
            "style_keywords": ["cute", "kawaii", "anime", ...],
            "negative_prompt": "static, blurry, distorted, ..."
        }}
    }}
    """

    response = client.models.generate_content(
        model=model_name,
```

```
            contents=[
                types.Content(parts=[
                    types.Part(text=prompt),
                    types.Part(video=types.Video(url=video_url))
                ])
            ]
        )

        return extract_json(response.text)
```

핵심 특징:

- Gemini가 YouTube 영상을 **직접 시청**하여 분석
- `@retry_on_rate_limit`: Rate limit 자동 재시도 (60초 대기)
- motion_prompt: Sora용 50-100 단어 영어 프롬프트

4.3 Phase 2: ScenarioAgent (LangGraph)

파일: `backend/scenario/graph.py`

LangGraph StateGraph를 사용한 시나리오 생성 파이프라인.

```
def create_scenario_graph():
    """시나리오 생성 그래프 생성"""
    workflow = StateGraph(ScenarioState)

    # 노드 추가
    workflow.add_node("skeleton", skeleton_node)    # GPT-4.1
    workflow.add_node("dialogue", dialogue_node)    # GPT-4o-mini
    workflow.add_node("finalizer", finalizer_node)  # GPT-4.1
    workflow.add_node("evaluate", evaluate_node)    # 품질 평가
    workflow.add_node("save", save_node)            # DB 저장

    # 순차 엣지
    workflow.set_entry_point("skeleton")
    workflow.add_edge("skeleton", "dialogue")
    workflow.add_edge("dialogue", "finalizer")

    # 조건부 엣지: 60점 미만이면 retry
    workflow.add_conditional_edges("evaluate", should_retry_or_save, {
        "retry": "skeleton",  # 골격부터 재생성
        "save": "save",
        "end": END,
    })

    return workflow.compile()


def should_retry_or_save(state: ScenarioState) -> str:
```

```
    """evaluate 후 라우팅: 60점 미만 + 재시도 미달이면 retry"""
    score = state.get("quality_score")
    retry_count = state.get("retry_count", 0)
    max_retries = 2

    if score < 60 and retry_count < max_retries:
        return "retry"

    return "save" if state.get("save_to_db") else "end"
```

핵심 특징:

- 품질 평가 후 **자동 재생성** (60점 미만, 최대 2회)
- skeleton → dialogue → finalizer 3단계 생성
- 프롬프트 버전 추적 (`prompt_version_ids`)

4.4 Phase 2: 골격 생성 (SkeletonGenerator)

파일: `backend/scenario/nodes/skeleton.py`

5단 구조의 시나리오 골격을 생성.

```
def skeleton_node(state: ScenarioState) → ScenarioState:
    """골격 생성 노드"""
    meme_data = state["meme_data"]
    characters = state["characters"]

    generator = SkeletonGenerator(model="gpt-4.1")
    skeleton = generator.generate(meme_data, characters)

    return {
        "messages": [AIMessage(content=f"[skeleton] 골격 생성 완료: {skeleton.title}")],
        "phase": "dialogue",
        "skeleton": skeleton.model_dump(),
    }
```

출력 스키마 (5단 구조):

```
{
  "scenes": [
    {
      "scene_type": "hook",
      "purpose": "시청자 관심 유도",
      "beats": [
        {"beat_id": "1-1", "type": "dialogue", "character": "사원", "text": "[PLACEHOLDER: 밈 언급]"}
      ]
    },
    {"scene_type": "setup"},
    {"scene_type": "buildup"},
```

```
    {
      "scene_type": "meme",
      "beats": [
        {"beat_id": "4-1", "type": "action", "character": "부장", "motion_prompt": "캐릭터가 밈 동작 수
행..."}
      ]
    },
    {"scene_type": "punchline"}
  ]
}
```

4.5 Phase 3: VideoGenerator (오케스트레이터)

파일: `backend/video/generator.py`

시나리오 → 최종 영상까지의 전체 파이프라인 오케스트레이션.

```
class VideoGenerator:
    """영상 생성 파이프라인 오케스트레이터"""

    def __init__(self):
        self.sora = SoraClient(model="sora-2")
        self.tts = TTSGenerator(model=파인튜닝된 TTS 모델)
        self.composer = VideoComposer()
        self.repo = VideoRepository()

    def generate(self, script_id: int) → GenerationResult:
        """script_id로 영상 생성"""

        # 1. 시나리오 조회
        script = self.repo.get_script(script_id)

        # 2. scene_clip 레코드 생성
        clips = self._create_scene_clips(script)

        # 3. TTS 생성 (dialogue beats)
        self._generate_tts_batch(clips)

        # 4. Sora 영상 생성 (action beats)
        self._generate_videos_batch(clips)

        # 5. 클립별 합성 (video + audio)
        composed_clips = self._compose_clips(clips)

        # 6. 최종 합성
        final_path = self._compose_final(script_id, composed_clips)

        # 7. DB 저장
        gen_id = self.repo.save_generated_video(script_id, final_path, cost)
```

```
    return GenerationResult(
        script_id=script_id,
        gen_id=gen_id,
        final_video_path=final_path,
        total_cost=cost
    )
```

4.6 Phase 3: SoraClient

파일: `backend/video/sora.py`

OpenAI Sora API를 사용한 영상 생성.

```
class SoraClient:
    """OpenAI Sora API 클라이언트"""

    def generate_from_image(
        self,
        image_path: str,
        prompt: str,
        duration: int = 4,          # 4, 8, 12초
        resolution: str = "portrait" # 720×1280 (세로 숏폼)
    ) → VideoResult:
        """이미지 + 텍스트 → 영상 생성"""

        # 이미지 base64 인코딩
        with open(image_path, "rb") as f:
            image_data = base64.standard_b64encode(f.read()).decode("utf-8")

        # API 호출
        video = self.client.videos.create(
            model=self.model,  # sora-2 또는 sora-2-pro
            prompt=prompt,
            image=f"{image_data}",
            duration=duration,
            resolution="720×1280"
        )

        # 비동기 폴링 (완료까지 대기)
        return self._wait_for_completion(video.id, timeout=300)

    def _wait_for_completion(self, video_id: str, timeout: int = 300) → VideoResult:
        """완료까지 폴링"""
        start_time = time.time()

        while time.time() - start_time < timeout:
            result = self.client.videos.retrieve(video_id)
```

```
            if result.status == "completed":
                return VideoResult(
                    video_id=video_id,
                    status="completed",
                    video_url=result.url,
                    cost_usd=self.COSTS[self.model] * result.duration
                )

            time.sleep(10)  # 10초 간격 폴링

        return VideoResult(video_id=video_id, status="timeout")
```

핵심 특징:

- 이미지를 **first frame**으로 사용하여 영상 생성
- 비동기 Job 기반: create → poll → download
- 비용 자동 계산: $0.10/초 (sora-2), $0.30/초 (sora-2-pro)

4.7 Phase 3: TTSGenerator

파일: `backend/video/tts.py`

파인튜닝한 TTS 모델을 사용한 음성 생성.

```
# 캐릭터별 음성 매핑 -> 파인튜닝한 목소리
CHARACTER_VOICES = {
    "부장": ,    # 저음, 무게감
    "사원": ,   # 중성, 젊음
}

class TTSGenerator:
    """OpenAI TTS 생성기"""

    def generate(
        self,
        text: str,
        character: Optional[str] = None,
        output_path: Optional[str] = None
    ) -> TTSResult:
        """텍스트 → 음성 생성"""

        # 캐릭터 → 음성 매핑
        voice = CHARACTER_VOICES.get(character, 부장목소리)

        response = self.client.audio.speech.create(
            model="tts-1",
            voice=voice,
            input=text,
            response_format="mp3"
```

```
        )

        response.stream_to_file(output_path)

        return TTSResult(
            audio_path=output_path,
            duration_seconds=self._get_audio_duration(output_path),
            cost_usd=(len(text) / 1_000_000) * 15.0  # $15/1M chars
        )
```

5. 기술 스택

5.1 LLM/AI 서비스

서비스	용도	모델
OpenAI	텍스트 생성, TTS, 영상 생성	GPT-4.1, GPT-4o, TTS-1, Sora-2
Google Gemini	YouTube 영상 분석	Gemini 2.0 Flash
YouTube API	쇼츠 검색	Data API v3
Serper	웹 검색	Google Search API

5.2 프레임워크

프레임워크	용도
LangGraph	Agent 파이프라인 (StateGraph)
LangChain	LLM 도구 바인딩
Pydantic	스키마 검증
FFmpeg	영상/오디오 합성
librosa	오디오 운율 분석

5.3 인프라

구성요소	기술
DB	PostgreSQL (memedb)
패키지 관리	uv
환경 관리	dotenv

6. 실행 명령어

6.1 Phase 1: 밈 데이터 수집

```
# 단일 밈 수집 + DB 저장
uv run python -m scripts.agent.meme_agent_v8 "밈이름" --db
```

```
# 배치 실행
uv run python -m scripts.agent.batch_runner
```

6.2 Phase 2: 시나리오 생성

```
# 시나리오 생성 + 평가 + DB 저장
uv run python -m backend.scenario.agent --meme-id 2 --save --evaluate
```

6.3 Phase 3: 영상 생성

```
# 영상 생성
uv run python -m backend.video.generator --script-id 1

# 개별 테스트
uv run python -m backend.video.tts --text "안녕하세요" --character 부장
uv run python -m backend.video.sora --prompt "A cartoon character waving"
```

---

7. 참고 자료

- OpenAI Sora API Documentation
- LangGraph Documentation
- Google Gemini API
