CNF

2026.01

구축형 AI Agent Builder 비교: 기업을 위한 Agentic AI 플랫폼 구축 가이드

생성형 AI 도입이 기업 경쟁력을 좌우하는 시대입니다.
그러나 많은 기업은 여전히 단순한 챗봇이나 일회성 PoC에 머물러 있습니다.
보안 문제로 **내부 데이터를 외부로 내보낼 수 없는 상황에서 최신 AI 에이전트를 어떻게 도입**할지, **우리 인프라에 가장 적합한 오픈소스는 무엇**인지
고민하고 있다면, 이 백서가 해답을 제시합니다.

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

Contents

1장. LLM 도입 이후, 왜 AI Agent Builder가 필요한가

1.1 기업 내 AI 컴포넌트 도입 현황: 열기 속의 파편화

오늘날 기업 환경에서 생성형 AI(Generative AI)와 대규모 언어모델(LLM)은 더 이상 실험적인 기술이 아닙니다. 2024년을 기점으로 많은 기업이 파일럿(시범 도입) 단계를 넘어, 실제 비즈니스의 핵심 도구로 AI를 적극 채택하고 있습니다. 실제로 전년 대비 관련 투자가 6배 이상 급증했다는 통계는 이러한 변화를 증명합니다.

하지만 급격한 성장 이면에는 '파편화된 도입'이라는 문제가 자리 잡고 있습니다. 전사적인 로드맵 아래 체계적으로 AI를 구축하기보다는, 각 부서나 팀이 필요에 따라 개별적으로 솔루션을 도입하는 경우가 많습니다. 이로 인해 기업 내 AI 시스템은 서로 연결되지 못한 채 고립된 실험 수준에 머무르는 경향이 짙습니다.

1.1.1 따로 노는 도구들: LLM, Naive RAG, 임베딩 검색의 분산

현재 기업들의 AI 도입 형태를 자세히 들여다보면, 마치 퍼즐 조각이 서로 맞지 않는 것처럼 분산된 구조를 보입니다.

- LLM 챗봇: 어떤 부서는 OpenAI API 등을 활용해 단순 대화형 챗봇을 도입합니다.
- 단순형 RAG (Naive RAG): 다른 부서는 사내 매뉴얼을 찾아주는 기초적인 검색 증강 생성(RAG) 시스템을 구축합니다.
- 벡터 검색: 또 다른 팀은 문서를 벡터 DB에 넣어 유사도 기반 검색 시스템을 만듭니다.

무엇이 문제인가요? RAG 기술은 기업 AI의 핵심으로 떠올랐지만, 대부분은 단순히 질문에 답을 한 번 하고 끝나는 1회성 질의응답 수준에 그칩니다. 기업의 72%가 생성형 AI 확대를 계획하고 있고, 28%는 사내 곳곳에 흩어진 데이터를 통합하려 시도하고 있지만, 현실은 녹록지 않습니다.

각 부서가 제각각 도구를 도입하다 보니 시스템 간에 대화가 불가능합니다. 이는 '데이터의 사일로(Silo, 고립)' 현상을 초래하여, 전사적인 지식 통합이나 시너지를 기대하기 어렵게 만듭니다. 즉, AI 도입 열기는 뜨겁지만, 통합된 전략의 부재로 인해 개별 도구들이 따로 노는 비효율이 발생하고 있습니다.

1.1.2 두 마리 토끼 잡기: VectorDB와 GraphDB의 병행 도입

이러한 한계를 극복하기 위해 기업들은 데이터를 저장하고 검색하는 방식에서 '하이브리드 전략'을 취하기 시작했습니다. 바로 Vector DB(예: Qdrant)와 Graph DB(예: Neo4j)를 함께 사용하는 것입니다.

왜 두 가지 DB가 모두 필요한가요?

1. Vector DB (유사성 검색의 강자):
    - 역할: 텍스트의 의미를 숫자로 변환(임베딩)하여 저장합니다.
    - 장점: 사용자의 질문과 의미적으로 유사한 문서를 아주 빠르고 폭넓게 찾아줍니다. (예: "매출"을 검색하면 "수익" 문서도 찾음)
    - 단점: "왜 이 문서가 나왔는지"에 대한 논리적 근거나, 정보 간의 복잡한 관계를 설명하지 못합니다.
2. Graph DB (관계 추적의 강자):
    - 역할: 데이터(개체)와 데이터 사이의 관계를 그물망처럼 저장합니다.
    - 장점: 정보가 연결된 맥락을 파악하고, 복잡한 추론이 가능합니다. 특히 금융이나 규제 산업에서 중요한 답변의 근거(출처 추적)를 명확히 제시할 수 있습니다.
    - 단점: 정확한 키워드나 관계가 없으면 검색 범위가 좁아질 수 있습니다.

결론: Vector + Graph 시너지 기업은 이 둘을 결합하여 '정확성'과 '설명 가능성'을 동시에 확보하려 합니다. 이를 통해 질문과 유사한 자료를 폭넓게 찾으면서도(Vector), 그 답이 도출된 논리적 경로를 투명하게 보여주는(Graph) 고도화된 RAG 시스템을 구축하고 있습니다.

1.1.3 연결되지 않은 파이프라인: OCR과 크롤링의 고립

마지막으로, 기업 데이터의 80%를 차지하는 비정형 데이터(문서, 이메일, 웹 정보 등)를 처리하는 과정에서도 단절이 발생합니다.

- 별도 운영되는 데이터 수집: 종이 문서를 디지털화하는 OCR(광학문자인식)이나 웹 정보를 긁어오는 크롤링(Crawling) 작업은 보통 RPA 팀이나 데이터 엔지니어링 팀에서 별도로 관리합니다.

- AI 시스템과의 단절: 문제는 이렇게 수집된 데이터가 실시간으로 AI 에이전트나 검색 엔진에 전달되지 않는다는 점입니다.
  - 예를 들어, 새로운 규정 문서를 스캔(OCR)했지만, 이 데이터가 사내 AI 챗봇의 지식 베이스에 업데이트되기까지 며칠이 걸리거나 수동으로 옮겨야 하는 식입니다.

결과적인 비효율 문서 스캔부터 지식화까지의 과정이 하나로 연결(End-to-End)되지 않아 정보의 지연이 발생합니다. 아무리 똑똑한 AI 모델을 도입하더라도, 최신 정보를 실시간으로 공급받지 못하는 파이프라인 구조에서는 낡은 정보를 기반으로 답변할 수밖에 없습니다.

요약하자면, 현재 기업의 AI 도입은 양적으로는 팽창했지만, 질적으로는 ① 부서별 도구의 분산, ② DB 전략의 복잡성 증가, ③ 데이터 수집 파이프라인의 단절이라는 과제에 직면해 있습니다. 이것이 바로 단순한 Chatbot 형태의 AI 도입을 넘어, 이 모든 요소를 통합하고 조율할 수 있는 지능형 업무를 개발 하기 위한 AI Agent Builder가 필요한 이유입니다.

1.2 기존 AI 적용 방식의 한계

위에서 살핀 바와 같이 기업들은 각자 필요한 AI 솔루션을 산발적으로 도입해왔지만, 이러한 기존 접근 방식에는 여러 한계가 드러나고 있습니다. PoC(개념 검증) 단계에서 머무르는 개발의 악순환, 질의응답 중심 정적 구조의 확장 어려움, 그리고 부서별 사일로화된 지식과 워크플로우의 문제점들이 대표적입니다. 이러한 한계들은 구조적 패턴으로 반복되고 있어, 단순한 기술 성능의 문제가 아니라 조직과 아키텍처의 문제로 바라봐야 합니다.

1.2.1 PoC 중심 개발의 반복과 운영 이관 실패 패턴

많은 기업들이 AI 프로젝트를 소규모 PoC로 시작하지만, 운영 환경으로 이관하지 못한 채 종료되는 패턴이 반복되고 있습니다. 연구에 따르면 기업 AI 프로젝트의 95%가 파일럿 단계에서 실패하고, 단 5%만이 실질적인 생산 환경 배치와 성과 달성으로 이어지는 것으로 나타났습니다. 이처럼 대부분의 프로젝트가 출발선에서 멈추는 이유는 기술 자체의 문제라기보다, PoC 결과물을 실제 업무 흐름에 통합하고 운영화하는 데 실패하기 때문입니다.

MIT 보고서에서도 “엔터프라이즈 AI 솔루션의 95%가 실패하는 근본 원인은 모델 품질이 아닌 잘못된 엔터프라이즈 통합”이라고 지적합니다. PoC 단계에서는 특정 부서나 소수 인원이 신속히 프로토타입을 만들지만, 이걸 운영팀에 넘겨 장기 유지보수하려면 보안, 성능, 모니터링 등 많은 추가 작업이 필요합니다. 그러한 운영 품질 격차를 메우지 못해 PoC만 반복하고 실서비스로 전환되지 못하는 악순환에 빠진 기업이 많습니다. 그 결과 기업 입장에서는 비슷한 PoC를 여러 번 시도하느라 리소스만 소모하고, 현업에 즉각 도움이 되는 AI 시스템은 부족한 상황이 발생합니다. 실제로 전세계적으로 수십조 원이 생성형 AI PoC에 투입되고도 정작 운영상 성과로 이어지지 않는 비효율이 지적되고 있으며, 이는 기획–개발–운영 간 단절이라는 구조적 문제로 인한 것입니다.

1.2.2 정적 RAG(질의응답) 중심 구조의 확장 한계

현재 많은 AI 활용이 정적 RAG 구조, 즉 질문에 대해 한 번 문서를 검색하고 답변을 생성하면 끝나는 형태에 머물러 있습니다. 이러한 일회성 질의응답 구조는 단순한 FAQ에는 유용하지만, 조금만 복잡한 요구사항이 나오면 한계에 봉착합니다. 기존의 RAG 챗봇은 사용자의 질문을 받으면 임베딩 유사도 검색으로 관련 문서를 한 번 조회하고 답변 생성 후 종료합니다. 추가 탐색이나 단계별 추론이 없기 때문에 처음 찾은 정보로 답이 안 되면 그대로 실패하게 됩니다. 예를 들어 질문이 모호하거나 다단계 추론이 필요한 경우, 정적 RAG로는 첫 시도에서 놓친 부분을 재검색하거나 답변을 보완할 방법이 없습니다. 이로 인해 복잡한 비즈니스 질문이나 연속된 대화 맥락을 요구하는 시나리오에서는 전통적 RAG의 유용성이 급격히 떨어집니다】. 또한 정적 구조에서는 모델이 사용자 피드백을 통해 학습하거나 맥락을 누적시키는 기능이 없어서, 매 세션이 초기화되고 과거 교훈이 반영되지 않습니다loris.ai. 결국 현재 많은 기업이 구축한 “문서 검색 + 답변 생성” 형태의 정적 Q&A 봇은 업무 프로세스 자동화나 복잡한 문제 해결로 확장되기 어렵습니다. 단발성 답변에 그치는 시스템으로는 사용자의 실제 업무를 대행하거나 능동적으로 작업을 수행하는 수준에 도달할 수 없는 것이죠. 이러한 한계를 극복하려면 질의응답 이상으로 유연하게 행동할 수 있는 새로운 접근이 필요합니다.

1.2.3 부서별 Prompt/Flow/지식베이스 사일로 발생 구조

엔터프라이즈 내 AI 도입이 부서 주도로 산발적으로 이뤄지다 보니, 프롬프트 템플릿, 대화 플로우, 지식베이스가 부서별 사일로(silo)로 쪼개지는 문제가 두드러집니다. 각 부서가 자기 업무에 필

요한 AI 챗봇이나 지식 검색 시스템을 따로 구축하면서, 콘텐츠와 노하우의 중복이 발생하고 교차 활용이 불가능한 섬들이 만들어집니다. 예컨대 인사부서는 인사 정책 FAQ 챗봇을 구축하고, 고객 지원부서는 제품 Q&A 봇을 별도로 구축하며, R&D 부서는 자체 기술문서용 GPT 서비스를 운용하는 식입니다. 이때 각 시스템은 서로 다른 프롬프트 엔지니어링 기법, 서로 다른 데이터 소스를 사용하고, 심지어 동일한 질문에 대해서도 부서별로 상이한 답변을 낼 수 있습니다. AI 에이전트들이 고립되어 상호 정보 공유를 못 하면 필연적으로 비효율과 중복 노력이 발생합니다. Confluent 등의 분석에 따르면, 기업 내 여러 AI 에이전트가 서로 단절되어 작동하면 중요한 정보나 인사이트가 공유되지 않아 비효율적이고, 동일한 문제를 여기저기서 반복 해결하는 상황이 벌어진다고 합니다 또한 부서마다 다른 툴과 워크플로우를 쓰면 통합된 거버넌스나 품질 관리가 어려워 위험요소가 높아집니다. 요컨대, 중앙 조율 없이 부서별로 AI를 운영한 결과 데이터도, 대화 흐름도, 생성된 지식도 사일로화되는 구조적 문제가 나타나고 있는 것입니다. 이는 기업 차원의 AI 자산 활용도와 일관성을 저해하여, AI 도입 효과를 국지적인 수준에 머물게 만듭니다.

1.3 AI Agent와 Agent Builder의 역할 정의

앞서 언급한 '파편화된 AI' 문제를 해결하기 위한 열쇠는 단순히 대화만 하는 챗봇을 넘어, 실제 업무를 수행하는 "행동하는 에이전트(Acting Agent)"로의 전환에 있습니다. 그리고 이러한 에이전트를 기업 환경에서 쉽고 빠르게 만들어주는 공장 역할을 하는 것이 바로 Agent Builder입니다.

최근 주목받는 Dify, LangFlow, Flowise, n8n과 같은 오픈소스 도구들은 코딩 지식이 부족해도 에이전트를 조립할 수 있는 Low-Code/No-Code 환경을 제공합니다. 이들은 흩어져 있던 AI 기술들을 하나로 묶어주는 오케스트레이션(지휘자) 레이어로서 작동합니다. 구체적으로 이 도구들이 어떻게 챗봇을 에이전트로 진화시키는지, 그리고 어떤 핵심 역할을 수행하는지 살펴보겠습니다.

1.3.1 말만 하는 "Chatbot" vs 일하는 "Agent"의 차이

우리가 흔히 접하는 챗봇과 지금 이야기하려는 AI 에이전트는 겉보기엔 비슷해 보이지만, 능력의 깊이가 다릅니다.

- 챗봇 (수동적 응답기):
  - 특징: 사용자의 질문에 대해 학습된 데이터나 연결된 문서를 바탕으로 답변만 합니다.
  - 한계: “내일 오후 2시 회의 잡아줘”라고 말하면, “일정을 잡는 기능은 없습니다”라고 대답하거나, 단순히 대화 흉내만 낼 뿐 실제로 캘린더 앱을 켜지는 못합니다.
  - 구조: LLM(두뇌) + 채팅창(입”만” 있음)
- AI 에이전트 (능동적 해결사):
  - 특징: 목표를 달성하기 위해 도구(Tool)를 사용하고, 스스로 계획(Plan)을 세워 연속적인 행동을 합니다.
  - 예시 (n8n 활용): “내일 오후 2시 회의 잡아줘”라고 하면, 에이전트는 ①구글 캘린더 API를 조회해 빈 시간을 확인하고, ②비어 있다면 일정을 등록한 뒤, ③슬랙(Slack)으로 팀원들에게 알림을 보냅니다.
  - 구조: LLM(두뇌) + Tool(손과 발) + Planning(계획 능력)

기술적 차이의 핵심: 다단계 실행(Multi-step Execution)Flowise나 LangFlow 같은 도구를 보면 이 차이가 명확합니다. 챗봇은 입력→출력의 단순 흐름이지만, 에이전트는 “생각의 사슬(Chain of Thought)”을 가집니다.”이 질문에 답하려면 검색이 필요한가? → 검색 도구 실행 → 결과 분석 → 답변 생성”과 같이 스스로 판단하고 움직이는 과정이 포함되어 있습니다. 즉, 챗봇이 ‘질문-답변 머신’이라면, 에이전트는 ‘문제-해결 머신’입니다.

1.3.2 Orchestration Layer로서 Agent Builder의 책임 범위

그렇다면 Dify나 LangFlow 같은 Agent Builder는 구체적으로 어떤 역할을 할까요? 이들은 에이전트의 두뇌(LLM)와 신체(도구), 그리고 기억(메모리)을 연결하고 관리하는 오케스트레이션(통합 관리) 계층을 담당합니다.

Agent Builder가 수행하는 5가지 핵심 역할은 다음과 같습니다.

1. LLM “두뇌” 갈아 끼우기 (Model Management):
   - 역할: 상황에 맞는 최적의 모델을 선택하고 관리합니다.

- 도구 예시: Dify에서는 드롭다운 메뉴 하나로 에이전트의 두뇌를 GPT-4에서 Claude 3.5나 로컬 모델(Llama 3)로 즉시 교체할 수 있습니다. 복잡한 코딩 없이 프롬프트만 수정하여 모델의 성격을 바꿉니다.

2. 기억력 부여하기 (Memory & Context):
   - 역할: 에이전트가 이전 대화 내용을 기억하거나, 방대한 사내 지식을 참고할 수 있게 합니다.
   - 도구 예시: Flowise의 'Memory' 노드를 연결하면, 에이전트는 사용자와의 지난 대화 맥락을 유지합니다. 또한 RAG(검색 증강 생성) 파이프라인을 구축하여 사내 문서를 벡터 DB에서 찾아 답변에 활용하도록 돕습니다.
3. 행동 계획 수립 (Planning & Workflow):
   - 역할: 복잡한 문제를 작은 단위로 쪼개고 순서대로 처리하는 로직을 짭니다.
   - 도구 예시: LangFlow나 n8n의 시각적 인터페이스(캔버스)를 통해 "사용자가 불만을 제기하면 → 감정 분석을 하고 → 심각하면 매니저에게 이메일을 보내라"는 식의 흐름도(Workflow)를 드래그 앤 드롭으로 그릴 수 있습니다.
4. 도구와 시스템 연결 (Tool Integration):
   - 역할: 에이전트에게 '손과 발'을 달아줍니다. 외부 API, DB, 사내 시스템과 연결합니다.
   - 도구 예시: n8n은 1,000개가 넘는 앱(Jira, Notion, Gmail 등)과 연결되는 노드를 제공합니다. Agent Builder는 LLM이 "메일을 보내야겠다"고 판단하면, n8n의 이메일 노드를 호출하여 실제 발송을 수행하는 중개자 역할을 합니다.
5. 감시와 통제 (Monitoring & Governance):
   - 역할: 에이전트가 엉뚱한 짓을 하지 않는지 감시하고, 로그를 남깁니다.
   - 도구 예시: Dify의 대시보드에서는 에이전트가 사용자와 어떤 대화를 나눴는지, 왜 그런 답변을 했는지(추론 과정), 토큰 비용은 얼마나 썼는지를 한눈에 보여줍니다. 이는 기업이 AI를 안전하게 운영하기 위한 필수 기능입니다.

결론적으로,Dify, LangFlow, Flowise, n8n과 같은 Agent Builder는 흩어져 있던 부품(LLM, DB, API)들을 표준화된 방식으로 조립해주는 플랫폼입니다. 이 계층이 없다면 개발자들이 일일이 코드를 짜서 연결해야 하지만, Agent Builder를 도입함으로써 기업은 복잡한 코딩 없이도 고성능의 '행동하는 AI 에이전트'를 대규모로 양산하고 관리할 수 있게 됩니다.

1.3.3 No-Code/Low-Code가 엔터프라이즈에 필요한 이유 (자산화·협업·표준화)

엔터프라이즈에서 Agent Builder를 활용할 때 Low-Code/No-Code 방식을 채택하는 것이 각광받는 이유는, AI 솔루션의 "자산화·협업·표준화"를 촉진하기 때문입니다. Low-Code/No-Code 플랫폼은 그래픽 인터페이스나 간단한 설정만으로 에이전트 플로우를 구축할 수 있게 해주므로, 전문 개발자뿐 아니라 현업 담당자도 개발 과정에 참여할 수 있습니다. 이는 곧 업무 지식을 가진 비개발자와 IT 부서 간 협업을 강화하여 AI 적용 범위를 넓히고 현업 요구사항을 신속히 반영하는 데 도움이 됩니다.

또한 No-Code 방식으로 구성된 에이전트 흐름은 일반 코드보다 시각적으로 명확하고 자체 문서화(self-documenting)된 성격이 있습니다. 한 IT 관리자는 "로우코드 통합은 거의 자체 문서화가 되어 있어, 새로운 담당자에게 인계하더라도 흐름을 빠르게 파악할 수 있다"고 평가합니다. 실제로 Low-Code 플랫폼에서는 모든 통합 과정이 표준화된 블록과 연결선으로 표현되기 때문에, 개별 개발자의 코드 스타일에 의존하지 않고도 일관된 이해가 가능합니다. 이는 조직 내 AI 에이전트 개발 산출물이 공용 자산이 되어 재사용과 유지보수가 용이해진다는 뜻입니다. 예컨대 한 기업은 과거에 부서별로 커스텀 C# 코딩으로 통합 작업을 했으나, 지금은 통합 플랫폼을 통해 모든 흐름이 일관된 형태로 구축되고 재사용 가능한 라이브러리로 축적되어 있다고 보고합니다. "모든 것이 플랫폼을 거치면서 기술 스택, DB 연결, 프로세스 흐름 등이 표준화되었고, 결과적으로 일관되고 재사용 가능한 통합 자산 라이브러리가 생겼다"는 현장의 평가처럼, No-Code 기반 Agent Builder를 도입하면 AI 적용 지식과 구성요소들이 기업의 자산으로 축적됩니다.

마지막으로 Low-Code/No-Code 플랫폼은 거버넌스와 협업 표준화 측면에서도 이점이 있습니다. 모든 팀이 동일한 플랫폼에서 에이전트를 개발하면, 접근 권한 관리, 버전 관리, 변경 이력 추적 등이 중앙에서 이루어져 통제와 규정 준수가 쉬워집니다. 코드로 산재되어 있을 때는 리뷰나 표준 준수가 어려웠던 부분도, 플랫폼 상에서는 정해진 모듈과 설정만 사용하므로 자연스럽게 표준화된 베스트프랙틱스 준수가 가능합니다. 또한 플랫폼 상에서 시각적으로 프로세스를 검

증하고 잘못된 부분을 발견하기 쉬워 품질 향상이 용이합니다. 요약하면, 엔터프라이즈 환경에서는 Low-Code Agent Builder를 통해 AI 에이전트 개발을 민주화하고, 산출물을 조직 자산으로 축적하며, 부서 간 협업과 표준화를 촉진하는 것이 지속가능한 AI 도입을 위한 열쇠라고 볼 수 있습니다.

결론: Agent Builder 계층 부재 시 발생하는 구조적 병목

지금까지 살펴본 바와 같이, Agent Builder라는 오케스트레이션 계층의 도입 여부가 기업 AI 활용의 성공을 가르는 결정적 요인이 되고 있습니다. 만약 이 계층이 부재한 상태로 각종 LLM 활용을 이어갈 경우, 기업은 다음과 같은 구조적 병목에 직면합니다. 첫째, 확장이 불가능한 구조입니다. 부서별로 산발적으로 구축된 정적 챗봇이나 RAG 솔루션들은 서로 연계되지 않기에 새로운 업무 지능화를 전사적으로 확대 적용하기 어렵습니다. 한 부서에서 만든 솔루션을 다른 부서에 접목하려 해도 기술 스택과 데이터 연동 방식이 제각각이라 재개발이 필요하고, 복잡한 시나리오 (예: 여러 부서 데이터 종합 분석)는 애초에 구현되지 못합니다. 둘째, AI 자산의 파편화입니다. Agent Builder가 없으면 프롬프트, 지식베이스, 통합 커넥터 등이 중복 개발되고 부서별 섬으로 남아 있게 됩니다. 이는 비용 낭비일 뿐 아니라, 각 솔루션의 성능 편차와 비일관성을 낳아 사용자 경험에도 악영향을 줍니다. 셋째, 재현 불가능성과 거버넌스 부재 문제가 있습니다. 에이전트 빌더 없이 개별 팀이 AI 솔루션을 만들면 로그 일관성, 프로세스 투명성 확보가 어려워 동일한 질문에 대한 결과 재현이나 오류 원인 추적이 힘들어집니다. 예를 들어, 벡터 검색 기반 답변 시스템을 부서마다 다르게 구현하면, 어떤 임베딩 모델과 파이프라인을 썼느냐에 따라 답이 달라지지만 이를 중앙에서 재현하거나 검증하기가 어렵습니다. 이처럼 표준화된 오케스트레이션 계층이 없을 때의 혼란은 단순한 비효율이 아니라 업무 지능화의 구조적인 한계점으로 작용하여, 기업 전반의 AI 도입 ROI를 떨어뜨리고 리스크를 높이는 병목이 됩니다.

반대로, Agent Builder 계층을 도입하면 이러한 병목이 해소되고 업무 지능화가 가속화될 수 있습니다. 앞서 언급한 대로 많은 선도 기업들은 이미 정적 Q&A를 넘어 에이전트 기반 아키텍처로 전환 중이며, 생산 환경에 에이전트를 도입한 기업이 절반을 넘어섰고 88%가 긍정적 ROI를 보고하고 있습니다. 이는 에이전트 중심으로 AI 전략을 재편할 때 얻을 수 있는 효율성과 가치를 방증합니다. 결론적으로, Agent Builder가 없는 환경에서는 기업의 AI 활용이 PoC의 늪, 사일로의 벽, 확장의 한계에 갇혀버리지만, Agent Builder라는 계층을 제대로 구축하면 기업 내 산재

된 AI 요소들을 하나로 묶어 지속적 학습과 협업이 가능한 지능형 워크플로우로 승화시킬 수 있습니다. 업무 지능화를 조직 전반으로 확장하려는 의사결정권자라면, “왜 Agent Builder가 필요한가?”라는 질문에 대한 답은 곧 “우리의 AI 도입을 조직의 자산이 되는 구조로 만들기 위해서”임을 깨닫게 될 것입니다.

2장. 엔터프라이즈 구축 환경 정의: 폐쇄망·보안·플랫폼 전제

본 장은 솔루션의 기능을 비교하기에 앞서, 공공기관 및 금융·대기업 환경에서 AI 시스템을 구축할 때 반드시 충족해야 하는 물리적·논리적 제약 사항을 정의합니다.

시중의 많은 AI Agent Builder(n8n, LangFlow 등)는 인터넷이 연결된 클라우드(SaaS) 환경을 가정하고 설계되었습니다. 그러나 우리가 구축하려는 엔터프라이즈 환경은 외부와 단절된 폐쇄망(Air-gapped)이며, 엄격한 보안 규정을 따르는 COP(Container Orchestration Platform) 위에서 동작해야 합니다.

이 장에서 정의하는 요건을 충족하지 못하는 솔루션은 아무리 기능이 화려해도 도입 자체가 불가능합니다. 따라서 본 장은 향후 5장과 6장의 기술 평가를 수행하기 위한 ‘Cut-off(부적합 배제)’ 기준점이 됩니다.

2.1 폐쇄망(Air-gapped) 환경 요구사항

폐쇄망 환경의 가장 큰 특징은 ‘인터넷 연결이 없다’는 점입니다. 이는 단순히 Wi-Fi가 안 터진다는 수준이 아니라, 개발자가 습관적으로 사용하는 외부 저장소(GitHub, PyPI, npm, Docker Hub)로의 접근이 원천 차단됨을 의미합니다.

2.1.1 인터넷 비연결 환경에서의 설치·업데이트·패키지 관리 이슈

일반적인 오픈소스 AI 도구들은 설치 시 인터넷을 통해 의존성 패키지를 실시간으로 다운로드합니다(예: `pip install langchain`). 하지만 폐쇄망에서는 이 과정이 불가능하므로 다음과 같은 요건이 필수적입니다.

- Offline Installer 및 Full Image 제공: 모든 의존성 라이브러리(Python 패키지, Node.js 모듈, 시스템 바이너리)가 사전에 포함된 단일 컨테이너 이미지(Docker Image) 혹은 오프라인 설치 패키지(Tarball) 형태로 반입이 가능해야 합니다.
- 빌드 타임 의존성 해결: 런타임(실행 중)에 추가적인 패키지를 다운로드하려 시도하는 솔루션은 운영 중 장애를 유발합니다. 예를 들어, 특정 노드를 실행할 때 백그라운드에서 모델 가중치(Model Weights)를 다운로드하는 구조라면, 해당 모델 파일도 미리 로컬 경로에 마운트(Mount) 할 수 있는 옵션을 제공해야 합니다.
- 버전 관리의 경직성 극복: 폐쇄망에서는 '업데이트'가 곧 '재설치'와 같습니다. 잦은 패치보다는 안정적인 LTS(Long Term Support) 버전이 제공되는지, 혹은 Docker 이미지 태그 관리가 명확한지가 중요한 선정 기준이 됩니다.

2.1.2 외부 API 의존성 제거(모델/툴/플러그인)

많은 Agent Builder들이 기본 기능으로 "Google Search", "Slack Webhook", "Zapier Integration" 등을 내장하고 있습니다. 그러나 내부망에서는 이러한 외부 SaaS 연동 기능이 무용지물이거나 보안 위규 사항이 됩니다.

- Local LLM/SLM 호환성: OpenAI(GPT-4)나 Anthropic(Claude) API 호출을 전제로만 동작하는 솔루션은 제외해야 합니다. 내부망에 구축된 vLLM, Ollama, TGI 등의 로컬 모델 추론 서버와 표준 프로토콜(OpenAI-compatible Endpoint)로 통신할 수 있어야 합니다.
- SaaS 의존형 기능의 격리: 솔루션이 제공하는 플러그인 중 외부 인터넷 통신이 필수인 기능(예: 외부 날씨 API, 공용 환율 정보 등)을 비활성화하거나, 이를 내부 API로 대체(Mocking/Proxy) 할 수 있는 유연성을 갖춰야 합니다.
- 자체 인증 메커니즘: 솔루션 로그인이나 라이선스 검증을 위해 개발사 서버와 통신해야 한다면 도입이 불가능합니다. 완전한 On-Premise Standalone 구동이 보장되어야 합니다.

2.1.3 보안·망분리·감사(로그) 관점의 필수 통제 요건

기업의 보안 담당자가 AI 도입을 주저하는 가장 큰 이유는 '데이터 유출'과 '설명 불가능성(Black Box)' 때문입니다. 이를 해소하기 위한 통제 요건은 다음과 같습니다.

- 망간 자료 전송 통제 준수: Agent가 수집한 데이터나 생성한 결과물이 허가받지 않은 외부 채널로 전송되지 않도록, Egress(아웃바운드 트래픽) 제어가 가능한 구조여야 합니다.
- 데이터 마스킹 및 PII(개인정보) 필터링: LLM으로 데이터를 보내기 전, 주민번호나 전화번호 같은 민감 정보를 식별하여 마스킹(Masking) 할 수 있는 전처리 파이프라인(Pre-processing Pipeline)이 워크플로우 내에 포함되어야 합니다.
- Full Audit Logging (감사 로그): 단순한 시스템 접속 로그(System Log)를 넘어, "누가(Who), 언제(When), 어떤 프롬프트(Prompt)를 입력해서, 어떤 데이터(Context)를 참조해, 무엇을 답변(Answer)했는가"에 대한 완전한 이력을 별도 DB나 로그 서버에 적재할 수 있어야 합니다. 이는 향후 환각(Hallucination) 현상 분석이나 보안 감사 시 필수적입니다.

2.2 타겟 플랫폼 전제: Kubernetes 기반 운영

우리가 구축할 AI Agent 시스템의 무대는 단순한 가상머신(VM) 한두 대가 아닙니다. 수많은 컨테이너를 효율적으로 관리하고 조율하는 Kubernetes(K8s) 기반의 환경을 기본 전제로 합니다.

Dify나 LangFlow 같은 최신 도구들은 이미 Docker 컨테이너 형태의 배포를 표준으로 지원하고 있습니다. 이를 Kubernetes 위에서 운영함으로써, 기업은 유동적인 AI 워크로드를 안정적으로 감당할 수 있는 엔터프라이즈급 인프라를 갖추게 됩니다.

2.2.1 Kubernetes 기반 배포를 기본으로 보는 이유 (표준화·격리·확장)

AI Agent는 일반적인 웹 서비스와 달리 자원 사용량이 롤러코스터처럼 널뜁니다. 평소에는 조용하다가도, 수백 페이지 문서를 요약하거나 복잡한 추론을 시작하면 CPU와 메모리 사용량이 치솟습니다. Kubernetes는 이러한 특성을 제어하기에 최적입니다.

- 오토스케일링 (HPA – 자동 확장): 갑자기 전사 직원이 동시에 Dify 챗봇에 접속하거나 대량의 문서 파싱 요청이 들어오면 어떻게 될까요? Kubernetes는 이를 감지하고 자동으로 Pod(실행 단위)의 개수를 늘려 트래픽을 분산 처리합니다. 요청이 줄어들면 다시 Pod를 줄여 비용을 아낍니다.
- 리소스 격리 (Quota & Limit – 이웃 간섭 방지): "인사팀이 돌리는 무거운 엑셀 분석 에이전트 때문에 재무팀 봇이 멈췄다"는 상황을 막아야 합니다. Kubernetes의 네임스페이스(Namespace) 기능을 활용하면, 부서별로 사용할 수 있는 CPU/RAM 총량을 제한(Quota)하여 특정 부서의 과부하가 전체 시스템에 영향을 주지 않도록 격리할 수 있습니다.
- 표준화된 배포 (Manifest):n8n을 쓰든 Flowise를 쓰든, 운영팀 입장에서는 관리 방식이 동일해야 합니다. 모든 솔루션의 배포 설정이 Kubernetes의 표준 명세서(Deployment, Service, Ingress Manifest)로 정의되므로, 도구가 바뀌어도 운영 프로세스는 일관되게 유지됩니다.

2.2.2 이미지 레지스트리/Helm/내부 CI를 통한 배포 흐름

보안이 중요한 기업(폐쇄망) 환경에서는 인터넷에서 `docker pull` 명령어로 이미지를 바로 내려받을 수 없습니다. 따라서 "검증된 아티팩트(Artifact)의 이관"이라는 체계적인 절차가 필요합니다.

- Private Registry (사내 이미지 저장소) 활용:LangFlow나 Dify의 최신 버전이 나오면, 먼저 보안 스캔을 거친 후 사내 이미지 레지스트리(예: Harbor, Nexus)에 안전하게 저장(Push)합니다. 우리가 선정할 솔루션은 설정 파일에서 이미지 주소만 내부 레지스트리로 바꿔주면 즉시 구동될 수 있는 구조여야 합니다.
- Helm Chart 지원 (설정의 템플릿화): AI 도구는 단독으로 돌지 않습니다. DB(PostgreSQL), 벡터 DB(Qdrant), 캐시(Redis) 등이 한 세트로 묶여야 합니다. Helm Chart는 이 복잡한 세트 설정을 하나의 패키지로 묶어 관리하게 해줍니다. Dify와 같은 도구는 공식 Helm Chart를 제공하므로, 복잡한 설치 과정을 명령어 한 줄로 단순화할 수 있어 운영 편의성이 매우 높습니다.
- 내부 CI/CD 파이프라인 연동 (자동화): 개발자가 Flowise에서 만든 워크플로우 파일(JSON)이나 설정 변경 사항을 Git에 올리면, Jenkins나 GitLab CI가 이를 감지하여 자동

으로 사내 서버에 배포합니다. 이를 위해 솔루션의 모든 설정은 텍스트 코드로 관리 가능한 “Configuration as Code” 형태를 지원해야 합니다.

2.2.3 내부 인증·권한 체계(SSO/LDAP)와의 결합 지점

수천 명의 임직원이 사용하는 시스템에서, 별도의 ID/PW를 또 만들게 해서는 안 됩니다. 기존 사내 계정 시스템과 물 흐르듯 연결되어야 합니다.

- SSO (Single Sign-On – 통합 로그인) 연동: 사용자는 평소 쓰던 사내 그룹웨어 계정으로 Dify나 n8n에 로그인해야 합니다. 이를 위해 솔루션은 OIDC, SAML 2.0, LDAP/AD 같은 표준 인증 프로토콜을 반드시 지원해야 합니다. (오픈소스 버전에서는 이 기능이 제한적일 수 있으므로 엔터프라이즈 라이선스 검토 시 핵심 체크사항이 됩니다.)
- RBAC (역할 기반 접근 제어 – 권한 쪼개기): 모든 사람이 에이전트를 수정할 수 있으면 사고가 납니다.
  - Admin: 시스템 전체 설정 변경 가능
  - Builder (개발자): LangFlow 캔버스에서 로직 수정 가능
  - User (일반 직원): 만들어진 챗봇과 대화만 가능 이처럼 사용자 역할에 따라 실행(Execute), 수정(Edit), 조회(View) 권한을 세밀하게 제어할 수 있는 기능이 필수적입니다. 특히 Dify의 경우 워크스페이스 개념을 통해 팀별로 권한을 격리하는 기능을 잘 제공하고 있습니다.

2.3 핵심 연동 컴포넌트 범위 정의: 에이전트 빌더는 '지휘자'다

Agent Builder는 모든 일을 혼자 처리하는 만능 도구가 아닙니다. 오히려 외부의 전문화된 시스템들과 끊임없이 소통하며 작업을 지시하고 결과를 받아오는 오케스트레이터(Orchestrator, 지휘자)에 가깝습니다.

성공적인 에이전트 시스템을 구축하기 위해 Agent Builder는 다음 세 가지 핵심 컴포넌트(데이터 수집기, 검색 엔진, 데이터베이스)와 유기적으로 데이터를 주고받아야 합니다. Dify나 n8n과 같은 도구들이 이들과 어떻게 역할을 분담하고 협력하는지 살펴보겠습니다.

2.3.1 무거운 짐 덜어내기: 크롤러(Crawler4AI)·OCR·ETL과의 역할 분리

에이전트가 똑똑해지려면 데이터가 필요하지만, 데이터를 수집하고 가공하는 과정은 매우 무겁습니다.

- 책임의 분리 (Why): 만약 Flowise나 Dify 내부에서 직접 수천 페이지의 웹사이트를 크롤링하거나, 고해상도 PDF 수만 장을 OCR(문자인식) 하려고 한다면 어떻게 될까요? 시스템 전체가 느려지거나 멈춰버릴 것입니다. 따라서 Agent Builder는 '지휘'만 하고, 실제 '노동'은 전문 도구에 맡겨야 합니다.
- 연동 구조 (How): "비동기 처리(Asynchronous Processing)"가 핵심입니다.
    1. 지시(Trigger): n8n과 같은 워크플로우 도구가 Crawler4AI나 사내 OCR 서버에 "이 URL을 긁어와" 또는 "이 파일을 텍스트로 바꿔줘"라고 API 요청을 보냅니다.
    2. 대기 및 수신(Webhook): Agent Builder는 그동안 다른 작업을 수행합니다. 전문 도구가 작업을 마치고 "다 됐습니다"라고 신호(Webhook)를 보내거나 DB에 데이터를 쌓아두면, 그때 결과를 가져와 활용합니다. 즉, Builder는 작업 지시서만 발송하고, 실제 데이터 파싱은 외부의 전용 파이프라인이 담당하는 구조입니다.

2.3.2 똑똑하게 찾아내기: 검색(BM25)·임베딩·리랭킹의 3중 주

기업 데이터 검색은 구글 검색보다 까다롭습니다. 단순히 의미가 비슷한 것을 찾는 것만으로는 부족하며, 정확한 키워드나 최신성을 따져야 할 때가 많습니다. Dify와 LangFlow는 이를 위해 정교한 검색 파이프라인을 지원합니다.

- 하이브리드 검색 (Hybrid Search): "2024년 1월 규정"을 찾을 때, 의미 기반(Vector) 검색은 "규정"이라는 맥락은 잘 찾지만 "2024년 1월"이라는 숫자를 놓칠 수 있습니다. 반면 키워드 검색(BM25)은 숫자는 잘 찾지만 맥락을 모릅니다. Dify의 지식베이스 설정에서는 이 두 가지를 동시에 수행하여 장점만 취하는 하이브리드 검색을 옵션으로 제공합니다.
- 리랭킹 (Reranking): 검색된 100개의 문서 중 진짜 정답이 50번째에 있다면 LLM이 이를 놓칠 수 있습니다. 이를 해결하기 위해 LangFlow나 Flowise에서는 Rerank 노드를 워크플로우 중간에 배치할 수 있습니다. 1차로 검색된 문서들을 정밀한 AI 모델(Cross-Encoder)

이 다시 꼼꼼히 읽어보고, 가장 정확한 순서대로 재정렬(Reranking)하여 LLM에게 전달합니다.

2.3.3 두 개의 기억 저장소: VectorDB와 GraphDB의 전략적 투입

에이전트가 "문맥"과 "관계"를 모두 파악하기 위해서는 두 종류의 데이터베이스를 도구(Tool)로서 자유자재로 다룰 수 있어야 합니다.

- VectorDB (예: Qdrant) – 문맥 담당: 비정형 텍스트(매뉴얼, 회의록, 이메일)를 저장합니다.
  - 활용: 사용자가 "지난달 프로젝트 이슈가 뭐였지?"라고 물으면, Flowise의 Vector Store 노드는 Qdrant를 뒤져 의미적으로 유사한 텍스트 조각을 찾아냅니다.
- GraphDB (예: Neo4j) – 관계 담당: 데이터 간의 연결 고리(조직도, 시스템 의존성, 인물 관계)를 저장합니다.
  - 활용: 사용자가 "A 프로젝트 담당자가 관리하는 다른 시스템이 멈추면 어디에 영향이 가지?"라고 묻는다면, 단순 텍스트 검색으론 답하기 어렵습니다. 이때 에이전트는 LangChain의 Graph Chain을 통해 Neo4j에 쿼리를 날려, 인물과 시스템 간의 연결 관계를 추적하여 정확한 답을 냅니다.
- 통합 전략 (Tool Use): 중요한 것은 Agent Builder가 이 두 DB를 단순 저장소가 아닌 '선택 가능한 도구'로 인식한다는 점입니다. Dify나 LangFlow에서 에이전트를 설정할 때, 질문의 성격에 따라 "이건 텍스트 검색(Vector)이 필요해", "이건 관계 추적(Graph)이 필요해"라고 스스로 판단하고 적절한 DB를 호출하여 답변의 근거(Grounding)를 확보하는 패턴을 구현해야 합니다.

3장. AI Agent Builder 아키텍처: 실행 모델·흐름 제어·컨텍스트

AI Agent Builder는 단순한 LLM 호출 도구를 넘어, 복잡한 비즈니스 로직을 자동화하고 지능화하는 엔터프라이즈 플랫폼의 핵심 구성 요소입니다. 본 장에서는 Agent Builder의 근간을 이루는

실행 모델, LLM 오케스트레이션 구조, 그리고 컨텍스트 관리 및 메모리 활용 방안을 상세히 기술하여, “Agent = LLM 호출”이라는 통념을 명확히 차단하고 Agent Builder의 진정한 가치를 조명합니다.

3.1 실행 모델(Execution Model): 에이전트를 움직이는 엔진

n8n, Dify, LangFlow, Flowise와 같은 Agent Builder는 단순히 코드를 실행하는 것이 아니라, 다양한 상황에 맞춰 에이전트를 깨우고, 작업을 관리하며, 부하를 분산시키는 유연한 실행 엔진을 탑재하고 있습니다.

이들은 공통적으로 ①외부 신호에 반응(Trigger)하고, ②작업 과정을 안전하게 관리(Runtime)하며, ③무거운 작업을 효율적으로 처리(Worker Pattern)하는 구조를 갖추고 있습니다. 각 도구가 이러한 실행 모델을 어떻게 구현하고 있는지 공통적인 메커니즘을 중심으로 살펴보겠습니다.

3.1.1 Trigger 기반 실행: 에이전트를 깨우는 세 가지 신호

모든 Agent Builder는 잠들어 있는 에이전트 워크플로우를 시작시키기 위한 ‘시동 버튼’, 즉 트리거(Trigger)를 가지고 있습니다. 이 도구들은 공통적으로 다음 세 가지 방식의 실행을 지원합니다.

- 웹훅 기반 실행 (Webhook – “외부 시스템의 호출”):
  - 공통 기능: Flowise나 LangFlow, Dify 등 모든 도구는 워크플로우를 생성하면 고유한 API URL(Endpoint)을 발급해줍니다.
  - 작동 방식: 외부의 웹사이트, 슬랙(Slack) 봇, 혹은 사내 레거시 시스템이 이 URL로 데이터를 보내면(POST 요청), 에이전트가 즉시 깨어나 해당 데이터를 받아 처리를 시작합니다. 이는 가장 보편적이고 표준화된 연동 방식입니다.
- 스케줄 기반 실행 (Scheduled – “정해진 시간의 반복”):
  - 공통 기능: 사람이 직접 실행하지 않아도, 주기적인 업무를 처리할 수 있도록 Cron(시간 예약) 기능을 지원합니다.

  - 작동 방식: n8n의 Cron 노드나 Dify의 스케줄 트리거를 통해 “매일 아침 9시 뉴스 요약”, “매주 금요일 주간 데이터 동기화”와 같은 배치성 작업을 자동화합니다.

- 이벤트 기반 실행 (Event-driven – “특정 사건 발생 시”):
  - 공통 기능: 시스템 내외부의 특정 변화를 감지하여 실행됩니다.
  - 작동 방식: 특히 n8n은 “이메일 수신 시”, “구글 시트 행 추가 시”와 같은 앱 통합 트리거에 강점이 있으며, Dify나 Flowise 역시 채팅창의 “메시지 입력 시작”과 같은 사용자 이벤트를 감지하여 동작합니다.

3.1.2 워크플로우 런타임: 실행 상태 관리와 안전장치

에이전트가 실행되는 동안(Runtime), 이 도구들은 작업이 엉뚱한 길로 빠지지 않도록 감시하고 오류를 제어하는 ‘신호등’ 역할을 수행합니다.

- 상태 시각화 및 추적 (Observability):
  - 공통 기능: 에이전트가 현재 어떤 단계(노드)를 처리 중인지 시각적으로 보여주고 로그를 남깁니다.
  - 작동 방식: LangFlow나 Flowise의 UI, 혹은 Dify의 ‘로그(Logs)’ 탭에서는 데이터가 흘러가는 경로를 실시간으로 확인할 수 있습니다. 입력값이 무엇이었고, LLM이 어떤 생각을 했으며(Reasoning), 최종 출력값이 무엇인지 투명하게 추적하여 디버깅을 돕습니다.

- 재시도 메커니즘 (Retry Policy):
  - 공통 기능: 외부 API 호출이나 LLM 응답이 일시적으로 실패했을 때, 전체 작업을 중단하지 않고 다시 시도하는 기능을 제공합니다.
  - 작동 방식: LLM API 타임아웃이나 DB 연결 오류 발생 시, “1초 뒤 재시도, 최대 3회 반복”과 같은 설정을 통해 시스템의 회복 탄력성(Resilience)을 보장합니다.

- 타임아웃 설정 (Timeout):
  - 공통 기능: 특정 작업이 무한정 대기하거나 리소스를 독점하는 것을 방지합니다.
  - 작동 방식: 워크플로우 전체 혹은 개별 노드에 실행 제한 시간을 설정하여, 예기치 않게 작업이 멈추거나 지연될 경우 강제로 종료시키고 리소스를 회수합니다.

3.1.3 작업 분리(Worker)와 확장 구조: “접수처”와 “주방”의 분리

엔터프라이즈 환경에서는 수백, 수천 건의 요청이 동시에 들어올 수 있습니다. 이를 처리하기 위해 이 도구들은 요청을 받는 곳(API Server)과 실제로 일을 하는 곳(Worker)을 분리하는 아키텍처를 공통적으로 채택합니다.

- 동기/비동기 처리의 조화 (Sync/Async):
  - 동기(Sync): 챗봇 대화처럼 즉각적인 응답이 필요한 작업은 API 서버에서 빠르게 처리합니다.
  - 비동기(Async): 대량의 PDF 문서 분석이나 긴 리포트 작성처럼 시간이 걸리는 작업은 백그라운드에서 처리하도록 넘깁니다.
- 워커 패턴과 큐(Queue) 시스템:
  - 공통 구조: Dify, Flowise, n8n 모두 무거운 작업을 처리하기 위해 내부적으로 Redis와 같은 메시지 큐(Queue)를 사용합니다.
  - 작동 방식 (식당 비유):
    1. API 서버(웨이터): 사용자의 요청(주문)을 받아 큐(주문서)에 넣습니다.
    2. 워커 프로세스(요리사): 대기 중인 워커들이 큐에서 작업을 하나씩 가져가 수행합니다.
- 확장성 (Scalability):
  - 이 구조의 장점은 우리가 앞서 정의한 Kubernetes 환경에서 극대화됩니다. 작업량이 폭증하면, ’주문 받는 서버’는 그대로 두고 힘든 일을 하는 ’워커 컨테이너’의 개수만 자동으로 늘려서(Auto-scaling) 시스템 성능을 유연하게 유지할 수 있습니다.

3.2 LLM 오케스트레이션 구조

Agent Builder는 LLM을 단순한 텍스트 생성기가 아닌, 특정 기능을 수행하는 도구(Tool)로 활용하고 복잡한 의사결정 과정을 지원하는 오케스트레이션 엔진 역할을 수행합니다.

3.2.1 단일 호출 vs Multi-step Agent 실행(Plan-Act-Observe)

- 단일 호출 (Single-step Agent): 가장 기본적인 형태로, 사용자 또는 시스템의 입력에 대해 LLM에게 단 한 번의 프롬프트를 전달하고 응답을 받는 방식입니다. 이는 간단한 질의응답, 텍스트 요약, 번역 등 즉각적인 결과가 필요한 작업에 적합합니다.
  - 예시: “오늘 날씨 알려줘” -> LLM이 날씨 정보 응답
- Multi-step Agent 실행 (Plan-Act-Observe - ReAct 패턴): 복잡한 문제를 해결하기 위해 여러 단계의 추론과 행동을 순차적으로 수행하는 Agent입니다. 이는 ReAct(Reasoning and Acting) 프레임워크와 유사하게 작동합니다.
  - Plan (계획): Agent는 주어진 목표를 달성하기 위한 일련의 단계를 계획합니다.
  - Act (행동): 계획된 단계에 따라 특정 도구(Tool)를 사용하거나 LLM에 추가적인 질문을 합니다. 이는 Function Calling/Tool Calling을 통해 외부 API를 호출하거나, 내부 함수를 실행하는 것을 포함합니다.
  - Observe (관찰): 행동의 결과를 관찰하고, 이 정보를 바탕으로 다음 단계를 결정하거나 계획을 수정합니다.
  - 반복: 목표가 달성될 때까지 Plan-Act-Observe 과정을 반복합니다.
  - 장점: 복잡한 문제 해결, 여러 정보원을 통합, 동적인 의사결정, 외부 도구 활용 능력 등을 갖추게 됩니다. LangChain의 Agent Executor나 Dify의 Agent 기능이 이 모델을 기반으로 합니다.

이러한 Multi-step Agent 실행은 Agent Builder가 단순히 텍스트를 생성하는 것을 넘어, 실제 업무 환경에서 발생하는 복잡한 과제를 해결하는 “행동하는 Agent”로서의 역할을 수행하게 합니다.

3.2.2 Function Calling과 Tool Calling: LLM에게 손과 발을 달아주는 기술

Agent Builder의 가장 중요한 역할은 “말만 하는 뇌(LLM)”와 “실제 행동하는 손(외부 시스템)”을 연결해 주는 것입니다. 이를 가능하게 하는 기술이 바로 Function Calling과 Tool Calling입니다.

n8n, Dify, LangFlow, Flowise는 이 기술들을 활용해, LLM이 대화 도중에 “지금은 계산기가 필요해”, “지금은 사내 DB 검색이 필요해”라고 판단하고 실제 도구를 호출할 수 있도록 지원합니다. 두 방식의 기술적 차이와 이들 도구에서의 구현 방식을 살펴보겠습니다.

1) Function Calling (OpenAI 스타일의 정교한 호출)

이 방식은 주로 OpenAI(GPT 시리즈) 모델이 “함수 호출을 위해 특화된 훈련”을 받았다는 점을 활용합니다.

- 특징: LLM이 모호한 텍스트가 아니라, 기계가 바로 실행할 수 있는 완벽한 JSON 포맷으로 도구 사용 요청을 보냅니다.
- 도구별 구현 (Dify, n8n 등):
  - 설정: Dify나 n8n에서 GPT-4 모델을 선택하고 도구를 연결하면, 시스템은 자동으로 이 방식을 우선 사용합니다.
  - 작동 방식:
    1. 사용자가 “서울 날씨 어때?”라고 묻습니다.
    2. GPT 모델은 내부적으로 `get_weather(city="Seoul")`라는 함수가 필요하다고 판단하고, `{ "name": "get_weather", "args": { "city": "Seoul" } }`라는 깔끔한 JSON 데이터를 생성해 반환합니다.
    3. Dify/n8n은 이 JSON을 받아 즉시 해당 API를 실행합니다.
  - 장점: 매우 정확하고, 복잡한 인자(Argument)도 잘 처리하여 에러율이 낮습니다.

2) Tool Calling (일반화된 도구 사용, LangChain 스타일)

Function Calling을 포함하는 더 넓은 개념으로, OpenAI뿐만 아니라 모든 LLM(Llama 3, Claude, Mistral 등)이 도구를 사용할 수 있도록 프레임워크 차원에서 추상화한 방식입니다. LangFlow와 Flowise는 주로 LangChain의 `Tool` 개념을 기반으로 이 방식을 구현합니다.

- 특징: 모델에게 “너에게는 이런 도구들이 있어”라고 설명서(Prompt)를 주면, 모델이 “그럼 A 도구를 쓸게”라고 텍스트로 답하는 방식입니다. (ReAct 패턴 등 활용)
- 도구별 구현 (LangFlow, Flowise):
  - 설정: 캔버스 화면에서 [Agent 노드]에 [Calculator 노드]나 [**Custom Tool 노드]**를 선으로 연결합니다.

- 작동 방식:
  1. Flowise는 LLM에게 프롬프트 앞단에 “사용 가능한 도구 목록: [계산기, 구글 검색]”을 몰래 포함시켜 보냅니다.
  2. 사용자가 질문하면 LLM은 “검색 도구를 사용해야 함. 검색어: ‘2024년 금리’ ”와 같은 형식의 텍스트를 출력합니다.
  3. Flowise의 파서(Parser)가 이 텍스트를 해석하여 실제 도구를 실행합니다.
- 장점: 특정 모델(OpenAI)에 종속되지 않고, 다양한 오픈소스 모델이나 커스텀 도구를 유연하게 연결할 수 있습니다.

3) Agent Builder에서의 통합 실행 루프 (The Execution Loop)

Dify, LangFlow, Flowise, n8n 모두 내부적으로는 아래와 같은 ’생각-행동 루프’를 통해 이 두 가지 방식을 통합하여 지원합니다. 사용자는 복잡한 코드를 몰라도 이 흐름을 자동화할 수 있습니다.

1. 도구 등록 (Registration): 사용자가 n8n의 화면에서 ‘Gmail 보내기’, ‘Slack 알림’, ‘SQL 쿼리’ 등의 노드를 추가하고 에이전트에 연결합니다. Builder는 이 도구들의 사용법(이름, 필요한 정보)을 LLM에게 알려줍니다.
2. 판단 및 호출 (Thinking & Calling): LLM이 사용자 질문을 분석한 후, Function Calling(JSON) 또는 Tool Calling(텍스트 추론) 방식을 통해 “이 도구를, 이 값으로 실행해줘”라고 Builder에게 요청합니다.
3. 실행 및 결과 반환 (Execution):Agent Builder가 중간에서 이 요청을 가로채 실제 API를 호출하거나 DB를 조회합니다. 그 결과(예: “검색 결과: 25도”)를 다시 LLM에게 “도구 실행 결과야”라며 돌려줍니다.
4. 최종 답변 (Final Response): LLM은 도구가 가져온 정보를 바탕으로 사용자에게 자연스러운 최종 답변을 생성합니다.

결론적으로, 우리가 사용할 Agent Builder는 OpenAI의 정교한 Function Calling을 지원함과 동시에, 다양한 모델과 커스텀 기능을 수용할 수 있는 범용적인 Tool Calling 구조를 모두 갖추고 있어, LLM의 지능을 실제 비즈니스 업무(API 호출, 데이터 조회 등)로 확장하는 데 필수적인 역할을 수행합니다.

3.2.3 프롬프트/정책/가드레일의 자산화와 버전 관리

Agent Builder는 LLM의 행동을 제어하고 일관성을 유지하기 위해 프롬프트, 정책, 가드레일을 체계적으로 관리하고 버전 관리합니다.

- 프롬프트 (Prompts): LLM에게 전달되는 지시사항입니다.
  - 자산화: 다양한 목적(예: 요약, 분류, 질의응답)에 따른 프롬프트 템플릿을 생성하여 재사용성을 높입니다. 이러한 템플릿에는 동적으로 채워질 변수(예: 사용자 입력, 문서 내용)가 포함됩니다.
  - 버전 관리: 프롬프트의 성능 개선, 새로운 요구사항 반영 등을 위해 프롬프트의 변경 이력을 관리합니다. 이는 특정 버전의 프롬프트로 Agent를 재현하거나, A/B 테스트를 수행하는 데 중요합니다.
- 정책 (Policies): Agent가 수행해야 할 작업의 우선순위, 제한사항, 또는 특정 상황에서의 행동 지침 등을 정의합니다. 예를 들어, “민감한 정보를 처리할 때는 항상 암호화 절차를 거친다”와 같은 정책이 있습니다.
  - 자산화: 재사용 가능한 정책 모듈로 관리하여 여러 Agent에 적용할 수 있습니다.
  - 버전 관리: 정책 변경사항을 추적하고, 이전 버전으로 롤백할 수 있도록 합니다.
- 가드레일 (Guardrails): Agent의 행동이 안전하고 윤리적이며, 기업의 규정을 준수하도록 보장하는 안전 장치입니다. 여기에는 유해 콘텐츠 필터링, 개인정보 노출 방지, 부정확한 정보 생성 방지 등이 포함됩니다.
  - 자산화: 다양한 유형의 가드레일 모듈을 생성하고, Agent 빌더 내에서 쉽게 구성할 수 있도록 합니다.
  - 버전 관리: 가드레일의 효과를 지속적으로 모니터링하고 업데이트하여 최신 보안 및 규제 요구사항을 반영합니다.

이러한 자산화 및 버전 관리 메커니즘은 Agent Builder가 기업 환경에서 일관되고 안전하며 예측 가능한 방식으로 작동하도록 보장합니다. Dify는 이러한 프롬프트와 Agent 설정에 대한 명확한 관리 기능을 제공합니다.

3.3 Context 관리와 Memory

Agent Builder는 LLM이 대화의 맥락을 이해하고, 이전 상호작용을 기억하며, 필요한 정보를 효과적으로 활용할 수 있도록 정교한 컨텍스트 관리 및 메모리 시스템을 구축합니다.

3.3.1 세션 컨텍스트·업무 컨텍스트·지식 컨텍스트 구분

Agent Builder는 복잡한 Agent 실행에서 발생하는 다양한 종류의 컨텍스트를 구분하여 관리합니다.

- 세션 컨텍스트 (Session Context): 특정 대화 또는 상호작용 세션 동안 유지되는 정보입니다. 이는 사용자의 현재 요청, 대화 기록(최근 몇 턴), 현재 Agent의 상태 등을 포함합니다. 사용자 경험의 연속성을 보장하는 데 중요합니다.
- 업무 컨텍스트 (Task Context): Agent가 현재 수행 중인 특정 업무 또는 과제와 관련된 정보입니다. 이는 과제의 목표, 관련 데이터, 현재까지 수행된 단계, 성공/실패 여부, 필요한 도구 등을 포함합니다. Agent가 주어진 업무를 효율적으로 완수하도록 안내하는 역할을 합니다.
- 지식 컨텍스트 (Knowledge Context): Agent가 의사결정이나 응답 생성에 활용할 수 있는 외부 지식 정보입니다. 이는 기업 내부의 문서, 데이터베이스, 외부 API 응답, 또는 웹 검색 결과 등을 포함합니다. RAG(Retrieval-Augmented Generation) 시스템에서 검색된 정보가 여기에 해당됩니다.

이러한 컨텍스트 구분을 통해 Agent Builder는 특정 상황에 가장 적합한 정보를 활용하고, 불필요한 정보로 인해 LLM의 성능이 저하되는 것을 방지할 수 있습니다.

3.3.2 단기 메모리와 장기 메모리(지식베이스)의 결합

Agent Builder는 Agent의 정보 활용 능력을 극대화하기 위해 단기 메모리와 장기 메모리를 결합합니다.

- 단기 메모리 (Short-term Memory): 세션 컨텍스트와 유사하게, 단기 메모리는 Agent의 현재 대화 또는 작업 흐름 내에서 빠르게 접근하고 활용할 수 있는 정보를 저장합니다. 이는

LLM의 컨텍스트 창 크기 제한을 극복하기 위해 최근 대화 내용을 요약하거나, 특정 단계의 결과를 저장하는 데 사용됩니다. LangChain의 `ConversationBufferMemory`와 같은 메커니즘이 여기에 해당합니다.

- 장기 메모리 (Long-term Memory - Knowledge Base): Agent가 지속적으로 학습하고 참조할 수 있는 영구적인 정보 저장소입니다. 이는 기업의 내부 지식베이스, 과거 작업 기록, 사용자 프로필, 도구 사용 패턴 등을 포함할 수 있습니다. Vector Database (Qdrant, Weaviate 등)나 Graph Database (Neo4j)와 같은 기술을 활용하여 이 지식베이스를 구축하고, 필요에 따라 Agent가 검색하여 활용합니다.
  - 결합 구조: Agent Builder는 단기 메모리의 최신 정보와 장기 메모리의 축적된 지식을 결합하여, 보다 풍부하고 맥락에 맞는 응답을 생성하거나 복잡한 의사결정을 내립니다. 예를 들어, 단기 메모리의 현재 질문과 장기 메모리의 관련 과거 사례를 참조하여 응답을 생성하는 식입니다.

이러한 결합 구조는 Agent가 단순한 정보 전달자를 넘어, 학습하고 기억하며 맥락에 맞는 지능적인 행동을 수행하는 "지능형 Agent"로 발전할 수 있게 합니다.

3.3.3 감사 가능한 컨텍스트 로그(입력/출력/근거) 설계

엔터프라이즈 환경에서 Agent Builder는 모든 실행 과정을 투명하게 기록하고 추적할 수 있는 감사 기능을 제공해야 합니다.

- 입력 (Input): Agent가 받은 모든 외부 입력(사용자 요청, 이벤트 데이터, API 호출 파라미터 등)을 기록합니다.
- 출력 (Output): Agent가 생성한 모든 출력(LLM 응답, 함수 호출 결과, API 응답 등)을 기록합니다.
- 근거 (Reasoning/Evidence): Agent가 특정 결정을 내리거나 응답을 생성하게 된 근거를 상세히 기록합니다. 이는 어떤 프롬프트가 사용되었는지, 어떤 도구가 호출되었는지, 검색된 지식(RAG의 경우 검색된 문서 조각)은 무엇인지, 어떤 중간 단계를 거쳤는지 등을 포함합니다.

이러한 감사 가능한 컨텍스트 로그는 다음과 같은 목적으로 활용됩니다.

- 디버깅 및 문제 해결: Agent 실행 중 발생하는 오류의 원인을 신속하게 파악하고 수정하는 데 도움을 줍니다.
- 규정 준수 및 감사: 금융, 의료 등 규제가 엄격한 산업에서 Agent의 행동을 감사하고 규정을 준수했음을 입증하는 데 필수적입니다.
- 성능 분석 및 개선: Agent의 의사결정 과정을 분석하여 성능 병목 지점을 식별하고 개선 기회를 찾습니다.
- 재현성 확보: 동일한 입력에 대해 동일한 출력이 발생하도록 Agent의 실행을 재현할 수 있게 합니다.

Flowise나 Dify와 같은 도구는 시각적인 인터페이스를 통해 이러한 실행 이력과 근거를 추적할 수 있는 기능을 제공하며, 이는 엔터프라이즈 수준의 Agent Builder에 있어 매우 중요한 요구사항입니다. LangChain의 LangSmith와 같은 별도의 LLMOps 플랫폼은 이러한 로깅 및 추적 기능을 더욱 강화합니다.

이 장은 Agent Builder가 단순히 LLM API를 감싸는 래퍼(wrapper)가 아님을 명확히 보여주는 핵심적인 부분입니다. “Agent = LLM 호출”이라는 오해를 끊는 데 집중하십시오. Agent Builder가 제공하는 런타임의 복잡성, LLM과의 상호작용 방식, 그리고 정보를 기억하고 활용하는 메커니즘을 구체적인 기술 용어와 함께 설명함으로써, 독자들이 Agent Builder의 진정한 아키텍처적 가치를 이해하도록 유도해야 합니다.

각 섹션에서 설명된 내용들은 솔루션(n8n, LangFlow, Flowise, Dify)들이 실제로 제공하거나 제공해야 할 핵심 기능임을 암시하며, 이러한 기능들을 기반으로 각 솔루션을 평가할 근거를 제시합니다. 예를 들어, 3.1.3의 워커 패턴과 확장성은 Kubernetes 기반의 폐쇄망 환경에서 솔루션의 배포 및 운영 용이성을 평가하는 중요한 기준이 될 것입니다. 3.2.1의 Multi-step Agent 실행 능력은 솔루션이 복잡한 업무 자동화를 얼마나 효과적으로 지원할 수 있는지를 나타내며, 3.3.3의 감사 기능은 엔터프라이즈 보안 및 규정 준수 요구사항 충족 여부를 판단하는 기준이 됩니다.

그림이나 흐름도를 활용하여 각 개념(예: Plan-Act-Observe 사이클, 워커 패턴, 컨텍스트 흐름)을 시각적으로 설명한다면 독자의 이해도를 크게 높일 수 있습니다. 특히, 3.3.3의 감사 가능한 컨텍스트 로그 설계는 백서의 신뢰성을 높이는 데 기여할 것입니다.

4장. 업무 적용을 위한 연동 설계: 크롤링·검색·지식베이스 파이프라인

엔터프라이즈 AI 에이전트 구축의 핵심은 LLM 모델 자체보다, 파편화된 사내 데이터를 어떻게 에이전트의 컨텍스트(Context)로 주입하느냐에 달려 있습니다. 특히 폐쇄망(Air-gapped) 환경에서의 Agent Builder는 외부 API에 의존할 수 없으므로, 내부망에 배치된 수집기, 검색 엔진, 데이터베이스를 오케스트레이션(Orchestration)하는 'Control Plane' 역할을 수행해야 합니다.

본 장에서는 Agent Builder(n8n, LangFlow, Flowise, Dify 등)가 데이터를 직접 처리하는 것이 아니라, 전문 컴포넌트(Crawler, Search Engine, DB)와 입력/출력 계약(Contract)을 맺고 파이프라인을 제어하는 아키텍처를 상세히 기술합니다.

4.1 데이터 수집 및 크롤링 계층

Agent Builder는 대량의 데이터 처리에 최적화된 엔진이 아니므로, 무거운 수집 작업(Heavy Lifting)은 전용 서비스에 위임하고 그 결과(Payload)만을 워크플로우로 반입해야 합니다.

4.1.1 Crawler4AI 연동 방식(배치/실시간, API/컨테이너 호출)

Crawler4AI는 LLM 친화적인 포맷(Markdown/JSON)으로 웹 콘텐츠를 변환하는 데 특화된 오픈소스입니다. 폐쇄망 환경에서 Agent Builder는 Crawler4AI를 다음과 같은 방식으로 제어합니다.

1. 실행 모델: 컨테이너 기반 API 호출 (REST via Docker Network)
    - Agent Builder(예: n8n의 HTTP Request Node, LangFlow의 Custom API Tool)는 내부망에 배포된 Crawler4AI 컨테이너의 엔드포인트(`/crawl`)를 호출합니다.
    - 직접적인 Python 코드 실행보다는, Crawler4AI를 마이크로서비스로 띄우고 REST API로 통신하는 것이 의존성 격리와 확장성 측면에서 유리합니다.

- 요청 파라미터 제어: Agent Builder는 URL 뿐만 아니라 `css_selector`, `word_count_threshold` 등의 파라미터를 동적으로 주입하여 수집 범위를 조정합니다.

2. 배치(Batch) vs. 실시간(Real-time) 트리거 전략

- 정기 배치 (Knowledge Base Update): 스케줄러 트리거(Cron)를 사용하여 야간에 사내 기술 블로그나 공지사항을 일괄 수집합니다. 이때 Agent Builder는 '루프(Loop)' 노드를 통해 다수의 URL을 순차적으로 크롤러 서비스에 던지는 역할을 합니다.
- On-Demand 실시간 수집: 사용자가 "A 경쟁사 최신 뉴스 요약해줘"라고 질의할 때, Agent는 검색 도구(Serper Dev 등 대안 내부 검색)를 통해 URL을 확보한 뒤, 즉시 Crawler4AI를 호출하여 본문을 긁어옵니다.

4.1.2 내부 시스템(그룹웨어/문서함/포털) 연결 패턴

사내 레거시 시스템은 표준화된 API가 없는 경우가 많습니다. Agent Builder는 이를 추상화된 '커스텀 툴(Custom Tool)' 형태로 래핑하여 접근해야 합니다.

- Database Polling (CDC 유사 패턴):
  - 그룹웨어 게시판 DB(MySQL/Oracle)를 직접 조회하는 방식입니다. Agent Builder의 SQL 노드를 사용하여 `created_at > last_check_time` 조건으로 신규 게시물을 주기적으로 조회합니다.
  - n8n은 이러한 Polling Trigger 구현에 강점이 있으며, 변경 감지 시 자동으로 임베딩 파이프라인을 시작하도록 구성할 수 있습니다.
- Legacy Protocol Adapters:
  - SMB(파일 서버)나 SharePoint 구버전 등 Agent Builder가 네이티브로 지원하지 않는 프로토콜의 경우, 중간에 Python 기반의 어댑터 서비스(MCP Server 등)를 두는 패턴을 권장합니다.
  - Agent Builder는 표준화된 JSON 요청을 어댑터에 보내고, 어댑터가 레거시 프로토콜로 데이터를 가져와 텍스트만 반환합니다.

4.1.3 수집 데이터 정규화·메타데이터 표준(출처/시간/권한)

LLM이 할루시네이션(환각) 없이 답변하려면 데이터의 '맥락'이 필수적입니다. Agent Builder 내에서 데이터 변환(Transform) 노드를 통해 다음과 같은 표준 스키마를 강제해야 합니다.

필드명	설명	Agent Builder 처리 로직
content	LLM 학습용 본문	HTML 태그 제거, Markdown 변환, 불필요한 공백/특수문자 정제
source_url	근거 제시용 링크	사용자가 답변의 진위를 확인할 수 있도록 원문 링크 유지
timestamp	정보 최신성 판단	데이터 생성일 및 수집일을 기록하여, LLM에게 "최신 정보 우선" 지시 시 활용
permissions	접근 제어(ACL)	가장 중요. 문서 조회 가능한 `group_id`나 `department_code`를 리스트 형태로 메타데이터에 삽입

4.2 검색 및 질의 처리 계층

단순 Vector 검색(Semantic Search)만으로는 기업의 복잡한 질의(특정 모델명, 사번 등)를 처리하기 어렵습니다. Agent Builder는 Hybrid Search 파이프라인을 구성하여 정확도를 보완해야 합니다.

4.2.1 BM25 기반 키워드 검색의 역할(정확도/회수율)

- 필요성: 벡터 검색은 "서버"와 "컴퓨터"의 의미적 유사성은 잘 찾지만, "Error-503"이나 "프로젝트 코드명(Project Zeus)"과 같은 Exact Match(정확한 일치)에는 취약합니다.
- 구현: Elasticsearch나 Qdrant(최신 버전 지원)의 Full-text search 기능을 활용합니다.
- Builder 역할: 사용자의 질문에서 키워드 추출(Keyword Extraction) 에이전트를 선행 실행하여, 핵심 단어를 발라낸 후 BM25 검색 엔진에 질의를 던집니다.

4.2.2 Vector 검색과의 병행(Hybrid Query)

Agent Builder는 두 가지 검색 결과를 병합하는 RRF(Reciprocal Rank Fusion) 알고리즘을 워크플로우 상에서 구현해야 합니다.

1. Parallel Execution (병렬 실행):
    - 분기 1: 질의 임베딩 생성 → VectorDB 검색 (의미적 유사 문서 상위 10개)
    - 분기 2: 키워드 추출 → BM25 검색 (키워드 일치 문서 상위 10개)
2. Merge & Score:
    - 두 결과 집합을 합치고, 중복을 제거한 뒤 순위를 재산정합니다.
    - LangFlow/Flowise 등의 경우 이러한 로직이 캡슐화된 'Retriever' 컴포넌트를 제공하기도 하지만, 세밀한 제어를 위해 n8n 등에서 Function Node로 직접 병합 로직을 구현하는 것이 엔터프라이즈 요구사항 반영에 유리합니다.

4.2.3 리랭킹/필터링(권한/문서유형/기간) 파이프라인

검색된 문서가 LLM에 들어가기 전, 보안 필터링과 정교화 과정이 반드시 필요합니다.

- Pre-Retrieval Filtering (권한 통제):
    - 사용자의 요청 헤더(Header)나 세션 정보에서 `user_dept: 'HR'` 정보를 추출합니다.
    - Agent Builder는 VectorDB 쿼리 시 `filter: { department: 'HR' }` 조건을 동적으로 삽입하여, 애초에 권한 없는 문서는 검색되지 않도록 차단합니다. (보안 사고 방지 핵심)
- Post-Retrieval Reranking (재순위화):
    - 검색된 20~30개의 후보군(Candidates)을 Cross-Encoder 모델(BGE-Reranker 등)에 통과시켜 질문과의 관련성을 정밀 채점합니다.
    - Agent Builder는 이 Reranking 모델을 API로 호출하여, 점수가 낮은 문서를 잘라내고(Cut-off) 상위 3~5개만 LLM 프롬프트에 주입(Injection)합니다.

4.3 VectorDB 및 GraphDB 연계

단편적인 지식은 VectorDB로, 복잡한 관계(조직도, 시스템 의존성)는 GraphDB로 해결합니다. Agent Builder는 이 두 DB를 상황에 맞게 교차 조회합니다.

4.3.1 VectorDB(Quadrant/Qdrant) 연동 포인트(인덱싱/검색)

- Qdrant 선정 이유: Go/Rust 기반의 단일 바이너리 배포가 가능해 폐쇄망 컨테이너 환경에 적합하며, Payload Filtering 성능이 우수합니다.
- Upsert(저장) 파이프라인:
  - `Split Text(Chunking)` → `Embedding(Local LLM)` → `Qdrant Upsert`
  - Agent Builder는 위 과정을 순차적으로 수행하며, 실패 시 재시도(Retry) 로직을 관리합니다.
- Retrieval(검색) 파이프라인:
  - 단순 조회가 아니라, 앞서 언급한 메타데이터 필터(Metadata Filter)를 JSON 객체로 구성하여 Qdrant API에 전달하는 것이 핵심입니다.

4.3.2 Neo4j 기반 관계 질의(정책/조직/시스템 관계) 투입 방식

문서의 내용을 넘어 “A 시스템 장애 시 영향을 받는 부서는 어디인가?” 와 같은 질문은 벡터 검색으로 해결할 수 없습니다.

- Graph Schema Mapping: 노드(Node: 시스템, 부서, 사람)와 엣지(Edge: 관리한다, 의존한다, 소속된다)로 정의된 데이터를 조회합니다.
- Text-to-Cypher:
  - Agent Builder는 LLM에게 Graph Schema를 프롬프트로 제공하고, 자연어 질문을 Neo4j 쿼리 언어인 Cypher Query로 변환하도록 요청합니다.
  - 생성된 Cypher 쿼리를 Neo4j 드라이버(Bolt 프로토콜)를 통해 실행하고, 결과(관계 정보)를 텍스트로 변환하여 컨텍스트에 추가합니다.

4.3.3 Agent 워크플로우에서 Graph+Vector를 결합하는 구성 예시 (GraphRAG)

최종적으로 Agent Builder는 Vector와 Graph를 결합한 GraphRAG(Graph-Augmented Generation) 워크플로우를 구성하여 답변의 깊이를 더합니다.

[구성 시나리오: "김철수 책임이 작성한 '보안 규정' 문서의 핵심 내용과 관련 부서 알려줘"]

1. Entity Extraction (Agent): 질문에서 "김철수(Person)", "보안 규정(Topic)" 엔티티 추출.
2. GraphDB Query (Neo4j):
    - "김철수" 노드 조회 → `[:WROTE]` 관계 추적 → 문서 ID 확보.
    - 해당 문서와 `[:AFFECTS]` 관계로 연결된 "부서" 노드 목록 확보.
    - 결과: "문서 ID: DOC-101, 관련 부서: 인사팀, IT보안팀"
3. VectorDB Search (Qdrant):
    - GraphDB에서 얻은 `DOC-101`을 필터 조건(`filter: { doc_id: 'DOC-101' }`)으로 사용하여 벡터 검색 실행.
    - 문서의 실제 내용(Content) 요약 추출.
4. Context Assembly & Generation:
    - Graph의 구조적 정보(관련 부서) + Vector의 비정형 정보(문서 내용)를 프롬프트에 결합.
    - LLM 답변: "김철수 책임이 작성한 문서는 ~내용이며(Vector), 이는 인사팀과 IT보안팀(Graph)에 적용됩니다."

이처럼 Agent Builder는 단순한 "질문-답변" 봇을 넘어, 서로 다른 특성을 가진 데이터 저장소를 논리적으로 연결하고 제어하는 통합 허브(Hub) 역할을 수행해야 합니다.

5장. 후보 솔루션 기술 분석: n8n, LangFlow, Flowise, Dify

본 장에서는 엔터프라이즈 폐쇄망(Air-gapped) 환경 내 AI Agent 구축을 위한 후보 솔루션 4종(n8n, LangFlow, Flowise, Dify)을 심층 분석한다. 단순한 기능 비교를 지양하고, 아키텍처 성격(Architecture), 운영 환경 적합성(Operation), 확장성(Extensibility), 제품화 가능성(Productization) 관점에서 각 솔루션이 기업의 레거시 시스템 및 보안 요건과 어떻게 부합하는지 기술한다.

5.1 n8n: 워크플로우 자동화의 강자, AI 오케스트레이션의 조력자

n8n(nodemation)은 2019년 독일 베를린의 개발자 얀 오버하우저(Jan Oberhauser)가 창립한 회사이자 동명의 솔루션입니다.

- 시작과 배경(Why): 창립자는 Zapier나 Make(구 Integromat)와 같은 기존의 자동화 도구들이 비쌀 뿐만 아니라, 데이터가 외부 클라우드를 거쳐야 하는 보안 문제, 그리고 커스터마이징이 제한적인 '블랙박스' 형태라는 점에 문제의식을 느꼈습니다. 이에 "개발자가 코드를 섞어 쓸 수 있는, 내 서버에 설치 가능한(Self-hosted) 투명한 자동화 도구"를 목표로 n8n을 개발했습니다.
- 현재 위상: 처음에는 단순한 업무 자동화(IPA) 도구였으나, 2023년부터 LangChain 노드를 정식 통합하면서 AI Agent를 기존 레거시 시스템과 연결하는 강력한 미들웨어로 주목받고 있습니다.

5.1.1 범용 워크플로우 자동화 관점의 강점

n8n의 DNA는 AI가 아닌 '연동(Integration)'에 있습니다. 이는 기업 내 AI 도입 실패의 주원인인 "AI가 답은 잘하는데, 실제 업무 시스템(ERP, DB, 메신저)을 건드리지 못하는 문제"를 해결하는 데 결정적인 역할을 합니다.

- 시각적 접착제(Visual Glue) 아키텍처:
  - n8n은 노드(Node)와 선(Line)으로 데이터를 주고받는 흐름을 그리는 도구입니다. 개발자가 아니어도 직관적으로 이해할 수 있는 UI를 제공합니다.
  - 특히, AI 모델(LLM)이 내놓은 답변을 파싱(Parsing)하여 JSON 형태로 변환하고, 이를 사내 데이터베이스나 API로 쏘아주는 전처리·후처리 과정을 구현하는 데 있어 타의 추종을 불허합니다.
- 압도적인 연결성(Connectivity):
  - 이미 1,000개 이상의 사전 정의된 연동 모듈(Nodes)을 보유하고 있습니다.
  - 예를 들어, "고객의 이메일이 들어오면(Trigger) → AI가 내용을 요약하고(Process) → 사내 Slack 채널에 알림을 보낸 뒤(Action 1) → Jira에 티켓을 생성(Action 2)"하는 복잡한 업무를 코드 한 줄 없이 구현할 수 있습니다. 이는 AI 전용 빌더들이 갖지 못한 n8n만의 강력한 무기입니다.

5.1.2 AI Agent 구현 시 제약(에이전트 런타임/체인 모델 부재)

하지만 n8n을 본격적인 '생성형 AI 에이전트 빌더'로 메인으로 사용하기에는 태생적인 한계가 명확합니다. n8n은 '정해진 길'을 가는 것에는 능하지만, AI처럼 '상황에 따라 길을 바꾸는' 유연함은 부족합니다.

- 단방향(DAG) 구조의 한계:
  - 대부분의 AI Agent(ReAct 패턴 등)는 "생각 → 행동 → 관찰 → 다시 생각"이라는 루프(Loop, 순환) 구조를 가집니다. AI가 스스로 판단하여 만족스러운 답이 나올 때까지 작업을 반복해야 하기 때문입니다.
  - n8n은 기본적으로 시작에서 끝으로 흐르는 선형적 구조입니다. 루프를 구현할 수는 있지만, 'Go to' 노드를 억지로 연결해야 하거나 복잡한 IF 분기문을 수동으로 설계해야 합니다. 이로 인해 워크플로우가 매우 지저분해지고(Spaghetti Workflow), 유지보수가 어려워집니다.
- 메모리(Context) 관리의 부재:

- ChatGPT처럼 사용자의 이전 대화를 기억하려면 '상태(State)'를 저장해야 합니다.
- Dify나 LangFlow 같은 전용 도구는 이를 자동으로 관리해주지만, n8n은 기본적으로 한 번 실행되고 끝나는(Stateless) 구조입니다. 대화 맥락을 유지하려면 개발자가 직접 Redis나 DB에 대화 내용을 저장하고 불러오는 로직을 처음부터 끝까지 직접 구현해야 합니다.

5.1.3 라이선스·번들링 관점 리스크 포인트(상용 포함 시 주의)

기업이 n8n을 도입하거나 자사 제품에 포함(Bundling)하여 납품하려 할 때 가장 주의 깊게 검토해야 할 부분은 바로 라이선스입니다. n8n은 흔히 아는 오픈소스(MIT, Apache 2.0)와는 성격이 다릅니다.

- 라이선스 유형: Sustainable Use License
  - n8n은 소스 코드가 공개되어 있지만, OSI(Open Source Initiative)가 정의한 표준 오픈소스 라이선스는 아닙니다. 이들은 자체적인 'Sustainable Use License'와 'Fair-code' 정책을 따릅니다.
  - 출처 및 원문 확인: n8n 공식 문서 – 라이선스 정책 및 GitHub 저장소의 `LICENSE.md` 파일.
- 상용화 시 치명적인 제약 사항 (Commons Clause):
  - 내부 사용(Internal Use): 기업이 사내 업무 효율화를 위해 내부 서버에 설치하여 직원들만 사용하는 것은 무료이며 자유롭습니다.
  - 상용 번들링 불가: 만약 귀사가 개발하는 솔루션(예: "AI 업무 포털") 내부에 n8n을 탑재하여 고객사에게 돈을 받고 판매하거나, n8n을 활용한 SaaS 서비스를 제공하여 수익을 창출하는 경우, 이는 라이선스 위반입니다.
  - 이 라이선스는 "n8n과 경쟁하는 형태의 상용 서비스(SaaS 포함)를 만들지 말라"는 것을 핵심으로 합니다.
- 결론:
  - SI 프로젝트로서 고객사 서버에 단순 설치만 해주는 것은 가능할 수 있으나, 귀사의 패키지 솔루션의 일부로 n8n을 번들링(OEM)하여 납품하려면 n8n 본사와 별도의 상용

라이선스 계약(Enterprise License)을 체결해야 합니다. 비용 이슈가 발생할 수 있으므로 설계 단계에서 반드시 법무적 검토가 필요합니다.

5.2 LangFlow: 개발자를 위한 LangChain의 시각적 쌍둥이

LangFlow는 현재 LLM 애플리케이션 개발의 표준 프레임워크로 자리 잡은 LangChain(Python 버전)을 웹 화면으로 그대로 옮겨 놓은 도구입니다.

- 시작과 배경(Who & Why): LangFlow는 브라질의 소프트웨어 기업 Logspace가 2022년 말~2023년 초에 공개했습니다. (참고로, 2024년 4월, 글로벌 DB 기업인 DataStax가 Logspace 팀을 인수하며 현재는 DataStax 주도 하에 개발되고 있습니다.)
  - 개발 동기: LangChain은 기능이 강력하지만, 코드로만 작성할 경우 데이터의 흐름이 어떻게 이어지는지 파악하기 어렵고 디버깅이 난해했습니다. 개발자들은 “복잡한 파이썬 코드를 칠판에 그리듯이 연결하고, 즉시 테스트해볼 수 있는 UI”를 원했고, 이를 해결하기 위해 시작과 했습니다.
- 핵심 정체성: “Low-code for Developers”. 비개발자가 아닌, 파이썬과 AI 개념을 이해하는 엔지니어가 코딩 시간을 줄이기 위해 사용하는 도구에 가깝습니다.

5.2.1 LangChain 기반 시각적 빌더 구조

LangFlow는 LangChain 라이브러리의 내부 클래스(Chains, Prompts, LLMs, Agents)를 1:1로 매핑하여 GUI 컴포넌트로 제공합니다. 즉, 화면에 보이는 노드 하나하나가 실제 파이썬 코드의 객체와 동일합니다.

- 아키텍처 성격: ‘체인(Chain)’ 중심의 하드코어 제어
  - 다른 도구들이 복잡함을 숨기려 노력한다면, LangFlow는 복잡함을 그대로 드러내는 전략을 취합니다.

  - LLM의 `Temperature`(창의성 조절), `Top-P`, `Context Window` 사이즈 등 코드 레벨에서만 건드릴 수 있는 미세한 파라미터를 화면에서 직접 수정할 수 있습니다. 이는 정교한 튜닝이 필요한 엔지니어에게는 축복이지만, 일반 기획자에게는 거대한 진입 장벽이 됩니다.

- 실행 추적과 러닝 커브:
  - 입력 데이터가 프롬프트를 거쳐 모델로 들어가고, 다시 파서(Parser)를 거쳐 나오는 과정이 전선 회로도처럼 연결됩니다.
  - 단점: 시각적으로 매우 복잡합니다. 노드 간의 연결 선이 꼬이기 시작하면(Spaghetti Code), 비전문가는 전체 흐름을 전혀 파악할 수 없게 됩니다.

5.2.2 컴포넌트/커스텀 노드 확장 방식

LangFlow의 가장 큰 무기는 기반 언어가 AI 생태계의 표준인 Python이라는 점입니다.

- 확장성(Extensibility): Python 생태계의 흡수
  - LangFlow 내에는 사용자가 직접 파이썬 코드를 작성할 수 있는 'Custom Component' 기능이 있습니다.
  - 단순한 스크립트 실행을 넘어, 사내 데이터 분석팀이 이미 사용 중인 Pandas(데이터 처리), NumPy(수치 계산), Scikit-learn(머신러닝) 라이브러리를 `import`하여 즉시 노드로 변환할 수 있습니다. 이는 데이터 과학자 친화적인 환경을 제공합니다.

- Kubernetes 시 고려사항 (이질적인 런타임):
  - LangFlow의 백엔드는 파이썬의 FastAPI로 작성되어 있습니다.
  - 만약 귀사의 주력 시스템이 Java(Spring Boot) 기반이라면, LangFlow를 도입하기 위해 파이썬 런타임 환경을 별도로 구축하고 관리해야 합니다. 이는 운영 조직에게 '두 가지 언어를 모두 관리해야 하는' 부담을 줍니다.

5.2.3 폐쇄망 배포 및 운영 관점 적합성

인터넷이 차단된 폐쇄망(Air-gapped) 환경에서 LangFlow 운영은 '의존성 지옥(Dependency Hell)'과의 싸움이 될 가능성이 높습니다.

- 패키지 관리의 어려움:
  - AI 라이브러리(LangChain, OpenAI SDK 등)는 지금도 매주 업데이트됩니다. LangFlow는 이 라이브러리들에 강하게 의존하고 있습니다.
  - 인터넷이 되는 환경에서는 `pip install` 명령어 하나로 해결되지만, 폐쇄망에서는 수백 개의 의존성 패키지 파일(.whl)을 수동으로 반입하거나, 사내 전용 PyPI 미러 서버를 구축해야 합니다.
  - 버전이 조금만 안 맞아도 실행이 안 되는 호환성 충돌이 빈번하게 발생합니다.
- 안정성 이슈 (Prototyping vs Production):
  - LangFlow는 아직 '빠르게 만들어보는(Prototyping)' 도구로서의 성격이 강합니다.
  - 엔터프라이즈 환경에서 필수적인 고가용성(HA, 이중화), 대용량 트래픽 처리, 세션 클러스터링 기능은 상대적으로 약합니다. 상용 서비스 레벨로 올리기 위해서는 아키텍처적으로 많은 보완(예: 별도의 큐 관리, 로드 밸런싱)이 필요합니다.

5.2.4 라이선스 및 상업적 사용 가능성

LangFlow는 기업이 도입하거나 상용 솔루션에 포함시키기에 라이선스적으로 매우 안전한 편입니다.

- 라이선스 유형: MIT License
- 출처 확인: LangFlow GitHub Repository – LICENSE
- 상업적 사용 분석:
  - MIT 라이선스는 오픈소스 라이선스 중 가장 제약이 적은(Permissive) 라이선스입니다.
  - 사용 권한: 귀사의 상용 제품에 LangFlow 소스 코드를 포함시키거나, 수정해서 판매(SaaS 포함)하는 행위가 모두 허용됩니다.
  - 의무 사항: 저작권 고지(Copyright Notice)만 유지하면 되며, 소스 코드를 공개할 의무도 없습니다.
  - 결론: n8n과 달리, 별도의 계약 없이도 자유롭게 번들링하여 고객에게 납품할 수 있는 것이 큰 장점입니다. 단, 최근 DataStax 인수 후 정책 변화 가능성은 지속적으로 모니

터링할 필요가 있습니다.

5.3 Flowise: 웹 생태계를 위한 경량화된 AI 빌더

Flowise는 LangFlow와 유사한 시각적 빌더(GUI) 형식을 취하고 있지만, 그 기반 기술이 Python이 아닌 Node.js(JavaScript)라는 점에서 결정적인 차이가 있습니다.

- 시작과 배경(Who & Why): 2023년 초, 개발자 헨리 헹(Henry Heng)이 공개했습니다.
  - 개발 동기: 당시 AI 개발 흐름이 Python(LangChain) 중심으로만 쏠리는 것에 대한 대안으로 등장했습니다. 전 세계 개발자의 60% 이상이 사용하는 JavaScript/TypeScript 생태계에서도 손쉽게 LLM 애플리케이션을 만들 수 있도록, LangChain.js 라이브러리를 시각화한 도구입니다.
- 핵심 정체성: "Developer-First but Lightweight". Python 런타임이 무겁거나 익숙하지 않은 웹/앱 서비스 기업에게 가장 현실적인 대안입니다.

5.3.1 Node 기반 체인/에이전트 구성 모델

Flowise는 Node.js 특유의 아키텍처적 이점을 AI 서비스에 그대로 가져옵니다.

- 아키텍처 성격: 비동기(Async) 기반의 고성능 처리
  - 리소스 효율성: Python 기반 솔루션들이 무거운 컨테이너 이미지(수 GB)와 높은 메모리를 요구하는 반면, Flowise는 상대적으로 가볍습니다. Node.js의 논블로킹 I/O(Non-blocking I/O) 모델 덕분에, 적은 리소스로도 동시에 수많은 사용자의 채팅 요청을 처리(Concurrency)하는 데 유리합니다.
  - 속도: 초기 구동 속도(Cold Start)가 빠르기 때문에, 서버를 수시로 껐다 켜거나 스케일링해야 하는 환경에서 민첩하게 반응합니다.
- Agent 구현의 단순화 (ChatFlow):

- LangFlow가 모든 부품을 나열해 조립하는 '레고'라면, Flowise는 미리 조립된 모듈을 제공하는 '밀키트'에 가깝습니다.
- ChatFlow: 일반적인 체인과 달리 '대화형 에이전트' 구성에 특화된 캔버스입니다. 메모리 관리나 프롬프트 주입 과정이 간소화되어 있어, 코드를 모르는 기획자도 "챗봇 하나 만들어보라"는 지시를 받았을 때 가장 빠르게 결과물을 낼 수 있는 구조입니다.

5.3.2 운영·모니터링(실행 추적) 관점 특징

운영 관점에서 Flowise는 '구축'은 쉽지만, '깊이 있는 분석'은 외부 도구에 의존하는 경향이 있습니다.

- 관측성(Observability)의 한계:
  - 기본 UI에서도 "어떤 질문에 어떤 답을 했는지" 정도의 히스토리는 볼 수 있습니다. 하지만, 각 단계(Step)별 소요 시간, 토큰 비용, 검색된 문서의 정확도 점수 등 엔지니어링 레벨의 데이터는 시각화 기능이 약합니다.
  - 따라서 LangSmith나 Langfuse 같은 전문 LLMOps 도구와 연동하지 않으면, "왜 챗봇이 엉뚱한 대답을 했는지" 원인을 파악하기 어렵습니다.
- 폐쇄망 운영 이슈:
  - 문제는 위에서 언급한 전문 도구들이 대부분 SaaS(클라우드) 기반이라는 점입니다. 인터넷이 차단된 폐쇄망에서는 이들과 연동할 수 없습니다.
  - 결국 폐쇄망 내에서는 Flowise가 자체 데이터베이스(SQLite/PostgreSQL)에 쌓는 원본 로그를 직접 쿼리하거나, 별도의 로깅 서버를 구축해야 하는 운영 부담이 발생합니다.

5.3.3 기업 적용 시 확장/통합 전략

Flowise의 진정한 강점은 기존 기업 시스템(Legacy)에 '조용히 스며드는(Embed)' 능력에 있습니다.

- 확장 방식: 웹 개발자 친화적(JS/TS)

  - 기업 내 웹 포털이나 그룹웨어를 개발하는 인력들은 대부분 Java나 JavaScript에 익숙합니다.
  - Flowise는 JavaScript/TypeScript로 커스텀 툴(Custom Tool)을 작성할 수 있어, 사내 개발자들이 새로운 언어(Python)를 배우지 않고도 기존 사내 API와 연동하는 로직을 쉽게 짤 수 있습니다.

- API 우선(Headless) 전략:
  - Flowise UI는 오직 '관리자'만 사용합니다. 실제 최종 사용자(임직원)는 기존 사내 포털에서 채팅을 합니다.
  - Flowise는 만든 로직을 즉시 API Endpoint로 노출해줍니다. 기존 포털 사이트에서는 이 API만 호출하면 되므로, "기존 시스템을 뜯어고치지 않고 AI 기능만 애드온(Add-on)으로 붙이는" 전략에 최적화되어 있습니다.

5.3.4 라이선스 및 상업적 사용 가능성

상용화 및 번들링 관점에서 Flowise는 매우 안전하고 매력적인 선택지입니다.

- 라이선스 유형: Apache License 2.0
- 출처 확인: Flowise GitHub Repository - LICENSE
- 상업적 사용 분석:
  - Apache 2.0은 기업 친화적인 라이선스의 대명사입니다.
  - 자유로운 번들링: 귀사의 솔루션 내부에 Flowise를 탑재하여 판매하거나, 로고를 바꾸어(White-labeling) 납품하는 것이 법적으로 허용됩니다.
  - 특허 보복 방지: 특허 관련 조항이 포함되어 있어, 향후 법적 분쟁 리스크를 줄여줍니다.
  - 결론: n8n과 같은 라이선스 비용 이슈가 없으며, Python 기반인 LangFlow보다 가벼운 런타임 덕분에 경량형 구축 솔루션(On-premise AI Kit)을 기획한다면 최적의 베이스 캠프가 될 수 있습니다.

5.4 Dify: 단순 빌더를 넘어선 '완성형 LLMOps 플랫폼'

Dify는 현재 오픈소스 AI 진영에서 가장 빠르게 성장하고 있는 솔루션 중 하나입니다. n8n이나 LangFlow가 '도구(Tool)'라면, Dify는 그 자체로 하나의 '공장(Platform)'을 지향합니다.

- 시작과 배경(Who & Why): 2023년 상반기, LangGenius라는 팀(텐센트 클라우드 출신의 엔지니어들이 주축)이 설립하여 공개했습니다.
  - 개발 동기: 창립자들은 "개발자들이 LangChain 코드를 짜느라 시간을 허비하지 않고, 기획부터 배포, 운영까지 한 곳에서 끝낼 수 있는 미들웨어가 필요하다"고 판단했습니다. 단순히 챗봇을 만드는 것을 넘어, 실제 서비스 가능한(Production-ready) 애플리케이션을 만드는 데 초점을 맞추었습니다.
- 핵심 정체성: "The Innovation Engine for GenAI Applications". 단순한 워크플로우 빌더가 아니라, RAG 파이프라인, 백엔드 API, 운영 대시보드까지 모두 포함된 All-in-One 솔루션입니다.

5.4.1 Agent/Workflow/App 통합 플랫폼 구조

Dify의 아키텍처는 철저하게 '애플리케이션(App) 중심'입니다.

- 서비스 레벨의 통합:
  - 다른 빌더들은 "로직(Logic)"만 만듭니다. 하지만 Dify는 로직을 감싸는 사용자 인증, API 키 발급 관리, 프론트엔드 웹 페이지까지 자동으로 생성해줍니다.
  - 즉, Dify에서 '저장'을 누르면 즉시 사내 임직원에게 배포할 수 있는 URL이 생성되거나, 기존 시스템에서 호출할 수 있는 API 명세서가 나옵니다.
- 고급 에이전트 오케스트레이션:
  - 단순한 대화 흐름뿐만 아니라, 복잡한 업무 흐름(Workflow)을 노드 방식으로 설계할 수 있습니다.
  - 특히 '의도 분류(Intent Classification)'나 '파라미터 추출(Parameter Extraction)' 같은 고급 기능이 내장 컴포넌트로 제공됩니다. 이는 사용자의 질문이 "A/S 문의"인

지 “구매 문의”인지 AI가 판단하여 각기 다른 부서의 DB를 조회하게 만드는 ’라우팅(Routing)’ 구현을 클릭 몇 번으로 가능하게 합니다.

5.4.2 RAG/운영(LLMOps) 내장 기능 범위

Dify가 엔터프라이즈 시장에서 각광받는 가장 큰 이유는 RAG(검색 증강 생성)와 운영 도구의 내재화입니다.

- RAG 파이프라인의 완결성:
  - 타 솔루션(n8n, LangFlow)은 VectorDB에 이미 데이터가 들어있다는 전제하에 ’연결’만 지원하거나, 별도의 파이썬 스크립트로 데이터를 쪼개서 넣어야 합니다.
  - Dify는 ETL(Extract, Transform, Load) 기능이 내장되어 있습니다. PDF나 엑셀 파일을 업로드하면 Dify가 알아서 텍스트를 추출하고, 적절한 크기로 자르고(Chunking), 벡터로 변환하여 DB에 저장합니다. 이 과정이 자동화되어 있어 데이터 엔지니어의 도움 없이도 지식 베이스를 구축할 수 있습니다.
- Day 2 Operation (운영 및 최적화):
  - 앱을 만든 직후(Day 1)보다 중요한 것이 지속적인 운영(Day 2)입니다.
  - 프롬프트 디버깅: AI 답변이 마음에 들지 않을 경우, 운영자가 정답을 수정해서 등록하면 AI가 이를 학습하여 다음부터 개선된 답변을 내놓는 ‘어노테이션(Annotation)’ 기능이 강력합니다.
  - 모니터링: 토큰 사용량 비용, 사용자 피드백(좋아요/싫어요), 응답 지연 시간(Latency) 등을 시각화된 차트로 제공합니다.

5.4.3 번들 구성 시 장점과 비대해지는 운영 복잡도

Kuberentes 환경에 Dify를 번들링하여 납품할 때는 ‘강력한 기능’과 ’무거운 덩치’ 사이의 트레이드오프를 고려해야 합니다.

- 장점: 고객 만족도가 높은 완성형 패키지

  - 백엔드 API 서버, 비동기 처리를 위한 워커(Worker), 프론트엔드 UI, 벡터 처리기 등이 완벽하게 통합되어 있어, 설치 즉시 "그럴듯한 솔루션"을 고객에게 보여줄 수 있습니다. 고객사 입장에서는 별도의 UI 개발 없이 바로 활용 가능하다는 점이 큰 매력입니다.

- 단점: 헤비급 아키텍처에 따른 운영 복잡도
  - Dify는 전형적인 마이크로서비스 아키텍처입니다. 제대로 구동하기 위해서는 **API Server**, **Worker**, **Web**, **Redis**(캐시/큐), **PostgreSQL**(메타데이터), **Weaviate** 혹은 **Qdrant**(VectorDB), **Sandbox**(코드 실행 격리) 등 최소 7~8개 이상의 컨테이너가 동시에 떠야 합니다.
  - 폐쇄망 배포 난이도: 인터넷이 되는 환경에서는 **docker-compose up** 한 줄로 실행되지만, 폐쇄망에서는 이 많은 이미지들의 버전을 맞추고, 각 컨테이너 간의 네트워크 통신을 설정하는 데 상당한 엔지니어링 리소스가 소모됩니다. 또한, 하드웨어 리소스(CPU/Memory) 요구량도 타 솔루션 대비 높습니다.

5.4.4 라이선스 및 상업적 사용 가능성

Dify는 상업적 활용에 있어 매우 유연하고 개방적인 정책을 취하고 있습니다.

- 라이선스 유형: Apache License 2.0
- 출처 확인: Dify GitHub Repository - LICENSE
- 상업적 사용 분석:
  - Dify는 Apache 2.0 라이선스를 따르므로, 기업이 이를 수정하거나 그대로 사용하여 상용 서비스를 구축하는 데 법적 제약이 거의 없습니다.
  - 번들링 및 재판매: 귀사의 솔루션 패키지에 Dify를 포함시켜 고객사에 구축형으로 납품하거나, 로고와 UI 색상을 변경(Rebranding)하여 자사 솔루션인 것처럼 제공하는 것도 가능합니다.
  - 결론: 기술적으로는 무겁지만, 라이선스적으로는 n8n과 달리 추가 비용 없이 자유롭게 활용 가능하여, 중대형 규모의 엔터프라이즈 AI 플랫폼 사업을 수주할 때 가장 적합한 베이스 엔진입니다.

5.5. [종합 비교 분석] 엔터프라이즈 AI Agent Builder 선정 매트릭스

구분	n8n	LangFlow	Flowise	Dify
핵심 정체성	Workflow Automation (시스템 통합 미들웨어)	Visual LangChain (Py) (개발자용 프로토타이핑 도구)	Visual LangChain (JS) (웹 친화적 경량 빌더)	LLMOps Platform (구축·운영·RAG 올인원)
라이선스 및 상업적 사용(중요)	□ 주의: Fair-code(Sustainable Use License) • 사내 사용: 무료 • 상용 판매/SaaS: 불가 (별도 Enterprise 계약 필수)	□ 안전: MIT License • 소스 수정/배포 자유 • 상용 솔루션 포함 가능 • 법적 제약 가장 적음	□ 안전: Apache 2.0 • 상용 번들링 허용 • 특허 보복 방지 조항 포함 • 기업 친화적	□ 안전: Apache 2.0 • 상용 번들링 허용 • 리브랜딩(OEM) 가능 • 기업 친화적
아키텍처 및 런타임	Node.js (DAG 기반) • 선형적 작업 처리에 최적 • 복잡한 루프(Loop) 구현 난해	Python (FastAPI) • LangChain 1:1 매핑 • 무거운 Python 런타임 필요	Node.js (Async I/O) • 가볍고 빠른 실행 속도 • 동시성 처리에 유리	Microservices (Go/Py) • API/Worker/Web 분리 • 확장성은 좋으나 무거움
폐쇄망 배포 (Air-gapped)	용이 (Easy) • 단일 Docker 이미지 • 의존성이 내장되어 있음	어려움 (Hard) • Python 패키지 의존성 지옥 • 버전 충돌 빈번 발생	보통 (Medium) • Node 모듈 관리 필요 • 비교적 이미지 사이즈 작음	복잡함 (Complex) • 최소 7~8개 컨테이너 구동(Redis, DB, VectorDB 등)
확장성(Extensibility)	JS/TS Code Node • HTTP Request 처리가 강력 • 1,000+ 사전 연동 모듈	Python Component • Pandas/NumPy 활용 가능 • 데이터 분석가에게 유리	JS Custom Tool • 웹/앱 개발자 친화적 • 기존 레거시 API 연동 용이	Plugin / API 표준 • 표준화된 도구 정의 지원 • 코드 노드 및 HTTP 지원
운영/모니터링 (Observability)	실행 결과 중심• LLM 추론 과정 추적 미흡• 프롬프트 디버깅 어려움	상세 추적 가능 • But, UI가 매우 복잡함 • 흐름 파악에 러닝커브 존재	기본 기능 미흡 • LangSmith 등 외부 도구의존도가 매우 높음	매우 우수 (Built-in) • 토큰/비용/로그 분석 통합 • 사용자 피드백 수집 가능

제품화 적합성 (Bundling)	☐ 낮음 (Risky) • 라이선스 비용 이슈 발생 • 내부 시스템 연동용 권장	☐ 중간 • 엔지니어용 도구 성격 • 일반 사용자에게 불친절	☐ 중간/높음 • 기존 웹 시스템에 '기능 추가' 형태로 적합	☐ 높음 (Best) • 완성된 플랫폼 납품 시 최적 • RAG 파이프라인 내장 강점

[상세 분석] 라이선스 및 도입 전략 가이드

5.5.1. 라이선스 리스크 상세 분석 (n8n 주의 요망)

- n8n (Sustainable Use License):
  - 출처: n8n License Policy
  - 핵심: "n8n과 경쟁하는 상용 서비스(SaaS, 관리형 서비스 등)를 만들지 않는 한 무료"라는 조건입니다. 하지만 기업이 자사 솔루션에 n8n을 내장하여 고객에게 돈을 받고 파는 행위는 '상업적 배포'로 간주되어 라이선스 위반 소지가 매우 높습니다. 따라서 n8n 본사와 'Commercial License' 계약을 체결해야 하므로 원가 상승 요인이 됩니다.
- LangFlow (MIT) / Flowise & Dify (Apache 2.0):
  - 출처: LangFlow GitHub, Flowise GitHub, Dify GitHub
  - 핵심: 이들은 OSI(오픈소스 이니셔티브)가 승인한 표준 오픈소스입니다. 귀사가 이 소스코드를 수정하여 패키징하고, 고객사 폐쇄망에 설치하여 수억 원의 사업비를 받아도 라이선스 비용을 지불할 의무가 없습니다. (단, Apache 2.0은 변경 사항에 대한 고지 의무가 있습니다.)

5.5.2. 폐쇄망(Air-gapped) 환경 구축 전략

- Dify: 기능은 가장 강력하지만, 설치 난이도가 가장 높습니다. 폐쇄망 내에 `PostgreSQL(벡터 확장 포함)`, `Redis`, `MinIO(S3 대용)`, `Weaviate/Qdrant` 등 다수의 미들웨어를 미리

구성해야 합니다. 운영 조직의 Kubernetes(K8s) 역량이 필수적입니다.

- LangFlow: 설치보다 '유지보수'가 지옥이 될 수 있습니다. 폐쇄망 내부에서 특정 Python 라이브러리(예: `pydantic`, `numpy` 등)의 버전 충돌이 발생하면, 인터넷 없이 이를 해결하는 데 며칠이 소요될 수 있습니다.
- Flowise: 기능과 운영 용이성 사이에서 가장 균형 잡힌 선택(Sweet Spot)일 수 있습니다. Node.js 기반이라 컨테이너가 가볍고, RAG나 복잡한 기능은 사내에 구축된 다른 API를 호출하는 방식으로 유연하게 대처할 수 있습니다.

5.5.3. 최종 선정 가이드 (제언)

- Scenario A: "우리는 완성된 AI 포털 솔루션을 고객에게 납품해야 한다."
  - → Dify가 정답입니다. 로그인 화면부터 관리자 페이지까지 이미 다 만들어져 있어 개발 공수를 획기적으로 줄여줍니다.
- Scenario B: "기존 사내 그룹웨어에 챗봇 기능만 살짝 추가하고 싶다."
  - → Flowise가 적합합니다. 가볍게 띄워서 API만 따온 뒤, 기존 그룹웨어 화면에 연동하면 됩니다.
- Scenario C: "다양한 레거시 시스템(ERP, HR)을 연동하는 것이 AI보다 더 중요하다."
  - → n8n이 기능적으로는 최고이나, 라이선스 계약(유료)을 전제로 도입해야 합니다. 비용이 문제라면 n8n의 대안으로 Apache 2.0 라이선스인 Activepieces 등을 검토할 수 있습니다.

6장. 요구사항 매핑 기반 실증 검증: 연결 가능 vs 운영 가능

기업의 엔터프라이즈 환경, 특히 폐쇄망(Air-gapped) 환경에서의 AI Agent Builder 선정은 단순한 기능의 유무(Connectivity)를 넘어, 실제 업무 프로세스로서의 운영 지속성(Operability)과 안정성(Stability)이 핵심 검증 기준이 되어야 합니다. 본 장에서는 n8n, LangFlow, Flowise, Dify 4종의 솔루션을 대상으로 실무 레벨의 상세 검증 결과를 기술합니다.

6.1 연동 방식 검증 (Connectors & Integrations)

AI Agent가 엔터프라이즈 환경에서 실제 가치를 내기 위해서는 “답변 품질”만으로는 부족합니다. Agent가 ERP, CRM, 레거시 DB, 파일/그룹웨어, 최신 VectorDB 등과 안정적으로 연결되어야 업무 흐름에 들어갈 수 있습니다.

따라서 이 장에서는 단순히 “연동이 된다/안 된다”가 아니라 다음을 중심으로 봅니다.

- 인증 복잡도 대응: OAuth2(3-legged), SSO/토큰 갱신, API Key 혼재 환경
- 대량 데이터 처리 안정성: 페이징, 재시도, 타임아웃, 레이트리밋, 에러 핸들링
- 운영 관점의 재사용성: 크리덴셜 중앙 관리, 노드/컴포넌트 표준화, 감사(로그) 추적성

6.1.1 네이티브 커넥터 vs HTTP/API 호출 vs 코드 노드 비교

기업 환경의 현실은 “표준 REST API만 있는 세상”이 아닙니다. SOAP, 사내 표준이 아닌 API, 오래된 인증 방식, 불완전한 문서 등 비정형 연동이 상수입니다. 각 도구가 이를 어떤 방식으로 흡수하는지 비교합니다.

비교 표 (Flowise / Dify / n8n / LangFlow)

구분	n8n (워크플로 자동화 중심)	Dify (LLM 앱/워크플로 빌더 중심)	LangFlow (LLM 로직 설계 중심, Python)	Flowise (LLM 로직 설계 중심, Node/JS)
네이티브 커넥터(사전 정의 노드)	강점: 업무 SaaS·자동화 관점의 노드가 풍부하고, “연동 허브”로 쓰기 좋습니다. 크리덴셜을 중앙 관리해 재사용성이 높습니다.	중간: LLM/검색/지식베이스 연동은 빠르지만, 전통적 엔터프라이즈 SaaS/레거시 연동 폭은 n8n 대비 제한적입니다.	중하: 생태계의 툴/컴포넌트를 UI로 붙이는 구조라 “있긴 한데 운영 친화적이지 않은” 경우가 있습니다.	중하: LangChain 기반 도구를 UI로 묶는 성격이 강합니다. 커넥터 다양성은 확장 가능하나 “기본 제공 품질/운영성”은 케이스별 편차가 큽니다.

HTTP/API 호출(범용 통신)	강점: REST/GraphQL/SOAP까지 폭넓게 대응하며, 페이징·재시도·데이터 매핑을 UI로 구성하기가 좋습니다.	표준 지향: OpenAPI 기반 "툴 등록"에 강합니다. 표준 명세가 있는 API는 빠르지만, 비표준·예외 많은 API는 번거로울 수 있습니다.	기본 제공: 단순 호출은 쉽지만, 복잡한 인증 플로우/에러 처리/대량 페이징을 운영 수준으로 다듬는 데는 추가 구성이나 코드가 필요해질 수 있습니다.	기본 제공: 기본 요청은 가능하나, 엔터프라이즈에서 요구하는 세밀한 통제(재시도 정책, 레이트 리밋 대응, 토큰 갱신 등)는 커스텀 쪽으로 기울기 쉽습니다.
코드 기반 확장(커스텀 로직)	유연: JS 기반으로 데이터 변환/전처리/매핑에 강합니다. 다만 코드가 늘수록 워크플로가 "개발 산출물"로 변질될 위험은 관리 필요합니다.	제한적: 간단한 코드 블록/후처리는 가능하나, 무거운 라이브러리/시스템 수준 접근은 제한될 수 있습니다. "앱 빌더"로서의 범위에 머무는 편입니다.	강점: Python 기반 커스텀 컴포넌트 확장이 자연스럽습니다. 개발자에겐 강력하나, 운영/현업 관점에선 표준화가 없으면 진입장벽이 커질 수 있습니다.	강점: JS/Node 생태계로 커스텀 노드/컴포넌트 확장이 가능합니다. 다만 구현 방식이 팀에 의존하면 유지보수 리스크가 커집니다.
운영 포인트(재사용·통제·감사)	강점: 크리덴셜/재시도/로깅/워크플로 분리 등 "운영형 자동화"에 유리합니다.	강점(LLM 앱 관점): 앱/프롬프트/도구 관리의 틀이 있어 "서비스 형태"로 만들기 좋습니다. 다만 대규모 연동 허브로 쓰려면 한계가 나타날 수 있습니다.	개발 친화: 설계·실험은 빠르지만, 운영 표준(런북, 장애 대응, 변경관리)을 별도로 세우지 않으면 파편화가 쉽습니다.	개발 친화: LangFlow와 유사합니다. PoC→운영 전환 시 표준화/배포 전략이 중요합니다.

핵심 정리

- n8n은 "기업 내외 시스템을 촘촘히 연결해서 데이터를 흘려보내는" 연동 파이프라인/자동화 허브에 가장 강합니다. 레거시·예외·대량 처리의 현실을 운영 관점에서 흡수하기 좋습니다.
- Dify는 "LLM 기능을 중심으로 앱을 빠르게 만들고, 표준화된 API를 도구로 붙이는" LLM 앱 빌더/서비스 기획에 유리합니다. 다만 엔터프라이즈형 연동 허브로 무게중심을 두면 제약이 보일 수 있습니다.
- LangFlow / Flowise는 "LLM 체인/에이전트 로직을 시각적으로 설계하고, 필요한 경우 개발자가 컴포넌트를 확장하는" LLM 로직 설계·개발자 중심 도구입니다. 즉, 연동을 '제품 수준 운영'으로 만들려면 커스텀과 표준화(배포/버전/테스트)가 필수입니다.

6.1.2 Crawler4AI / OCR / BM25 연동 난이도 평가 (전처리 파이프라인)

RAG(검색 증강 생성)의 성능은 “얼마나 깨끗한 데이터를 가져오느냐”에 달려 있습니다. 비정형 데이터(웹페이지, PDF)를 정형화하는 과정에서의 각 도구별 능력을 검증합니다.

1) Crawler4AI (웹/문서 크롤링 및 파싱)

- n8n (프로세스 제어의 강자):
  - Docker로 띄운 Crawler4AI 서버의 API를 호출하고, 응답받은 Markdown 데이터를 즉시 JSON으로 변환하여 DB에 넣는 흐름(Flow) 제어가 매우 매끄럽습니다.
  - 특히 URL 1,000개를 순차적으로 도는 Loop(반복문) 처리와 중간에 실패했을 때의 에러 핸들링이 4개 툴 중 가장 직관적이고 강력합니다.
- LangFlow / Flowise (도구 의존형):
  - LangChain의 `WebBaseLoader`나 `Puppeteer` 노드를 사용합니다.
  - 기능 자체는 강력하지만, 크롤링 결과를 받아 청킹(Chunking)하고 메타데이터를 입히는 과정을 시각적으로 연결할 때 선(Wire)이 복잡해지고 디버깅이 어렵습니다.
- Dify (제한적):
  - 자체적인 웹 스크래핑 도구가 있으나, Crawler4AI처럼 세밀한 옵션(User-Agent 변경, 지연 로딩 처리 등) 제어는 어렵습니다.

2. OCR & BM25 (이미지 인식 및 키워드 검색)

- Dify (운영 편의성 최상 - Blackbox):
  - 가장 쉽습니다. 파일만 업로드하면 내부적으로 OCR(텍스트 추출), 청킹, 임베딩, 키워드 인덱싱을 알아서 처리합니다.
  - 단점: “어떤 OCR 엔진을 쓸지”, “BM25 가중치를 어떻게 줄지”와 같은 세부 튜닝이 불가능한 ‘블랙박스’ 형태입니다.
- n8n / LangFlow / Flowise (구현 자유도 최상 - DIY):

- Elasticsearch, Weaviate 등의 API를 직접 호출하여 하이브리드 검색(Vector + Keyword) 로직을 직접 구현해야 합니다.
- 구현 공수는 많이 들지만, 기업 고유의 도메인 사전에 맞춘 토크나이저 설정이나 랭킹 알고리즘 수정이 가능하여 고성능 RAG 구축에 필수적입니다.

6.1.3 VectorDB / Neo4j 연동의 구현 공수 및 표준화 가능성

데이터베이스는 AI 에이전트의 '기억(Memory)'을 담당합니다. 단순 저장을 넘어, 추론을 위한 데이터 구조를 어떻게 다루는지 비교합니다.

1) VectorDB (Qdrant, Pinecone 등)

- n8n (명시적 데이터 파이프라인):
  - 데이터의 Insert(삽입), Upsert(수정), Query(조회) 과정을 일반 SQL DB 다루듯 단계별로 명시합니다.
  - 장점: 데이터가 어떤 형태로 변환되어 들어가는지 눈으로 확인(Debug)하기 가장 좋습니다. 데이터 엔지니어링 관점에서 적합합니다.
- Dify / LangFlow (추상화된 지식베이스):
  - "이 문서를 지식베이스에 추가해"라는 식의 고수준 명령으로 처리합니다.
  - 장점: 구축 속도가 매우 빠릅니다.
  - 단점: 특정 청크(Chunk)가 잘못 저장되었을 때, 이를 찾아내고 수정하는 디버깅 과정이 n8n에 비해 불투명합니다.

2) GraphDB (Neo4j – 관계형 지식 추론)

이 부분에서 도구 간 성격 차이가 극명하게 갈립니다.

- LangFlow / Flowise (추론 로직 중심):
  - 가장 강력합니다. LangChain의 `GraphCypherQAChain` 같은 고급 체인을 시각적으로 배치할 수 있습니다.

  - 자연어 질문("CEO와 연결된 이사들은 누구지?")을 입력하면, 이를 Graph Query(Cypher)로 변환해주는 Reasoning(추론) 에이전트를 쉽게 만들 수 있습니다.

- n8n (트랜잭션 중심):
  - Neo4j에 미리 짜여진 Cypher 쿼리를 전송하고 결과를 JSON으로 받아오는 방식입니다.
  - "질문을 쿼리로 변환"하는 창의적 작업보다는, "정해진 관계 데이터를 조회해서 리포트를 만드는" 정형적 작업에 적합합니다.
- Dify (기본 지원):
  - 최근 지식 그래프(Knowledge Graph) 지원을 시작했으나, 아직은 텍스트 청크를 그래프 구조로 단순 매핑하는 수준이거나, 별도의 외부 설정을 통해야 하는 경우가 많습니다. LangFlow만큼의 유연한 그래프 질의 생성 능력은 아직 부족합니다.

종합 비교 요약

특징	n8n	Dify	LangFlow / Flowise
주 사용 목적	엔터프라이즈 워크플로우 자동화, 데이터 파이프라인(ETL)	대고객용 AI 챗봇 서비스, 사내 지식 검색 시스템(RAG)	LLM 네이티브 애플리케이션 프로토타이핑 및 로직 설계
연동 강점	레거시 시스템, 복잡한 API, 대량 데이터 처리	표준화된 API(OpenAPI), 간편한 RAG 구성	Python/JS 라이브러리 기반의 복잡한 AI 로직 구현
추천 대상	데이터 엔지니어/백엔드 개발자 • (안정적 시스템 연동 중요)	기획자/PM/운영자 • (빠른 서비스 런칭 및 운영 중요)	AI 엔지니어/풀스택 개발자 • (LLM 로직의 세밀한 튜닝 중요)

6.2 MCP 지원 검증 (Model Context Protocol)

2024년 말 등장하여 2025년 AI 생태계의 핵심 표준으로 부상한 MCP(Model Context Protocol)는 쉽게 말해 “AI 에이전트를 위한 USB-C 표준”입니다.

과거에는 AI 에이전트가 사내 DB나 특정 SaaS(Notion, Slack)에 접속하려면 각 플랫폼(Dify, Flowise 등)에 맞는 ’전용 플러그인’을 각각 개발해야 했습니다. 하지만 MCP 표준을 따르면, 단 한 번의 연결 도구(MCP 서버) 개발로 모든 AI 플랫폼에서 즉시 사용할 수 있습니다.

이는 폐쇄망 환경에서 수많은 레거시 시스템을 관리해야 하는 엔터프라이즈 기업에게 연동 개발 공수를 획기적으로 줄여줄 핵심 기술입니다.

6.2.1 Native MCP 지원 여부 (클라이언트/서버 관점)

“우리 플랫폼은 MCP로 만든 도구를 바로 가져다 쓸 수 있는가?”(MCP Client 기능)에 대한 각 솔루션의 2025년 1월 기준 대응 현황입니다.

비교 항목	Dify	Flowise	LangFlow	n8n
지원 방식	☐ Native UI (최상) • 최신 버전에서 MCP 서버 연결을 위한 전용 설정 UI를 제공합니다. MCP 서버 URL만 입력하면 해당 서버의 기능(Tool)이 자동으로 Dify 도구 목록에 동기화됩니다.	☐ JS/Custom Tool (중) • Node.js 기반인 LangChain.js 생태계를 따릅니다. UI 상의 ‘Custom Tool’ 노드에서 MCP JS SDK를 활용하여 연결합니다. JavaScript/TypeScript 개발자에게 익숙한 환경입니다.	☐ Python/Adapter (중) • Python 기반인 LangChain 라이브러리의 `MCPToolAdapter`에 의존합니다. 코드로 된 설정 블록을 추가하거나, Custom Component 내에서 MCP 클라이언트를 선언하는 방식이 주를 이룹니다.	☐ Request 기반 (하) • 아직 공식적인 ’MCP Client Node’는 로드맵 단계입니다. 현재는 HTTP Request 노드를 이용해 MCP 프로토콜(JSON-RPC) 규격에 맞춰 직접 통신하거나, 커뮤니티 노드를 사용해야 합니다.
동기화 편의성	☐ 매우 높음 • 외부 툴이 업데이트되면 Dify 화면에서 버튼 클릭만으로 즉시 반영됩니다. (No-Code 친화적)	보통 • 연동 코드를 ‘Custom Tool’ 안에 직접 심어야 하는 경우가 많아, 도구 변경 시 코드 수정이 필요할 수 있습니다.	보통 • Python 코드 레벨에서의 제어가 필요하므로, UI 상에서 도구 목록을 시각적으로 관리하기는 다소 번거롭습니다.	낮음 • 프로토콜 규격을 이해하고 수동으로 설정해야 하므로 러닝 커브가 높습니다.
특이 사항	가장 빠른 적용 속도. 기획자/운영자가 쓰기에 적합.	웹/JS 개발자 친화적. 웹 기술 기반의 사내 시스템과 연동하기 좋음.	백엔드/AI 개발자 친화적. Python 라이브러리와의 결합력이 강력함.	자동화 중심. 아직은 기존의 네이티브 노드 방식이 훨씬 강력함.

6.2.2 MCP 적용 시 커스텀 툴 개발 공수 절감 포인트

기업 내부 시스템(예: 사내 인사 시스템)을 AI와 연결한다고 가정할 때, 기존 방식과 MCP 방식의 업무 효율을 비교합니다.

기존 방식 (The Hard Way – N배의 공수)

만약 사내 시스템을 여러 플랫폼에서 테스트해야 한다면, 아래와 같이 4번의 개발을 해야 합니다.

1. Dify용: OpenAPI(Swagger) 명세서 작성 및 등록.
2. Flowise용: JavaScript 기반의 'Custom Tool' 코드 작성.
3. LangFlow용: Python 기반의 'Custom Component' 코드 작성.
4. n8n용: TypeScript 기반의 'Community Node' 개발.

- 문제점: 인사 시스템 API가 변경되면 4곳 모두 코드를 뜯어고쳐야 합니다. (유지보수 비용 폭증)

MCP 방식 (The Smart Way – 1번의 개발)

사내 인사 시스템을 바라보는 'MCP 서버(표준 인터페이스)'를 딱 하나만 개발합니다.

1. 개발: Python이나 Node.js로 간단한 MCP 서버 구축 (인사 DB 조회 함수 포함).
2. 적용:
    - Dify: 설정 화면에 URL 입력 ☐ 즉시 사용 가능
    - Flowise: MCP 클라이언트 툴 하나만 배치 ☐ 즉시 사용 가능
    - LangFlow: MCP 어댑터 컴포넌트 배치 ☐ 즉시 사용 가능
    - Claude Desktop: 로컬 설정 파일에 추가 ☐ 즉시 사용 가능
3. 결과: 플랫폼을 변경하더라도, 만들어둔 연결 도구(MCP 서버)는 그대로 재사용합니다.

- 검증 결론: Vendor Lock-in(솔루션 종속)을 피하고 "도구의 자산화"를 위해서는 MCP 호환성이 필수입니다. 특히 Flowise(JS)와 LangFlow(Python) 간의 언어 장벽을 허무는 데 MCP가 결정적인 역할을 합니다.

6.2.3 사내 레거시 시스템의 MCP 호환화 전략 (어댑터 설계)

오래된 사내 시스템(Legacy)은 최신 기술인 MCP를 알지 못합니다. 이를 연결하기 위한 아키텍처 전략입니다.

MCP Bridge Server 전략 (어댑터 패턴)

폐쇄망 내의 복잡한 레거시 시스템 앞단에 '통역사' 역할의 가벼운 서버를 하나 둡니다.

1. 구성 요소:
    - Backend: 레거시 DB(Oracle, MS-SQL)나 SOAP 기반 구형 시스템.
    - Bridge Server (Middle): Python(FastAPI) 또는 Node.js로 작성된 경량 서버. 레거시 프로토콜을 MCP 표준(JSON-RPC 유사)으로 변환.
    - Frontend (Agent Builder): Dify, Flowise, LangFlow 등.
2. 동작 흐름:
    - Agent (Flowise/Dify 등): "김철수 사원 정보 줘" (MCP 프로토콜로 요청)
    - Bridge Server: 요청 수신 → 레거시가 이해하는 SQL 쿼리나 SOAP XML로 변환 → 질의 실행.
    - Bridge Server: 결과 수신 → 깔끔한 텍스트/JSON으로 변환 → Agent에 반환.
3. 장점:
    - 보안: AI 에이전트가 민감한 DB 접속 정보를 알 필요가 없습니다. (Bridge Server에서만 관리)
    - 추상화: Flowise나 Dify 사용자(운영자)는 복잡한 SQL을 몰라도 "함수 이름"만 보고 도구를 끌어다 쓸 수 있습니다.

요약 권장 사항

- 운영/기획 중심 팀: Dify의 Native MCP 기능을 사용하여, 개발팀이 만들어준 MCP 서버를 손쉽게 연결해 쓰십시오.
- Node.js/웹 개발팀: Flowise를 사용하여, JavaScript/TypeScript 생태계의 MCP 도구들을 활용하고 웹 서비스와 긴밀하게 통합하십시오.
- Python/AI 엔지니어팀: LangFlow를 사용하여, Python 라이브러리와 MCP를 결합한 고도화된 추론 로직을 설계하십시오.
- 워크플로우 자동화팀: n8n은 아직 HTTP 연동이 주력이므로, n8n의 강력한 네이티브 커넥터를 우선 사용하되 MCP 지원 업데이트를 주시하십시오.

6.3 운영 관점 검증 (Observability & Reproducibility)

PoC(개념 증명) 단계에서는 “신기한 기능을 보여주는 것”이 중요하지만, 실제 운영 단계에서는 “문제가 생겼을 때 원인을 얼마나 빨리 찾고 해결할 수 있는가?”가 핵심입니다.

LLM은 확률적으로 작동하기 때문에, “왜 어제는 잘 되던 답변이 오늘은 이상한가?”를 추적할 수 있는 관측성(Observability)과, 에러 발생 시 시스템이 멈추지 않게 하는 안정성(Resilience)이 필수적입니다.

6.3.1 실행 이력·입출력·근거 추적 (감사/재현)

“AI가 거짓말을 했다”는 리포트가 들어왔을 때, 운영자가 로그를 열어 원인을 분석하는 시나리오를 비교합니다.

비교 항목	n8n (투명한 데이터 파이프라인)	Dify (사용자 경험 중심)	LangFlow / Flowise (개발자 도구 의존)
로그 시각화	☐ 현미경 수준 (Node-Level) • 모든 노드의 Input/Output JSON 데이터가 시각적으로 완벽하게 보존됩니다. 데이터가 어디서 변형되었는지, 어떤 API가 500 에러를 뱉었는지 100% 추적 가능합니다. 백엔드 디버깅에 최적화되어 있습니다.	☐ 대화/근거 중심 (App-Level) • 사용자와 주고받은 메시지 로그와 '인용(Citation)' 정보를 보여주는 데 특화되어 있습니다. "어떤 문서를 참고해서 답변했는지"를 현업 담당자가 확인하기 좋습니다. 단, 내부 변수의 복잡한 흐름을 추적하기는 n8n보다 약합니다.	☐ 외부 도구 필수 (LangSmith) • 자체 로그는 휘발성 콘솔 로그에 가깝습니다. LangSmith나 Arize Phoenix 같은 별도의 관측 도구(SaaS/Self-hosted)를 붙이지 않으면, 운영 환경에서 데이터 흐름을 파악하기가 매우 어렵습니다.
데이터 보존	DB 저장 • 실행 이력이 내부 DB(Postgres 등)에 저장되어, 과거 특정 시점의 실행 결과를 다시 열어보고 재실행(Retry) 해볼 수 있습니다.	내장 로그 • 별도 설정 없이 대화 이력이 저장되며, 관리자 페이지에서 쉽게 조회할 수 있습니다.	설정 필요 • 기본 설정만으로는 재부팅 시 로그가 사라질 수 있으며, 별도의 영구 저장소 설정을 직접 해야 합니다.

6.3.2 장애 분석: Flow 추적, 재시도, 타임아웃, 에러 핸들링

새벽 2시에 외부 API가 다운되거나 LLM 응답이 1분 넘게 지연될 때, 시스템이 어떻게 대처하는지 비교합니다.

1. 에러 핸들링 (Error Handling)

- n8n (엔터프라이즈급 제어):
  - "Error Trigger" 노드: 특정 노드가 실패했을 때만 실행되는 별도의 경로(Branch)를 만들 수 있습니다. (예: DB 저장 실패 시 슬랙 알림 발송 후 엑셀로 백업)
  - 정책 설정: 노드별로 "에러 무시하고 진행(Continue on Fail)", "3회 재시도(Retry)", "즉시 중단" 등을 UI에서 상세하게 설정할 수 있습니다.
- Dify (서비스 관점):

  - 최근 워크플로우 기능 도입으로 개선되었으나, 기본적으로는 에러 발생 시 사용자에게 "죄송합니다, 오류가 발생했습니다"를 띄우고 멈추는 경우가 많습니다. 복잡한 복구 로직(Fallback)을 짜기에는 아직 유연성이 부족합니다.

- Flowise / LangFlow (코드 의존):
  - UI 상에서 재시도 로직을 직관적으로 짜기 어렵습니다. LangChain 라이브러리 내부의 기본 재시도 설정에 의존하거나, Custom Code 안에서 `try-catch` 문을 직접 작성해야 안전합니다.

2. 타임아웃 및 병목 관리

- n8n: 명시적인 타임아웃 설정이 가능하며, 대량 요청이 들어오면 큐(Queue)에 쌓아두고 순차 처리하는 부하 제어(Throttling) 능력이 탁월합니다.
- LangFlow / Flowise: Python/Node.js 프로세스의 기본 타임아웃을 따르며, LLM 응답이 늦어질 경우 프로세스가 행(Hang)에 걸릴 위험이 있어 별도의 배포 전략(Gunicorn/PM2 설정 등)이 필요합니다.

6.3.3 유지보수: 버전업, 플러그인 호환성, 배포 롤백

시스템을 1년 이상 운영하면서 기능을 업그레이드할 때 겪게 될 미래입니다.

1. 버전 관리 (Versioning & Rollback)

- n8n (DevOps 친화적):
  - 워크플로우를 JSON 파일로 추출하여 Git(GitLab/GitHub)과 연동할 수 있습니다. 개발자가 익숙한 방식(Commit, Push, Rollback)으로 형상 관리가 가능합니다.
- Dify (운영자 친화적):
  - 앱(App) 내부에서 "발행 기록(History)"을 리스트로 보여줍니다. 클릭 한 번으로 "어제 버전으로 되돌리기"가 가능하여 비개발자도 쉽게 롤백할 수 있습니다.
- Flowise / LangFlow (파일 기반):

- 내보내기(Export)한 JSON 파일을 수동으로 관리해야 합니다. 팀원 A가 수정한 내용을 팀원 B가 덮어쓰는 충돌(Conflict) 사고가 발생하기 쉽습니다.

2. 업그레이드 안정성 (Breaking Changes)

- LangFlow / Flowise (위험 높음):
  - 기반이 되는 LangChain 라이브러리가 매우 빠르게 변합니다. 엔진(Docker Image)을 업데이트했더니, 기존에 잘 돌던 그래프가 "호환되지 않는 노드"라며 깨지는 경우가 빈번합니다. 운영 팀의 높은 기술적 대응 역량이 요구됩니다.
- n8n (매우 안정적 □):
  - 노드 버전(v1, v2)을 내부적으로 관리합니다. n8n 엔진을 업데이트해도, 기존 워크플로우는 구버전 노드를 그대로 유지하므로 깨지지 않고 돌아갑니다. 가장 안정적인 운영이 가능합니다.
- Dify (안정적):
  - 플랫폼 자체의 완결성이 높아, 업데이트 시 내부 기능이 깨지는 일은 드뭅니다. 오픈소스 업데이트 주기도 빠르고 안정화되어 있습니다.

종합 운영 점수표

평가 항목	n8n	Dify	Flowise	LangFlow
로그 추적성	□□□□□ (상세함)	□□□□ (직관적)	□□ (도구 필요)	□□ (도구 필요)
에러 복구력	□□□□□ (강력함)	□□□ (보통)	□□ (코드 필요)	□□ (코드 필요)
버전 관리	□□□□□ (Git 연동)	□□□□ (자체 UI)	□□ (수동)	□□ (수동)
업데이트 안정성	□□□□□ (하위호환)	□□□□ (안정적)	□□ (파손 주의)	□□ (파손 주의)

결론 및 제언

- 안정성이 최우선인 기간계 시스템 연동: n8n이 유일한 대안입니다. 에러가 나면 안 되거나, 나더라도 확실하게 기록하고 재처리해야 하는 업무에 적합합니다.
- 현업 대상의 AI 서비스 운영: Dify가 가장 적합합니다. 답변의 근거를 확인하고, 문제가 생기면 버튼 하나로 롤백하는 운영 편의성이 압도적입니다.
- 빠른 기술 실험 및 프로토타이핑: LangFlow / Flowise는 최신 LLM 기술을 가장 빨리 써볼 수 있지만, 운영 환경(Production)으로 가져가기 전에는 철저한 에러 핸들링 코드 보강과 모니터링 시스템 구축이 선행되어야 합니다.

7장. Kubernetes 통합 관점 최종 선정 기준 및 참조 아키텍처

본 장에서는 앞서 분석한 기술적 특성과 엔터프라이즈 요구사항을 종합하여, 폐쇄망(Air-gapped) Kubernetes 환경에 최적화된 AI Agent Builder 선정 기준을 정량화하고, 이를 기반으로 한 '구축형 AI 번들 플랫폼'의 참조 아키텍처를 제시한다.

선정 기준은 단순한 기능의 유무(Feature Checklist)가 아니라, '고객사에 납품하여 지속 가능한 운영이 가능한가'에 초점을 맞춘다.

7.1 최종 선정 기준 (Scorecard)

후보 솔루션(n8n, LangFlow, Flowise, Dify)에 대해 폐쇄망 구축성, 연동성, 운영성, 제품화 난이도의 4가지 차원에서 평가를 수행한다. 평가는 5점 척도(5: 우수/즉시 적용 가능, 1: 미흡/대규모 개발 필요)를 기준으로 한다.

7.1.1 폐쇄망 구축성 (설치/업데이트/의존성)

인터넷이 차단된 환경에서 Docker Image 반입 및 Helm Chart 기반 배포 시 발생하는 의존성 문제를 중점적으로 평가한다.

- 주요 평가 항목:
  - Docker Image 완결성: 외부 리포지토리(npm, pip, huggingface) 런타임 호출 차단 여부.
  - Helm Chart 성숙도: K8s 네이티브 배포 지원 및 설정 유연성(ConfigMap/Secret).
  - 오프라인 의존성 관리: Custom Node/Tool 추가 시 로컬 파일시스템/사설 저장소 지원 여부.

솔루션	점수	평가 상세
n8n	4.5	단일/소수 컨테이너로 구성되어 배포가 매우 용이함. 대부분의 로직이 내장되어 있어 런타임 다운로드가 적으나, 커뮤니티 노드 추가 시 주의 필요.
Dify	3.5	아키텍처가 복잡함(Frontend, API, Worker, Plugin, DB, Redis, Sandbox 등 9+ 컨테이너). Helm Chart는 제공되나 폐쇄망 환경에서 Sandbox(SSRF 방지) 구성 난이도가 높음.
Flowise	4.0	Node.js 기반의 경량 컨테이너 제공. 비교적 의존성이 적으나, 특정 LangChain 도구 사용 시 추가 라이브러리 설치가 필요할 수 있음.
LangFlow	3.0	Python 기반으로 `pip` 의존성이 강함. 폐쇄망 내 사설 PyPI 미러가 없으면 커스텀 컴포넌트 확장이 매우 까다로움.

7.1.2 연동성 (MCP/크롤러/검색/DB) 및 확장 방식

기업 내부의 레거시 시스템 및 데이터 파이프라인(Crawler, VectorDB)과의 표준화된 연결 방식을 평가한다. 특히 MCP(Model Context Protocol) 지원 여부는 향후 확장성의 핵심 지표이다.

- 주요 평가 항목:
  - MCP 지원: 표준화된 방식의 외부 툴/데이터 연결 지원 여부.

- RAG 파이프라인: VectorDB(Qdrant 등) 및 GraphDB(Neo4j) 네이티브 커넥터 품질.
- API 유연성: Crawler4AI, OCR 엔진 등 외부 마이크로서비스 호출의 용이성.

솔루션	점수	평가 상세
Dify	5.0	가장 강력한 RAG 파이프라인 내장 (Qdrant, Milvus 등 네이티브 지원). Crawler, 리랭킹 모델 등 전처리 도구가 플랫폼 레벨에서 통합되어 있음. MCP 지원 계획이 구체적임.
n8n	4.0	'HTTP Request' 노드를 통해 모든 API 연동 가능(가장 유연함). 단, VectorDB/RAG 로직을 사용자가 직접 설계해야 하므로 공수가 듦.
Flowise	4.0	LangChain 생태계의 모든 Vector Store 지원. GraphDB(Neo4j) 체인 구성이 용이함. MCP 등 최신 스펙 반영 속도가 빠름.
LangFlow	3.5	LangChain 파이썬 라이브러리 의존. 기능은 강력하나, 비개발자가 복잡한 RAG 파이프라인(Graph+Vector Hybrid)을 구성하기엔 진입 장벽이 높음.

7.1.3 운영성 (모니터링/감사/재현)과 제품화 난이도

시스템 납품 후 고객사가 자체적으로 운영 및 감사를 수행할 수 있는지, 그리고 라이선스 이슈 없이 제품에 번들링(Bundling)할 수 있는지를 평가한다.

- 주요 평가 항목:
  - Observability: 실행 이력(Trace), 토큰 사용량, 입출력 로그의 UI 시각화 및 재현(Debug) 기능.
  - 라이선스: 상용 제품 포함 배포(Redistribution) 가능 여부.
  - 멀티 테넌시: 부서별/사용자별 작업 공간 격리 및 권한 관리.

솔루션	점수	평가 상세
Dify	4.5	운영성 최상. 로그, 어노테이션 (Annotation), 사용자 관리, 토큰 분석 기능 내장. Apache 2.0 라이선스로 번들링 용이. 단, 무거운 리소스가 단점.
Flowise	4.0	실행 이력 확인 가능하나 엔터프라이즈급 감사(Audit) 기능은 부족. Apache 2.0/ MIT 계열로 라이선스 리스크 낮음.
LangFlow	3.0	로그 및 모니터링 기능이 개발자 디버깅 수준에 머무름. 운영자(Ops) 관점의 대시보드 부재. MIT 라이선스.
n8n	2.0	라이선스 치명적. 'Sustainable Use License'는 내부 업무용 사용은 무료이나, 상용 제품에 포함하여 배포(SaaS/ On-prem)할 경우 별도 상용 계약 필수. 운영/디버깅 UI는 훌륭하나 제품화 시 비용 리스크 큼.

7.2. Kubernetes 구성 전략 (단일 vs 조합)

평가 결과, 단일 솔루션만으로는 '워크플로우 자동화'와 'LLM 앱 운영'을 모두 완벽히 충족하기 어렵다. 따라서 다음과 같은 번들링 전략을 수립한다.

7.2.1 단일 Agent Builder로 충족되지 않는 영역의 보완 방식

- 전략: Core Platform + Microservice Toolkit 구조
- Core Platform 선정: Dify (또는 Flowise)
  - 이유: 최종 사용자(End-user)에게 제공할 UI/UX, RAG 파이프라인, 권한 관리, 로그 감사가 가장 완성되어 있음. n8n은 라이선스 문제로 제외하거나 별도 계약 필요.
- 보완 방식 (Microservice Toolkit):

- Agent Builder 내에서 복잡한 전처리를 모두 수행하지 않고, 기능별 전용 컨테이너(Pod)를 K8s에 배포하여 API로 호출한다.
- Crawler: `Crawler4AI` 기반의 전용 수집기 컨테이너 (Headless Browser 포함).
- OCR/Parsing: 비정형 문서 파싱 전용 서비스 (GPU 할당).
- Interface: Agent Builder는 이들 마이크로서비스를 `Custom Tool` 또는 `HTTP Request`로 오케스트레이션하는 역할에 집중한다.

7.2.2 표준 툴 체계 (사내 Tool 카탈로그)와 재사용 구조

고객사 개발자가 매번 API 연동 코드를 작성하지 않도록 표준 툴 카탈로그(Standard Tool Catalog)를 제공해야 한다.

1. OpenAPI Specification (Swagger) 기반 도구 정의:
   - 사내 레거시 시스템(ERP, HR 등)의 API를 OpenAPI Spec으로 정의하여 Import 하면, Agent Builder가 자동으로 Tool로 인식.
2. MCP(Model Context Protocol) 서버 구축:
   - 사내 데이터 조회용 MCP 서버를 별도 Pod로 띄우고, Agent Builder는 이를 MCP Client로서 연결.
   - 이를 통해 DB 스키마 변경 시 Agent 로직 수정 없이 MCP 서버만 업데이트하면 되는 느슨한 결합(Loose Coupling) 구현.

7.2.3 고객사별 커스터마이징을 최소화하는 템플릿 전략

구축 시마다 바닥부터 개발하는 SI성 사업을 지양하기 위해 템플릿(DSL) 기반 배포를 수행한다.

- DSL(Domain Specific Language) Export/Import:
  - 선정된 빌더(Dify/Flowise)의 워크플로우를 YAML/JSON 형태로 형상 관리(Git)한다.
  - ConfigMap 주입: K8s 배포 시 고객사별 환경변수(DB 접속정보, API Key, 조직명)를 ConfigMap으로 분리하여, 동일한 DSL 파일이 환경에 따라 동적으로 설정되도록 구성한다.

- Pre-built Agent Pack:
  - ‘IT 헬프데스크’, ‘규정 검색’, ‘회의록 요약’ 등 공통 유즈케이스를 사전에 빌드하여 ‘블루프린트’ 형태로 번들에 포함시킨다.

7.3 참조 아키텍처 (Reference Architecture)

MSAP(Microservices Architecture Platform) 기반의 COP(Container Orchestration Platform) 환경을 가정한 최종 참조 아키텍처는 다음과 같다.

7.3.1 Kubernetes 상 배포 청사진 (네임스페이스/서비스/워커)

전체 시스템은 논리적으로 격리된 Namespace 내에 구성되며, 보안을 위해 Control Plane과 Data Plane을 분리한다.

- Namespace: `ai-platform-system`
- Ingress Controller:
  - 외부/내부 트래픽 라우팅, SSL Termination, IP 화이트리스트 관리.
- Service Layer (Agent Builder):
  - Web/API Pod: 사용자 UI 및 API 게이트웨이 (Replicas: 2+).
  - Worker Pod: 긴 시간이 소요되는 Agent 추론 및 Tool 실행 담당 (Celery/BullMQ 기반 비동기 처리).
- Data Persistence Layer:
  - PostgreSQL: 사용자 정보, 워크플로우 메타데이터, 실행 로그 저장 (StatefulSet).
  - Redis: 캐시, Pub/Sub 메시지 브로커, 세션 관리.
  - S3 Compatible Storage (MinIO): 업로드 된 파일, 파싱된 문서 원본 저장.

7.3.2 데이터 파이프라인: Crawler → OCR → Embedding → Vector/Graph

Agent Builder가 통제하는 데이터 처리 흐름(DAG)은 다음과 같이 컨테이너 간 파이프라인으로 구성된다.

1. Ingestion (수집):
    - Agent가 `Trigger` → Crawler Pod (Crawler4AI) 호출 → 웹/사내 게시판 데이터 수집 (Markdown 변환).
2. Processing (전처리):
    - 수집된 데이터 → Parser Pod (OCR/Layout Analysis) → 텍스트 청킹(Chunking).
3. Embedding (벡터화):
    - 청크 데이터 → Embedding Pod (HuggingFace `BGE-m3` 등 로컬 모델 서빙) → 벡터 값 생성.
4. Indexing (저장):
    - 벡터 값 → VectorDB (Qdrant) 저장.
    - 문서 간 관계/메타데이터 → GraphDB (Neo4j) 저장.
5. Retrieval (검색 – Hybrid):
    - 사용자 질문 → Keyword(BM25) + Vector(Dense) + Graph(Relation) → Reranker Pod → 최종 컨텍스트 생성.

7.3.3 내부 LLM 연동 (vLLM/Ollama 등) 및 보안 정책 결합 지점

외부 LLM(OpenAI, Claude) 사용이 불가능한 폐쇄망 환경을 고려한 Local LLM Serving 구조이다.

- LLM Serving Layer:
    - Engine: `vLLM` (고성능 추론) 또는 `Ollama` (경량/관리 용이).

  - Resource: K8s GPU Node (NVIDIA A100/H100 MIG 분할 권장)에 DaemonSet 또는 Deployment로 배포.
  - Interface: OpenAI Compatible API (`/v1/chat/completions`)를 노출하여 Agent Builder가 표준 라이브러리로 호출 가능하도록 통일.

- Security Gateway (Guardrails):
  - Agent Builder와 LLM 사이에 AI Gateway를 배치.
  - 역할: PII(개인정보) 마스킹, 프롬프트 인젝션 방지, 비속어 필터링, 부서별 토큰 사용량 제어(Rate Limiting).
  - 모든 프롬프트와 답변은 감사 로그(Audit Log)로 별도 저장되어 보안 감사에 대응한다.

[7장 요약 결론]

최종적으로 엔터프라이즈 폐쇄망 환경을 위한 번들 전략은 “운영성이 검증된 Dify(또는 Flowise)를 UI/Orchestrator로 채택하고, Crawler/OCR/LLM을 독립적인 K8s 마이크로서비스로 배포하여 표준 프로토콜(API/MCP)로 느슨하게 결합하는 구조”가 가장 적합하다. 특히 n8n은 강력하나 라이선스 이슈로 번들링 시 제외하거나 별도 검토가 필요함을 명시한다.

References & Links

- Loris – Conversation Intelligence for Leading Brands – https://loris.ai/
- MIT Study: 95% of AI Projects Fail. Here’s How to Be The 5%. – https://loris.ai/blog/mit-study-95-of-ai-projects-fail/
- AI Agents Must Act, Not Wait: A Case for Event-Driven Multi-Agent Design – https:// seanfalconer.medium.com/ ai-agents-must-act-not-wait-a-case-for-event-driven-multi-agent-design-d8007b50081f
- Sustainable Use License | n8n Docs – `https://docs.n8n.io/sustainable-use-license/`

- GitHub - FlowiseAI/Flowise: Build AI Agents, Visually - https://github.com/FlowiseAI/Flowise
- Flowise/LICENSE at master · FlowiseAI/Flowise · GitHub - https://github.com/FlowiseAI/Flowise/blob/master/LICENSE
- GitHub - langflow-ai/langflow: Langflow is a powerful tool for building and deploying AI-powered agents and workflows. - https://github.com/langflow-ai/langflow
- langflow/LICENSE at dev · langflow-ai/langflow · GitHub - https://github.com/langflow-ai/langflow/blob/dev/LICENSE
- GitHub - langgenius/dify: Production-ready platform for agentic workflow development. - https://github.com/langgenius/dify
- dify/LICENSE at main · langgenius/dify · GitHub - https://github.com/langgenius/dify/blob/main/LICENSE

Contact Us

hello@cncf.co.kr

02-469-5426

www.cncf.co.kr

CNF Blog

다양한 콘텐츠와 전문 지식을 통해
더 나은 경험을 제공합니다.

CNF eBook

이제 나도 클라우드 네이티브 전문가
쿠버네티스 구축부터 운영 완전 정복

CNF Resource

Community Solution의 최신 정보와
유용한 자료를 만나보세요.

씨엔에프 | CNF

전화 : (02) 469 - 5426
팩스 : (02) 469 - 7247
메일 : hello@cncf.co.kr