MODULE 04

Ollama, llama.cpp, vLLM — 뭘 언제 쓰나

모델 파일을 실제로 돌려주는 프로그램인 서빙 엔진을 비교합니다. 세 가지 대표 엔진이 각각 어떤 상황에 맞는지 정리하고, 우리 연구회의 서버 구성이 왜 그렇게 짜여 있는지도 확인합니다.

01 서빙이란 무엇인가

모델 파일(GGUF)은 그 자체로는 그냥 커다란 파일에 불과합니다. 이 파일을 메모리에 올리고, 채팅창이나 **API** 요청을 받아 답변을 만들어 돌려주는 프로그램이 바로 **서빙 엔진(serving engine)**입니다. API란 프로그램끼리 정해진 형식으로 요청과 답을 주고받는 창구입니다 — 자판기에 정해진 버튼을 누르면 정해진 물건이 나오듯, 정해진 형식으로 요청하면 정해진 형식의 답이 돌아옵니다. 더 자세히는 워밍업을 참고하세요.

> 💡 **비유**
> 모델 파일이 악보라면, 서빙 엔진은 그 악보를 연주하는 연주자입니다. 같은 악보라도 어떤 연주자가 맡느냐에 따라 필요한 무대(장비)의 조건과 동시에 받을 수 있는 관객(사용자) 수가 달라집니다.

서빙 엔진을 고를 때 자주 듣게 될 말이 **"OpenAI 호환 API"**입니다. ChatGPT API와 같은 형식의 주소로 요청을 받는다는 뜻으로, n8n(코드 없이 업무 자동화 흐름을 만드는 도구) 같은 외부 도구들이 주소만 바꿔 그대로 붙을 수 있다는 의미입니다.

02 llama.cpp — 로컬 추론의 본가

C++(프로그램을 만드는 데 쓰는 언어 중 하나)로 만들어진 **llama.cpp**는 로컬 추론 엔진의 사실상 표준이자, 앞서 배운 GGUF 포맷의 본가입니다.

가장 큰 강점은 **지원 폭**입니다. 최신 소비자 GPU는 물론 구형 GPU, 심지어 GPU 없이 CPU만으로도 실행할 수 있습니다. GPU 여러 장에 모델을 나눠 싣는 것도 가능합니다.

`llama-server` 실행 파일 하나만 띄우면 그대로 OpenAI 호환 API 서버가 되고, 도커(Docker — 프로그램과 실행 환경을 상자 하나로 포장해 어디서든 똑같이 돌게 하는 기술, 이 모듈 뒷부분에서 자세히)로도 배포할 수 있습니다.

용어 짚기 ① — 컴파일러와 "빌드"
llama.cpp의 "cpp"는 **C++**라는 프로그래밍 언어의 흔적입니다. 사람이 쓴 소스코드는 그 자체로는 실행되지 않습니다 — **컴파일러**가 이를 각 장비에 맞는 기계 명령으로 번역해야 하고, 이 번역 과정을 **빌드**라고 부릅니다. 같은 llama.cpp라도 어떤 CPU·어떤 GPU 세대용으로 빌드했느냐에 따라 돌아가는 곳과 속도가 달라지는 이유가 이것입니다. 비유하면 소스코드는 레시피, 컴파일러는 그 레시피를 우리 주방 기계어로 옮기는 번역가입니다.

용어 짚기 ② — SDK와 CUDA 툴킷
GPU용으로 번역하려면 NVIDIA가 제공하는 도구 상자가 따로 필요합니다. 이 도구 상자를 **CUDA 툴킷**이라 부르는데, 컴파일러·라이브러리(자주 쓰는 기능을 미리 만들어둔 부품 모음)·문서·예제를 한데 묶은 **SDK**(Software Development Kit, 개발 도구 모음)입니다. 비유를 이어가면, SDK는 특정 주방(NVIDIA GPU) 전용 조리도구 세트입니다.

실무 감각 하나: "서버에 도커 이미지를 빌드해뒀다"는 말은 이 번역을 이미 끝내 포장까지 해뒀다는 뜻입니다. 그래서 **한 번 검증된 이미지를 재사용하는 것**이 구형 서버 운영의 안전 수칙이 됩니다 — 매번 새로 빌드하면 컴파일러·SDK 버전 궁합 문제를 매번 다시 풉니다.

03 Ollama — 가장 쉬운 시작

Ollama는 llama.cpp 엔진을 감싸 `ollama pull 모델명`, `ollama run 모델명` 단 두 개의 명령으로 쓸 수 있게 만든 도구입니다. 이 "명령"은 브라우저 주소창이 아니라 **터미널(명령창)**이라는 별도 프로그램에 타이핑하는 것입니다 — 지금 당장 따라 하지 않아도 됩니다, 실습은 모임에서 함께 합니다. 터미널이 낯설다면 워밍업에서 먼저 익히고 오세요.

모델 저장, 전환, API 제공까지 전부 자동으로 처리해줘서 개인 실습의 표준으로 자리 잡았습니다. 다만 동시 접속 슬롯 수나 KV 캐시의 세밀한 제어 같은 **깊은 서버 설정은 llama-server를 직접 실행하는 것보다 제한적**입니다.

04 LM Studio — 코드 없이 화면 클릭만으로

LM Studio는 모델 검색부터 다운로드, 채팅까지 전부 화면 클릭만으로 끝낼 수 있는 도구입니다. 서버 기능을 켜는 것도 버튼 하나면 충분합니다.

코드를 다뤄본 적 없는 사람이 로컬 LLM을 처음 경험하기에 가장 알맞고, 6월 발표 때 핸즈온 자료로 소개한 그 도구입니다.

05 vLLM — 운영급, 단 신형 GPU 전제

vLLM은 연속 배칭(continuous batching — 여러 사용자의 요청을 묶어 한 번에 처리하는 기법, 자세한 설명은 이 모듈 뒷부분에서) 같은 고급 기법으로 수십에서 수백 명의 동시 사용자를 감당하도록 만들어진 고성능 서빙 엔진입니다.

다만 중요한 제약이 있습니다. **최신 세대 GPU를 전제로 개발되어, 파스칼 세대(2016년경 나온 구형 GPU 세대명)처럼 오래된 GPU는 지원하지 않습니다.**

> **교훈: 하드웨어 세대가 소프트웨어 선택지를 제약한다.**
> "무엇을 쓸 수 있는가"는 결국 장비를 보고 정해집니다. 서빙 계획을 세울 때 하드웨어 조사가 가장 먼저 오는 이유가 바로 이것입니다.

06 선택 가이드

지금까지 본 세 엔진을 상황별로 정리하면 다음과 같습니다.

상황	추천	이유
개인 PC에서 가볍게 실습 (GPU 없어도 됨)	Ollama 또는 LM Studio	설치·실행 최단 경로, CPU만으로 1B~4B급 구동 가능
팀이 공유하는 사내 서버(구형 GPU 포함)	llama.cpp의 llama-server	지원 폭·멀티 GPU·API 제공

신형 GPU 다수, 대규모 동시 사용자	vLLM	처리량 최적화

우리 연구회에선

우리 구성이 정확히 이 표 그대로입니다. 공유 서버는 llama-server(도커)로 OpenAI 호환 API를 제공하고, 여러분은 그 API에 접속해 실습합니다. 각자 사무실 PC에는 GPU가 없지만, Ollama나 LM Studio를 CPU 모드로 설치해 1B~4B급 소형 모델을 직접 받아 돌려보는 체험은 가능합니다 — 느린 이유까지 스스로 설명할 수 있다면 이 모듈을 제대로 이해한 겁니다.

07 엔진 너머 — LLM 서빙 생태계 지도

엔진 이름을 아는 것과 생태계 전체가 어떻게 맞물리는지 아는 것은 다릅니다. 로컬 LLM이 여러분 앞에 도착하기까지의 전체 경로를 한 층씩 쌓아보면 이렇습니다.

로컬 LLM 스택 — 다섯 층

① **모델 허브**: HuggingFace — 전 세계 오픈 모델의 배포처
② **모델 포맷**: GGUF(llama.cpp 계열용), AWQ/GPTQ(GPU 서빙용 양자화), safetensors(원본)
③ **서빙 엔진**: 아래 표 — 포맷과 하드웨어에 따라 선택지가 갈림
④ **API 표준**: OpenAI 호환 — 엔진이 달라도 도구는 그대로 붙는 공용 규격
⑤ **배포·운영**: Docker(환경 고정·이식), 필요 규모에선 k8s(여러 서버 오케스트레이션)

⑤층의 용어 두 개만 짚고 갑시다. **Docker(도커)**는 프로그램과 그 실행 환경(라이브러리, 설정)을 통째로 상자 하나에 포장하는 기술입니다. "내 컴퓨터에선 되는데 저 서버에선 안 돼요" 문제를 없애주기 때문에, 서버에 서빙 엔진을 올릴 때 사실상 표준으로 쓰입니다. **k8s(쿠버네티스)**는 그런 상자를 수십~수백 개 규모로 여러 서버에 자동 배치·복구하는 도구인데, 우리 규모(서버 1대)에서는 아직 필요 없습니다. 그리고 vLLM 설명에 나온 **배칭(batching)**은 여러 사용자의 요청을 묶어 GPU가 노는 시간 없이 한꺼번에 처리하는 기법입니다 — 동시 사용자가 많을수록 처리량을 지키는 핵심 기술입니다.

엔진 층을 조금 더 넓게 보면, 이 판에는 저마다의 전문 분야를 가진 선수가 더 있습니다. 아래 표는 훑어보기용입니다 — SGLang·TensorRT-LLM 같은 이름까지 외울 필요는 없고, "이런 전문 엔진들도 있다" 정도만 눈에 담아두면 충분합니다.

엔진	주특기	전제 조건
llama.cpp	지원 폭(구형 GPU·CPU·멀티 GPU), GGUF의 본가	거의 없음 — 그래서 우리의 기본기
Ollama / LM Studio	개인 실습 편의 (llama.cpp 기반)	거의 없음
vLLM	대규모 동시 사용자 처리량	신형 GPU
SGLang	구조화된 출력·에이전트 워크로드 고속 처리	신형 GPU
TGI	HuggingFace 생태계 통합 운영	비교적 신형 GPU
TensorRT-LLM	NVIDIA 칩 한계 성능 추출	NVIDIA 신형 + 전담 인력

표가 길어 보여도 판단 구조는 단순합니다. **내 하드웨어가 선택지를 먼저 거르고(모듈 1), 남은 것 중 운영 요구(동시 사용자·처리량)가 최종 선택을 정합니다.** 그리고 어느 엔진을 고르든 API 표준 층이 같기 때문에, 이 API 위에 올리게 될 워크플로우·에이전트(다음 모듈에서 배웁니다)는 엔진을 갈아타도 거의 그대로 살아남습니다. 이것이 생태계가 표준으로 수렴하면서 생긴 실무적 이점입니다.

↗ 심화 학습

서빙·운영을 더 깊게: **위키독스 「Just Read, LLMOps」** — RAG·서빙(vLLM/TGI)·평가까지 다루는 무료 한국어 실전서(중급). · 엔진의 원전: **llama.cpp 공식 저장소** (영어 — 하드웨어별 지원 범위·옵션의 공식 문서)

CHECK

이해했는지 확인해 봅시다

틀려도 괜찮습니다 — 오답을 고르면 왜 아닌지 설명이 나오고, 정답을 찾을 때까지 다시 고를 수 있습니다. 점수는 첫 시도 기준입니다.

문제 1 / 6

Q1. '서빙 엔진'의 역할은?

① 모델 소스코드를 장비에 맞게 컴파일(빌드)한다

② 모델 파일을 메모리에 올려 채팅·API 요청을 받아 처리한다

③ 모델 파일을 압축(양자화)해 가볍게 만든다

④ 질문의 종류를 보고 알맞은 모델을 골라 연결해준다

← 이전
모듈 3 · 양자화

다음 →
모듈 5 · RAG와 에이전트