Local AI, Agents, and the `gpt-oss-20b` Stack

Building Useful AI Tools with Privacy and Control

Hyun-Hwan Jeong

January 22, 2026

Welcome

What You'll Build Today

```python
# Before: Dependent on APIs
client = OpenAI(api_key="sk-proj-...")

# After: Independent Infrastructure
client = OpenAI(base_url="http://localhost:8080/v1")
```

You will:

- Run 20B parameter models on your laptop
- Build tool-calling agents from scratch
- Deploy autonomous coding assistants
- Save thousands in API costs

Foundation

The Model Landscape

Size	Characteristics	Hardware Reality
8B	Fast, capable of basic tools, but lacks nuance.	Runs on any modern GPU/CPU.
20B–35B	The Sweet Spot. Reliable reasoning & instruction following.	Fits on a single high-end consumer GPU (24GB VRAM).
70B+	State-of-the-art knowledge & logic.	Dual consumer GPUs or Mac Studio required (not just A100).

`gpt-oss-20b` **is the sweet spot:**

- Fits on consumer hardware (with MoE and quantization)
- Production-grade reasoning (comparable with o3-mini)
- Apache 2.0 license (making it permissible for commercial use, modification, and distribution.)

Hardware Requirements

Option 1 (NVIDIA):

- RTX 3090/4090 (24GB VRAM)
- Inference: ∼40 tokens/sec

Option 2 (Apple Silicon):

- M2/M3 with 32GB+ RAM
- Inference: ∼25 tokens/sec, can be better with llama.cpp with quantization

Option 3 (CPU Only):

- 32GB RAM minimum
- Inference - slow but works: ∼5-10 tokens/sec

Key Privacy Benefit: Your data **never leaves your machine**. No API calls, no cloud logging, complete data control.

Installing llama.cpp

Method 1: Pre-built Binaries (Fast)

```
# macOS (Apple Silicon)
brew install llama.cpp

# Linux/macOS (Manual)
wget https://github.com/ggerganov/llama.cpp/releases/\
  latest/download/llama-cli-linux-x64.zip
unzip llama-cli-linux-x64.zip
chmod +x llama-cli
```

Method 2: Build from Source (For GPU)

```
git clone https://github.com/ggerganov/llama.cpp
cd llama.cpp
make LLAMA_CUDA=1  # CUDA support
make LLAMA_METAL=1  # Metal (macOS)
```

Downloading the Model

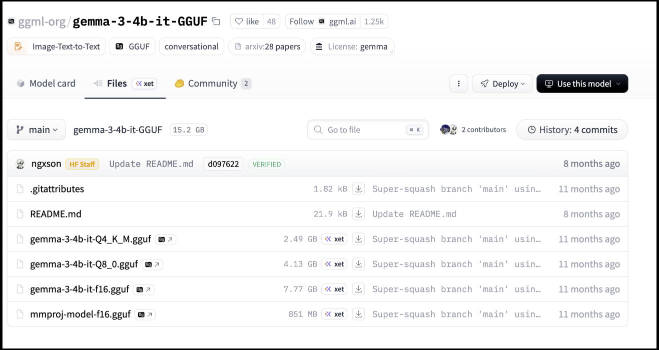
```
# Using huggingface-cli (recommended)
pip install huggingface-hub

hf download \
  ggml-org/gemma-3-4b-it-GGUF \
  gemma-3-4b-it-Q4_K_M.gguf \
  --local-dir ./models

```

- **Expected size:** ∼2. GB
- **Download time:** 1 minute (depends on connection)

Quantization: Speed vs Accuracy


ggml-org/gemma-3-4b-it-GGUF

gemma-3-4b-it-GGUF 15.2 GB

ngxson HF Staff Update README.md d097622 VERIFIED 8 months ago

.gitattributes	1.82 kB	Super-squash branch 'main' usin…	11 months ago
README.md	21.9 kB	Update README.md	8 months ago
gemma-3-4b-it-Q4_K_M.gguf	2.49 GB xet	Super-squash branch 'main' usin…	11 months ago
gemma-3-4b-it-Q8_0.gguf	4.13 GB xet	Super-squash branch 'main' usin…	11 months ago
gemma-3-4b-it-f16.gguf	7.77 GB xet	Super-squash branch 'main' usin…	11 months ago
mmproj-model-f16.gguf	851 MB xet	Super-squash branch 'main' usin…	11 months ago

- Quantization reduces model size and increases throughput.
- Higher-precision quantization (e.g., Q4_K_M) preserves accuracy better but is slower and uses more memory.
- Lower-precision quantization (e.g., Q4_K_S or Q2) is faster and smaller but can

Lab 1: First Inference

Running the Model (CLI Mode)

```
./llama-cli \
  -m models/gemma-3-4b-it-Q4_K_M.gguf \
  -p "Write a Python function to check if a number is prime" \
  -n 512 \
  -ngl 99 \
  -c 4096
```

Flag breakdown:

- `-m`: Model path
- `-p`: Prompt
- `-n`: Max tokens to generate
- `-ngl`: GPU layers (99 = all)
- `-c`: Context window

Server Mode: The Game Changer

Starting llama-server

```
./llama-server \
  -m models/gemma-3-4b-it-Q4_K_M.gguf \
  --host 0.0.0.0 \
  --port 8080 \
  -ngl 99 \
  -c 8192

# Server running at http://localhost:8080
```

Server Mode: The Game Changer

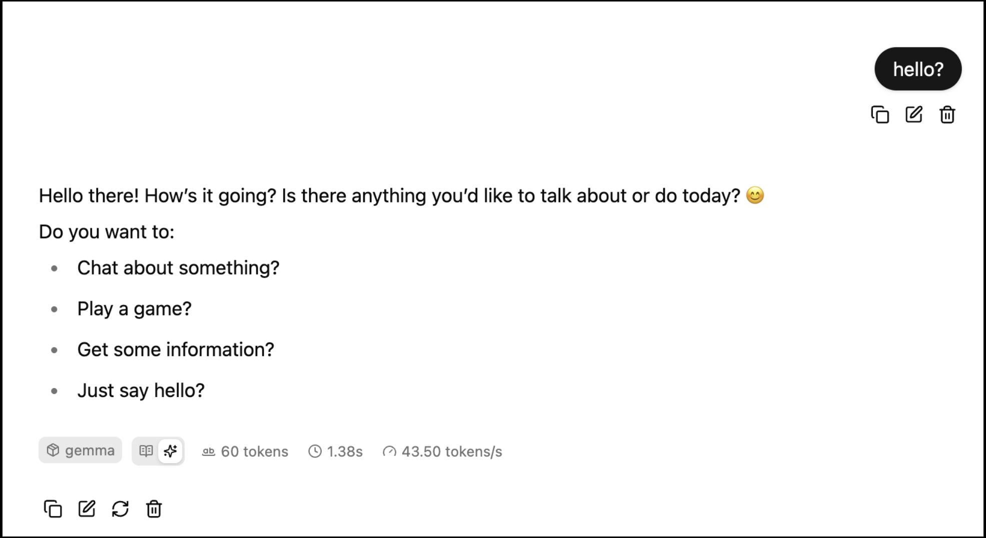
Starting llama-server (gpt-oss-20b)

```
./llama-server \
  llama-server --gpt-oss-20b-default --port 8080 -a gpt-oss-20b

# --gpt-oss-20b-default                   use gpt-oss-20b (note: can download weights from the
↪  internet)
# --gpt-oss-120b-default                  use gpt-oss-120b (note: can download weights from the
↪  internet)
```

Now your model is an API server!

`llama-server` Features - OpenWebUI Integration



hello?

Hello there! How's it going? Is there anything you'd like to talk about or do today? 😊

Do you want to:

- Chat about something?
- Play a game?
- Get some information?
- Just say hello?

Testing the Server

```
curl http://localhost:8080/v1/chat/completions \
  -H "Content-Type: application/json" \
  -d '{
    "model": "gpt-oss-20b",
    "messages": [
      {"role": "user", "content": "Say hello"}
    ]
  }'
```

Response:

```
{
  "id": "chatcmpl-abc123",
  "object": "chat.completion",
  "choices": [{
    "message": {"role": "assistant", "content": "Hello!"}
  }]
}
```

Python Integration

The Drop-In Replacement Pattern

```python
from openai import OpenAI

# Option 1: OpenAI Cloud
client_cloud = OpenAI(api_key="sk-proj-xxx")

# Option 2: Local llama.cpp server
client_local = OpenAI(
    base_url="http://localhost:8080/v1",
    api_key="not-needed"  # llama.cpp ignores this
)

# Same interface, different backend!
```

Lab 2: Your First Local Client

```python
from openai import OpenAI

client = OpenAI(
    base_url="http://localhost:8080/v1",
    api_key="local"
)

response = client.chat.completions.create(
    model="gpt-oss-20b",
    messages=[
        {"role": "system", "content": "You are a helpful assistant."},
        {"role": "user", "content": "Write a function to reverse a string."}
    ],
    temperature=0.7,
    max_tokens=500
)

print(response.choices[0].message.content)
```

Run it: `python test_local.py`

Streaming Responses

```python
from openai import OpenAI

client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")

stream = client.chat.completions.create(
    model="gpt-oss-20b",
    messages=[{"role": "user", "content": "Explain async/await"}],
    stream=True
)

for chunk in stream:
    if chunk.choices[0].delta.content:
        print(chunk.choices[0].delta.content, end="", flush=True)
```

Temperature and Sampling

```python
# Temperature: 0.0 (Deterministic)
response = client.chat.completions.create(
    model="gpt-oss-20b",
    messages=[{"role": "user", "content": "What is 2+2?"}],
    temperature=0.0  # Always same answer
)

# Temperature: 1.0 (Creative)
response = client.chat.completions.create(
    model="gpt-oss-20b",
    messages=[{"role": "user", "content": "Write a creative story"}],
    temperature=1.0  # More variety
)
```

- **For code:** Use 0.2-0.4
- **For creative writing:** Use 0.7-1.0

Recommended Inference Settings - for `gpt-oss-20b`

- **Sampling:** `temperature=1.0`, `top_p=1.0`, `top_k=0` (or try `top_k=100` for experimentation)
- **Context:** Recommended minimum context: 16 384
- **Max window:** Maximum context length: 131 072

Lab 3: Chat Loop with Memory

```python
from openai import OpenAI

client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")

messages = [
    {"role": "system", "content": "You are a helpful assistant."}
]

while True:
    user_input = input("You: ")
    if user_input.lower() == "exit": break
    messages.append({"role": "user", "content": user_input})

    response = client.chat.completions.create(
        model="gpt-oss-20b", messages=messages,
    )

    assistant_msg = response.choices[0].message.content
    messages.append({"role": "assistant", "content": assistant_msg})
    print(f"Assistant: {assistant_msg}\n")
```

Tool Calling & Agents

What is Tool Calling?

Traditional LLM:

- User: "What's the weather in NYC?"
- LLM: "I don't have access to real-time data..."

Tool-Calling LLM:

- User: "What's the weather in NYC?"
- LLM: $\rightarrow$ Calls `weather_api("NYC")`
- LLM: "It's 72°F and sunny in New York City."

The LLM decides WHEN and HOW to use tools.

Tool Schema Structure

```python
tools = [
    {
        "type": "function",
        "function": {
            "name": "get_weather",
            "description": "Get current weather for a city",
            "parameters": {
                "type": "object",
                "properties": {
                    "city": {"type": "string", "description": "City name"},
                    "units": {"type": "string", "enum": ["celsius", "fahrenheit"]}
                },
                "required": ["city"]
            }
        }
    }
]
```

Implementing Python Functions

```python
def get_weather(city: str, units: str = "fahrenheit") -> dict:
    """Mock weather API call."""
    weather_data = {
        "New York": {"temp": 72, "condition": "Sunny"},
        "London": {"temp": 15, "condition": "Rainy"},
        "Tokyo": {"temp": 28, "condition": "Cloudy"}
    }

    data = weather_data.get(city, {"temp": 20, "condition": "Unknown"})

    if units == "celsius":
        data["temp"] = int((data["temp"] - 32) * 5/9)

    return {
        "city": city,
        "temperature": data["temp"],
        "condition": data["condition"],
        "units": units
    }
```

Tool-Calling Request

```python
from openai import OpenAI

client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")

response = client.chat.completions.create(
    model="gpt-oss-20b",
    messages=[
        {"role": "user", "content": "What's the weather in New York?"}
    ],
    tools=tools,
    tool_choice="auto"  # Let model decide when to use tools
)

print(response.choices[0].message)
```

Lab 4: Complete Tool-Calling Agent

```python
available_functions = {"calculate": calculate}

def run_agent(user_query: str):
    import json
    from openai import OpenAI
    messages = [
        {"role": "system", "content": "You are a math assistant."},
        {"role": "user", "content": user_query}
    ]

    client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")

    while True:
        response = client.chat.completions.create(
            model="gpt-oss-20b", messages=messages,
            tools=tools, tool_choice="auto"
        )

        message = response.choices[0].message
        messages.append(message)

        if not message.tool_calls:
```

Agent Loop Implementation

```python
def run_agent(user_query: str):
    messages = [
        {"role": "system", "content": "You are a math assistant."},
        {"role": "user", "content": user_query}
    ]

    while True:
        response = client.chat.completions.create(
            model="gpt-oss-20b", messages=messages,
            tools=tools, tool_choice="auto"
        )

        message = response.choices[0].message
        messages.append(message)

        if not message.tool_calls:
            return message.content

        for tool_call in message.tool_calls:
            func_name = tool_call.function.name
            func_args = json.loads(tool_call.function.arguments)
            result = available_functions[func_name](**func_args)
            messages.append({"role": "tool", "tool_call_id": tool_call.id,
                             "content": json.dumps(result)})
```

Running the Agent

```python
# If the script doesn't expose an entrypoint, add a simple main:

def main():
    result = run_agent("What is 2 + 2 * 3?")
    print(result)

if __name__ == "__main__":
    main()
```

Advanced Agent Patterns

Multi-Tool Agent: File System Navigator

```python
def list_directory(path: str) -> dict:
    try:
        items = os.listdir(path)
        return {"path": path, "items": items, "count": len(items)}
    except Exception as e:
        return {"error": str(e)}

def read_file(filepath: str) -> dict:
    try:
        with open(filepath, 'r') as f:
            return {"filepath": filepath, "content": f.read()}
    except Exception as e:
        return {"error": str(e)}

def run_command(command: str) -> dict:
    try:
        result = subprocess.run(command, shell=True,
                                capture_output=True, text=True)
        return {"stdout": result.stdout, "stderr": result.stderr}
    except Exception as e:
        return {"error": str(e)}
```

Lab 5: Autonomous Code Analyzer

```python
def code_analyzer_agent(task: str, max_iterations=10):
    messages = [{
        "role": "system",
        "content": """You are a code analysis agent. You can:
- list_directory: Browse folders
- read_file: Read source code
- run_command: Run tests or linters"""
    }, {"role": "user", "content": task}]

    for iteration in range(max_iterations):
        response = client.chat.completions.create(
            model="gpt-oss-20b", messages=messages,
            tools=tools, tool_choice="auto"
        )

        message = response.choices[0].message
        if not message.tool_calls:
            return message.content

        # Execute tools and continue...
```

Model Context Protocol (MCP)

What is MCP?

Model Context Protocol enables AI models to:

- Access external tools and data sources
- Follow standardized interfaces
- Integrate with your applications

Why use FastMCP?

- Minimal boilerplate code
- Decorator-based tool definition
- Built-in HTTP client with retry logic
- Perfect for data fetching services

Building Your First MCP Server

```python
import httpx
import xml.etree.ElementTree as ET
from fastmcp import FastMCP

mcp = FastMCP("PMC Fetching Test")

@mcp.tool
def get_abstract_by_pmcid(pmcid: str = None) -> str:
    url = f"https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pmc&id={pmcid}"

    response = httpx.get(url, timeout=10.0)
    response.raise_for_status()

    root = ET.fromstring(response.content)
    abstract_elem = root.find(".//abstract")

    return "".join(abstract_elem.itertext())

if __name__ == "__main__":
    mcp.run()


```

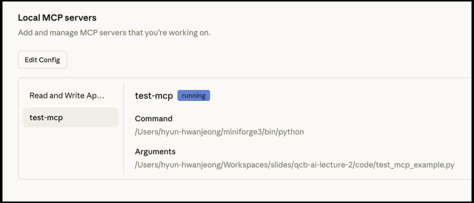
Connecting to Claude Desktop

Configuration file: `~/Library/Application Support/Claude/claude_desktop_config.json`

```json
{
  "mcpServers": {
    "test-mcp": {
      "command": "/Users/hyun-hwanjeong/miniforge3/bin/python",
      "args": [
        "/Users/hyun-hwanjeong/Workspaces/slides/qcb-ai-lecture-2/code/26_test_mcp_example.py"
      ]
    }
  }
}
```

- Claude Desktop acts as the **MCP Client**.
- It automatically spawns the server process and connects via stdio.
- Tools appear directly in the Claude UI.

MCP in Action



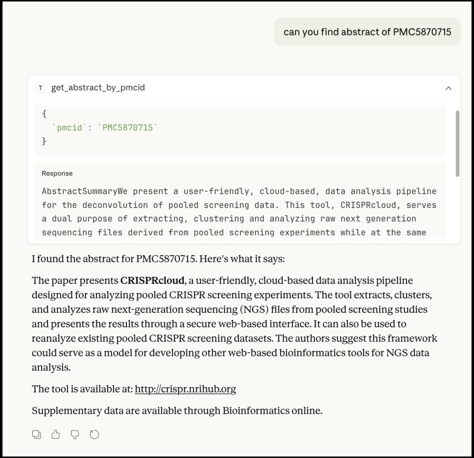
Local MCP servers

Add and manage MCP servers that you're working on.

test-mcp running

Command
/Users/hyun-hwanjeong/miniforge3/bin/python

Arguments
/Users/hyun-hwanjeong/Workspaces/slides/qcb-ai-lecture-2/code/test_mcp_example.py



can you find abstract of PMC5870715

get_abstract_by_pmcid

```
{
  `pmcid`: `PMC5870715`
}
```

Response

```
AbstractSummaryWe present a user-friendly, cloud-based, data analysis pipeline
for the deconvolution of pooled screening data. This tool, CRISPRcloud, serves
a dual purpose of extracting, clustering and analyzing raw next generation
sequencing files derived from pooled screening experiments while at the same
```

I found the abstract for PMC5870715. Here's what it says:

The paper presents **CRISPRcloud**, a user-friendly, cloud-based data analysis pipeline designed for analyzing pooled CRISPR screening experiments. The tool extracts, clusters, and analyzes raw next-generation sequencing (NGS) files from pooled screening studies and presents the results through a secure web-based interface. It can also be used to reanalyze existing pooled CRISPR screening datasets. The authors suggest this framework could serve as a model for developing other web-based bioinformatics tools for NGS data analysis.

The tool is available at: http://crispr.nrihub.org

Supplementary data are available through Bioinformatics online.

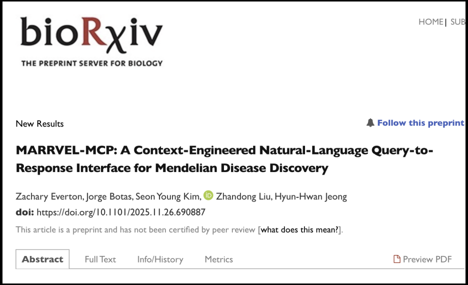
- **Left:** The MCP server is recognized and running in Claude's settings.
- **Right:** Claude uses the `get_abstract_by_pmcid` tool to answer a query.

MARRVEL-MCP: Rare Disease Discovery

Our Latest Work: MARRVEL-MCP — Everton et al. (2025, in revision)

Abstracting Complexity in Genetics:

- **Paper:** "A Context-Engineered Natural-Language Query-to-Response Interface for Mendelian Disease Discovery"
- **Impact:** Enables clinicians/researchers to query 35+ genetics databases using plain English.
- **Open Source:** Full MCP server implementation available on GitHub.



New Results

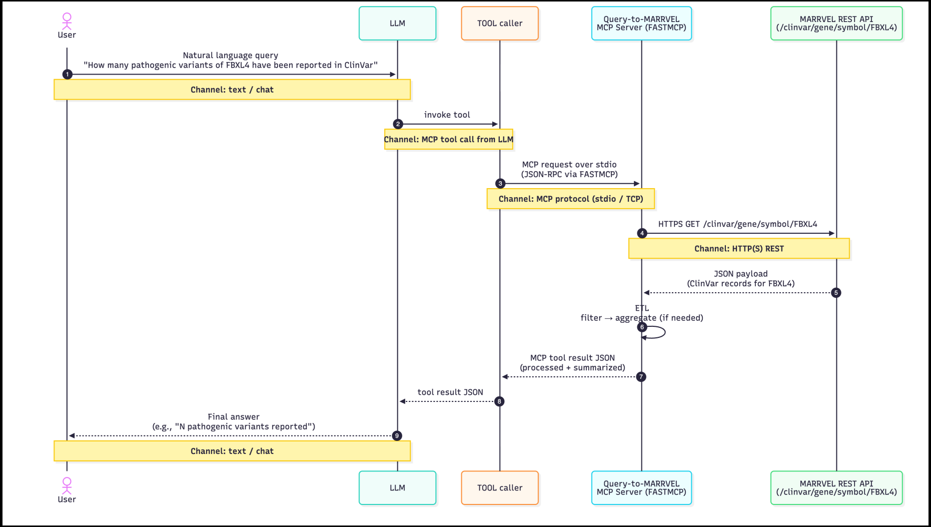
MARRVEL-MCP: A Context-Engineered Natural-Language Query-to-Response Interface for Mendelian Disease Discovery

Zachary Everton, Jorge Botas, Seon Young Kim, Zhandong Liu, Hyun-Hwan Jeong

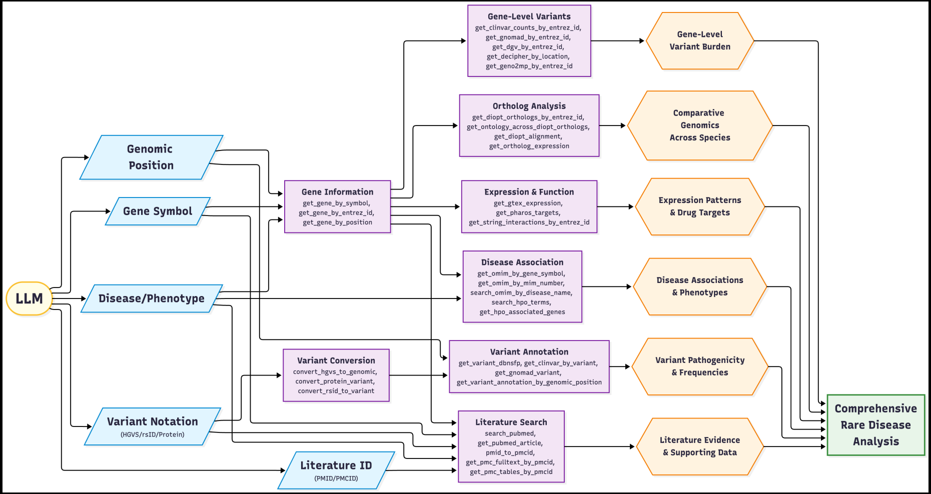
doi: https://doi.org/10.1101/2025.11.26.690887

This article is a preprint and has not been certified by peer review [what does this mean?].

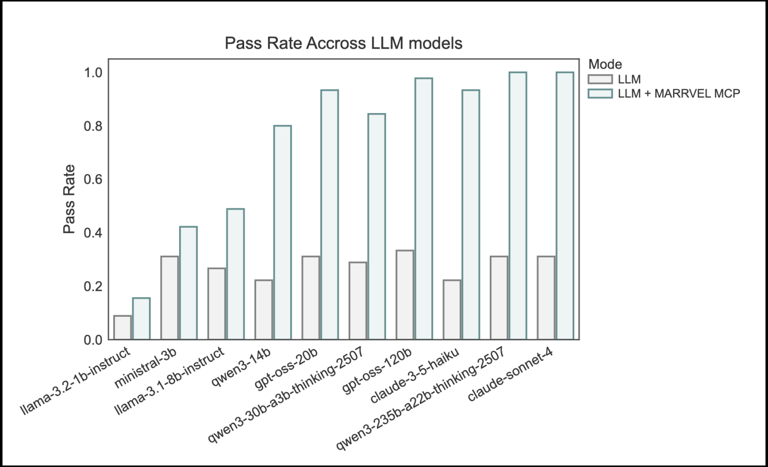
System Architecture


User
LLM
TOOL caller
Query-to-MARRVEL MCP Server (FASTMCP)
MARRVEL REST API (/clinvar/gene/symbol/FBXL4)
1 Natural language query "How many pathogenic variants of FBXL4 have been reported in ClinVar"
Channel: text / chat
2 invoke tool
Channel: MCP tool call from LLM
3 MCP request over stdio (JSON-RPC via FASTMCP)
Channel: MCP protocol (stdio / TCP)
4 HTTPS GET /clinvar/gene/symbol/FBXL4
Channel: HTTP(S) REST
5 JSON payload (ClinVar records for FBXL4)
6 ETL filter → aggregate (if needed)
7 MCP tool result JSON (processed + summarized)
8 tool result JSON
9 Final answer (e.g., "N pathogenic variants reported")
Channel: text / chat
User
LLM
TOOL caller
Query-to-MARRVEL MCP Server (FASTMCP)
MARRVEL REST API (/clinvar/gene/symbol/FBXL4)

The tool Landscape: 35+ Specialized Tools across six categories



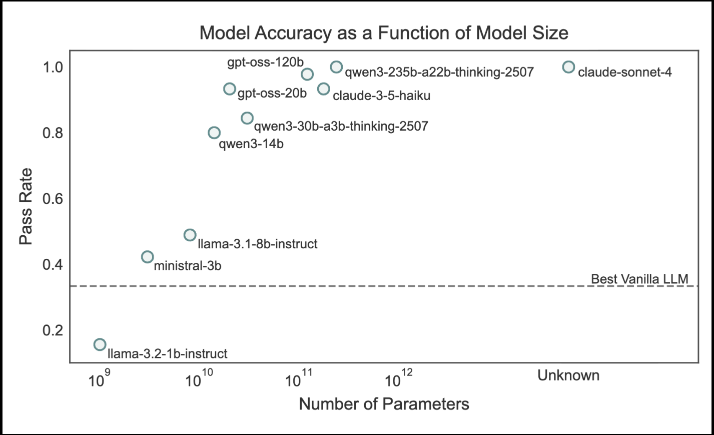
Benchmarking Performance



The "MCP Jump":

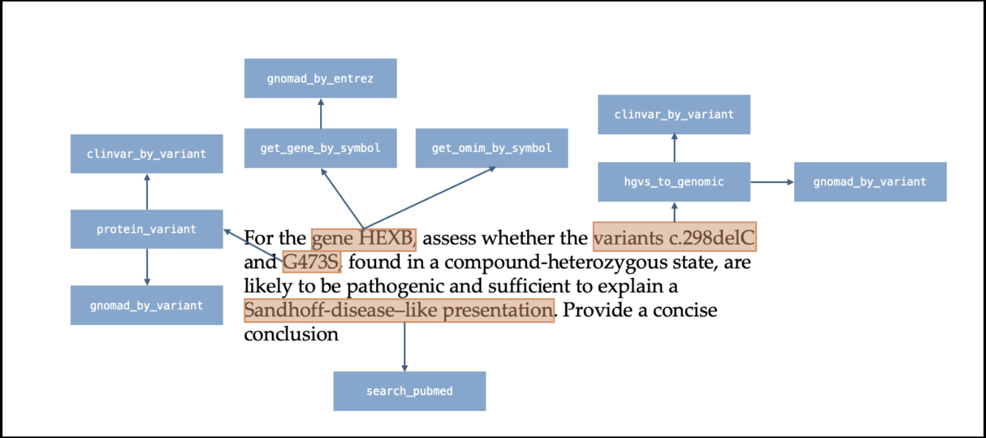
- Vanilla LLMs (grey) fail on >70% of complex genetics queries.
- LLM + MARRVEL-MCP (teal) achieves **near-perfect accuracy** (90-100%) for 20B+ models.
- Even small models (e.g., 3B) become significantly more useful with MCP.

Accuracy vs. Model Size



- **The Scaling Law:** MCP tools provide a massive baseline boost.
- `gpt-oss-20b` **(Local):** Outperforms the best "vanilla" cloud LLMs

Case Study: Complex Multi-Step Reasoning



For the gene HEXB, assess whether the variants c.298delC and G473S, found in a compound-heterozygous state, are likely to be pathogenic and sufficient to explain a Sandhoff-disease–like presentation. Provide a concise conclusion

Production Patterns

Configuration Management

```python
from dataclasses import dataclass
import os
from openai import OpenAI
@dataclass
class LLMConfig:
    base_url: str = "http://localhost:8080/v1"
    api_key: str = "local"
    model: str = "gpt-oss-20b"
    temperature: float = 1.0
    max_tokens: int = 2000

    @classmethod
    def from_env(cls):
        return cls(base_url=os.getenv("LLM_BASE_URL", cls.base_url),
                   model=os.getenv("LLM_MODEL", cls.model)
        )

config = LLMConfig.from_env()
client = OpenAI(base_url=config.base_url, api_key=config.api_key)
```

Lab 6: Agent Class

```python
class ProductionAgent:
    def __init__(self, config: AgentConfig):
        self.config = config
        self.client = OpenAI(base_url="http://localhost:8080/v1")
        self.tools = {}

    def register_tool(self, schema: dict, func: Callable):
        func_name = schema["function"]["name"]
        self.tools[func_name] = {"schema": schema, "func": func}

    def execute_tool(self, name: str, args: dict) -> dict:
        try:
            return self.tools[name]["func"](**args)
        except Exception as e:
            return {"error": str(e)}

    def run(self, task: str) -> str:
        # Agent loop with error handling
        # ... implementation
```

Deployment

Docker Deployment

```dockerfile
FROM nvidia/cuda:12.1.0-devel-ubuntu22.04

RUN apt-get update && apt-get install -y git build-essential

WORKDIR /app
RUN git clone https://github.com/ggerganov/llama.cpp.git
WORKDIR /app/llama.cpp
RUN make LLAMA_CUDA=1

EXPOSE 8080

CMD ["/app/llama.cpp/llama-server", \
     "-m", "/app/models/gpt-oss-20b.Q4_K_M.gguf", \
     "--host", "0.0.0.0", \
     "--port", "8080", \
     "-ngl", "99"]
```

Conclusion

What You've Learned

Technical Skills:

- Running 20B parameter models on consumer hardware
- Building OpenAI-compatible inference servers
- Implementing tool-calling agents from scratch
- Production patterns for error handling and logging
- Deployment strategies for local AI

Architecture Patterns:

- Drop-in replacement pattern (local API)
- ReAct agent loop implementation
- Model Context Protocol (MCP) for tool sharing
- Case Study: MARRVEL-MCP for clinical genetics
- Multi-tool coordination

What You've Learned

Privacy & Control:

- Keep sensitive data on your own hardware
- No vendor lock-in or dependency
- Full transparency and auditability

Cost Comparison

Local gpt-oss-20b:

- Unlimited tokens
- Break-even: ∼50M tokens
- **Privacy: Complete data control**

For heavy users: Local pays off in weeks! And your data is yours alone.

Resources

Essential Links:

- llama.cpp GitHub
- Model Download (HuggingFace)
- OpenAI Python SDK
- LangGraph for Agents

Community:

- r/LocalLLaMA
- HuggingFace

Thank You!

Questions?