

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

```
1 # Before: Dependent on APIs
2 client = OpenAI(api_key="sk-proj-...")
3
4 # After: Independent Infrastructure
5 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: $\sim 5-10$ 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)

```
1 # macOS (Apple Silicon)
2 brew install llama.cpp
3
4 # Linux/macOS (Manual)
5 wget https://github.com/ggerganov/llama.cpp/releases/\
6     latest/download/llama-cli-linux-x64.zip
7 unzip llama-cli-linux-x64.zip
8 chmod +x llama-cli
```

Method 2: Build from Source (For GPU)

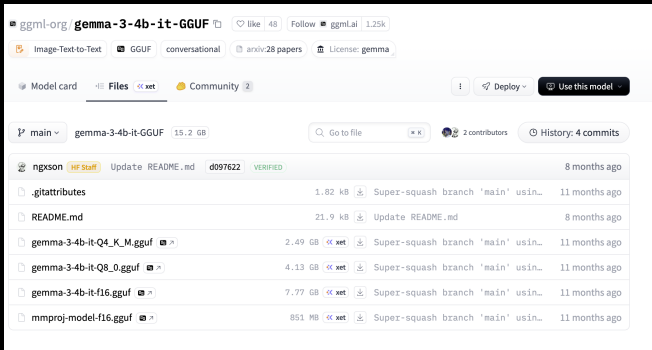
```
1 git clone https://github.com/ggerganov/llama.cpp
2 cd llama.cpp
3 make LLAMA_CUDA=1 # CUDA support
4 make LLAMA_METAL=1 # Metal (macOS)
```


Downloading the Model

```
1 # Using huggingface-cli (recommended)
2 pip install huggingface-hub
3
4 hf download \
5     ggml-org/gemma-3-4b-it-GGUF \
6     gemma-3-4b-it-Q4_K_M.gguf \
7     --local-dir ./models
8
```

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

Quantization: Speed vs Accuracy



The screenshot shows the GitHub repository for `ggml-org/gemma-3-4b-it-GGUF`. The repository has 48 likes and 1.25k followers. It includes tags for `Image-Text-to-Text`, `GGUF`, `conversational`, `arxiv:28 papers`, and `License: gemma`. The repository is a Model card with Files (xet) and Community (2) tabs. The main branch is `main` for the `gemma-3-4b-it-GGUF` repository, which is 15.2 GB in size. It has 2 contributors and 4 commits. The file list includes:

File	Size	Language	Commit	Branch	Usin...	Time
<code>.gitattributes</code>	1.82 kB		d097622	VERIFIED		8 months ago
<code>README.md</code>	21.9 kB					8 months ago
<code>gemma-3-4b-it-Q4_K_M.gguf</code>	2.49 GB	xet				11 months ago
<code>gemma-3-4b-it-Q8_0.gguf</code>	4.13 GB	xet				11 months ago
<code>gemma-3-4b-it-f16.gguf</code>	7.77 GB	xet				11 months ago
<code>mmpj-model-f16.gguf</code>	851 MB	xet				11 months ago

- Quantization reduces model size and increases throughput.
- Higher-precision quantization (e.g., Q8_0) 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)

```
1 ./llama-cli \  
2 -m models/gemma-3-4b-it-Q4_K_M.gguf \  
3 -p "Write a Python function to check if a number is prime" \  
4 -n 512 \  
5 -ngl 99 \  
6 -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

```
1 ./llama-server \  
2   -m models/gemma-3-4b-it-Q4_K_M.gguf \  
3   --host 0.0.0.0 \  
4   --port 8080 \  
5   -ngl 99 \  
6   -c 8192  
7  
8 # Server running at http://localhost:8080
```

Server Mode: The Game Changer

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

```
1 ./llama-server \  
2   llama-server --gpt-oss-20b-default --port 8080 -a gpt-oss-20b  
3  
4 # --gpt-oss-20b-default          use gpt-oss-20b (note: can download weights from the  
  ↪ internet)  
5 # --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?

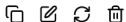
 gemma



 60 tokens

 1.38s

 43.50 tokens/s



Testing the Server

```
1 curl http://localhost:8080/v1/chat/completions \  
2   -H "Content-Type: application/json" \  
3   -d '{  
4     "model": "gpt-oss-20b",  
5     "messages": [  
6       {"role": "user", "content": "Say hello"}  
7     ]  
8   }'
```

Response:

```
1 {  
2   "id": "chatcpl-abc123",  
3   "object": "chat.completion",  
4   "choices": [{  
5     "message": {"role": "assistant", "content": "Hello!"}  
6   }]  
7 }
```

Python Integration

The Drop-In Replacement Pattern

```
1 from openai import OpenAI
2
3 # Option 1: OpenAI Cloud
4 client_cloud = OpenAI(api_key="sk-proj-xxx")
5
6 # Option 2: Local llama.cpp server
7 client_local = OpenAI(
8     base_url="http://localhost:8080/v1",
9     api_key="not-needed" # llama.cpp ignores this
10 )
11
12 # Same interface, different backend!
```

Lab 2: Your First Local Client

```
1 from openai import OpenAI
2
3 client = OpenAI(
4     base_url="http://localhost:8080/v1",
5     api_key="local"
6 )
7
8 response = client.chat.completions.create(
9     model="gpt-oss-20b",
10    messages=[
11        {"role": "system", "content": "You are a helpful assistant."},
12        {"role": "user", "content": "Write a function to reverse a string."}
13    ],
14    temperature=0.7,
15    max_tokens=500
16 )
17
18 print(response.choices[0].message.content)
```

Run it: `python test_local.py`

Streaming Responses

```
1 from openai import OpenAI
2
3 client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")
4
5 stream = client.chat.completions.create(
6     model="gpt-oss-20b",
7     messages=[{"role": "user", "content": "Explain async/await"}],
8     stream=True
9 )
10
11 for chunk in stream:
12     if chunk.choices[0].delta.content:
13         print(chunk.choices[0].delta.content, end="", flush=True)
```

Temperature and Sampling

```
1 # Temperature: 0.0 (Deterministic)
2 response = client.chat.completions.create(
3     model="gpt-oss-20b",
4     messages=[{"role": "user", "content": "What is 2+2?"}],
5     temperature=0.0 # Always same answer
6 )
7
8 # Temperature: 1.0 (Creative)
9 response = client.chat.completions.create(
10     model="gpt-oss-20b",
11     messages=[{"role": "user", "content": "Write a creative story"}],
12     temperature=1.0 # More variety
13 )
```

- **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

```
1 from openai import OpenAI
2
3 client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")
4
5 messages = [
6     {"role": "system", "content": "You are a helpful assistant."}
7 ]
8
9 while True:
10     user_input = input("You: ")
11     if user_input.lower() == "exit": break
12     messages.append({"role": "user", "content": user_input})
13
14     response = client.chat.completions.create(
15         model="gpt-oss-20b", messages=messages,
16     )
17
18     assistant_msg = response.choices[0].message.content
19     messages.append({"role": "assistant", "content": assistant_msg})
20     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: → 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

```
1 tools = [  
2     {  
3         "type": "function",  
4         "function": {  
5             "name": "get_weather",  
6             "description": "Get current weather for a city",  
7             "parameters": {  
8                 "type": "object",  
9                 "properties": {  
10                    "city": {"type": "string", "description": "City name"},  
11                    "units": {"type": "string", "enum": ["celsius", "fahrenheit"]}  
12                },  
13                "required": ["city"]  
14            }  
15        }  
16    }  
17 ]
```

Implementing Python Functions

```
1 def get_weather(city: str, units: str = "fahrenheit") -> dict:
2     """Mock weather API call."""
3     weather_data = {
4         "New York": {"temp": 72, "condition": "Sunny"},
5         "London": {"temp": 15, "condition": "Rainy"},
6         "Tokyo": {"temp": 28, "condition": "Cloudy"}
7     }
8
9     data = weather_data.get(city, {"temp": 20, "condition": "Unknown"})
10
11     if units == "celsius":
12         data["temp"] = int((data["temp"] - 32) * 5/9)
13
14     return {
15         "city": city,
16         "temperature": data["temp"],
17         "condition": data["condition"],
18         "units": units
19     }
```

Tool-Calling Request

```
1 from openai import OpenAI
2
3 client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")
4
5 response = client.chat.completions.create(
6     model="gpt-oss-20b",
7     messages=[
8         {"role": "user", "content": "What's the weather in New York?"}
9     ],
10    tools=tools,
11    tool_choice="auto" # Let model decide when to use tools
12 )
13
14 print(response.choices[0].message)
```

Lab 4: Complete Tool-Calling Agent

```
1 available_functions = {"calculate": calculate}
2
3 def run_agent(user_query: str):
4     import json
5     from openai import OpenAI
6     messages = [
7         {"role": "system", "content": "You are a math assistant."},
8         {"role": "user", "content": user_query}
9     ]
10
11     client = OpenAI(base_url="http://localhost:8080/v1", api_key="local")
12
13     while True:
14         response = client.chat.completions.create(
15             model="gpt-oss-20b", messages=messages,
16             tools=tools, tool_choice="auto"
17         )
18
19         message = response.choices[0].message
20         messages.append(message)
21
22         if not message.tool_calls:
```

Agent Loop Implementation

```
1 def run_agent(user_query: str):
2     messages = [
3         {"role": "system", "content": "You are a math assistant."},
4         {"role": "user", "content": user_query}
5     ]
6
7     while True:
8         response = client.chat.completions.create(
9             model="gpt-oss-20b", messages=messages,
10            tools=tools, tool_choice="auto"
11        )
12
13        message = response.choices[0].message
14        messages.append(message)
15
16        if not message.tool_calls:
17            return message.content
18
19        for tool_call in message.tool_calls:
20            func_name = tool_call.function.name
21            func_args = json.loads(tool_call.function.arguments)
22            result = available_functions[func_name](**func_args)
23            messages.append({"role": "tool", "tool_call_id": tool_call.id,
24                           "content": json.dumps(result)})
```

Running the Agent

```
1 # If the script doesn't expose an entrypoint, add a simple main:
2
3 def main():
4     result = run_agent("What is 2 + 2 * 3?")
5     print(result)
6
7 if __name__ == "__main__":
8     main()
```

Advanced Agent Patterns

Multi-Tool Agent: File System Navigator

```
1 def list_directory(path: str) -> dict:
2     try:
3         items = os.listdir(path)
4         return {"path": path, "items": items, "count": len(items)}
5     except Exception as e:
6         return {"error": str(e)}
7
8 def read_file(filepath: str) -> dict:
9     try:
10         with open(filepath, 'r') as f:
11             return {"filepath": filepath, "content": f.read()}
12     except Exception as e:
13         return {"error": str(e)}
14
15 def run_command(command: str) -> dict:
16     try:
17         result = subprocess.run(command, shell=True,
18                                 capture_output=True, text=True)
19         return {"stdout": result.stdout, "stderr": result.stderr}
20     except Exception as e:
21         return {"error": str(e)}
```


Lab 5: Autonomous Code Analyzer

```
1 def code_analyzer_agent(task: str, max_iterations=10):
2     messages = [{
3         "role": "system",
4         "content": """"You are a code analysis agent. You can:
5 - list_directory: Browse folders
6 - read_file: Read source code
7 - run_command: Run tests or linters""""
8     }, {"role": "user", "content": task}]
9
10    for iteration in range(max_iterations):
11        response = client.chat.completions.create(
12            model="gpt-oss-20b", messages=messages,
13            tools=tools, tool_choice="auto"
14        )
15
16        message = response.choices[0].message
17        if not message.tool_calls:
18            return message.content
19
20        # 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

```
1 import httpx
2 import xml.etree.ElementTree as ET
3 from fastmcp import FastMCP
4
5 mcp = FastMCP("PMC Fetching Test")
6
7 @mcp.tool
8 def get_abstract_by_pmcid(pmcid: str = None) -> str:
9     url = f"https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pmc&id={pmcid}"
10
11     response = httpx.get(url, timeout=10.0)
12     response.raise_for_status()
13
14     root = ET.fromstring(response.content)
15     abstract_elem = root.find("./abstract")
16
17     return "".join(abstract_elem.itertext())
18
19 if __name__ == "__main__":
20     mcp.run()
21
22
```

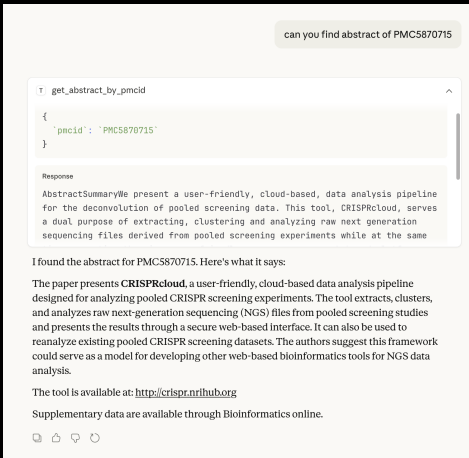
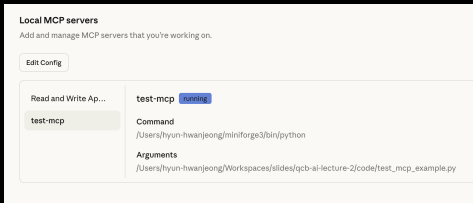
Connecting to Claude Desktop

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

```
1 {  
2   "mcpServers": {  
3     "test-mcp": {  
4       "command": "/Users/hyun-hwanjeong/miniforge3/bin/python",  
5       "args": [  
6         "/Users/hyun-hwanjeong/Workspaces/slides/qcb-ai-lecture-2/code/26_test_mcp_example.py"  
7       ]  
8     }  
9   }  
10 }
```

- 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



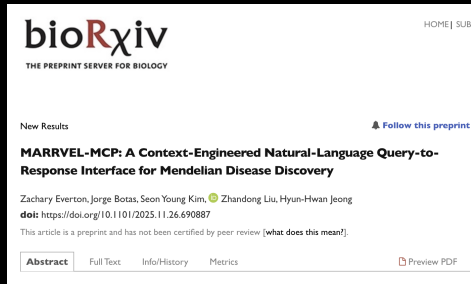
- **Left:** The MCP server is recognized and running in Claude's settings.
- **Right:** Claude uses the `get_abstract_by_pmcid` tool to answer a query.

MARRVEL-MCP: Rare Disease Discovery

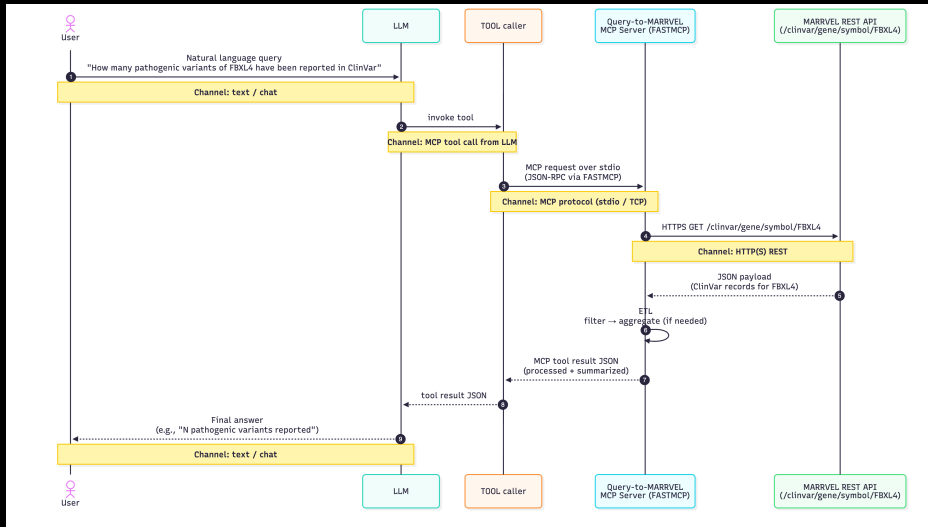
Our Latest Work: MARRVEL-MCP — Everton et al. (2025, in revision)

Abstracting Complexity in Genetics:

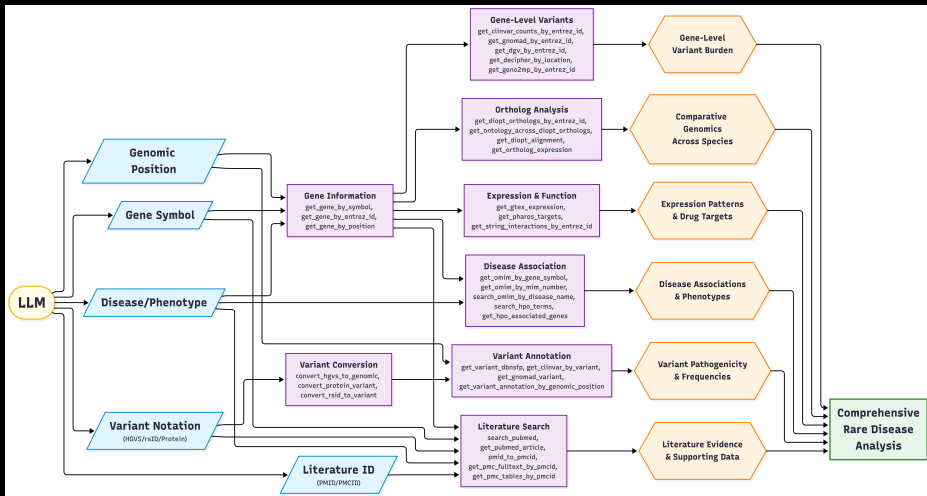
- **Paper:** "A Context-Engineered Natural-Language Query-to-Response Interface for Mendelian Disease Discovery"
- **Impact:** Enables clinicians/researchers to query 35+ genetics databases using plain English.
- **Open Source:** Full MCP server implementation available on GitHub.



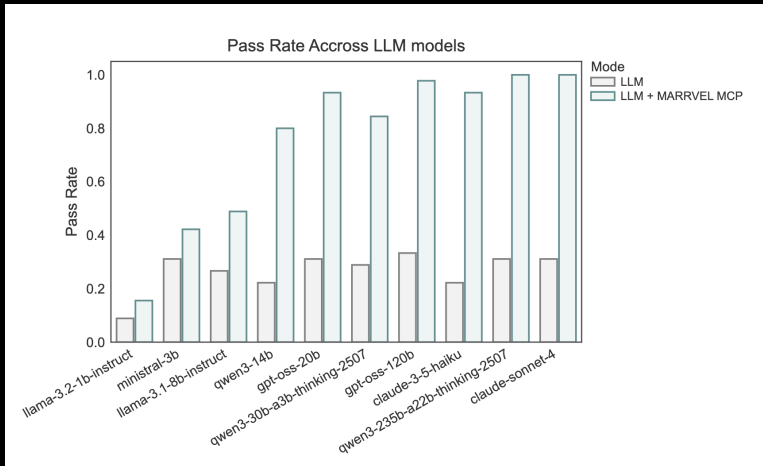
System Architecture



The tool Landscape: 35+ Specialized Tools across six categories



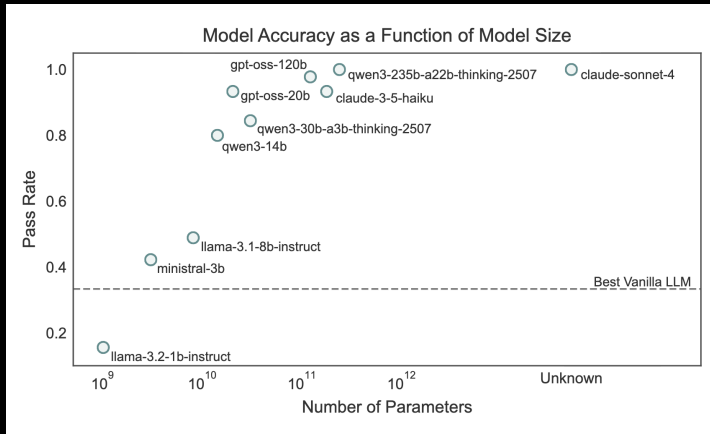
Benchmarking Performance



The "MCP Jump":

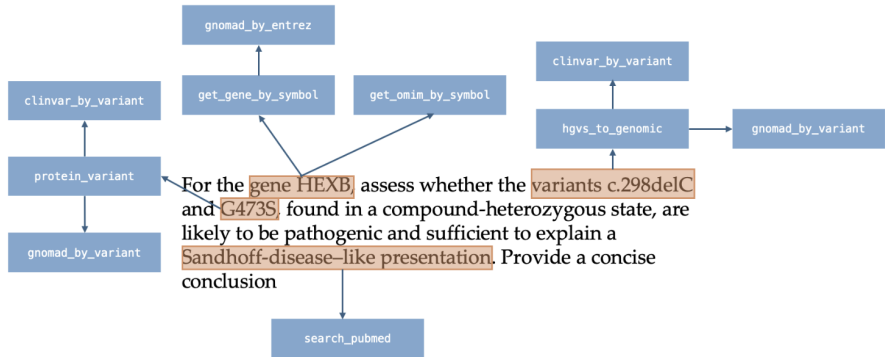
- Vanilla LLMs (grey) fail on >70% of complex genetics queries.
- LLM + MARRVEL-MCP (teal) achieves **near-perfect accuracy** (90-100%) for 20B+ models.
- Even small models (e.g., 3B) become significantly more useful with MCP.

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



Production Patterns

Configuration Management

```
1 from dataclasses import dataclass
2 import os
3 from openai import OpenAI
4 @dataclass
5 class LLMConfig:
6     base_url: str = "http://localhost:8080/v1"
7     api_key: str = "local"
8     model: str = "gpt-oss-20b"
9     temperature: float = 1.0
10    max_tokens: int = 2000
11
12    @classmethod
13    def from_env(cls):
14        return cls(base_url=os.getenv("LLM_BASE_URL", cls.base_url),
15                  model=os.getenv("LLM_MODEL", cls.model)
16                )
17
18 config = LLMConfig.from_env()
19 client = OpenAI(base_url=config.base_url, api_key=config.api_key)
```

Lab 6: Agent Class

```
1 class ProductionAgent:
2     def __init__(self, config: AgentConfig):
3         self.config = config
4         self.client = OpenAI(base_url="http://localhost:8080/v1")
5         self.tools = {}
6
7     def register_tool(self, schema: dict, func: Callable):
8         func_name = schema["function"]["name"]
9         self.tools[func_name] = {"schema": schema, "func": func}
10
11    def execute_tool(self, name: str, args: dict) -> dict:
12        try:
13            return self.tools[name]["func"](**args)
14        except Exception as e:
15            return {"error": str(e)}
16
17    def run(self, task: str) -> str:
18        # Agent loop with error handling
19        # ... implementation
```


Deployment

Docker Deployment

```
1 FROM nvidia/cuda:12.1.0-devel-ubuntu22.04
2
3 RUN apt-get update && apt-get install -y git build-essential
4
5 WORKDIR /app
6 RUN git clone https://github.com/ggerganov/llama.cpp.git
7 WORKDIR /app/llama.cpp
8 RUN make LLAMA_CUDA=1
9
10 EXPOSE 8080
11
12 CMD ["/app/llama.cpp/llama-server", \
13     "-m", "/app/models/gpt-oss-20b.Q4_K_M.gguf", \
14     "--host", "0.0.0.0", \
15     "--port", "8080", \
16     "-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: $\sim 50\text{M}$ tokens
- **Privacy: Complete data control**

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

Essential Links:

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

Community:

- [r/LocalLLaMA](#)
- [HuggingFace](#)

Thank You!

Questions?