Local Agent Workshop @ IFI

0. Overview, prerequisites
1. LLM inference (via Ollama)
2. Tool calling
3. Model Context Protocol
4. Solving issues from GitHub
5. Ready harnesses

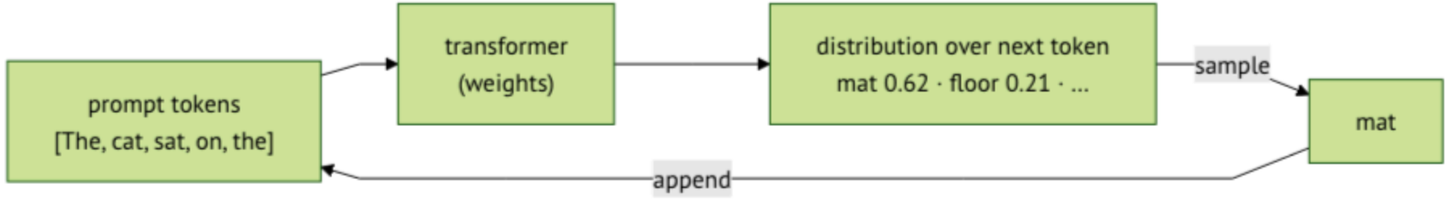
By the end you run `uv run agent.py -- --issue <url>` and it opens a PR. Orange boxes are exercises. Solutions are in `agent.py`.

0.1 `whoami`

- studied at TU Delft
- experimented with transformers
- now training Mellum models

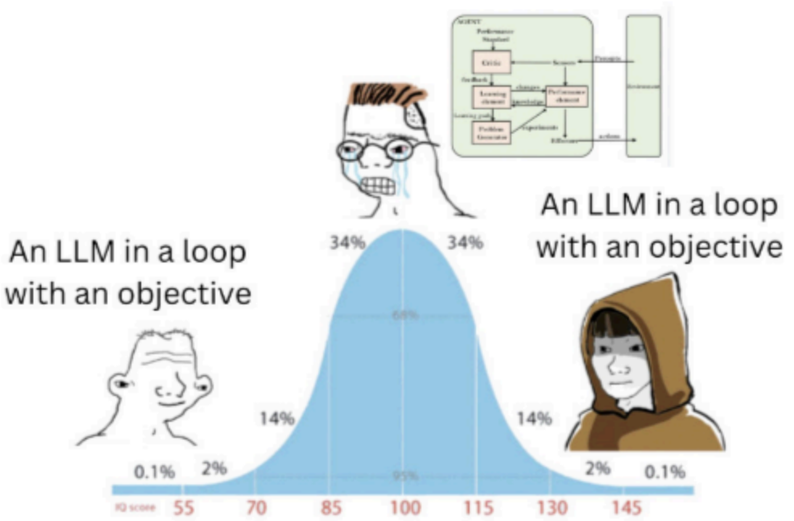
0.2 What is an LLM?

A transformer predicting the next token, over and over.



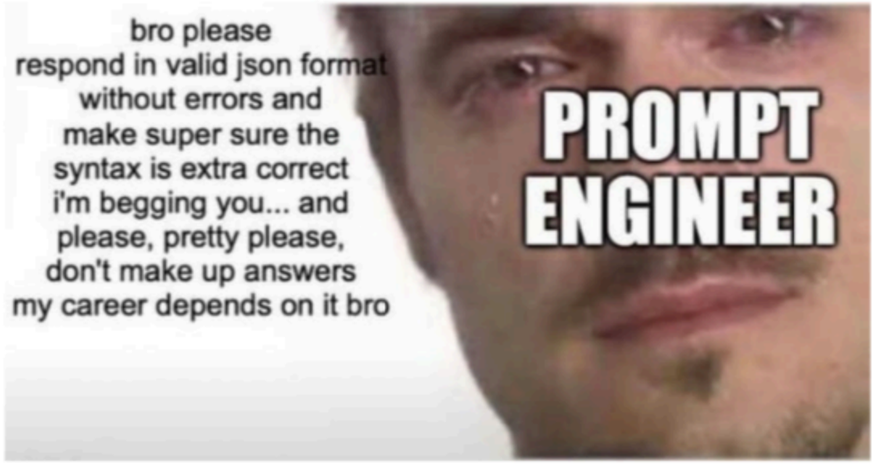
0.3 What is an agent?

Model + tool calls, in a loop.



0.4 What is a harness?

Different ways of implementing agents; mainly just a system prompt, model loop, and tool implementations.



👉 Your turn

0.5 What do you need?

- `ollama`, `uv`, `git`, `gh` (run `gh auth login`), `node` ⩾ 20
- `git clone https://github.com/Ar4l/local-agent-workshop && cd local-agent-workshop`
- `uv run marimo edit notebook.py`

```
brew install ollama uv gh node                                          # macOS
winget install Ollama.Ollama astral-sh.uv GitHub.cli OpenJS.NodeJS.LTS Git.Git   # Windows
# Linux: see README.md
```

1. LLM inference

Generate text using model weights, a large set of numbers.

👉 Your turn

1.1 Choose a model for your RAM

RAM	model	download
8 GB	`qwen3.5:4b` (or `gemma4:e4b-it-qat`)	3.4 GB
16 GB	`gemma4:12b`	7.4 GB
32 GB	`qwen3.8:27b`	17 GB

- Pull it yourself in a terminal now: the notebook never downloads models
- Ollama must be running: the macOS/Windows app starts it, on Linux run `ollama serve`
- Then pick the same model in the dropdown below

```
ollama pull qwen3.5:4b      # pick yours from the table
```

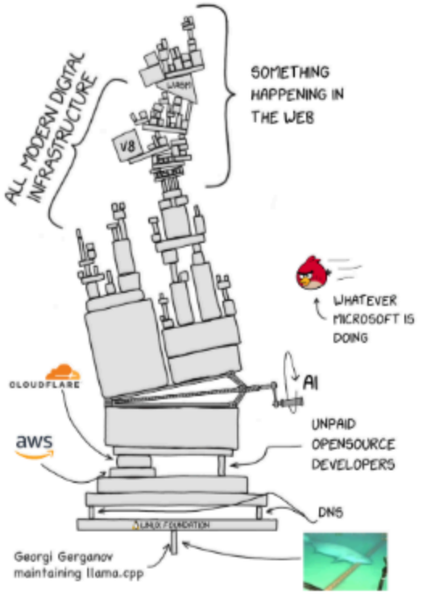
model qwen3.5:4b

✅ `qwen3.5:4b` is available

1.2 While that downloads: what is an inference server?

- It loads weights onto the GPU once, then answers HTTP requests
- Ollama provides pull, run and API commands. It launches llama.cpp's `llama-server`
- Ollama defaults to a 4k context on small GPUs. Agents need `num_ctx` ⩾ 16k

	Ollama	**llama.cpp** (`llama-server`)	**DwarfStar** (`antirez/ds4`)
what	Docker-for-models: pull, run, one local API	the engine itself, every knob exposed	one-model-class C engine by the Redis author
run	`ollama run qwen3.5:9b`	`llama-server -hf ggml-org/…-GGUF`	`./ds4-server --ctx 32768`
hardware	Metal, CUDA, ROCm, Vulkan, CPU	same + SYCL, OpenCL, RPC …	96 GB Macs, CUDA, ROCm



1.3 Four API shapes, one model

- `/api/chat` is Ollama's, `/v1/chat/completions` and `/v1/responses` OpenAI's, `/v1/messages` Anthropic's
- Existing clients like OpenAI SDK, Claude Code and Codex can use a local model
- The same prompt produced the same tool call. Only the JSON envelope differed

endpoint	tool call lives in	thinking lives in
`POST /api/chat`	`message.tool_calls[].function.{name, arguments: object}`	`message.thinking`
`POST /v1/chat/completions`	`choices[0].message.tool_calls[].function.arguments` (JSON **string**)	`message.reasoning`
`POST /v1/responses`	`output[]` item `type: "function_call"`	`output[]` item `type: "reasoning"`
`POST /v1/messages`	`content[]` block `type: "tool_use"`, `input: object`	`content[]` block `type: "thinking"`

We use `/api/chat` through the `ollama` Python package because only it accepts `num_ctx`.

👉 Your turn

1.4 Your first local chat

- Implement `chat(messages, tools)`: call `ollama.chat` once and return `.message`
- Pass `MODEL` and `OPTIONS`. The message contains `.content`, `.thinking` and `.tool_calls`
- Run the next cell and read the thinking trace

2. Tool calling

2.0 A tool call is just text

- The model emits the tags it was trained on. The server parses them into JSON
- Qwen3.5 writes `<tool_call><function=run_shell><parameter=command>`. Ollama returns `tool_calls`
- Your program decides whether to run it. Valid syntax does not grant permission

```
<think>The user wants to list files. I'll use run_shell.</think>
<tool_call>
<function=run_shell>
<parameter=command>
ls -la
</parameter>
</function>
</tool_call>
```

> 👉 Your turn
>
> ## 2.1 Tool number one: `run_shell`
>
> - Implement `run_shell(command: str) -> str`: run `subprocess.run` in `workshop.workdir()`, then return output and exit code
> - Always return a string. Never raise because errors inform the model
> - The model sees the docstring as its schema. Run the next cell to inspect it

2.1 The schema the model sees

This is everything the model knows about your tool:


```
{ 2 Items
  "type": "function"
  "function": { 3 Items
    "name": "run_shell"
    "description": "Run a shell command in the repo root and return its output and exit code."
    "parameters": { 3 Items
      "type": "object"
      "required": [ 1 Item
        0: "command"
      ]
      "properties": { 1 Item
        "command": { 2 Items
          "type": "string"
          "description": "the command to run, e.g. `node --check app.js`"
        }
      }
    }
  }
}
```

👉 Your turn

2.2 The loop

- `execute(call, tools)`: find the name, call it with `**arguments`, return a string, cap its length
- `agent_loop(messages, tools)`: chat, append reply, run tool calls, append results and repeat
- Stop when the reply has no tool calls, or after `max_turns`. Return the reason

```
for turn in range(max_turns):
    reply = chat(messages, schemas);  messages.append(reply)
    if not reply.tool_calls: return "stopped"
    for call in reply.tool_calls:
        messages.append({"role": "tool", "tool_name": call.function.name, "content": execute(call, tools)})
return "max_turns"
```

> 👉 Your turn
>
> ## 2.3 Try it on a real repo
>
> - The next cell clones `Ar4l/simple-todo-app` and forks it under your account
> - Ask: "How many lines does app.js have, and which function deletes a todo?"
> - Read the trace. Did it call the tool or guess?

2.3 Ask the agent

working directory: **`work/simple-todo-app`**

```
How many lines does app.js have, and which function deletes a todo?
```

Ask the agent

press Ask the agent first

2.4 Do special tool tokens matter?

- Every vendor uses different tags: `<tool_call>`, `[TOOL_CALLS]`, `<|DSML|>`. Parsers lag months behind
- No vendor publishes an ablation. One internal controlled run found BFCL −0.8 to +1.6 points
- Training on many formats and using fewer atomic tools made the difference

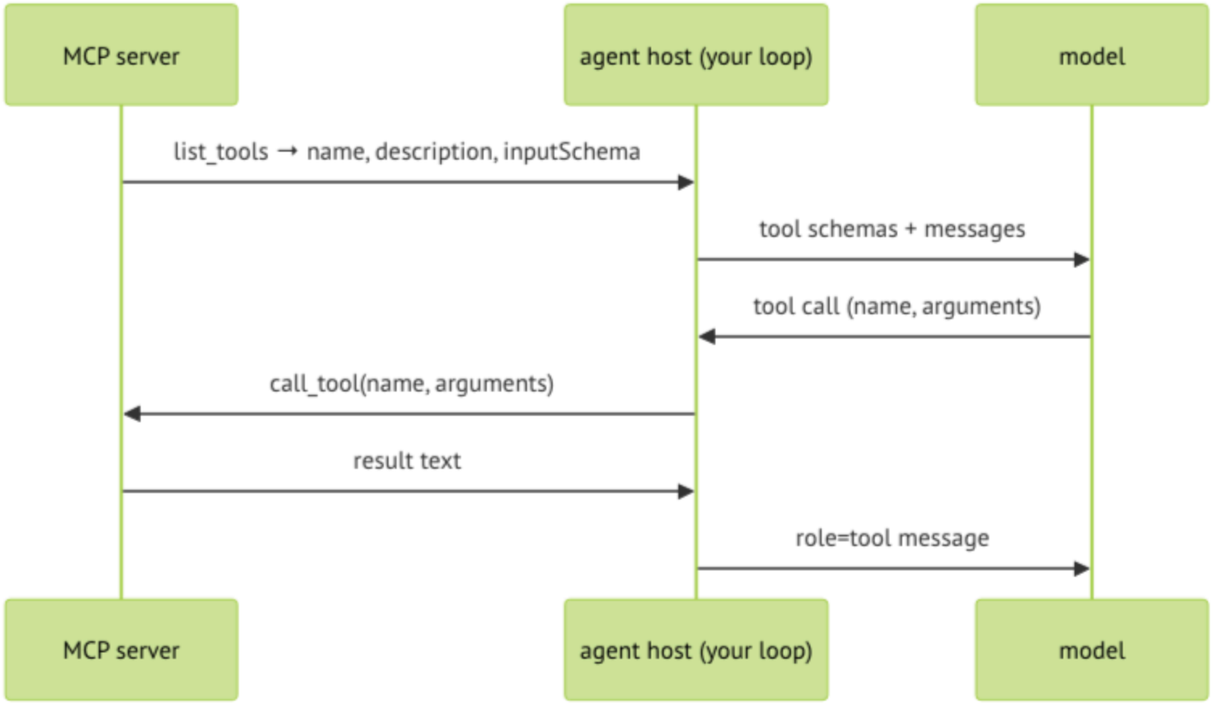
family	delimiter	argument encoding
Qwen3.5 / Qwen3-Coder	`<tool_call><function=…><parameter=…>`	XML per argument, raw strings
DeepSeek V3.2 / V4	`<｜DSML｜invoke name=…>`	XML per argument
Gemma 4	`<｜tool_call>call:NAME{…}<tool_call｜>`	key:value, quote token
Mistral	`[TOOL_CALLS]` name `[ARGS]` json	JSON
Hermes, Granite 4, Mellum2	`<tool_call>{…}</tool_call>`	JSON

2.5 Small-model realities

- Keep 3–5 atomic tools in context to improve tool choice
- Thinking improves small-model tool use by +9 to +12 BFCL Live points for Qwen3 4B/8B
- Never combine JSON `format=` with `tools=`. The grammar mask cannot emit `<tool_call>`

3. Model Context Protocol

A server (local or remote) uses MCP to expose tools and data to agents.



3.1 MCP or CLI?

- GitHub work does not require MCP. Scaffolding, not interface, drives token cost [1]
- Structured atomic edit tools reduce execution errors and improve consistency, especially for weaker models [2]
- Fewer tools in context improves tool choice: 93% at ~2 tools vs 87% at 5 [3]

Choose MCP for remote services, delegated authorization (OAuth 2.1) and hosts without a shell [4].

---

1. https://arxiv.org/abs/2608.08654 ↩
2. https://arxiv.org/abs/2608.11386 ↩
3. https://arxiv.org/abs/2605.24660 ↩
4. https://modelcontextprotocol.io/specification/2026-07-28/basic/authorization ↩

👉 Your turn

3.2 Plug an MCP server into your loop

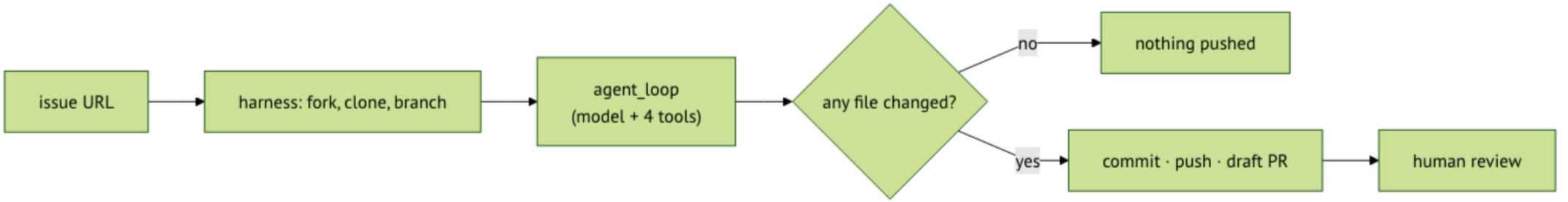
- `workshop/mcp_server.py` provides `read_file`, `write_file` and `edit_file`, restricted to the repo
- Start it with `McpSource([...])`, then create `FILE_TOOLS` from `src.tools`
- Build each entry as `(to_ollama_tool(t), (lambda name: lambda **args: src.call(name, args))(t.name))`: a plain lambda would late-bind `t`

3.2 Tools from the MCP server

MCP server exposes: `read_file`, `write_file`, `edit_file`

4. Fixing a GitHub issue end to end

- The model receives the issue text and four tools. The harness handles everything else
- When the model stops, the harness commits the diff and opens a draft PR
- Nothing is tested automatically. A human reviews the PR, as with any coding agent



> 👉 Your turn
>
> ### 4.1 Run it
>
> - Pick an issue and run it. Watch the trace, then open its PR
> - Or run: `uv run agent.py -- --issue <url> --model qwen3.5:4b`
> - Issue 1 fixes a bug. Issues 2 and 3 add features. Try 1 first

4.2 What to expect (15 runs, 32k context, no gate)

- Every model that calls tools fixes the bug issue in under 5 minutes
- Only the 27B passes issue 3 and keeps existing behaviour intact on all three
- Small models ship plausible PRs fast and quietly break features: review the diff

tier	model	memory	1: bug	2: description	3: due date
32 GB	`qwen3.8:27b`	18 GB	✅ 1.5 min	✅ 7 min	✅ 7 min
16 GB	`gemma4:12b`	8.4 GB	✅ 4.5 min	✅ 18 min	✗ silent, killed at 25 min
8 GB	`gemma4:e4b-it-qat`	5.7 GB	✅ 1 min	✅ but broke Escape-to-cancel	✗ half done
8 GB	`qwen3.5:4b`	4.3 GB	✅ 45 s	partial, 11 min	✗ 25 turns, broke add/delete

Most tool errors were `edit_file` with an `old` string that did not match the file verbatim.

4.1 Pick an issue



press Run the agent on this issue first

5. Ready harnesses

- The same loop, with more polish: `pi` and DeepSeek's `dsh` add structured tools, sessions and compaction
- `mini-swe-agent` uses only bash: a 190-line agent loop, near state of the art on SWE-bench
- DeepSeek's `dsh` uses `pi`'s model layer. `pi` intentionally ships without MCP

	mini-swe-agent	**pi**	**DeepSeek Harness (`dsh`)**
language	Python	TypeScript	TypeScript + Python SDK
tools	bash only	read, bash, edit, write, grep, find, ls	17 plugins + MCP client
agent loop	190 lines	803 lines	2.4k lines
install → answer (`qwen3.5:9b`)	8 s → 61 s, 10 calls	19 s → 9 s, 1 call	3.5 min → 41 s, 1 call
Ollama config	6-line YAML	`~/.pi/agent/models.json`	`settings.yaml` plugin composition

👉 Your turn

5.1 Try one: `pi`

- Install it, connect it to Ollama and ask the question your loop answered in 2.3
- Compare its trace with yours. What did the harness add?
- Find configurations for all three in `harnesses/`

```
npm install -g @earendil-works/pi-coding-agent
mkdir -p ~/.pi/agent && cp harnesses/pi/models.json ~/.pi/agent/   # edit the model id
cd work/simple-todo-app && pi --provider ollama --model qwen3.5:9b -p "How many lines does app.js have?"
```

Takeaways

- An agent is a loop: the model proposes, your code validates and executes, then returns results
- Small local models work with few tools, enough context and a human reviewing the PR
- MCP supplies tools, not intelligence. Harnesses provide prompts, loops and stop rules