***Project Files Summary***

**Project file summaries (what each does)**

* **app/agent/llm\_agent.py**  
  The brain. Orchestrates the conversation with the LLM, enforces JSON-only outputs, validates/repairs tool calls, runs MCP tools, and composes the final answer (e.g., “The sum is 19.75. Hello from REST API!”).
* **app/agent/prompts.py**  
  Holds the **System Prompt** instructions that strictly tell the LLM how to format responses (single JSON object, tool schema, no prose), with valid/invalid examples.
* **app/agent/tool\_catalog.py**  
  Reads the MCP tool list and renders a compact “Tools” section (names + arg schemas) that gets appended to the system prompt so the LLM knows exactly which tools exist and how to call them.
* **app/api/rest\_api.py**  
  Minimal REST service exposing the tools:
  + POST /add\_numbers → {a, b} → sum
  + GET /hello → "Hello from REST API!"  
    The MCP server calls this; you can also test with Postman.
* **app/mcp/mcp\_server.py**  
  STDIO MCP server that registers tools (add\_numbers, say\_hello) and, when called, forwards to the REST API using aiohttp. Uses mcp.server.fastmcp.FastMCP to register tools easily.
* **app/mcp/mcp\_server\_http.py**  
  Optional HTTP wrapper around the same tool functions, so you can poke tools directly over HTTP during development (nice for Postman testing).
* **app/mcp/mcp\_client\_utils.py**  
  Spawns/attaches to the MCP server via STDIO using **the same interpreter** as the agent (important on Windows). Returns a ClientSession and offers a small helper to call a tool.
* **app/config/settings.yaml**  
  Central config for the stack—LLM model name (Ollama), timeouts, server host/ports, and (if you set it) absolute path for the MCP server script.
* **app/config/logging.yaml**  
  Logging configuration (formatters, levels).
* **requirements.txt**  
  The dependency versions (modern mcp with Pydantic v2, compatible FastAPI/uvicorn, aiohttp, yaml, ollama).
* **README.md / docs/...**  
  Setup instructions, Postman collection, diagram.

**Deep dive: llm\_agent.py (how it works & why it’s robust)**

**Responsibilities**

* Build strict messages (system + user) and call Ollama with format: "json" to nudge JSON-only replies.
* Parse **only the first valid JSON object** using a brace-balancing extractor (tolerates extra junk).
* Validate model outputs into exactly one of:
  + {"tool":"<name>", "args":{...}}
  + {"final":"<string>"}
* **Repair** common mistakes:
  + Args provided as a list → merge into a dict
  + Numbers as strings → coerce to floats/ints
  + Wrong arg names → canonicalize (e.g., number1/number2 → a/b)
  + Missing a/b → extract from the user prompt or ask the model once for a corrected JSON
  + Unknown tool name → re-prompt the model with allowed tool names only
* Execute tools via MCP (ClientSession.call\_tool) and collect observations.
* **Deterministic sequencing** when needed (e.g., user asked to “say hello”: ensure say\_hello runs before finalizing).
* Compose a clean final answer even if the model misbehaves mid-flow.

**Key pieces to know**

* **JSON enforcement:** options={"format": "json"} + system prompt + parsing/validation layer.
* **Brace-balancer:** makes sure we only decode the first { ... } object even if the model returns multiple.
* **Tool-name whitelist:** compares model’s tool field to the actual MCP tool list; fixes if needed.
* **Arg canonicalizer:** maps synonyms → required names and backfills from the prompt to prevent Pydantic validation errors.
* **Deterministic control:** if the user asked for two steps (sum → hello), the agent ensures both happen—even if the model tries to finalize early.

**Typical fixes it handles (based on your journey)**

* “Unknown tool: next\_tool” → re-prompt to pick from add\_numbers, say\_hello.
* Field required: a/b → rewrite args or backfill from prompt numbers.
* Model returns raw number (e.g., 30.0) → agent composes a human final sentence.
* Extra prose/markdown fences → strip and parse the first valid JSON.

**Why we need prompts.py (System Prompt is critical)**

The **System Prompt** is your contract with the LLM. It tells the model:

* **Exactly what shape to return**: one minified JSON object—either a tool call or a final answer.
* **When to use which tool**, with strict examples:
  + Valid: {"tool":"add\_numbers","args":{"a":12.5,"b":7.25}}
  + Invalid: arrays, mixed tool & final, stringified numbers, wrong arg names.
* **No prose, no Markdown**—prevents rambling and reduces parsing failures.

This matters because small local models are **chatty and error-prone**. Without a precise system prompt:

* They add explanations before/after JSON,
* Use wrong field names,
* Emit multiple JSON objects,
* Or finalize too early.

By isolating the system prompt in prompts.py, you can tune formatting rules without touching logic. It’s your **policy layer**: change it to add new tools, update examples, adjust constraints, etc.

**What tool\_catalog.py does (and why it’s handy)**

* Pulls the **actual tool list** from the MCP session (names, descriptions, arg schemas).
* Renders a **compact, readable Tools section** and appends it to the system prompt at runtime.
* Guarantees the model always sees the **true** tool names and required arg shapes—even if you add new tools later.

This is a big robustness boost:

* Reduces hallucinated tool names (“next\_tool”).
* Reminds the model of exact arg names (a, b) so it formats correctly.
* Lets you **grow** your toolset without rewriting the agent: add a tool on the server, and the agent will advertise it automatically.

**YAML configuration (what matters & where to change it)**

**app/config/settings.yaml** — typical keys:

llm:

model: "llama3.2:3b-instruct-q4\_K\_M" # change to tinyllama if needed

timeout\_s: 30

servers:

rest\_host: "127.0.0.1"

rest\_port: 8000

mcp\_http\_host: "127.0.0.1"

mcp\_http\_port: 7001

# Optional: if you prefer absolute paths for Windows reliability

mcp:

# server\_path is inferred in code when launching via stdio; leave blank unless you need a custom path

# server\_path: "C:/Your/Path/app/mcp/mcp\_server.py"

* **Model**: choose tinyllama for faster, llama3.2:3b-instruct-q4\_K\_M for better quality (both work offline with Ollama).
* **Ports**: make sure they’re free; your Ollama runs on 11434 (default).

**The biggest issues we hit (and solutions)**

1. **System Python vs venv Python**
   * Symptom: Tracebacks show C:\Python\Python312\... instead of .venv.
   * Fix: Always activate venv; spawn MCP with sys.executable (done in mcp\_client\_utils.py).
2. **Windows WinError 10106**
   * Symptom: OSError: [WinError 10106] importing asyncio.windows\_events.
   * Fix: Run inside venv; reboot if socket stack got borked; ensure we don’t pass empty env dict when spawning; set Windows selector loop early.
3. **pydantic conflicts (\_BaseUrl import error)**
   * Symptom: ImportError: cannot import name '\_BaseUrl' from pydantic.networks.
   * Cause: Old third-party fastmcp required Pydantic v1; mcp requires Pydantic v2.
   * Fix: Stop using external fastmcp; use from mcp.server.fastmcp import FastMCP and keep mcp>=1.2.0.
4. **Model format violations (extra prose, multiple JSON objects)**
   * Fix: options={"format":"json"}, strict system prompt, brace-balancing extractor, one-shot repair prompt.
5. **Wrong/missing args (a/b)**
   * Fix: Argument canonicalizer + backfill from user prompt; one-shot repair if still missing.
6. **Premature finalization (only returns sum)**
   * Fix: Deterministic controller logic—if greeting was requested, force say\_hello before finalizing; compose the final message yourself.

**TL;DR — How to explain your project to others**

* **What:** A local-first AI agent that uses MCP to call local REST “tools” and an Ollama LLM.
* **Why it’s robust:** JSON-only enforcement, tool whitelist, arg canonicalizer, and deterministic step sequencing.
* **How to customize:** Add new tool functions to rest\_api.py & register them in the MCP server; they show up automatically to the agent via tool\_catalog.py; tweak model/paths in YAML.
* **How to run:** Start REST → start MCP STDIO server → run agent with a plain-English instruction.