**1. Function Definition & Structure**

* **Purpose**: Encapsulate reusable logic (e.g., data lookup, computation, API calls).
* **Core Components**:
  + **Signature**: Name + parameters + return type.
  + **Body**: Implementation steps, error handling.
  + **Documentation**: Docstring or comments explaining inputs, outputs, side-effects.
* **Example**:

def fetch\_user\_profile(user\_id: str) -> dict:

"""

Retrieves user profile from database.

Args:

user\_id: Unique user identifier.

Returns:

A dict with user details or raises NotFoundError.

"""

# …implementation…

**2. Wrapping Functions as “Tools”**

* **Tool Interface**:
  + **Name**: Identifier used by Agents.
  + **Description**: Natural-language summary of what it does.
  + **Parameters Schema**: JSON-compatible spec (types, required/optional).
* **Registration**:
  + Add to a registry (list or dict) that Agents consult.
  + Example entry:

json

CopyEdit

{

"name": "fetch\_user\_profile",

"description": "Get details for a user by their ID",

"parameters": {

"type": "object",

"properties": {

"user\_id": {"type": "string"}

},

"required": ["user\_id"]

}

}

* **Binding**:
  + The Agent runtime uses the schema to serialize arguments and call the underlying function.

**3. Agents & Tool Invocation**

* **Agent Loop**:
  1. **Planner** (LLM) decides next action: ask user, call tool, or finish.
  2. **Tool Selection**: Choose best-matching tool by name/description.
  3. **Parameter Fill**: LLM provides arguments in JSON per schema.
  4. **Execution**: Runtime invokes function, captures output.
  5. **Response Integration**: Agent incorporates tool output into next prompt or final answer.
* **Example Interaction**:

User: “Show me Alice’s profile.”

Agent: Calls fetch\_user\_profile({"user\_id":"alice123"})

Tool: Returns {name: "Alice", age: 29, …}

Agent → User: “Alice is 29 years old and …”

**4. Converting New Functions into Tools**

1. **Identify**: Choose functionality that benefits from automated calls (e.g., database query).
2. **Implement & Test**: Build function with clear API and error handling.
3. **Define Schema**: Write JSON Schema for inputs (types, defaults).
4. **Register**: Add to tool registry used by the Agent.
5. **Document**: Update Agent’s tool manifest so LLM “knows” it.
6. **Validate**: Simulate Agent calls with sample JSON to ensure smooth integration.

**5. Advantages of Tool-Enabled Agents**

* **Reliability**: Deterministic function outputs vs. pure LLM hallucinations.
* **Modularity**: Isolate logic into separate, testable units.
* **Extensibility**: Easy to add new capabilities without retraining LLM.
* **Security & Access Control**: Gate sensitive operations (e.g., financial transactions) behind functions.

**6. Best Practices**

* **Granularity**: Keep tools focused—one purpose per function.
* **Clear Descriptions**: LLM needs concise, unambiguous tool docs.
* **Robust Schemas**: Validate inputs to catch errors early.
* **Error Feedback**: Functions should return structured errors for Agents to handle.
* **Monitoring**: Log tool usage, performance, and failures.