Tool Design and Naming Best Practices for AI Agents

Designing Effective Tools for AI Agents

When designing tools for an AI agent, the goal is to provide a **limited, well-defined set of functions** that are as specific as possible to the agent's intended task. Well-designed tools reduce ambiguity, improve reliability, and help the AI execute actions correctly without misinterpretation.

Why Tool Design Matters

Sometimes, if tools are too generic, such as a single `list_files` or `read_file` function, the AI might struggle to use them correctly. For instance, an agent might attempt to read a file but specify the wrong directory, leading to errors. Instead, tools should be structured to enforce correctness while minimizing the agent's margin for error.

Agents **can** use generic tools as well, but more specialized tools are easier to manage and less prone to misuse by the agent. There is a trade-off between the specificity of tools and the flexibility they provide. More specific tools also limit reuse, so finding the right balance is crucial. When building an agent inititally, err on the side of specificity.

Let's assume that rather than a generic file agent, we are writing an agent that works with Python code and documentation. Instead of defining broad functions like:

- `list_files(directory: str)` → Returns all files in a specified directory.
- `read_file(file_path: str)` → Reads a file from an arbitrary path.
- `write_file(file_path: str, content: str)` → Writes to an arbitrary file.

We should define **task-specific functions** like:

- `list_python_files()` → Returns Python files **only** from the `src/` directory.
- `read_python_file(file_name: str)` → Reads a Python file **only** from the `src/` directory.
- `write_documentation(file_name: str, content: str)` → Writes documentation **only** to the `docs/` directory.

In the context of the more limited scope of Python documentation, the constraints reduce the chances of incorrect agent behavior while making it clear what each tool does.

Example: Reading a Python File

```json
{
  "tool_name": "read_python_file",
  "description": "Reads the content of a Python file from the src/
directory.",
  "parameters": {
    "type": "object",
    "properties": {
      "file_name": { "type": "string" }
    },
    "required": ["file_name"]
  }
}
```

Example: Writing Documentation

```json
{
  "tool_name": "write_documentation",
  "description": "Writes a documentation file to the docs/
directory.",
  "parameters": {
    "type": "object",
    "properties": {
      "file_name": { "type": "string" },
      "content": { "type": "string" }
    },
    "required": ["file_name", "content"]
  }
}
```

Step 2: Naming Matters – Best Practices for Tool Naming

Naming plays a crucial role in AI comprehension. If we name a tool `proc_handler`, the AI might struggle to infer its purpose. Instead, naming it `process_file` provides better clarity.

Example: Naming Comparison

Poor Name	Better Name
`list_pf`	`list_python_files`
`rd_f`	`read_python_file`
`wrt_doc`	`write_documentation`

Even with well-named tools, we still need structured descriptions for disambiguation, especially in specialized domains.

Step 3: Robust Error Handling in Tools

Each tool should be designed to handle errors gracefully and provide **rich error messages** back to the agent. This helps prevent failures and enables the agent to adjust its actions dynamically when unexpected issues occur.

Example: Improving `read_python_file` with Error Handling

```
import os

def read_python_file(file_name):
    """Reads a Python file from the src/ directory with error
handling."""
    file_path = os.path.join("src", file_name)

    if not file_name.endswith(".py"):
        return {"error": "Invalid file type. Only Python files can
be read."}

    if not os.path.exists(file_path):
        return {"error": f"File '{file_name}' does not exist in the
src/ directory."}

    with open(file_path, "r") as f:
        return {"content": f.read()}
```

This version ensures:

- **Only Python files are read**.
- **Non-existent files return an informative error**.
- **All responses are structured for easy agent parsing**.

Instructions in Error Messages

When we know that certain tools will always be used together or that there is a *right* way to handle a particular error, we can provide that information back to the agent in the error message. For example, in our `read_python_file` function, if the file does not exist, we can suggest that the agent call the `list_python_files` function to get an accurate list of file names.

```
def read_python_file(file_name):
    """Reads a Python file from the src/ directory with error
handling."""
    file_path = os.path.join("src", file_name)

    if not file_name.endswith(".py"):
        return {"error": "Invalid file type. Only Python files can
be read. Call the list_python_files function to get a list of valid
files."}
  ...
```

We could put this information into the original agent rules, but then the agent would have to remember it. Further, adding it there creates a more complicated set of rules for the agent to remember and may have unexpected side effects on how it handles errors in other tools. By injecting the instruction here, we can ensure that the agent has the information it needs to handle this error without having to remember it in its rules. It also gets the instruction “just in time” when it needs it.

Conclusion

When integrating AI into real-world environments, tool descriptions must be **explicit, structured, and informative.** By following these principles:

- **Use descriptive names.**
- **Provide structured metadata.**
- **Leverage JSON Schema for parameters.**
- **Ensure AI has contextual understanding.**
- **Include robust error handling.**
- **Provide informative error messages.**
- **Inject instructions into error messages.**

This approach ensures that AI agents can interact with their environments effectively while minimizing incorrect or ambiguous tool usage. In the next tutorial, we will explore **dynamic tool registration using decorators**, further improving flexibility and maintainability in AI agent design.