ChatGPT

Improving Tool Documentation and Metadata for the Local Git MCP Connector

To enhance how the tools and their behavior are explained to an LLM, we propose improvements across the codebase. Below, we organize recommendations by module/file, focusing on clearer function definitions, standardized metadata, and more explanatory documentation patterns.

Module: Local Git MCP Connector Implementation

Issues Identified: In the main connector code (responsible for `search` and `fetch` actions), the function docstrings and action descriptions are terse or generic. For example, the current description for the search tool is simply **"Searches for resources using the provided query string and returns matching results."** This wording is somewhat vague ("resources" could be clarified) and provides no context or usage example. Similarly, the fetch action description **"Retrieves detailed content for a specific resource identified by the given ID."** omits details about what a "resource" is (likely a file or code snippet) or how the ID is obtained.

Recommendations:

- **Clarify Action Descriptions:** Revise the `search` action's description to be more explicit about scope and results. For instance: *"Searches the connected code repository for files or content matching the given query string, and returns a list of matching file references."* This explicitly mentions *code repository* and *file references*, which guides the LLM to understand what "resources" means in context. Likewise, update the `fetch` description to: *"Fetches the contents of a file or resource identified by an ID (such as one returned from the search results) and returns its detailed content."* This clarifies the relationship between search and fetch and what the ID represents. Ensuring both descriptions use consistent tone and level of detail helps the LLM form a correct mental model of each tool's purpose.

- **Enhance Function Docstrings:** Add or expand the docstrings for the `search` and `fetch` methods in the connector class. Use a structured format that clearly explains purpose, parameters, and return value. For example, for `search`:

```
def search(self, query: str, topn: int = 10, recency_days: int = None) ->
SearchResults:
    """
    Search the repository for content matching a query string.

    Args:
        query (str): The search query string (e.g., keywords or code
snippets to find).
        topn (int, optional): Maximum number of results to return. Defaults
```

```
to 10.
        recency_days (int, optional): If provided, limit search to content
from the last N days.

    Returns:
        SearchResults: A list of matching resources (files or code
snippets) each with an identifier and context snippet.

    Example:
        >>> connector.search("function defineTool")
        [ { "id": "src/tools.py:45", "snippet": "def
define_tool(...)" }, ... ]
    """
    # ... implementation ...
```

This docstring explicitly defines what the function does, the meaning of each parameter (including optional ones like `topn` and `recency_days`), and the structure of the return. An **Example** usage illustrates how the function might be invoked and what a result looks like, which can be very helpful for an LLM to see usage patterns. The same style should be applied to `fetch`, for example:

```
def fetch(self, id: str) -> FileContent:
    """
    Fetch the content of a resource (file) by its identifier.

    Args:
        id (str): Identifier of the resource to fetch. Typically obtained
from a search result (e.g., a file path or reference).

    Returns:
        FileContent: The full contents of the file or resource. Text files
are returned as text content; binary files as binary data.

    Example:
        >>> connector.fetch("src/tools.py:45")
        "# Contents of the tools.py file starting at line 45...\\nimport
os\\n..."
    """
    # ... implementation ...
```

These improvements ensure that if the LLM has access to docstrings (through browsing code via `fetch` or other means), it will encounter well-structured, semantically clear documentation. Even if the LLM only sees the action schema, writing clear docstrings forces clarity in how we think about the function's explanation, which translates into better metadata.

- **Use Consistent Terminology:** Update terminology within this module for consistency. If the code alternates between calling the target of search a "resource" vs. a "file" or "document," choose one

term and stick with it. Given this connector deals with code, **"file"** (or "file resource") may be clearer than just "resource." Ensure that comments and docstrings consistently use the chosen term. For example, instead of "Searches for resources…", use "Searches for files…" in both code comments and the action description. This consistency helps the LLM not get confused by multiple terms for the same concept.

- **Inline Comments for Complex Logic:** If the `search` implementation has any non-trivial behavior (e.g., filtering by recency or ranking logic), include brief inline comments explaining the steps. For instance, if searching involves reading multiple directories or using regex, a comment like `# Scan all .py files in repository for the query string` before that logic can guide an LLM (and human readers) through the code's intent. Clear comments can prevent an LLM from misinterpreting code during any analysis or summarization tasks.

Module: Tool Action Schema and Metadata (e.g., Action Definitions)

Issues Identified: The connector defines its tools for the LLM using a structured schema (likely via an `Action` class or a JSON schema). The current metadata for parameters and return types is present but could be more uniform and descriptive. For example, the parameter schema for `search` only lists a `query` with description "Search query." (very brief) and doesn't explicitly mention optional fields like `topn` or `recency_days` in the action definition shown to the LLM. Inconsistencies might exist if other connectors or parts of the code use different names for similar concepts (e.g., some tool might call it `keywords` instead of `query`). Additionally, the return schema (with `$defs` for `TextContent`, `ImageContent`, etc.) is quite complex – it's likely automatically provided to the LLM but without any plain-language explanation of what the result contains.

Recommendations:

- **Standardize Parameter Definitions:** Ensure every action's parameters in the schema have a `description` field and follow a consistent format. For the `search` action, include the optional parameters in the schema definition that the LLM sees, not just in code. For example:

```
Action(
    name="search",
    description="Searches the code repository for matching files or
content.",
    params={
        "type": "object",
        "properties": {
            "query": {"type": "string", "description": "The search query
string (keywords or code to find)"},
            "topn": {"type": "number", "description": "Maximum number of
results to return (default 10)", "default": 10},
            "recency_days": {"type": "number", "description":
"If provided, limit search to content updated in the last N days"}
        },
        "required": ["query"]
```

```
    },
    ...
)
```

Here, we've added descriptions to **all** properties, and included default or optional indicators. The phrasing is consistent (imperative or noun phrases starting with capital letter and ending with a period for each description). By standardizing this, whichever parameters the LLM sees, it will have a clear idea of their role. Do the same for the `fetch` action's parameters:

```
Action(
    name="fetch",
    description="Retrieves the content of a specified file by ID.",
    params={
        "type": "object",
        "properties": {
            "id": {"type": "string", "description":
"Identifier of the file or resource to fetch (from search results)"}
        },
        "required": ["id"]
    },
    ...
)
```

Note: We replaced "resource" with **file** in the description for clarity. Little additions like "(from search results)" give the LLM context about where the ID comes from, which can prevent misuse of the tool.

- **Expand Return Type Clarity:** While the return type schema is detailed, consider adding high-level descriptions either in comments or in a simplified form for the LLM. For instance, if the `search` action returns an array of resources, we might not easily encode that in the JSON schema shown to the model (since function calling uses JSON Schema mainly for inputs), but we can mention in the **action description** what the result will contain. E.g., append to the search description: *"Returns a list of matching resource identifiers with brief context."* This sentence in the description field succinctly tells the model what to expect from the function's output, complementing the formal schema. Similarly, for fetch, the description can note: *"Returns the full content of the file (text or binary)."*

- **Consistency Across Tools:** Review all tool metadata across the connector for consistency in style. Ensure each description is written in the third person singular present tense (the current ones use "Searches" and "Retrieves", which is good). All descriptions should either all end with a period or none do – currently they do, so maintain that. Ensure similar phrasing for similar actions (if another connector had a search, its description should be phrased similarly). Consistent metadata style helps the LLM not treat similar actions as different. If any action descriptions or parameter names in this codebase deviate (for example, a hypothetical "open" action might use "filepath" vs this connector using "id"), consider aligning them or clearly distinguishing them with rationale in comments.

- **Metadata for LLM Guidance:** Although the JSON schema is primarily for validation, you can include **additional guiding info** in non-functional schema fields like `"description"` or even add a `"title"` for the overall schema object. For example:

```
params={
    "type": "object",
    "title": "SearchParameters",
    "description": "Parameters required to perform a search on the code
repository",
    ...
}
```

While the OpenAI function calling might not expose the object's description to the model, including it in the code improves clarity for any future tool that might present it. In general, leaning on descriptive fields wherever possible in the metadata makes the code self-documenting and helps future maintainers or tools.

- **Centralize Common Definitions:** The complex return schema (with `$defs` for `TextContent`, `ImageContent`, `EmbeddedResource`, etc.) appears in the connector's action definitions. If this schema is repeated for multiple actions or connectors, consider centralizing it – e.g., define a common `CallToolResult` schema in a shared module and reference it, or generate it via a helper function. This reduces the chance of inconsistencies (such as one action accidentally missing a field in the return schema). It also allows you to document that schema in one place. For example, a module `tool_schema.py` could contain:

```
CALL_TOOL_RESULT_SCHEMA = {
    "type": "object",
    "title": "CallToolResult",
    "description": "Standard structure for tool call results, including
possible text or image content.",
    "$defs": {
        ...  # definitions for TextContent, ImageContent, etc.
    },
    "properties": {
        "content": { ... },
        "isError": { ... },
        "_meta":  { ... }
    }
}
```

with comments explaining each part. Then each action's `return_type` can just refer to this, or copy it in a single line. Even if the LLM doesn't see this directly, having one well-documented schema ensures uniform behavior and easier updates (which indirectly benefits the LLM's consistent understanding).

Module: Documentation and Usage Guides (e.g., README or Developer Docs)

Issues Identified: If the repository includes a README or other documentation, it may not currently emphasize how the LLM should use the tools or what they do. The focus might be on installation or integration, rather than on model comprehension. There may also be missing examples of tool usage in context (which could serve as templates for the LLM's prompt engineering).

Recommendations:

- **Augment the README with LLM-Facing Descriptions:** Include a section in the README (or a separate docs file) specifically describing the connector's tools in simple terms. For example:
- *Tool:* **search** – *Use this to find relevant code or text in the repository. The LLM should call this when it needs to locate where something is defined or mentioned.*
- *Tool:* **fetch** – *Use this to retrieve the full content of a file or result identified by an ID from a search result. Call this after using* `search` *to get details from a specific file.*

By articulating the usage like instructions, you provide a pattern that can be mirrored in prompt engineering or in fine-tuning data. This also helps any developer or prompt writer understand the intended use of each action.

- **Provide Example Scenarios:** Add one or two realistic example interactions in the documentation showing how an LLM (or user) would utilize these tools. For instance:

```
**Example:**
*User asks:* "How does the connector define its search parameters?"
*LLM reasoning:* The LLM should use the search tool to find where in the
code `search` parameters are defined.
*LLM action:* `browser.search` with query `"def search"` -> (receives
results with file IDs)
*LLM action:* `browser.fetch` with the ID of the relevant file to read the
content.
```

  This kind of step-by-step example in documentation solidifies the pattern for both developers and the LLM. It guides the LLM on when and how to invoke each tool. Including such examples in the repository (even if not directly fed to the model) can influence how future prompt templates or fine-tuning might be designed.

- **Address Ambiguities Explicitly:** If there are any known points of confusion (for example, what constitutes a "resource ID", or limitations like binary files cannot be rendered as text), spell these out in the docs. E.g., *"Note: The* `search` *tool currently looks at text content of files; it won't find matches in binary files."* or "`fetch` *should be used on IDs returned by* `search`*. The ID format is* `<filepath>:<line range>`." Clearing up these ambiguities in documentation ensures that anyone crafting LLM prompts or improving the model will align with the intended use, thereby indirectly helping the model's understanding.

General Style and Consistency Guidelines

Beyond specific files, here are overarching improvements to enforce throughout the codebase:

- **Adopt a Docstring Style Guide:** Consistently use one docstring style (Google style as shown in examples, NumPy style, or reStructuredText). A consistent structure (Description, Args, Returns, Examples) in every function's docstring will make it easier for an LLM to parse information if needed. It also ensures all important aspects are documented. For instance, if previously some functions lacked an "Returns" section, add it for every public function.

- **Semantic Function Naming:** Double-check function and variable names for clarity. For example, if the code had cryptically named functions or variables (perhaps not in this connector, but as a general check), rename them to be self-explanatory. An LLM can struggle with arbitrary abbreviations. In this connector, names like `search` and `fetch` are clear. But if there are internal helper functions, ensure they also follow clarity (e.g., `def _grep_repo()` is clearer than `def _srch()`). While these might not be exposed to the LLM directly, clear naming reduces the cognitive load on any code interpreter (AI or human).

- **Avoid Ambiguous Terms in Comments/Docs:** Words like "it", "this", or context-dependent terms can confuse an AI. Make comments self-contained where practical. For example, instead of `# do this for each result`, say `# for each search result, retrieve its file content`. This way, even if the LLM reads a single comment in isolation, it has meaning. Consistently disambiguate references in comments and docs.

- **Include Purpose in Metadata:** Some tool interfaces might support a high-level `purpose` or category field (if the underlying framework allows it). If so, categorize these tools (e.g., search and fetch could be tagged as "file-access" or "code-navigation" tools). This could help an LLM that might one day reason about types of tools. If no such field exists, you've effectively done this by naming and describing them clearly, which is sufficient.

- **Review for Consistency with Other Connectors:** If the Local Git MCP connector is one of several, ensure its style now matches the others. For example, if there's a "Local Email" or "Web Search" connector, compare their action descriptions and docstrings. After updating the Local Git MCP, its clarity and structure should serve as a template. If others are less clear, consider applying similar changes there for uniform quality. A model that sees a unified documentation style across connectors will generalize tool use more reliably.

By implementing these improvements, the Local Git MCP connector's code and documentation will provide a much clearer semantic structure for an LLM to understand. The tools' purposes, parameters, and usage patterns will be explicit and standardized. This not only helps the LLM in choosing and using tools appropriately but also aids future developers and maintainers in quickly grasping how to work with or modify the connector. Each recommendation above strives to remove ambiguity and introduce consistency, which are key to effective communication with large language models.
