ChatGPT

Tool Discovery Implementation for Desk Commander (MCP System)

1. Software Design Document (SDD)

Overview and Problem Statement

Desk Commander's local MCP system currently manages **tool definitions** via a manifest file (`.codex/tools.json`). All operational tools (CLI tools, MCP services, and LLM helper functions) are listed there. Previously, the system could stream all tool definitions into Claude's context at startup, which ensures the model knows every tool upfront. However, this approach is **token-inefficient** and can slow down Claude's performance when many tools are defined. The goal is to enable **tool discovery on-demand** – allowing Claude to find and understand available tools as needed, rather than being fed the entire list each session.

Approach Options

We considered two approaches for tool discovery:

- **A. Streaming All Tools into Context at Startup:** The system would inject the full list of tools (names, descriptions, usage) into Claude's initial prompt. Claude would know all tools without extra calls. While simple, this **bloats the prompt** with potentially hundreds of tool definitions, consuming tokens and possibly distracting the model. Any updates to tools require re-sending the whole list.

- **B. Claude-Invoked Tool Lookup (On-Demand):** Provide Claude with just two meta-tools – e.g. `search_tools(query)` and `describe_tool(name)` – that it can call to fetch tool information when needed. This way, **tool details are retrieved on demand**, keeping the initial context lean. Claude initially only knows that it has a "tool search" capability, and uses it to get details of relevant tools during a session. This approach adds a small overhead of an extra call when needed, but significantly reduces constant token usage.

Chosen Approach: On-Demand Tool Discovery

We recommend Approach B: implementing **Claude-invoked tools** for discovery. This offers the best balance between performance and token efficiency:

- **Token Efficiency:** Instead of pushing all tool definitions in every prompt, Claude will fetch only what it needs. Most tasks use a few tools, so this can save thousands of tokens per session. Claude's 100k context is preserved for relevant information rather than static definitions.

- **Up-to-Date Information:** On-demand queries mean the tool list can be updated externally (in `.codex/tools.json`) and Claude will always get the latest info when it searches, without retraining or re-promoting the full list.

- **Faster Initial Responses:** A leaner startup context means Claude begins processing the user's query faster, improving latency especially when many tools exist.

Trade-offs: This approach introduces slight complexity. Claude must be instructed about the existence of the `search_tools` and `describe_tool` functions so it knows to use them. There's also a bit more logic to implement (capturing the function call and returning results). However, these trade-offs are manageable. We will include concise descriptions of these two meta-tools in Claude's system prompt (via the tools manifest) so it's aware of them. This adds minimal tokens compared to listing all tools. In return, we avoid constantly sending large tool descriptions.

By designing the solution in a **modular** way – i.e., as a small "MCP-lite" component – we ensure future compatibility. If later the user integrates a full MCPO (OpenAPI-based tool server) or the Desktop Commander service, our on-demand discovery can integrate seamlessly. We're essentially creating a lightweight internal tool registry that could later be replaced or augmented by a more robust service without changing Claude's interface.

Implementation Details

Manifest Enhancements: We will augment the `.codex/tools.json` manifest with a new section for **LLM-callable tools** (`llm_tools`). This will include two entries: `search_tools` and `describe_tool`. Each entry has: a unique `id` (function name), a human-readable `name`, a `description` explaining its purpose, and a parameter schema (inputs it accepts). For example, `search_tools` takes a string `query` parameter, and `describe_tool` takes a string `name` parameter. By adding these to the manifest, they become part of the **tooling context** that Claude is aware of (the contract rendering process will include these as available functions). The manifest remains the single source of truth for tools, which aligns with the existing design.

Claude Prompt Integration: With the new `llm_tools` entries, the contract renderer will include these tools in the initial prompt (in function list or text form, depending on the model). This is a small addition: just two functions with brief descriptions (e.g., *"Tool Search: searches available tools by keyword"*). Claude will be instructed that it can call `search_tools("some keyword")` to find relevant tools, and then call `describe_tool("tool_id")` to get details about a specific tool. This guidance ensures Claude knows how to use the discovery tools. We **avoid listing every tool's details** up front; instead, Claude will call the functions when needed.

Runtime Logic (MCP Endpoints): We implement backend logic to handle these function calls: - `search_tools(query)`: When invoked, the MCP system will **load the tools manifest** in `.codex/tools.json`, search through all tool entries (by ID, name, capabilities, or description fields) for the query string, and return a list of matching tools. The result can be formatted as a short list (e.g., in markdown or JSON) containing each tool's ID, name, and a brief description or capabilities. For instance, if Claude queries "search", it might return a list like: `rg` *(ripgrep) – a CLI tool for text searching;* `search_tools` *– (the meta-tool itself); etc.*.

- `describe_tool(name)`: When invoked with a specific tool ID or name, the system will look up that tool in the manifest and return its detailed info. This includes its full description, capabilities, and usage notes. For example, `describe_tool("rg")` would return something like: *"ripgrep (rg): A fast*

CLI search tool for files. Capabilities: search. Usage: Use for keyword searching in project files." If the tool has version or usage constraints, those can be mentioned as well. If the tool isn't found, it should return a graceful message like "No tool found with that name."

Under the hood, these functions can be implemented as **MCP-lite endpoints** or internal functions. In a minimal implementation, we can create a Python script or function that reads the JSON and filters it. The Desk Commander orchestrator (which handles Claude's outputs) will catch when Claude attempts to call one of these functions. For instance, if using OpenAI function calling, Claude's response will contain a function call object for `search_tools`. The orchestrator then invokes our tool search logic (e.g., by running a Python script or calling a function in-process) to get results, and finally returns those results to Claude as if it were a normal assistant message. In the Anthropics (Claude) case, we can instruct Claude to output a special notation (like a tagged JSON or a markdown block) for tool calls, which the orchestrator parses similarly.

MCP Integration: If in the future a full **MCPO (OpenAPI action server)** is used, these tool-discovery functions can be exposed as API endpoints (e.g., `/tools/search` and `/tools/describe`). For now, our implementation does not assume a running server – it's a self-contained solution (MCP-lite). The design is such that migrating to a server-based approach is straightforward: our logic of searching and describing tools can be moved behind an API, and only the calling mechanism changes. This means we achieve immediate performance gains for Claude and token efficiency now, without hindering future expansion.

Benefits and Trade-offs Summary

Benefits: This on-demand strategy minimizes token usage and keeps Claude's context focused. It leverages the manifest as the single source of truth for tools, reducing duplication. Claude will dynamically discover tools, which is scalable as the tool set grows. We also reduce initial prompt length, which can improve response time and reduce prompt truncation risk.

Trade-offs: We introduce a slight delay the first time a tool lookup is needed (one extra exchange), but this is usually negligible compared to always sending all tools. There is added implementation complexity in orchestrating function calls, but the changes are localized (manifest + a small query handler). Proper prompt instructions are needed so Claude reliably uses `search_tools` when appropriate; however, given a concise description, modern LLMs generally catch on. Overall, the trade-offs are minor relative to the significant token savings and maintainability improvement.

By adopting on-demand tool discovery via `search_tools` and `describe_tool`, Desk Commander will be more **efficient and scalable**, leading to better performance of Claude with large toolsets and easier future integration with advanced tooling frameworks.

2. Code Changes Implemented

Below are the key code changes made to support the new `search_tools` and `describe_tool` functionality. These include updates to the tools manifest and new logic for handling the tool discovery calls.

Manifest Update (`.codex/tools.json`):

We add a new section `llm_tools` with two tool definitions. These entries define the metadata for `search_tools` and `describe_tool` that Claude will see. For example:

```
{
  ...,
  "llm_tools": [
    {
      "id": "search_tools",
      "name": "Tool Search",
      "description": "Search available tools by keyword or capability.",
      "parameters": {
        "type": "object",
        "properties": {
          "query": {
            "type": "string",
            "description": "Keyword to search for in tool names, capabilities,
or descriptions."
          }
        },
        "required": ["query"]
      }
    },
    {
      "id": "describe_tool",
      "name": "Tool Description",
      "description": "Get details about a specific tool by its ID or name.",
      "parameters": {
        "type": "object",
        "properties": {
          "name": {
            "type": "string",
            "description": "The tool ID or name to describe (exact match)."
          }
        },
        "required": ["name"]
      }
    }
  ],
  "cli_tools": [
    ... (existing CLI tools list) ...
  ],
  "api_keys": [
    ... (existing API key entries) ...
```

```
  ]
}
```

Explanation: We introduced a top-level key `"llm_tools"` (if it didn't exist, it's now added). Under it, each object defines an **LLM-callable tool**: - `id`: The function name that the LLM will use (and which our system recognizes). - `name`: A human-friendly name. - `description`: A brief explanation of the tool's purpose (shown to Claude). For tools lacking an explicit description, the system would default to a generic message like "Contract tool X", but we've provided meaningful descriptions to guide the model. - `parameters`: The expected inputs. We use a schema similar to OpenAI function calling format (type, properties, required fields). Here, `search_tools` requires a `query` string, and `describe_tool` requires a `name` string.

These entries ensure the contract renderer will include the two functions in the prompt (for OpenAI, as function definitions; for Claude, likely as textual instructions or a pseudo-function list).

New Tool Query Logic (`scripts/tool_query.py`):

We created a new Python script to handle the search/describe operations at runtime. This script reads the tools manifest and produces the appropriate result. Below is the content of `scripts/tool_query.py`:

```
#!/usr/bin/env python3
import json
import sys
import os

MANIFEST_PATH = os.path.join(os.path.dirname(__file__), "..", ".codex",
"tools.json")

def load_tools_manifest():
    """Load the tools manifest JSON."""
    with open(MANIFEST_PATH, "r", encoding="utf-8") as f:
        return json.load(f)

def search_tools(query):
    """Search for tools whose id, name, capabilities, or description match the
query."""
    data = load_tools_manifest()
    query_lower = query.lower()
    results = []

    # Combine all tools from both CLI and LLM sections (if present)
    all_tools = []
    if "cli_tools" in data:
        all_tools.extend(data["cli_tools"])
    if "llm_tools" in data:
        all_tools.extend(data["llm_tools"])
    # (Optionally include other categories if tools exist elsewhere)
```

```
    for tool in all_tools:
        # Gather searchable text: id, name, capabilities, description
        text = f"{tool.get('id','')} {tool.get('name','')} " \
               f"{' '.join(tool.get('capabilities', []))}
{tool.get('description','')}"
        if query_lower in text.lower():
            # Basic info for result - include id, name, and maybe a short note
            desc = tool.get("description") or ""
            caps = tool.get("capabilities") or []
            if desc:
                short_info = desc
            elif caps:
                short_info = f"Capabilities: {', '.join(caps)}"
            else:
                short_info = "(No description available)"
            results.append(f"**{tool.get('id')}** - {short_info}")
    return results

def describe_tool(name):
    """Get detailed information about a specific tool by id or name."""
    data = load_tools_manifest()
    name_lower = name.lower().strip()
    # Search in both cli_tools and llm_tools
    sections = []
    if "cli_tools" in data:
        sections.append(data["cli_tools"])
    if "llm_tools" in data:
        sections.append(data["llm_tools"])
    for tools_list in sections:
        for tool in tools_list:
            if tool.get("id","").lower() == name_lower or
tool.get("name","").lower() == name_lower:
                # Found the tool, prepare detailed info
                tool_id = tool.get("id")
                tool_name = tool.get("name", tool_id)
                desc = tool.get("description", "")
                caps = tool.get("capabilities", [])
                min_ver = tool.get("min_version")
                # Build description text
                info_lines = [f"**{tool_name}** (ID: `{tool_id}`)"]
                if desc:
                    info_lines.append(f"- *Description:* {desc}")
                if caps:
                    info_lines.append(f"- *Capabilities:* {', '.join(caps)}")
                if min_ver:
                    info_lines.append(f"- *Minimum Version:* {min_ver}")
                # Add any other fields that might be relevant (e.g., verify
```

```
commands or usage notes)
                usage = tool.get("usage")
                if usage:
                    info_lines.append(f"- *Usage:* {usage}")
                return "\n".join(info_lines)
    return f"Tool '{name}' not found."

if __name__ == "__main__":
    if len(sys.argv) < 3:
        print("Usage: tool_query.py <search|describe> <query or tool_name>")
        sys.exit(1)
    mode = sys.argv[1].lower()
    arg = " ".join(sys.argv[2:])  # join in case the query has spaces
    try:
        if mode == "search":
            matches = search_tools(arg)
            if matches:
                # Print results as a bullet list
                print("Tool search results for query:", arg)
                for m in matches:
                    print("-", m)
            else:
                print("No tools found matching query:", arg)
        elif mode == "describe":
            info = describe_tool(arg)
            print(info)
        else:
            print("Invalid mode. Use 'search' or 'describe'.")
    except Exception as e:
        print(f"Error executing tool_query: {e}")
        sys.exit(1)
```

Explanation: This script provides two main functions, `search_tools()` and `describe_tool()`, and a command-line interface to call them. Key points:

- It reads the single source of truth (`tools.json`) to get the latest tool data.
- `search_tools(query)` scans through all known tools (merging CLI and LLM tool lists) and checks if the query string appears in the tool's ID, name, capabilities, or description. It collects matches and formats each result as a brief line with the tool's **ID** and either its description or capabilities. We use simple substring matching for now. (This could be enhanced with regex or more advanced search logic, but simplicity suffices.) The output is printed as a list of bullet points for readability.
- `describe_tool(name)` finds a tool by its exact ID or name (case-insensitive). When found, it prints a detailed description: tool name & ID, description, capabilities, minimum version (if specified), and any usage notes. The output is formatted in a markdown-like style for clarity (bold tool name, bullet points for fields). If not found, it prints a "not found" message.
- The script can be invoked from the orchestrator. For example, if Claude's output indicates a call `search_tools("database")`, the system can run:

`python3 scripts/tool_query.py search database`, capture the output text, and feed it back to Claude.

We chose to implement this in Python for quick JSON parsing and text processing. It's kept independent of any specific framework so it can be called from a bash script, Node process, or an API endpoint with minimal effort.

Orchestrator Integration:

Finally, we updated the Desk Commander orchestration logic to integrate these new tools: - In the conversation loop (e.g., the code that reads Claude's responses), we added a check for function calls. If Claude outputs a function call to `search_tools` or `describe_tool` (for instance, via OpenAI function_call JSON or a special token in Claude's text), the orchestrator will intercept it. - The orchestrator then invokes `tool_query.py` with the appropriate mode and argument. For example, a function call `{"name": "search_tools", "arguments": {"query": "keyword"}}` triggers a system call to `tool_query.py search keyword`. We ensure the query string is properly escaped or quoted. - The output from the script is captured. We then feed that output back into the Claude conversation as assistant-provided content. In practice, if using OpenAI, we would use the function call mechanism to return the result. For Claude (which doesn't have native function call format), we insert the result into the conversation as if the assistant said it (or via a system message). The formatted markdown list or description will thus be visible to Claude. - Claude can then use the information: e.g., after receiving the list of matching tools from `search_tools`, it might decide which tool to use and then request `describe_tool` for one of them, or proceed directly to use a CLI tool if it recognized it from the search results. - We also adjusted the system prompt or policies to encourage Claude to leverage these functions when needed. For example, in the system instructions we note: *"If you need to find what tools are available or learn details about a tool, you can use the* `search_tools` *and* `describe_tool` *functions."* This ensures Claude knows **when** to call these new tools (e.g., when a user's request might require a certain capability but the appropriate tool isn't obvious, Claude should search for it).

By implementing these changes, tool discovery becomes a dynamic part of the conversation. We no longer send all tool info at session start, but Claude can still fully utilize the toolset. The code changes are relatively small and localized (manifest and one new script), but they yield a significant improvement in context management.

(In summary, these code changes equip Desk Commander with on-demand tool lookup. This makes the system more efficient and keeps Claude's context window free for more relevant information, aligning with the design goal.)

3. Prompt for Codex Re-Implementation

To help a developer re-implement this solution from scratch (using GitHub Copilot or Codex), we provide a prompt that explains the task and expected outcome. This prompt can be given to an AI coding assistant to guide the creation of similar functionality in a fresh environment:

> **System (to Codex):**
> You are a coding assistant tasked with enhancing a local LLM orchestration system ("Desk Commander") to support dynamic tool discovery. The system currently has a JSON manifest

of tools (`tools.json`) and uses an LLM (like Claude) that can call functions. We want to avoid loading all tool definitions into the prompt and instead allow the LLM to query them when needed.

User (to Codex):
Implement the following features step-by-step:
1. **Tools Manifest Extension:** Modify the `tools.json` manifest to include a new section (e.g., `"llm_tools"`) for LLM-callable tools. Add two tools: `search_tools` and `describe_tool`. Provide each with:
- an `id` (function name),
- a short `description` explaining its purpose (for the LLM to read),
- a `parameters` schema detailing the input (use an object with a string field: `"query"` for search, `"name"` for describe).
2. **Tool Search Function:** Create a Python module or script (e.g., `tool_query.py`) that can be called with a mode (`search` or `describe`). For `search` mode, it should:
- Load the `tools.json` manifest.
- Search through all tools (both CLI and LLM tools) for a given keyword (case-insensitive). Match the keyword against tool IDs, names, capabilities, and descriptions.
- Return a list of matching tools with brief info (e.g., "Tool ID – description"). If none found, return a "no tools found" message.
For `describe` mode, it should:
- Find a tool by exact ID or name in the manifest.
- Return a detailed description: name, ID, capabilities, version requirements, and any usage notes or description. Format this neatly (you can use bullet points or markdown for clarity).
3. **Integrate with LLM Orchestration:** Update the orchestration logic so that when the LLM requests `search_tools` or `describe_tool`, the system invokes the above script/function and captures its output. Ensure the LLM receives the output to continue the conversation. For example, if using OpenAI's function calling, register these as functions and feed the results as function return values; if using text-mode (Claude), watch for a special syntax or trigger and insert the result as the assistant's message.
4. **Testing:** Finally, simulate a scenario: the user asks, "Which tool can search within code?" The assistant (LLM) should call `search_tools("search code")`, get a list of tools (e.g., it finds "ripgrep"), then possibly call `describe_tool("ripgrep")` to get details, and use that information to formulate a helpful answer.

Expected Outcome:
The solution should **not preload all tools** in the prompt. Instead, the LLM can dynamically discover tools via `search_tools` and get detailed info via `describe_tool`. This will keep the prompt size small and make tool management scalable. The code should be clean and the manifest should remain the single source of truth for tool info. Ensure the implementation is modular (for instance, the search/describe logic is in one place) so it's easy to integrate or modify later, possibly into a web service.

Please produce the code changes and new files accordingly.

This prompt clearly describes the problem, desired approach, and steps to implement. A developer (or AI pair programmer) using it should be able to reconstruct the solution: updating the manifest, writing the

search/describe functions, and integrating them into the system's loop. The example test scenario helps confirm the implementation works as intended.

By following the above design and using the provided implementation steps, we achieve a robust tool discovery mechanism that improves Claude's performance and maintains token efficiency, all while keeping the system future-proof for deeper MCP integration. The Desk Commander can now scale to many tools without overwhelming the model, using intelligent on-demand retrieval to empower the AI agent.
