AutoKit FUEL: Tool Retrieval Agent with Feedback-driven Update & Enrichment through Lookup

Ethan Epp, Jonathan Cheng

University of California, Santa Barbara

***Abstract*-** We introduce an agent-driven system that automatically discovers, integrates, and maintains a comprehensive, singular knowledgebase of tool descriptions ("tool cards") for use by large language model (LLM) agents. Our system interprets user (or agent) requests, retrieves relevant tools from this centralized knowledgebase, and searches online to discover new tools whenever necessary. Crucially, the agent continuously improves and refines existing tool cards by updating essential details for tool access. It also autonomously detects and corrects broken or outdated links and documentation, ensuring the knowledgebase remains accurate, robust, and consistently up-to-date. In a controlled evaluation, our agent successfully identified, repaired, and qualitatively enhanced 85% of broken tool links, and their corresponding toolcard. The agent itself functions as a powerful standalone toolfinder that can be readily integrated into existing agentic workflows. Additionally, the singular, self-improving knowledgebase it constructs can independently be incorporated into any agent-based system, significantly reducing the overhead of tool discovery and integration for future applications.

To encourage broader adoption and promote agent interoperability, we have released our agent's code and the methodology for constructing this centralized knowledgebase publicly, enabling researchers, developers, and other LLM agents to easily leverage and contribute to this resource at https://github.com/EthanEpp/autoKit-FUEL-tool-retriever

1. Introduction

As the number of tools available to enhance large language model (LLM) capabilities continues to grow, it becomes increasingly important to maintain a reliable and centralized way for agents to access and understand how to use these tools. Many existing systems rely on hardcoded tool descriptions or manually curated databases, which can quickly become outdated or incomplete. This creates friction for developers and limits the adaptability of LLM agents in real-world settings, and this not only hinders discoverability but also makes maintenance difficult as documentation links break or interfaces change—a common problem known as link rot.

In this project, we propose **AutoKit FUEL** (Feedback-driven Update & Enrichment through Lookup), a system that automates tool discovery, integration, and maintenance. Our agent, **AutoKit**, can retrieve tools from a shared knowledgebase, perform online searches to discover new tools, and utilizes **FUEL** to continuously update tool descriptions ("toolcards") by verifying their accuracy and fixing broken documentation links. This not only improves the tool's usefulness over time, but also reduces the effort required to maintain a usable toolset of toolcards for LLM-based agents.

The goal of this system is to provide both a capable toolfinder agent and a clean, up-to-date tool repository that can be reused across many applications. By making both the agent and the knowledgebase publicly available, we aim to support ongoing improvements by other agents and developers working in the space.

2. Background & Related Work

2.1 Retrieval-Augmented Generation (RAG)

Retrieval-Augmented Generation (RAG) combines a neural language model with an external retrieval component, allowing the model to access and ground its outputs in large, dynamic document collections rather than relying solely on fixed parameters. First popularized by Lewis et al. (2020), RAG architectures issue retrieval queries based on input prompts, retrieve relevant documents, and then condition the generation step on both the prompt and retrieved context.

2.2 ReAct Agents

ReAct (Reason+Act) agents interleave deliberation (Thought) with action calls to external tools, enabling transparent chain-of-thought reasoning and dynamic decision-making. ReAct agents significantly improve task performance by allowing models to explicitly articulate reasoning steps and tool usages. We adopt this pattern, enabling the agent to iteratively search for LangChain tool documentation and react to observations, which is crucial for robust link recovery and stub refinement. We both explicitly use ReAct LangChain agents, and use ReAct inspired prompting techniques in other components.

2.3 Reflexion

Reflexion extends ReAct by injecting higher-level self-reflection steps, where an agent reviews past actions, identifies failures or hallucinations, and adapts its strategy. While our current pipeline does not implement full Reflexion loops, the human-feedback and automated verifier stages embody a similar corrective principle: the system evaluates its outputs, identifies broken or outdated tool cards, and initiates automated repair subroutines, effectively closing the loop on continuous improvement.

2.4 Tool Integration in LangChain and LangGraph

LangChain and LangGraph provide abstractions for seamlessly integrating external tools—APIs, database connectors, and custom code—into LLM-driven workflows. LangChain's `Tool` interface standardizes name, description, and function signatures, while LangGraph's `StateGraph` orchestrates directed-graph execution over chains of tool calls. AutoKit FUEL builds on these foundations by ingesting tool metadata via `ToolDocsLoader`, indexing with a Chroma vectorstore retriever, and orchestrating through a StateGraph that unifies retrieval, generation, search, addition, and verification nodes.

3. System Architecture

AutoKit FUEL is composed of two primary components: **AutoKit**, the core retrieval and reasoning agent responsible for identifying and returning useful tools based on agent prompts, and **FUEL** (Feedback-driven Update & Enrichment through Lookup), a set of mechanisms for self-improving tool discovery and maintenance. AutoKit serves as the active agent interface that interprets queries, retrieves results, and reasons over candidate tools. FUEL operates behind the scenes, autonomously updating and enriching a centralized knowledgebase of tool descriptions—referred to as "toolcards."

This modular design allows AutoKit to function independently in agentic workflows while FUEL enhances the long-term reliability, coverage, and quality of the tool repository. Importantly, the FUEL approach—automated verification, repair, and enrichment of structured tool metadata—is generalizable. It can be integrated into any agent-based system seeking to maintain a high-quality, self-correcting knowledgebase of documentation.

3.1 AutoKit: Agentic Tool Discovery and Recommendation

AutoKit comprises the core discovery and retrieval components of our system. It is responsible for interpreting user queries, retrieving relevant toolcards from the knowledgebase, and leveraging external search when necessary. The system is orchestrated using a LangGraph `StateGraph` that manages node execution across a directed graph structure.

- **Document Loader & Retriever**: Tool documentation is scraped and processed using the `ToolDocsLoader`, which extracts structured content from online sources. These documents are split into manageable chunks and embedded into a Chroma vectorstore to enable fast similarity-based retrieval.
- **Query Rewriting & Candidate Selection**: The `transform_query` node rewrites vague user prompts into optimized retrieval queries, increasing recall. The `generate` node uses a custom RAG pipeline—dubbed "ToolFinderGPT"—to synthesize toolcards from the vectorstore results.
- **Web Search Fallback**: If the retrieved toolcards are insufficient, the agent transitions to the `web_search` node, where it uses either plain Tavily search or a ReAct-based agent to query developer documentation and public APIs for additional candidates. These are also converted into toolcards for downstream use.

AutoKit forms the initial, proactive layer of our system—focused on surfacing and recommending the best-fit toolcards to satisfy agent needs in real time.

3.2 FUEL: Feedback-driven Update & Enrichment through Lookup

FUEL extends AutoKit with robust mechanisms for autonomous knowledgebase maintenance. Its focus is on ensuring the toolcard corpus remains accurate, complete, and current.

- **Tool Adder** (`add_tool_to_database`): When a new tool is identified through search, this node prompts the model to generate a valid toolcard. It parses metadata such as function name, module path, and documentation link into a standardized JSON format and inserts the card into the shared database.
- **Verifier & Fixer** (`verify_tool_entry`): This component fetches live documentation URLs and uses our `extract_main_text` function to confirm that the link is valid and contentful. A structured verifier chain then evaluates the toolcard fields and flags errors.
- **ReAct-based Fixer Agent**: If a toolcard fails verification, a dedicated ReAct agent is invoked to recover a working documentation URL and refine the metadata. This agent searches LangChain sources first, then expands outward to general tools.
- **Human-in-the-Loop Feedback**: The `human_feedback_satisfaction` node allows users to affirm or reject the toolcard. If positive, the system generates code stubs (`handle_positive_feedback`) using the provided metadata. If negative, the query is refined and rerun.

FUEL is responsible for enriching the singular, standardized toolcard knowledgebase. This self-correcting loop ensures that even as tool documentation evolves or disappears, the system can autonomously restore functionality and improve metadata quality.

Together, **AutoKit** handles intelligent discovery and recommendation, while **FUEL** ensures continual self-verification and toolcard enhancement—forming a unified pipeline for robust, reusable tool integration across agentic applications.

4. Implementation Details

4.1 Core Implementation

AutoKit FUEL is implemented using several key libraries and frameworks, notably **LangChain-community tools**, **Anthropic's Claude 3.5 model**, and the **Tavily Search API** for web searches. We choose these libraries and tools for resource limitation reasons, however they can be switched for any relevant tool.

Our implementation is organized around a modular and maintainable code structure:

- **Core Components**: Organized into clear, reusable nodes (e.g., `document_search`, `transform_query`, `generate`, and `web_search`) orchestrated by a `LangGraph StateGraph`. This allows flexible yet structured control flow through the AutoKit pipeline.
- **Data Model**: Defined through a `GraphState` structure, encapsulating the messages, retrieved documents, candidate answers, retry counts, and user feedback. This structured state management simplifies data handling and improves readability and interpretability across the agentic pipeline.

4.2 Prompting Methodology

AutoKit FUEL uses prompting strategies based on the ReAct (Reason+Act) framework. In this approach, prompts guide the model to explain its reasoning ("Thought") and choose when to use external tools ("Action"). This helps the system make decisions more clearly and effectively.

We use explicit ReAct agents for tasks like web search and tool discovery, where the agent issues search queries, reads results, and decides what to do next based on what it finds. Other parts of the system—like the verifier and fixer—also follow this general pattern. They break tasks into steps, explain their reasoning, and use available information to update toolcards accurately.

Example prompts used in the system, including the verifier and fixer prompts, can be found in the source code.

5. Results

We evaluate AutoKit FUEL across two core dimensions: (1) self toolcard verification/repair accuracy and (2) tool retrieval quality. While the system is primarily designed for tool retrieval and recommendation, our results for those components rely on qualitative evaluation. Below, we summarize the outcomes of our experiments with self-verification and discuss our observational insights into retrieval performance

5.1 Toolcard Verification and Repair

To evaluate the self-healing capabilities of the FUEL subsystem, we manually injected faults into 40 toolcards by corrupting their documentation URLs, simulating the common real-world failure mode of link rot. These corrupted toolcards were then passed through our system's automated "toolcard fix" routine to verify the validity of the existing links, search for updated documentation sources, and update the toolcard with retrieved documentation links.

Out of the 40 corrupted toolcards:

- **34 links were successfully repaired**, meaning the agent located a valid, current documentation source and updated the toolcard accordingly.
- This corresponds to an **85% end-to-end repair rate**, demonstrating the system's ability to autonomously recover from documentation failure scenarios.
- For each repaired entry, the agent also extracted fresh documentation to update the toolcard's structure fields like `module`, `class`, and `init_args`.
- The toolcard improver component further enhanced entries by rewriting tool descriptions to be more descriptive and aligned with actual usage patterns. Examples of these improvements are also included in the Appendix.

These results highlight the robustness of the FUEL pipeline in handling one of the most persistent maintenance problems in tool-based LLM ecosystems: outdated or broken links. Our system enables the agent to restore the functionality of its knowledgebase with minimal manual intervention.

5.2 Qualitative Insights on Tool Retrieval and Improvement

To gain informal insights into the performance of the tool retrieval pipeline, we conducted a small set of manual evaluations. We passed in a range of natural-language prompts describing desired tool functionality and observed whether the system's generated toolcards appropriately addressed the user's request.

These evaluations were not conducted using a formal benchmark or predefined scoring rubric, but they offered useful insights into system behavior. Across the test prompts, the system generally produced relevant tool suggestions, with most responses being coherent and reasonably grounded.

We also evaluate the agent's self improvement capabilities, with it automatically verifying and improving the toolcard descriptions if it finds the description, documentation, or information about a tool to be wrong, limited, or having room for improvement.

We consistently observe the agent fixing incorrect class and module definitions, pricing structures, and adding more clarity and explanation to the description of the tool cards, ideally improving the retrieval of the tool cards down the line.

Below are a few other key observations from this qualitative evaluation phase:

- The hallucination detection stage in the pipeline almost always passes. This suggests that the system rarely generates unsupported or fabricated tool information when the retrieval step succeeds, and that the use of RAG is likely sufficient for guarding against hallucinations.
- The tool card improver almost always expanded on the description of the tool, and as far as we could find, there were no hallucinations. It did sometimes add in stylistic words, but we do not believe this is inherently bad.

We provide examples of improved toolcards in the appendix

While this qualitative testing does not constitute a comprehensive evaluation, the results indicate that the pipeline behaves sensibly under realistic conditions and retrieves relevant, grounded results to user queries. Further work will be needed to quantify retrieval precision and recall under controlled conditions.

5. Conclusion

In this work, we presented AutoKit FUEL, a feedback-driven system for automated tool discovery, verification, and maintenance. By combining agentic retrieval, hallucination detection, and self-validated toolcard updates, our system provides a robust foundation for tool integration in LLM-based workflows. While our initial results demonstrate strong performance in maintaining up-to-date tool metadata, further work is needed to evaluate and extend the system's capabilities for tool retrieval.

Future improvements include integrating the Model Context Protocol (MCP) for better agent interoperability, developing a standardized evaluation framework for tool retrieval accuracy, and incorporating user preferences and context to personalize results. We also aim to refine code generation, enable direct execution of retrieved tools, and explore the use of larger or more specialized language models to improve retrieval and reasoning quality.

6. References

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W., Rocktäschel, T., Riedel, S., & Kiela, D. (2021). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. arXiv. https://arxiv.org/abs/2005.11401

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). *ReAct: Synergizing Reasoning and Acting in Language Models*. arXiv. https://arxiv.org/abs/2210.03629

Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., & Yao, S. (2023). *Reflexion: Language Agents with Verbal Reinforcement Learning*. arXiv. https://arxiv.org/abs/2303.11366

LangChain Inc. (2025). LangGraph: Low-level orchestration for stateful agents [Computer software]. https://github.com/langchain-ai/langgraph

LangChain AI. (2025). LangChain: Building applications with LLMs through composability [Computer software]. https://github.com/langchain-ai/langchain

Appendices

A. Toolcard Improver Results

Below, we give an example of a toolcard in our knowledge base before and after our toolcard verification and repair step. Here it provides a much more detailed description, and also fixes incorrectly defined modules and classes

Original Entry:

```
{
	"name": "DuckDuckGo Search",
	"description": "Privacy-focused search via DuckDuckGo API",
	"programming_language": "python",
	"module": "langchain.tools.duckduckgo",
	"class": "DuckDuckGoSearch",
	"init_args": {},
	"openapi": null,
	"pricing": "Free",
	"documentation": "https://python.langchain.com/docs/integrations/tools/ddg/"
},
```

Self Improved Entry:

```
{
	"name": "DuckDuckGo Search",
	"description": "Privacy-focused search tool that provides access to DuckDuckGo's search capabilities. It offers two main functionalities: DuckDuckGoSearchRun for direct search results and DuckDuckGoSearchResults for more detailed results including snippets, titles, and links. The tool can perform general web searches and news-specific searches.",
	"programming_language": "python",
	"module": "langchain_community.tools",
	"class": "DuckDuckGoSearchRun, DuckDuckGoSearchResults",
	"init_args": {
		"output_format": "string",
		"backend": "text"
	},
	"openapi": null,
	"pricing": "Free",
	"documentation": "https://python.langchain.com/docs/integrations/tools/ddg/",
	"verified": true
}
```

B. Sample Response

Below, we highlight the most relevant parts of our agent's response to a user's query. Here is a link to the full model output, with all intermediate steps: full output.

```
---RETRIEVE---

[document_search] Retrieved 4 document(s) for question: 'I want a tool that can integrate with my google calendar'
---DECISION: GENERATION IS GROUNDED IN DOCUMENTS---
---DECISION: GENERATION DOES NOT ADDRESS QUESTION---
...
[generate] (retry #3)
I'm sorry, I couldn't find a tool that meets your requirements. The provided tool descriptions do not include any tools
specifically for integrating with Google Calendar. The available tools are related to news, Google Search, and a general
search API, but none of them mention Google Calendar
integration.
---RUNNING ReAct-BASED TOOL SEARCH---
QUESTION: I want a tool that can integrate with my google
calendar
---GENERATE---
[generate] (retry #4)
Recommended Tool: Google Calendar Tool
...
This Google Calendar Tool is part of the LangChain.js library and provides functionality to interact with Google
Calendar directly from your LangChain applications. It allows you to view and create events using natural language
prompts, which aligns well with your request for Google Calendar
integration.
---COLLECTING HUMAN FEEDBACK---
USER FEEDBACK: no
---HANDLING USER FEEDBACK---
[human_feedback_satisfaction] User answered: 'no'---RUNNING ReAct-BASED TOOL SEARCH---
QUESTION: I want a tool that can integrate with my google calendarBased on the user feedback and the previously
retrieved tool, find a better tool than the previously retrieved one.
 User Feedback: I want it for python not js ---GENERATE---
[generate] (retry #5)
Recommended Tool: CalendarToolkit
Description: Toolkit for interacting with Google Calendar in Python.
...
---COLLECTING HUMAN FEEDBACK---
USER FEEDBACK: yes
---HANDLING USER FEEDBACK---
[human_feedback_satisfaction] User answered: 'yes'
---HANDLING POSITIVE FEEDBACK---
[handle_positive_feedback] Generated code stub:Certainly! I'll create a Python script that uses the CalendarToolkit to
automatically add concert information to your Google Calendar when you book a new one. This script will include
functions to create an event with the provided concert details.```python
...
calendar_toolkit = CalendarToolkit()def add_concert_to_calendar(artist, venue, date, doors_time, opener):
    . . .
    result = calendar_toolkit.create_event(event_title,start_time,end_time,description=event_description,location=venue)
    return result# Example usage
def book_concert():
   ...
    result = add_concert_to_calendar(artist, venue, date, doors_time, opener)
```This script does the following:
1. We import the necessary modules, including the `CalendarToolkit` and `datetime` for handling dates and times.
2. We initialize the `CalendarToolkit`.
3. We define an `add_concert_to_calendar` function that takes the concert details as parameters and creates an event in
Google Calendar using the `CalendarToolkit`.
4. The function formats the event title, description, start time, and end time based on the provided information
5. We use the `create_event` method from the `CalendarToolkit` to add the event to the calendar.
6. We also define a `book_concert` function that prompts the user for input about the concert details and then calls
`add_concert_to_calendar` with those details.
7. Finally, we have a simple main block that runs the `book_concert` function when the script is executed.
To use this script:1. Make sure you have set up the necessary authentication for Google Calendar API.
The script will automatically add the concert to your Google Calendar with all the provided details.This implementation
allows you to easily add concert information to your calendar as soon as you book a new one, including all the specified
details like artist name, venue, date, doors opening time, and
opener.
---ADDING TOOL TO DATABASE---
Successfully added tool 'CalendarToolkit' to database.
[add_tool_to_database] Attempted to add a new tool stub (searched=True).
[verify_tool_entry] Finished verifying tools. Check JSON files for missing/fixed/improved stubs.
```
```