

RepoAgent: A Natural-Language, Approval-Gated Agent for Operating GitHub through Schema-Derived Tool Calling

Zamam Ansari*, Manoj Singh**

**(MSc Computer Science, SIES College of Arts, Science and Commerce, Sion, Mumbai, India)*

Email: ansarizamam1@gmail.com

*** (MSc Computer Science, SIES College of Arts, Science and Commerce, Sion, Mumbai, India)*

Abstract

Developers spend a large share of their day navigating repository hosting interfaces: listing repositories, triaging issues, reviewing pull requests and collecting simple statistics. Large language models (LLMs) can turn plain-language requests into API calls, but giving a model write access to a live account raises obvious safety concerns. This case study presents RepoAgent, a web application in which a user connects a GitHub account once and then operates it through conversation. The system discovers every available GitHub operation at start-up, converts each operation's structured input schema into a JSON-Schema tool definition, and exposes one tool per operation to an LLM reached through an AI gateway. Each operation carries a risk level; any operation that is not read-only is placed behind explicit user approval, enforced in the server code rather than left to the model's discretion. Credentials are held per tenant by the Corsair integration layer, and users are identified by a lightweight cookie-backed session with no password. The interface streams the model's reasoning, shows every tool call as an inspectable card, and pauses at write operations with Approve and Deny controls. This paper describes the design, implementation and security properties of the prototype, defines an evaluation protocol, and analyses limitations, including the dependence of the guarantee on accurate risk metadata and the absence of real authentication.

Keywords — LLM agents, tool calling, GitHub automation, human-in-the-loop approval, natural-language interfaces, Next.js, Vercel AI SDK, multi-tenant integration.

I. Introduction

Repository hosting platforms expose hundreds of operations through REST and GraphQL interfaces, yet most day-to-day questions are simple: which of my repositories are private, what pull requests are waiting on me, which issues need attention. Answering them through a web interface means several page loads and filters; answering them through the API requires knowing endpoint names, parameters and pagination rules. A natural-language front end promises to hide that complexity.

Recent LLMs can select a function and fill its arguments from a user's request, and agent frameworks interleave reasoning with tool use so that a model can gather context before acting [1], [2]. The difficulty is that the same capability that makes the assistant useful also lets it create, modify, merge or delete resources on a real account. A mis-parsed instruction, an over-eager plan or injected text inside an issue body could trigger an irreversible change [7], [8].

This paper reports a case study that addresses that tension. RepoAgent is a chat application in which the model has access to every operation of a connected GitHub integration, but cannot execute any state-changing operation until the human approves that specific call in the interface. The work contributes: (1) a method for generating the tool catalogue automatically from the integration's own schema instead of hand-writing tool definitions; (2) a server-enforced approval policy derived

from per-operation risk levels; (3) a tenant-scoped execution path that keeps credentials out of the model's context; and (4) a transparent interface that exposes reasoning, tool inputs and tool outputs to the user.

II. Problem Statement and Objectives

A user wants to ask questions and issue commands about their GitHub account in ordinary language and receive concise answers, without exposing credentials to the model and without risking unintended modifications. The problem is framed as constrained tool use: the system must (a) choose correct operations and arguments from a very large operation set, (b) remain safe when the model is wrong, and (c) keep the user informed about what is happening.

The primary objective is a working prototype that satisfies these requirements for GitHub. The supporting objectives are to: (1) generate tool definitions automatically from integration schema; (2) classify every operation as read or write and gate all writes behind user approval; (3) isolate credentials and execution per tenant; (4) provide a streaming chat interface with visible reasoning and tool calls; (5) support a configurable model through a gateway; and (6) define a reproducible protocol for evaluating task success and safety.

III. Related Work

- 1) **Reasoning and Acting:** ReAct showed that interleaving reasoning traces with actions lets a model use external information sources and recover from errors [1]. Toolformer demonstrated that language models can learn when and how to call external APIs [2], and a survey of augmented language models places retrieval, tool use and reasoning in a common framework [9]. Gorilla and ToolLLM study how models handle very large API collections, a setting that closely resembles a platform with hundreds of GitHub operations [3], [4].
- 2) **Software-Engineering Agents:** SWE-bench evaluates whether models can resolve real GitHub issues by editing code [5], and SWE-agent shows that the design of the agent-computer interface strongly affects performance [6]. RepoAgent differs in scope: it does not edit source code but operates the platform

IV. Technology Stack and Data Model

The application is a single full-stack TypeScript project. Table I lists the main components.

TABLE I — TECHNOLOGY STACK

Layer	Technology	Role
Web framework	Next.js 16 with React 19	Server routes and client UI
Styling	Tailwind CSS 4	Interface styling
LLM access	Vercel AI SDK (ai, @ai-sdk/react)	Streaming, tool calling, approval flow

The relational schema has two parts. The application's own table, `sessions`, stores a random identifier, a display name and a creation timestamp. The Corsair tables (`corsair_integrations`, `corsair_accounts`, `corsair_entities`, `corsair_events`) store integration definitions, per-tenant accounts, cached entities and event payloads. Integrations and accounts carry a data-encryption-key field, and the integration is configured with a key-encryption key, which is consistent with envelope encryption of stored credentials. The application uses two database clients: Prisma for its own tables and a raw `pg` pool handed to Corsair.

V. Methodology

A. Operation Discovery and Tool Catalogue

At first use the server asks the Corsair instance for the list of API operations, parses the returned listing into dotted paths such as `github.api.repos.list`, and validates each path against a strict pattern. For every path it retrieves the structured input schema and description. Operations without a schema are skipped with a warning. The resulting catalogue is cached for the lifetime of the server process because the operation set is static for a given deployment.

itself (repositories, issues, pull requests and metadata), and it concentrates on operational safety rather than patch quality.

- 3) **Safety of LLM-Integrated Applications:** Indirect prompt injection shows that text retrieved by a tool, such as an issue description, can carry instructions that hijack an agent [7]. The OWASP guidance on LLM applications lists excessive agency, insecure output handling and prompt injection among the principal risks [8]. Human approval of consequential actions is a standard mitigation and is the central design choice in this work.
- 4) **Tool Protocols and Integration Layers:** The Model Context Protocol standardises how models discover and call external tools [10]. RepoAgent follows the same idea of schema-described tools but obtains them from the Corsair integration layer, which also manages credentials and multi-tenant accounts [11].

Layer	Technology	Role
Model	Configurable via gateway (default: Anthropic Claude Sonnet 4.5)	Reasoning and tool selection
Integration layer	Corsair with the GitHub plugin	Operation catalogue, OAuth-style connect, credential storage
Database	PostgreSQL (Docker Compose), Prisma ORM, pg pool	Sessions and Corsair tables
Rendering	react-markdown with remark-gfm	Formatted model answers

Each path becomes a tool name by replacing dots with double underscores (for example `github__api__repos__list`). The double-underscore convention is reversible: the executor splits on the same separator to find the function to call, and the interface splits it again to show a readable label.

B. Schema Translation

Corsair describes inputs in its own form-field format (string, number, boolean, literal, object, array, unknown). A translator converts each field recursively into JSON Schema: strings keep their enumerations, literals become constants with an inferred type, objects list their non-optional fields as required, and arrays convert their item type. Objects deliberately allow additional properties so that endpoints that accept fields not yet reflected in the schema remain usable. The translated schema is supplied to the model as the tool's input schema, so argument names and types come from the integration and are not hand-maintained.

C. Risk Classification and Approval Gating

The plugin is introspected for documentation metadata, which includes a risk level per endpoint. Any operation whose level is not read is a write operation. Write tools

receive two protections: (1) their description is extended with a notice that the operation requires explicit user approval; and (2) their names are registered with the AI SDK as user-approval tools, so the framework suspends the call and emits an approval request instead of executing it.

TABLE II — SAFETY LAYERS FOR WRITE OPERATIONS

Layer	Mechanism	Enforced by
Risk metadata	Per-endpoint risk level from plugin introspection	Server
Approval gate	user-approval registration for every non-read tool	AI SDK pipeline
Tool description	Warning text appended to write tools	Model guidance
System prompt	Read-first, confirm-before-write, no retry after denial	Model guidance
Interface	Approve / Deny buttons on the pending tool card	User

D. Tenant Isolation and Credential Handling

Every request resolves a session from an HTTP-only cookie. The session identifier doubles as the Corsair tenant identifier. When the model calls a tool, the executor resolves the operation on the tenant-scoped instance (`withTenant(tenantId)`), walks the dotted path through the object tree, verifies that the final node is a function, and invokes it. The model therefore never sees tokens or secrets; it sees only serialized results. Results are converted to text and truncated at 20,000 characters, with a note stating how much was cut, which bounds context growth. Errors are returned to the model as a structured failure value so that it can adapt its next call.

The second protection is enforced by the server's tool-execution pipeline and does not depend on the model obeying its instructions. The system prompt adds a behavioural layer on top: it tells the model to prefer reads for context, to explain a write before requesting it, and not to retry after a denial. Table II summarises the layers.

The connection step is a separate route. It validates the plugin name, asks Corsair to create a connect link for the tenant and returns the link, and the browser is redirected to authorize GitHub. A status route calls the authenticated-user operation for the tenant and reports whether the account is connected and which login it belongs to.

E. Agent Loop

The chat route validates the request, creates the tenant's tool set, and calls the model with streaming enabled. The loop stops after 15 steps, and the route allows up to 300 seconds of execution. Reasoning output is optional: the client sends a flag, and when it is disabled the model is called without reasoning. After the user answers an approval prompt, the client automatically resumes the conversation, so the model continues with the approved result or acknowledges the denial.

VI. System Architecture

The request path is: browser chat component → `/api/chat` → session lookup → tenant tool construction → model via the gateway → tool call → approval gate (for writes) → executor → Corsair tenant instance → GitHub API → serialised result → model → streamed answer. Credential storage and the connect flow sit beside this path in Corsair and PostgreSQL. Fig. 1 shows this path, and Figs. 2 and 3 show the user interface.

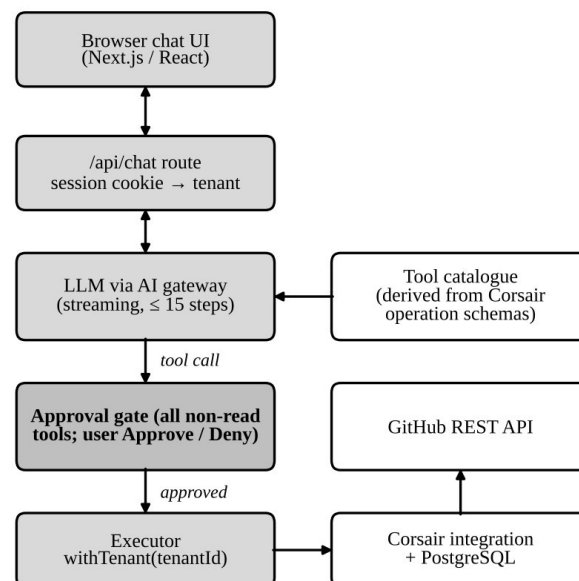


Fig. 1 System architecture of RepoAgent: request path from the browser through the approval gate and tenant-scoped executor to the GitHub API.

The design follows three principles. First, the catalogue is derived, not written: adding another Corsair plugin should add its operations without new tool code. Second,

safety is enforced outside the model. Third, everything the agent does is visible to the user.

VII. Implementation

The interface consists of a name gate, a header showing connection status, a chat area and a prompt bar.

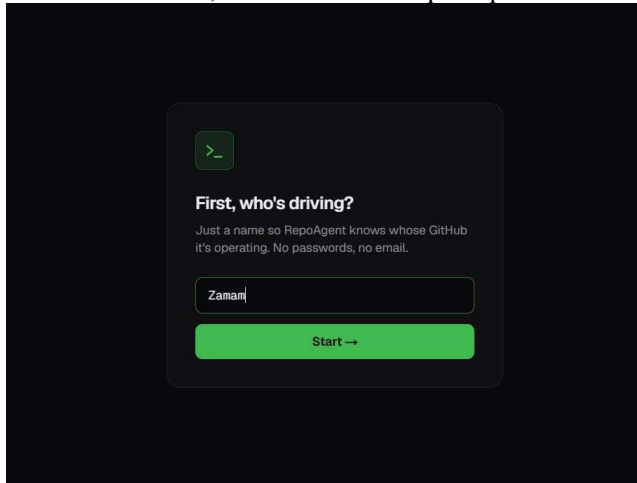


Fig. 2 Name gate: the only identity step asks for a display name (no password or e-mail).

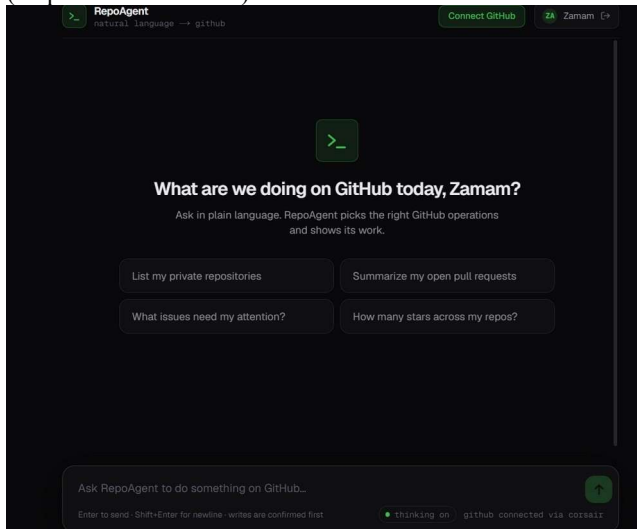


Fig. 3 Main chat screen showing connection status, suggested prompts and the notice that writes are confirmed first.

- 1) **Identity:** The first screen asks only for a display name of 1–40 characters. The server creates a session with a 24-byte random token, stores it, and sets an HTTP-only, same-site cookie that lasts one year (secure in production). There is no password or e-mail address. A control in the header deletes the session and returns to the name gate.
- 2) **Connection:** If the status route reports that GitHub is not connected, the header shows a Connect GitHub button that redirects to the authorization link. Once connected the header shows the GitHub login.
- 3) **Chat:** An empty chat offers four suggested prompts: listing private repositories, summarizing open pull requests, finding issues that need attention, and counting stars across repositories. Model answers are rendered as Markdown. A collapsible reasoning card streams the model's thinking with an elapsed-time counter. A working indicator is shown until the first visible content arrives.
- 4) **Tool Cards:** Each tool call is a card showing the scope, the action name, a status label and a short

summary of the result (for example the number of items returned). Expanding the card reveals the exact JSON input and output. A display-only classifier assigns each operation a verb category from its name (read, create, update, delete, state change or other) with a distinct glyph. For pending writes the card shows Approve and Deny buttons. Table III lists the life cycle states the card renders.

TABLE III — TOOL-CALL STATES SHOWN IN THE INTERFACE

State	Meaning	Card label
input-streaming / input-available	Model is forming or has issued the call	running
approval-requested	Write call suspended for the user	needs approval
approval-responded	User answered; call resumes	responded
output-available	Operation finished	done
output-denied	User rejected the call	denied
output-error	Execution failed	failed

- 5) **Tooling:** Two smoke-test scripts check the pipeline outside the browser: one confirms that every listed operation path resolves to a callable function on the instance, and the other builds the catalogue and prints the parsed paths with a sample schema.

VIII. Evaluation Plan and Results

The prototype is evaluated using a planned set of GitHub tasks to assess its functionality, efficiency, and safety. Since quantitative testing has not yet been conducted, actual performance results are not reported.

- 1) **Test Account and Tasks:** A dedicated GitHub test account containing known repositories, issues, and pull requests is used. The tasks include read-only queries, multi-step operations, and write operations performed on a test repository.

TABLE IV — EVALUATION TASKS

ID	Task	Type	Expected operations
T1	List my private repositories	Read	repository listing
T2	Summarise my open pull requests	Multi-step read	repository + pull-request listing
T3	What issues need my attention?	Multi-step read	issue listing
T4	How many stars across my repos?	Aggregate read	repository listing
T5	Create an issue titled "test" in [repo]	Write	issue creation (approval required)
T6	Same as T5, user denies	Write	none executed

- 2) **Evaluation Metrics:** The system will be assessed using task success rate, tool-selection accuracy, tool

calls, execution time, and token usage. Safety will be measured by checking whether any write operation executes without user approval. The target is zero unauthorized write operations.

The number of available operations and write operations will be recorded after running the registry smoke test. Actual measurements will be included once testing is completed.

IX. Case Study: From Question to Approved Action

Consider a developer who opens RepoAgent, enters a display name and connects GitHub. They type, "What issues need my attention?" The model has no data yet, so it issues a read call to list issues. The call is not gated, so it runs immediately; the card shows "running" and then a result count. The model summarises the output in a few sentences rather than printing raw JSON.

The developer then types, "Close the stale issue about the login bug." The model first reads the issue to confirm it is the right one, explains what it intends to do, and requests the close operation. Because the operation is classified as a write, the call is suspended and the card shows "needs approval" with Approve and Deny buttons. If the developer approves, the executor runs the call on their tenant and the model reports the outcome. If the developer denies, the card shows "denied", nothing is executed, and the model acknowledges the denial and asks how to proceed instead of retrying.

X. Discussion

The central design choice is to put the safety boundary in the server's tool pipeline instead of the prompt. A prompt can be ignored, overridden or manipulated by text inside a retrieved issue; a registered approval requirement cannot be skipped by the model. Showing the full input of a pending write lets the user judge the real arguments instead of the model's description of them.

Generating tools from schemas keeps the system small. The agent module is a few dozen lines, and support for further integrations depends on the integration layer, not on new tool code. The cost is that the model sees a very large tool list, which can increase prompt size and the chance of choosing a near-duplicate operation. The 15-step limit and the 20,000-character result cap are blunt controls for runaway loops and oversized outputs.

Finally, approval is only as informative as the card. A user who approves every request without reading it provides little protection, which is a general limitation of human-in-the-loop designs.

XI. Limitations

- *Safety depends on risk metadata.* If plugin introspection fails or an endpoint lacks a risk level, the registry defaults to read, so a write operation

could be exposed without a gate. A safer default would treat unknown operations as writes.

- *Identity is not authentication.* The session token is a bearer cookie that also serves as the tenant identifier, and there is no password. Anyone who obtains the cookie can use the connected GitHub account, and clearing cookies loses access to the connection.
- *Prompt injection is mitigated, not eliminated.* Read results are returned to the model as text. Approval gating limits the damage, but a manipulated model could still mislead the user in its summary or request a harmful write that a hurried user approves.
- *Single integration.* Only GitHub is connected, and the system prompt is written for it.
- *No benchmark yet.* Accuracy, latency and cost are unmeasured until Section VIII is completed.
- *Cost and availability.* The system relies on an external gateway, a model provider and a hosted integration service.
- *Catalogue is cached per process,* so changes in the plugin's operation set need a restart.

XII. Future Work

Planned improvements include treating unknown risk levels as writes, adding real authentication and per-user authorization, scoping the catalogue per task (for example by retrieving only the most relevant operations) to reduce prompt size, persisting conversation history, adding audit logging of approved writes, supporting additional integrations such as issue trackers and chat tools, and building an automated test harness with a mocked integration so that safety properties can be verified in continuous integration. Richer approval cards that show a plain-language diff of the intended change would also make approvals more meaningful.

XIII. Conclusion

RepoAgent shows that a conversational agent can operate a real developer account while keeping the human in control of every state-changing action. Deriving tools from integration schemas removes hand-written glue, tenant-scoped execution keeps credentials away from the model, and server-enforced approval gating turns the safety policy into a property of the system and not of the prompt. The prototype has not been evaluated at scale, and its guarantees rest on correct risk metadata and on a session mechanism that is not yet real authentication. Completing the evaluation protocol and hardening these two points are the next steps before the design can be recommended for general use.

Acknowledgment

The author thanks the Corsair Integration Layer and the maintainers of the open-source libraries on which this project builds.

References

- [1] S. Yao *et al.*, "ReAct: Synergizing reasoning and acting in language models," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023, arXiv:2210.03629.
- [2] T. Schick *et al.*, "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023, arXiv:2302.04761.
- [3] S. G. Patil, T. Zhang, X. Wang and J. E. Gonzalez, "Gorilla: Large language model connected with massive APIs," arXiv:2305.15334, 2023.
- [4] Y. Qin *et al.*, "ToolLLM: Facilitating large language models to master 16000+ real-world APIs," in *Proc. ICLR*, 2024, arXiv:2307.16789.
- [5] C. E. Jimenez *et al.*, "SWE-bench: Can language models resolve real-world GitHub issues?" in *Proc. ICLR*, 2024, arXiv:2310.06770.
- [6] J. Yang *et al.*, "SWE-agent: Agent-computer interfaces enable automated software engineering," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2024, arXiv:2405.15793.
- [14] s.github.com/rest
- [7] K. Greshake *et al.*, "Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection," in *Proc. ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023, arXiv:2302.12173.
- [8] OWASP Foundation, "OWASP Top 10 for Large Language Model Applications." [Online]. Available: <https://owasp.org/www-project-top-10-for-large-language-model-applications/>
- [9] G. Mialon *et al.*, "Augmented language models: A survey," *Trans. Machine Learning Research*, 2023, arXiv:2302.07842.
- [10] Anthropic, "Model Context Protocol specification." [Online]. Available: <https://modelcontextprotocol.io>
- [11] Corsair, "Corsair documentation and GitHub plugin," [add URL and access date].
- [12] Vercel, "AI SDK documentation." [Online]. Available: <https://ai-sdk.dev>
- [13] GitHub, "GitHub REST API documentation." [Online]. Available: <https://doc>