International Journal of Scientific Research in Engineering and Management (IJSREM)
Volume: 10 Issue: 04 | April - 2026 SJIF Rating: 8.659 ISSN: 2582-3930

ScrapeFlow: An AI-Augmented Visual Web Scraping and Data Workflow Platform

Prof. Seema Pawar
Dept. of Artificial Intelligence and Data Science
Vasantdada Patil Pratishthan's College of Engineering
Mumbai, India

Ayush Gupta
Dept. of Artificial Intelligence and Data Science
Vasantdada Patil Pratishthan's College of Engineering
Mumbai, India

Omkar Manjrekar
Dept. of Artificial Intelligence and Data Science
Vasantdada Patil Pratishthan's College of Engineering
Mumbai, India

Ritesh Trimukhe
Dept. of Artificial Intelligence and Data Science
Vasantdada Patil Pratishthan's College of Engineering
Mumbai, India

Ganesh Dubey
Dept. of Artificial Intelligence and Data Science
Vasantdada Patil Pratishthan's College of Engineering
Mumbai, India

***Abstract*—Building a complete web data pipeline today means choosing between two incompatible options: API orchestrators like n8n and Zapier offer visual DAG editors but no native browser automation, while scraping frameworks such as Apify provide headless browser control at the cost of requiring developer expertise [4] [5]. Prior academic work on web automation scripting [1], natural-language parameterization [2], and hierarchical scraping visualization [3] advanced individual concerns yet remained dependent on brittle DOM-structural selectors and did not converge visual workflow design, browser execution, and AI-driven extraction into a single system. To close this gap, we built ScrapeFlow—an open-source platform on Next.js 16 that brings together a typed-handle DAG editor, a phase-based Puppeteer execution engine with merge-point-aware branching and three loop variants, and a provider-agnostic AI layer covering seven LLM backends including local Ollama. A capability evaluation against eight commercial platforms across ten feature dimensions shows that ScrapeFlow is the only system satisfying all dimensions simultaneously, and a case study demonstrates an end-to-end pipeline from AI-powered data extraction through sentiment analysis to conditional webhook delivery. Taken together, these results suggest that semantic web orchestration is a practical design approach—one that gives non-technical users access to enterprise-grade control-flow without writing a single line of code.**

***Index Terms*—web scraping, workflow automation, visual programming, large language models, directed acyclic graph, browser automation**

I. Introduction

Today, collecting and working with web data has become something almost every researcher, business, or developer ends up needing at some point. But the tools available right now make it harder than it should be. You either get a nice visual platform like Zapier or Make that handles workflows well but can't actually open a browser, or you use something like Apify which does the browser part but requires you to write code [4] [5]. And if you also need AI to process the extracted data, that's a third separate tool on top. Nobody has really put all three together in one place.

On the research side, some good work has been done on individual pieces of this problem. Krosnick and Oney [1] looked at exactly why writing automation scripts trips up developers, and that work grew into a system for parameterizing browser macros using plain language [2], then a visual storyboard approach to help users see what their scraper was actually doing [3]. Each paper tackled something real. Still, all three kept CSS and XPath selectors as a given — that dependency never went away — and none of them attempted to fold visual workflow design, browser execution, and AI extraction into one thing a non-programmer could actually pick up and use.

Commercially, the same split exists [4] [5].API orchestrators such as n8n, Make, and Zapier provide mature DAG editors with hundreds of integrations yet offer no native headless-browser automation. Dedicated scraping platforms such as Apify deliver browser control and anti-detection measures but require programmatic expertise. Bridge tools such as Bardeen and Octoparse narrow the gap yet compromise on scaling, merge-point-aware branching, or impose closed cloud lock-in.

ScrapeFlow is our attempt to close this gap. We built it as a single open-source platform on Next.js 16 that unifies a visual DAG editor, a phase-based browser engine, and a provider-agnostic AI layer. In practice, this means a non-technical user can build, run, and schedule a complete scraping pipeline without writing any code.

This paper makes the following contributions:

1) A typed-handle DAG editor supporting 37 task types across six categories, merge-point-aware conditional branching, and a three-variant unified loop system with visual structure indicators.
2) A phase-based execution engine with stealth Puppeteer browser automation and first-class flow-control signals including break, continue, stop, and return.
3) A provider-agnostic AI layer spanning seven LLM providers—including local Ollama inference—that delivers semantic extraction, summarization, sentiment analysis, and natural-language workflow generation.
4) A capability evaluation against eight commercial platforms across ten feature dimensions, demonstrating that ScrapeFlow is the only system satisfying all dimensions simultaneously and thereby closing the cumulative gap identified in prior research [1] [2] [3].

II. Problem Statement

Web data extraction pipelines increasingly require three capabilities in combination: visual workflow design for non-technical users, browser-native automation to handle JavaScript-rendered pages and anti-bot mechanisms, and AI-driven enrichment to interpret unstructured content. In our observation, no existing tool satisfies all three within a single deployable system. Web data extraction pipelines today generally need three things working together: a visual way to design the workflow, a browser that can handle modern JavaScript-heavy websites and bypass bot detection, and some kind of AI layer to make sense of unstructured content. The problem is that no single tool actually does all three. Tools like n8n or Make do a fine job on the workflow side, but the moment you actually need to open a browser and load a page, they just hand it off to some outside scraping service. It can create risks, and we may not be able to trace them because the system is complex. Dedicated scraping tools handle the browser side fine but they're built for developers, not general users. And AI enrichment, if the platform supports it at all, usually comes in as an afterthought at the end rather than being a proper part of the pipeline.

What we realized during the development process is that this isn't really a features problem — it's more of an architectural one. These three capabilities were built by completely different communities, with different assumptions and different abstractions, so they never naturally fit together. ScrapeFlow came out of trying to fix that, by building all three as equal, first-class parts of a single system instead of bolting them on top of each other.

III. Related Work

Krosnick and Oney [1] ran a two-part study where they observed developers building web automation scripts and then tested a modified IDE with better selector tooling. This work sits within a broader tradition of visual programming [8] and programming by demonstration research [9]. What they found was that the biggest bottleneck wasn't logic or control flow — it was just writing the right CSS or XPath selector to target the right element. They came up with five design guidelines around giving developers better contextual feedback while they write selectors. One limitation we noticed is that the fix is still working around the same root cause: you still need a selector, it's just slightly easier to write one. ScrapeFlow takes a different approach — instead of helping you write better selectors, we just removed the need for them entirely by letting an LLM figure out the structure from the page content. One limitation we noted is that the intervention operates exclusively at the IDE level—it makes selector writing less painful, but the underlying dependency on DOM structure remains. ScrapeFlow takes a different approach: rather than improving how developers write selectors, it removes the need for selectors entirely through semantic AI extraction.

ParamMacros [2] shifted the target audience from developers to end users by enabling natural-language parameterization of browser macros. Users annotated recorded actions with free-text descriptions, and a structural inference algorithm based on prefix/suffix XPath pattern matching generalized those annotations into reusable parameters. A controlled user study demonstrated that non-programmers could successfully parameterize queries across multiple websites. In our testing of similar sites, this structural fragility was a recurring issue—even minor layout changes broke recorded macros. ParamMacros does not address this because its inference is inherently tied to DOM structure at recording time. ScrapeFlow's extractWithAI() function sidesteps this by reasoning over page content semantically, making it resilient to structural changes so long as the meaning of the data remains stable.

ScrapeViz [3] introduced a visual storyboard paradigm that provides bi-directional data provenance linking between output table cells and captured page snapshots. A 12-participant evaluation validated that the storyboard representation improved user comprehension of scraping pipelines, accelerated anomaly detection, and enabled real-time validation during macro authoring. This is useful for understanding what went wrong, but ScrapeViz offers no path to recovery—the user must manually fix the selector and re-run. We found this to be a meaningful gap: transparency without recoverability only partially solves the usability problem. ScrapeFlow addresses this through its per-phase execution viewer combined with prompt-refinement and selective re-execution, enabling iterative repair without rebuilding the entire pipeline.

Same split exists in commercial tools too [4] [5]. Platforms like n8n [12], Make and Zapier have decent visual editors, and n8n even lets you run models locally via Ollama, but none of them can open a browser on their own. You always end up calling some third-party service just to do the scraping part. Apify on the other hand is powerful for browser automation but its basically developer-only, theres no visual builder for someone without coding knowledge. Tools like Bardeen and Octoparse try to sit somewhere in the middle, but browser extensions dont scale well, most of them dont handle merge-point branching properly, and they all require you to use their own cloud. From what we could find, no tool that you can self-deploy covers all five things at once — a proper visual editor,

real browser automation with anti-bot, merge-point branching, local LLM support and natural language workflow building.

IV. System Architecture

A. Overview

ScrapeFlow is architected as a monolithic Next.js 16 application following a vertical-slice modular pattern. The system is represented into four layers, which are given below in Table I.

TABLE I
ScrapeFlow Four-Layer Architecture

Layer	Responsibility	Key Technologies
Frontend / Visual	DAG workflow editor, dashboard analytics, execution viewer	React 19, @xyflow/react 12, TanStack React Query, Recharts
Backend / API	Type-safe RPC procedures, authentication, session management, data access	tRPC 11, Better Auth, Drizzle ORM, Neon PostgreSQL
Execution Engine	Phase-based DAG traversal, loop handling, branch evaluation, browser automation	Puppeteer with stealth plugin, Cheerio, @sparticuz/chromium-min
AI Layer	Multi-provider LLM abstraction, structured extraction, sentiment analysis, summarization, workflow generation	OpenAI, Anthropic, Gemini, Groq, Ollama, OpenRouter

The four-layer architectural design of ScrapeFlow is represented below in Fig. 1.

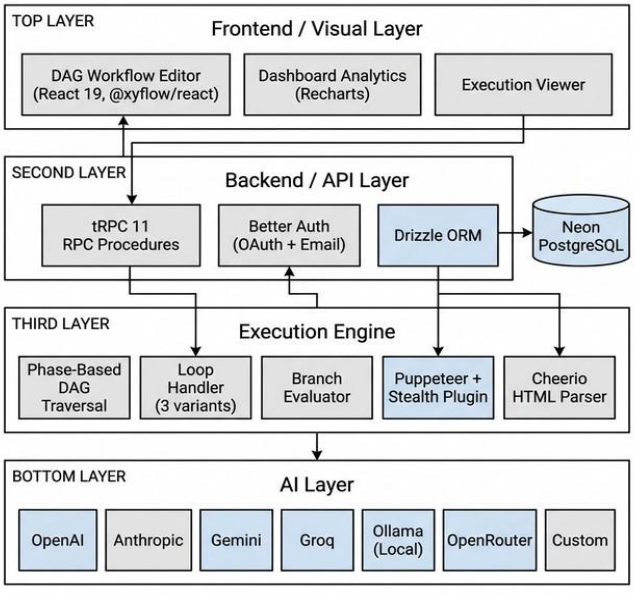

Fig. 1. The four-layer architecture of ScrapeFlow.

B. Visual Workflow Builder

The visual workflow builder is a directed acyclic graph (DAG) editor built on the @xyflow/react 12 library. It exposes 37 task types organized into six color-coded categories: Browser Actions, Data Extraction, Data Processing, Flow Control, Data Analytics, and Output & Delivery. Each category uses a distinct card color to aid visual parsing of complex workflows.

Connections between nodes are governed by an eight-type typed handle system. The isValidConnection() call-back enforces that only handles of matching types—String, Browser_Instance, Credential, Session_Param, Select, Extrac-tion_Fields, Filter_Conditions, and Chart_Config—can form edges, preventing invalid data flow at design time rather than at execution time.

The editor provides undo/redo via reference-based history stacks limited to 50 entries, avoiding unnecessary re-render cycles. A crash-recovery mechanism auto-saves the graph state to localStorage with timestamps. On subsequent page load the system compares the local timestamp against the server-side updatedAt field and, if local state is newer, offers a restoration dialog. To show loops visually in the editor, we run a BFS traversal inside detectLoopStructures() that scans for matching Loop_Start–Loop_End pairs. Once found, the editor draws a dashed SVG rectangle around the nodes that belong to that loop, so users can see the loop boundary without needing to trace edges manually. Users can also type a plain-English description of what they want the pipeline to do, and the AI layer turns that into a validated DAG through convertAIWorkflowToReactFlow() (see Section III-D). Fig. 2 shows the editor with typed handles, category coloring, and the loop visualization overlay.

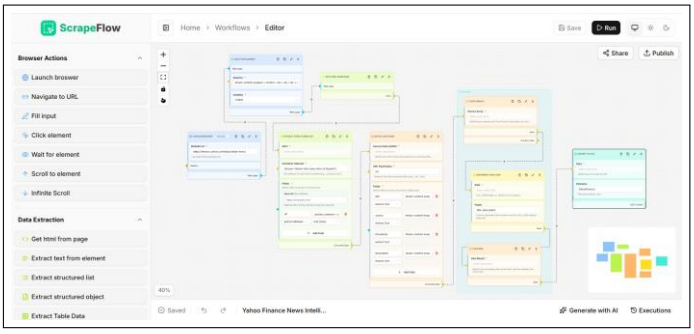

Fig. 2. ScrapeFlow visual DAG editor with typed handles, color-coded task categories, and loop visualization overlay.

C. Execution Engine

Workflow execution begins with FlowToExecutionPlan(), which performs a topological BFS sort from entry-point nodes (those with no incoming edges) and groups the resulting ordering into numbered phases [14]. A phase contains all nodes whose dependencies are satisfied by preceding phases.

The engine supports three loop variants—Loop_Start_Array (iterate over a JSON array), Loop_Start_Repeat (fixed *N*

iterations, $N \leq 10{,}000$), and Loop_Start_Until_Condition (conditional with a maximum iteration bound). All three normalize to an internal source-array representation: Repeat generates the index sequence $[0, N-1]$ and Until_Condition generates $[0, max-1]$. Because all three loop types end up as the same internal array, we only needed one execution path to handle all of them. That was a nice side effect — a bug fix in loop logic fixed all three types at once, no need to patch each one separately.

For branching, the If_Condition node handles either a basic yes/no check or something more complex with up to 12 operators and AND/OR grouping. After it runs, it marks outgoing edges as BRANCH_TRUE or BRANCH_FALSE. When shouldSkipPhaseForBranching() later decides if a node should execute, it checks all incoming edges. A node only gets skipped if every incoming edge is from a branch that didnt activate. So any node connected to both paths — we call these merge-point nodes — always runs regardless.

D. AI Integration Layer

The AI layer implements a provider-agnostic abstraction over seven LLM backends: OpenAI, Anthropic, Google Gemini, Groq, Ollama (local), OpenRouter, and a user-configurable custom endpoint. All requests pass through a unified pipeline: callAI() resolves the encrypted credential, dispatches to a provider-specific request builder via callProvider(), and normalizes the response into an AINormalizedResponse containing the content string, token usage counts, and the raw provider response. Adding a new provider requires only a request builder and a response normalizer; no consuming code changes.

The extractWithAI() function pairs with a 200+ line system prompt that enforces strict JSON-only output and implements a six-tier CSS selector priority hierarchy: test attributes, accessibility roles, form attributes, semantic class names, partial string matching, and element IDs. Recent work on LLM-based web agents [10] and structured web extraction [11] demonstrates the viability of this approach. The prompt instructs the model to avoid hashed class names and framework-generated identifiers, addressing the brittle-selector problem identified in prior work [1] [3]. If extraction doesn't work for whatever reason, the system just returns empty arrays or nulls rather than crashing the whole workflow. This made it much easier to handle partial failures without writing a bunch of error-handling code in every pipeline.

Workflow generation uses a prompt inside generateWorkflowAIPrompt() that documents all 37 task types — their input names, output names, what types they accept, and how they're supposed to connect. The prompt also includes over 15 worked examples of common patterns. The prompt includes 15+ worked examples covering common scraping patterns and supports two modes: *replace*, which generates a new workflow from scratch, and *modify*, which edits an existing workflow using serialized context. Generated structures are validated against the TaskRegistry before insertion into the editor.

All provider API keys are encrypted at rest using AES-256-GCM with a key derived via scryptSync from the application authentication secret. The encryption module carries a server-only import guard that prevents client-side code from accessing decryption routines.

V. Implementation

A. Design Rationale

During development, five architectural decisions had an outsized effect on how the system behaved. We present each as a problem–decision–trade-off triple.

Serialized JSON definitions over normalized graph tables. Workflow definitions (nodes and edges) and execution plans are stored as serialized JSON in single text columns rather than in normalized node and edge tables. This permits atomic save and load of the entire graph state, simplifies undo/redo to storing and restoring a single string, and elimi-nates impedance mismatch between the ReactFlow serializa-tion format and the persistence layer. The obvious trade-off is losing the ability to query individual nodes via SQL—but in practice this never came up, since workflows are always loaded and persisted as a unit.

Phase-based execution over event-driven activation. The engine works through phases sequentially rather than using an event bus. Browser automation essentially requires this — you can't extract anything before the page loads, and authenticated requests have to come after login. What we didn't anticipate was how useful the per-phase logging would turn out to be during development. Each phase independently stores its own status, inputs, outputs, and logs, so a mid-run failure doesn't force a full restart. You pull up the broken phase, see exactly what went in, and work backwards from there.

B. Technology Stack

The below Table II represents the core libraries which have been used in the project along with the reasons for why we choose them.

C. Runtime Execution Flow

Fig. 3 shows how actually a user-defined workflow moves through the system, from the initial graph to the final output. The four stages are explained below.

Stage 1: User Workflow Creation The user creates a workflow visually as a directed node graph. It consists of several nodes such as a start point, decision nodes, action nodes, and an end node. This graph is then converted into a JSON format and passed to the next stage.

Stage 2: Execution Engine. The engine reads the entier workflow and uses topological sort to decide the execution plan. Each phase groups nodes whose dependencies are already resolved.

Stage 3: Runtime Environment Three main modules run inside the runtime. The Task Executor creates multiple executor instances and runs them in parallel whenever possible. The AI Processing Module takes care of tasks that need an LLM whereas, The Browser Automation Module handles the

TABLE II
SCRAPEFLOW TECHNOLOGY STACK

Technology	Role	Key Justification
Next.js 16	Full-stack framework	Supports server components for SSR and allows running Puppeteer on the server in the same deployment.
@xyflow/react 12	DAG editor	Good for rendering workflow graphs and already includes features like typed connections and a minimap.
tRPC 11	API layer	Keeps client and server types automatically synchronized, so manual API type definitions are not needed.
Drizzle ORM	Database access	SQL queries are checked during compilation, and database migrations can be written directly in code.
Neon PostgreSQL	Persistent storage	Provides serverless PostgreSQL with an HTTP driver that works well with edge runtimes.
Puppeteer 24 + stealth	Browser automation	Runs headless Chrome and uses stealth patches to avoid common bot-detection checks.
Cheerio 1.1	HTML parsing	Allows fast server-side DOM parsing without needing to run a full browser.
sentiment 5	Local NLP	Uses the AFINN lexicon for sentiment analysis and works very fast without external APIs.
cron-parser 5	Scheduling	Reads cron expressions and calculates the next execution time for tasks.
Better Auth 1.3	Authentication	Supports email login and OAuth providers like GitHub and Google with secure session handling.

Puppeteer instance for tasks that require a real browser, like navigating pages or interacting with elements.

Stage 4: Output Delivery Systems Once the execution is completed, the results are sent to the outputs defined in the workflow. This can include writing to a database, sending data to cloud storage or an API, or triggering a webhook or notification.

VI. RESULTS AND ANALYSIS

A. Capability Comparison

To better understand where ScrapeFlow stands, we compared it with six well-known platforms: n8n, Zapier, Make, Octoparse, Apify, and Bardeen. The comparison was based on ten important capabilities that were identified earlier as missing or limited in existing tools. These six platforms were selected because they represent the most relevant examples from a larger review of eight different tools [4] [5].

From the comparison, two general groups can be observed. Platforms such as n8n, Zapier, and Make are good for workflow automation and also provide some AI features. However,

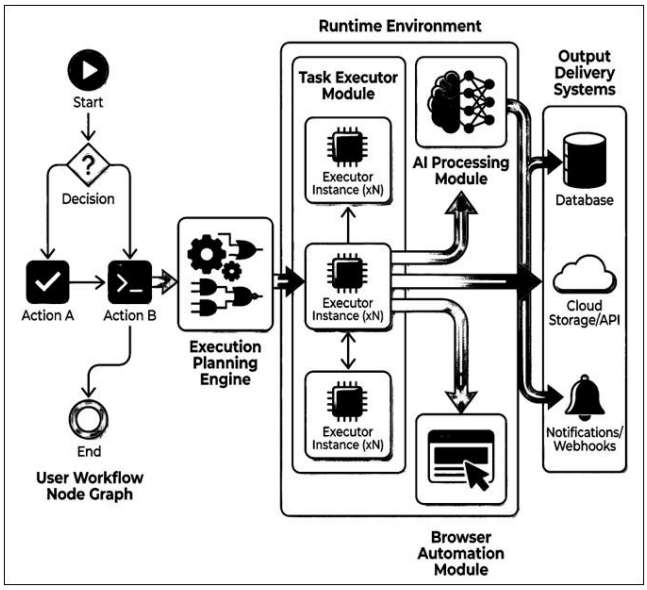

Fig. 3. Runtime execution flow: from user workflow graph to output delivery.

they cannot run a browser directly for scraping tasks. Instead, they depend on external services to perform scraping, which can introduce additional failure points and reduce control over that part of the workflow [4].

B. Case Study: Intelligent Product Monitoring Pipeline

Loop-based enrichment of product information. Next, a Loop_Start_Array node processes the list of products one at a time. For each product, the system opens the corresponding review page and collects the review content. The review text is then passed to an AI summarization step, and a sentiment value is calculated locally using the AFINN dictionary. After every product has been processed, the Loop_End node combines the enriched records into a single dataset. Tools such as Zapier do not include a loop mechanism for this kind of operation. Platforms like Octoparse and Bardeen support pagination, but they are not designed for flexible per-item processing that includes AI operations within the loop.

CONCLUSION

The long-standing bifurcation between visual DAG orchestration platforms and browser-native scraping tools is an architectural artifact, not a fundamental constraint. ScrapeFlow shows that these three capabilities—a typed-handle DAG editor, a phase-based stealth execution engine, and a seven-backend AI layer—can in fact coexist within a single deployable system. A capability evaluation against eight commercial platforms across ten feature dimensions showed that ScrapeFlow is the only tool satisfying all dimensions simultaneously, bridging the gap between API orchestrators that lack native browser automation and scraping frameworks that require programmatic expertise. Against the three gaps from prior work: the DOM selector bottleneck of [1] is removed through semantic AI extraction; the structural fragility of [2]'s XPath

TABLE III
CAPABILITY COMPARISON ACROSS TEN FEATURE DIMENSIONS

Capability Dimension	ScrapeFlow	n8n	Zapier	Make	Octoparse	Apify	Bardeen
Visual DAG builder	Yes	Yes	Yes	Yes	Yes	Partial	Yes
In-pipeline AI extraction	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Headless browser + anti-bot	Yes	No	No	No	Yes	Yes	Yes
Loop + flow control signals	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Merge-point-aware branching	Yes	Yes	No	Partial	Basic	Yes	Basic
Multi-provider & local LLM	Yes	Yes	Partial	Partial	Partial	Yes	Partial
NL→workflow generation	Yes	Yes	Yes	Yes	Yes	No	Yes
In-pipeline analytics	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Self-hostable	Yes	Yes	No	No	Partial	Yes	No
Unified scrape→export pipeline	Yes	Partial	Partial	Partial	Yes	Yes	Yes
All 10 satisfied	**Yes**	No	No	No	No	No	No

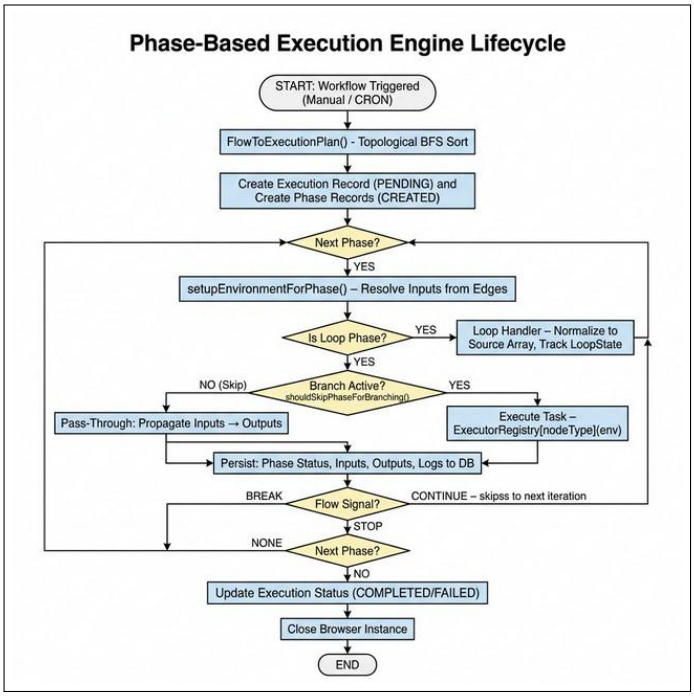

Fig. 4. Phase-based execution engine lifecycle.

inference is avoided by reasoning over content rather than markup; and the pipeline transparency of [3] is matched—and extended with active recovery through prompt refinement and selective re-execution. Future work includes parallel execution of independent phases, CAPTCHA-solving integration, and real-time collaborative workflow editing.

FUTURE SCOPE

Several directions remain open for further development of ScrapeFlow. We plan to introduce parallel phase execution for workflow nodes whose dependency graphs have no order-ing constraints, which would meaningfully reduce end-to-end pipeline latency on multi-step scraping tasks. Integration of CAPTCHA-solving services—either via third-party APIs or on-device vision models—would extend coverage to heavily protected targets. Longer term, real-time collaborative work-flow editing (similar to multiplayer document editing) would make the platform more viable in team settings. We also intend to instrument the execution engine to collect anonymized performance telemetry, which would support a quantitative follow-up study comparing ScrapeFlow against baseline tools on standardized extraction tasks.

REFERENCES

[1] R. Krosnick and S. Oney, "Understanding the challenges and needs of programmers writing web automation scripts," in *Proc. 2021 IEEE Symp. VL/HCC*, pp. 1–10.
[2] R. Krosnick and S. Oney, "ParamMacros: Creating UI automation leveraging end-user natural language parameterization," in *Proc. 2022 IEEE Symp. VL/HCC*, pp. 1–9.
[3] R. Krosnick and S. Oney, "ScrapeViz: Hierarchical representations for web scraping macros," in *Proc. 2024 IEEE Symp. VL/HCC*, pp. 1–11, doi: 10.1109/VL/HCC60511.2024.00040.
[4] Digidop, "n8n vs Make vs Zapier," 2026. [Online]. Available: https://www.digidop.com/blog/n8n-vs-make-vs-zapier
[5] Apify, "Integrate with Apify platform," 2026. [Online]. Available: https://docs.apify.com/platform/integrations
[6] W3C, "WebDriver," W3C Recommendation, 2018. [Online]. Available: https://www.w3.org/TR/webdriver/
[7] Google Chrome Team, "Puppeteer: A Node.js library for headless Chrome," GitHub, 2024. [Online]. Available: https://github.com/puppeteer/puppeteer
[8] B. A. Myers, "Visual programming, programming by example, and program visualization: A taxonomy," in *Proc. ACM CHI*, 1986, pp. 59–66.
[9] A. Cypher and D. C. Halbert, Eds., *Watch What I Do: Programming by Demonstration*. Cambridge, MA: MIT Press, 1993.
[10] S. Yao *et al.*, "WebArena: A realistic web environment for building autonomous agents," in *Proc. ICLR*, 2024.
[11] I. Gur *et al.*, "A Real-World WebAgent with Planning, Long Context Understanding, and Program Synthesis," *arXiv preprint arXiv:2307.12856*, 2023.
[12] n8n GmbH, "n8n: Fair-code workflow automation," GitHub, 2024. [Online]. Available: https://github.com/n8n-io/n8n
[13] berstend, "puppeteer-extra-plugin-stealth," GitHub, 2024. [Online]. Available: https://github.com/berstend/puppeteer-extra
[14] J. M. Hellerstein *et al.*, "Ground: A data context service," in *Proc. CIDR*, 2017.