ChatDev 2.0: A No-Code Multi-Agent Platform for Developing Everything

Yufan Dang[★†] **Shu Yao**[♣§†] **Bowen Lai**[♠] **Chenting Xu**[‡]
Ruijie Shi[◇] **Leung Wai-Shing**[★] **Huatao Li**[♣] **Chen Qian**[♣*] **Zhiyuan Liu**[★*]
[★]Tsinghua University [♣]Shanghai Jiao Tong University [§]Shanghai Innovation Institute
[♠]Fudan University [‡]Peking University [◇]Massachusetts Institute of Technology
dangyf25@mails.tsinghua.edu.cn yao.shu2004@outlook.com
qianc@sjtu.edu.cn liuzy@tsinghua.edu.cn

Abstract

Large language model (LLM)-based multi-agent systems (MAS) have shown strong potential for solving complex tasks, yet their development forces a tradeoff: code frameworks are expressive but engineering-intensive, while no-code builders simplify authoring but constrain agent interactions to author-defined workflows. We present **ChatDev 2.0: DevAll** (hereafter DevAll), a no-code platform for building, executing, and inspecting heterogeneous MAS that delivers both high expressiveness and ease of use. In terms of expressiveness, DevAll pairs a declarative executable graph abstraction with a cycle-aware execution engine, so that heterogeneous agents and dynamic and cyclic interactions can be represented and executed within a single framework. For ease of use, an integrated visual interface lets users author, run, monitor, and inspect MAS, including human-in-the-loop steps, entirely without writing code. Experiments demonstrate that DevAll reproduces state-of-the-art MAS across three representative tasks at competitive performance and without task-specific orchestration code, highlighting its effectiveness as a general-purpose platform for LLM-based MAS. DevAll is available at https://github.com/OpenBMB/ChatDev.

1 Introduction

LLM-based multi-agent systems (MAS) coordinate specialized agents to decompose complex tasks, exchange intermediate results, and respond to runtime feedback (He et al., 2025; Chen et al., 2024; Wu et al., 2024). They have been applied to software engineering (Qian et al., 2024; Hong et al., 2024; Jimenez et al., 2024), deep research and information synthesis (Chen et al., 2025; Huang et al., 2025), and structured content and visualization generation (Chen et al., 2026; Yang et al., 2024).

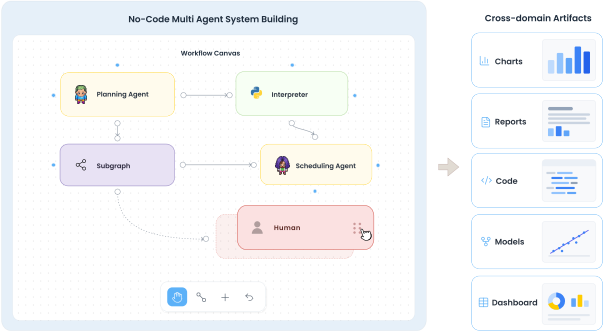

Figure 1: Users drag heterogeneous components onto the canvas and connect them into executable multi-agent systems. DevAll supports cross-domain artifact generation, including charts, research reports, and so on.

As multi-agent systems grow in scale and heterogeneity, building and maintaining them requires developers to configure agents and tools, specify communication and orchestration, manage shared state, and diagnose failures across long execution traces (Dibia et al., 2024; Gao et al., 2024; Cemri et al., 2025). Programming-oriented frameworks such as AutoGen, MetaGPT, AgentScope, and LangGraph provide reusable abstractions for these fundamental primitives (Wu et al., 2024; Hong et al., 2024; Gao et al., 2024; LangChain, 2026). Because customizing these frameworks beyond their built-in abstractions typically requires directly programming agent communication and orchestration logic, it can make development cumbersome and raise the barrier to rapid prototyping for users with limited programming experience (Dibia et al., 2024; Gao et al., 2024).

To lower this development barrier, no-code and visual agent builders, including AutoGen Studio, AI2Apps, Dify, Langflow, and Flowise, offer reusable canvas components (Dibia et al., 2024; Pang et al., 2024; LangGenius, 2026a; Langflow, 2026a; Flowise AI, 2026b). However, agents remain steps in author-defined workflows: users must predefine control flow and loop boundaries and

[†]Equal Contribution.
[*]Corresponding Author.

Proceedings of the 2026 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pages 1–12
October 24-29, 2026 ©2026 Association for Computational Linguistics

manually bind persistent outputs to later agent inputs (LangGenius, 2026b; Langflow, 2026b; Flowise AI, 2026a). This state plumbing becomes cumbersome when multiple feedback cycles interact, and some platforms further restrict nested loops (weijiashao, 2025), complicating MAS patterns with evolving contexts such as planner–executor and generator–reviewer–reviser loops (Fourney et al., 2024; Chen et al., 2025; Qian et al., 2024; Jimenez et al., 2024; Li et al., 2025).

We therefore present **ChatDev 2.0: DevAll** (hereafter DevAll), a no-code platform that combines an intuitive user interface for high-level interaction with a robust underlying engine as shown in Fig 1. Specifically, this engine integrates two core subsystems: (i) the MAS Compilation Engine, which compiles seven heterogeneous node types and semantic edges that decouple data flow from control flow, dynamically assembling agent contexts from runtime messages; and (ii) the MAS Execution Engine, which is powered by our proposed **CADET** (**C**ycle-**A**ware **D**ynamic **E**xecution **T**opology) algorithm—a cycle-aware scheduler that orchestrates multiple and nested feedback cycles while preserving iteration-level states, thereby enabling the correct and efficient execution of MAS graphs with arbitrary cycles. As an integrated whole, DevAll enables users to author, execute, and inspect dynamic MAS without writing any task-specific orchestration code.

To evaluate DevAll's expressiveness, we reconstruct representative systems for scientific data visualization, deep research report generation, and software development using only declarative graph specifications. Across the three scenarios, DevAll reproduces the core workflows of specialized systems and achieves competitive performance within a unified platform. This paper makes three contributions:

- We open-source DevAll, a no-code platform for authoring, executing, and inspecting heterogeneous multi-agent systems.
- We enable the execution of MAS graphs with arbitrary cycles via **CADET**, supporting rich and dynamic multi-agent interaction patterns.
- We validate DevAll across visualization, deep research, and software development, demonstrating that declarative graphs can reproduce representative specialized workflows with competitive performance.

2 Related Work

LLM-Based Multi-Agent Systems. LLM-based multi-agent systems coordinate specialized agents through role decomposition, message exchange, and feedback to solve complex tasks (Chen et al., 2024; Wu et al., 2024). General-purpose frameworks such as AutoGen (Wu et al., 2024) and AgentScope (Gao et al., 2024) provide programmable abstractions for stateful agents, tools, messaging, and orchestration. AutoGen further offers GraphFlow for explicitly specified conditional and cyclic agent graphs and Studio for visual team configuration and testing (Microsoft, 2026a; Dibia et al., 2024). Other graph-based methods represent or optimize agent collaboration structures (Zhuge et al., 2024; Zhang et al., 2025; Qian et al., 2025). Despite their broad use in software engineering, deep research, and visualization, specifying and debugging multi-agent orchestration remains engineering-intensive (Qian et al., 2024; Prabhakar et al., 2025; Chen et al., 2026; Gao et al., 2024; Wang et al., 2026), motivating infrastructure that preserves MAS-level interaction semantics while reducing orchestration code.

No-Code and Visual Agent Builders. No-code and visual platforms expose agents, tools, retrieval components, and control operators on a canvas (Dibia et al., 2024; LangGenius, 2026a; Flowise AI, 2026b; Langflow, 2026a; Coze, 2026; OpenAI, 2026; Microsoft, 2026b). Although these platforms increasingly support conversation memory and shared workflow state, their primary abstraction remains workflow-centric: the workflow owns control and state, while agents are invoked as steps whose inputs are assembled through node parameters, variable references, or prompt templates. Cyclic behavior is likewise expressed through explicit workflow constructs: Dify uses Loop and Iteration containers (LangGenius, 2026b), Langflow provides a list-oriented Loop component (Langflow, 2026b), and Flowise uses Iteration subflows and an explicit jump-back Loop node (Flowise AI, 2026a). Authors must therefore delineate loop scopes and explicitly route persistent outputs or shared variables across interacting feedback paths; some visual editors further restrict direct composition of nested loop constructs (weijiashao, 2025). DevAll instead adopts an agent-centric abstraction: each agent owns and evolves its local state, while interaction contexts are accumulated and propagated automatically along semantic

edges. Cyclic and acyclic interactions use the same edge abstraction, so the visual graph directly corresponds to the MAS being executed, without special loop nodes.

3 System Design

DevAll is a no-code platform for developing heterogeneous artifacts across scenarios like data visualization, software engineering, and deep research. Its two core components are the *MAS Compilation Engine*, which turns declarative specifications into executable graphs, and the cycle-aware *MAS Execution Engine*, which schedules and runs those graphs. Together they decouple authoring from runtime: users configure agents, tools, and dependencies on a visual canvas, while the engine handles instantiation, scheduling, and monitoring. When a request runs, DevAll binds inputs, routes information through connected nodes, and surfaces outputs and traces for inspection and optional human feedback.

3.1 MAS Compilation Engine

The first engine, the *MAS Compilation Engine*, compiles a human-readable YAML specification (Amazon Web Services, n.d.), which serves as the canonical on-disk representation for distributing and sharing MAS graph designs, into an executable directed graph (Gao et al., 2024; Yun et al., 2026), which provides a uniform abstraction over heterogeneous multi-agent systems.

$$\mathcal{G} = (\mathcal{V}, \mathcal{E}, \Theta)$$

The specification declares the global configuration Θ, which holds graph-level runtime settings such as shared memory, model and tool registries, workspace, and permissions, together with the nodes and edges that the compilation engine instantiates below.

The node set $\mathcal{V}$ defines the executable units of the MAS. During graph compilation, the engine instantiates each node through a registry that maps node types to configuration schemas and executors, enabling a unified interface while preserving type-specific semantics. The seven built-in node types in table 4 capture the common operations observed across practical MAS workloads.

The edge set $\mathcal{E}$ carries the abstraction that distinguishes DevAll. Unlike an edge that merely forwards context, a *semantic edge* specifies both how information is passed and how downstream execution is activated, expressed as a tuple $e = (u, v, c_e, \phi_e, \psi_e)$, where $\mathrm{src}(e) = u$ and $\mathrm{dst}(e) = v$ denote the source and target nodes, respectively. The activation condition c_e checks whether the edge should fire after the source u emits an output; the data-flow policy ϕ_e prescribes how that output is merged, transformed, retained, or cleared in the target context of v; and the control-flow policy ψ_e governs how v's scheduling state is updated. By decoupling data movement from scheduling triggers, DevAll can route information and orchestrate execution independently, enabling richer MAS semantics than fixed workflows. MAS compilation engine resolves each edge's endpoints and attaches these policies from the specification, turning the edge into an executable communication rule rather than a static link.

After instantiating all nodes and edges, the compilation engine validates graph consistency, and passes the resulting graph $\mathcal{G}$ to the MAS Execution Engine.

3.2 MAS Execution Engine

The MAS Execution Engine interprets $\mathcal{G}$ through two primitives: *node execution* and *semantic edge propagation*. A node becomes eligible to run when it is triggered by the initial user request or by an activated incoming edge; its executor consumes the local context, performs node-specific computation, generates artifacts, and emits messages. The runtime then evaluates the node's outgoing semantic edges according to Section 3.1: each active edge applies its data-flow policy ϕ_e to the target context and its control-flow policy ψ_e to the target execution state. Nodes whose readiness conditions become satisfied join the execution frontier. For an acyclic graph, these primitives advance the frontier along a topological order, yielding a deterministic layer-wise schedule. This is the baseline execution model we preserve.

MAS graphs may contain cycles, for which no topological order exists. Consequently, the baseline layer-wise schedule cannot be applied directly. We therefore introduce CADET to address this challenge by condensing cyclic regions into executable strongly connected components (SCCs) subgraphs, yielding an acyclic condensation graph for global scheduling while handling each SCC through scoped iterative execution, as illustrated in Figure 3.

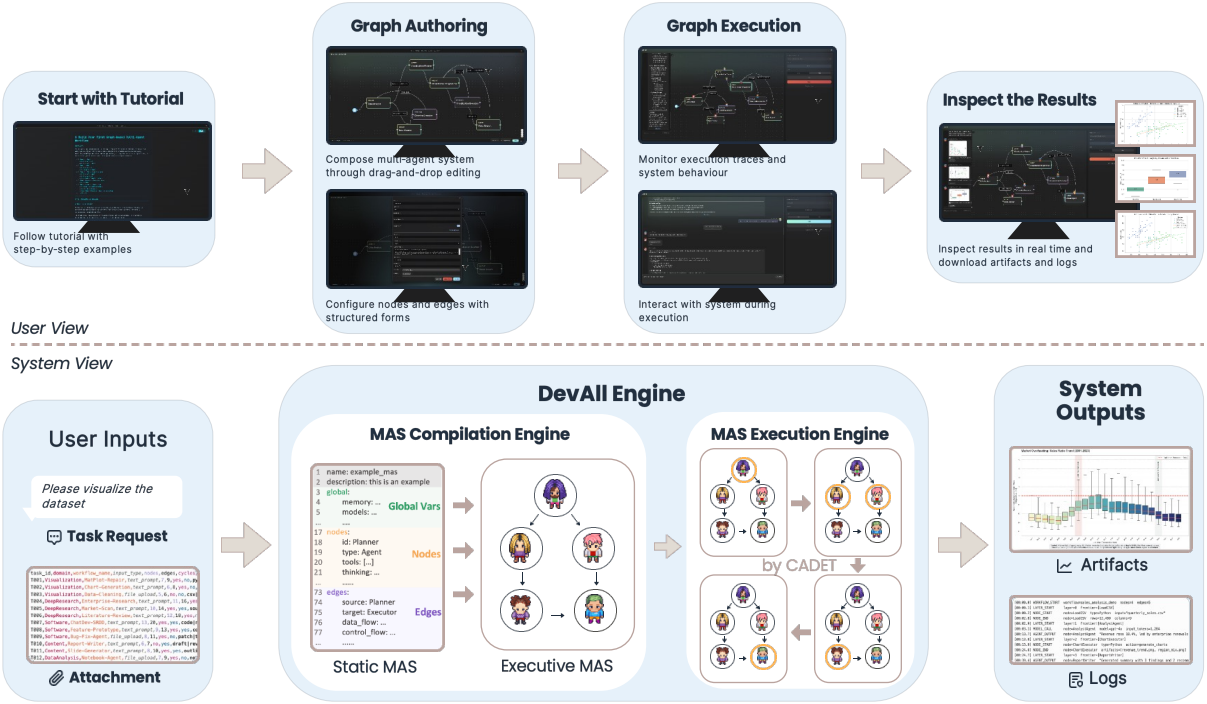

Figure 2: Overview of DevAll. The upper part shows the user interface for graph authoring, execution launching, and result inspection; the lower part presents the underlying engine, which includes the MAS Compilation Engine that compiles declarative specifications into executable graphs, and the CADET runtime that schedules cycle-aware execution over semantic edges to produce final results along with intermediate traces and artifacts.

Graph condensation CADET decomposes $\mathcal{G}$ into SCCs $\{\mathcal{C}_1, \ldots, \mathcal{C}_K\}$ (Tarjan, 1971). Singleton SCCs without self-loops remain ordinary nodes, while cyclic SCCs, including multi-node SCCs and singleton self-loop components, are contracted into *condensed nodes* (Bang-Jensen and Gutin, 2009). Let

$$\mathcal{V}^\dagger = \{\mathcal{C}_1, \ldots, \mathcal{C}_K\},$$
$$\mathcal{E}^\dagger = \big\{(\mathcal{C}_i, \mathcal{C}_j) \;\big|\; i \neq j,\ \exists e \in \mathcal{E} : \mathrm{src}(e) \in \mathcal{C}_i,\quad \mathrm{dst}(e) \in \mathcal{C}_j\big\}.$$

and define the condensation graph as $\mathcal{G}^\dagger = (\mathcal{V}^\dagger, \mathcal{E}^\dagger)$. Because edges only connect different components, $\mathcal{G}^\dagger$ is a DAG and therefore schedulable like the baseline; each condensed node retains its internal subgraph $\mathcal{C}_k$ for iterative execution.

Cyclic dependency handling Inside a condensed SCC $\mathcal{C}_k$, CADET executes it as a scoped iterative subgraph with internal edges $\mathcal{E}_k = \{(u, v) \in \mathcal{E} \mid u, v \in \mathcal{C}_k\}$. Starting from the unique active entry of $\mathcal{C}_k$, CADET identifies $\mathcal{B}_k$ recursively: at each SCC scope, it marks the internal edges targeting that scope's active entry as back edges and applies the same rule to any cyclic SCC remaining after those edges are removed, partitioning the internal edges as

$$\mathcal{E}_k = \widehat{\mathcal{E}}_k \cup \mathcal{B}_k, \qquad \widehat{\mathcal{E}}_k \cap \mathcal{B}_k = \emptyset.$$

The non-back edges $\widehat{\mathcal{E}}_k$ form an acyclic skeleton and execute within the current iteration, while back edges $\mathcal{B}_k$ are treated as loop-carried dependencies whose outputs are deferred to the next iteration. Each iteration activates nodes triggered by incoming external edges or back edges from the previous round. If an active edge exits $\mathcal{C}_k$, CADET propagates its output to the downstream node in $\mathcal{G}^\dagger$; otherwise, if back edges remain deferred, it proceeds to iteration $t + 1$ with updated contexts and trigger states. A configurable iteration limit bounds this process and ensures termination.

Layer-wise scheduling On the acyclic scheduling view $\mathcal{G}^\dagger$, CADET first computes topological layers

$$\mathcal{L}^\dagger = (\mathcal{L}_0^\dagger, \ldots, \mathcal{L}_M^\dagger) = \mathrm{TopoLayers}(\mathcal{G}^\dagger).$$

The user request is bound to the configured start nodes, which initialize the trigger states for the first scan. CADET then scans the layers in order. At layer t, an item $x \in \mathcal{L}_t^\dagger$ is executed only when $\mathrm{Ready}_t(x)$ holds, where Ready_t captures

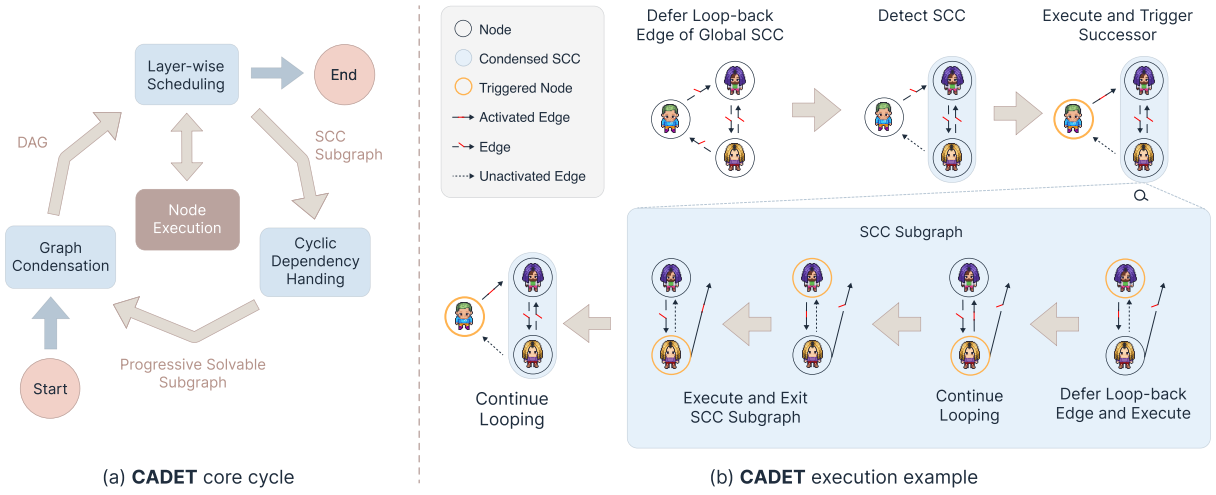

Figure 3: CADET execution over MAS graphs. CADET contracts cyclic regions into condensed nodes for layer-wise scheduling, and executes the corresponding SCC subgraphs through a scoped iterative process.

both topological dependency order and runtime trigger states induced by the user request and activated semantic edges. An ordinary node consumes its current trigger state, executes on its local context, and propagates outputs through activated outgoing edges. When CADET encounters a condensed node $\mathcal{C}_k$, it expands the node and delegates execution to the cyclic dependency handling described above.

3.3 System Interface

Figure 2 shows DevAll's integrated no-code interface, which serves as the user-facing layer over the MAS Compilation Engine and MAS Execution Engine. The interface organizes MAS development into four connected views: Tutorial, Authoring, Execution, Inspection. Together, these views support example-based onboarding, visual graph construction, interactive execution, and batch-level experimentation, while keeping both the graph structure and runtime behavior inspectable.

Graph authoring The Tutorial view provides example MAS graphs that users can inspect and reuse as starting points. In the authoring view, users construct a MAS by placing heterogeneous node types on a canvas and connecting them with semantic edges. Configuration panels are generated from metadata exported by the MAS Compilation Engine. Each node type, edge processor, memory store, and related option declares its own configurable fields, and the interface renders the relevant fields recursively after a concrete type or option is selected. This metadata-driven design improves system extensibility while preserving the no-code user experience: users edit MAS components through structured controls, and developers can add new component types by registering their configuration metadata rather than designing a separate interface panel.

Interactive execution The Launch view binds a user request and optional input files to the selected executable graph, then passes them to the configured start nodes. During execution, it receives runtime events from the MAS Execution Engine and displays node status, intermediate messages, and generated artifacts as they are produced. If execution reaches a Human node, DevAll pauses the corresponding path, collects feedback through the same interface, and resumes downstream execution after the feedback is submitted. The completed run remains inspectable through its node-level traces, execution logs, token-usage records, and workspace artifacts. Users can download execution records for offline inspection, sharing, or subsequent reuse.

Experiments For repeated runs and batch experiments, DevAll provides a Laboratory interface together with a Python SDK for programmatic execution and large-scale evaluation. This separation keeps the launch-time interface lightweight while still supporting systematic inspection and evaluation when users need it.

4 Experiment

4.1 Experimental Setup

The goal of our evaluation is not to surpass domain-specific systems, but to test whether a single no-

Table 1: Detailed metric breakdown across benchmarks. For MatPlotBench, M1–M3 denote execution pass rate, code quality score, and visualization success rate; for DeepResearchBench, M1–M4 denote comprehensiveness, insight, instruction-following, and readability; for SRDD, M1-M3 denote completeness, executability, and consistency, respectively. Overall denotes the benchmark-specific aggregate score. Δ denotes DevAll $-$ Reference, and higher values indicate better performance. Detailed metric definitions are provided in the appendix B.

Benchmark	Method	M1	M2	M3	M4	Overall
MatPlotBench	CoDA	0.8300	0.8750	0.6730	–	0.7130
	DevAll	**0.9900**	**0.8950**	**0.7160**	–	**0.7950**
	Δ	+0.1600	+0.0200	+0.0430	–	+0.0820
DeepResearchBench	EDR	**0.3230**	**0.3274**	**0.3700**	**0.4192**	**0.3500**
	DevAll	0.3048	0.3132	0.3390	0.4147	0.3319
	Δ	-0.0182	-0.0142	-0.0310	-0.0045	-0.0181
SRDD	ChatDev 1.0	0.9670	**0.8380**	**0.7916**	–	**0.6574**
	DevAll	**0.9830**	0.8250	0.7762	–	0.6509
	Δ	+0.0160	-0.0130	-0.0154	–	-0.0065

code platform can reproduce the core workflow behavior of specialized MAS across heterogeneous domains. We therefore select three representative settings with substantially different output modalities: scientific data visualization on MatPlotBench (Yang et al., 2024), deep research report generation on DeepResearchBench (Du et al., 2025), and software generation on SRDD (Qian et al., 2024). These settings correspond to three workflow-based reference systems: CoDA (Chen et al., 2026), Enterprise Deep Research (Prabhakar et al., 2025), and ChatDev 1.0 (Qian et al., 2024), respectively.

For each setting, we re-express the original workflow as a declarative MAS graph in DevAll, *without introducing any task-specific orchestration code*; users author only YAML specifications of nodes and semantic edges through the platform. To isolate workflow reproduction from confounding factors, we standardize the backbone model as GPT-4o and keep tool configurations consistent across comparisons wherever possible, so that performance differences primarily reflect the fidelity of the reproduced workflow rather than variations in the underlying model or tools. In the DeepResearchBench setting, we use Serper as the search backend and GPT-5.5 as the LLM judge; in the SRDD setting, we use `text-embedding-ada-002` as the embedding model. Detailed metric definitions are provided in Appendix B.

4.2 Cross-Domain Results

Table 1 reports results across visualization, deep research, and software generation. The main observation is *generality*: the same no-code platform can reproduce three specialized workflow systems and attain competitive performance with each domain-specific implementation, without relying on task-specific orchestration code.

On MatPlotBench, the DevAll reproduction matches and slightly exceeds CoDA, with the overall score moving from 0.7130 to 0.7950. The gain is concentrated in execution pass rate (0.8300 to 0.9900), which is consistent with DevAll's unified runtime providing stable workspace management and error handling rather than changing the workflow logic itself. On DeepResearchBench, the reproduced workflow yields an overall score of 0.3319 versus 0.3500. Given that DeepResearchBench relies entirely on LLM-as-judge, which exhibits non-negligible variance, this small difference is within the expected range of evaluation noise. On SRDD, the reproduced workflow achieves higher completeness but slightly lower executability and consistency, with the overall score remaining nearly identical.

Taken together, these results show that DevAll can host structurally and behaviorally distinct multi-agent workflows within one unified platform while preserving task effectiveness. This supports our central claim that a multi-agent system can be developed, executed, and reproduced as a no-code executable object across diverse domains.

4.3 Workflow Construction and Runtime Overhead

We evaluate the overhead introduced by DevAll at two stages of the workflow lifecycle. Construction overhead captures the one-time effort required to represent and configure a workflow, quantified by

Table 2: Structural construction complexity of the reproduced workflows. Est. ops. is the sum of nodes, edges, and configuration items and serves as a proxy for canvas-level configuration operations.

Method	Nodes	Edges	Config. items	Est. ops.
CoDA	14	26	88	128
EDR	8	11	65	84
ChatDev 1.0	26	54	178	258

specification size and estimated canvas-level operations. Runtime overhead captures the additional latency introduced by CADET during graph compilation, planning, and local scheduling, excluding network requests and LLM inference.

We quantify workflow construction overhead using three structural metrics: *Nodes*, *Edges*, and *Config. items*. Nodes denote workflow components, Edges denote directed connections, and Config. items denote non-empty semantic node settings or non-default edge conditions. We define estimated operations (Est. ops.) as $N_{\text{node}} + N_{\text{edge}} + N_{\text{config}}$. As shown in Table 2, reproducing the three workflows required 84–258 estimated structural operations. In DevAll, these operations typically correspond to point-and-click interactions for placing nodes, connecting components, and selecting configuration options. Est. ops. therefore approximates structural configuration burden rather than the exact number of mouse clicks or keystrokes. Additional comparisons of YAML and reference-source sizes are reported in Appendix Table 5. Across the evaluated workflows, construction burden remained below 300 estimated operations, indicating that structurally diverse multi-agent workflows can be configured with a controlled amount of canvas-level interaction.

We next measure the additional runtime overhead introduced by CADET. We scale the number of cyclic SCCs from 1 to 16 and separately measure compilation, planning, and execution. Compilation covers YAML loading, validation, executable-graph construction, and planning. The planning benchmark isolates SCC detection, graph condensation, and topological layering on a preconstructed graph. Execution measures the local scheduling path without network requests or LLM inference.

As shown in Table 3, median compilation, planning, and execution times increased approximately linearly over the evaluated graph sizes. At the largest configuration, median compilation and execution required 16.23 and 14.46 ms, respectively, while planning required 91.96 μs. These results indicate that CADET adds modest local processing latency compared with the latency typically incurred by model inference.

Table 3: Compilation and CADET overhead scaling. Times are p50/p95 over 1,000 runs after five warm-ups; compilation includes planning.

SCCs	$\lvert\mathcal{V}\rvert/\lvert\mathcal{E}\rvert$	Compile (ms)	Plan (μs)	Execute (ms)
1	6/7	1.53/1.60	7.13/7.25	1.19/1.33
2	10/14	2.51/2.64	11.54/11.79	2.10/2.49
4	18/28	4.43/4.60	21.38/21.75	3.88/4.26
8	34/56	8.35/8.81	42.29/42.88	7.27/7.71
16	66/112	16.23/35.36	91.96/93.04	14.46/15.62

Together, these results show that DevAll balances workflow expressiveness with engineering efficiency. Across the evaluated settings, its unified abstraction supported structurally diverse multi-agent workflows while maintaining moderate construction burden and modest runtime orchestration overhead.

5 Conclusion

We presented **DevAll**, a no-code platform that treats a multi-agent system, rather than a static workflow, as the executable object. DevAll compiles a declarative YAML specification into an executable graph whose *semantic edges* decouple data flow from control flow, and runs it with **CADET**, a scheduling protocol that restores deterministic layer-wise execution to cyclic MAS graphs by condensing strongly connected components into an acyclic topology. An integrated interface lets users author, launch, monitor, and inspect these graphs without writing code, with support for human-in-the-loop intervention and trace inspection. Across visualization, deep research, and software generation, DevAll reproduces three domain-specific workflow systems within a single unified platform, attaining competitive performance without task-specific orchestration code. These results demonstrate that dynamic, feedback-driven multi-agent systems can be built, executed, and inspected as no-code executable objects. Future work includes extending the scheduling algorithm with more sophisticated loop control mechanisms, and building a wider range of representative MAS on top of DevAll to serve diverse application domains including scientific discovery, automated data analysis, and interactive education.

Limitations

Although DevAll reduces implementation effort, users must still translate task requirements into workflow structures and manually specify agent roles, prompts, dependencies, and control conditions. Thus, no-code authoring simplifies implementation but does not yet automate workflow design. Moreover, the current graph abstraction and component ecosystem may not cover every domain-specific interaction pattern, and specialized integrations may require custom extensions. As workflows grow, visual organization, modular maintenance, version evolution, and the transfer of accumulated design experience also offer opportunities for improvement. Future work will explore intelligent authoring assistance, reusable workflow templates, stronger lifecycle management, and user studies that assess authoring efficiency and learnability across different levels of expertise.

Acknowledgments

This work was supported by the Tsinghua University (Department of Computer Science and Technology)-Siemens Ltd., China Joint Research Center for Industrial Intelligence and Internet of Things (JCIIOT), and by the Shanghai Municipal Special Program for Basic Research on General AI Foundation Models (Grant No. 2025SHZDZX026D04), in collaboration with the Shanghai Artificial Intelligence Laboratory. And we thank Fu Fangyu for his contributions to the frontend development of the DevAll platform.

References

Amazon Web Services. n.d. Yaml vs json - difference between data serialization formats. https://aws.amazon.com/compare/the-difference-between-yaml-and-json/.

Jørgen Bang-Jensen and Gregory Z. Gutin. 2009. *Digraphs: Theory, Algorithms and Applications*, 2 edition. Springer Monographs in Mathematics.

Mert Cemri, Melissa Z Pan, Shuyi Yang, Lakshya A Agrawal, Bhavya Chopra, Rishabh Tiwari, Kurt Keutzer, Aditya Parameswaran, Dan Klein, Kannan Ramchandran, Matei A Zaharia, Joseph Gonzalez, and Ion Stoica. 2025. Why do multi-agent llm systems fail? In *The Thirty-Ninth Annual Conference on Neural Information Processing Systems (NeurIPS)*.

Weize Chen, Yusheng Su, Jingwei Zuo, Cheng Yang, Chenfei Yuan, Chi-Min Chan, Heyang Yu, Yaxi Lu, Yi-Hsin Hung, Chen Qian, Yujia Qin, Xin Cong, Ruobing Xie, Zhiyuan Liu, Maosong Sun, and Jie Zhou. 2024. Agentverse: Facilitating multi-agent collaboration and exploring emergent behaviors. In *The Twelfth International Conference on Learning Representations (ICLR)*.

Zehui Chen, Kuikun Liu, Qiuchen Wang, Jiangning Liu, Wenwei Zhang, Kai Chen, and Feng Zhao. 2025. Mindsearch: Mimicking human minds elicits deep ai searcher. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

Zichen Chen, Jiefeng Chen, Sercan O Arik, Misha Sra, Tomas Pfister, and Jinsung Yoon. 2026. CoDA: Agentic systems for collaborative data visualization. In *The Fourteenth International Conference on Learning Representations (ICLR)*.

Coze. 2026. Coze studio: An ai agent development platform. https://github.com/coze-dev/coze-studio.

Victor Dibia, Jingya Chen, Gagan Bansal, Suff Syed, Adam Fourney, Erkang Zhu, Chi Wang, and Saleema Amershi. 2024. AUTOGEN STUDIO: A no-code developer tool for building and debugging multi-agent systems. In *The Twenty-Ninth Annual Conference on Empirical Methods in Natural Language Processing: System Demonstrations (EMNLP)*.

Mingxuan Du, Benfeng Xu, Chiwei Zhu, Xiaorui Wang, and Zhendong Mao. 2025. Deepresearch bench: A comprehensive benchmark for deep research agents. *Preprint*, arXiv:2506.11763.

Flowise AI. 2026a. Agentflow v2 – flowise documentation. https://docs.flowiseai.com/using-flowise/agentflowv2.

Flowise AI. 2026b. Flowise documentation. https://docs.flowiseai.com/.

Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang, Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. 2024. Magentic-one: A generalist multi-agent system for solving complex tasks. *Preprint*, arXiv:2411.04468.

Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, Liuyi Yao, Hongyi Peng, Zeyu Zhang, Lin Zhu, Chen Cheng, Hongzhu Shi, Yaliang Li, Bolin Ding, and Jingren Zhou. 2024. Agentscope: A flexible yet robust multi-agent platform. *Preprint*, arXiv:2402.14034.

Junda He, Christoph Treude, and David Lo. 2025. Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead. *ACM Trans. Softw. Eng. Methodol.*, 34(5).

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. 2024. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations (ICLR)*.

Lisheng Huang, Yichen Liu, Jinhao Jiang, Rongxiang Zhang, Jiahao Yan, Junyi Li, and Wayne Xin Zhao. 2025. Manusearch: Democratizing deep search in large language models with a transparent and open multi-agent framework. *Preprint*, arXiv:2505.18105.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations (ICLR)*.

LangChain. 2026. Graph api overview – langgraph documentation. https://docs.langchain.com/oss/python/langgraph/graph-api.

Langflow. 2026a. Langflow: Low-code ai builder for agentic and rag applications. https://www.langflow.org/.

Langflow. 2026b. Loop – langflow documentation. https://docs.langflow.org/loop.

LangGenius. 2026a. Dify: Leading agentic workflow builder. https://dify.ai/.

LangGenius. 2026b. Loop – dify documentation. https://docs.dify.ai/en/use-dify/nodes/loop.

Bingxuan Li, Yiwei Wang, Jiuxiang Gu, Kai-Wei Chang, and Nanyun Peng. 2025. METAL: A multi-agent framework for chart generation with test-time scaling. In *ACL (1)*, pages 30054–30069. Association for Computational Linguistics.

Microsoft. 2026a. Graphflow (workflows): Autogen agentchat. https://microsoft.github.io/autogen/stable/user-guide/agentchat-user-guide/graph-flow.html. Accessed: 2026-08-24.

Microsoft. 2026b. Microsoft copilot studio. https://www.microsoft.com/en-us/microsoft-365-copilot/microsoft-copilot-studio.

OpenAI. 2026. Agent builder. https://developers.openai.com/api/docs/guides/agent-builder.

Xin Pang, Zhucong Li, Jiaxiang Chen, Yuan Cheng, Yinghui Xu, and Yuan Qi. 2024. Ai2apps: A visual ide for building llm-based ai agent applications. *Preprint*, arXiv:2404.04902.

Akshara Prabhakar, Roshan Ram, Zixiang Chen, Silvio Savarese, Frank Wang, Caiming Xiong, Huan Wang, and Weiran Yao. 2025. Enterprise deep research: Steerable multi-agent deep research for enterprise analytics. *Preprint*, arXiv:2510.17797.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2024. ChatDev: Communicative agents for software development. In *The Sixty-Second Annual Meeting of the Association for Computational Linguistics (ACL)*.

Chen Qian, Zihao Xie, YiFei Wang, Wei Liu, Kunlun Zhu, Hanchen Xia, Yufan Dang, Zhuoyun Du, Weize Chen, Cheng Yang, Zhiyuan Liu, and Maosong Sun. 2025. Scaling large language model-based multi-agent collaboration. In *The Thirteenth International Conference on Learning Representations (ICLR)*.

Robert Tarjan. 1971. Depth-first search and linear graph algorithms. In *12th Annual Symposium on Switching and Automata Theory (SWAT)*.

Yuheng Wang, Runde Yang, Lin Wu, Jie Zhang, Jingru Fan, Tianle Zhou, Ruoyu Fu, Huatao Li, Ruijie Shi, Siheng Chen, Weinan E, and Chen Qian. 2026. Teachmaster: Generative teaching via code. *Preprint*, arXiv:2601.04204.

weijiashao. 2025. Enhance workflow control features. GitHub issue #24851, LangGenius Dify.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. 2024. Autogen: Enabling next-gen LLM applications via multi-agent conversations. In *The First Conference on Language Modeling (COLM)*.

Zhiyu Yang, Zihan Zhou, Shuo Wang, Xin Cong, Xu Han, Yukun Yan, Zhenghao Liu, Zhixing Tan, Pengyuan Liu, Dong Yu, Zhiyuan Liu, Xiaodong Shi, and Maosong Sun. 2024. MatPlotAgent: Method and evaluation for LLM-based agentic scientific data visualization. In *The Sixty-Second Annual Meeting of the Association for Computational Linguistics (ACL)*.

Sukwon Yun, Jie Peng, Pingzhi Li, Wendong Fan, Jie Chen, James Zou, Guohao Li, and Tianlong Chen. 2026. Graph-of-agents: A graph-based framework for multi-agent LLM collaboration. In *The Fourteenth International Conference on Learning Representations (ICLR)*.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. 2025. G-designer: Architecting multi-agent communication topologies via graph neural networks. In *Forty-second International Conference on Machine Learning (ICML)*.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen

Schmidhuber. 2024. GPTSwarm: Language agents as optimizable graphs. In *The Forty-first International Conference on Machine Learning (ICML)*.

A Implementation Details

We provide omitted implementation details: built-in node types and CADET pseudocode.

A.1 Built-in Node Types

Table 4 summarizes the built-in nodes used to construct executable MAS graphs.

Table 4: Built-in node types in DevAll.

Node type	Role
Agent	LLM-backed agent configured with prompt, tools, memory, and an optional thinking module.
Human	Human-feedback checkpoint that pauses execution until input is submitted.
Python	Workspace Python executor that returns standard output.
Subgraph	Nested MAS graph defined inline or loaded from YAML.
Literal	Static message emitter.
Passthrough	Context forwarder that leaves inputs unchanged.
LoopCounter	Iteration gate that releases output after configured rounds.

A.2 CADET Execution Algorithm

CADET executes MAS graphs by scheduling the condensation graph and resolving cycles inside condensed components. Algorithms 1 and 2 give the two procedures. SEMPROP applies ϕ_e to target context and ψ_e to target state; RUN returns activated edges.

Algorithm 1 CADET Execution over the Condensation Graph

Require: Executable MAS graph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \Theta)$, start nodes $\mathcal{S}$, runtime context Γ

$\mathcal{G}^\dagger = (\mathcal{V}^\dagger, \mathcal{E}^\dagger) \leftarrow$ CONDENSE(SCCDECOMPOSE($\mathcal{G}$))
$\mathcal{F} \leftarrow$ INITFRONTIER($\mathcal{S}, \Gamma$); $S \leftarrow \emptyset$
while $\mathcal{F} \neq \emptyset$ **do**
  Execute each $x \in \mathcal{F}$ in parallel when possible
  For ordinary nodes, run EXECUTENODE(x, Γ) and SEMPROP(x, Γ); for SCCs, run EXECUTESCC(x, Γ)
  $S \leftarrow S \cup \mathcal{F}$; $\mathcal{F} \leftarrow \{x \in \mathcal{V}^\dagger \setminus S \mid$ READY(x)$\}$
end while

Algorithm 2 Cyclic Dependency Handling inside a Condensed SCC

Require: Condensed SCC $\mathcal{C}_k$, internal edges $\mathcal{E}_k$, runtime context Γ

$\mathcal{B}_k \leftarrow$ LOOPBACKEDGES($\mathcal{C}_k, \mathcal{E}_k$); $\widehat{\mathcal{E}}_k \leftarrow \mathcal{E}_k \setminus \mathcal{B}_k$
$Q \leftarrow$ INITTRIGGERS($\mathcal{C}_k$); $t \leftarrow 0$
repeat
  APPLYDEFERRED(Q, Γ)
  $\mathcal{L} \leftarrow$ TOPOLOGICALLAYERS(BUILDDAG($\mathcal{C}_k, \widehat{\mathcal{E}}_k, Q$))
  $exit \leftarrow$ FALSE; $Q' \leftarrow \emptyset$
  for layer $L \in \mathcal{L}$ **do**
    for triggered $x \in L$ in parallel when possible **do**
      for activated edge $e = (u, w) \in$ RUN(x, Γ) **do**
        if $w \notin \mathcal{C}_k$ **then**
          SEMPROP(e, Γ); $exit \leftarrow$ TRUE
        else if $e \in \mathcal{B}_k$ **then**
          $Q' \leftarrow Q' \cup \{e\}$
        else
          SEMPROP(e, Γ)
        end if
      end for
    end for
  end for
  $Q \leftarrow Q'$; $t \leftarrow t + 1$
until $exit$ or $Q = \emptyset$ or a loop limit is reached

B Evaluation Details

We follow the evaluation protocols used in the original MatPlotBench, DeepResearchBench, and SRDD benchmark papers and summarize the corresponding metrics below.

B.1 MatPlotBench Metrics

MatPlotBench evaluates code execution and chart quality using EPR, CS, VSR, and OS.

Execution Pass Rate (EPR). EPR $= |\{q \in Q : \text{exec}(c_q) = \text{success}\}|/|Q|$, where c_q denotes the generated code for query q.

Code Quality Score (CS). CS $= \frac{1}{|Q|}\sum_{q \in Q} s_c(q)$, where $s_c(q)$ is the LLM-evaluated code score on a 0–100 scale.

Visualization Success Rate (VSR). VSR $= \frac{1}{|Q_{\text{exec}}|}\sum_{q \in Q_{\text{exec}}} s_v(q)$, where $s_v(q)$ is the vision-based score and Q_{exec} contains executable cases.

Overall Score (OS). OS $= \frac{1}{|Q|}\sum_{q\in Q}(s_c(q) + s_v(q))/2$, where $s_v(q) = 0$ for non-executable cases.

B.2 DeepResearchBench Evaluation Procedure

DeepResearchBench evaluates reports with RACE: Comprehensiveness (COMP), Insight/Depth (DEPTH), Instruction-Following (INST), and Readability (READ).

Dynamic Weight Estimation. The judge LLM estimates task-specific dimension weights w_d, with $\sum_d w_d = 1$.

Criteria Generation. The judge LLM generates criteria and weights for each dimension.

Reference-Based Pairwise Scoring. RACE scores the target and reference reports under the same criteria. It aggregates dimension scores as $S_{\text{int}}(R) = \sum_d s_d \cdot w_d$ and normalizes by $S_{\text{final}}(R_{\text{tgt}}) = S_{\text{int}}(R_{\text{tgt}})/(S_{\text{int}}(R_{\text{tgt}}) + S_{\text{int}}(R_{\text{ref}}))$; 0.5000 indicates parity.

Evaluation Dimensions. COMP, DEPTH, INST, and READ measure coverage, analysis, requirement adherence, and organization.

B.3 SRDD Metrics

SRDD evaluates software projects using Completeness (C), Executability (E), Consistency (CS), and Quality (Q).

Completeness (C). C is the percentage of generated projects without placeholder code.

Executability (E). E is the percentage of generated projects that compile and run successfully.

Consistency (CS). CS is the embedding cosine similarity between the requirement and generated code.

Quality (Q). Q integrates the three metrics as $Q = C \times E \times CS$.

C Supplementary Results and Case Studies

We report workflow specification cost and representative generated artifacts.

Table 5: Size of DevAll workflow specifications relative to reference workflow source code.

Method	Lines			Characters		
	YAML	Source	YAML/Source	YAML	Source	YAML/Source
CoDA	928	10,276	9.0%	34,406	445,046	7.7%
EDR	664	14,286	4.6%	28,910	659,760	4.4%
ChatDev 1.0	1,042	1,916	54.4%	39,253	90,503	43.4%

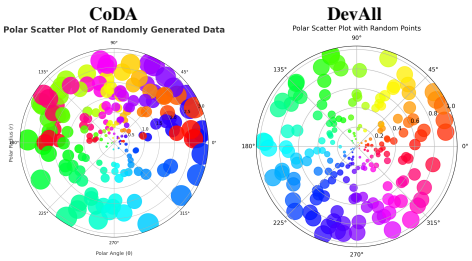

Figure 4: Qualitative comparison of generated visualization artifacts on a representative MatPlotBench case.

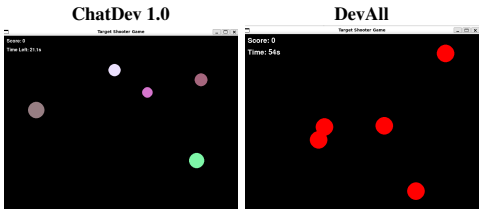

Figure 5: Running screenshots of the target shooter game generated by ChatDev 1.0 and DevAll.

C.1 Workflow Specification Cost

We estimate the authoring effort required to reproduce each specialized workflow. In DevAll, the YAML graph specification records the canvas configuration produced by user drag-and-drop and parameter-setting operations; a smaller YAML specification therefore indicates fewer user-side configuration actions. For each reference system, we count the task-specific orchestration and configuration code needed to define agents, prompts, control flow, state handling, and message passing. Reusable libraries, model APIs, evaluation scripts, data files, environment files, and generated outputs are excluded.

Table 5 reports both line and character counts. YAML/Source is the DevAll specification size divided by the corresponding reference source size; lower ratios indicate fewer canvas-level configuration operations relative to task-specific code implementation.

C.2 Qualitative Case Studies

We show representative outputs for visualization and software generation. As shown in Figures 4 and 5, DevAll produces visible artifacts that are qualitatively comparable to the corresponding specialized baselines.