Hanzo Flow: Visual Workflow Builder for AI Agent Orchestration

David Wei Marcus Chen Zach Kelling
Hanzo AI Research
research@hanzo.ai

February 2026

Abstract

We present **Hanzo Flow**, a visual workflow builder for constructing, debugging, and deploying multi-agent AI systems. While code-based agent frameworks provide flexibility, they impose high barriers to iteration, lack visibility into agent decision-making, and make debugging multi-agent interactions intractable. Hanzo Flow introduces three innovations: (i) a *typed dataflow graph* formalism where nodes represent agent operations (LLM calls, tool invocations, control flow, human-in-the-loop gates) and edges represent typed data channels, enabling static verification of workflow correctness before execution; (ii) a *visual debugger* with execution replay, token-level attribution, and branching exploration that reduces debugging time by 67% compared to log-based debugging; and (iii) an *adaptive execution engine* that supports parallel branch execution, dynamic subgraph expansion, and automatic retry with model fallback, achieving 94% end-to-end success rate on complex multi-step tasks. We evaluate Hanzo Flow on the Agent-Bench suite, a novel visual workflow benchmark (FlowBench-200), and a user study with 48 developers. Results show that visual workflow construction reduces development time by 3.2x compared to code-based frameworks while achieving equivalent or superior task completion rates. Hanzo Flow has been used in production to build 1,200+ workflows processing 8.4 million agent invocations over 10 months.

1 Introduction

The rise of AI agents—autonomous systems that combine LLM reasoning with tool use and memory [18, 20]—has created demand for frameworks that simplify agent construction and orchestration. Current approaches fall into two categories:

1. **Code-first frameworks** (LangChain [11], CrewAI [4], AutoGen [19]): Flexible but opaque. Agent behavior is encoded in Python/-TypeScript code, making it difficult to visualize control flow, debug failures, or modify workflows without deep understanding of the codebase.
2. **No-code builders** (Zapier AI, Make.com): Accessible but limited. Support simple linear chains but lack the expressiveness for conditional branching, parallel execution, loops, and multi-agent coordination required for complex workflows.

Hanzo Flow occupies the middle ground: a visual programming environment with the expressiveness of code-first frameworks and the accessibility of no-code builders.

1.1 Design Principles

1. **Graphs, not chains**: Workflows are directed acyclic graphs (DAGs) with typed edges, supporting arbitrary branching, merging, and parallel execution.
2. **Static verification**: Type checking and constraint validation catch errors before execution.
3. **Observable execution**: Every node execution is logged with full input/output traces, enabling replay and debugging.
4. **Incremental deployment**: Workflows can be tested node-by-node, branch-by-branch, or end-to-end before deployment.

1.2 Contributions

1. A typed dataflow graph formalism for agent workflows with static verification (§2).
2. A visual debugger with execution replay and branching exploration (§3).
3. An adaptive execution engine with parallel branching and model fallback (§4).
4. Evaluation on benchmarks and user study (§6).
5. Production deployment analysis (§7).

2 Typed Dataflow Graphs

2.1 Formal Definition

Definition 1 (Flow Graph)**.** *A Hanzo Flow graph is a tuple $G = (V, E, \Sigma, \tau)$ where:*

- $V = \{v_1, \dots, v_n\}$ *is a set of nodes (operations).*
- $E \subseteq V \times V \times Port$ *is a set of directed edges with port annotations.*
- Σ *is a type system with base types* $\mathcal{B} = \{\texttt{string}, \texttt{number}, \texttt{boolean}, \texttt{json}, \texttt{image}, \texttt{embedding}, \texttt{message[]}\}$ *and constructors* $\texttt{List}[\cdot]$, $\texttt{Optional}[\cdot]$, $\texttt{Union}[\cdot, \cdot]$.
- $\tau : Port \to \Sigma$ *assigns types to input and output ports.*

Definition 2 (Well-Typed Flow)**.** *A flow graph G is well-typed if for every edge $(u, v, (p_u, p_v)) \in E$:*

$$\tau(p_u^{out}) \preceq \tau(p_v^{in}), \tag{1}$$

where $\preceq$ denotes subtype compatibility (e.g., $\texttt{string} \preceq \texttt{Optional[string]}$).

2.2 Node Types

Hanzo Flow provides 14 built-in node types organized into four categories:

Category	Node Type	Ports
AI	LLM Call	3 in, 2 out
	Embedding	1 in, 1 out
	Classifier	1 in, 2 out
	Structured Output	2 in, 1 out
Control	Conditional Branch	1 in, n out
	Parallel Split	1 in, n out
	Merge/Join	n in, 1 out
Data	Transform (code)	n in, m out
	API Call	2 in, 2 out
	Database Query	2 in, 1 out
	File I/O	1 in, 1 out
Human	Approval Gate	1 in, 2 out
	Input Request	0 in, 1 out
	Feedback Loop	2 in, 2 out

Table 1: Built-in node types in Hanzo Flow.

2.3 LLM Call Node

The LLM Call node is the fundamental AI operation. It accepts three inputs:

1. **System prompt** (`string`): The system-level instruction.
2. **Messages** (`message[]`): Conversation history.
3. **Tools** (`json[]`): Available tool definitions.

And produces two outputs:

1. **Response** (`string | json`): The model's text or structured output.
2. **Tool calls** (`json[]`): Any tool invocations requested by the model.

Configuration includes model selection (or "auto" for routing), temperature, max tokens, response format (text, JSON, or schema-constrained), and retry policy.

2.4 Conditional Branch Node

The Conditional Branch node evaluates a predicate on its input and routes data to one of n output ports:

Definition 3 (Branch Semantics)**.** *Given input x and predicates $\{P_1, \dots, P_n\}$ with a default branch* *d:*

$$output(x) = \begin{cases} port_i & \text{if } P_i(x) \text{ is first true predicate,} \\ port_d & \text{if no predicate is true.} \end{cases} \quad (2)$$

Predicates can be:

- **Expression-based**: JavaScript expressions (e.g., `x.score > 0.8`).
- **LLM-based**: Natural language conditions evaluated by a classifier (e.g., "Is this response about code?").
- **Schema-based**: JSON Schema validation (e.g., "Does the output contain a `name` field?").

2.5 Static Verification

Before execution, Hanzo Flow performs four verification passes:

Algorithm 1 Static Flow Verification

```
Require: Flow graph $G = (V, E, \Sigma, \tau)$
1:                                        ▷ Pass 1: Type checking
2: for each edge $(u, v, (p_u, p_v)) \in E$ do
3:     Verify $\tau(p_u^{\text{out}}) \preceq \tau(p_v^{\text{in}})$
4: end for
5:                                        ▷ Pass 2: Connectivity
6: Verify all non-optional input ports have exactly one incoming edge
7: Verify graph has exactly one start node and at least one end node
8:                ▷ Pass 3: Acyclicity (for non-loop subgraphs)
9: Verify topological sort exists for non-feedback edges
10:                                       ▷ Pass 4: Resource bounds
11: Verify no unbounded parallel expansion
12: Estimate max token consumption and cost
13: return verification result with warnings/errors
```

Theorem 1 (Type Safety)**.** *If a flow graph passes static verification, then at runtime, no type mismatch error will occur at any edge, assuming node implementations respect their declared types.*

Proof. By induction on topological order. The start node produces outputs of declared type. Each subsequent node receives inputs of compatible type (by verification), and by the contract of node implementations, produces outputs of declared type. The inductive step preserves type safety at every edge. □

3 Visual Debugger

3.1 Execution Traces

Every execution of a flow graph produces a trace $T = [(v_i, t_i, x_i, y_i, \delta_i)]$ where v_i is the node, t_i is the timestamp, x_i is the input, y_i is the output, and δ_i is the duration. Traces are stored in a structured log and indexed for retrieval.

3.2 Replay Mode

The visual debugger supports full execution replay:

1. **Step-through**: Execute one node at a time, inspecting inputs/outputs at each step.
2. **Breakpoints**: Pause execution at specific nodes or when conditions are met.
3. **Time travel**: Rewind to any previous step and re-execute from that point with modified inputs.
4. **Branch exploration**: At conditional nodes, explore all branches (not just the one taken) to understand alternative paths.

3.3 Token-Level Attribution

For LLM Call nodes, the debugger provides token-level attribution showing which input tokens influenced which output tokens, computed via gradient-based attention analysis:

$$\text{Attr}(x_i, y_j) = \sum_{h=1}^{H} \sum_{l=1}^{L} \alpha_{h,l}(x_i, y_j), \quad (3)$$

where $\alpha_{h,l}$ is the attention weight from head h in layer l. This enables developers to understand why a model produced a particular output.

3.4 Diff View

When modifying a workflow, the debugger provides a diff view comparing execution traces before and after the change:

Algorithm 2 Execution Diff Analysis

```
Require: Traces $T_{\text{old}}$, $T_{\text{new}}$, test inputs $\mathcal{X}$
 1: for each test input $x \in \mathcal{X}$ do
 2:     $y_{\text{old}} \leftarrow \text{Replay}(T_{\text{old}}, x)$
 3:     $y_{\text{new}} \leftarrow \text{Execute}(G_{\text{new}}, x)$
 4:     diff $\leftarrow \text{StructuredDiff}(y_{\text{old}}, y_{\text{new}})$
 5:     Highlight changed nodes and data paths
 6: end for
 7: return aggregate diff summary
```

3.5 Debugging Effectiveness

In a user study with 24 developers debugging intentionally broken workflows (see §6 for details):

Debugging Method	Time (min)	Fix Rate
Log-based (print/grep)	18.4	71%
IDE debugger (breakpoints)	12.7	83%
LangSmith traces	9.2	87%
Hanzo Flow debugger	**6.1**	**96%**

Table 2: Debugging time and fix rate comparison.

4 Adaptive Execution Engine

4.1 Execution Model

The execution engine processes flow graphs using a topological executor with support for parallelism, dynamic expansion, and error recovery.

Algorithm 3 Adaptive Flow Execution

```
Require: Flow graph $G = (V, E, \Sigma, \tau)$, input $x_0$
 1: ready $\leftarrow \{v \in V : \text{in-degree}(v) = 0\}$
 2: results $\leftarrow \{\}$
 3: while ready $\neq \emptyset$ do
 4:     batch $\leftarrow$ independent nodes in ready
 5:     Execute batch in parallel:
 6:     for each $v \in$ batch in parallel do
 7:         $x_v \leftarrow \text{GatherInputs}(v, E, \text{results})$
 8:         $y_v \leftarrow \text{ExecuteNode}(v, x_v)$
 9:         if $y_v$ is error and $v$ has retry policy then
10:             $y_v \leftarrow \text{RetryWithFallback}(v, x_v)$
11:         end if
12:         results[$v$] $\leftarrow y_v$
13:     end for
14:     Update ready with newly satisfiable nodes
15: end while
16: return results[end node]
```

4.2 Parallel Branch Execution

Parallel Split nodes create independent execution branches that run concurrently. The Merge node waits for all (or a configurable quorum of) branches to complete:

Definition 4 (Merge Strategies)**.**

- ***All****:* *Wait for all branches; output is a list of results.*
- ***First****:* *Return the first completed branch; cancel others.*
- ***Quorum(k)****:* *Wait for k of n branches; return the k results.*
- ***Best****:* *Wait for all; return the result with the highest quality score.*

The "Best" strategy is particularly useful for LLM calls: generate multiple responses in parallel with different models or temperatures, then select the best one using a quality evaluator.

4.3 Dynamic Subgraph Expansion

Certain patterns require dynamic workflow modification at runtime:

1. **Map-Reduce**: When an LLM Call returns a list, the engine can dynamically spawn a subgraph instance per list item, process them in parallel, and reduce the results.

2. **Recursive decomposition**: A task decomposition node can generate subtasks, each of which is processed by a copy of a designated subgraph.

3. **Tool-triggered expansion**: When a tool call returns structured data requiring further processing, the engine instantiates a handler subgraph.

Algorithm 4 Map-Reduce Expansion

```
Require: Input list $[x_1, \ldots, x_n]$, map subgraph $G_m$, reduce node $R$
1: ▷ Expand: create $n$ instances of map subgraph
2: for $i = 1, \ldots, n$ in parallel do
3:     $y_i \leftarrow \text{Execute}(G_m, x_i)$
4: end for
5:                         ▷ Reduce: aggregate results
6: $y \leftarrow R([y_1, \ldots, y_n])$
7: return $y$
```

4.4 Retry and Fallback

Each node can be configured with a retry policy:

Strategy	Behavior	Use Case
Fixed retry	Same model, k attempts	Transient errors
Backoff retry	Exp. backoff, same model	Rate limits
Model fallback	Try next model in list	Model outage
Temperature retry	Increase temp each try	Format failures
Prompt retry	Append error to prompt	Content errors

Table 3: Retry strategies available per node.

4.5 Cost and Latency Estimation

Before execution, the engine estimates total cost and latency:

$$\text{Cost}(G) = \sum_{v \in V} \text{Cost}(v) \cdot \mathbb{E}[\text{retries}(v)], \quad (4)$$

$$\text{Latency}(G) = \max_{\text{path } P \in G} \sum_{v \in P} \text{Latency}(v) \cdot \mathbb{E}[\text{retries}(v)], \quad (5)$$

where the latency is determined by the critical path (longest sequential chain) due to parallel execution of independent branches.

5 Workflow Patterns

We identify seven canonical workflow patterns that cover 89% of production workflows:

5.1 Sequential Chain

The simplest pattern: a linear sequence of LLM calls, each refining the previous output.

Example: Research assistant that (1) generates search queries, (2) retrieves documents, (3) synthesizes an answer, (4) formats the output.

5.2 Router Pattern

A classifier node routes input to specialized sub-workflows based on query type.

Example: Customer support that routes to billing, technical, or general sub-agents.

5.3 Parallel Ensemble

Multiple models process the same input in parallel; results are merged by a selector.

Example: Code generation with three models, selecting the one that passes unit tests.

5.4 Iterative Refinement

A feedback loop where output is evaluated and iteratively improved until quality criteria are met.

Example: Essay writing with LLM-as-judge refinement until score exceeds 8/10.

5.5 Map-Reduce

Input is split into parts, each processed independently, then results are aggregated.

Example: Document analysis where each page is summarized independently, then summaries are merged.

5.6 Human-in-the-Loop

Automated processing with human approval gates at critical decision points.

Example: Content moderation pipeline with human review for borderline cases.

5.7 Recursive Decomposition

Complex tasks are decomposed into subtasks, each handled by a sub-workflow, with results aggregated recursively.

Example: Project planning that breaks a goal into tasks, each task into steps, and each step into actions.

6 Evaluation

6.1 AgentBench Results

We evaluate Hanzo Flow on AgentBench [13], a benchmark suite for AI agents:

Framework	OS	DB	Web	Avg.
LangChain	42.3	38.7	51.2	44.1
CrewAI	39.8	41.2	48.9	43.3
AutoGen	45.1	43.8	53.4	47.4
Hanzo Flow	**51.7**	**48.2**	**58.3**	**52.7**

Table 4: AgentBench success rates (%). Hanzo Flow benefits from adaptive execution and model fallback.

6.2 FlowBench-200

We created FlowBench-200, a benchmark of 200 workflow construction tasks spanning seven pattern categories. Participants are given a task description and must build a working workflow. Scoring is based on correctness (automated test cases) and efficiency (number of nodes, execution cost).

Approach	Correct	Time (min)	Nodes
LangChain (code)	72%	34.2	—
LangGraph (code)	78%	28.7	—
Flowise (visual)	61%	19.4	12.3
Hanzo Flow	**89%**	**10.6**	**8.7**

Table 5: FlowBench-200 results. Visual workflow construction with Hanzo Flow is 3.2x faster than code-based approaches.

6.3 User Study

We conducted a controlled user study with 48 developers (24 experienced with agent frameworks, 24 novices) building three workflows of increasing complexity:

Group	Tool	Task 1	Task 2	Task 3
Expert	Code	8.2 min	22.4 min	41.7 min
Expert	Flow	3.1 min	8.3 min	15.2 min
Novice	Code	24.1 min	DNF	DNF
Novice	Flow	5.4 min	14.7 min	28.3 min

Table 6: User study completion times. DNF = did not finish within 60 min.

Key findings:

1. Experts were 2.6x faster with Hanzo Flow than with code.
2. Novices who could not complete tasks in code successfully completed them in Hanzo Flow.
3. Visual debugging was cited by 92% of participants as the most valuable feature.
4. Static type checking caught 78% of errors before execution.

6.4 Satisfaction Survey

Post-study satisfaction (Likert scale 1–5):

Dimension	Code	Hanzo Flow
Ease of use	2.8	4.4
Debugging experience	2.3	4.6
Confidence in correctness	3.1	4.2
Would recommend	3.4	4.7

Table 7: User satisfaction survey results.

7 Production Deployment

7.1 Infrastructure

Hanzo Flow is deployed as part of the Hanzo Platform:

- **Editor**: React application with canvas-based graph editor, running on CDN edge nodes.
- **Execution engine**: Go service on Kubernetes (4 replicas, auto-scaling to 16), processing workflow executions.

- **Trace store**: PostgreSQL + S3 for execution traces (compressed, 90-day retention).
- **LLM access**: Via Hanzo LLM Gateway with per-workflow cost budgets.

7.2 Usage Statistics (10 Months)

Metric	Value
Total workflows created	1,247
Total executions	8,412,893
Avg. nodes per workflow	9.2
Avg. execution time	12.4s
End-to-end success rate	94.1%
Unique users	3,218
LLM calls processed	34,891,247
Total cost (LLM + compute)	$847,231
Avg. cost per execution	$0.10

Table 8: Production statistics (May 2025 – Feb 2026).

7.3 Pattern Distribution

Pattern	Workflows	Success Rate
Sequential chain	31.2%	97.3%
Router	18.4%	95.8%
Parallel ensemble	14.7%	92.1%
Iterative refinement	12.3%	89.4%
Map-reduce	10.1%	93.7%
Human-in-the-loop	8.2%	96.2%
Recursive decomposition	5.1%	84.3%

Table 9: Workflow pattern distribution in production.

7.4 Common Failure Modes

Analysis of failed executions reveals:

Failure Mode	Frequency	Recovery
LLM format error	34%	Prompt retry
API timeout	21%	Backoff retry
Model rate limit	18%	Model fallback
Type mismatch (runtime)	12%	Manual fix
Cost budget exceeded	9%	User notification
Infinite loop detected	6%	Auto-terminate

Table 10: Failure mode analysis of production executions.

8 Related Work

8.1 Agent Frameworks

LangChain [11] provides a code-first framework for LLM application development with chains, agents, and memory. LangGraph [12] extends LangChain with graph-based workflow definition. CrewAI [4] specializes in multi-agent role-based collaboration. AutoGen [19] enables multi-agent conversation-driven workflows. DSPy [9] compiles declarative language model programs. Hanzo Flow complements these frameworks by providing a visual layer with static verification and debugging.

8.2 Visual Programming for AI

ComfyUI [3] provides node-based visual programming for Stable Diffusion workflows. Flowise [6] offers a drag-and-drop UI for LangChain flows. n8n [14] is a general-purpose workflow automation tool with AI nodes. Rivet [15] provides a visual IDE for AI agent prototyping. Hanzo Flow differs in its typed dataflow formalism, static verification, and adaptive execution engine.

8.3 Dataflow Programming

The dataflow programming paradigm has roots in Lucid [17], Lustre [7], and LabVIEW [10]. Modern implementations include Apache Beam [2] for data processing and TensorFlow's computational graphs [1]. Hanzo Flow applies dataflow principles to AI agent orchestration with domain-specific typing and execution semantics.

8.4 Workflow Verification

Formal verification of workflows has been studied in business process management [16] and scientific workflows [5]. Type systems for dataflow have been explored in stream processing [8]. Hanzo Flow introduces a lightweight type system tailored to LLM input/output types with practical static verification.

9 Discussion

9.1 When to Use Code vs. Visual

Hanzo Flow is most effective for:

- Workflows with clear input/output contracts at each step.
- Multi-model or multi-agent systems requiring visibility.
- Rapid prototyping and iteration.
- Teams with mixed technical skill levels.

Code-based approaches remain preferable for:

- Highly dynamic workflows where structure changes per-input.
- Integration with complex existing codebases.
- Performance-critical inner loops.

9.2 Limitations

1. **Expressiveness ceiling**: Some algorithms are awkward to express as dataflow graphs (e.g., recursive algorithms with complex state).
2. **Large graphs**: Workflows with 50+ nodes become visually cluttered; subgraph abstraction partially addresses this.
3. **Custom nodes**: Extending with custom node types requires TypeScript knowledge.
4. **Version control**: Graph-based workflows have less mature diffing/merging than text-based code.

9.3 Future Work

- **AI-assisted construction**: Use LLMs to generate workflow graphs from natural language descriptions.
- **Automated optimization**: Apply graph optimization passes (node fusion, reordering, caching) to reduce cost and latency.
- **Collaborative editing**: Real-time multi-user workflow editing with conflict resolution.
- **Marketplace**: Community-contributed workflow templates and custom nodes.

10 Conclusion

We have presented Hanzo Flow, a visual workflow builder for AI agent orchestration that combines the expressiveness of code-based frameworks with the accessibility of visual programming. The typed dataflow graph formalism enables static verification of workflow correctness, the visual debugger with execution replay reduces debugging time by 67%, and the adaptive execution engine with parallel branching and model fallback achieves 94% end-to-end success rate. User studies confirm 3.2x faster workflow construction, and production deployment with 1,200+ workflows and 8.4M executions validates practical viability. Hanzo Flow is available as part of the Hanzo Platform at `flow.hanzo.ai`.

References

[1] M. Abadi, P. Barham, J. Chen, et al. TensorFlow: A system for large-scale machine learning. In *OSDI*, 2016.

[2] Apache Software Foundation. Apache Beam: An advanced unified programming model. *Apache Documentation*, 2016.

[3] ComfyUI. ComfyUI: A powerful and modular stable diffusion GUI. *GitHub Repository*, 2023.

[4] CrewAI. CrewAI: Framework for orchestrating role-playing autonomous AI agents. *CrewAI Documentation*, 2024.

[5] E. Deelman, G. Singh, M.-H. Su, et al. Pegasus: A framework for mapping complex scientific workflows onto distributed systems. *Scientific Programming*, 13(3):219–237, 2005.

[6] Flowise. Flowise: Drag and drop UI to build LLM flows. *GitHub Repository*, 2023.

[7] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[8] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):1–34, 2014.

[9] O. Khattab, A. Singhvi, P. Maheshwari, et al. DSPy: Compiling declarative language model calls into self-improving pipelines. *arXiv preprint arXiv:2310.03714*, 2023.

[10] National Instruments. LabVIEW: The graphical programming language. *NI Documentation*, 2003.

[11] LangChain. LangChain: Building applications with LLMs through composability. *LangChain Documentation*, 2023.

[12] LangChain. LangGraph: Building stateful, multi-actor applications with LLMs. *LangGraph Documentation*, 2024.

[13] X. Liu, H. Yu, H. Zhang, et al. AgentBench: Evaluating LLMs as agents. In *ICLR*, 2024.

[14] n8n. n8n: Workflow automation tool. *n8n Documentation*, 2024.

[15] Ironclad. Rivet: The open-source visual AI programming environment. *GitHub Repository*, 2023.

[16] W. M. P. van der Aalst. Workflow verification: Finding control-flow errors using Petri-net-based techniques. In *Business Process Management*, 2003.

[17] W. W. Wadge and E. A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[18] L. Wang, C. Ma, X. Feng, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), 2024.

[19] Q. Wu, G. Bansal, J. Zhang, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155*, 2023.

[20] Z. Xi, W. Chen, X. Guo, et al. The rise and potential of large language model based agents: A survey. *arXiv preprint arXiv:2309.07864*, 2023.

[21] S. Yao, J. Zhao, D. Yu, et al. ReAct: Synergizing reasoning and acting in language models. In *ICLR*, 2023.

[22] N. Shinn, F. Cassano, A. Gopinath, et al. Reflexion: Language agents with verbal reinforcement learning. In *NeurIPS*, 2023.

[23] M. Zhuge, W. Liu, B. Peng, et al. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.

[24] S. Hong, X. Zhuge, J. Chen, et al. MetaGPT: Meta programming for multi-agent collaborative framework. In *ICLR*, 2024.

[25] Y. Shen, K. Song, X. Tan, et al. HuggingGPT: Solving AI tasks with ChatGPT and its friends in Hugging Face. In *NeurIPS*, 2023.