Flows: Building Blocks of Reasoning and Collaborating AI

Anonymous ACL submission

Abstract

Recent advances in artificial intelligence (AI) have produced highly capable and controllable systems. This creates unprecedented opportunities for structured reasoning as well as collaboration among multiple AI systems and humans. To fully realize this potential, it is essential to develop a principled way of designing and studying such structured interactions. For this purpose, we introduce the conceptual framework *Flows*. Flows are self-contained building blocks of computation, with an isolated state, communicating through a standardized message-based interface. This modular design allows Flows to be recursively composed into arbitrarily nested interactions, with a substantial reduction of complexity. Crucially, any interaction can be implemented using this framework, including prior work on AI–AI and human–AI interactions, prompt engineering schemes, and tool augmentation. We demonstrate the potential of *Flows* on competitive coding, a challenging task on which even GPT-4 struggles. Our results suggest that structured reasoning and collaboration substantially improve generalization, with AI-only Flows adding +21 and human–AI Flows adding +54 absolute points in terms of solve rate. To support rapid and rigorous research, we introduce the `aiFlows` library embodying *Flows*.

1 Introduction

The success of large language models (LLMs) largely lies in their remarkable emergent ability to adapt to information within their context (i.e., prompt) (Brown et al., 2020; Wei et al., 2022; Kojima et al., 2022). By strategically crafting the context, LLMs can be conditioned to perform complex reasoning (Wei et al., 2022; Nye et al., 2021) and effectively utilize external tools (Schick et al., 2023), significantly enhancing their capabilities. Some of the most exciting recent developments involve defining *control flows*, wherein an LLM controls a set of tools, orchestrated to solve increasingly complex tasks. Examples of such control flows include ReAct (Yao et al., 2023b), AutoGPT (Richards, 2023), or BabyAGI (Nakajima, 2023). However, these represent but a few of the many conceivable control flows, offering only a glimpse into the vast potential of structured LLM interactions. To realize this potential, we need to develop ways for systematically studying such interactions.

Currently, no general yet efficient abstraction exists for effectively modeling structured interactions. Previous work and existing frameworks, such as LangChain (Chase, 2022), Chameleon (Lu et al., 2023), and HuggingGPT (Shen et al., 2023), have converged on modeling agents as entities that use LLMs to select and execute actions towards specific tasks, where the set of possible actions is predefined by the available tools. In this view, tools serve a narrow, well-defined goal and can perform sophisticated tasks (e.g., querying a search engine or executing code). However, their behavior is limited to a single interaction. To highlight the implications of this limitation, consider the following scenario: Alice wants to apply for a job at HappyCorp. If Alice is an agent, she would need to explicitly plan the entire process, including preparing the application, sending it, and evaluating it, which may involve a background check, organizing an interview, and more. Alice would need the knowledge and the "computational" ability to plan every detail. Furthermore, unforeseen events may arise (e.g., the interviewer being on parental leave), requiring Alice to adapt. In reality, most of the complexity is hidden from Alice behind an interface to HappyCorp's hiring process that might itself be composed of sub-processes involving many other *agents* and *tools*. The hiring process, carefully designed by experts, can be reused by many agents, and its sub-processes can be modified or improved with minimal or no impact on the other components. This makes it evident that agents and tools should

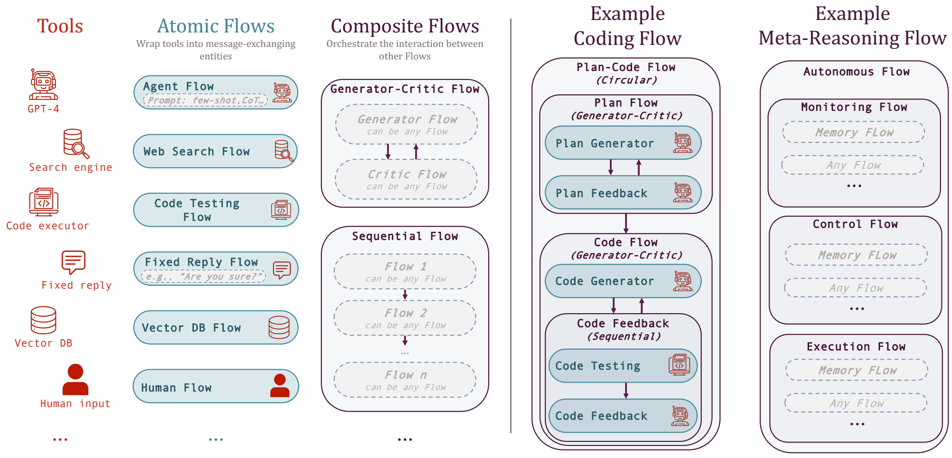

Figure 1: ***Flows* framework exemplified.** The first column depicts examples of tools. The second column depicts Atomic Flows constructed from the example tools. The third column depicts examples of Composite Flows defining structured interaction between Atomic or Composite Flows. The fourth column illustrates a specific Composite competitive coding Flow as those used in the experiments. The fifth column outlines the structure of a hypothetical Flow, defining a meta-reasoning process that could support autonomous behavior.[1]

be able to interact in complex, dynamic or static, ways as parts of nested, modular processes, and the distinction between the two becomes blurred as they both serve as computational units in a complex computational process.

Starting from the observation that everything is a (control) flow defining a potentially complex interaction between many diverse components, we introduce a conceptual framework where Flows are the fundamental building blocks of computation. Flows are independent, self-contained, goal-driven entities able to complete semantically meaningful units of work. To exchange information, Flows communicate via a standardized message-based interface. The framework is depicted in Fig. 1.

The *Flows* abstraction ensures modularity. Alice, a higher-level meta-reasoning Flow that can support autonomous behavior, does not need to know anything beyond how to interface with HappyCorp's hiring Flow. This substantially reduces complexity (Alice is interacting with a deeply nested, compositional structured interaction through a simple interface) and provides flexibility, allowing sub-Flows to be swapped without consequences as long as they have the same interface. Indeed, HappyCorp's pre-filtering Flow can be swapped from a rule-based system to an AI model or even a human Flow without affecting the structure of the overall process. The abstraction also enables reusability and the composition of sub-Flows into new Flows for different tasks. Furthermore, the framework shares key design choices with the Actor model, one of the most prominent models of concurrent computation (cf. Sec. 3). Certainly, once Alice submits her application to HappyCorp, she does not need to wait for the response; she can move to her next goal while the other Flows run concurrently.

We showcase the potential of the proposed framework and library by investigating complex collaborative and structured reasoning patterns on the challenging task of competitive coding, a mind sport involving participants trying to solve problems defined by a natural language description.

Contributions. (i) We propose *Flows*, a conceptual framework providing an abstraction that enables the design and implementation of arbitrarily nested interactions with a substantial reduction of complexity and increase in flexibility in comparison to existing frameworks. *Flows* can represent *any* interaction and provides a common framework for reasoning about interaction patterns, specifying hypotheses, and structuring research, more broadly. (ii) We open-source the `aiFlows` library, which embodies *Flows*, together with the visualization toolkit FlowViz and FlowVerse, a repository of Flows that can be readily used, extended, and composed into novel, more complex Flows. (iii) We leverage *Flows* and the accompanying library to systematically investigate the benefits of complex interactions for solving competitive coding problems and develop AI-only Flows adding +21 and human–AI Flows adding +54 absolute points in terms of solve rate.

[1] For more details on meta-reasoning Flows see Sec. 7

2 Related Work

Existing libraries for modeling structured interactions. LangChain (Chase, 2022) has become the go-to library for creating applications using large language models. However, most recent works involving structured interaction, such as Cameleon (Lu et al., 2023), Camel (Li et al., 2023), HuggingGPT (Shen et al., 2023), AutoGPT (Richards, 2023), and BabyAGI (Nakajima, 2023) come with their own library. We argue that the reason why researchers opt to implement bespoke solutions is the lack of a general yet efficient abstraction for modeling structured interactions that makes it easy to explore novel ideas. *Flows*, with its modular design, provides such an abstraction (cf. Appendix A.3).

Competitive coding (CC). With the advent of transformers, Li et al. (2022) finetuned an LLM on GitHub code repositories and a dataset scraped from Codeforces. Recently, Zelikman et al. (2022) proposed decomposing CC problems into function descriptions and, for each function description, using an LLM to generate the implementation in a modular way. While these methods yield promising results, CC remains a challenging task far from being solved (OpenAI, 2023). This is why it presents itself as an ideal testbed for studying collaborative and structured reasoning interactions.

3 Flows

This section introduces *Flows* as a conceptual framework, describes its benefits, and presents the `aiFlows` library, which embodies the framework.

3.1 *Flows* as a Conceptual Framework

The framework is centered around *Flows* and *messages*. Flows represent the fundamental building block of computation. They are independent, self-contained, goal-driven entities able to complete a semantically meaningful unit of work. To exchange information, Flows communicate via a standardized message-based interface. Messages can be of any type the recipient Flow can process.

We differentiate between two types of Flows: Atomic and Composite.[2] Atomic Flows complete the work directly by leveraging *tools*. Tools can be as simple as a textual sequence specifying a (simple) Flow's fixed response or as complex as a compiler, a search engine, powerful AI systems like LLaMA (Touvron et al., 2023a,b), Stable Diffusion (Rombach et al., 2021), and GPT-4; or even a human. Notably, in the *Flows* framework, AI systems correspond to tools. An Atomic Flow is effectively a minimal wrapper around a tool and achieves two things: (i) it fully specifies the tool (e.g., the most basic Atomic Flow around GPT-4 would specify the prompts and the generation parameters); and (ii) it abstracts the complexity of the internal computation by exposing only a standard message-based interface for exchanging information with other Flows. Examples of Atomic Flows include wrappers around chain-of-thought prompted GPT-4 for solving math reasoning problems, few-shot prompted LLaMA for question answering, an existing chatbot, a search engine API, or an interface with a human.

Composite Flows accomplish more challenging, higher-level goals by leveraging and coordinating other Flows. Crucially, thanks to their local state and standardized interface, Composite Flows can readily invoke Atomic Flows or other Composite Flows as part of compositional, structured interactions of arbitrary complexity. Enabling research on effective patterns of interaction is one of the main goals of our work. General examples of such patterns include (i) factorizing the problem into simpler problems (i.e., divide and conquer); (ii) evaluating (sub-)solutions at inference time (i.e., feedback); and (iii) incorporating external information or a tool. Importantly, Flows can readily invoke other, potentially heavily optimized, specialized Flows to complete specific (sub-)tasks as part of an interaction, leading to complicated behavior. One example of a Composite Flow is ReAct (Yao et al., 2023b). ReAct is a sequential Flow that structures the problem-solving procedure in two steps: a Flow selects the next action out of a predefined set of actions, and another Flow executes it. The two steps are performed until an answer is obtained. Another prominent example, AutoGPT, extends the ReAct Flow with a Memory Flow and an optional Human Feedback Flow. More generally, our framework provides a unified view of prior work, which we make explicit in Appendix A.3.

Importantly, as illustrated in Fig. 1, Composite Flows can script an arbitrarily complex pattern (i) precisely specifying an interaction (e.g., generate code, execute tests, brainstorm potential reasons for failure, etc.); or (ii) defining a high-level, meta-reasoning process in which a Flow could bring about dynamic unconstrained interactions.

[2]The concept of a Flow is sufficient for modeling any interaction. We introduce this distinction as it improves the exposition and simplifies the implementation.

Key properties. The proposed framework is characterized by the following key properties:

- Flows are the compositional building blocks of computation.
- Flows encapsulate a local, isolated state.
- Flows interact only via messages.
- Flows' behaviour depends only on their internal state and the input message.
- Flows can send messages to other Flows and create new Flows.

Connection to the *Actor* model. *Flows* is fundamentally a framework modeling the computation underlying interactions. As such, it shares key design principles with the *Actor* model (Hewitt et al., 1973) — a mathematical model of concurrent computation. Similarly to *Flows*, in the *Actor* model, an Actor is a concurrent computation entity that can communicate with other Actors exclusively through an asynchronous message-passing interface. By encapsulating the state and the computation within individual Actors, the model provides a high-level abstraction for effectively managing and reasoning about complex concurrent and distributed systems, completely avoiding issues associated with shared states, race conditions, and deadlocks. These benefits are similar in nature to those observed in the domain of interactions. The main distinction between the proposed framework and the *Actor* model lies in their respective communication protocols. Concretely, while the *Actor* model prescribes purely asynchronous communication, *Flows* natively supports synchronous communication, which is essential for the implementation of structured reasoning. Interestingly, a similar deviation from the "pure" *Actor* model can be identified in the implementation of Erlang, a concurrent programming language based on it (Armstrong, 2003). Overall, the shared design choices still make *Flows* inherently concurrency-friendly from the practical perspective and are sufficient for important results from the five decades of extensive studies of the *Actor* model, such as the fact that every physically possible computation can be directly implemented using Actors (Hewitt, 2010), to transfer to *Flows*.

3.2 Why *Flows*?

Modularity. *Flows* introduces a higher-level abstraction that isolates the state of individual Flows and specifies message-based communication as the only interface through which Flows can interact. This ensures perfect modularity by design.

Reduction of complexity. The framework ensures the complexity of the computation performed by a Flow is fully abstracted behind the universal message-based interface. This enables an intuitive and simple design of arbitrarily complex interactions from basic building blocks.

Systematicity, flexibility, and reusability. The separation of responsibility allows for modules to be developed and studied systematically in isolation or as part of different interactions. Once the correctness and the benefits of a Flow have been established, it can be readily used in developing novel Flows or as a drop-in replacement for less effective Flows leveraged in completing similar goals.

Concurrency. The proposed framework's design is consistent with the Actor model, one of the most prominent models of concurrent computation. As a consequence, *Flows* can readily support any setting in which Flows run concurrently.

3.3 The `aiFlows` Library

Accompanying *Flows*, we release the `aiFlows` library, which embodies the framework. In addition to the inherent benefits that come with the framework, the library comes with the following add-ons: (i) FlowVerse: a repository (to which anyone can contribute) of Flows that can be readily used, extended, or composed into novel, more complex Flows. *Flows* allows for existing "tools" (as well as "models", "chains", "agents", etc.) to be readily incorporated by wrapping them in an Atomic Flow; (ii) a detailed logging infrastructure enabling transparent debugging, analysis, and research in optimizing (i.e., learning or fine-tuning) Flows; (iii) FlowViz: a visualization toolkit to examine the Flows' execution through an intuitive interface.

4 Competitive Coding Flows

This work investigates the potential of structured interactions for solving competitive coding (CC) problems. In CC, given a natural language description and a few input–output examples, the task is to generate code that will produce the expected output for all of the hidden input–output test cases associated with the problem. Fig. 4 provides examples.

We focus the analysis on three canonical dimensions of interactions: (i) problem decomposition as structured reasoning; (ii) human-AI collabora-

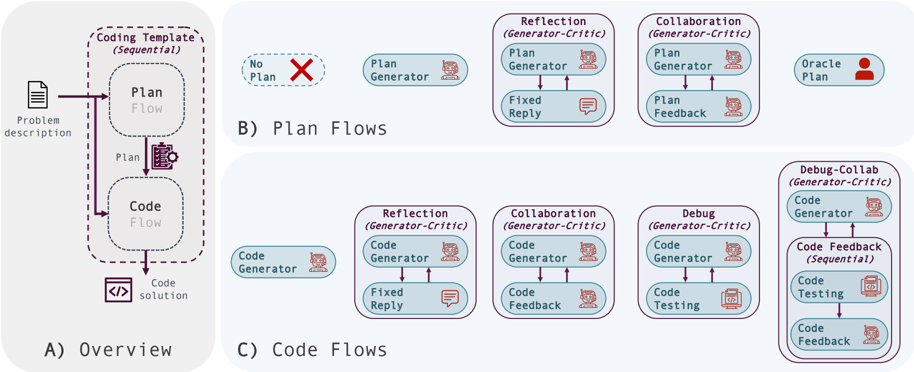

Figure 2: **Competitive coding Flows.** At the highest level, we consider planning as a specific structured reasoning pattern for problem decomposition. In particular, the Plan Flow generates a solution strategy and passes it to the Code Flow, which implements it, as depicted in A). B) and C) depict the different choices of sub-Flows used as Plan and Code Flows in the experiments. Notably, we explore the impact of human-AI collaboration at the plan level and refinement with different types of *feedback*: i) fixed reply encouraging reflection; ii) AI generated feedback; iii) code testing results as feedback; iv) AI generated feedback grounded in code testing results.

tion; and (iii) refinement with various feedback types. By providing a common language for clearly specifying interactions as well as the capability to flexibly compose, exchange, and extend them, the framework makes it possible to study the space of complex interactions in a principled fashion. In the rest of the section, we describe the specific Flows used in the experiments, depicted in Fig. 2.

Problem decomposition. Planning has been an integral intermediate step in recent work (Lu et al., 2023; Shen et al., 2023; Yao et al., 2023b). Similar decomposition is natural in the context of CC as well. In particular, we approach the task in two steps: generating a solution strategy by a Plan Flow and then generating the corresponding code by a Code Flow. This is depicted by panel A in Fig. 2.

Human-AI collaboration. When designing human-AI collaborations, it is essential to take the costs of human interaction into account (Horvitz, 1999; Amershi et al., 2019; Mozannar et al., 2023). By providing immense flexibility, *Flows* can support research in the design of interactions involving humans as computational building blocks in a way that maximizes the utility of the overall computation with a minimal human effort. In the context of CC, we hypothesize that a human can be effectively incorporated at the plan level to provide a short "oracle" plan in natural language. We operationalize this by an (Atomic) Human Flow, illustrated in Panel B of Fig. 2 as the *Oracle Plan* Flow.

Refinement with various feedback types. Iterative refinement is a general problem-solving strategy successfully deployed across various disciplines (Perrakis et al., 1999; Reid and Neubig, 2022; Schick et al., 2022; Saharia et al., 2021). The strategy revolves around the idea that a solution can be gradually improved through a mechanism for analysis, modification, and re-evaluation. The design of this "*feedback*" mechanism is critical for the effectiveness of the problem-solving strategy. The conceptual framework, paired with the accompanying library, provides the infrastructure to support the design, implementation, and principled research of effective refinement strategies and feedback mechanisms. In this work, we consider a canonical iterative refinement setup where a *generator* Flow is tasked with generating the solution, and a *critic* Flow provides feedback on the proposed solution. We consider two feedback types in the context of both the Plan and the Code Flow: (i) Reflection Flow: the feedback consists of a fixed message encouraging the model to reflect on important aspects of the proposed solution; (ii) Collaboration Flow: the feedback is provided by an AI system that "evaluates" the proposed solution. Furthermore, we explore two more code-specific feedback types: (i) Debug Flow: the feedback message corresponds to the results from executing the code and testing it against the examples provided in the problem description; (ii) Debug–Collab Flow: the feedback is provided by an AI system with access to the code testing results, effectively, grounding the feedback and allowing more systematic reasoning about the potential causes of failure.

We refer to Flows using the following convention: *CodeFlowName* when no plan is generated and *PlanFlowName-CodeFlowName* otherwise.

5 Experimental Setup

Data. We scrape publicly available problems from one of the most popular websites hosting CC contests, Codeforces (Mirzayanov, 2023), and LeetCode (LeetCode, 2023), which cover a broad spectrum of problems ranging from easy interview questions to hard CC problems (see Appendix A.1 for more details). The datasets cover problems from 2020-August-21 to 2023-March-26 for CodeForces, and from 2013-October-25 to 2023-April-09 for LeetCode. Importantly, to study the effect of structured interactions (i.e., different Flows) in a principled manner, it is crucial to account for the possibility of *data contamination*, i.e., that some of the test data has been seen during training (Magar and Schwartz, 2022). Containing problems published over an extended period up to a few months ago (at the time of writing), our datasets allow for reliable identification of the training data cutoff date that can help with addressing this issue. Prior code evaluation datasets like APPS (Hendrycks et al., 2021), HumanEval (Chen et al., 2021), and CodeContests (Li et al., 2022) lack problem release dates, and considering the lack of publicly available information about LLMs' training data, can likely lead to confounded evaluation of models' memorization and generalization abilities.

Code testing and solution evaluation. Just like a human participant, the Debug Flow has access only to the input–output example pairs contained in the problem description and, at inference time, uses a local code testing infrastructure to evaluate (intermediate) solution candidates. Crucially, these examples cover only a few simple cases, and generating outputs consistent with them does not imply the code corresponds to a correct solution. A solution is considered correct if it passes all the hidden test cases. To determine correctness, we leverage online evaluators that submit candidate solutions to the websites' online judges, ensuring authoritative results. For many of the Codeforces problems, we also support local evaluation based on a comprehensive set of hidden test cases we managed to scrape. For more details, see Appendix A.2.

Models and Flows. We experiment with the competitive coding Flows described in Sec. 4, and GPT-4 (OpenAI, 2023) as the LLM tool of choice. See Appendix A.4 for the specific prompts. Also, the code to reproduce the experiments in the paper is available in the project's GitHub repository.

Evaluation metrics. The most common evaluation metric for code generation is pass@k, corresponding to the probability that in a set of k sampled candidates, there will be at least one correct solution (Chen et al., 2021). To better align with practical use cases, we focus on pass@1, i.e. the solve rate when averaged across the problem set. We report a point estimate and a 95% confidence interval constructed from 1000 bootstrap resamples.

Compute and cost. All the experiments, including the most complex Flows, can be performed on commodity hardware relatively cheaply. For instance, the costs associated with querying the OpenAI API for generating Table 1 amount to $1000.

6 Experimental Results

We first study the generalization ability of representative Flows and empirically identify GPT-4's knowledge-cutoff date. Next, we perform a focused analysis along the dimensions described in Sec. 4.

6.1 Performance of Coding Flows on Pre- vs. Post-Knowledge-Cutoff-Date Data

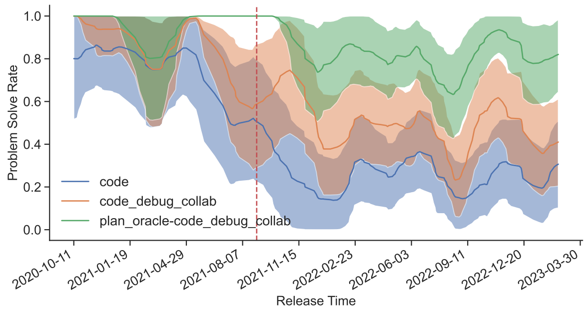

Figure 3: **Temporal analysis.** Performance is averaged over a sliding window of two months. The substantial drop in performance around the reported knowledge cutoff date for GPT-3/4 (the crimson vertical line) reveals limited generalization ability that can be alleviated through structured interactions.

In this experiment, we consider three representative Flows: (i) Code: the simplest Code Generator Flow corresponding to a single GPT-4 API call; (ii) Code_Debug_Collab: the most complex code Flow; (iii) Plan_Oracle-Code_Debug_Collab: the most complex code Flow with human guidance at the plan level. We perform the analysis by running the three Flows on Codeforces problems released from October 2020 to April 2023 and averaging the performance over a sliding window of two months. The results are reported in Fig. 3.

We observe a substantial drop in performance centered around September 2021, consistent with the knowledge cutoff date reported by OpenAI, and denote it by a vertical line on the plot. With Codeforces problems appearing in contexts outside of the contest itself (e.g., editorials), it is reasonable to assume the model has been exposed to older problems more frequently during training. This would explain why the drop spans multiple months, from May 2021 to November 2021, depending on when which data was published and crawled.

Notably, there is a stark difference in the performance of the Code Flow on problems published before and after the knowledge cutoff data, with the solve rate decreasing from around 80% to 23%. While still experiencing a substantial performance drop, the Code_Debug_Collab Flow doubles the solve rate on novel problems to around 45%. Provided with human input at the plan level, the same Flow reaches 85%. Overall, this highlights that GPT-4 performs poorly on novel complex reasoning problems, but structured interactions have the potential to enhance its generalization capabilities. As both GPT-4 (i.e., the Code Flow) and the more complex interactions (Flows) exhibit qualitatively different behavior on novel data, to draw accurate conclusions, it is critical that data contamination is taken into serious consideration when designing experiments and interpreting results.

6.2 Comparing Competitive Coding Flows

Table 1 reports the performance of the systematically chosen set of Flows described in Sec. 4. Rows 6–10 correspond to Flows comprising planning and coding, while rows 1–5 perform the coding directly. In line with the findings of the previous section, we separately consider the performance on problems published before and after the knowledge cutoff date of September 2021.

Problem decomposition. The idea behind planning before implementing the solution is to decouple the high-level reasoning from the code implementation. To analyze the effectiveness of this pattern, we compare the Code and the Plan-Code Flow. Looking at the point estimates, in the pre-cutoff problems, introducing the plan Flow leads to decreased performance (-1.6 for Codeforces and -3.1/2.3/-9.2 for LeetCode easy/medium/hard). However, in the post-cutoff problems, incorporating a plan Flow leads to gains for Codeforces (+8) and LeetCode easy and medium (+2.3 and +3.2). While these trends are consistent, considering the confidence intervals, we see that they are not statistically significant. Crucially, these results do not imply that this specific problem decomposition is not valuable as it creates a lot of potential in designing an effective human-AI collaboration.

Human-AI collaboration. After every contest, the Codeforces community publishes an editorial that, in addition to the code implementation, provides a short natural language description of the solution. To simulate a Flow where a human provides high-level guidance at the core of the reasoning process, we scrape the solution descriptions and pass them as human-generated plans. The results are striking: despite being only a few sentences long, human-provided plans lead to a substantial performance increase (from 26.9% to 74.5% and from 47.5% to 80.8% on novel problems, when the code is generated by Code and Code_Debug_Collab Flows, respectively). First and foremost, these results showcase the opportunities created by *Flows* for designing, implementing, and studying Human-AI collaboration as a key component of structured interactions. Second, specific to the problem of competitive coding, they validate the hypothesis that high-quality plans are important, suggesting that the design of more effective plan Flows is a promising direction to explore in the future. Last but not least, the results highlight the necessity of more systematic research, as patterns seemingly not valuable in one Flow, such as the simple plan-code structured reasoning problem decomposition, can provide immense value as part of another Flow.

Refinement with various feedback types. Among the code Flows, we find that Code_Reflection and Code_Collaboration lead to limited improvements. The two exceptions are Codeforces pre-cutoff (+9.3) for the former and Codeforces post-cutoff (+9.6) for the latter pattern. While close, these results are not statistically significant. On the other hand, the Flows providing grounded feedback, Code_Debug and Code_Debug_Collab, lead to consistent and statistically significant improvements, most notable on the novel Codeforces problems where performance increases from 26.9, without feedback, to 47.5, when the refinement is based on AI-generated feedback grounded in tests. On LeetCode these improvements are smaller in magnitude. We suspect this is a consequence of the examples provided with the problem description being more simplistic than those in Codeforces,

	Codeforces		Leetcode					
	Pre-cutoff	Post-cutoff	Pre-cutoff			Post-cutoff		
			Easy	Medium	Hard	Easy	Medium	Hard
Code	71.8 ±11.0	26.9 ±11.0	97.8 ±3.1	93.4 ±5.4	66.7 ±10.9	76.3 ±8.6	25.1 ±8.9	8.0 ±5.5
Code_Reflection	81.1 ±9.7	26.9 ±10.6	97.8 ±3.1	93.4 ±5.4	67.9 ±10.6	77.4 ±8.1	30.5 ±9.4	11.5 ±6.6
Code_Collaboration	76.6 ±10.5	36.5 ±11.8	97.8 ±3.1	91.1 ±6.0	66.6 ±10.9	73.1 ±8.7	25.1 ±8.7	9.2 ±5.9
Code_Debug	84.5 ±8.6	34.8 ±11.6	97.8 ±3.1	94.5 ±5.0	73.6 ±10.0	84.0 ±7.3	32.8 ±9.6	10.4 ±6.3
Code_Debug_Collab	84.4 ±8.9	47.5 ±12.1	97.8 ±3.1	93.4 ±5.4	72.2 ±10.4	83.8 ±7.4	34.9 ±9.7	9.2 ±6.0
Plan-Code	70.2 ±11.0	34.9 ±11.6	94.7 ±4.5	91.1 ±5.9	57.0 ±11.2	78.6 ±8.3	28.3 ±9.1	4.6 ±4.3
Plan_Reflection-Code	68.5 ±11.6	31.7 ±11.6	95.7 ±4.1	88.9 ±6.6	63.6 ±10.7	77.5 ±8.3	21.8 ±8.5	8.0 ±5.5
Plan_Collaboration-Code	67.0 ±11.5	33.2 ±11.4	96.7 ±3.7	91.1 ±6.1	59.5 ±11.2	74.3 ±8.6	25.2 ±9.0	9.2 ±5.8
Plan_Oracle-Code	82.8 ±9.4	74.5 ±10.7	–	–	–	–	–	–
Plan_Oracle-Code_Debug_Collab	95.4 ±5.2	80.8 ±9.5	–	–	–	–	–	–

Table 1: **Main Results.** Performance of competitive coding Flows on Codeforces and LeetCode.

leading to false positives and, thereby, incorrect grounding, affecting the feedback quality. This could be addressed by generating additional tests with a Test_Case_Generator Flow, a direction we leave for future work to explore. Finally, in the plan Flows, where we consider Reflection and Collaboration (without grounding), we find that refinement does not provide statistically significant benefits.

Overall, our findings offer several important insights: (i) the direct benefit of problem decomposition hinges on the quality of the intermediate steps; (ii) involving humans at the core high-level reasoning process yields major improvements as humans can easily provide high-quality, grounded feedback; (iii) strategic problem decomposition is a powerful strategy for creating opportunities for effective Human–AI collaboration; (iv) the effectiveness from refinement patterns is not universal and depends on the quality of the starting solution and the feedback (e.g., the level of grounding), and the model's ability to incorporate that feedback modulated through the feedback's specificity and the model's capabilities. The analysis paints a substantially more complex picture than what is reported by prior work for simple interactions.

7 Discussion and Conclusion

Simplicity and systematicity. Thanks to its key properties, *Flows*, together with `aiFlows`, provides an infrastructure that greatly simplifies the design and implementation of open-ended interactions, with a capability to flexibly isolate, compose, replace, or modify sub-Flows. The experiments demonstrate that carefully designed interactions can substantially improve generalization. However, our analysis also reveals that the effectiveness of particular interaction patterns is not universal; instead, there are many factors at play. As researchers, we need to clearly specify the patterns we are studying, clearly communicate our hypotheses, and study them both in isolation and as subparts of other interactions across different datasets or/and tasks. Furthermore, it is critical that data contamination is taken into serious consideration when designing experiments and drawing conclusions, and error bars become a standard in the field.

Cost and Performance Optimization. In our experiments, we used "off-the-shelf" LLMs that have not been specifically optimized for collaboration. We posit that we can substantially improve performance and/or compute cost by fine-tuning models to collaborate more effectively, generally or toward specialized roles (e.g., controller or critic). To support research in this direction, `aiFlows` implements detailed logging mechanisms of Flow runs.

Meta-reasoning Flows. Cognitive science research in metacognition and meta-reasoning suggests the existence of meta-level monitoring and control processes underlying cognition (Ackerman and Thompson, 2017). Exploring the development of similar mechanisms in the context of powerful autonomous AI systems and moving beyond a single LLM call serving as a controller (Nakajima, 2023; Richards, 2023) could be a promising area of research. Flows can support such research in higher-level meta-reasoning patterns of interaction.

On the one hand, *Flows* provides a high-level abstraction enabling the design and implementation of interactions of arbitrary complexity. On the other, it offers a common framework for reasoning about interaction patterns, specifying hypotheses, and structuring research. We hope the framework will serve as a solid basis for practical and theoretical innovations, paving the way toward ever more useful AI, similar to the Actor model's role for concurrent and distributed systems.

Limitations

Cost and latency. `aiFlows` is fundamentally a framework modeling the computation that underlies structured reasoning and collaboration, which inherently involves multiple calls. Naturally, this will result in higher latency, which impacts the user experience, and cost in comparison to a single call.

Evaluation limitations.. This work provides the infrastructure to support a systematic study of structured interactions, and demonstrates its utility by providing a thorough evaluation using a single model and a specific subset of interactions on the task of competitive coding. However, as discussed in Sec. 7, many factors determine the effectiveness of structured interactions, and future work should continue exploring the vast space of models and conceivable interactions across the many complex tasks that can be addressed in this setup.

Risk and biases associated with tools. Flows rely on the computation performed by the tools (e.g., LLMs, search engines, etc.) and, therefore, will be exposed to the risks and biases associated with their usage.

Cost and performance optimization. As discussed in Sec. 7, the "off-the-shelf" LLM used in the experiments has not been specifically optimized for effectiveness in structured interactions. Albeit for the better, fine-tuning with `aiFlows` in mind would substantially affect cost and performance.

References

Rakefet Ackerman and Valerie A. Thompson. 2017. Meta-reasoning: Monitoring and control of thinking and reasoning. *Trends in Cognitive Sciences*, 21(8):607–617.

Saleema Amershi, Daniel S. Weld, Mihaela Vorvoreanu, Adam Fourney, Besmira Nushi, Penny Collisson, Jina Suh, Shamsi T. Iqbal, Paul N. Bennett, Kori Inkpen, Jaime Teevan, Ruth Kikin-Gil, and Eric Horvitz. 2019. Guidelines for human-ai interaction. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, page 3. ACM.

Joe Armstrong. 2003. *Making reliable distributed systems in the presence of software errors*. Ph.D. thesis, Royal Institute of Technology, Stockholm, Sweden.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc.

Harrison Chase. 2022. Langchain. https://github.com/hwchase17/langchain.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harrison Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, David W. Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William H. Guss, Alex Nichol, Igor Babuschkin, S. Arun Balaji, Shantanu Jain, Andrew Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew M. Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *ArXiv*, abs/2211.12588.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching large language models to self-debug. *ArXiv*, abs/2304.05128.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with apps. *NeurIPS*.

Carl E. Hewitt. 2010. Actor model of computation: Scalable robust information systems. *arXiv: Programming Languages*.

Carl E. Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A universal modular actor formalism for artificial intelligence. In *International Joint Conference on Artificial Intelligence*.

Eric Horvitz. 1999. Principles of mixed-initiative user interfaces. In *Proceeding of the CHI '99 Conference on Human Factors in Computing Systems: The CHI is the Limit, Pittsburgh, PA, USA, May 15-20, 1999*, pages 159–166. ACM.

Geunwoo Kim, Pierre Baldi, and Stephen McAleer. 2023. Language models can solve computer tasks. *ArXiv*, abs/2303.17491.

Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. In *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213. Curran Associates, Inc.

LeetCode. 2023. Leetcode.com.

Guohao Li, Hasan Abed Al Kader Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. 2023. Camel: Communicative agents for" mind" exploration of large scale language model society. *arXiv preprint arXiv:2303.17760*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Pan Lu, Baolin Peng, Hao Cheng, Michel Galley, Kai-Wei Chang, Ying Nian Wu, Song-Chun Zhu, and Jianfeng Gao. 2023. Chameleon: Plug-and-play compositional reasoning with large language models. *ArXiv*, abs/2304.09842.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. *arXiv preprint arXiv:2303.17651*.

Inbal Magar and Roy Schwartz. 2022. Data contamination: From memorization to exploitation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pages 157–165. Association for Computational Linguistics.

Mike Mirzayanov. 2023. Codeforces.com.

Hussein Mozannar, Gagan Bansal, Adam Fourney, and Eric Horvitz. 2023. When to show a suggestion? integrating human feedback in ai-assisted programming. *CoRR*, abs/2306.04930.

Yohei Nakajima. 2023. Babyagi. `https://github.com/yoheinakajima/babyagi`.

Maxwell I. Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, Charles Sutton, and Augustus Odena. 2021. Show your work: Scratchpads for intermediate computation with language models. *CoRR*, abs/2112.00114.

OpenAI. 2023. Gpt-4 technical report. *ArXiv*, abs/2303.08774.

Debjit Paul, Mete Ismayilzada, Maxime Peyrard, Beatriz Borges, Antoine Bosselut, Robert West, and Boi Faltings. 2023. Refiner: Reasoning feedback on intermediate representations. *arXiv preprint arXiv:2304.01904*.

Anastassis Perrakis, Richard J. Morris, and Victor S. Lamzin. 1999. Automated protein model building combined with iterative structure refinement. *Nature Structural Biology*, 6:458–463.

Machel Reid and Graham Neubig. 2022. Learning to model editing processes. In *Conference on Empirical Methods in Natural Language Processing*.

Toran Bruce Richards. 2023. Autogpt. `https://github.com/Significant-Gravitas/Auto-GPT`.

Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2021. High-resolution image synthesis with latent diffusion models. *CoRR*, abs/2112.10752.

Chitwan Saharia, Jonathan Ho, William Chan, Tim Salimans, David J. Fleet, and Mohammad Norouzi. 2021. Image super-resolution via iterative refinement. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45:4713–4726.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. *ArXiv*, abs/2302.04761.

Timo Schick, Jane Dwivedi-Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. 2022. Peer: A collaborative language model. *ArXiv*, abs/2208.11663.

Yongliang Shen, Kaitao Song, Xu Tan, Dong Sheng Li, Weiming Lu, and Yue Ting Zhuang. 2023. Hugginggpt: Solving ai tasks with chatgpt and its friends in huggingface. *ArXiv*, abs/2303.17580.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aur'elien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023a. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971.

Hugo Touvron, Louis Martin, Kevin R. Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Daniel M. Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin

Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony S. Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel M. Kloumann, A. V. Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, R. Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023b. Llama 2: Open foundation and fine-tuned chat models. *arXiv*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. 2023. Generating sequences by learning to self-correct. In *The Eleventh International Conference on Learning Representations*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. 2023a. Tree of thoughts: Deliberate problem solving with large language models. *ArXiv*, abs/2305.10601.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023b. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Ori Yoran, Tomer Wolfson, Ben Bogin, Uri Katz, Daniel Deutch, and Jonathan Berant. 2023. Answering questions by meta-reasoning over multiple chains of thought. *ArXiv*, abs/2304.13007.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. 2022. Parsel: A (de-)compositional framework for algorithmic reasoning with language models.

Zhuosheng Zhang, Aston Zhang, Mu Li, Hai Zhao, George Karypis, and Alexander J. Smola. 2023. Multimodal chain-of-thought reasoning in language models. *ArXiv*, abs/2302.00923.

A Appendix

A.1 Data

Example Codeforces and LeetCode problems are provided in Fig. 4.

In the first experiment, the temporal analysis, we use 239 Codeforces problems ranging from October 2020 to April 2023. In the second experiment, we have 136 problems for Codeforces (some problems are dropped in order to keep the pre-cutoff and post-cutoff buckets equal to 68) and 558 problems for LeetCode (93 for each of the six buckets). Additionally, to support research in the area, we set up an AI competitive coding challenge based on a dataset of Codeforces problems of various difficulties published after the knowledge cutoff date. More details about the CC competition are available in Appendix A.5.

A.2 Code Testing and Solution Evaluation

The solution evaluation requires a set of input–output pairs, hidden from the user, that comprehensively test the behavior of the program. To compute the final results, we have implemented an online evaluation infrastructure that submits the candidate solutions to the websites' online judges and automatically scrapes the judgment. This mechanism ensures authoritative results.

For many of the Codeforces problems, we managed to scrape (sometimes a subset) of the hidden tests, allowing us to use a faster, local infrastructure for evaluating candidate solutions. On the other hand, LeetCode does not expose any of the hidden tests publicly.

For code testing at inference time, just like a human would, we rely on tests constructed from the (public) input–output example pairs contained in the problem description.

A.3 Concurrent and Previous Works as Specific Instances of Flows

The introduction of LLMs such as BARD, GPT-3, ChatGPT, and its latest version, GPT-4, has led to a breakthrough in AI. This has enabled many exciting developments like CoT, HuggingGPT, AutoGPT, AgentGPT, and BabyAGI. In this section, we demonstrate how *Flows* provides a unified view encompassing concurrent and previous work as specific Flow instances. The details are provided in Figure 5 and Table. 2.

1. **Few shot Prompting** (FS) (Brown et al., 2020) consists in providing a few input-output examples within the prompt, acting as demonstrations to enable the LLM to perform a specific task. This technique relies on the LLM's emergent in-context learning ability to extrapolate from these limited examples and infer how to solve the task in general.

2. **Chain of Thoughts** (CoT) (Wei et al., 2022) is a prompting method (atomic Flow) that allows LLMs to generate a series of intermediate natural language reasoning steps that lead to the final output.

3. **Tree of Thoughts** (ToT) (Yao et al., 2023a) is a framework that enables (*orchestration*) exploration over coherent units of text (thoughts) that serve as intermediate steps toward problem-solving. ToT allows LLMs to perform deliberate decision-making by considering multiple different reasoning paths and self-evaluating choices to decide the next course of action, as well as looking ahead or backtracking when necessary to make global choices.

4. **Program of Thoughts** (PoT) (Chen et al., 2022) is a prompting method that allows language models (mainly Codex) to express the reasoning process as a program. The computation is relegated to an external program, which executes the generated programs to derive the answer.

5. **Mutimodal CoT** (M-CoT) (Zhang et al., 2023) is a method that incorporates language (text) and vision (images) modalities into a two-stage framework that separates rationale generation and answer inference. To facilitate the interaction between modalities in M-CoT, smaller language models (LMs) are fine-tuned by fusing multimodal features.

6. **ToolFormer** (Schick et al., 2023) is a model that is trained to decide which APIs to call, when to call them, what arguments to pass, and how to incorporate the results into future tokens prediction.

7. **ReAct** (Yao et al., 2023b) is a framework that uses LLMs to generate reasoning traces and task-specific actions sequentially. The framework allows for greater synergy between the two: reasoning traces help the model induce, track, and update action plans and han-

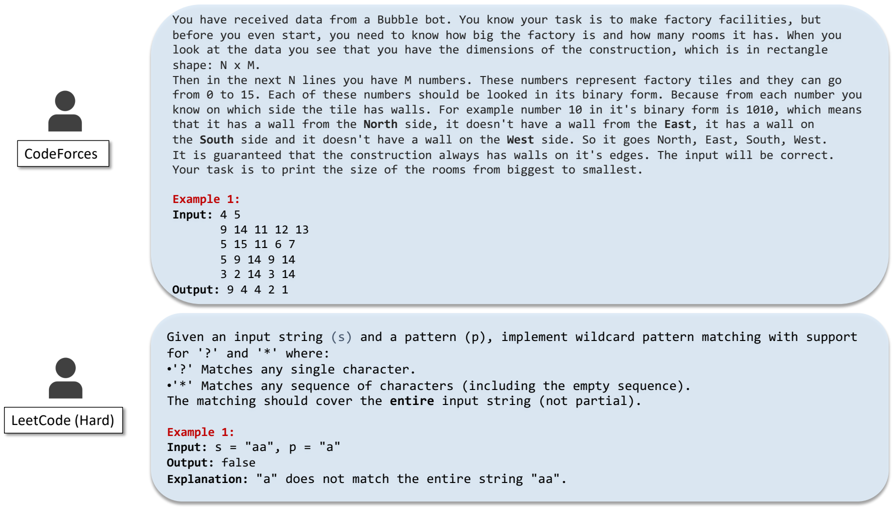

Figure 4: **Examples of competitive coding problems from Codeforces and LeetCode.**

dle exceptions, while actions allow it to interface with external sources, such as knowledge bases or environments, to gather additional information.

8. **Parsel** (Zelikman et al., 2022) is a framework that enables the automatic implementation and validation of complex algorithms with code LLMs. The framework first synthesizes an intermediate representation based on the Parsel language and can then apply a variety of post-processing tools. Code is generated in a next step.

9. **REFINER** (Paul et al., 2023) is a framework for LMs to explicitly generate intermediate reasoning steps while interacting with a critic model that provides automated feedback on the reasoning.

10. **Self-Refine** (Madaan et al., 2023) is a framework for LLMs to generate coherent outputs. The main idea is that an LLM will initially generate an output while the same LLM provides feedback for its output and uses it to refine itself iteratively.

11. **Recursively Criticize and Improve** (RCI) (Kim et al., 2023) showed that a pre-trained large language model (LLM) agent could execute computer tasks guided by natural language using a simple prompting scheme where the agent Recursively Criticizes and Improves its output (RCI). Unlike Self-refine, this method uses two separate LLMs (ChatGPT), one for performing the task and another for criticizing.

12. **Self-Correct** (Welleck et al., 2023) is a framework that decouples a flawed base generator (an LLM) from a separate corrector that learns to iteratively correct imperfect generations. The imperfect base generator can be an off-the-self LLM or a supervised model, and the corrector model is trained.

13. **Self-Debug** (Chen et al., 2023) is a framework that relies on external tools (SQL application or Python interpreter) to help large language models revise and debug SQL commands or Python code with bugs.

14. **Reflexion** (Shinn et al., 2023) is a framework that provides a free-form reflection on whether a step was executed by LLM correctly or not and potential improvements. Unlike self-refine and self-debug, Reflexion builds a persisting memory of self-reflective experiences, which enables an agent to identify its own errors and self-suggest lessons to learn from its mistakes over time.

15. **Meta-Reasoner** (Yoran et al., 2023) is an approach which prompts large language models to meta-reason over multiple chains of thought rather than aggregating their answers. This approach included two steps: (i) ask LLM to generate multiple reasoning chains, (ii) ask another LLM (meta-reasoner) to reason over the multiple reasoning chains to arrive at the correct answer.

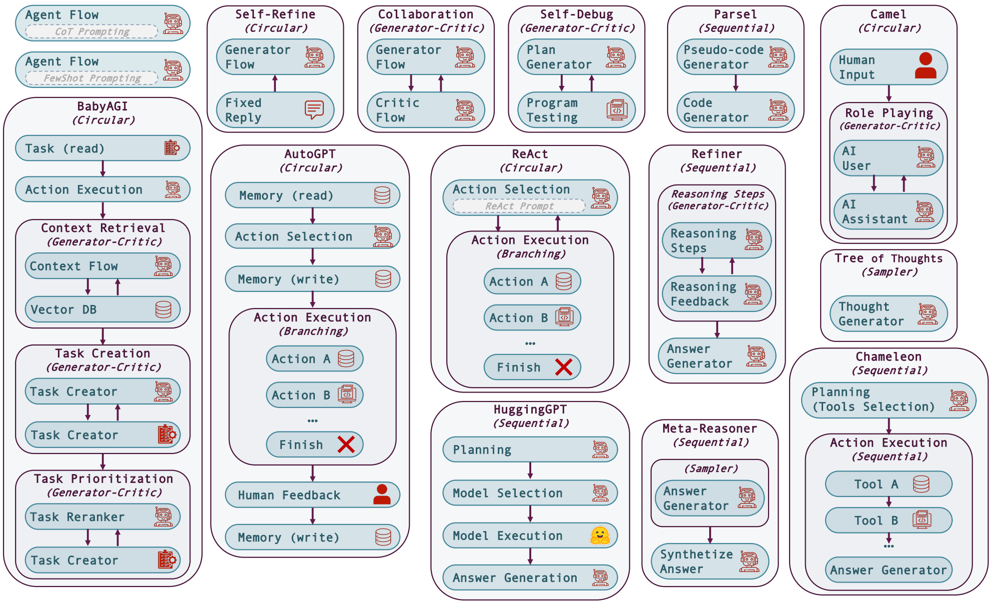

Figure 5: **Previous works are specific Flows.** We depict a selected subset of previous works incorporating structured reasoning and/or interactions between AI agents, tools, and humans, through the lens of the Flows framework. This demonstrates that Flows is a powerful language for describing, conceptualizing, and disseminating structured interaction patterns.

16. **HuggingGPT** (Shen et al., 2023) is a framework that leverages LLMs (e.g., ChatGPT) to connect various AI models in machine learning communities (e.g., Hugging Face) to solve numerous sophisticated AI tasks in different modalities (such as language, vision, speech) and domains.

17. **Camel** (Li et al., 2023) is a communicative agent framework involving inception prompting to guide chat agents toward task completion while maintaining consistency with human intentions.

18. **Chameleon** (Lu et al., 2023) is a plug-and-play compositional reasoning framework that augments external tools with LLMs in a plug-and-play manner. The core idea is that an LLM-based planner assembles a sequence of tools to execute to generate the final response. The assumption is that this will be less error-prone, easily expandable to new modules, and user-friendly.

19. **AutoGPT** (Richards, 2023) is an experimental open-source application that leverages the capabilities of large language models (LLMs) and Chatbots such as OpenAI's GPT-4 and Chat-GPT to create fully autonomous and customizable AI agents. It has internet access, long-term and short-term memory management.

20. **BabyAGI** (Nakajima, 2023) is an intelligent agent capable of generating and attempting to execute tasks based on a given objective. BabyAGI operates based on three LLM flows: Task creation flow, Task prioritization flow, and Execution flow.

A.4 Prompting

We provide the prompts used to obtain the results in Section 6. Our evaluation is made possible thanks to the modular and compositional nature of *Flows*. Some of the experimental setups are deeply nested, and in cases where Flows build on each other, we avoid repetition. Note that the project's GitHub repository provides the code and data to reproduce all of the experiments in the paper.

Direct prompting for a solution is shown in Listing 1. To add reflection, we use a Generator-Critic Flow to combine the code generation with a fixed reply, as shown in Listing 2. In the collaboration setting, we use Listing 3 as the generator and Listing 4 as the critic.

Debugging is incorporated via a testing Flow that adds formatting to the output of a code executor. The formatting templates are shown in Listing 6. To respond to the debug output, we rely on an

Flows	Flow Type	Interactions				Reasoning Patterns		Feedback	Learning
		Self	Multi-Ag.	Human	Tools	Struct.	Plan		
FS (Brown et al., 2020)	Atomic	✗	✗	✗	✗	✗	✗	✗	✗
CoT (Wei et al., 2022)	Atomic	✗	✗	✗	✗	✓	✗	✗	✗
ToT (Yao et al., 2023a)	Circular	✓	✗	✗	✓	✓	✗	✗	✗
PoT (Chen et al., 2022)	Seq.	✗	✗	✗	✓	✓	✗	✗	✗
M-CoT (Zhang et al., 2023)	Seq.	✗	✗	✗	✗	✓	✗	✗	✓
ToolFormer (Wei et al., 2022)	Seq.	✗	✗	✗	✓	✓	✗	✗	✓
ReAct (Yao et al., 2023b)	Circular	✗	✗	✗	✓	✓	✗	✗	✗
Parsel (Zelikman et al., 2022)	Seq.	✗	✓	✗	✓	✓	✓	✗	✗
REFINER (Paul et al., 2023)	Gen-Crit	✗	✓	✓	✗	✓	✗	✓	✓
Self-Refine (Madaan et al., 2023)	Gen-Crit	✓	✗	✗	✗	✓	✗	✓	✗
RCI (Kim et al., 2023)	Gen-Crit	✓	✗	✗	✓	✓	✗	✓	✗
Self-Correct (Welleck et al., 2023)	Gen-Crit	✓	✗	✗	✓	✓	✗	✓	✗
Self-Debug (Chen et al., 2023)	Gen-Crit	✓	✗	✗	✓	✓	✗	✓	✗
Reflexion (Shinn et al., 2023)	Gen-Crit	✓	✗	✗	✓	✗	✗	✓	✗
Meta-Reasoner (Yoran et al., 2023)	Seq.	✓	✓	✗	✗	✓	✗	✗	✗
HuggingGPT (Shen et al., 2023)	Seq.	✗	✓	✗	✓	✓	✓	✗	✗
Camel (Li et al., 2023)	Circular	✗	✓	✓	✗	✓	✗	✓	✗
Chameleon (Lu et al., 2023)	Seq.	✗	✓	✗	✓	✓	✓	✗	✗
AutoGPT (Richards, 2023)	Circular	✓	✓	✗	✓	✓	✓	✓	✗
BabyAGI (Nakajima, 2023)	Circular	✗	✓	✗	✓	✓	✓	✗	✗

Table 2: **Previous work.** We compare previous work across relevant dimensions.

adjusted coding Flow 5. Adding collaboration in the debugging setting is done by introducing a critic that provides feedback grounded in the test results. This Flow is detailed in Listing 3.

The scenarios explained above also support the addition of a planning Flow. An example of plan generation is shown in Listing 8.

Listing 1: Prompts for Code Flow (Codeforces)

```
"prompt templates":
  "system_message": |-
    Your goal is to provide
       executable Python code
       that solves a competitive
       programming problem. The
       code should correctly
       handle all corner cases in
        order to pass the hidden
       test cases, which are used
        to evaluate the
       correctness of the
       solution.

    The user will specify the
       problem by providing you
       with:
      - the problem statement
      - input description
      - output description
      - example test cases
      - (optional) explanation of
          the test cases

    The user will provide you
       with a task and an output
       format that you will
       strictly follow.
  "query_message": |-
    # Problem statement
    {{problem_description}}

    # Input description
    {{input_description}}

    # Output description
    {{output_description}}

    {{io_examples_and_explanation
       }}


    The input should be read from
        the standard input and
       the output should be
       passed to the standard
       output.
    Return Python code that
       solves the problem. Reply
       in the following format:
    ```python
    {{code_placeholder}}
    ```
  "human_message": |-
    {{query}}
```

Listing 2: Prompts for Fixed-Reply Flow

```
"prompt templates":
  "fixed_reply": |-
    Consider the problem
       statement and the last
       proposed solution. Are you
        sure that the solution is
        provided in the requested
        format, and crucially,
       solves the problem?
    If that is not the case,
       provide the corrected
       version of the code in the
        following format:
    ```python
    {{python_code}}
    ```
    otherwise, reply:
    "Final answer."
```

Listing 3: Prompts for Code-Collab Flow (Codeforces)

```
"prompt templates":
  "system_message": |-
    Your goal is to provide
       executable Python code
       that solves a competitive
       programming problem. The
       code should correctly
       handle all corner cases in
        order to pass the hidden
       test cases, which are used
        to evaluate the
       correctness of the
       solution.

    The user will specify the
       problem by providing you
       with:
      - the problem statement
      - input description
      - output description
      - example test cases
      - (optional) explanation of
          the test cases

    The user will provide you
       with a task and an output
       format that you will
       strictly follow.
  "query_message": |-
    # Problem statement
    {{problem_description}}

    # Input description
    {{input_description}}

    # Output description
    {{output_description}}

    {{io_examples_and_explanation
       }}


    The input should be read from
        the standard input and
       the output should be
       passed to the standard
       output.
    Return Python code that
       solves the problem. Reply
       in the following format:
    ```python
    {{code_placeholder}}
    ```
  "human_message": |-
    # Feedback on the last
       proposed solution
    {{code_feedback}}

    Consider the original problem
        statement, the last
       proposed solution and the
       provided feedback. Does
       the solution need to be
       updated? If so, provide
       the corrected version of
       the code in the following
       format:
    ```python
    {{code_placeholder}}
    ```
    otherwise, reply:
    "Final answer."
```

Listing 4: Prompts for Code-Collab-Critic Flow (Codeforces)

```
"prompt templates":
  "system_message": |-
    Your goal is to identify
       potential issues with a
       competitive programming
       solution attempt.
```

```
    The user will specify the problem by providing you with:
      - the problem statement
      - input description
      - output description
      - example test cases
      - (optional) explanation of the test cases
      - a Python solution attempt

    Crucially, your goal is to correctly identify potential issues with the solution attempt, and not to provide the code implementation yourself.
    The user will provide you with a task and an output format that you will strictly follow.
  "query_message": |-
    # Problem statement
    {{problem_description}}

    # Input description
    {{input_description}}

    # Output description
    {{output_description}}

    {{io_examples_and_explanation}}

    # Python solution attempt:
    ```python
    {{code}}
    ```


    Consider the problem statement and the solution attempt. Are there any issues with the proposed solution or it is correct? Explain your reasoning very concisely, and do not provide code.
  "human_message": |-
    {{query}}
```

Listing 5: Prompts for Code-Debug Flow (Codeforces)

```
"prompt templates":
  "system_message": |-
    Your goal is to provide executable Python code that solves a competitive programming problem. The code should correctly handle all corner cases in order to pass the hidden test cases, which are used to evaluate the correctness of the solution.

    The user will specify the problem by providing you with:
      - the problem statement
      - input description
      - output description
      - example test cases
      - (optional) explanation of the test cases

    The user will provide you with a task and an output format that you will strictly follow.
  "query_message": |-
    # Problem statement
    {{problem_description}}

    # Input description
    {{input_description}}

    # Output description
    {{output_description}}

    {{io_examples_and_explanation}}


    The input should be read from the standard input and the output should be passed to the standard output.
    Return Python code that solves the problem. Reply in the following format:
```

```
      ```python
      {{code_placeholder}}
      ```
    "human_message": |-
      {{testing_results_summary}}

      Consider the problem statement, the last proposed solution, and its issue. Provide a corrected version of the code that solves the original problem and resolves the issue, without any explanation, in the following format:
      ```python
      {{code_placeholder}}
      ```
```

Listing 6: Formatting templates for Code-Testing Flow (Codeforces)

```
"formatting templates":
  "no error template": |-
    ${.issue_title}
    All of the executed tests passed.
  "all tests header": |-
    ${.issue_title}
    The Python code does not solve the problem in the problem description due to logical errors. It fails on the following tests.
  "compilation error template": |-
    ${.issue_title}
    The execution resulted in a compilation error.
    ## Compilation error message:
    {{error_message}}
  "timeout error template": |-
    ${.issue_title}
    The execution timed out, the solution is not efficient enough.
  "runtime error template": |-
    ${.issue_title}
    The execution resulted in a runtime error on the following test.
    ## [Failed test] Input
    ```
    {{test_input}}
    ```
    ## [Failed test] Runtime error message
    {{error_message}}
  "single test error": |-
    ${.issue_title}
    The Python code does not solve the problem in the problem description due to logical errors. It fails the following test:
    ## [Failed test] Input
    ```
    {{test_input}}
    ```
    ## [Failed test] Expected output
    ```
    {{expected_output}}
    ```
    ## [Failed test] Generated output
    ```
    {{generated_output}}
    ```
  "test error": |-
    ## [Failed test {{idx}}]
    ### [Failed test {{idx}}] Input
    ```
    {{test_input}}
    ```
    ### [Failed test {{idx}}] Expected output
    ```
    {{expected_output}}
    ```
    ### [Failed test {{idx}}] Generated output
    ```
    {{generated_output}}
    ```
```

Listing 7: Prompts for Code-Debug-Collab Flow (Codeforces)

```
"prompt templates":
  "system_message": |-
    Your goal is to identify the issues with an incorrect
```

```
            competitive programming
            solution attempt.

        The user will specify the
            problem by providing you
            with:
          - the problem statement
          - input description
          - output description
          - example test cases
          - (optional) explanation of
              the test cases
          - an incorrect Python
              solution attempt and a
              description of its issue

        Crucially, your goal is to
            consider all aspects of
            the problem and pinpoint
            the issues with the
            solution attempt, and not
            to provide the code
            implementation yourself.
        Some aspects to consider: Is
            the input correctly parsed
            ? Is the output correctly
            formatted? Are the corner
            cases correctly handled?
            Is there a logical mistake
             with the algorithm itself
            ?
        Use the code execution
            results provided in the
            issue description to guide
             your reasoning/debugging.
    "query_message": |-
        # Problem statement
        {{problem_description}}

        # Input description
        {{input_description}}

        # Output description
        {{output_description}}

        {{io_examples_and_explanation
            }}

        # Solution attempt to be
            fixed
        ```python
        {{code}}
        ```

        {{testing_results_summary}}

        Consider the problem
            statement, the solution
            attempt and the issue. Why
             is the solution attempt
            incorrect? How should it
            be fixed? Explain your
            reasoning very concisely,
            and do not provide code.
    "human_message": |-
        {{query}}
```

Listing 8: Prompts for Plan Flow (Codeforces)

```
"prompt templates":
    "system_message": |-
        Your goal is to provide a
            high-level conceptual
            solution that, if
            implemented, will solve a
            given competitive
            programming problem.

        The user will specify the
            problem by providing you
            with:
          - the problem statement
          - input description
          - output description
          - example test cases
          - (optional) explanation of
              the test cases

        The proposed algorithm should
             be computationally
            efficient, logically
            correct and handle all
            corner cases.

        The user will provide you
            with a task and an output
            format that you will
            strictly follow.
    "query_message": |-
        # Problem statement
        {{problem_description}}

        # Input description
        {{input_description}}
```

```
        # Output description
        {{output_description}}

        {{io_examples_and_explanation
           }}


        Return a high-level
           conceptual solution that
           would solve the problem.
           Be very concise, and do
           not provide code.
        Reply in the following format
           :
        # Conceptual solution
        {{plan_placeholder}}
      "human_message": |-
        {{query}}
```

A.5 The CC-Flows-competition: a new form of competitive coding

Solving competitive coding challenges is an eminently hard problem. The solve rate of only 27% by directly attempting the problem and 47% by the best-performing code Flow, paired with a reliable automatic evaluation metric, make competitive programming an ideal benchmark for AI systems. Motivated by this, we propose a competition where instead of people, proposed Flows solve competitive programming problems.

The competition will leverage the comprehensive dataset of publicly available Codeforces problems and the open-source infrastructure for inference and testing used in the experiments, available at anonymous . The competition will only include problems published after the knowledge-cutoff date of GPT-4. Furthermore, not to overload the Codeforces online evaluation infrastructure, we further filter this dataset to problems for which public and private tests are available, and the output format is compatible with our local code testing infrastructure. Codeforces ranks the difficulty of each problem from 800 to 2100. At the time of publishing, we have the following number of problems per difficulty (total of 416):

- difficulty 800: 149
- difficulty 900 to 1500 (inclusive): 185
- difficulty 1600 to 220 (inclusive): 82

We will curate a leaderboard of best-performing Flows that will be publicly available on FlowVerse and provide the predictions that reproduce the reported scores using the provided infrastructure.