Flows: Building Blocks of Reasoning and Collaborating AI

Martin Josifoski,[*◇] **Lars Klein,**[*◇] **Maxime Peyrard,**[◇] **Yifei Li,**[**◇] **Saibo Geng,**[**◇]
Julian Paul Schnitzler,[◇] **Yuxing Yao,**[◇] **Jiheng Wei,**[♣] **Debjit Paul,**[◇] **Robert West**[◇]
[◇]EPFL [♣]PSL University
{martin.josifoski, lars.klein, maxime.peyrard, robert.west}@epfl.ch

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 of *Flows*: a systematic approach to modeling complex interactions. 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 the task of 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. The library comes with a repository of Flows that can be easily used, extended, and composed into novel, more complex Flows.

The `aiFlows` library is available at `https://github.com/epfl-dlab/aiflows`. Data and Flows for reproducing our experiments are available at `https://github.com/epfl-dlab/cc_flows`.

1 Introduction

Scaling up language models has brought about a revolutionary transformation in the field of artificial intelligence (AI) [6, 12]. In particular, the success of large language models (LLMs) lies in their remarkable emergent ability to adapt to information within their context (i.e., prompt) [6, 21, 47]. By strategically crafting the context, LLMs can be conditioned to perform complex reasoning [31, 47] and effectively utilize external tools [40], 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[50], AutoGPT [36], or BabyAGI [30]. However, these examples represent but a few of the many conceivable control flows, offering only a glimpse into the vast potential of structured LLM interactions. To fully realize this potential, it is essential to develop principled ways of designing and studying such interactions.

[*,**] Equal contribution

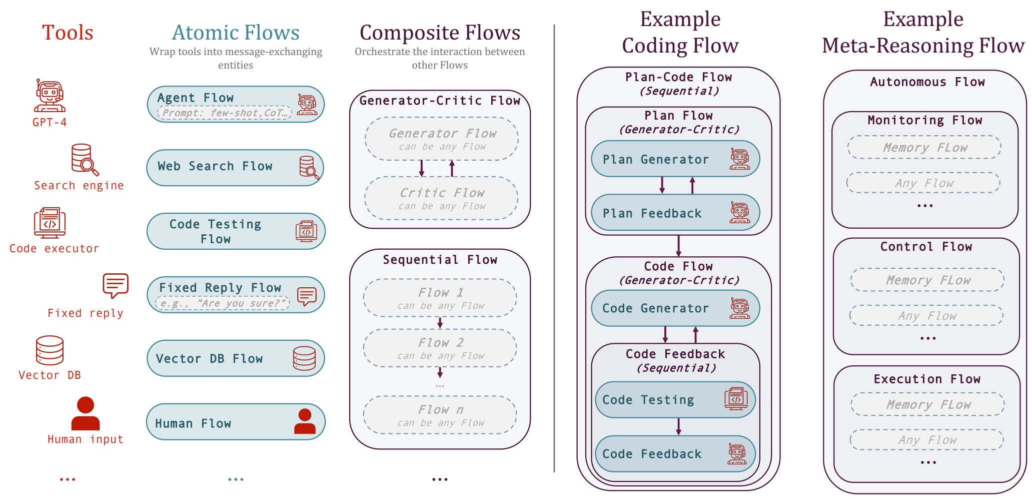


Figure 1: ***Flows* framework exemplified.** The first column depicts examples of tools, such as a search engine or a code executor. The second column depicts Atomic Flows constructed from the example tools. The third column depicts examples of Composite Flows defining structured interaction between Atomic or Composite Flows. The fourth column illustrates a specific Composite competitive coding Flow as an example of 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]

Currently, no general yet efficient abstraction exists for effectively modeling structured interactions. Previous work and existing frameworks, such as LangChain [7], Chameleon [25], HuggingGPT [42], 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 pre-defined 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, scheduling 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 went on parental leave), requiring Alice to adapt and modify the plan. 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 sub-processes can be modified or improved with minimal or no impact on the other components. It becomes evident that agents and tools should 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.

In this work, we argue that everything is a (control) flow defining potentially complex interactions between many diverse tools, where agents are just one type of tool. This induces a paradigm shift in conceptualizing and designing structured interactions involving AI agents, tools, and humans. For instance, an AI model used as a tool to pre-filter résumés is a flow of the simplest form — an atomic flow — comprising a single call.

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. Flows can be categorized into two types: *Atomic* Flows, which encapsulate the execution of a tool, and *Composite* Flows, which organize interactions between other Flows. In the *Flows* abstraction, we can model Alice as a Flow representing a higher-level meta-reasoning process that can support autonomous behavior and HappyCorp's hiring

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

process as a Flow orchestrating interactions between its sub-Flows (e.g., application pre-filtering and background checks) that can involve multiple sub-Flows themselves (e.g., recruiters and interviewers).

The *Flows* abstraction ensures modularity. Alice 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. 2). 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.

Overall, *Flows* is general enough to represent any collaboration, including all prior work on AI–AI and human–AI interactions, prompt engineering schemes, and tool augmentation (cf. Appendix A.1). Accompanying the conceptual *Flows* framework, we release the `aiFlows` library, which embodies the framework. The library comes with (i) FlowVerse: a repository of Flows that can be easily used, extended, and composed into novel, more complex Flows. Notably, *Flows* allows for existing "tools" (such as "models", "chains", "agents", etc.) to be readily incorporated by wrapping them in an Atomic Flow; (ii) a detailed logging infrastructure; and (iii) FlowViz: a visualization toolkit for inspecting the execution of Flows.

To showcase the potential of our framework and library, we have selected the task of competitive coding, a mind sport involving participants trying to solve problems defined by a specification. This task is highly challenging, even for advanced models like GPT-4. Therefore, it serves as an ideal scenario for exploring the potential benefits of employing various collaborative mechanisms and structured reasoning patterns. We have designed specific Flows that include planning Flows, allowing AI agents to strategize their approach; reflective Flows, which encourage AI agents to analyze and improve their previous answers; collaborative Flows, where one AI agent seeks feedback from another; and code testing Flows, which involve executing the code and refining the solution based on the code testing results. By combining these building blocks, we have created multiple competitive coding Flows, which we evaluated using newly curated coding problems from two popular platforms, CodeForces and LeetCode. Our study evaluates which patterns of interactions are most beneficial for competitive coding.

In summary, our contributions are as follows:

1. We propose *Flows*, a conceptual framework providing a high-level abstraction that enables the design and implementation of arbitrarily nested interactions with a substantial reduction of complexity, enabling flexibility considerably beyond existing libraries such as LangChain. *Flows* can represent *any* interaction and provides a common framework for reasoning about interaction patterns, specifying hypotheses, and structuring research, more broadly.
2. We demonstrate the potential of *Flows* by performing a systematic empirical study of complex interactions for solving competitive coding problems. Our results suggest that GPT-4 generalizes poorly on such difficult reasoning problems, with solve rates dropping from 72% on problems published before to 27% on problems published after the knowledge-cutoff date. However, complex interactions greatly improve performance: (ii) AI–AI interactions and tool augmentation improve the post-cutoff solve rate by over 20% (absolute); (iii) minimal human-AI collaboration improves it by 54%. Additionally, the experiments offer several more general 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. Strategic problem decomposition is a versatile strategy for creating opportunities where the human can be effectively incorporated as part of the Flow; (iii) the benefit from refinement depends on the level of grounding and the "novelty" of the information the feedback mechanism provides.
3. We open-source the `aiFlows` library, including a repository of Flows that can be easily used, extended, and composed into a novel, more complex Flows, as well as detailed documentation and tutorials.

2 Flows

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

2.1 *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 [44, 45], Stable Diffusion [37], 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 notable instance of a Composite Flow is ReAct [50]. ReAct corresponds to 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, after which the Flow corresponding to the selected action 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. This connection is made obvious thanks to *Flows*. More generally, the framework provides a unified view of all prior, and we make this explicit in Appendix A.1.

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

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 [15] — 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

[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.

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, as well as race conditions and deadlocks. These benefits are similar in nature to those observed in the domain of collaborations. The main distinction between the proposed framework and the *Actor* model resides in their respective communication protocols. Concretely, while the *Actor* model prescribes purely asynchronous communication, *Flows* also support 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 [3]. Nevertheless, 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 on the *Actor* model, such as the fact that every physically possible computation can be directly implemented using Actors [14], to transfer to *Flows*.

2.2 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 and interact with each other concurrently.

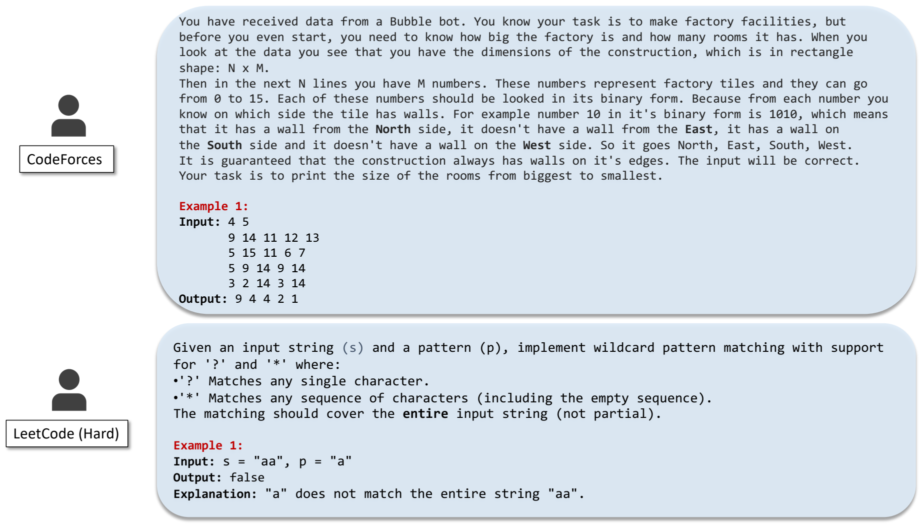
2.3 The `aiFlows` Library

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 Flows; (iii) FlowViz: a visualization toolkit to examine the Flows' execution through an intuitive interface.

3 Competitive Coding Flows

Flows provides an abstraction that makes it easy to design, implement, and reason about interactions of arbitrary complexity. In this work, we demonstrate the potential of the framework and the accompanying library on competitive coding, a particularly complex and challenging task. Fig. 2 presents illustrative examples of competitive coding problems consisting of a natural language description and input–output examples. The goal is to write executable code that will successfully pass all the hidden input–output test cases associated with the problem.

We focus our analysis on three canonical dimensions of interactions: (i) problem decomposition as structured reasoning; (ii) refinement with various feedback types; and (iii) human-AI collaboration. By providing a common language for clearly specifying interactions as well as a compositional capability to flexibly 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. They are depicted in Fig. 3.



CodeForces

```
You have received data from a Bubble bot. You know your task is to make factory facilities, but
before you even start, you need to know how big the factory is and how many rooms it has. When you
look at the data you see that you have the dimensions of the construction, which is in rectangle
shape: N x M.
Then in the next N lines you have M numbers. These numbers represent factory tiles and they can go
from 0 to 15. Each of these numbers should be looked in its binary form. Because from each number you
know on which side the tile has walls. For example number 10 in it's binary form is 1010, which means
that it has a wall from the North side, it doesn't have a wall from the East, it has a wall on
the South side and it doesn't have a wall on the West side. So it goes North, East, South, West.
It is guaranteed that the construction always has walls on it's edges. The input will be correct.
Your task is to print the size of the rooms from biggest to smallest.

Example 1:
Input: 4 5
       9 14 11 12 13
       5 15 11 6 7
       5 9 14 9 14
       3 2 14 3 14
Output: 9 4 4 2 1
```

LeetCode (Hard)

```
Given an input string (s) and a pattern (p), implement wildcard pattern matching with support
for '?' and '*' where:
•'?' Matches any single character.
•'*' Matches any sequence of characters (including the empty sequence).
The matching should cover the entire input string (not partial).

Example 1:
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Figure 2: **Examples of competitive coding problems from Codeforces and LeetCode.**

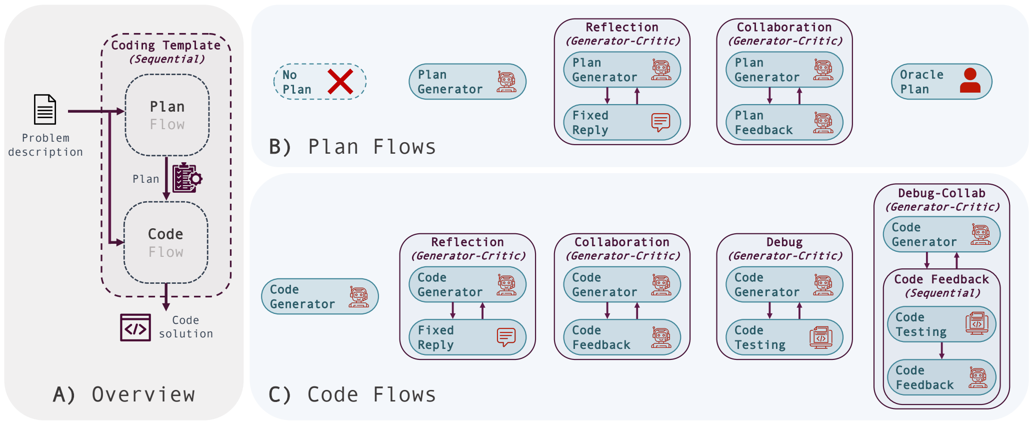


Figure 3: **Competitive coding Flows.** At the highest level, we consider planning as a specific structured reasoning pattern for problem decomposition. In particular, the Plan Flow generates a solution strategy and passes it to the Code Flow, which implements it, as depicted in A). B) and C) depict the different choices of sub-Flows used as Plan and Code Flows in the experiments. Importantly, 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.

Problem decomposition. Planning has been an integral intermediate step in recent work [25, 42, 50]. Similar problem decomposition is natural in the context of competitive coding as well. In particular, we approach the competitive coding 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. 3.

Human-AI collaboration. When designing human-AI collaborations, it is essential to take into account the costs of human interaction [2, 16, 29, 46]. By providing immense flexibility, *Flows* can support research on the design of interactions involving humans as computational building blocks in a way that maximizes the utility of the overall computation with minimal human effort. In the context of competitive coding, we hypothesize that human input would be most effectively incorporated at the plan level, asked to provide a short "oracle" plan in natural language. We operationalize this by an Atomic Human Flow, illustrated in Panel B of Fig. 3 as the *Oracle Plan*, serving as a plan Flow.

Refinement with various feedback types. Iterative refinement is a general problem-solving strategy that is successfully deployed across various disciplines [34, 35, 39, 41]. The principle 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 pattern 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 previously 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 the Flows using the following naming convention: *CodeFlowName* when no plan is generated and *PlanFlowName-CodeFlowName* otherwise.

4 Experimental Setup

Data. We conduct the evaluation on publicly available problems scraped from one of the most popular websites hosting competitive coding contests, Codeforces [28], and LeetCode [22], which covers a broad spectrum of problems ranging from easy interview questions to hard competitive coding problems (examples are illustrated in Fig. 2). The data collection spanned 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 consider *data contamination* [27]. Existing datasets for code evaluation like APPS [13], HumanEval [8], and CodeContests [24] lack problem release date information, confounding the assessment of models' memorization and generalization abilities. Our datasets, comprising problems published over an extended time period, up to a few months ago, rectify this issue.

Code testing and solution evaluation. Just like a human participant, the Debug Flow only has access to the input–output example pairs provided with the problem description and, at inference time, uses a local code testing infrastructure to evaluate (intermediate) code generations. Crucially, the examples provided with the problem description cover only a few simple cases. Therefore, generating outputs consistent with the examples does not imply that the generated code is correct. A solution is considered correct if it passes all the hidden test cases.

For computing the final results, we have implemented online evaluators that submit candidate solutions to Codeforces and LeetCode, and automatically scrape the resulting judgment. This mechanism ensures authoritative results. The performance reported in Table 1 is based on the online evaluations of the generated candidate solutions.

For many of the Codeforces problems, we managed to scrape a comprehensive set of hidden test cases, which allows us to use the local infrastructure for evaluating the final solutions as well. This enables quick development cycles with a very low number of false positives. LeetCode does not expose any hidden test cases.

Models and Flows. We experiment with the competitive coding Flows described in Sec. 3, where the Atomic Flows relying on an AI tool use GPT-4 [32]. The complete Flow configurations are 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 at least a single correct solution will be present in a set of k sampled candidates [8]. To better align with practical use cases, this work focuses on pass@1, where only one candidate solution is generated per problem, averaged over the problem set. We report a point estimate corresponding to the solve rate and a 95% confidence interval constructed from 1000 bootstrap samples.

Compute and cost. All the experiments, including the most complex Flows, can be performed on commodity hardware at a relatively low cost. For instance, the cost associated with querying the OpenAI API for generating Table 1 amount to $500.

5 Experimental Results

We first study the generalization ability of a few representative Flows and empirically identify GPT-4's knowledge-cutoff date. Next, we perform a focused analysis along the dimensions described in Sec. 3.

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

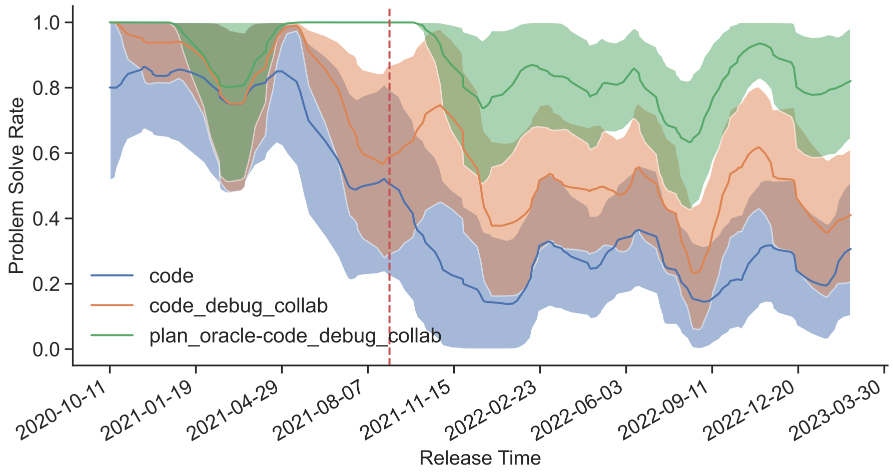


Figure 4: **Temporal analysis.** Performance is averaged over a sliding window of two months. The substantial drop in performance after the reported knowledge cutoff date for GPT-3/4 (the crimson vertical line) reveals limited generalization ability that can be alleviated through structured interactions.

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 ranging from October 2020 to April 2021 and averaging the performance over a sliding window of two months. The results are reported in Fig. 4.

We observe a substantial drop in performance centered around September 2021, which is consistent with the knowledge-cutoff date reported by OpenAI, denoted 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 that older problems are exposed to the model 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 publiched and crawled.

Notably, the results show a stark difference in performance between problems published pre- and post-knowledge-cutoff date for the Code Flow, decreasing from around 80% to around 23%. While still experiencing a substantial performance drop, the Code_Debug_Collab Flow doubles the solve rate on novel problems to around 45%. When provided with human input on a 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 structured 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.

5.2 Comparing Flows

Table 1 reports the performance of the systematically chosen set of Flows described in Sec. 3. 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 analyze the performance of problems published before and after the knowledge-cutoff date of September 2021.

	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.

Problem decomposition. The idea behind breaking down problem-solving into multiple steps is that each step provides information that makes the problem easier to solve. Concretely, planning before implementing the solution decouples the high-level reasoning from the code implementation. To analyze the effectiveness of this approach we can compare two Flows: the Atomic Code Flow, which generates code directly, and the Plan-Code Flow, which first generates a plan and then converts it into code. 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 in Codeforces (+8) and LeetCode easy and medium (+2.3 and +3.2). This trend reinforces the previous observations that AI systems exhibit different behavior when exposed to familiar versus novel data. More specifically, we hypothesize that for data potentially seen in training, the model is most effective in a context similar to the original one, and therefore adding intermediate steps not occurring in the original context harms performance. Finally, taking the confidence intervals into account, we see that while these trends are consistent, they are not statistically significant. Crucially, that does 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. We start by considering arguably the most basic form of feedback, a fixed string encouraging reflection. In the code Flows, we find that Code_Reflection improves performance for pre-cutoff problems and has no effect otherwise. Similarly, Code_Collaboration, in which the feedback is generated by another AI system (without grounding), leads to limited improvements, except in Codeforces post-cutoff problems, where an increase of +9.6 absolute points is observed. The most consistent and effective feedback relies on grounding, achieved by executing and testing the code (such feedback is provided in the Code_Debug and Code_Debug_Collab Flows). These improvements are most significant in the post-cutoff scenario of Codeforces, where performance increases from 26.9 without feedback to 47.5 when the refinement is based on AI-generated grounded feedback. While consistent, the improvements brought by these two types of feedback for LeetCode problems are smaller in magnitude. We hypothesize that this is a consequence of the examples provided in the LeetCode problem descriptions being more simplistic than those in Codeforces. As a consequence, this would lead to false positives and, thereby, incorrect grounding, directly affecting the quality of the feedback. This problem could be addressed by designing a Code_Testing Flow that, in addition to the examples used as public tests, uses a Test_Case_Generator Flow to generate additional test cases. We leave that for future work to explore Finally, in the plan Flows, where we consider Reflection and (non-grounded) Collaboration, we find that refinement does not lead to consistent statistically significant improvements. We can summarize these results as follows. In their essence, LLMs are conditional probability distributions. As refinement strategies contain the previous solution(s), used as a starting point, and feedback in their context, it is evident that the success of the process will depend on whether the combination of the two leads to an increased probability of a correct solution [19]. Therefore, the most useful feedback mechanism would provide novel (“surprising”) information in the context of the generator Flow that maximally increases the probability of a correct answer. This is why the best-performing refinement Flow involves grounding, which surfaces concrete problems with the current solution, paired with a Code_Feeeback Flow, which by providing concrete ideas on how to solve the specific problems with the given solution, substantially increases the probability of a correct solution. On the other hand, the gap between Plan_Code and Code provides evidence that the predicted plans are a noisy starting point for refinement, which in combination with the high level of abstraction and the lack of grounding, does not provide sufficient signal for the procedure to be effective.

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. Strategic problem decomposition is a versatile strategy for creating opportunities where the human can be effectively incorporated as part of the Flow; (iii) the benefit from refinement depends on the level of grounding and the “novelty” of the information the feedback mechanism provides.

This highlights the necessity for systematic research on interaction patterns and their corresponding benefits across diverse contexts. The *Flows* framework and the accompanying `aiFlows` library enable researchers to explore this direction.

6 Related Work

Existing libraries for modeling structured interactions. LangChain [7] has become the go-to library for creating applications using large language models. However, most recent works involving structured interaction, such as Cameleon [25], Camel [23], HuggingGPT [42], AutoGPT [36], BabyAGI [30], 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. Indeed, in Appendix A.1 we show how all of these works are specific Flow instances. The proposed framework makes it easy to change the structure of the interaction (the sub-Flows and how they interact), as well as flexibly nest and reuse Flows.

Competitive coding. Solving competitive coding problems involves more than just generating code. It requires reasoning and problem-solving abilities. Initial efforts focused on utilizing traditional approaches such as SMT solvers and enumerative search to generate solutions [4]. With the introduction of Transformers, AlphaCode [24] achieved significant progress by finetuning LLMs on GitHub code repositories and the CodeContests dataset (scraped from Codeforces). Recently, [52] proposed decomposing competitive coding 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, competitive coding is far from being solved. Even GPT-4 struggles with LeetCode and Codeforces problems, solving only a fraction of them[32]. Therefore, competitive coding presents itself as a perfect testbed for testing and developing complex reasoning Flows. To that aim, we have crated new datasets of Leetcode and Codeforces problems by scraping problems before and after the knowledge-cutoff date of GPT-4.

7 Discussion and Conclusion

Simplicity in implementation and design. The encapsulated Flow state and the standardized message-based interface between Flows make *Flows* highly modular. From the practical perspective, as demonstrated in our experiments, this results in a substantial (i) reduction of complexity in the design and implementation of open-ended interactions; (ii) flexibility in composing and swapping sub-Flows, making it easy to add complexity in a controlled manner. Furthermore, FlowViz simplifies development by offering an intuitive way to inspect Flow execution, while FlowVerse provides a repository of Flows that can be readily used in isolation or as a starting point in developing new Flows or extending existing ones.

Meta-reasoning Flows supporting autonomous behavior. In our experiments, we focus on the ability to design and implement Flows defining a complex but completely specified behavior. Crucially, this ability creates unprecedented opportunities for designing, implementing, and systematically studying higher-level, abstract, meta-reasoning patterns of interaction for AI systems. More concretely, we posit that "autonomous" AI systems [30, 36] need to move beyond single-turn interaction with an LLM as a control mechanism. Indeed, cognitive science research in metacognition and meta-reasoning suggests the existence of meta-level processes monitoring and controlling the processes performing the basic cognitive work [1]. The metacognitive monitoring reflects a subjective assessment of the probability of success or failure in a given task at different stages of problem-solving. Based on this signal, control processes are hypothesized to trigger different control decisions, such as attempting a solution or allocating resources toward acquiring more information about the problem. Exploring the development of similar mechanisms in the context of powerful AI could be a promising area of research.

Bounded rationality and optimization. Our experiments suggest that carefully designed complex interactions substantially improve generalization. However, this improvement comes with additional computation and latency costs. In practice, limits on time and resources apply to both AI systems and humans, with initial efforts in addressing the "finitary predicament" [11] problem in the scope of metareasoning for AI tracing back to more than three decades ago [5, 17, 18, 38]. Ideally, we would leverage the simplest Flow that could solve the task. While *Flows* provides the flexibility to support pattern, learning, and optimization requires data. To support research in optimizing the intrinsic tradeoff between the utility of execution and the time cost of deliberation, in `aiFlows`, we have implemented a mechanism for detailed logging of the execution runs.

Systematic research in interaction patterns. The results of our experiments reveal that the effectiveness of particular interaction patterns is not necessarily universal, but instead, there are many factors at play. For instance, when designing a collaboration between a generator and a critic, one needs to carefully reason about the information brought in the context of the generator by the feedback from the critic (e.g., how grounded and specific it is and why it could lead to an improved solution). As researchers, we need to clearly specify the patterns we are studying, clearly communicate our hypotheses, and study them carefully 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.

Human-AI collaboration Flows. The ultimate goal is to build AI systems that can extend our problem-solving abilities or at least make us more efficient. In the experiments, we demonstrate how strategic problem decomposition can lead to intermediate representation around which an interface for effective collaboration with a human can be designed such that a few sentences of input in natural language increase the solve rate by almost 50 absolute points. *Flows* provides the infrastructure that greatly simplifies the design, implementation, and systematic study of Human-AI collaboration as a key component of open-ended structured interactions.

On the one hand, *Flows* provides a high-level abstraction that enables the design and implementation of interactions of arbitrary complexity. On the other, it provides a common framework for reasoning about interaction patterns, specifying hypotheses, and structuring research. We hope that the framework will serve as a solid basis supporting practical and theoretical innovations in AI similar to how the Actor model has done the same for concurrent and distributed systems, paving the way toward ever more useful AI systems and taking us a small step closer to artificial general intelligence.

Acknowledgments

We would like to thank Doug Burger, Manoel Horta Ribeiro, Tim Davidson, and Venia Veselovsky for their very insightful feedback on the paper. West's lab is partly supported by grants from the Swiss National Science Foundation (200021_185043), Swiss Data Science Center (P22_08), H2020 (952215), Google, and Microsoft.

References

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

[2] 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. Guidelines for human-ai interaction. In Stephen A. Brewster, Geraldine Fitzpatrick, Anna L. Cox, and Vassilis Kostakos, editors, *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019*, page 3. ACM, 2019.

[3] Joe Armstrong. *Making reliable distributed systems in the presence of software errors*. PhD thesis, Royal Institute of Technology, Stockholm, Sweden, 2003.

[4] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. In *International Conference on Learning Representations*, 2017.

[5] Michael E Bratman, David J Israel, and Martha E Pollack. Plans and resource-bounded practical reasoning. *Computational intelligence*, 4(3):349–355, 1988.

[6] 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. Language models are few-shot learners. In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901. Curran Associates, Inc., 2020.

[7] Harrison Chase. Langchain. `https://github.com/hwchase17/langchain`, 2022.

[8] 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. Evaluating large language models trained on code. *ArXiv*, abs/2107.03374, 2021.

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

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

[11] Christopher Cherniak. Local optimization of neuron arbors. *Biol. Cybern.*, 66(6):503–510, 1992.

[12] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

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

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

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

[16] Eric Horvitz. Principles of mixed-initiative user interfaces. In Marian G. Williams and Mark W. Altom, editors, *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, 1999.

[17] Eric Horvitz and John S. Breese. Ideal partition of resources for metareasoning. *CoRR*, abs/2110.09624, 2021.

[18] Eric Joel Horvitz. *Computation and action under bounded resources*. stanford university, 1991.

[19] Martin Josifoski, Maxime Peyrard, Frano Rajič, Jiheng Wei, Debjit Paul, Valentin Hartmann, Barun Patra, Vishrav Chaudhary, Emre Kiciman, and Boi Faltings. Language model decoding as likelihood–utility alignment. In *Findings of the Association for Computational Linguistics: EACL 2023*, pages 1455–1470, Dubrovnik, Croatia, May 2023. Association for Computational Linguistics.

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

[21] Takeshi Kojima, Shixiang (Shane) Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 22199–22213. Curran Associates, Inc., 2022.

[22] LeetCode. Leetcode.com, 2023.

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

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

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

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

[27] Inbal Magar and Roy Schwartz. Data contamination: From memorization to exploitation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *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, 2022.

[28] Mike Mirzayanov. Codeforces.com, 2023.

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

[30] Yohei Nakajima. Babyagi. `https://github.com/yoheinakajima/babyagi`, 2023.

[31] 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. Show your work: Scratchpads for intermediate computation with language models. *CoRR*, abs/2112.00114, 2021.

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

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

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

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

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

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

[38] Stuart Russell and Eric Wefald. Principles of metareasoning. *Artif. Intell.*, 49(1-3):361–395, 1991.

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

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

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

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

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

[44] 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. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023.

[45] 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. Llama 2: Open foundation and fine-tuned chat models. *arXiv*, 2023.

[46] Helena Vasconcelos, Matthew Jörke, Madeleine Grunde-McLaughlin, Tobias Gerstenberg, Michael S. Bernstein, and Ranjay Krishna. Explanations can reduce overreliance on AI systems during decision-making. *Proc. ACM Hum. Comput. Interact.*, 7(CSCW1):1–38, 2023.

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

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

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

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

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

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

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

A Appendix

A.1 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) [6] 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) [47] 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) [49] 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) [9] 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) [53] 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**[40] 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**[50] 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 handle exceptions, while actions allow it to interface with external sources, such as knowledge bases or environments, to gather additional information.
8. **Parsel** [52] 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 postprocessing tools. Code is generated in a next step.
9. **REFINER** [33] 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** [26] 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) [20] 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** [48] 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.

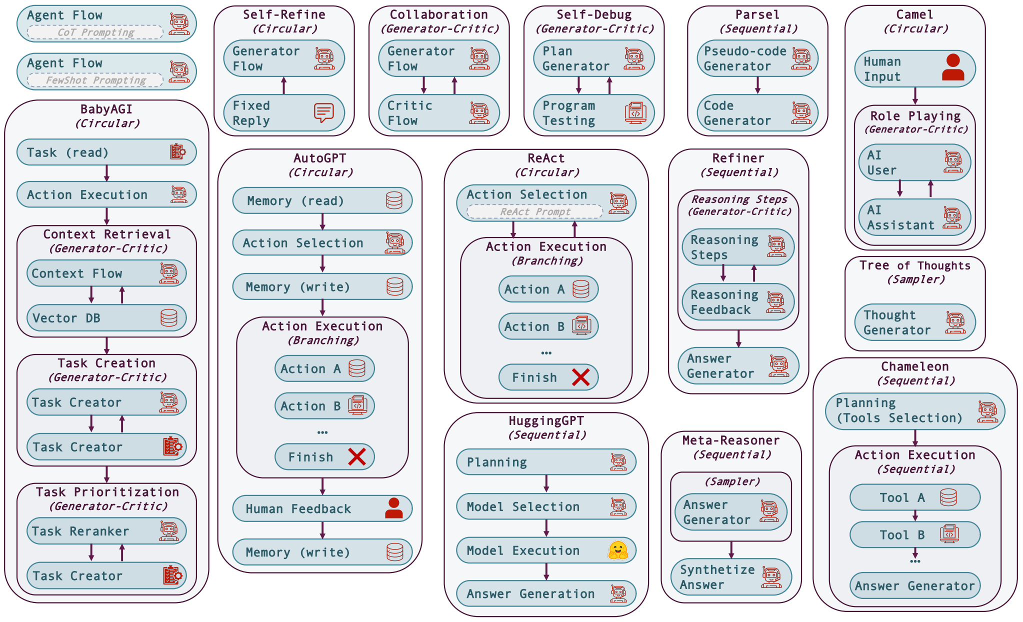


Figure 5: **Previous works are specific Flows.** We depict a selected subset of previous works incorporating structured reasoning and/or interactions between AI agents, tools, and humans, through the lens of the Flows framework. This demonstrates that Flows is a powerful language for describing, conceptualizing,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

13. **Self-Debug** [10] 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**[43] 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** [51] 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.
16. **HuggingGPT** [42] 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** [23] is a communicative agent framework involving inception prompting to guide chat agents toward task completion while maintaining consistency with human intentions.
18. **Chameleon** [25] 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**[36] 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** [30] 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.

Flows	**Flow Type**	**Interactions**				**Reasoning Patterns**		**Feedback**	**Learning**
		Self	Multi-Ag.	Human	Tools	Struct.	Plan		
FS [6]	Atomic	✗	✗	✗	✗	✗	✗	✗	✗
CoT [47]	Atomic	✗	✗	✗	✗	✓	✗	✗	✗
ToT [49]	Circular	✓	✗	✗	✓	✓	✗	✗	✗
PoT [9]	Seq.	✗	✗	✗	✓	✓	✗	✗	✗
M-CoT [53]	Seq.	✗	✗	✗	✗	✓	✗	✗	✓
ToolFormer [47]	Seq.	✗	✗	✗	✓	✓	✗	✗	✓
ReAct [50]	Circular	✗	✗	✗	✓	✓	✗	✗	✗
Parsel [52]	Seq.	✗	✓	✗	✓	✓	✓	✗	✗
REFINER [33]	Gen-Crit	✗	✓	✓	✗	✓	✗	✓	✓
Self-Refine [26]	Gen-Crit	✓	✗	✗	✗	✓	✗	✓	✗
RCI [20]	Gen-Crit	✓	✗	✗	✓	✓	✗	✓	✗
Self-Correct [48]	Gen-Crit	✓	✗	✗	✓	✓	✗	✓	✗
Self-Debug [10]	Gen-Crit	✓	✗	✗	✓	✓	✗	✓	✗
Reflexion [43]	Gen-Crit	✓	✗	✗	✓	✗	✗	✓	✗
Meta-Reasoner [51]	Seq.	✓	✓	✗	✗	✓	✗	✗	✗
HuggingGPT [42]	Seq.	✗	✓	✗	✓	✓	✓	✗	✗
Camel [23]	Circular	✗	✓	✓	✗	✓	✗	✓	✗
Chameleon [25]	Seq.	✗	✓	✗	✓	✓	✓	✗	✗
AutoGPT [36]	Circular	✓	✓	✗	✓	✓	✓	✓	✗
BabyAGI [30]	Circular	✗	✓	✗	✓	✓	✓	✗	✗

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

A.2 Prompting

We provide the prompts that were used to obtain the results in Section 5. Our evaluation is made possible thanks to the modular and compositional nature of *Flows*. Some of our experimental setups are deeply nested. We do not record all configurations and prompts, in cases where Flows build on each other, we try to avoid repetition. For the sake of brevity, we also only show prompts used for problems from the Codeforces dataset. A complete and exact replication of our results is made possible by the code in the project's GitHub repository.

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 adjusted coding Flow 5. Adding collaboration in the debugging setting is done by introducing a critic that provided feedback grounded on 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.3 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 `https://github.com/epfl-dlab/cc_flows`. 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.