Domain-Specific Programming For Agentic Workflows

Anonymous ACL submission

Abstract

While the use of agentic workflows becomes increasingly contextual, the programmability of it remains general (e.g., prompted steps and tools). In enterprise-related applications, such generalty often leads to undesired downgrade of workflow performance due to the mismatch between a business intention and the specific programming construct a workflow uses to implement it. In this paper, we highlighted the idea of domain-specific programming for agentic workflows, i.e., domain-specific languages (DSL) as a programming interface for both human and AI to create new agentic workflows. This path recovers the nature of agentic workflows as a type of program therefore introduces a systematic way to capture both the structure and semantics of it. Well-defined programming optimizations then become naturally compatible, e.g., check and infer "types". We instantiated the idea by a DSL SOPLang and a multi-agent system which automates the generation of agentic workflows (especially in industrial applications). The preliminary evaluation demonstrated the potential of our proposal to scale in real-world domains.

1 Introduction

While the use of agentic workflows becomes increasingly contextual, the programmability of it remains general (e.g., prompted steps and tools) (Hu et al., 2025; Zhang et al., 2025; Chen et al., 2024). In enterprise-related applications, such generalty often leads to undesired downgrade of workflow performance due to the mismatch between a business intention and the specific programming construct a workflow uses to implement it (Su et al., 2025). For example, contract review in enterprise legal settings requires structured extraction, risk analysis, compliance verification, and multi-role coordination (Su et al., 2025). General-purpose workflow generation methods often fall short in such high-stakes domains due to limited structural precision and insufficient domain awareness (Chalkidis et al., 2020). Certain rules—such as "All mandatory provisions shall be preserved without revision"—demand strict enforcement that prevailing methods cannot reliably guarantee (Zhang et al., 2024).

To solve the problem, we highlight the idea of domain-specific programming for agentic workflows, i.e., domain-specific languages (DSL) as a programming interface for both human and AI to create new agentic workflows. This path recovers the nature of agentic workflows as a type of program therefore introduces a systematic way to capture both the structure and semantics of it (Su et al., 2025). In this paper, we introduce SOPLang, a DSL inspired by the standard operating procedure, to orchestrate agents for industrial use. The key insight behind SOPLang is to enable arbitrary orchestration and domain adaptability for various scenarios.

Since workflow generation is reframed as a program synthesis task, well-defined programming optimizations then become naturally compatible, e.g., check and infer "types" (Hu et al., 2025). We prompt LLMs to generate candidate programs based on user instructions and apply simulated annealing to search for a near-global optimum that best aligns with the intended business logic. The preliminary evaluation demonstrates the potential of our proposal to scale in real-world domains.

The **main contributions** of this paper are summarized as follows.

1. We propose SOPLang, a DSL for agentic workflow generation in enterprise environments.
2. We highlight a multi-agent system instantiated from SOPLang programs and optimized via simulated annealing to search for high-quality workflow implementations under complex domain-specific instructions.

3. We introduce a Domain-Related Instruction Following dataset, on which our method outperforms prior approaches, demonstrating its potential to scale in real-world settings.

2 Language

2.1 Syntax

The core idea behind introducing the SOPLang language is to reframe agentic workflows as verifiable, composable, and optimizable programmatic representations, thereby enhancing the ability of LLMs and humans to collaboratively construct complex systems in enterprise contexts.

The syntax of SOPLang is formally defined using Extended Backus–Naur Form. Figure 1 presents a representative set of its syntax. A SOPLang program consists of a set of nodes connected via dependencies. There are two types of nodes: content nodes and operation nodes. Operation nodes are further divided into declarative and imperative categories. Declarative nodes specify logical constraints by evaluating expressions that determine whether certain criteria are met. Imperative nodes describe concrete actions (e.g., retrieving, extracting, summarizing) with explicit intent and execution semantics.

To support domain adaptability, SOPLang adopts a plugin-based architecture. New instructions can be registered externally by specifying their input/output schemas, operational semantics, and optional prompt templates. Each instruction is referenced via the instruction type and scoped by a domain tag, allowing workflows to incorporate specialized behaviors without modifying the core language specification.

2.2 Semantics

Interpreting a SOPLang program translates its node-based specification into an agentic workflow and executes it accordingly. We describe the process through symbolic evaluation with a set of evaluation rules. The state of the SOPLang program can be represented by 4-tuple $< p, \delta, \kappa, \mathfrak{D} >$. We denote by p the program counter points to the next node, δ the program store that accesses to program memory, κ the state of a node, and $\mathfrak{D}$ the dependency graph encoding control and data flow between nodes.

Figure 2 describes the principle evaluation rules of the SOPLang program. The (DEPE) rule describes the overall execution sequence. Intuitively, the prerequisite of executing a node is all its parents nodes are marked as "done". The (CONT) rule writes existing or generated data to the program memory for subsequent constraint validation or agent execution. The (DECL) rule serves as a control flow, enabling logic-based gating by evaluating conditions. If the specified condition is satisfied, the declarative node completes and its downstream dependencies are allowed to proceed. The (IMPE) rule resolves each imperative instruction against the current domain context. It looks up the corresponding plugin implementation and initiates the associated agent behavior accordingly. This allows the same instruction type to exhibit different behavior in different domains.

$\mathcal{P}$	$::= \{\mathcal{N}\}\{\mathcal{D}\}$	SOPLang program
$\mathcal{N}$	$::= \mathcal{C} \mid \mathcal{O} \mid \mathcal{J}$	Node type

$\mathcal{C}$	::= `"node"` *node_id*
	`"type" ":" "content"`
	`["data" ":" {`*data_id* `":"` *literal* `}]`
		Content node

$\mathcal{O}$	$::= \mathcal{O}_{\mathcal{D}} \mid \mathcal{O}_{\mathcal{I}}$	Operation node

$\mathcal{O}_{\mathcal{D}}$	::=`"node"` *node_id*
	`"operation_type" ":" "declarative"`
	`"input" ":" {`*data_id*`}`
	`"output" ":" {`*data_id*`}`
	`"condition" ":"` $\mathcal{L}$
		Declarative operation

$\mathcal{O}_{\mathcal{I}}$	::=`"node"` *node_id*
	`"operation_type" ":" "imperative"`
	`"instruction_type" ":"` $\mathcal{I}$
	`"input" ":" {`*data_id*`}`
	`"output" ":" {`*data_id*`}`
	`"prompt" ":" {` *data_id* | *literal* `}`
	`["domain" ":"` *domain_tag*`]`
		Imperative operation

$\mathcal{I}$	::=`"Retrieve"` | `"Extract"` | `"Summarize"` | `"Propose"` | `"Refine"` | `"Refactor"`	Instruction type
$\mathcal{D}$	$::=$ *node_id* $\rightarrow$ *node_id*	Dependency
$\mathcal{L}$	$::=$ *data_id* $\oplus$ *literal* $\mid \neg \mathcal{L}$ $\mid$ `"("` $\mathcal{L}$ `")"` $\mid \mathcal{L} \oplus \mathcal{L}$	Logic expression
$\oplus$	$::= = \mid \neq \mid < \mid \leq \mid > \mid \geq$ $\mid \in \mid \subset \mid \wedge \mid \vee \mid \Rightarrow$	Binary operation

Figure 1: A representative set of the syntax of SOPLang

3 Program Synthesis and Optimization

Figure 3 describes the overall architecture of our system. We employ top-p sampling to generate multiple alternative SOPLang programs and apply a parallel simulated annealing algorithm to refine and select high-quality solutions.

Reward function. To guide the process, the reward function is constructed to follow a hierarchical objective optimization, where the primary ob-

$$\frac{\begin{array}{c}\forall i, j \in \{1, \ldots, n\} \\ \forall \mathcal{N}_j \in \mathfrak{D}^{-1}(\mathcal{N}_i) \wedge \kappa[\mathcal{N}_j] = done\end{array}}{\langle \mathcal{N}_i, \delta_{i-1}, \kappa_{i-1}, \mathfrak{D} \rangle \rightarrow \langle \emptyset, \delta_i, \kappa_i, \mathfrak{D} \rangle} \quad \text{(DEPE)}$$

$$\frac{lookup(\mathcal{C}.data) \Rightarrow v}{\langle \mathcal{C}, \delta, \kappa, \mathfrak{D} \rangle \rightarrow \langle \emptyset, \delta[output := v], \kappa[\mathcal{C} := done], \mathfrak{D} \rangle} \quad \text{(CONT)}$$

$$\frac{\begin{array}{c}\forall \mathcal{L} \in \mathcal{O}_{\mathcal{D}}.condition \\ eval(\mathcal{L}) = true\end{array}}{\langle \mathcal{O}_{\mathcal{D}}, \delta, \kappa, \mathfrak{D} \rangle \rightarrow \langle \emptyset, \delta, \kappa[\mathcal{O}_{\mathcal{D}} := done], \mathfrak{D} \rangle}$$

$$\frac{\begin{array}{c}\forall \mathcal{L} \in \mathcal{O}_{\mathcal{D}}.condition \\ eval(\mathcal{L}) = false\end{array}}{\langle \mathcal{O}_{\mathcal{D}}, \delta, \kappa, \mathfrak{D} \rangle \rightarrow \texttt{halt}} \quad \text{(DECL)}$$

$$\frac{\begin{array}{c}\mathcal{O}_{\mathcal{I}}.instruction\_type \Rightarrow \mathcal{I} \\ \mathcal{O}_{\mathcal{I}}.domain \Rightarrow \mathcal{D} \\ \mathcal{O}_{\mathcal{I}}.prompt \Rightarrow \mathcal{P} \\ initiateAgent(\mathcal{I}, \mathcal{D}) \Rightarrow \mathcal{A} \\ \mathcal{A}.\text{exec}(\mathcal{P}) \Rightarrow v\end{array}}{\langle \mathcal{O}_{\mathcal{I}}, \delta, \kappa, \mathfrak{D} \rangle \rightarrow \langle \emptyset, \delta[output := v], \kappa[\mathcal{O}_{\mathcal{I}} := done], \mathfrak{D} \rangle} \quad \text{(IMPE)}$$

Figure 2: A representative set of the symbolic evaluation of the SOPLang program

jective ensures accuracy, and the secondary objective promotes efficiency. We denote the accuracy of program x by f_0, token costs, LLM calls, and other secondary objective by f_i.

$$R(x) = \begin{cases} -1, & f_0(x) < \epsilon_0 \\ \alpha(f_0(x) - \epsilon_0) - \sum \beta_i f_i(x), & \epsilon_0 \leq f_0(x) < \epsilon_1 \\ 1 - \sum \beta_i f_i(x), & f_0(x) \geq \epsilon_1 \end{cases}$$

Simulated annealing. Algorithm 1 outlines the simulated annealing process. At each iteration, a random mutation is applied to the current program x, such as modifying a node, adding or removing a node, or altering a dependency. The mutated program x' is evaluated by the reward function $R(x')$ and replaces x with a probability based on evaluation.

Multi-chain strategy. We leverage multi-chain strategy to enhance search diversity and improve the ability to escape local optima. High-temperature chains promote exploration through more flexible acceptance of worse solutions, while low-temperature chains are more conservative and focus on local refinement. Periodic state exchanges between chains allow diverse candidates to be refined, improving overall search effectiveness.

Algorithm 1 Multi-Chain Simulated Annealing

Require: Initial programs $\{x_1, \ldots, x_K\}$, temperatures $\{T_1 < T_2 < \cdots < T_K\}$, reward function $R(x)$, total steps N, exchange interval E
Ensure: Optimal solution x^*

Initialize $r_k \leftarrow R(x_k)$ and $x_k^* \leftarrow x_k$ for all k
for $t = 1$ to N **do**
  for $k = 1$ to K **do**
    $x' \leftarrow \text{Mutate}(x_k)$
    if $R(x') - R(x_k) > 0$ **then**
      $x_k \leftarrow x'$
    else
      $p_{\text{accept}} \leftarrow \exp\left(\frac{R(x') - R(x_k)}{T_k}\right)$
      if Random(0,1) $< p_{\text{accept}}$ **then**
        $x_k \leftarrow x'$
      end if
    end if
    if $R(x_k) > r_k$ **then**
      $x_k^* \leftarrow x_k$; $r_k \leftarrow R(x_k)$
    end if
  end for
  if $t \bmod E = 0$ **then**
    for $k = 1$ to $K - 1$ **do**
      $\Delta \leftarrow \left(\frac{1}{T_k} - \frac{1}{T_{k+1}}\right)(R(x_{k+1}) - R(x_k))$
      $p_{\text{swap}} \leftarrow \min(1, \exp(\Delta))$
      if Random(0,1) $< p_{\text{swap}}$ **then**
        Swap $x_k \leftrightarrow x_{k+1}$
      end if
    end for
  end if
end for
$x^* \leftarrow \arg\max_{x_k^*} R(x_k^*)$
return x^* =0

4 Experiment

4.1 Experiment Setup

Datasets. We introduce the Domain-Related Instruction Following (DRIF) dataset for our experiments. DRIF consists of 300 instructions derived from real-world industrial demands, categorized into three groups based on their level of domain relevance (i.e., The hard set refers to instructions with strong domain specificity, requiring specialized knowledge or structured understanding of domain semantics).

Metrics. For each instruction, we generate a serial of questions for LLM to evaluate the accuracy of generated content.

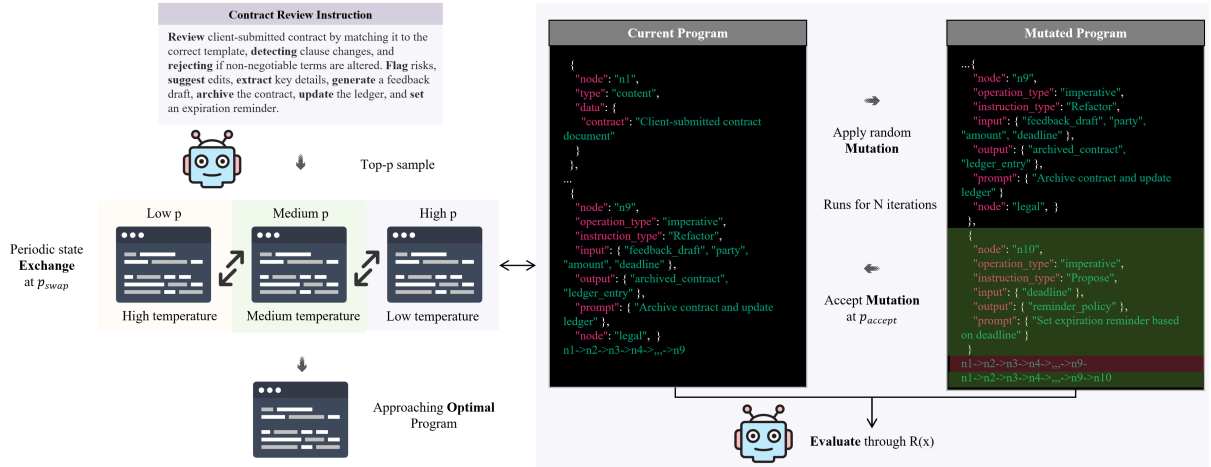

Figure 3: Illustration of the system architecture.

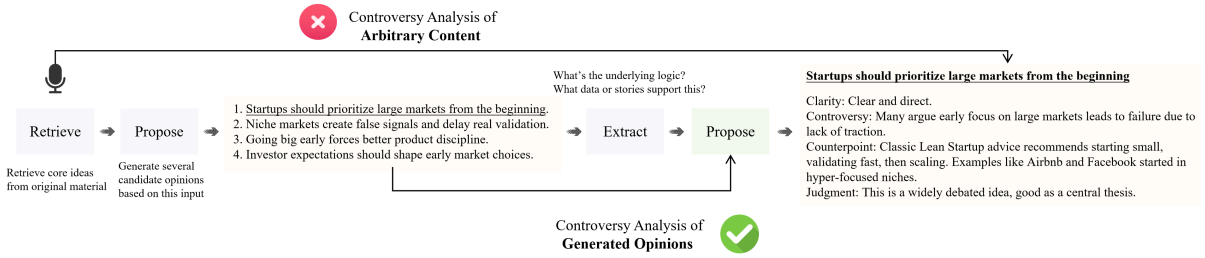

Figure 4: Illustrative example of experiment on DRIF.

Baselines. We compare the workflow generated through our method to manual design of agentic systems, including Chain-of-Thought (CoT) (Wei et al., 2022), Self Consistency CoT (Cot SC) (Wang et al., 2023), MultiPersona Debate (Wang et al., 2024), and Role assigment (Xu et al., 2025), and automated design of agentic workflow by ADAS (Hu et al., 2025) and AFLOW (Zhang et al., 2025).

Implementation. We employ different models at distinct stages of the pipeline. During the execution stage, we use DeepSeek-R2, whereas the program optimization stage leverages GPT-4o-mini. To ensure a fair comparison across all approaches, we apply the same model configuration and set the temperature to 1 for other approaches.

4.2 Results and Analysis

Our method outperforms other approaches on the DRIF dataset. Figure 4 depicts a workflow generated by our method that emphasizes targeted controversy analysis on opinions derived from a specific propose node, rather than performing such analysis on unstructured or arbitrary inputs. This precise scoping is enabled by domain-specific workflow design, which assigns clear semantics to each node and supports cascading reasoning over structured content.

5 Related-work

Agentic workflow. Early approaches to workflow construction primarily rely on manually designed strategies. Multi-agent methods such as MultiPersona Debate (Wang et al., 2024) and Role Assignment (Xu et al., 2025) coordinate specialized agents via fixed prompting templates. More recent work has shifted toward automated agentic workflow design. ADAS (Hu et al., 2025) encodes workflows as executable programs and optimizes them through historical reuse, while AFLOW (Zhang et al., 2025) employs search and refinement to construct workflows automatically.

6 Conclusion

This paper introduces SOPLang, a DSL for automating workflow generation in enterprise settings. By supporting agent orchestration and domain-adapatability, SOPLang captures industrial task semantics more effectively than general-purpose approaches. Preliminary results indicate the effectiveness of our approach across diverse scenarios.

Limitations

This work assumes access to structured domain knowledge, as SOPLang requires manually defined instruction schemas and semantics to support domain extensibility. This reliance may limit scalability in low-resource or rapidly evolving domains, where formal task definitions are unavailable or costly to produce. The need for manual schema design and domain-specific prompt templates also introduces overhead that may hinder deployment in dynamic or unfamiliar settings.

The optimization procedure is based on parallel simulated annealing, a heuristic method that can be computationally expensive and prone to suboptimal convergence, especially for long or highly interdependent workflows. While effective in guiding program search, this approach lacks theoretical convergence guarantees and may require extensive sampling to reach satisfactory solutions in complex settings.

In addition, evaluation is limited to a domain-specific dataset centered on industrial tasks, which constrains the assessment of generalizability to broader application areas. The instruction types and task structures in the dataset may not reflect the diversity or ambiguity present in other domains such as programming, math, or open-domain question answering. As a result, the empirical findings may not fully capture the performance and adaptability of the proposed approach in more heterogeneous or less formalized environments.

References

Ilias Chalkidis, Manos Fergadiotis, Prodromos Malakasiotis, Nikolaos Aletras, and Ion Androutsopoulos. 2020. LEGAL-BERT: The muppets straight out of law school. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 2898–2904, Online. Association for Computational Linguistics.

Minghao Chen, Yihang Li, Yanting Yang, Shiyu Yu, Binbin Lin, and Xiaofei He. 2024. Automanual: Constructing instruction manuals by llm agents via interactive environmental learning. In *Advances in Neural Information Processing Systems*, volume 37, pages 589–631. Curran Associates, Inc.

Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated design of agentic systems. *Preprint*, arXiv:2408.08435.

Jinwei Su, Yinghui Xia, Ronghua Shi, Jianhui Wang, Jianuo Huang, Yijin Wang, Tianyu Shi, Yang Jingsong, and Lewei He. 2025. Debflow: Automating agent creation via agent debate. *Preprint*, arXiv:2503.23781.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *Preprint*, arXiv:2203.11171.

Zhenhailong Wang, Shaoguang Mao, Wenshan Wu, Tao Ge, Furu Wei, and Heng Ji. 2024. Unleashing the emergent cognitive synergy in large language models: A task-solving agent through multi-persona self-collaboration. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 257–279, Mexico City, Mexico. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA. Curran Associates Inc.

Benfeng Xu, An Yang, Junyang Lin, Quan Wang, Chang Zhou, Yongdong Zhang, and Zhendong Mao. 2025. Expertprompting: Instructing large language models to be distinguished experts. *Preprint*, arXiv:2305.14688.

Honghua Zhang, Po-Nien Kung, Masahiro Yoshida, Guy Van den Broeck, and Nanyun Peng. 2024. Adaptable logical control for large language models. In *Advances in Neural Information Processing Systems*, volume 37, pages 115563–115587. Curran Associates, Inc.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. 2025. Aflow: Automating agentic workflow generation. *Preprint*, arXiv:2410.10762.