Fine-Tune an SLM or Prompt an LLM? The Case of Generating Low-Code Workflows

Orlando Marquez Ayala
ServiceNow
orlando.marquez@servicenow.com

Patrice Bechard
ServiceNow
patrice.bechard@servicenow.com

Emily Chen
ServiceNow
emily.chen@servicenow.com

Maggie Baird
ServiceNow
maggie.baird@servicenow.com

Jingfei Chen
ServiceNow
jingfei.chen@servicenow.com

Abstract

Large Language Models (LLMs) such as GPT-4o can handle a wide range of complex tasks with the right prompt. As per token costs are reduced, the advantages of fine-tuning Small Language Models (SLMs) for real-world applications — faster inference, lower costs — may no longer be clear. In this work, we present evidence that, for domain-specific tasks that require structured outputs, SLMs still have a *quality* advantage. We compare fine-tuning an SLM against prompting LLMs on the task of generating low-code workflows in JSON form. We observe that while a good prompt can yield reasonable results, fine-tuning improves quality by 10% on average. We also perform systematic error analysis to reveal model limitations.

CCS Concepts

• **Computing methodologies** → **Natural language processing**; **Machine learning**; • **Information systems** → *Specialized information retrieval*; Enterprise information systems.

Keywords

Large Language Models, Generative AI, Retrieval-Augmented Generation, Task Decomposition, Workflows

1 Introduction

Generative AI is pushing the boundaries of AI-based software. As Large Language Models (LLMs) become more capable, they have become an integral part of the software stack. Although prompting state-of-the-art models such as GPT-4o [15], Gemini 2.0 Flash [12], or Claude 3.5 Sonnet [1] generally suffice for traditional tasks like question-answering or summarization [9], it is not always the case on domain-specific tasks [24].

While building *Flow Generation*, an application that generates low-code workflows based on textual user requirements, we explored whether fine-tuning a language model was necessary to achieve the desired software quality. To reduce infrastructure needs, we only considered Small Language Models (SLMs), having less than 15 billion parameters.

Enterprise workflows are a common way to automate repetitive yet complex processes. They are represented as a series of steps that are executed under certain conditions to fulfill specific goals. While low-code workflows tend to be built in easy-to-use interfaces, creating them still requires expert knowledge of the enterprise system.

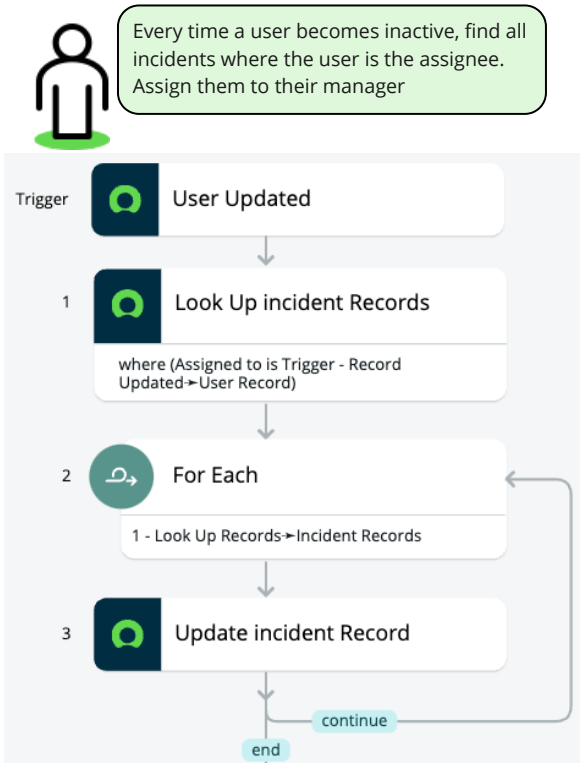

Figure 1: Task consists of generating a complete workflow from a user requirement.

Figure 1 shows an example of a simple four-step workflow. Some of the challenges in this task are:

- Every step must exist in the system environment; they vary by installation and users can add custom steps.
- Workflow structure must follow a set of well-defined rules, using concepts such as conditions (e.g., `IF`) and loops (e.g., `FOREACH`).
- Every step generates outputs of data types such as `integer` or `boolean`, which can be used in subsequent steps.
- Steps have inputs that can refer to database tables, columns, and values. For example, the *Look Up Incident Records* step in Figure 1 matches rows in the `incident` table where the column `assigned_to` has the same value as the `user` record from the trigger step.

KDD Workshop '25, Toronto, Canada
2025.

The simplest approach to generate low-code workflows is to use prompt engineering on an off-the-shelf LLM. This would be satisfactory if the steps were high-level descriptions and the logic elements were simple. But to obtain reasonable results in this task, we had to decompose it into sub-tasks [28] and use Retrieval-Augmented Generation (RAG) [10] at various points in the pipeline. We found out that, while prompting an LLM gives reasonable results, we can do better by fine-tuning an SLM.

Our approach was to create a small yet representative training dataset for the sub-tasks comprising *Flow Generation* and fine-tune Mistral-Nemo-12B-Base [20]. For evaluation, we labeled about a thousand workflows coming from ten domains (around 100 per domain). We also created a special dataset by inviting expert enterprise users to interact with the tool and collecting data generated from their usage. We compare our approach against an instruction fine-tuned SLM as the baseline, as well as against a variety of closed and open-source LLMs, showing the benefits of fine-tuning. As our task is domain-specific, we devised a metric called *Flow Similarity* (`FlowSim`), a version of tree edit distance that represents workflows as trees.

Our contributions are the following:

- We present a case study for how we built an application that generates low-code workflows based on textual user requirements.
- We demonstrate that fine-tuning an SLM gives better results than prompting much larger LLMs in this domain-specific task.
- We introduce a process to conduct systematic error analysis that reveals model limitations and complements our metrics.

2 Related Work

Prompting [5, 19] has emerged as the predominant method for using LLMs, significantly reducing the effort required compared to fine-tuning [23, 27]. However, for domain-specific applications requiring extensive prior knowledge, practitioners often encounter limitations such as constrained context windows [22], which hinder the model's ability to capture the full scope of the task. To address this, alternative approaches such as RAG [16] or domain-specific fine-tuning [14] are employed to incorporate relevant knowledge. In various industry-specific settings, specialized models have demonstrated superior performance over state-of-the-art general-domain LLMs [6, 25, 30]. We add to this body of work by showing that a fine-tuned SLM performs better than prompting LLMs in low-code workflow generation.

The domain of *workflows* has been extensively explored in previous work (2, 18, 29, 31, *inter alia*). Notably, Bassamzadeh and Methani [3] compare an approach using RAG against fine-tuning a model for workflow generation, while Fan et al. [8] propose a synthetic data generation pipeline to enhance generalization. In our case, we combine RAG with fine-tuning an SLM to obtain the highest possible quality.

Evaluation methodologies vary across tasks, with each task having its own quantitative metric for model quality but also its own qualitative error analysis approach [4, 7, 11, 26]. In this work, we outline an error analysis approach for a structured output task and leverage it to gain deeper insights into model failure modes.

3 Methodology

Because workflows are complex structured outputs requiring diverse data in their steps and inputs, we designed a pipeline that relies on RAG and generates the workflow iteratively by asking the LLM to solve sub-tasks.

3.1 Task Decomposition

In our domain, workflows are represented as JSON documents, which allows us to break the generation of complete workflows in two stages:

(1) Given a natural language requirement, generate the plan or ***outline*** of the workflow, identifying the step names, the order of execution and, crucially, extracting an annotation (description) from the requirement for each step.
(2) For every step in the outline, use the annotation to gather the necessary data from the environment and generate the ***step inputs***.

The glue between the two stages are annotations, which also serve as an explanation for the generated step. They allow the model to fill step details, as long as these details are provided by the user. When we retrieve data such as table or column names, these annotations are a key part of the search input.

Figure 2 shows the JSON representation of the trigger and first component steps for the workflow in Figure 1. The lines in green are generated in the first stage, as part of the outline, and the step inputs in red are generated as part of the second stage.

```
"trigger": {
  "annotation": "every time a user becomes inactive",
  "type": "record_update",
  "inputs": [
   {
     "name": "table",
     "value": "sys_user"
   },
   {
     "name": "condition",
     "value": "activeCHANGESTOfalse"
   }
  ]
},
"components": [
{
  "annotation": "find all incidents where the user is the assignee",
  "name": "look_up_records",
  "step": 1,
  "inputs": [
    {
      "name": "table",
      "value": "incident"
    },
    {
      "name": "conditions",
      "value": "assigned_to={{record_update.current}}"
    }
  ]
 ...
```

Figure 2: Sample JSON representation of the trigger and first component steps for the workflow in Figure 1.

3.2 Retrieval-Augmented Generation

It has been shown that RAG is necessary to reduce hallucinations when generating structured outputs [2, 3]. In our pipeline, RAG is used to suggest the following to the LLM:

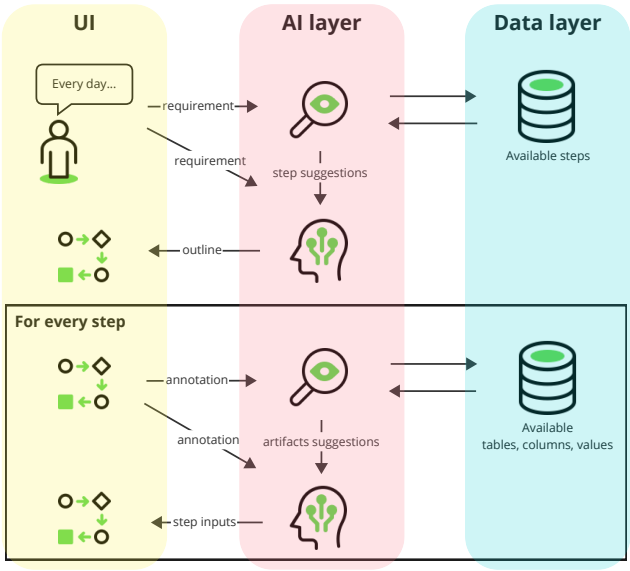

Figure 3: System architecture with UI, AI, and data layers.

- Steps to use in the outline. Since the available steps vary across installations of the system, this includes custom steps added by users.
- Table names, column names, and values used in steps that read and modify database records in the system. These names and values are used in step inputs.

For example, referring to Figure 2, step names such as `record_update` and `look_up_records` can be retrieved based on the text *Every time a user becomes inactive, find all incidents where the user is the assignee. Assign them to their manager.*. Then the table name `sys_user`, column name `active`, and column value `false` are retrieved from *every time a user becomes inactive*.

3.3 Pipeline

Figure 3 shows the architecture for the overall pipeline. The UI receives the user requirement and displays the workflow outline and step inputs as they are generated. The AI layer contains the LLM and the retriever. The data layer stores the indexed sources of data, from where the retriever suggests steps and artifacts (tables, columns, values) to the LLM. This layer can be replaced in every installation of the system to allow the LLM to generate output specific to each customer. Note that for a workflow containing N steps, there will be a maximum of $N + 1$ LLM calls: one for the outline and one to generate the inputs of each step that accept inputs.

Fine-tuning the SLM and the retriever was done using standard approaches: supervised fine-tuning and contrastive loss training.

4 Experiments

4.1 Datasets

The training dataset comprises two tasks, each corresponding to the two pipeline stages. The first task is `createFlow`, where the model generates the outline given the natural language requirement and basic workflow metadata. The second task is `populateInputs`, where given a flow at step N, the annotation of step N, and any required environment data, the model generates the inputs of this step. In the former, the model receives suggestions for steps while in the latter, it receives suggestions for artifacts that cannot be found deterministically, such as table names that users refer to in the requirement.

We extracted thousands of workflows from internal deployments of the enterprise system for labeling. After discarding those too short or too long, we selected 1,512 workflows. A team was trained to express in natural language a description or requirement of what these workflows accomplish. A downside of using existing workflows is that they are complete while users will create their workflows in draft form and will add complexity iteratively. We therefore created 766 simpler workflows synthetically using domain expertise aiming to cover the data distribution as much as possible.

For evaluation, we similarly extracted and labeled workflows from 10 customer deployments, around 100 from each. Since each deployment comes from different domains such as retail and banking, the steps and table names differ from those found in the training dataset. We call this dataset the out-of-domain (OOD) set. However, the same downside applies to this evaluation dataset: these are complete workflows. To address this, we invited expert users of the manual system to simulate interacting with the application and generate workflows. Their submitted requirements formed a TEST set of 108 workflows. Table 1 summarizes the number of samples used for training and evaluation.

Table 1: Number of examples for training (Internal and Synthetic) and evaluation (TEST and OOD).

Dataset	# `createFlow`	# `populateInputs`
Internal	1,512	4,709
Synthetic	766	2,310
TEST	108	367
OOD	1,072	4,958

4.2 Models

For SLMs, we considered several models such as StarCoderBase-7B [17] and Llama-3.1-8B [13], but we chose Mistral-Nemo-12B-Base [20] because this model architecture is well-supported and optimized in our enterprise system. However, we obtained similar results with other SLMs as the key ingredient for our fine-tuning is the training data.

As the baseline, we used a model of the same architecture and size that has been instruction fine-tuned but not on data from our workflow domain: Mistral-Nemo-12B-Instruct [20]. This allows us to validate that domain-specific data is helpful. As for LLMs, we try to cover the current landscape by comparing against:

- **Closed-source:** GPT-4o-mini, GPT-4o, and Gemini-2.0-Flash.
- **Open-source:** Llama-3.3-70B-Instruct [13]
- **Reasoning:** o3-mini [21] with medium reasoning

We crafted two prompt templates, one for each task, to use on all models that were not fine-tuned. Both contain sections for Context, Task definition, Inputs, Guidelines, Constraints with examples, and Output format. Before prompting the LLMs, these templates get populated with data from the system and the retrieved suggestions. In the case of the `populateInputs` prompt, we dynamically fill

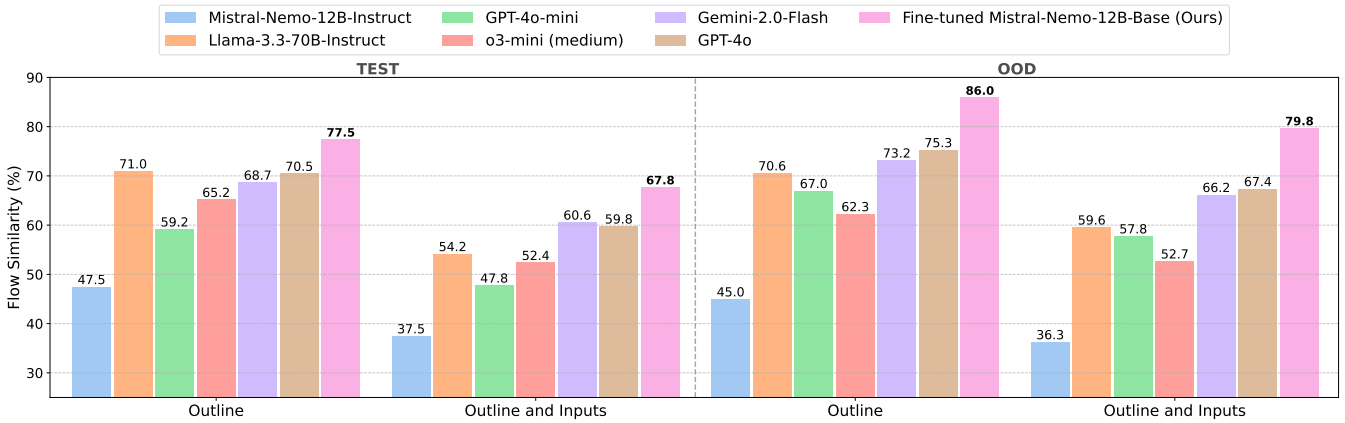

Figure 4: Flow similarity results obtained on the TEST and OOD sets for Outline and Outline with inputs.

the template with instructions according to the inputs to populate (e.g., input types). Table 2 gives an idea of the complexity of our prompts by listing the number of input tokens using the GPT-4o tokenizer on the TEST set.

Table 2: Number of tokens in templates and average / standard deviation for prompts sent to the LLMs.

	`createFlow`	`populateInputs`
# tokens in template	3,948	2,225
Avg # prompt tokens	5,958	3,732
Std Dev # prompt tokens	282	728

4.3 Metrics

We devised our own metric called *Flow Similarity* (`FlowSim`) to compare generated and expected workflows. As workflows can be represented as trees, similar to how computer programs can be represented as abstract syntax trees, we use the tree edit distance algorithm [32] to compute a similarity score, where higher is better. We found that this metric correlates well with human evaluation (details in Appendix A). A labeled expected workflow is required per user requirement, but evaluation can be automated. Appendix B includes a sample workflow tree.

Our metric, however, has a few limitations. First, we compare against only one workflow whereas there may be several possible versions for the same requirement. Second, we compute exact matches for input strings. Third, it ignores that generated workflows may break basic structure rules (e.g., having an `ELSE` step without an `IF`). The first two drawbacks can be addressed by having more than one reference workflow and by using sentence similarity for string inputs, but we leave this to future work. We address the third drawback by computing the percentage of generated examples with at least one such structure error. While the most complete metric is **`FlowSim` of outline and inputs**, we also report **`FlowSim` of outline**, and the **percentage of examples with structure errors**.

5 Results

Figure 4 shows the results for our fine-tuned SLM and all other LLMs on the small TEST and the larger OOD datasets. Mistral-Nemo-12B-Instruct performs poorly across the board confirming that SLMs require workflow in-domain training data. LLMs of the "mini" sort, even o3-mini with medium reasoning, also do not perform well.

When we prompt larger LLMs, we see competitive results, especially with Gemini-2.0-Flash and GPT-4o. As generating the outline is a simpler task, we see a larger gap between our fine-tuned SLM and these models when we generate the complete workflow (outline and inputs). This gap is 7.2% on TEST but 12.4% on OOD, around 10% if we average the two. All models perform worse when the requirements denote unfinished workflows (TEST set), suggesting a discrepancy between requirements written by expert users and our annotators.

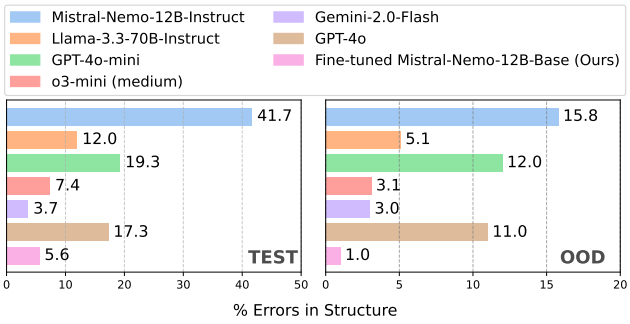

Figure 5: Percentage of examples with structure errors on the TEST and OOD sets. Lower is better.

While `FlowSim` values may be high, it could be that the generated workflow had obvious structure errors, which could severely impact usability or require complex post-processing rules. As shown in Figure 5, while GPT-4o performs reasonably well overall, it makes many such errors. With the same prompt, Gemini-2.0-Flash yields even fewer structure errors than our fine-tuned SLM on the TEST set. Refer to Appendix C for further analysis.

Our last quantitative comparison helps us determine whether suboptimal RAG is primarily responsible for errors. Here, we consider only outline generation, as only one retriever call is made to obtain step suggestions, and we consider only the TEST dataset. Table 3 shows that all models perform better with simulated perfect RAG (i.e., all steps in the expected workflow are part of the suggestions) as opposed to the standard case where the retriever may miss some required steps. Nevertheless, we see that the improvement is only a maximum of 4%, suggesting that most errors are model errors.

Table 3: Outline results with perfect RAG (TEST set). Best results are in bold and second best are underlined.

Model	RAG	Perfect RAG
Mistral-Nemo-12B-Base (Ours)	**77.5**	**78.8**
GPT-4o	70.5	74.7
LLama-3.3-70B-Instruct	70.6	74.4
Gemini-2.0-Flash	68.7	70.2

6 Error Analysis

6.1 Approach

To perform a more fine-grained evaluation, our approach to error analysis complements the `FlowSim` metric by adding interpretability. It makes the process less time-consuming due to its reusability and extensibility.

We began with a qualitative error analysis of our current production model and based on the patterns found, identified features in the ground truth dataset that appeared to negatively impact model output. These findings were then organized into a binary matrix, where 1 indicates a feature's presence in each ground truth sample and 0, an absence (see Table **??**). We then aligned model scores with the matrix, making it easy to identify the features that could be contributing to underperformance.

Table 4: Sample binary matrix that characterizes each ground truth sample along *n* feature dimensions, with model scores for each sample.

sample id	feat1	feat2	...	feat *n*	model score
0	0	1	...	0	29
1	1	1	...	1	67
2	0	0	...	0	34
⋮	⋮	⋮	...	⋮	⋮
m	0	0	...	1	100

This approach greatly expedites error analysis as model comparison is streamlined. New runs and models can be quickly assessed on particular features without reviewing individual samples. If new model runs have poor performance that does not pattern with current features, we can quickly identify items to review for error analysis and potentially add features to the matrix.

6.2 Findings

We focused our error analysis on the TEST dataset, as it best simulates customer usage of our enterprise application. We identified 24 features across the 108 samples that appeared to impact output quality (see Appendix D). We then subdivided the dataset based on sets of related features to better understand each model's ability to (1) create the outline structure, (2) populate inputs, and (3) handle services specific to our enterprise system:

- **STRUCTURE** contains structural logic features: `FOREACH`, `PARALLEL`, and `TRY/CATCH`.
- **INPUT** contains input-related features: *worknotes/descriptions*, *trigger conditions*, and *multiple conditions*.
- **ENTERPRISE** contains features unique to our enterprise system: *service-level agreement (SLA)*, *service catalog*, and *Glide datetime*.

We compare our fine-tuned SLM and the LLMs that performed best: Gemini-2.0-Flash and GPT-4o. As Table 5 shows, our fine-tuned SLM underperforms the LLMs on flows containing structural logic steps. Closer analysis suggests the SLM frequently misses the dependency steps associated with them (e.g. `FOREACH` is frequently paired with a prior `look_up_records` step, `PARALLEL` should always consist of more than one branch).

Table 5: Model results on three subsets of features, STRUCTURE, INPUT, and ENTERPRISE. Best results are in bold and second best are underlined.

Feature	# Samples	Ours	Gemini	GPT-4o
STRUCTURE				
`FOREACH`	22	64.1	**72.7**	71.7
`PARALLEL`	10	56.0	**65.7**	55.1
`TRY/CATCH`	5	62.2	**68.6**	50.4
Average		63.8	**70.0**	64.5
INPUT				
Worknotes	26	**66.8**	63	64.6
Triggers	37	**70.0**	60.4	63.9
Multiple	23	**60.1**	55.0	54.3
Average		**67.3**	59.8	59.9
ENTERPRISE				
Service Catalog	7	**63.6**	58.9	51.0
SLA	5	**69.6**	58.8	52.2
Glide	5	**68.6**	67.8	62.2
Average		**66.8**	61.6	54.6

The fine-tuned SLM nevertheless consistently outperforms the LLMs on the remaining two subdivisions. The largest margin is on the **ENTERPRISE** set, where the fine-tuned SLM's average `FlowSim` score is 12.16% higher than GPT-4o's. The smallest margin is also on **ENTERPRISE**, where the fine-tuned SLM surpasses Gemini-2.0-Flash by 5.35%. We suspect that learning by demonstrations is more efficient than including complex instructions in a prompt due to the intricacies and specifics of the workflow domain.

Lastly, we observed that workflow steps and conditions were often expressed implicitly in the TEST dataset. The requirement *lookup incident tasks and close them*, for example, entails a `FOREACH` and `update` step that are not stated overtly. Our results indicate that the fine-tuned SLM is far better than the LLMs on such examples, with a `FlowSim` score of 65.1 to Gemini's 57.6 and GPT-4o's 58.5. This indicates the value of labeling as these sort of examples were part of the labeling instructions derived from expectations of how the application would be used.

7 Conclusion

We present a case study for building an enterprise generative AI application called *Flow Generation*, which translates textual user requirements into low-code workflows. While prompting state-of-the-art LLMs yield reasonable results, quality can be significantly improved by fine-tuning an SLM. To complement our quantitative metrics, we performed systematic error analysis that reveals strengths and weaknesses of the fine-tuned SLM and the best-performing LLMs. Future work includes improving our custom metric and addressing the gaps identified by our error analysis approach.

References

[1] Anthropic. 2024. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://www.anthropic.com/news/claude-3-opus-sonnet-haiku. Model card.

[2] Orlando Ayala and Patrice Bechard. 2024. Reducing hallucination in structured outputs via Retrieval-Augmented Generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, Yi Yang, Aida Davani, Avi Sil, and Anoop Kumar (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 228–238. doi:10.18653/v1/2024.naacl-industry.19

[3] Nastaran Bassamzadeh and Chhaya Methani. 2024. A Comparative Study of DSL Code Generation: Fine-Tuning vs. Optimized Retrieval Augmentation. *arXiv preprint arXiv:2407.02742* (2024).

[4] Daniel Bolya, Sean Foley, James Hays, and Judy Hoffman. 2020. Tide: A general toolbox for identifying object detection errors. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part III 16*. Springer, 558–573.

[5] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[6] Pierre Colombo, Telmo Pires, Malik Boudiaf, Rui Melo, Dominic Culver, Etienne Malaboeuf, Gabriel Hautreux, Johanne Charpentier, and Michael Desa. 2024. Saullm-54b & saullm-141b: Scaling up domain adaptation for the legal domain. *Advances in Neural Information Processing Systems* 37 (2024), 129672–129695.

[7] Shihan Dou, Haoxiang Jia, Shenxi Wu, Huiyuan Zheng, Weikang Zhou, Muling Wu, Mingxu Chai, Jessica Fan, Caishuang Huang, Yunbo Tao, et al. 2024. What's Wrong with Your Code Generated by Large Language Models? An Extensive Study. *arXiv preprint arXiv:2407.06153* (2024).

[8] Shengda Fan, Xin Cong, Yuepeng Fu, Zhong Zhang, Shuyan Zhang, Yuanwei Liu, Yesai Wu, Yankai Lin, Zhiyuan Liu, and Maosong Sun. 2024. WorkflowLLM: Enhancing Workflow Orchestration Capability of Large Language Models. *arXiv preprint arXiv:2411.05451* (2024).

[9] Xue-Yong Fu, Md Tahmid Rahman Laskar, Elena Khasanova, Cheng Chen, and Shashi Tn. 2024. Tiny Titans: Can Smaller Large Language Models Punch Above Their Weight in the Real World for Meeting Summarization?. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, Yi Yang, Aida Davani, Avi Sil, and Anoop Kumar (Eds.). Association for Computational Linguistics, Mexico City, Mexico, 387–394. doi:10.18653/v1/2024.naacl-industry.33

[10] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, Qianyu Guo, Meng Wang, et al. 2023. Retrieval-Augmented Generation for Large Language Models: A Survey. *CoRR* (2023).

[11] Gabrielle Gauthier-melancon, Orlando Marquez Ayala, Lindsay Brin, Chris Tyler, Frederic Branchaud-charron, Joseph Marinier, Karine Grande, and Di Le. 2022. Azimuth: Systematic Error Analysis for Text Classification. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, Wanxiang Che and Ekaterina Shutova (Eds.). Association for Computational Linguistics, Abu Dhabi, UAE, 298–310. doi:10.18653/v1/2022.emnlp-demos.30

[12] Gemini, Rohan Anil, Sebastian Borgeaud, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, Katie Millican, et al. 2023. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805* (2023).

[13] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783* (2024).

[14] Suchin Gururangan, Ana Marasović, Swabha Swayamdipta, Kyle Lo, Iz Beltagy, Doug Downey, and Noah A Smith. 2020. Don't stop pretraining: Adapt language models to domains and tasks. *arXiv preprint arXiv:2004.10964* (2020).

[15] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[16] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in neural information processing systems* 33 (2020), 9459–9474.

[17] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[18] Zelong Li, Shuyuan Xu, Kai Mei, Wenyue Hua, Balaji Rama, Om Raheja, Hao Wang, He Zhu, and Yongfeng Zhang. 2024. Autoflow: Automated workflow generation for large language model agents. *arXiv preprint arXiv:2407.12821* (2024).

[19] Bryan McCann, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. 2018. The natural language decathlon: Multitask learning as question answering. *arXiv preprint arXiv:1806.08730* (2018).

[20] Mistral. 2024. Mistral Nemo. https://mistral.ai/news/mistral-nemo Accessed: 2024.

[21] OpenAI. 2025. OpenAI o3-mini System Card. https://cdn.openai.com/o3-mini-system-card-feb10.pdf. Accessed: March 19, 2025.

[22] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2023. Yarn: Efficient context window extension of large language models. *arXiv preprint arXiv:2309.00071* (2023).

[23] Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, et al. 2021. Multitask prompted training enables zero-shot task generalization. *arXiv preprint arXiv:2110.08207* (2021).

[24] Fouad Trad and Ali Chehab. 2024. Prompt Engineering or Fine-Tuning? A Case Study on Phishing Detection with Large Language Models. *Machine Learning and Knowledge Extraction* 6, 1 (2024), 367–384. doi:10.3390/make6010018

[25] Tao Tu, Shekoofeh Azizi, Danny Driess, Mike Schaekermann, Mohamed Amin, Pi-Chuan Chang, Andrew Carroll, Charles Lau, Ryutaro Tanno, Ira Ktena, et al. 2024. Towards generalist biomedical AI. *Nejm Ai* 1, 3 (2024), AIoa2300138.

[26] David Vilar, Jia Xu, Luis Fernando D'Haro, and Hermann Ney. 2006. Error Analysis of Statistical Machine Translation Output. In *Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC'06)*, Nicoletta Calzolari, Khalid Choukri, Aldo Gangemi, Bente Maegaard, Joseph Mariani, Jan Odijk, and Daniel Tapias (Eds.). European Language Resources Association (ELRA), Genoa, Italy. https://aclanthology.org/L06-1244/

[27] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[28] Noam Wies, Yoav Levine, and Amnon Shashua. 2023. Sub-Task Decomposition Enables Learning in Sequence to Sequence Tasks. In *The Eleventh International Conference on Learning Representations*. https://openreview.net/forum?id=BrJATVZDWEH

[29] Michael Wornow, Avanika Narayan, Krista Opsahl-Ong, Quinn McIntyre, Nigam Shah, and Christopher Re. 2024. Automating the enterprise with foundation models. *Proceedings of the VLDB Endowment* 17, 11 (2024), 2805–2812.

[30] Shijie Wu, Ozan Irsoy, Steven Lu, Vadim Dabravolski, Mark Dredze, Sebastian Gehrmann, Prabhanjan Kambadur, David Rosenberg, and Gideon Mann. 2023. Bloomberggpt: A large language model for finance. *arXiv preprint arXiv:2303.17564* (2023).

[31] Zhen Zeng, William Watson, Nicole Cho, Saba Rahimi, Shayleen Reynolds, Tucker Balch, and Manuela Veloso. 2023. FlowMind: automatic workflow generation with LLMs. In *Proceedings of the Fourth ACM International Conference on AI in Finance*. 73–81.

[32] Kaizhong Zhang and Dennis Shasha. 1989. Simple Fast Algorithms for the Editing Distance between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (1989), 1245–1262. doi:10.1137/0218082 arXiv:https://doi.org/10.1137/0218082

A Correlation of Flow Similarity with Human Evaluation

To validate that our custom metric indicated that a generated workflow satisfied the user requirement, we compared it with human scores. We selected 30 random samples from the TEST dataset, generated workflows using our fine-tuned SLM, and asked domain experts in our Quality Engineering department to assign scores from 0 to 10 to each result. A score of zero meant that the generated workflow does not at all represent the user requirement and a score of ten meant a perfect generated workflow.

	Outline	**Outline and inputs**
Pearson	0.78 (4.4e-7)	0.78 (2.9e-7)
Spearman	0.83 (1.5e-8)	0.76 (9.1e-7)

Table 6: Correlation between human evaluation scores and `FlowSim` metrics. P-value is shown in parentheses.

We performed correlation tests for the outline only as well as for outline and inputs, computing both Pearson and Spearman correlation coefficients along with their associated p-values. As Table 6

shows, the correlation coefficients are higher than 75% in all cases and the p-values are very small, confirming that the correlation numbers are statistically significant. This gives us confidence that the `FlowSim` metric is appropriate for our use case.

B Example of representing a workflow as a tree

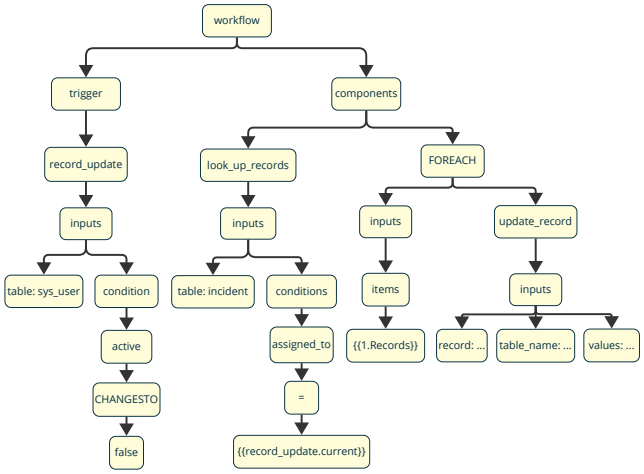

Figure 6: Workflow in Figure 1 represented as a tree.

The tree in Figure 6 is the representation of the workflow that can be generated from the requirement *Every time a user becomes inactive, find all incidents where the user is the assignee. Assign them to their manager*. This workflow is also shown in Figure 1.

C Flow Similarity Results with Structure Validation

As Figure 5 shows, many LLMs generate workflows that have obvious structure errors. For instance, on the TEST set, 17.3% of the 108 workflows generated by GPT-4o, and 12% of those generated by Llama-3.3-70B-Instruct have such errors.

While some of these errors can be corrected via post-processing, it requires additional engineering effort. The best model would be the one that minimizes this percentage. To take into account this failure mode, we computed another set of metrics where all these "broken" workflows received an Outline score of zero, as these are unusable. Consequently, the score for Outline and inputs also becomes zero.

Figure 7 shows the `FlowSim` results obtained for all models after this structure validation is applied. While our fine-tuned SLM still gives significant improvements over prompting LLMs, we observe that GPT-4o suffers a substantial degradation compared to Gemini-2.0-Flash. Excluding structure errors, both of these LLMs performed virtually the same.

D TEST Dataset Features

A total of 24 features were identified during the qualitative error analysis. They are presented in Table 7, grouped by theme. Some elements were not included in any training data or were rare in the training data. These, we grouped to address training gaps in the future. Others, including flow logics, are common in training data and have regular patterning of model behavior. They are useful groupings to check model performance at a granular level without reviewing individual samples. Finally, some of the TEST samples are adversarial and are particularly useful for assessing default behavior and seeing if anything 'breaks' the model.

Received 30 May 2025

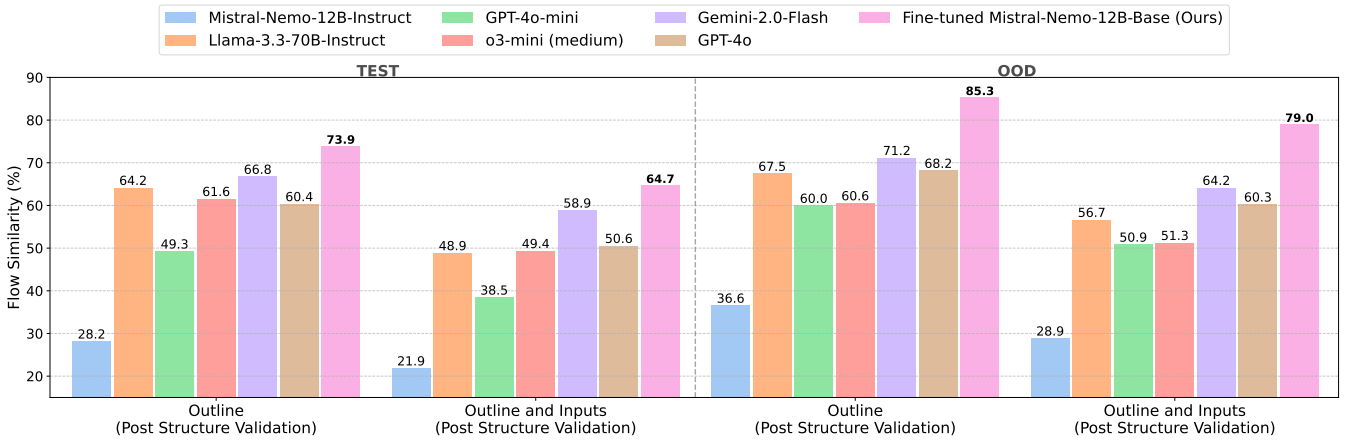

Figure 7: `FlowSim` results obtained on the TEST and OOD sets for Outline and Outline with inputs, after structure validation is applied.

Table 7: Features identified during the error analysis process, grouped by type.

Features by Type
NOT INCLUDED IN TRAINING
Triggers: null, service catalog
Actions: get catalog variables, ask for approval
Inputs: glide query, dynamic ME
UNCOMMON IN TRAINING DATA
Triggers: SLA, conditionals, weekly
Inputs: requestor-related
FLOW LOGICS
`DOUNTIL`, `PARALLEL`, `FOREACH`, `TRY-CATCH`
COMMON
Actions: MS Teams / Slack message, notifications / emails, log, worknotes / descriptions
Inputs: Complex inputs
DIFFICULT / EDGE CASES
Out of scope, ambiguous, implicit actions, misleading input, incorrect / incomplete