Structuring the Unstructured: A Multi-Agent LLM Framework for Transforming Ambiguous SOPs into Code

Sachin Kumar Giroh*, **Pushpendu Ghosh, Aryan Jain, Harshal Paunikar, Anish Nediyanchath, Aditi Rastogi, Promod Yenigalla**

RBS Tech Sciences, Amazon

Abstract

This paper introduces, a three-stage multi agent LLM framework designed to transform unstructured and ambiguous Standard Operating Procedure (SOP) into a structured plan and an executable code template. Unstructured SOPs—common across industries such as finance, retail, and logistics—frequently suffer from ambiguity, missing information, and inconsistency, all of which hinder automation. We address this through: (1) a Clarifier module that disambiguate the SOP using large language models, internal knowledge base (RAG) and human-in-the-loop , (2) a Planner that converts refined natural language instructions into a structured plan of hierarchical task flows through function (API) tagging, conditional branches and human-in-the-loop checkpoints, and (3) an Implementor that generates executable code fragments or pseudocode templates. We evaluate our solution on real-world SOPs and synthetic variants, demonstrating an 88.4% end-to-end accuracy and a significant reduction in inconsistency compared to leading LLM baselines. Ablation studies highlight the necessity of each component, with performance dropping notably when modules are removed. Our findings show that structured multi-agent pipelines like ours can meaningfully improve consistency, reduce manual effort, and accelerate automation at scale.

1 Introduction

Standard Operating Procedures (SOPs) play an important role in helping organizations maintain consistency and reliability in their day-to-day operations. These documents describe how tasks should be carried out and are often used to guide employees across departments. However, in many cases, SOPs are written as free-form text without following a consistent structure. We denote such type of SOPs as Unstructured SOPs (uSOPs), they typically include vague language, missing information, and informal decision points. For example, references like "the system" or phrases such as "if needed" can be unclear. Some steps may not list the required inputs or expected outcomes. In other cases, human judgment is expected at key points, which is hard to replicate consistently. These issues make it difficult to understand the exact steps involved in a procedure, and they also hinder efforts to automate routine tasks based on these documents. In large organizations, the impact of such inefficiencies becomes more noticeable. Many employees spend a significant amount of time each week carrying out tasks defined in SOPs.

To address this, we introduce <SYNTACT>: System for Natural-language Task Abstraction and Code Translation, a three-stage AI framework: Clarifier, Planner, and Implementor. First, the Clarifier module resolves ambiguous references, missing details, and informal logic. Next, Planner converts the clarified SOP into a structured plan consisting of hierarchical task flows with typed inputs, conditional branches, and human-in-the-loop checks. Finally, the Implementor module generates executable Python, API calls, and reasoning prompts.

2 Related Works

Early efforts to extract process knowledge from SOPs used handcrafted linguistic rules like T-regex over dependency parses. These delivered precise matches for familiar patterns but required heavy rule engineering and struggled with syntactic variation (e.g., Quishpi et al., 2020; Bellan et al., 2023). Supervised learning later reframed the task as sentence or token labeling, using hierarchical neural networks to identify actions, actors, and control-flow cues (Qian et al., 2020). With large language models, the focus shifted from labeling to directly *generating* structured outputs. Prompted GPT-style models, when guided by exemplar demonstrations

Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: Industry Track, pages 2367–2376

and strict schemas, have outperformed traditional classifiers in extracting procedural structure (Neuberger et al., 2024). Hybrid pipelines combine classical NLP segmentation with GPT-4 to generate BPMN-style JSON outputs (Licardo et al., 2024), and multi-agent systems like NL2ProcessOps orchestrate LLM stages to emit executable workflow code (Monti et al., 2024). These LLM-based methods leverage broad contextual knowledge but must address hallucination via schema-constrained prompting and post-generation checks. More recent frameworks enhance reliability by integrating LLMs with explicit planning and agentic control. SOPSTRUCT constructs a DAG representation of the SOP and validates logical soundness via a PDDL planner and GPT-4-based semantic checks (Garg et al., 2025). FLOW-OF-ACTION embeds SOPs into a multi-agent debugging workflow to mitigate hallucination (Pei et al., 2025), while AGENT-S demonstrates that specialized GPT-4-o-mini agents can execute end-to-end customer-service SOPs with high reliability (Kulkarni, 2025).

3 Methodology

We formulate SOP-to-workflow conversion as a three-stage pipeline that, given an unstructured SOP description S_{raw} and an available tool set $\mathcal{T}$, deterministically produces a Program-Design-Language (PDL) workflow W. Let $\mathcal{H}$ denote human-in-the-loop inputs and $\mathcal{A}$ denote auxiliary agents (e.g., LLMs, retrieval modules). We instantiate three modules $\mathcal{M}_c$, $\mathcal{M}_s$, $\mathcal{M}_i$:

(1) **Clarifier $\mathcal{M}_c$:** Produces a context C of critical question–answer pairs to resolve ambiguities (e.g., missing inputs/outputs, tool usage, step granularity):

$$C = \mathcal{M}_c\big(S_{\text{raw}}, \mathcal{T} \mid \mathcal{H}, \mathcal{A}\big) \quad (1)$$

(2) **Planner $\mathcal{M}_s$:** Uses C to converts S_{raw} to a semi-structured pseudocode S_{struc} conforming to grammar $\mathcal{G}$ [see Appendix A], ensuring parseability by parser P:

$$S_{\text{struc}} = \mathcal{M}_s\big(S_{\text{raw}}, \mathcal{T}, C\big) \in \mathcal{G} \quad (2)$$

(3) **Implementor $\mathcal{M}_i$:** Parses S_{struc} via P and emits an executable PDL workflow W, where each node corresponds to a runnable tool invocation or code fragment:

$$W = \mathcal{M}_i\big(S_{\text{struc}}, \mathcal{T} \mid P\big) \quad (3)$$

The end-to-end pipeline (Figure 1) is thus:

$$S_{\text{raw}} \xrightarrow{\mathcal{M}_c} C \xrightarrow{\mathcal{M}_s} S_{\text{struc}} \xrightarrow{\mathcal{M}_i} W \quad (4)$$

Finally, the generated workflow W is indexed in the tool repository as a heterogeneous, non-atomic component, enabling search and reuse via downstream orchestrators or chat interfaces.

3.1 Clarifier Module ($\mathcal{M}_c$)

Unstructured SOPs (S_{raw}) often omit key details—inputs, outputs, or tool-specific constraints—either due to author oversight or assumptions of user intuition. Their free-form style, implicit steps, and reliance on human interpretation make them ambiguous and non-executable as-is. Without first resolving these gaps, automated conversion into structured workflows is infeasible.

The Clarifier Module $\mathcal{M}_c$ (Eq. 1) addresses this challenge by producing a context C of critical question–answer (QA) pairs that resolve underspecification in the original SOP. It comprises three sub-agents working in sequence:

(i) **Question Generation Agent ($\mathcal{M}_c^{(gen)}$):** We execute the language model k times with a fixed configuration π, generating multiple candidate clarification questions independently. This sampling strategy increases coverage and robustness, accounting for the stochasticity of generative models and improves consistency in the generated questions.

(ii) **Question Filtering Agent ($\mathcal{M}_c^{(fil)}$):** The aggregated question set is deduplicated and clustered. First, cosine similarity over sentence embeddings is used to identify near-duplicates (above a predefined threshold(γ)). Clustering is performed using the union-find algorithm. For each cluster of n questions with embeddings $\mathbf{e}_1, \ldots, \mathbf{e}_n$, a representative question q_i is selected based on one of two heuristics: $\text{CentroidScore}(q_i) = \frac{1}{n-1}\sum_{j \neq i} \cos(\mathbf{e}_i, \mathbf{e}_j)$ or by selecting the longest-form question. Each remaining question is then assigned a *relevance score* with respect to the SOP-to-workflow conversion task. Only questions exceeding a threshold(κ) are retained and passed to the answering agent.

(iii) **Answering Agent ($\mathcal{M}_c^{(ans)}$):** The filtered questions are passed to a Retrieval-Augmented Generation (RAG) system that

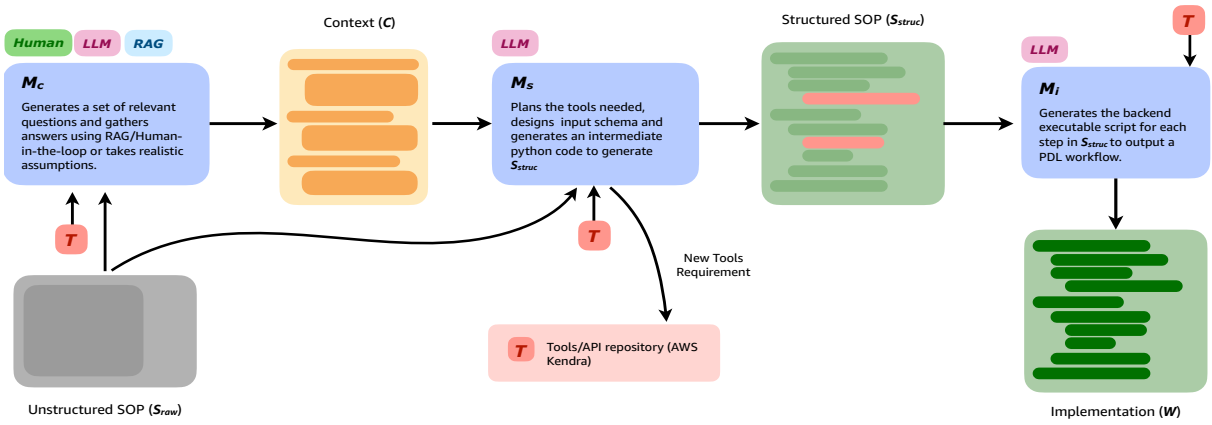

Figure 1: The end-to-end pipeline consists of three major modules: the *Clarifier* ($\mathcal{M}_c$), the *Planner* ($\mathcal{M}_s$), and the *Implementor* ($\mathcal{M}_i$). Each of them has access to tools repository T.

queries internal tool descriptions and documentation. If an answer cannot be retrieved, answering agent is used to either (a) make a justifiable assumption, or (b) flag the question for human input $\mathcal{H}$, ensuring coverage of all critical gaps.

The output context, $C = \mathcal{M}_c^{(\text{ans})}\left(\mathcal{M}_c^{(\text{fil})}\left(\mathcal{M}_c^{(\text{gen})}(S_{\text{raw}}, \mathcal{T})\right), \mathcal{T}, \mathcal{H}, \mathcal{A}\right)$, produced by the $\mathcal{M}_c$ is crucial for the next stage: it provides the necessary disambiguation and semantic grounding required by the $\mathcal{M}_s$, which uses it to convert the raw SOP into a structured intermediate representation S_{struc}.

3.2 Planner Module ($\mathcal{M}_s$)

The *Planner* takes an unstructured SOP S_{raw} and produces a structured pseudocode using 4 phases: (1) *API Planning*, (2) *Input Schema Planning*, (3) *Code Skeleton Planning* and (4) *Code Conversion Agent*

1. API Planning We first identify a candidate list of LLM-invoked tasks $\Lambda = \{a_1, \ldots, a_n\}$ from S_{raw}. For each task a_i, the module queries the tool repository $\mathcal{T} = \{\tau_1, \ldots, \tau_m\}$ to find any existing implementation:

$$\mathcal{M}(\Lambda) \equiv \{a_i \xrightarrow{\text{match}} (\tau_j | \Delta_i) \text{ or } \perp (\text{no match})\}$$

If τ_j partially covers a_i, we extract the missing subtasks Δ_i and re-prompt the LLM to check if any other tool completes the pending task. Tasks with no match ($\perp$) are tested for direct LLM execution; those that fail are flagged for manual API development.

2. Input Schema Planning This phase follows a two-step process. First, the LLM—using a task-and-critic agentic system—is prompted to extract all base-level variables $\{p_i\}$ that are not derived from any tool or transformation. These represent the core inputs needed to execute the SOP. Second, each variable is classified as either *mandatory* or *optional*, recognizing that such classifications may vary across SOP instances. For optional parameters, the LLM proposes realistic default values to ensure completeness.

The outcome is a minimal and well-typed parameter set: $\mathcal{P} = \{p_k\}$ required to invoke the SOP.

3. Code Skeleton Planning We select Python for skeleton generation, exploiting its ubiquity in LLM pretraining and strong model proficiency (Chen et al., 2021). The Code Planning agent ingests $(S_{\text{raw}}, C, \mathcal{M}(\Lambda), \mathcal{P})$ and prompts an LLM to produce a Python script that invokes each tool $t \in \mathcal{T}$ according to its API signature. If a required capability—such as ad hoc reasoning or image-based inference—is not expressible via static code, the agent instead emits calls to two fallback primitives, `ask_llm(task, context)` and `ask_vqa(task, image_url)`.

A verification sub-agent then iteratively reviews the generated code to ensure that every step in S_{raw} is represented, that all tool calls conform to their documented APIs, and that no tool τ_j is incorrectly assumed to perform subtasks Δ_i outside its specification.

Despite this review and use of best LLM models, many tools return nested, heterogeneous objects (e.g. a `Ticket` with `List[Attachment]` items embedding `Mail` or `Image` objects), which induce

hallucinations and type mismatches. This limitation motivates our subsequent decomposition into a type-annotated PDL grammar, providing strict enforcement of data structures and call semantics.

4. Code Conversion Agent We utilise the Code Conversion Agent to transform the Python code into structured pseudocode S_{struc} that strictly adheres to grammar $\mathcal{G}$. This conversion removes the need for strict type consistency (e.g., explicit casts, null checks, as these can be correctly handled in a later stage) and isolates each action into atomic steps—either a single API call or a one-line statement (Python execution step, Miscellaneous step or LLM block step). By construction, S_{struc} is fully parseable, and each line in the structured pseudocode is free of complex control flow, greatly simplifying subsequent workflow generation.

3.3 Implementor Module ($\mathcal{M}_i$)

Execution Context: While translating the structured SOP S_{struc} into a PDL, we maintain an *execution context* ϵ_i that accumulates all variables (with their types) defined up to line i. Mathematically:

$$\epsilon_i = \bigcup_{k=1}^{i} \{v_k\}, \quad v_k = (\phi(s_k, n_k), \delta_k), \quad \epsilon_0 = \varnothing,$$

where each entry v_k, comprises of a unique identifier $\phi(s_k, n_k)$ (built using JSON-Path style from the PDL statement index s_k and the local variable name n_k), its corresponding data class type δ_k.

Equivalently, one can define ϵ_i recursively: $\epsilon_i = \epsilon_{i-1} \cup \{v_i\}, \quad \epsilon_0 = \varnothing$, so that the execution context grows monotonically as the line number i increases. This enriched context ϵ_i enables precise type-aware reference resolution in the generated PDL.

3.3.1 Implementation Pipeline Overview

Building on the execution context $\{\epsilon_i\}$, the Implementor takes the structured SOP S_{struc} and parses it using the grammar and does line-by-line processing of each line in S_{struc}.

Parsing and Block Extraction A deterministic parser P scans S_{struc} to recognize looping constructs (`FOR`), conditional branches (`IF-TRUE/IF-FALSE`), flat execution lines, and wildcard commands (e.g. `CONTINUE`, `BREAK`, `RETURN`).

Execution statement Translation Each line L_i tagged as execution line is classified into one of three types and translated using only the prior context ϵ_{i-1}:

(1) *API Call:* Since $\mathcal{M}_s$ specifies the tool, identifying the API $\tau_j \in \mathcal{T}$ for line L_i is straightforward. The main challenge lies in generating the correct arguments to invoke τ_j. Let τ_j take n_j arguments with expected input types $\Delta_j = (\delta_{j,1}, \delta_{j,2}, \ldots, \delta_{j,n_j})$. The LLM must construct the argument vector $\Lambda_i = (\texttt{arg}_{i,1}, \ldots, \texttt{arg}_{i,n_j})$, where each $\texttt{arg}_{i,k} = f_{i,k}(\epsilon_{i-1} \mid \tau_j) \in \delta_{j,k}$ for $1 \leq k \leq n_j$, ensuring that $\Lambda_i \in \Delta_j$.

(2) *Inline Code:* A Python snippet $\rho_i(\epsilon_{i-1})$ is generated to perform the described transformation, and its output type τ_i is inferred from the expression.

(3) *LLM Step:* The Auto-Prompting submodule, `Auto Prompt`, composes a dynamic prompt with guardrails and an output schema based on the step description. A relevant variable from ϵ_{i-1} is selected as the action object, serving as the input argument during inference.

In all cases, the step's resulting output v_i is saved in ϵ_i.

Control Structures In `FOR` blocks, the loop's iterable object (either a `List` or `Set`) is detected, and the loop variable is added to ϵ_i with a special tag `<iterator_d>` where `d` is an integer representing the depth of the loop, alongside the identifier of the iterable variable.

For an `IF` block, the module checks ϵ_{i-1} for an existing Boolean variable to serve as the branch condition; if none is found, it synthesizes a predicate $\psi = g_i(\epsilon_{i-1})$ and records it in ϵ_i. This Boolean—whether reused or newly generated—then drives the true/false branch selection.

3.3.2 Deterministic PDL Generation

Once all lines and blocks are processed, the Implementor uses a deterministic logic to generate a PDL workflow. This PDL forms the executable backbone for downstream code interpretation.

4 Experiments

4.1 Dataset

We use the SOP bench dataset (Nandi et al., 2025) which contains SOPs for over 1,800 tasks across 10

industrial domains, each with APIs, tool interfaces, and human-validated test cases. The framework used for the generation of these SOPs is designed to replicate real-world scenarios with an average of 181 tasks, 11 tools and 132 tokens per SOP, and therefore serves as a robust dataset for evaluating our SYNTACT solution.

To further enrich the data, we include synthetic variations of some of the real-world SOPs through random sentence removal and shortening via LLMs. We limit the number of lines removed to a maximum of 5% of the original length, ensuring that the new SOP remains meaningful.

To quantify the ambiguity introduced in a SOP, we come up with a noise factor which refers to the proportion of SOP steps that are rendered ambiguous, incomplete, or entirely removed by an LLM. In this paper, Claude Sonnet 3.5 was tasked to add noise, i.e. eliminate steps or shorten them based on criticality, to a randomly selected subset of SOP steps. The noise factor (n) is then defined as k/|S| where |S| denotes the total number of steps in the SOP.

Example of corruption transformation:

- **Original Step**: Create a SIM Ticket describing the issue for the ATP code "Retail", assignee from the mail, set the severity to "High".
- **Modified Step**: Create a SIM Ticket describing the issue.

4.2 Benchmarking and Ablation

We evaluate the full three-stage pipeline against five alternative configurations:

1. **Vanilla LLM:** This configuration involves one-shot prompting of state-of-the-art LLMs (Claude Haiku 3.5, Sonnet 3.5 and Sonnet 3.7) to directly translate the raw SOP S_{raw} into: (a) executable Python code, which is then evaluated; (b) an intermediate representation W, which is deterministically converted into PDL; and (c) PDL, generated in a single step using the PDL grammar. In this setup, we adopt a task-critic-improver approach, requiring three calls to complete the task.

2. **Reasoning-LLM:** In this configuration, Claude Sonnet 3.7 is prompted using a dynamic chain-of-thought strategy that interleaves planning and execution. The reasoning-token budget is set to $k \times |I|$, enabling more extensive multi-step reasoning.

3. **<SYNTACT>$\setminus \mathcal{M}_c$:** Ablation of the Clarifier stage; the Planner directly processes S_{raw}, bypassing the context creation step.

4. **<SYNTACT>$\setminus \mathcal{M}_s$:** Ablation of the Planner module; Clarifier outputs, along with S_{raw}, are fed into a one-shot prompt, which converts them into pseudocode following the grammar G.

5. **<SYNTACT>$\setminus \mathcal{M}_i$:** Ablation of the Implementor module; the pipeline is truncated after the structured pseudocode step, and an LLM is prompted to convert the structured pseudocode into PDL without invoking the Implementor.

4.3 Ground-Truth Creation Methodology

For any task T and input I, we run the state-of-the-art model, Sonnet 3.7, k times, using a reasoning token limit of $4 \times |I_t|$, with temperature 1.0. We found that any agent can emit either of these 2 types of output:

(1) Single unique answer: We embed each of the k outputs using Cohere embeddings and perform clustering. Clusters with frequency $\geq \alpha k$, for some $0 < \alpha < 1$, are retained. From each selected cluster, we choose the answer closest to the cluster centroid as the ground-truth.

(2) Multi-answer selection: Let each model run produce a set $o_i = \{a_{i,1}, a_{i,2}, \ldots, a_{i,|o_i|}\}$ for $1 \leq i \leq k$. We define the candidate universe $\mathcal{A} = \bigcup_{i=1}^{k} o_i$, and for each element $a \in \mathcal{A}$, compute its support $f(a) = |\{i : a \in o_i\}|$. The ground-truth set G is then defined as $G = \{a \in \mathcal{A} : f(a) \geq \alpha k\}$, i.e., all options that appear in at least α fraction of the runs. If a fixed-size output of m elements is required, we rank candidates in G by support $f(a)$ (breaking ties using average embedding-distance to the runs where a appears) and select the top m candidates.

End-to-End PDL ground truth: For the end-to-end PDL ground truth creation, we manually verified the 4 PDL and corrected them if needed to get the perfect PDL considered as ground truths. Additionally we created T testcases designed to cover every line and execution path of the workflow.

Table 1: End-to-End Performance comparison

Method	Accuracy	Time (s)
Haiku 3.5[1]	0.065 ± 0.051	33.1 ± 7.3
Sonnet 3.5[1]	0.178 ± 0.088	54.6 ± 9.8
Sonnet 3.7[1]	0.161 ± 0.090	78.5 ± 13
Sonnet 3.7 (Reasoning)[1]	0.208 ± 0.102	212 ± 46
<SYNTACT> $\setminus \mathcal{M}_c$	0.801 ± 0.092	715 ± 120
<SYNTACT> $\setminus \mathcal{M}_s$	0.774 ± 0.117	601 ± 98
<SYNTACT> $\setminus \mathcal{M}_i$	0.481 ± 0.136	530 ± 96
<SYNTACT> [Proposed]	$\mathbf{0.884 \pm 0.096}$	$\mathbf{929 \pm 148}$

[1]Experimental setup for the vanilla LLM tests follows the task-critic-improver approach, involving 3 LLM calls.

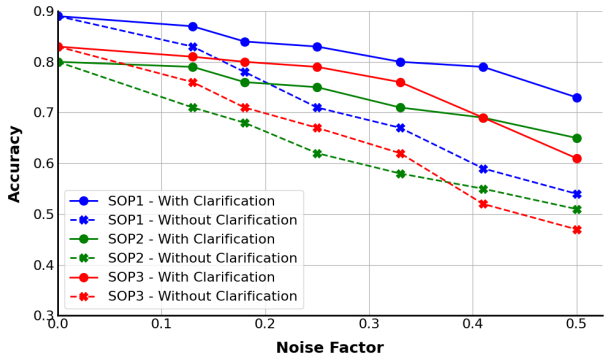

Figure 2: Impact of noise on accuracy and the role of the clarifier module, in enhancing performance under higher noise SOPs.

5 Results and Discussions

Our experiments demonstrate that the proposed SYNTACT pipeline substantially outperforms all baseline and ablation configurations in end-to-end workflow synthesis. As shown in Table 1, SYNTACT achieves the highest test accuracy (0.884) with an inconsistency score of 0.096, clearly outperforming vanilla prompting strategies that directly invoke LLMs like Haiku 3.5, or Sonnet 3.5/3.7 (with or without reasoning). While Sonnet 3.7 with reasoning (dynamic chain-of-thought) improves over its non-reasoning variant (0.208 vs. 0.161), it remains far below SYNTACT in both accuracy and robustness. This shows that the conversion of real SOPs is a complex task and cannot be reliably accomplished using a single prompt or the task-critic-improver setup.

Among the ablations, removing the *Implementor* module (SYNTACT $\setminus \mathcal{M}_i$) leads to the most severe performance drop ($0.884 \rightarrow 0.481$), confirming that direct LLM-based PDL synthesis from structured pseudocode is error-prone without programmatic grounding like type casting, execution context management, etc. This highlights the critical role of planning and building the code incrementally in a complex and ambiguous problem space. Removing the Clarifier (SYNTACT $\setminus \mathcal{M}_c$) or Planner (SYNTACT $\setminus \mathcal{M}_s$) also leads to substantial accuracy degradation, underscoring the necessity of both interactive disambiguation and intermediate planning steps. The robustness of the Clarifier module is further validated through controlled noise experiments (Figure 2). When injected noise corrupts a perfect SOP, systems without clarification degrade significantly—62% accuracy at a noise rate of 0.25. In contrast, our full pipeline with the Clarifier module maintains much higher robustness, reaching 76% at the same error level. Task-level evaluations (Table 2) reinforce the rationale behind selective model usage. Although Sonnet 3.7 with reasoning consistently achieves the highest precision and recall across nearly all subtasks, it is computationally prohibitive for routine use. Instead, we find that Sonnet 3.5 without reasoning offers a favorable trade-off—achieving near-optimal performance (e.g., API Planning: 0.95/0.92) on complex steps while being significantly faster. For simpler or high-accuracy deterministic tasks—such as For loop implementation, If-condition filling, or Execution LLM filling—Haiku suffice with accuracies nearing 0.93–0.99. Notably, Clarifier tasks (which are highly ambiguous and open-ended), like question generation and filtering, benefit most from reasoning-based models, with Sonnet 3.7 with reasoning boosting precision and recall. We thereby choose different LLM models for each subcomponent based on the results, balancing time and performance. Overall, these findings validate our modular approach and adaptive model selection strategy, which balances accuracy and cost to maximize effectiveness across both clean and noisy SOP scenarios.

Conclusion

This paper introduced <SYNTACT>, a novel three-stage framework that transforms ambiguous SOP narratives into executable code templates. Through our evaluation on real-world and synthetic SOPs, we demonstrated that our modular approach—comprising the Clarifier for disambiguation, the Planner for organizing instructions, and the Implementor for generating code. Beyond immediate technical achievements, <SYNTACT> addresses a crucial organizational challenge by enabling more consistent execution across teams, re-

Table 2: Performance scores (Precision/Accuracy and Recall/Consistency) of various LLMs- Haiku, and different versions of Sonnet—on key subtasks grouped under the Clarifier ($\mathcal{M}_c$), Planner ($\mathcal{M}_s$), and Implementor ($\mathcal{M}_i$) modules. Sonnet 3.7$^{\#}$ indicates Sonnet 3.7 with reasoning tokens of up to $4\times$ input tokens.

Task	**Type**	**Haiku**		**Sonnet 3.5**		**Sonnet 3.7**		**Sonnet 3.7$^{\#}$**	
		M_1	M_2	M_1	M_2	M_1	M_2	M_1	M_2
Clarifier Module ($\mathcal{M}_c$)									
Question Generation	Multi	0.53	0.4	0.85	0.77	0.81	0.78	0.89	0.85
Question Filtering	Multi	0.60	0.56	0.70	0.51	0.69	0.63	0.70	0.73
Answer Generation-RAG	Single	-	-	-	-	-	-	-	-
Answer Generation	Single	0.82	0.07	0.91	0.09	0.90	0.09	0.92	0.10
Planner Module ($\mathcal{M}_s$)									
API Planning	Multi	0.88	0.90	0.95	0.92	0.95	0.93	0.94	0.98
Input Schema Planning	Single	0.80	0.12	0.85	0.10	0.86	0.12	0.81	0.13
Code Skeleton Planning	Single	0.72	0.10	0.85	0.10	0.85	0.10	0.87	0.11
Code Conversion	Single	0.89	0.05	0.94	0.05	0.93	0.04	0.93	0.05
Implementor Module ($\mathcal{M}_i$)									
Execution API	Single	0.84	0.08	0.91	0.07	0.92	0.08	0.89	0.11
Execution Code	Single	0.74	0.12	0.85	0.15	0.86	0.14	0.87	0.16
Execution LLM[2]	Single	0.93	0.05	0.96	0.05	0.96	0.06	0.96	0.08
For Loop	Single	0.97	0.04	0.98	0.04	0.99	0.04	0.99	0.05
If Condition Generation	Single	0.94	0.04	0.97	0.04	0.97	0.04	0.96	0.05

Note: For Multi-answer tasks, M_1 is Precision and M_2 is Recall, while for single-answer tasks, M_1 is Accuracy and M_2 is Inconsistency.
[2]For Execution LLM, the metrics is only for generating the right task description and choosing the correct JSON path for the inference input variable. It does not include the metrics for auto prompting.

ducing barriers to automation, and providing a foundation for scaling process automation initiatives across diverse domains.

Limitations

While <SYNTACT> demonstrates strong performance in SOP-to-code translation, several limitations remain. First, we observe a decline in performance as the length of input SOPs increases, particularly beyond 100 lines. This is primarily due to context compression and increased ambiguity propagation. Moreover, LLMs are typically trained on code generation datasets that contain a majority of examples with fewer than 50 lines of code (LOC), limiting their generalization to long-form generation. As a potential solution, future work will explore hierarchical SOP decomposition, where long procedures are first segmented into modular, independently processable sub-SOPs that are later reassembled into a coherent plan. Second, the current system is limited to purely textual SOPs and cannot yet handle multimodal instructions that include images, weblinks, tables, audio, or video—despite their prevalence in real-world industrial SOPs. Finally, although <SYNTACT> achieves high accuracy, it requires multiple LLM invocations across its multi-agent pipeline, resulting in high computational cost and latency. Reducing this overhead through model distillation, caching, or speculative execution is a promising direction for future work.

Acknowledgements

We thank the authors of *SOP-Bench: Complex Industrial SOPs for Evaluating LLM Agents* (Nandi et al., 2025) for their benchmark and valuable insights, which supported the evaluation and improvement of our proposed methodology.

References

Patrizio Bellan, Mauro Dragoni, Chiara Ghidini, Han van der Aa, and Simone Paolo Ponzetto. 2023. Process extraction from text: Benchmarking the state of the art and paving the way for future challenges. *Preprint*, arXiv:2110.03754.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, and 39 others. 2021. Evaluating large language models trained on code. *Preprint*, arXiv:2107.03374.

Deepeka Garg, Sihan Zeng, Sumitra Ganesh, and Leo Ardon. 2025. Generating structured plan rep-

resentation of procedures with llms. *Preprint*, arXiv:2504.00029.

Mandar Kulkarni. 2025. Agent-s: Llm agentic workflow to automate standard operating procedures. *Preprint*, arXiv:2503.15520.

Josip Tomo Licardo, Nikola Tanković, and Darko Etinger. 2024. A method for extracting bpmn models from textual descriptions using natural language processing. *Procedia Computer Science*, 239:483–490. CENTERIS – International Conference on ENTERprise Information Systems / ProjMAN - International Conference on Project MANagement / HCist - International Conference on Health and Social Care Information Systems and Technologies 2023.

Flavia Monti, Francesco Leotta, Juergen Mangler, Massimo Mecella, and Stefanie Rinderle-Ma. 2024. NL2ProcessOps: Towards LLM-guided Code Generation for Process Execution. In *Business Process Management Forum - BPM 2024 Forum, Proceedings*, volume 526 of *Lecture Notes in Business Information Processing*, pages 127–143. Springer.

Subhrangshu Nandi, Arghya Datta, Nikhil Vichare, Indranil Bhattacharya, Huzefa Raja, Jing Xu, Shayan Ray, Giuseppe Carenini, Abhi Srivastava, Aaron Chan, Man Ho Woo, Amar Kandola, Brandon Theresa, and Francesco Carbone. 2025. Sop-bench: Complex industrial sops for evaluating llm agents. *Preprint*, arXiv:2506.08119.

Julian Neuberger, Lars Ackermann, Han van der Aa, and Stefan Jablonski. 2024. A universal prompting strategy for extracting process model information from natural language text using large language models. *Preprint*, arXiv:2407.18540.

Changhua Pei, Zexin Wang, Fengrui Liu, Zeyan Li, Yang Liu, Xiao He, Rong Kang, Tieying Zhang, Jianjun Chen, Jianhui Li, Gaogang Xie, and Dan Pei. 2025. Flow-of-action: Sop enhanced llm-based multi-agent system for root cause analysis. *Preprint*, arXiv:2502.08224.

Chen Qian, Lijie Wen, Akhil Kumar, Leilei Lin, Li Lin, Zan Zong, Shuang Li, and Jianmin Wang. 2020. An approach for process model extraction by multi-grained text classification. *Preprint*, arXiv:1906.02127.

Luis Quishpi, Josep Carmona, and Lluís Padró. 2020. Extracting annotations from textual descriptions of processes. In Dirk Fahland, Chiara Ghidini, Jörg Becker, and Marlon Dumas, editors, *Business Process Management. BPM 2020*, volume 12168 of *Lecture Notes in Computer Science*, pages 184–201. Springer, Cham.

A Grammar Definition ($\mathcal{G}$)

The following grammar ($\mathcal{G}$) is defined using **Backus–Naur Form (BNF)** and specifies the syntax of our structured pseudocode language. This formal grammar captures common procedural constructs such as execution steps, conditionals, loops, list operations, input gathering, and return statements. Each script is composed of a sequence of steps, and indentation is used to express nested control structures. The grammar ensures that the pseudocode remains both human-readable and programmatically parsable, supporting reliable downstream interpretation and translation.

```
<pseudocode>      ::= <step> ("\n" <step>)*

<step>            ::= <execution_step>
                    | <if_step>
                    | <for_step>
                    | <list_creation>
                    | <list_addition>
                    | <input_step>
                    | <return_step>

<execution_step>  ::= "EXECUTION: " <description>

<if_step>         ::= "IF: "<condition>"\n"
                      <indent> "TRUE:\n" <
                           indented_steps> "\n"
                      <indent> "FALSE:\n" <
                           indented_steps>

<for_step>        ::= "FOR: " <iterator> "\n" <
    indented_steps>

<list_creation>   ::= "CREATE_EMPTY_LIST: " <list> =
    List

<list_addition>   ::= "ADD_TO_LIST: " <list>.add(<
    object>)

<input_step>      ::= "INPUT: " <var> = <input_desc>

<return_step>     ::= "RETURN: " (<description> | "")

<indented_steps>  ::= <indent> (<step>|"pass") ("\n"<
    indent><step>)*

<indent>          ::= "\t" | "    "

<description>     ::= <text>
<condition>       ::= <text>
<iterator>        ::= <text>
<input_desc>      ::= <text>
<list>            ::= <identifier>
<object>          ::= <text>
<var>             ::= <identifier>

<identifier>      ::= <letter> (<letter>|<digit>|"_")*
<text>            ::= <printable>+
```

Table 3: Pseudocode Grammar for Structured SOP Format

B Evaluation Protocol

We evaluate generated workflows at two levels:

1. **PDL-Level Metrics.** Let T be the number of curated testcases, and let $a_i^{(r)} \in \{0, 1\}$ denote the outcome of testcase i in run r, where k is the total number of runs. The *Test*

Run Accuracy for run r is defined as:

$$\alpha^{(r)} = \frac{1}{T} \sum_{i=1}^{T} a_i^{(r)}.$$

Across the k runs, we calculate the *Average Accuracy* and its standard deviation as:

$$\bar{\alpha} = \frac{1}{k} \sum_{r=1}^{k} \alpha^{(r)},$$
$$\text{std}(\alpha^{(1)}, \ldots, \alpha^{(k)}) = \sqrt{\frac{1}{k} \sum_{r=1}^{k} (\alpha^{(r)} - \bar{\alpha})^2}. \quad (5)$$

2. **Task-Level Metrics.** We distinguish between single-answer (binary) and multi-answer tasks. For single-answer tasks, we report accuracy (mean and standard deviation); for multi-answer tasks, we report precision and recall. Let G and $\hat{G}$ denote the reference and predicted answer sets. All metrics are computed via a matching operator $\Theta(G, \hat{G})$, which may involve fuzzy matching, LLM-based scoring, `CODEBLEU`, or other task-specific methods. [See Appendix C].

To assess the Clarifier module, we follow a structured evaluation using both synthetic SOPs and the SOP-Bench dataset (Nandi et al., 2025). A set of gold-standard SOPs—complete and unambiguous—are created via manual curation and LLM assistance, and augmented with SOP-Bench examples. Controlled ambiguities are then injected into specific steps, and for each, corresponding ground-truth clarification questions are crafted. These questions represent the minimal and necessary information needed to resolve the ambiguity.

We define **Noise** as the fraction of SOP steps made ambiguous. The Clarifier's output is compared against the ground-truth using precision and recall. We assume gold SOPs are ambiguity-free (Clarifier should generate no questions), and that inserted ambiguities are isolated and resolvable. This setup ensures that the evaluation reflects the Clarifier's true ability to detect and resolve missing or unclear information in procedural instructions.

C Θ: Ground Truth Matching Operator

The Θ operator defines how predicted outputs are evaluated against ground truth across different task types. Its instantiation varies according to the nature of the task:

- **Code-Level Tasks**: For tasks such as *Execution Code Implementation*, *Code Writing*, and *Code Conversion*, Θ is instantiated as `CODEBLEU`, a metric that captures syntactic correctness, token-level overlap, and semantic equivalence between the predicted and reference code.
- **Variable Identification Tasks**: In tasks like *Execution API Implementation*, *Execution LLM Implementation*, *For Loop Implementation*, and *If Condition Implementation*, where the objective is to identify the correct `jsonPath` reference from the execution context, Θ is implemented as a strict exact-match function.
- **High-Level Planning Tasks**: For more abstract planning tasks—including *Question Generation*, *API Planning*, *Input Schema Planning*, and *Code Skeleton Planning*—Θ is defined using LLM-based relevance scoring. This may be optionally augmented with sentence embeddings and cosine similarity to assess semantic alignment and coverage.

D Hyperparameters

Table 4: Hyperparameters for each sub-module

Task	Model	Hyperparameters
Clarifier Module ($\mathcal{M}_c$)		
Question Generation	Sonnet 3.5	$T = 0.5, \pi, k = 10,$
Question Filtering*	Sonnet 3.5	$T = 0.5, \pi, \gamma = 0.75,\ \kappa = 0.7$
Answer Generation†	Sonnet 3.5	$T = 0.5, \pi$
Planner Module ($\mathcal{M}_s$)		
API Planning	Sonnet 3.5	$T = 0.7, \pi$
Input Schema Planning	Sonnet 3.5	$T = 0.7, \pi$
Code Skeleton Planning	Sonnet 3.5	$T = 0.7, \pi$
Code Conversion	Sonnet 3.5	$T = 0.5, \pi$
Implementor Module ($\mathcal{M}_i$)		
Execution API	Sonnet 3.5	$T = 0.5, \pi$
Execution Code	Sonnet 3.5	$T = 0.7, \pi$
Execution LLM	Haiku	$T = 0.5, \pi$
For Loop	Haiku	$T = 0.5, \pi$
If Condition Generation	Haiku	$T = 0.5, \pi$

Note: Let the defaults be denoted by $\pi = (\text{Top}_p = 0.999,\ \text{Top}_k = 200)$.
* Cohere.embed-english-v3
†Internal Knowledge Database (RAG) is used)

E Clarification Questions for Movie Classification SOP

Movie Classification SOP Extract the movie name from ticket description. Search the movie in IMDb and get the movie plot. Also get the IMDb rating. Similarly, search the movie on Rotten Tomatoes and get the plot and rating. Analyse

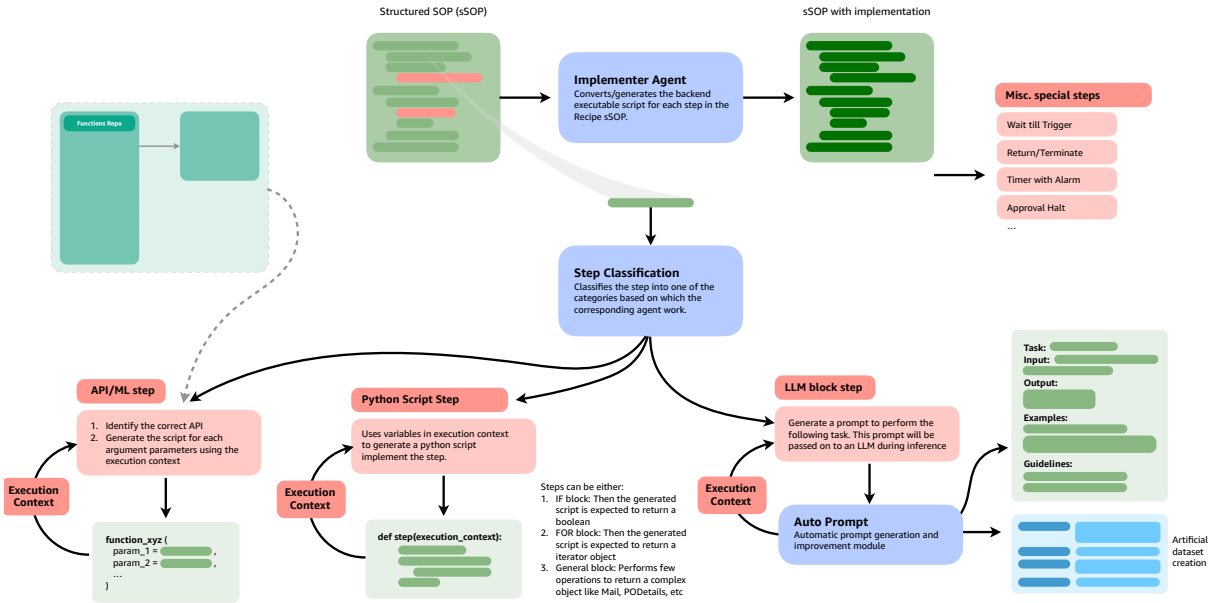

Figure 3: The working of the Implementor module.

the movie plot from both websites and classify the movie as *kid friendly* or *not kid friendly*. If the view count of the movie's trailer on YouTube is greater than 10 million, then the case needs human review. If IMDb rating is greater than 8.5 and Rotten Tomatoes rating is greater than 60%, then the case needs human review. If either the IMDb or Rotten Tomatoes rating is not found, then also consider the case for human review. Else, consider the movie does not need human review. If the movie is classified as *human review needed*, then assign a human to validate the classification. Else if the movie is classified as *human review not needed*, proceed as follows: If both IMDb and Rotten Tomatoes indicate the movie is kid friendly, update the ticket with a comment stating it is kid friendly. If both indicate the movie is not kid friendly, update the ticket with a comment stating it is not kid friendly. Else, ask a human to classify the movie plot and update the ticket with the classification.

Movie Classification SOP (Ambiguous) Get movie plot and rating. From Rotten Tomatoes, get plot and rating. Classify as kid friendly or not. Check the view count of the movie's trailer in Youtube and Decide if case needs human review. If IMDb and RT movie ratings are high, then send for human review. If either of IMDb or RT rating is not found, then also consider the case. Else consider movie does not need human review. If the movie is classified as human review needed, then ask a human to validate the classification. Else if the movie is classified as human review not needed and. If both (IMDb and RT) suggest it to be kid friendly, update the ticket with a comment mentioning it is kid friendly. If both suggest it is not kid friendly, update the ticket with a comment. Else ask human to classify and update ticket.

Clarification Questions

- From which field in the ticket details should the movie name be extracted?
- What would be considered a significant view count threshold that would trigger human review for a movie trailer?
- What rating threshold should be considered as 'high' for IMDb and Rotten Tomatoes ratings to trigger human review?
- When a human reviews the movie classification, what are the possible values they can provide in response? (For example: 'Kid Friendly'/'Not Kid Friendly' or some other format