Multi-Agent Procedural Graph Extraction with Structural and Logical Refinement

Wangyang Ying[1]*, Yanchi Liu[2]†, Xujiang Zhao[2], Wei Cheng[2], Zhengzhang Chen[2], Wenchao Yu[2], Yanjie Fu[1], Haifeng Chen[2]

[1]Arizona State University, [2]NEC Labs America
{wangyang.ying, yanjie.fu}@asu.edu
{yanchi, xuzhao, weicheng, zchen, wyu, haifeng}@nec-labs.com

Abstract

Automatically extracting workflows as procedural graphs from natural language is promising yet underexplored, demanding both structural validity and logical alignment. While recent large language models (LLMs) show potential for procedural graph extraction, they often produce ill-formed structures or misinterpret logical flows. We present *text2flow*, a multi-agent framework that formulates procedural graph extraction as a multi-round reasoning process with dedicated structural and logical refinement. The framework iterates through three stages: (1) a graph extraction phase with the graph builder agent, (2) a structural feedback phase in which a simulation agent diagnoses and explains structural defects, and (3) a logical feedback phase in which a semantic agent aligns semantics between flow logic and linguistic cues in the source text. Important feedback is prioritized and expressed in natural language, which is injected into subsequent prompts, enabling interpretable and controllable refinement. This modular design allows agents to target distinct error types without supervision or parameter updates. Experiments demonstrate that *text2flow* achieves substantial improvements in both structural correctness and logical consistency over strong baselines.

1 Introduction

Extracting workflows as structured procedural graphs from natural language documents is a fundamental yet underexplored task (Ren et al., 2023a; Du et al., 2024). A procedural graph is defined as a directed graph representing a workflow described by natural language, where nodes correspond to defined procedural elements such as actors, actions, decision points (a.k.a., gateways), constraints, and designated start/end points, and edges denote sequential execution or conditional dependencies among these elements (Herbst and Karagiannis, 1999; Maqbool et al., 2018). This form of graph explicitly encodes control-flow logic and procedural dependencies, differing from traditional knowledge graph extraction that focuses on identifying entities and relations. Procedural graphs serve as executable structures, enabling downstream applications such as task digitization, automated compliance checking, and providing structural knowledge input to intelligent systems.

Procedural graphs require modeling not only sequential actions but also complex control structures, such as conditionally executed branches and parallel execution steps. In addition, it requires a comprehensive understanding of the global executability and logical validity of the graphs they produce. However, current methods only partially address these complexities. (1) Traditional information extraction techniques, such as entity extraction (Ma and Hovy, 2016; Liu et al., 2025) and knowledge graph construction (Ji et al., 2021; Pan et al., 2024), mainly focus on identifying static entities and their semantic relationships, lacking explicit modeling of dynamic execution flows, conditional logic, or iterative loops inherent in procedural documents. (2) Rule-based methods or customized neural architectures demonstrated limited generalization beyond predefined, simplified cases (Sholiq et al., 2022; Bellan et al., 2023). They fail to adequately capture complex, non-sequential scenarios such as "customers can order only dishes, only drinks, or both," due to their intrinsic limitations in modeling nonlinear and conditional control flows. (3) Large language models (LLMs) show promise in structured extraction tasks (Zhao et al., 2025), but current LLM-based methods mainly rely on one-pass zero-shot or few-shot prompting, lacking explicit guidance and iterative refinement. Moreover, current self-refinement paradigms (Madaan et al., 2023a; Hu et al., 2025) often rely on heuristic prompting or implicit internal feedback, which offer little

*Work done during an internship at NEC Labs America.
†Corresponding author.

Findings of the Association for Computational Linguistics: EACL 2026, pages 3021–3034
March 24-29, 2026

guidance when structural flaws block execution paths or when gateway types are subtly inconsistent with textual conditions. Consequently, these methods struggle with reliably capturing intricate conditional structures and execution logic in procedural graphs (Bellan et al., 2022).

To overcome the above limitations, in this work, we propose a multi-agent framework, named *text2flow*, for procedural graph extraction via multi-agent structural and logical alignment. Specifically, we extract two explicit types of external feedback: structural errors derived from multi-path simulation, and logical inconsistencies revealed by comparing execution logic against textual descriptions. We decouple the extraction process into three iterative stages: (1) a graph extraction phase with the graph builder agent, (2) a structural feedback phase powered by a simulation agent to detect and explain topological issues, and (3) a logical feedback phase that uses a semantic agent to align between flow logic and the linguistic patterns in the source text. Important feedback is prioritized and expressed in natural language, which is injected into the next-round prompt, enabling interpretable and controllable refinement. By organizing these signals in a unified sandbox and injecting them selectively across iterative reasoning rounds, our framework enables LLMs to revise graphs not only for surface correctness, but with respect to grounded, externalized procedural reasoning. Crucially, although LLM agents are employed to interpret and verbalize feedback, all signals are grounded in observable execution outcomes or explicit inconsistencies between graph logic and text, making them traceable, interpretable, and independent of the generation model's internal state. This design avoids the compounding hallucinations often observed in self-refinement loops, and enables grounded revisions driven by externally validated failures. Experiments demonstrate that our approach significantly improves structural correctness and logical consistency over baselines.

2 Related Work

Procedural graph extraction research has historically emphasized pipelines that detect events and then link them into sequential relations, often realized through rules or templates, but these approaches rarely generalize beyond linear action flows (Pal et al., 2021; López et al., 2021; Ren et al., 2023b). Later work introduced non-sequential structures such as branches and optional steps, though coverage and robustness remain limited (Bellan et al., 2023; Honkisz et al., 2018; Epure et al., 2015). Constraint modeling has also been fragmented: data constraints were explored without corresponding attention to action-level dependencies (Friedrich et al., 2011). Small-sample neural models exist but raise doubts about scalability. To systematically evaluate these shortcomings, PAGED (Du et al., 2024) offers a benchmark of paired documents and procedural graphs, showing both classical and LLM-based methods still struggle to construct coherent non-sequential structures.

Beyond extraction, recent work shifts to generation: Cascading LLMs first identify salient events and then generate temporal/hierarchical/-causal edges as code with iterative refinement (Tan et al., 2024a); the Set-Aligning Framework treats edges as an order-invariant set with regularizations to boost recall and zero-shot generalization (Tan et al., 2024b); G-PCGRL casts graph creation as reinforcement learning over an adjacency matrix under explicit constraints, enabling faster and more controllable valid-graph generation than evolutionary or random search (Rupp and Eckert, 2024).

3 Problem Formulation

A procedural graph is a directed process flow graph that represents the control structure of a multi-step procedure, including both sequential and non-sequential execution logic. Given a document, the goal is to generate a graph $G = (V, E)$, where 1) V is a set of nodes representing process components, which includes: *Start* and *End* markers, *Action* nodes that describe concrete procedural steps, *Actor* nodes denoting the responsible entity for each action, *Gateway* nodes that encode control logic (e.g., XOR, OR, AND), and *Constraint* nodes that impose conditions or requirements. 2) E is a set of directed edges, which represents directed flows among these nodes, such as *Sequence Flows* (e.g., action-to-action transitions), *Condition Flows* (e.g., branching conditions at gateways), and *Constraint Flows* (e.g., linking constraints to specific actions).

Our objective is to generate a graph that is both structurally valid and logically faithful to the input document. We formulate this as a multi-round refinement problem, where a multi-agent system iteratively improves the graph under guidance from external structural and logical feedback.

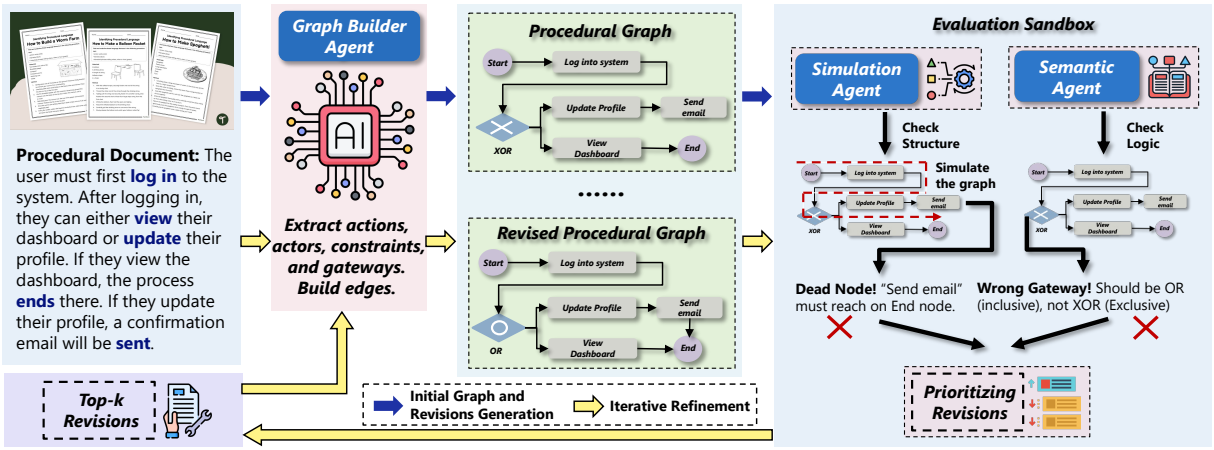


Figure 1: Overview of the *text2flow* multi-agent framework. The Graph Builder Agent generates an initial procedural graph from the document. The Evaluation Sandbox, with simulation and semantic agents, identifies structural and logic errors. Prioritized revisions are iteratively applied to the Graph Builder Agent until a valid graph is produced.

4 The *text2flow* Multi-agent Framework

4.1 Overview

Figure 1 shows that our framework treats procedural graph extraction as a multi-stage, feedback-driven process, where a large language model (LLM) iteratively generates and refines the output graph under the guidance of structural and logical feedback. Starting from a natural language document, we first use few-shot prompting to elicit an initial procedural graph that captures the core actions, control structures, and execution flows described in the original documents. Recognizing that such initial outputs may contain structural flaws (e.g., disconnected nodes, misused gateways) or logical misalignments (e.g., incorrect interpretation of conditional logic), we introduce a set of lightweight modules that serve as external evaluators. These modules examine the generated graph from two complementary perspectives: structural validity and logical consistency. Each module produces natural language feedback that identifies potential issues and suggests targeted revisions. The LLM then incorporates this feedback in the next round of generation, refining the graph through a loop of diagnosis and revision. By delivering all signals through language, our framework forms a controllable "evaluation sandbox" that enables interpretable, modular, and supervision-free correction of procedural graphs.

4.2 Graph Builder Agent

The first step in our framework is to construct an initial procedural graph from the input document using the graph builder agent. Given a natural language paragraph that describes a multi-step procedure, the agent is prompted to output a structured representation of the process as a graph, including the nodes, their types, the control flow edges, and the constraint flow edges. We adopt a few-shot prompting strategy, where the graph builder agent is provided with a small number of annotated examples in the prompt to guide its generation. Each example consists of a natural language procedure and its corresponding graph represented in a structured format. To make the format LLM-friendly and easy to post-process, the graph is represented as textual triples, such as 'action -> action' or 'gateway -> (condition) action'.

The initial graph generated by the agent serves as the first approximation of the underlying procedural structure described in the document. However, due to the complexity of procedural logic and the subtlety of control flow language (e.g., implicit conditions, disjunctions, or parallelism), this initial graph often contains structural errors or misinterpretations of gateway logic. These imperfections motivate the next stage of our framework, where feedback is collected to guide further refinement.

4.3 Evaluation Sandbox: Structural and Logical Feedback Collection

To iteratively refine the procedural graph generated by the graph builder agent, we construct an evaluation sandbox that identifies structural and logical issues in natural language. The sandbox operates through two complementary agents: (1) a **simulation agent** that executes the graph to reveal structural failures, and (2) a **semantic agent** that verifies gateway logic based on textual cues. We

further define a scoring and selection strategy to prioritize feedback items under prompt constraints.

4.3.1 Structural Feedback via Simulation Agent

(1) Constructing a simulator to detect the structural issues. Unlike heuristic-based graph checkers, we treat the generated graph as an executable representation of procedural logic. We design a simulator that mimics multi-path execution from the Start node to the End node, traversing all valid branches under stochastic or enumerative conditions. Each simulation yields a trace-level execution record and flags any violations encountered during traversal. These violations are emitted in a structured form:

$$\{(\tau, i, c)_k\}_{k=1}^{n} = \mathcal{S}(G), \tag{1}$$

where τ is the execution trace (sequence of nodes), i is the structured issue encountered (e.g., dead node), c is the condition choice at each gateway in the trace, n is the number of simulations, $\mathcal{S}$ is the notation of the simulator, and G is the graph that needs to be refined.

(2) Using LLM to provide revision feedback. Each error trace is passed to an LLM-based structural agent, which determines whether the issue reflects a genuine structural flaw and, if so, returns a corresponding repair feedback. This feedback may involve operations such as adding a missing edge, modifying a gateway type, or reconnecting a disjoint node:

$$f_k = \text{LLM}_{\text{stru}}((\tau, i, c)_k), f_k \in \mathcal{F}_t^{\text{stru}}, \tag{2}$$

where t is the timestep of the current iteration.

4.3.2 Logical Feedback via Semantic Agent

(1) Gateway logical feedback. Procedural graphs often rely on control flow nodes such as XOR, OR, and AND gateways to represent conditional or parallel behaviors. However, unlike structural errors that manifest during execution, gateway type mismatches arise from a misalignment between graph logic and the linguistic cues in the original text. To support this comparison, we first extract a local textual span for each gateway node, capturing the clause most relevant to its control logic. This step isolates the relevant context and reduces noise from surrounding text. The span is extracted using a semantic retrieval agent:

$$s_g = \text{LLM}_{extract}(g, G, D). \tag{3}$$

where s_g is the textual span corresponding to the gateway g, G is the graph that needs to be refined, and D is the original document.

We then ask an LLM to transfer the gateway g and related context back to text description d_g. In this way, the comparison of logic between the raw document and the procedural graph is under the same semantic space. Then a semantic consistency agent is used to judge whether the current gateway type matches the logic expressed in s_g. Crucially, we only retain feedback when the agent determines that the gateway type is inconsistent with the text, ignoring cases deemed valid or ambiguous:

$$f_g = \text{LLM}_{\text{sem}}(s_g, d_g), f_g \in \mathcal{F}_t^{\text{sem}}. \tag{4}$$

These feedback items flag specific gateway nodes where the control logic is semantically incorrect.

(2) Flow logical feedback. In addition to the gateways (XOR, OR, AND) which express branching and merging of logic. Flows like sequence, condition, and constraint present logic between nodes. To capture this flow logic, we segment the entire graph into multiple gateway-to-gateway fragments. For each gateway g, we extract its simulation trace t_g, defined as the subgraph that spans from g to the closest downstream gateway along the flow sequence. This trace includes: flow-sequence edges along the execution path from gateway g to the next gateway; flow-condition edges that originate from g and point to action nodes within the segment, representing control decisions made by the gateway; flow-constraint edges where constraint nodes point to any action node within this segment, specifying conditional requirements on their execution.

Similarly, we ask an LLM to transfer the trace t_g to a text description d_t. In this way, the comparison of logic between the raw document and the procedural graph is under the same semantic space. The semantic consistency agent is used to judge whether the current flow segment matches the logic expressed in the original document. We retain feedback when the agent determines that the gateway segment is inconsistent with the text:

$$f_t = \text{LLM}_{\text{sem}}(s_t, d_t), f_t \in \mathcal{F}_t^{\text{sem}}, \tag{5}$$

where s_t is the textual span of trace t_g in the original document. This feedback highlights segments whose assigned control types are semantically incorrect. All of the above feedback is then routed into the sandbox for prioritization and refinement in subsequent iterations.

4.3.3 Prioritizing High-Impact and Unresolved Feedback

Why prioritize feedback for refinement? In each iteration, the LLM-based generator receives a limited number of natural language feedback prompts due to context window and prompt clarity limitations. Presenting too many suggestions simultaneously can overwhelm the model or dilute the guidance signal. Therefore, we prioritize a subset of feedback items that are most likely to yield meaningful corrections. This motivates a scoring-and-selection framework that ranks feedback based on estimated utility.

(1) Prioritizing revisions by error frequency impact. While each simulation provides only a local view of structural issues, the distribution of errors across a large number of simulations reveals their global importance. Structural flaws that appear repeatedly across different execution traces are more likely to lie along central or high-traffic paths in the graph. Fixing such high-frequency errors can unlock access to a broader set of paths and downstream nodes. In contrast, rarely encountered issues may have limited procedural impact even if technically incorrect.

To capture this intuition, we conduct a large number of simulations (e.g., 10,000 trials) using uniform sampling at each gateway to ensure broad path coverage. We then count how frequently each structural issue appears across all wrong simulations. For a given repair suggestion, we define its marginal utility as the normalized frequency of the corresponding error simulation:

$$u(f_k) = \frac{\text{count}((\tau, i, c)_k)}{\Sigma_{j=1}^{n} \text{count}((\tau, i, c)_j)}, f_k \in \mathcal{F}_t^{\text{stru}}, \quad (6)$$

where count$(\cdot)$ indicates how many times a specific trace was generated in the simulations.

(2) Promoting unresolved feedback from previous rounds. To avoid repeatedly surfacing but ignoring the same issue, we incorporate a RepeatFailure score. A feedback item $f \in \mathcal{F}_t$ is considered a repeat if its semantic meaning is highly similar to a previously surfaced item that was not fixed. We detect repetition using BLEU similarity:

$$R(f) = \max_{f' \in \mathcal{F}_{<t}} [\text{BLEU}(f, f')] \quad (7)$$

where $\mathcal{F}_{<t}$ is the feedback in previous iteration.

(3) Unified feedback scoring and selection. While both structural and logical errors affect the overall correctness of the generated graph, structural flaws undermine the executability and interpretability of the graph as a whole. In particular, logic judgments, such as whether a gateway is logically consistent with its textual reference, are only meaningful when the underlying structure is coherent and complete. If a node is unreachable or a branch is disconnected, any attempt to validate its logic becomes unreliable or irrelevant.

Moreover, structural issues can be measured at fine granularity via simulation frequency, yielding a continuous estimate of procedural impact. In contrast, logical errors (e.g., mis-typed gateways) are assessed through discrete judgments and lack trace-level grounding. We encode this asymmetry in the scoring:

$$w(f) = \begin{cases} u(f) + R(f), & f \in \mathcal{F}^{\text{stru}} \\ R(f), & f \in \mathcal{F}^{\text{sem}} \end{cases} \quad (8)$$

where $u(f)$ is the simulation-derived impact of structural feedback f, $R(f)$ is a shared repair prior, $\mathcal{F}^{\text{stru}}$ denotes structural feedback, and $\mathcal{F}^{\text{sem}}$ denotes logical feedback. Consequently, structural issues with broad procedural impact receive higher priority during refinement, while logical feedback is incorporated once the graph is structurally stable enough to support reliable logical comparison.

(4) Budget-constrained selection. We solve a knapsack-style problem to select the top-scoring feedback items within a fixed token budget B:

$$\max_{S \subseteq \mathcal{F}_t} \sum_{f \in S} w(f) \quad \text{s.t.} \quad \sum_{f \in S} \ell(f) \leq B, \quad (9)$$

where $\ell(f)$ denotes the token length of feedback f. We greedily pick the top-k items by the efficiency score $w(f)/\ell(f)$ under the prompt budget.

4.4 Multi-Round Prompting for Graph Refinement

The feedback-guided graph construction process is conducted in multiple rounds. Each round consists of three stages: (1) generating a procedural graph based on the current prompt, (2) collecting structural and logical feedback via simulation and agent-based analysis, and (3) selecting a prioritized subset of feedback to include in the next prompt. This iterative process enables the model to correct both executional flaws and logical inconsistencies. The refinement process continues for a fixed number of rounds or until no high-impact feedback remains. This multi-round setup allows us to decompose

complex graph construction tasks into incremental, feedback-driven revisions, improving structural validity and logical consistency over time.

5 Experiments

5.1 Experimental Setup

Dataset Descriptions. We conduct our experiments on the PAGED benchmark (Du et al., 2024), a large-scale dataset constructed for procedural graph extraction. The corpus contains 3,394 documents with 37,226 sentences and more than 560K tokens in total. Each document is annotated with diverse process elements. **Table 4 in Appendix A** summarizes the statistics of these core components. The dataset represents procedural knowledge in a text-driven graph format, where sentences are aligned with nodes such as actions, actors, gateways, and constraints, and edges are explicitly labeled as flows (e.g., `'element -> element'` or `'element -> (condition) element'`) to capture temporal and logical dependencies. In our study, we exclusively utilize the official 20% held-out test split of PAGED to evaluate model performance. Since our framework operates in a fully unsupervised setting, no training signals from the benchmark are used; instead, we directly apply our method to the test data for evaluation.

Evaluation Metrics. The PAGED dataset represents procedural knowledge as tuples in the form of `'element -> element'` or `'element -> (condition) element'`. Model predictions are evaluated by comparing predicted tuples with gold tuples of the same type.

(1) Tuple Matching. For each predicted tuple, we compute its BLEU score (Liang et al., 2023) with all gold tuples. The gold tuple with the highest score is selected as the candidate match if the score exceeds 0.5. We then further check:

- Start and End Elements: considered matched if its BLEU score exceeds 0.75.
- Condition (if present): compared against the gold condition using BLEU.

(2) Element-Level Evaluation.

- Actions, Actors, and Constraints: Evaluated with soft F1 (Tandon et al., 2020). A predicted element is correct if it can be matched to a gold element with BLEU above the threshold.
- Gateways: Evaluated with hard F1. A predicted gateway is correct only if both (i) its type matches the gold gateway, and (ii) it connects to at least one element that is correctly matched (Dumas et al., 2018).

(3) Flow-Level Evaluation.

- A predicted flow is correct if its type and the two connected elements both match the corresponding gold flow.
- For Condition Flows, the condition text must also achieve BLEU alignment with the gold condition.

Baselines. (1) Machine Translation-like BPMN (MT-BPMN): (Sonbol et al., 2023) which is a semantic transfer-based machine translation approach that generates BPMN models from textual descriptions via intermediate Concept Maps. **(2) Beyond Rule-based Process extraction (BRP):** (Neuberger et al., 2023) which is a data-driven pipeline that extracts process models from text by combining NER, entity resolution, and relation extraction, outperforming rule-based methods. **(3) BPMN-Gen:** (Sholiq et al., 2022) which is a rule-based NLP method for generating BPMN diagrams from textual requirements. **(4) PET:** (Bellan et al., 2023) which uses Roberta-large (Liu et al., 2019) as the backbone-model and train the model on the PET dataset[1]. A total of 3 epochs are trained, the adopted optimizer is AdamW and the learning rate is set to 5e-6. **(5) CIS:** (Bellan et al., 2022) which applies GPT-3 with in-context learning to incrementally extract activities, participants, and control-flow relations from process descriptions via conversational question answering. **(6) Self-Refine:** (Madaan et al., 2023b) which introduces an iterative refinement framework where a single LLM alternates between generating outputs, providing feedback on them, and then refining its own responses. **(7) Actor-Critic:** (Shinn et al., 2023) which frames LLM agents in an actor–critic style, where an actor executes actions and a critic provides verbal feedback to guide iterative self-improvement. This paradigm has become a widely used baseline for multi-agent reasoning and self-correction in LLM research.

Models Details. We implement *text2flow* on 5 LLMs to evaluate the effectiveness. The details of the LLMs are described in **Appendix B**. And the implementation details are described in **Appendix C**.

[1] https://huggingface.co/datasets/patriziobellan/PET

Table 1: Overall performance. We report the **F1-score** of each procedural graph component — including action nodes, gateway nodes (XOR/OR/AND), and directed edges — to evaluate the extraction accuracy at both node and structure levels. Higher values indicate better performance. Best results in **bold**; second-best underlined.

Model		Actor	Action	Constraint		Gateway			Flow		
				Data	Action	XOR	OR	AND	Sequence	Condition	Constraint
MT-BPMN		0.028	0.308	0.213	-	0.485	-	0.279	0.056	0.047	0.017
BRP		0.027	0.276	-	-	0.469	-	0.337	0.074	0.061	-
BPMN-Gen		-	0.387	-	-	0.463	-	0.198	0.091	0.022	-
PET		0.085	0.430	0.069	-	0.493	-	-	0.164	0.026	-
CIS		0.633	0.639	-	-	0.455	-	-	0.203	0.157	-
Lamma3.1:8B	Few-Shot	0.403	0.649	0.534	0.151	0.701	0.141	0.366	0.315	0.186	0.288
	Self-Refine	0.568	0.708	0.562	0.172	0.702	0.152	0.347	0.368	0.204	0.399
	Actor-Critic	0.504	0.670	0.556	0.165	0.673	0.150	0.327	0.321	0.182	0.371
	text2flow	0.577	0.742	0.644	0.178	0.703	0.144	0.370	0.395	0.238	0.436
Gemma3:27B	Few-Shot	0.621	0.774	0.771	0.445	0.789	0.280	0.601	0.371	0.276	0.477
	Self-Refine	0.612	0.774	0.734	0.434	0.784	0.239	0.642	0.475	0.280	0.638
	Actor-Critic	0.469	0.742	0.625	0.339	0.728	0.217	0.574	0.417	0.248	0.472
	text2flow	0.642	**0.786**	0.791	0.452	**0.839**	0.345	0.648	0.479	0.342	0.682
Qwen3:30B	Few-Shot	0.623	0.762	0.816	0.421	0.762	0.201	0.557	0.339	0.232	0.499
	Self-Refine	0.653	0.762	0.846	0.421	0.792	0.201	0.587	0.339	0.252	0.499
	Actor-Critic	0.518	0.750	0.714	0.388	0.734	0.237	0.584	0.344	0.246	0.555
	text2flow	0.644	0.776	0.843	0.411	0.792	0.238	0.581	0.429	0.276	0.684
Mistral3.1:24B	Few-Shot	0.711	0.778	0.776	0.461	0.765	0.192	0.613	0.423	0.280	0.611
	Self-Refine	0.717	0.778	0.790	0.455	0.804	0.249	0.639	0.450	0.284	0.619
	Actor-Critic	0.660	0.754	0.709	0.335	0.815	0.256	0.633	0.455	0.291	0.556
	text2flow	**0.723**	0.781	0.812	0.460	0.832	0.282	0.677	0.476	0.317	0.653
GPT-4o	Few-Shot	0.684	0.769	0.841	0.481	0.792	0.296	0.648	0.434	0.302	0.594
	Self-Refine	0.691	0.769	0.838	0.483	0.796	0.299	0.673	0.476	0.306	0.673
	Actor-Critic	0.617	0.706	0.760	0.441	0.726	0.225	0.651	0.425	0.237	0.581
	text2flow	0.688	0.782	**0.859**	**0.485**	0.831	**0.356**	**0.692**	**0.484**	**0.357**	**0.710**

5.2 Overall performance compared with different baselines and LLM models

This experiment tests whether a multi-agent pipeline with an evaluation sandbox improves procedural-graph extraction over non-LLM parsers and strong LLM baselines across five backbones. **Table 1** shows that our system attains the highest F1 in most settings, with the largest gains in flow constraints and sequence, and clear gains in gateways follow the semantic consistency in the text. The underlying driver is that the evaluation sandbox provides verifiable signals of two kinds: structural signals from simulation, reachability, and end node checks that reveal broken or missing paths, and logical signals from consistency checks between the text and the candidate gateways. By converting simulation and semantic checks into verifiable constraints, the evaluation sandbox turns free-form generation into guided repair, producing graphs that satisfy global structure and semantics. Compared with Self-Refine, it provides checkable pass/-fail signals that indicate what to revise and when to stop; compared with Actor-Critic, specialized critics use simulation traces and text–logic consistency to target gateway typing, reachability, flow constraints, etc. Overall, the experiment shows that injecting external signals enables the LLM to generate procedural graphs that are more accurate and more globally consistent across backbones.

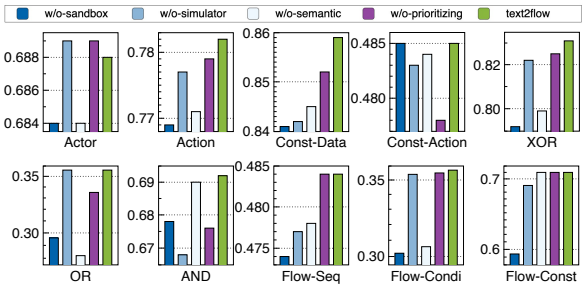


Figure 2: Ablation Study on GPT-4o backbone. Each bar shows the performance under different module removals: **w/o-sandbox** – few-shot generation without evaluation sandbox; **w/o-simulator** – iterative refinement using only the semantic agent as feedback; **w/o-semantic** – iterative refinement using only the simulator as feedback; **w/o-prioritizing** – all revision candidates are sent to the graph builder without filtering. ***text2flow*** denotes the full model with all components enabled.

5.3 Ablation study

The purpose of this experiment is to examine the contribution of each component in our framework

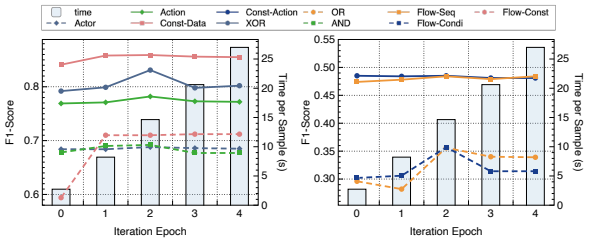
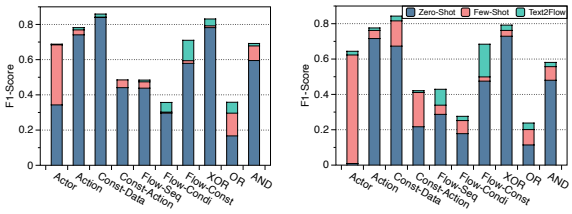


Figure 3: F1-scores and average time per sample across iterations (0 = few-shot). Gains peak around 2 iterations, after which improvements plateau while cost increases.



(a) GPT-4o (b) Qwen3:30B

Figure 4: F1-score comparison of Zero-Shot, Few-Shot, and *text2flow* on GPT-4o and Qwen3:30B. *text2flow* shows consistent gains across all categories.

on the GPT-4o backbone by progressively removing the evaluation sandbox, simulator, semantic agent, and prioritization mechanism. As shown in **Figure 2**, the full model consistently outperforms all ablated variants across different structural dimensions. Without the sandbox, performance drops notably on control-flow related metrics (e.g., OR, Flow-Condi), highlighting the necessity of external evaluation signals to constrain generation. When relying solely on the simulator (w/o-semantic) or solely on the semantic agent (w/o-simulator), the model suffers on gateway recognition and constraint alignment, suggesting that the two types of feedback are complementary: the simulator enforces structural validity while the semantic agent preserves logical consistency. In addition, disabling the prioritization module (w/o-prioritizing) leads to diminished accuracy on complex node types, since indiscriminate revisions propagate noise and overwhelm the graph builder. These results indicate that each component provides unique benefits, and their integration yields the most stable and robust improvements.

5.4 Performance–Cost Across Iterations

The purpose of this experiment is to evaluate the scalability of our framework on the GPT-4o backbone by examining how performance and computational cost change with the number of refinement iterations. As shown in **Figure 3**, F1-scores across most structural categories improve notably from the few-shot baseline (iteration 0) to the first two refinement rounds, but the gains quickly plateau afterwards. In contrast, the average time per sample grows steadily with more iterations, resulting in diminishing returns. This phenomenon is driven by the fact that early revisions effectively correct major structural and semantic errors, while later rounds tend to revisit already-correct patterns, incurring additional cost without substantial improvement. Consequently, we adopt two iterations as the default setting to strike a balance between accuracy and efficiency.

5.5 Fine-Grained Comparison with Zero- and Few-Shot Prompts

To evaluate the effectiveness of *text2flow*, we conduct a detailed comparison against Zero-Shot and Few-Shot prompting baselines across fine-grained graph components. **Figure 4** shows the results on GPT-4o and Qwen3:30B. We observe that *text2flow* consistently outperforms both Zero-Shot and Few-Shot settings across nearly all components, with especially notable improvements on logically complex or sparsely supervised types such as gateways and flows. These categories often require multi-hop reasoning or global structural consistency, which traditional prompting struggles to capture. The underlying driver of this improvement lies in our multi-agent feedback loop, which enables iterative correction guided by external structural and semantic constraints. Rather than relying solely on single-pass generation, the system revises outputs based on execution and alignment feedback, leading to more accurate and consistent graph construction.

5.6 Comparisons with SFT on Procedural Graph Extraction

Supervised Fine-Tuning with LoRA on Llama-3.1-8B was explored to assess whether task-specific adaptation can improve procedural graph extraction under limited supervision. Experimental results in **Table 2** show that fine-tuning yields localized improvements on structurally explicit components, most notably AND/OR gateway typing, where it outperforms text2flow under the same model scale. However, these gains do not translate into consistent improvements across semantic roles or global flow-level metrics, such as Flow-Seq and Flow-Constrain, where fine-tuning remains inferior to inference-based methods. Moreover, the overall performance of the fine-tuned model is still substantially lower than that of larger models (e.g., Gemma-27B or GPT-4o), while introducing non-

Table 2: Performance comparison of few-shot, reasoning-based, and supervised fine-tuning methods under the Llama-3.1-8B SFT setting for procedural graph extraction.

Model	Actor	Action	Constraint		Gateway			Flow		
			Data	Action	XOR	OR	AND	Sequence	Condition	Constraint
Few-Shot	0.403	0.649	0.534	151	0.701	0.141	0.366	0.315	0.186	0.288
Self-Refine	0.568	0.708	0.562	0.172	0.702	0.152	0.347	0.368	0.204	399
Actor-Critic	0.504	0.670	0.556	0.165	0.673	0.150	0.327	0.321	0.182	0.371
text2flow	0.577	0.742	0.644	0.178	0.703	0.144	0.370	0.395	0.238	0.436
Fine-tuning	0.467	0.697	0.527	0.187	0.667	0.163	0.406	0.317	0.191	0.323

Table 3: Token Cost on different methods.

Method	Zero-Shot	Few-Shot	Self-Refine	Actor-Critic	*text2flow*
Num Tokens	305	1256	1801	1887	2489

trivial computational overhead. These results suggest that, given limited annotated data and the semantic complexity of procedural graph extraction, supervised fine-tuning provides partial and complementary benefits, but cannot replace large-model reasoning capabilities.

5.7 Token Cost Analysis

We further analyze the token cost of different methods by reporting the average prompt and completion token usage per sample. As shown in **Table 3**, token consumption increases substantially with more complex prompting and multi-step reasoning strategies. Zero-shot inference is the most token-efficient (305 tokens), while few-shot prompting already incurs a notable increase (1,256 tokens). Iterative reasoning methods, such as Self-Refine and Actor-Critic, further amplify token usage due to multiple generations and feedback cycles. In contrast, *text2flow*exhibits the highest token cost (2,489 tokens per sample), reflecting its reliance on explicit reasoning steps and structured output generation. These results highlight a clear trade-off between performance gains and inference efficiency, suggesting that improvements achieved by advanced reasoning-based methods come at a nontrivial computational and cost overhead.

6 Conclusion

In conclusion, this work introduces *text2flow*, a multi-agent framework for procedural graph extraction that systematically addresses both structural validity and logical consistency, two critical yet often overlooked challenges in this domain. By decoupling extraction, structural refinement, and logical refinement into iterative, agent-driven stages, our approach provides an interpretable and modular solution that enhances reliability without requiring additional supervision or model training. Empirical results demonstrate that this framework yields significant improvements over existing baselines, underscoring the value of multi-round reasoning and feedback-driven refinement. Moreover, we see opportunities to extend this paradigm to broader workflow understanding tasks, integrate domain-specific simulators, and further explore multi-agent collaboration as a general strategy for improving structured reasoning in LLMs.

7 Limitations

Our framework is mainly evaluated on structured procedural texts, and its generalizability to informal or domain-specific documents remains to be verified. While the proposed multi-agent refinement process substantially improves structural and logical consistency, it also introduces additional computational overhead compared to single-pass generation. Moreover, the current simulator and semantic agents rely on rule-based heuristics and pretrained LLM judgments, which may not fully capture domain-specific reasoning patterns. Future extensions could incorporate adaptive reward functions or lightweight feedback mechanisms to improve scalability and robustness across diverse procedural domains.

8 Acknowledgement

Dr. Yanjie Fu is supported by the National Science Foundation (NSF) via the grant numbers: 2426340, 2416727, 2421865, 2421803.

References

Patrizio Bellan, Mauro Dragoni, and Chiara Ghidini. 2022. Leveraging pre-trained language models for conversational information seeking from text. Preprint, arXiv:2204.03542.

Patrizio Bellan, Han van der Aa, Mauro Dragoni, Chiara Ghidini, and Simone Paolo Ponzetto. 2023. Pet: An annotated dataset for process extraction from natural language text tasks. In Business Process Management Workshops - BPM 2022 International Workshops, Revised Selected Papers, Lecture Notes in Business Information Processing, pages 315–321. Springer Science and Business Media Deutschland GmbH. Publisher Copyright: © 2023, Springer Nature Switzerland AG.; Workshops on AI4BPM, BP-Meet-IoT, BPI, BPM and RD, BPMS2, BPO, DEC2H, and NLP4BPM 2022, co-located with the 20th International Conference on Business Process Management, BPM 2022 ; Conference date: 11-09-2022 Through 16-09-2022.

Weihong Du, Wenrui Liao, Hongru Liang, and Wenqiang Lei. 2024. PAGED: A benchmark for procedural graphs extraction from documents. In Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 10829–10846, Bangkok, Thailand. Association for Computational Linguistics.

Marlon Dumas, L Marcello Rosa, Jan Mendling, and A Hajo Reijers. 2018. Fundamentals of business process management. Springer.

Elena Viorica Epure, Patricia Martín-Rodilla, Charlotte Hug, Rebecca Deneckère, and Camille Salinesi. 2015. Automatic process model discovery from textual methodologies. In 2015 IEEE 9th international conference on research challenges in information science (RCIS), pages 19–30. IEEE.

Fabian Friedrich, Jan Mendling, and Frank Puhlmann. 2011. Process model generation from natural language text. In International conference on advanced information systems engineering, pages 482–496. Springer.

Joachim Herbst and DIMITRIS Karagiannis. 1999. An inductive approach to the acquisition and adaptation of workflow models. In Proceedings of the IJCAI, volume 99, pages 52–57. Citeseer.

Krzysztof Honkisz, Krzysztof Kluza, and Piotr Wiśniewski. 2018. A concept for generating business process models from natural language description. In International Conference on Knowledge Science, Engineering and Management, pages 91–103. Springer.

Shengran Hu, Cong Lu, and Jeff Clune. 2025. Automated design of agentic systems. In The Thirteenth International Conference on Learning Representations.

Shaoxiong Ji, Shirui Pan, Erik Cambria, Pekka Marttinen, and Philip S Yu. 2021. A survey on knowledge graphs: Representation, acquisition, and applications. IEEE transactions on neural networks and learning systems, 33(2):494–514.

Hongru Liang, Jia Liu, Weihong Du, Dingnan Jin, Wenqiang Lei, Zujie Wen, and Jiancheng Lv. 2023. Knowing-how & knowing-that: A new task for machine comprehension of user manuals. arXiv preprint arXiv:2306.04187.

Xuan Liu, John Ahmet Erkoyuncu, Jerry Ying Hsi Fuh, Wen Feng Lu, and Bingbing Li. 2025. Knowledge extraction for additive manufacturing process via named entity recognition with llms. Robotics and Computer-Integrated Manufacturing, 93:102900.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692.

Hugo A López, Rasmus Strømsted, Jean-Marie Niyodusenga, and Morten Marquard. 2021. Declarative process discovery: Linking process and textual views. In International Conference on Advanced Information Systems Engineering, pages 109–117. Springer.

Xuezhe Ma and Eduard Hovy. 2016. End-to-end sequence labeling via bi-directional LSTM-CNNs-CRF. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 1064–1074, Berlin, Germany. Association for Computational Linguistics.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. 2023a. Self-refine: iterative refinement with self-feedback. In Proceedings of the 37th International Conference on Neural Information Processing Systems, NIPS '23, Red Hook, NY, USA. Curran Associates Inc.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, and 1 others. 2023b. Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems, 36:46534–46594.

Bilal Maqbool, Farooque Azam, Muhammad Waseem Anwar, Wasi Haider Butt, Jahan Zeb, Iqra Zafar, Aiman Khan Nazir, and Zuneera Umair. 2018. A comprehensive investigation of bpmn models generation from textual requirements—techniques, tools and trends. In International Conference on Information Science and Applications, pages 543–557. Springer.

Julian Neuberger, Lars Ackermann, and Stefan Jablonski. 2023. Beyond rule-based named entity recognition and relation extraction for process model generation from natural language text. In International Conference on Cooperative Information Systems, pages 179–197. Springer.

Kuntal Kumar Pal, Kazuaki Kashihara, Pratyay Banerjee, Swaroop Mishra, Ruoyu Wang, and Chitta Baral.

2021. Constructing flow graphs from procedural cybersecurity texts. arXiv preprint arXiv:2105.14357.

Shirui Pan, Linhao Luo, Yufei Wang, Chen Chen, Jiapu Wang, and Xindong Wu. 2024. Unifying large language models and knowledge graphs: A roadmap. IEEE Transactions on Knowledge and Data Engineering, 36(7):3580–3599.

Haopeng Ren, Yushi Zeng, Yi Cai, Bihan Zhou, and Zetao Lian. 2023a. Constructing procedural graphs with multiple dependency relations: A new dataset and baseline. In Findings of the Association for Computational Linguistics: ACL 2023, pages 8474–8486, Toronto, Canada. Association for Computational Linguistics.

Haopeng Ren, Yushi Zeng, Yi Cai, Bihan Zhou, and Zetao Lian. 2023b. Constructing procedural graphs with multiple dependency relations: A new dataset and baseline. In Findings of the Association for Computational Linguistics: ACL 2023, pages 8474–8486.

Florian Rupp and Kai Eckert. 2024. G-pcgrl: Procedural graph data generation via reinforcement learning. In 2024 IEEE Conference on Games (CoG), pages 1–8. IEEE.

Noah Shinn, Federico Cassano, Beck Labash, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning, 2023. URL https://arxiv.org/abs/2303.11366, 1.

Sholiq Sholiq, Riyanarto Sarno, and Endang Siti Astuti. 2022. Generating bpmn diagram from textual requirements. Journal of King Saud University-Computer and Information Sciences, 34(10):10079–10093.

Riad Sonbol, Ghaida Rebdawi, and Nada Ghneim. 2023. A machine translation like approach to generate business process model from textual description. SN Computer Science, 4(3):291.

Xingwei Tan, Yuxiang Zhou, Gabriele Pergola, and Yulan He. 2024a. Cascading large language models for salient event graph generation. arXiv preprint arXiv:2406.18449.

Xingwei Tan, Yuxiang Zhou, Gabriele Pergola, and Yulan He. 2024b. Set-aligning framework for auto-regressive event temporal graph generation. arXiv preprint arXiv:2404.01532.

Niket Tandon, Keisuke Sakaguchi, Bhavana Dalvi Mishra, Dheeraj Rajagopal, Peter Clark, Michal Guerquin, Kyle Richardson, and Eduard Hovy. 2020. A dataset for tracking entities in open domain procedural text. arXiv preprint arXiv:2011.08092.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, and 3 others. 2025. A survey of large language models. Preprint, arXiv:2303.18223.

A Dataset Descriptions

We conduct our experiments on the PAGED benchmark (Du et al., 2024), a large-scale dataset constructed for procedural graph extraction. The corpus contains 3,394 documents with 37,226 sentences and more than 560K tokens in total. Each document is annotated with diverse process elements. **Table 4** summarizes the definitions and statistics of these core components. The dataset represents procedural knowledge in a text-driven graph format, where sentences are aligned with nodes such as actions, actors, gateways, and constraints, and edges are explicitly labeled as flows (e.g., `'element -> element'` or `'element -> (condition) element'`) to capture temporal and logical dependencies. In our study, we exclusively utilize the official 20% held-out test split of PAGED to evaluate model performance. Since our framework operates in a fully unsupervised setting, no training signals from the benchmark are used; instead, we directly apply our method to the test data for evaluation.

B Model Details

(1) Llama3.1:8B is an instruction-tuned model from Meta's LLaMA 3.1 family, designed for efficient reasoning and alignment in downstream tasks. **(2) Gemma3:27B** is a Google-released instruction-following model optimized for structured generation and robust alignment, particularly effective in long-context and knowledge-intensive tasks. **(3) QWen3:30B** is a large-scale model from Alibaba's Qwen3 family, trained with rich multilingual and domain-specific corpora, achieving strong performance in reasoning and information extraction. **(4) Mistral3.2:24B** is a 24B-parameter model from the Mistral 3.2 series, optimized for efficiency and high-quality instruction following, achieving competitive performance across reasoning and knowledge-intensive tasks. **(5) GPT-4o** is OpenAI's flagship multimodal model supporting text, vision, and speech, widely regarded as state-of-the-art in general-purpose reasoning and instruction following.

Table 4: Statistics of the dataset. We present statistical information on the number of documents and various elements in the dataset.

Component	Num	Definitions
Document	3,394	-
Sentence	37,226	-
Token	566,639	-
Action	36,537	A concrete operation or step that needs to be executed in the process.
Actor	22,775	The entity (human, system, or role) responsible for performing an action.
Exclusive Gateway (XOR)	7,024	A branching point where exactly one of the outgoing paths can be taken.
Inclusive Gateway (OR)	1,204	A branching point where one or more of the outgoing paths may be taken.
Parallel Gateway (AND)	2,050	A branching point where all outgoing paths must be executed in parallel.
Data Constraint	3,500	A specification of data required for an action.
Action Constraint	2,307	A restriction on how an action should be executed.
Sequence Flow	36,438	The directed edge that defines the temporal order between elements.
Condition Flow	10,598	A conditional dependency governing process transitions.
Constraint Flow	5,807	A relation capturing logical or structural restrictions between elements.

C Implementation Details

Our framework supports multiple LLM backbones, including ChatGPT, Gemma, LLaMA, etc. The initial procedural graph is generated via few-shot prompting with 3 in-context examples. Refinement proceeds for at most 2 iterations (epochs) and terminates early if the evaluation sandbox proposes no further edits. The simulator executes 10,000 trials per graph, sampling each gateway branch with equal probability. At each iteration, we apply no more than three revision suggestions. Open-source models run on 2× NVIDIA RTX A6000 GPUs, while GPT models are accessed via API.

D Prompts

D.1 Few Shot Graph Generation

Few Shot Graph Generation Prompt

Please generate procedrual graph based on the extraction rules and the procedural document.

DEFINITIONS and RULES:
The Procedural Graph contains the following types of "Nodes" and "Flows":

"Nodes":
"Start": start node indicates the start of a procedure, represented as "Start".
"End": end node indicates the ending of a procedure, represented as "End".
"Action": action node indicates a specific step in a procedure, represented as the step itself, such as "prepare the ingredients".
"XOR": exclusive gateway, indicates that only one of the following non-sequential actions can be executed, distinguish by numbers, such as "XOR1".
"OR": inclusive gateway, indicates that one or more of the following non-sequential actions can be executed, distinguish by numbers, such as "OR1".
"AND": parallel gateway, indicates that all of the following actions should be executed in parallel, distinguish by numbers, such as "AND1".
"DataObject": DataObject indicates the constraints for the necessary data of the actions, represented as "DataObject(data object)".
"TextAnnotation": TextAnnotation indicates essential notices need to be considered for the execution of the actions, represented as "TextAnnotation(essential notices)".

" Flows":
"SequenceFlow": flow that represents the execution of sequential actions, such as "Start -> prepare the ingredients".
"ConditionFlow": the condition flow is used to indicate that the following action is performed under the condition on the Condition Flow, such as "XOR1 -> (condition1) choose the first one".
"ConstraintFlow": flow that is used to connect the constraints with corresponding actions, such as "prepare the ingredients -> TextAnnotation(essential notices)".

In addition, the actor of corresponding actions is put in the front of corresponding elements to indicate the actor of the following actions if needed, such as "For actor1:".

You should generate the graph in the format of "Node -> Node" line by line until generating the whole graph for the given Procedural Document, and keep the text of the nodes and conditions consistent with the original Procedural Document.

Here are some examples:

"Procedural Document":
Firstly, the customer needs to find an empty seat. If the customer needs dishes, then choose the desired dishes and specify the taste. If the customer needs drinks, then order the drinks and specify the size. The customer then submits the order, which is added to the order list. After enjoying the meal, the customer should choose the payment method. If the credit card is available, the customer pays by credit card; else if the credit card is not available, the customer should pay in cash. For the restaurant, once receiving the order from the order list, it prepares the meal according to the order and prepares the tableware for the customer at the same time. The meal is then served for the customer to enjoy. After that, the restaurant asks the customer to pay for the order and then confirms the payment. Note that the restaurant should provide the receipt if the customer needs. And the procedure ends.

"Procedural Graph":

For the customer:
Start -> find an empty seat
find an empty seat -> OR1
OR1 -> (needs dishes) choose the desired dishes
OR1 -> (needs drinks) order the drinks
choose the desired dishes -> specify the taste
order the drinks -> specify the size
specify the taste -> OR2
specify the size -> OR2
OR2 -> submits the order
submits the order -> DataObject(order list)
submits the order -> enjoy the meal
enjoy the meal -> choose payment method
choose payment method -> XOR1
XOR1 -> (credit card is available) pay by credit card
XOR1 -> (credit card is unavailable) pay in cash
pay by credit card -> XOR2
pay in cash -> XOR2
XOR2 -> End

For the restaurant:
Start -> receive an order
receive an order -> DataObject(order list)
receive an order -> AND1
AND1 -> prepare the meal
AND1 -> prepare the tableware
prepare the meal -> AND2
prepare the tableware -> AND2
AND2 -> serve the meal
serve the meal -> ask the customer to pay for the order
ask the customer to pay for the order -> confirm the payment
confirm the payment -> TextAnnotation(provide the receipt if the customer needs)
confirm the payment -> End

"Procedural Document":
In the beginning, the staff will receive an order request, and then checks the order type. If the order is standard type, the sufficience of the stock is checked according to the stock table. If the order is special type, upload the order to the factory system. If the stock is sufficient for standard order, the goods will be directly shipped out, else if the stock is insufficient, they will need to be transferred from other warehouses. After that, the staff updates the order status and provide order information to the user. At the same time, the staff needs to bind order information to user account. Finally, the staff record the request status and the procedure ends.

"Procedural Graph":

For the staff:
Start -> receive an order request

receive an order request -> check the order type
check the order type -> XOR1
XOR1 -> (the order is standard type) check the sufficience of the stock
XOR1 -> (the order is special type) upload the order to the factory system
check the sufficience of the stock -> DataObject(the stock table)
check the sufficience of the stock -> XOR2
XOR2 -> (the stock is sufficient) directly shipped out the goods
XOR2 -> (the stock is insufficient) transfer the goods from other warehouses
directly shipped out the goods -> XOR3
transfer the goods from other warehouses -> XOR3
XOR3 -> XOR4
upload the order to the factory system -> XOR4
XOR4 -> AND1
AND1 -> update the order status
update the order status -> provide order information to the user
AND1 -> bind order information to user account
provide order information to the user -> AND2
bind order information to user account -> AND2
AND2 -> record the request status
record the request status -> End

"Procedural Document":
Start the service by receiving the email from the electronic mailbox, then parse the email content. If the email contains account query request, reply the account information to the user. If the email contains account modification request, record the information needs to be modified. After that, verify the validity of the account and verify the legality of the modified information at the same time if there exists account information to be modified. Otherwise update the verification timestamp of the account directly. Finally, synchronize the email content to the system and the procedure ends.

"Procedural Graph":

For the process:
Start -> receive the email
receive the email -> DataObject(electronic mailbox)
receive the email -> parse the email content
parse the email content -> OR1
OR1 -> (the email contains account query request) reply the account information to the user
OR1 -> (the email contains account modification request) record the information needs to be modified
reply the account information to the user -> OR2
record the information needs to be modified -> OR2
OR2 -> XOR1
XOR1 -> (there exists account information to be modified) AND1
XOR1 -> (otherwise) update the verification timestamp of the account directly
AND1 -> verify the validity of the account
AND1 -> verify the legality of the modified information
verify the validity of the account -> AND2
verify the legality of the modified information -> AND2
AND2 -> XOR2
update the verification timestamp of the account directly -> XOR2
XOR2 -> synchronize the email content to the system
synchronize the email content to the system -> End

Procedural document: ***{procedural_document}***

D.2 Structure Check Prompt

Structure Check Prompt

You are **Structure Checker**, an expert in reviewing procedural graphs.
Your task is to review the Procedural Graph generated by **GraphBuilder** and provide constructive feedback for refinement.

DEFINITIONS and RULES:
{extracted_rules}

INPUT:
1. The generated Procedural Graph: ***{generated_graph}***.
2. The original Procedural Document: ***{procedural_document}***.
3. The Structure Issues from simulator, which lists structural errors or issues detected in the graph: ***{structure_issues}***.

Structure Feedback Check:
- For each *Structure Issue*, carefully check whether it is a real structural problem in the graph.
- If the *Structure Issue* is correct, provide a clear, actionable suggestion for how to modify the graph to fix the issue. Prioritize suggestions that add or reconnect nodes/edges, rather than deleting, unless deletion is absolutely necessary.
- If you believe a structure-reported issue is not actually a problem, briefly explain why and state that no action is needed for that issue.
- If the issue is due to missing information in the document, you may suggest minimal additions to ensure graph connectivity, but avoid inventing unrelated content.
- Try to add reasonable nodes, edges, or conditions to the graph to fix the issues, but avoid deleting nodes or edges unless absolutely necessary.
- If the structural issue involves auxiliary nodes (such as **DataObject** or **TextAnnotation**), do **not remove** those nodes or their edges. Instead, **retain them for documentation purposes**, and **add a separate edge to reconnect the execution flow to a valid action or decision node**.

Important Note on Auxiliary Nodes:
- **DataObject** and **TextAnnotation** are auxiliary nodes used for annotation or reference. They are **not part of the executable process flow** and are **ignored in execution traces**.
- Edges **pointing to** these nodes (e.g., 'Action -> TextAnnotation') are allowed and useful for documentation. However, they **do not count as valid execution paths**.
- If a node only connects to auxiliary nodes, it is still considered a **dead end**. In such cases:
- **Add a new edge to the next action or decision**, **but keep the original auxiliary edge**.
- Do **not** delete the auxiliary node or its connection unless it is incorrect or redundant.
- **Do not** use 'DataObject' or 'TextAnnotation':
- As starting points in the graph ('DataObject -> Action' is invalid)
- As intermediates in loops, branches, or decisions
- The process flow must go through **Action**, **Gateway**, or **Condition** nodes only. Auxiliary nodes may appear in the graph, but only as **side annotations**, not as flow controllers.

OUTPUT:
If no issues:
APPROVED

If issues are present, use the following format for each, **don't** reply other information:

Issue N
- Problem: <copy or summarize the issue>
- Status: Confirmed / Not a real issue
- Suggestion (if confirmed): <how to fix, ideally by reconnecting, adding or splitting nodes>
- Explanation (if not a real issue): <short justification>

D.3 Logic Check Prompt

Logic Check Prompt

I want you to check whether the gateway segment and its associated text are logically consistent.

Input:
1. text of gateway segment trace extracted from simulator: ***{gateway_trace_text}***
2. original document segment: ***{original_document}***

Gateway Identification Guidelines

1. XOR Gateway (Exclusive Gateway)
- Use XOR when only **one** of the possible paths can be taken — the conditions are **mutually exclusive**.
- Typical linguistic signals:
- "if..., else if..."
- "if..., however, if..."
- "if..., on the other hand, if..."
- "either A or B"
- "choose one of the following options"
- "if A, skip B"
- "otherwise" / "else"
2. OR Gateway (Inclusive Gateway)
- Use OR when **one or more** of the paths may be taken independently or together.
- Typical linguistic signals: - "if..., if..., if..." (without "else" or "otherwise")
- "also, if..."
- "similarly, if..."
- "you may also..."
- "choose one or more..."
- "any combination of the following"
3. AND Gateway
- Use AND when **all** following actions must happen, either simultaneously or sequentially.
- Typical linguistic signals:
- "at the same time..."
- "meanwhile..."
- "in parallel..."
- "do both A and B"
- "must also do..."
- "simultaneously perform..."

Here are some examples:

Input:
OR1: If there is no information about the old supplier, then check the deadline of 4 business days. Otherwise, continue to do the check.

Output:
OR1: If there is no information about the old supplier, then check the deadline of 4 business days. Otherwise, continue to do the check.
- Status: wrong.
- Revision suggestion: Change OR1 to XOR1.
- Explanation: The phrase uses a classic XOR pattern — "if..., otherwise...". These are mutually exclusive conditions: either there is no information about the old supplier, or there is. Only one branch is taken at a time, so this logic requires an XOR, not an OR.

Now, I need you to check the following gateways and their corresponding text segments in the Procedural Graph
Only output the result block if the status is wrong, otherwise, respond with "APPROVED".

Only output the result block for the current input using the following for-

mat:
<Gateway Name>: <text from the document that corresponds to the gateway>
- Status: <status of the gateway, correct or wrong>
- Revision Suggestion: <suggestion to fix the issue, if any>
- Explanation: <explanation of the status, why it is correct or wrong>

D.4 Graph Refine Prompt

Graph Refine Prompt

The Procedural Graph contains the following types of "Nodes" and "Flows":

"Nodes":
"Start": start node indicates the start of a procedure, represented as "Start".
"End": end node indicates the ending of a procedure, represented as "End".
"Action": action node indicates a specific step in a procedure, represented as the step itself, such as "prepare the ingredients".
"XOR": exclusive gateway, indicates that only one of the following non-sequential actions can be executed, distinguish by numbers, such as "XOR1".
"OR": inclusive gateway, indicates that one or more of the following non-sequential actions can be executed, distinguish by numbers, such as "OR1".
"AND": parallel gateway, indicates that all of the following actions should be executed in parallel, distinguish by numbers, such as "AND1".
"DataObject": DataObject indicates the constraints for the necessary data of the actions, represented as "DataObject(data object)".
"TextAnnotation": TextAnnotation indicates essential notices need to be considered for the execution of the actions, represented as "TextAnnotation(essential notices)".

"Flows":
"SequenceFlow": flow that represents the execution of sequential actions, such as "Start -> prepare the ingredients".
"ConditionFlow": the condition flow is used to indicate that the following action is performed under the condition on the Condition Flow, such as "XOR1 -> (condition1) choose the first one".
"ConstraintFlow": flow that is used to connect the constraints with corresponding actions, such as "prepare the ingredients -> TextAnnotation(essential notices)".

In addition, the actor of corresponding actions is put in the front of corresponding elements to indicate the actor of the following actions if needed, such as "For <actor-name>:". If there is no specific actor mentioned, use "For the process" to indicate the actor of the following actions.

You should generate the graph in the format of "Node -> Node" line by line until generating the whole graph for the given Procedural Document, and keep the text of the nodes and conditions consistent with the original Procedural Document.

Here are some examples: ***{few_shot_examples}***

Previously, another model generated a Procedural Graph, and we have identified several structure issues with it. Please use the list of detected issues and solution suggestions as **references to avoid repeating the same mistakes**.
Don't copy the previously generated Procedural Graph, but use it as a reference to generate a new Procedural Graph that is more accurate and complete.

"Previously Generated Procedural Graph": ***{generated_graph}***

"Detected Issues and Solution Suggestions" (you may meet these issues, just refer them as references if available): ***{issues_and_revisions}***

Now you need to generate the corresponding Procedural Graph of the following Procedural Document, if there is no specific actor mentioned, use "For the process" to indicate the actor of the following actions:

"Procedural Document": ***{procedural_document}***