Incremental Synchronization of BPMN Models and Documentations by Leveraging Structural Algorithms and LLMs

David Cremer[1], Benjamin Dalmas[2], Quentin Nivon[1], and Gwen Salaün[1]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG 38000 Grenoble, France
{david.cremer,quentin.nivon,gwen.salaun}@inria.fr
[2] iGrafx, 35510 Cesson-Sévigné, France
benjamin.dalmas@igrafx.com

Abstract. Maintaining consistency between business process models and their textual descriptions is critical for operational clarity, compliance, and communication. However, as process models evolve, updating documentation remains costly and error-prone. Existing methods often require manual rewriting or full text regeneration, discarding valuable domain-specific language. This paper presents an edit-based synchronization approach that incrementally updates textual descriptions to reflect changes in Business Process Model and Notation (BPMN) diagrams while preserving unaffected content. We propose two complementary algorithms: the Longest Common Execution Subsequence (LCES) approach for balanced acyclic models, and a heuristic beam search for more complex structures with loops and unbalanced gateways. The resulting transformation steps are translated into structured prompts guiding a large language model to produce minimal, style-consistent revisions. A prototype system demonstrates high semantic accuracy and stylistic coherence across diverse process evolution scenarios.

Keywords: Process Modeling · Business Process Management (BPM) · Text Synchronization · Large Language Models (LLMs).

1 Introduction

Business processes prescribe how work unfolds within and across organizational boundaries. Expressed in tasks, decisions, and control-flow dependencies, they encode operational intent, compliance obligations, and institutional knowledge that guide day-to-day execution. Making these processes explicit and comprehensible is therefore vital for optimisation, regulatory assurance, and cross-functional alignment.

Organizations typically capture a process in two complementary ways: (i) a *graphical model* such as a Business Process Model and Notation (BPMN) diagram, which lends itself to simulation and automated execution [9, 4], and (ii) a *textual description* that narrates procedures, exceptions, and rationale in natural language, which remains indispensable for non-technical stakeholders [16,

6]. Maintaining both artifacts in lock-step is notoriously difficult. Models evolve quickly as regulations or business needs change, while the accompanying text lags behind, leading to inconsistencies that cause errors and erode trust [1, 11].

Current practices rely on analysts manually updating the prose whenever a model changes—a costly and error-prone activity that does not scale to repositories with hundreds of processes [25]. Full regeneration of the text with natural language generation techniques is a tempting alternative, but tends to discard domain-specific phrasing and validated narrative that users value [10]. We argue that *synchronization rather than regeneration* is the more effective paradigm: update only the fragments of prose affected by structural edits in the BPMN while preserving everything else. Realizing this vision poses two technical challenges: (1) detecting and representing model edits in a suitable form allowing textual revision, and (2) mapping those edits to minimal, stylistically coherent changes in the documentation.

To tackle these challenges, we present an automated approach that treats process-text consistency as an *edit-based* synchronization problem. Specifically, we combine graph-based model differencing with prompt-guided editing by large language models (LLMs). For balanced acyclic models, we introduce the *Longest Common Execution Subsequence* (LCES) algorithm to isolate shared control-flow, while for processes containing loops or unbalanced gateways, we employ a custom beam-search heuristic that explores plausible sequences of insertions, deletions, and refinements. The detected edits are then translated into structured prompts that steer an LLM to revise only the relevant sentences, thus preserving validated language and narrative voice. Experiments indicate that this approach can produce text that remains semantically aligned with updated processes and stylistically consistent with original descriptions.

Contributions and Paper Outline. The contribution of this paper is threefold: (i) it proposes an edit-based synchronization approach for aligning BPMN models with their textual descriptions, (ii) it introduces algorithms for structured differencing, specifically the Longest Common Execution Subsequence algorithm and a beam search heuristic, and (iii) it develops a prompt design strategy that enables minimal, style-preserving edits using large language models.

The paper is structured as follows. Section 2 reviews BPMN and language models. Section 3 overviews the approach. Section 4 details the algorithms and the prompting. Section 5 describes the implementation and experiments. Section 6 discusses related work, and Section 7 concludes the paper.

2 Preliminaries

BPMN and Graph Representation. Business Process Model and Notation 2.0 (BPMN) [13] is a widely used standard for modeling business processes. This work focuses on activity diagrams that describe control flow through a combination of events, tasks, gateways, and sequence flows. Specifically, the node types *start event* and *end event* delimit the execution, *tasks* represent atomic activities that have exactly one incoming and one outgoing flow, and *gateways*

define branching and merging points in the process. Gateways with one incoming flow and multiple outgoing flows are called *splits*, for example a split parallel gateway that initiates concurrent branches. Gateways with multiple incoming flows and one outgoing flow are called *merges*, which synchronize or join parallel or alternative paths. A *sequence flow* connects two nodes that are executed one after the other in a specific order.

To support analysis and transformation, BPMN models are often represented as directed, vertex-labeled graphs that capture structural and control-flow relationships between process elements. Formally, a process is modeled as a graph $G = (V, E, \lambda, \theta, v_{\text{start}}, v_{\text{end}})$, where V denotes the set of nodes, E the set of directed edges corresponding to sequence flows, λ assigns a unique name label to each node, and θ specifies the node type (start, end, task, exclusive gateway, or parallel gateway). Each process is assumed to have exactly one start event v_{start} and one end event v_{end}, forming a single-entry, single-exit structure. While BPMN permits multiple start or end events, these can be normalized by introducing synthetic gateways without affecting the semantics.

A BPMN process is considered *balanced* when all branches that split at a gateway rejoin at a single, unique merge point. This allows for clear sequencing and nesting of control flow. In contrast, an *unbalanced* process allows branches to merge at different points, lacking a unique correspondence between splits and merges.

Large Language Models. Large Language Models (LLMs) are pre-trained neural networks capable of understanding and generating human-like text. They are widely used in natural language processing tasks such as summarization, question answering, text classification, and structured data extraction. In this work, we rely on the GPT-4.1 model—developed by OpenAI—as an example of a state-of-the-art LLM. GPT, which stands for *Generative Pre-trained Transformer*, is based on a transformer architecture and operates through prompt-based interactions, where natural language prompts specify the task or question to be addressed. The model generates coherent and contextually appropriate text completions by predicting the most likely continuation of a prompt.

3 Approach Overview

Our method incrementally updates textual process descriptions to reflect changes in BPMN models while preserving the original wording and structure as much as possible. The *key idea* is to avoid regenerating the entire description from scratch. Instead, we progressively transform the initial BPMN model into the modified version through a sequence of *small, structured steps*. Each step captures a localized change—such as adding or removing a task, changing a gateway, or reordering elements—and directly guides how the text is revised.

This approach addresses two core challenges. The first is to decompose the overall model change into an *ordered sequence of discrete edits* that can be applied incrementally. Rather than computing a flat diff, we construct a transformation path that starts from the initial model and progressively produces the

target model. The second challenge is to ensure that each transformation step leads to a *local, targeted update of the text*. After each model edit, we prompt the language model to revise only the relevant part of the description, preserving the rest and maintaining stylistic consistency. The overall pipeline consists of three stages, recalled in Figure 1.

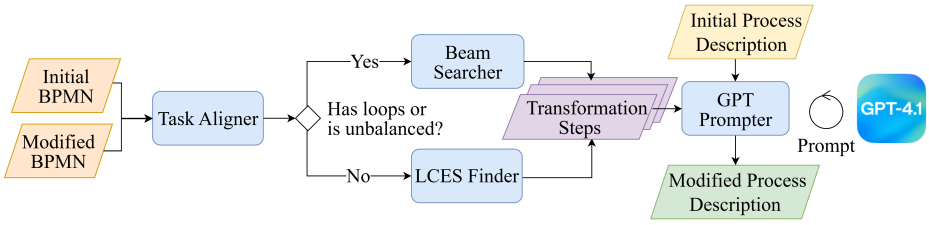

Fig. 1: High level overview of the pipeline

Stage 1: Identifying Additions and Deletions. We first detect tasks that have been added or removed between the initial and modified BPMN models. These are recorded explicitly as transformation steps and separated from structural changes affecting shared tasks. This ensures that later alignment focuses only on comparable parts of the model.

Stage 2: Computing a Transformation Sequence. For the remaining structure, where the set of tasks is shared between both models and only the control flow differs, we derive a sequence of edits needed to convert the initial model into the modified one. If the models are acyclic and balanced, we apply a *Longest Common Execution Subsequence (LCES)* algorithm that generalizes sequence alignment to process graphs and produces efficient, deterministic edit sequences. If the models are more complex—containing loops or unbalanced control flow—we instead use a *heuristic beam search* that explores alternative edit sequences and selects promising candidates based on structural similarity. In both cases, the output is an ordered list of transformation steps.

Stage 3: Incremental Text Update. Finally, each transformation step is applied in order. For each step, a targeted prompt is generated to guide the language model in revising only the corresponding part of the text. Because each prompt is limited in scope, the process achieves *edit-based textual synchronization*: it updates what has changed while preserving the rest. This stepwise, localized approach enables minimal, structure-aware revisions that maintain clarity and continuity in process documentation.

4 Process Model Differencing and Text Revision

4.1 Task Alignment

The first stage of the approach establishes a correspondence between the tasks of the initial and the modified BPMN models. This step isolates changes to the set of tasks themselves, separately from changes in how the tasks are connected or

ordered. Specifically, we compare the set of tasks in each model by their labels: tasks that appear only in the modified model are marked as *insertions*, while tasks that appear only in the initial model are marked as *deletions*. This interpretation is natural, since any task present in one version but absent in the other must have been added or removed during modification. After identifying these differences, we produce reduced models that retain only the shared tasks. This allows the following steps to focus exclusively on how the order and connections between tasks have changed, without being affected by tasks that no longer exist or have newly appeared.

4.2 Longest Common Execution Subsequence Approach

The first component of our approach for computing transformation steps is the *Longest Common Execution Subsequence* (LCES). This stage operates on the reduced versions of the models that include only the tasks present in both process definitions. The main goal of LCES is to systematically derive the edits needed to transform the reduced initial model into the reduced modified model. It does so by identifying a shared intermediate model that captures the parts of the process remaining unchanged in their relative order. From this common structure, the necessary deletions, insertions, and control-flow adjustments can be precisely determined.

More precisely, the LCES captures the largest subsequence of tasks and gateways that appear in the same relative order in both models and can be obtained by only deleting elements. When both BPMN models are *acyclic* and *balanced*, this approach yields a well-defined, terminating sequence of edits that incrementally transforms the initial model into the modified one. From this shared core, the remaining edits can be precisely determined: deletions of elements unique to the initial model, additions of elements unique to the modified model, and any necessary adjustments to gateway types to preserve structural consistency.

A key property of this approach is that all intermediate models produced during the transformation remain semantically correct and connected. This ensures that each step corresponds to a coherent and interpretable change, supporting clear and localized updates of the textual description. LCES requires that each split gateway has a corresponding merge, which is why we can only apply it to balanced models.

To increase the amount of elements in the LCES, we also apply a reduction step that removes structurally redundant gateways, such as nested splits of the same type or gateways without any branching behavior. While optional, this preprocessing often increases the number of the shared elements in the intermediate model and simplifies the resulting transformation sequence.

Common Execution Subsequence. A *common execution subsequence* (CES) is a set of elements occurring in both models in the same relative order, even if other elements are interleaved. In a sequential process, this corresponds to the longest list of tasks appearing in the same order. In models with branching and parallel flows, a CES must also respect partial orders and structural constraints.

Formally, a CES is a set of matched pairs of semantically equivalent nodes that preserve execution order and control-flow consistency. Two nodes are equivalent if they represent the same process role (start or end event), tasks with identical labels, or gateways with the same control-flow function (i.e., both splits or both merges). A CES aligns such nodes so that each node is matched at most once, the execution order remains consistent without introducing reversals, all control-flow relationships are respected, and all matched elements belong to corresponding branches in both models. This means that whenever a region of parallel or alternative paths is encountered, either the entire region is matched—including the split and merge gateways—or any matched nodes inside it must all come from a single branch. In other words, a CES never mixes elements from different branches within the same split-merge block.

LCES Computation. The Longest Common Execution Sequence (LCES) algorithm computes the CES containing the largest number of matched elements. It generalizes the classical longest common subsequence problem [12] to process models represented as directed acyclic graphs (DAGs).

At each pair of nodes, the algorithm explores three possibilities: (1) match the nodes if they are semantically equivalent (line 2), (2) advance in the first model by skipping the current node (line 8), or (3) advance in the second model (line 9). If the nodes are not equivalent, the algorithm simply skips one of them; if they do match, it considers both matching and skipping to find the longest possible alignment. In the matching case, it either continues the LCES computation recursively with the successor nodes in both models (line 3), or, if both nodes are split gateways, it applies a specialized flattening procedure to align their outgoing branches (lines 5-6).

In this case, the algorithm recursively computes an LCES for each pair of outgoing branches—one from each model—resulting in a set of candidate branch alignments. However, these alignments cannot be merged naively for two reasons. First, alignments that do not include the corresponding merge gateway are discarded, as they match elements that remain within the split-merge region in only one of the models. Second, combining alignments that reuse the same branch multiple times would mix activities from distinct paths and violate the control-flow structure. To prevent this, the algorithm selects a subset of branch alignments that are *pairwise disjoint*, meaning that each branch from each model is used at most once. These selected alignments are then concatenated into a single coherent, *flattened* region that provides a semantically consistent and non-overlapping correspondence between the matched split gateways and their respective merge gateways.

Including a matched pair does not always yield the longest sequence. Sometimes, it is better to skip a potential match at the current position in order to align more structurally significant elements further downstream. For instance, the algorithm may choose to skip matching a split gateway now so that it can be aligned with a more appropriate corresponding split gateway later in the other model—resulting in a longer common execution sequence. To account for such cases, the recurrence always considers all three possibilities at each step and

retains only those continuations that achieve maximal length (lines 11-12). Algorithm 1 illustrates how the LCES is computed recursively over pairs of nodes:

Algorithm 1 LCES(v, v')

```
1: C_1 ← ∅
2: if Match(v, v') then                                   ▷ Option 1: match nodes
3:     R ← {LCES(u, u') | u ∈ Succ(v), u' ∈ Succ(v')}     ▷ Recursively extend match
4:     C_1 ← {[(v, v')] ∪ A | A ∈ R}                      ▷ Non-flattened alignments
5:     if IsSplit(v) and IsSplit(v') then
6:         C_1 ← C_1 ∪ {[(v, v')] ∪ A | A ∈ Flatten(R)}   ▷ Flattened alignments
7:     end if
8: end if
9: C_2 ← {LCES(u, v') | u ∈ Succ(v)}                      ▷ Option 2: skip v
10: C_3 ← {LCES(v, u') | u' ∈ Succ(v')}                   ▷ Option 3: skip v'
11: C ← C_1 ∪ C_2 ∪ C_3
12: return sequences in C with maximal length
```

Example. To illustrate the computation, we consider the two BPMN models shown in Figure 2. The algorithm starts by matching the end events (◯) and the tasks `H` and `G`. After that, the models diverge: the initial model continues with task `F`, while the modified model reaches a merge gateway (◈). Since these elements do not match, the algorithm explores both options—skipping ahead in either model—to find the next possible alignment. In both cases, it arrives at a point where the control flow diverges, making linear continuation impossible. As a result, the algorithm enumerates all feasible combinations of branches between the corresponding split and merge gateways in the two models. Stopping at the split gateway, this yields the following six candidate execution subsequences:

$\pi_{11} = \langle (A, A),\ (B, B),\ (◈, ◈),\ (G, G),\ (H, H),\ (◯, ◯) \rangle$
$\pi_{12} = \langle (◈, ◈),\ (G, G),\ (H, H),\ (◯, ◯) \rangle$
$\pi_{13} = \{\langle (F, F),\ (G, G),\ (H, H),\ (◯, ◯) \rangle, \langle (◈, ◈),\ (G, G),\ (H, H),\ (◯, ◯) \rangle\}.$
$\pi_{21} = \langle (C, C),\ (E, E),\ (◈, ◈),\ (G, G),\ (H, H),\ (◯, ◯) \rangle$
$\pi_{22} = \langle (D, D),\ (◈, ◈),\ (G, G),\ (H, H),\ (◯, ◯) \rangle.$
$\pi_{23} = \{\langle (F, F),\ (G, G),\ (H, H),\ (◯, ◯) \rangle, \langle (◈, ◈),\ (G, G),\ (H, H),\ (◯, ◯) \rangle\}.$

Once all candidate subsequences are collected, the algorithm attempts to flatten them. Among the candidates, π_{13} and π_{23} each contain two alternative

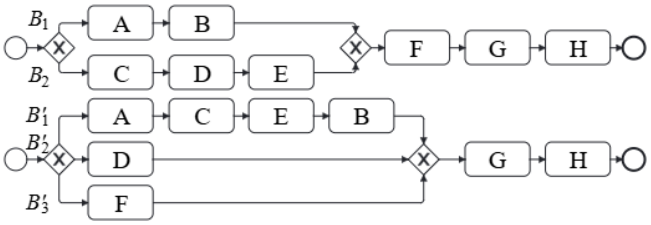

Fig. 2: Initial (top) and modified (bottom) BPMN models

subsequences. One of these includes a match for task F but omits the corresponding merge gateway. This indicates that task F lies within the split-merge region in one model but outside of it in the other. Since flattening only applies to matches fully enclosed by corresponding split and merge gateways in both models, these subsequences are excluded. Each of the valid candidate sequences is split into a prefix (before the merge) and a suffix (after the merge). Naively combining all prefixes would yield the union of tasks A, B, C, D, and E. However, this combination is structurally invalid: in the upper model, A and B belong to one branch, while C and E belong to another; in the lower model, all four tasks appear in the same branch. Simply deleting tasks from either model would not produce a consistent shared structure. Including all these tasks would require matching branch B_1 in the upper model to branch B'_1 (to align A and B) and also matching branch B_2 to the same branch B'_1 (to align C and E)—violating the constraint that each branch may only be used once. To resolve this, the algorithm selects the largest subset of matches that satisfies this restriction. The largest valid combination pairs branch B_1 with B'_1, contributing A and B, and branch B_2 with B'_2, contributing D. This defines the flattened region of consistently aligned tasks inside the split-merge block. From the merge gateway onward, the algorithm resumes linear matching, appending the suffix that includes G, H, and the end event. The resulting Longest Common Execution Subsequence is:

$\langle$(◯, ◯), (◈, ◈), (A, A), (B, B), (D, D), (◈, ◈), (G, G), (H, H), (◯, ◯)$\rangle$

As a consequence, tasks C, E, and F are excluded from the shared sequence. They must be deleted from their original positions and re-inserted at their new locations to transform the initial model into the modified one. This example shows how the algorithm maximizes alignment while preserving control-flow consistency and avoiding structurally invalid branch combinations.

4.3 Beam Search Approach

Beam search is used when BPMN models contain loops or unbalanced control-flow structures where split gateways no longer have a unique corresponding merge. In such cases, cycles and asymmetric flows require a more flexible search that incrementally explores alternative transformation paths without exhaustively enumerating all possibilities. Instead, beam search maintains a bounded set of promising candidates in parallel, extending each by applying domain-specific edit operations.

Each candidate represents a partial transformation path incrementally extended by applying domain-specific edit operations. Candidates are evaluated using a cost function that estimates how closely their control-flow structure matches the target. Only the most promising candidates are retained and further expanded, ensuring that the search remains computationally feasible while progressively approaching the target model. If no valid transformation path is found within a predefined time or resource budget, the system falls back to regenerating the process description from scratch, guaranteeing a usable result in all cases.

Unlike in the LCES approach, intermediate models produced by beam search are not guaranteed to be semantically correct BPMN models at every step. Structurally relaxed constructs—such as gateways with only one incoming and one outgoing flow, or inconsistently nested branches—are temporarily permitted to increase flexibility. These inconsistencies are corrected by the end of the transformation sequence.

Edit Operations. Beam search explores alternative models by applying domain-specific edit operations. The primary operation is *task movement*, which implicitly covers both removal and reinsertion. To move a task, the algorithm either bypasses it—connecting its predecessor and successor—or, if it lies between gateways, removes it with adjacent flows while preserving connectivity. The task is then reinserted either along any sequence flow or between a gateway (not a merge) and another gateway (not a split), ensuring clear and unambiguous control flow. Beyond task movement, sequence flows can be inserted between gateways, provided that each retains a consistent role as a split or a merge. Flows may be deleted if all nodes remain reachable from the start event. Gateways can also be added along sequence flows or removed when they act as structural pass-throughs (i.e., with exactly one incoming and one outgoing flow).

Cost Model. At each iteration, candidates are evaluated using a cost function that combines the cumulative cost of edits applied so far with a heuristic estimate of the remaining difference to the target BPMN model. The search strategy uses a classical formulation:

$$f(n) = g(n) + h(n)$$

where $f(n)$ is the estimated total cost of transforming candidate n. The term $g(n)$ denotes the cumulative cost of applied edits, and $h(n)$ is the heuristic component that guides the search toward the target model. This work adopts a uniform cost model for $g(n)$, where each operation contributes the same fixed cost. Although weighting edits differently could reflect their semantic impact, experiments showed that such weighting often distorts the search (for instance, heavily penalizing task movement can lead to excessive gateway insertions).

Structural Edge Abstraction and Heuristic. To robustly compare candidates, each sequence flow is abstracted as a *Structural Edge* that encodes the role and local structure of its source and target nodes. For tasks and events, this abstraction combines their type and label (e.g., distinguishing task `Approve Request` from task `Review Application`). For gateways, which often lack unique labels, it includes the gateway's type and a canonical hash summarizing its adjacent incoming and outgoing flows. Two Structural Edges are considered equal if both their source and target representations match exactly, ensuring that only flows occupying comparable positions in the process are treated as equivalent. The heuristic $h(n)$ quantifies the remaining difference between a candidate and the target model:

$$h(n) = |E_T| + 0.5 \cdot |E_n \setminus E_T| - |E_n \cap E_T|$$

where E_n and E_T are the sets of Structural Edges in the candidate and target, respectively. This scoring rewards matching edges, penalizes extra edges moderately, and penalizes missing edges more strongly. The asymmetry reflects that surplus edges often arise as temporary artifacts during construction and can be removed later, whereas missing edges indicate that parts of the target process are still absent. Although the heuristic is not necessarily admissible, beam search does not require admissibility and instead prioritizes efficient exploration. By comparing control flow at the level of individual edges and their local context, this approach provides fine-grained, practical guidance without relying on global graph similarity measures.

Practical Considerations. While beam search effectively captures a wide range of structural edits and produces good results in most cases, its performance can be less responsive when building deeply nested gateway structures. The structural-edge heuristic provides strong guidance for localized changes and incremental refinements but may only recognize progress in complex regions once sufficient surrounding context has been assembled. To maintain tractability, the search operates within configurable timeouts. If no valid transformation is identified within the allotted time, the method reliably falls back to generating a new process description, ensuring that an updated result is always produced. Finally, the beam width governs the balance between completeness and computational cost: wider beams can improve reconstruction success rates while requiring additional resources. In this work, the beam width is selected empirically.

4.4 GPT-Based Prompting Strategy

After computing a transformation path between the initial and modified BPMN models—using either the LCES approach or the beam search heuristic—the final step is to update the corresponding process description. Rather than regenerating the entire text, the method applies targeted, incremental edits that preserve the original wording, tone, and structure. Each structural change—such as inserting, removing, or reordering a task—is treated as a self-contained transformation. For each change, a prompt is constructed comprising several well-defined components:

Role Framing. The prompt begins by instructing the model to act as a BPMN process modeling expert. Research has shown that role-based prompting can improve the relevance and task alignment of large language model outputs [14]. In this context, the role is intended to steer the model toward domain-appropriate language and interpretations.

Change Description. Each prompt includes a short natural language sentence specifying the structural modification applied to the process model. This contextualizes the transformation for the language model, allowing it to focus on the relevant part of the process without relying solely on structural inference from the graphs.

Update Instructions. The model is instructed to revise the process description precisely and minimally, based on the structural change and adjacency lists.

It is guided to reason about the main flow using forward adjacency, and about prerequisites using backward adjacency. The style of the original text should be preserved, with each change integrated naturally and without redundancy. The output must consist only of the updated full description.

Full-Context Editing. The full process description is provided in each prompt to convey the original writing style and structure, allowing the model to locate and revise the relevant section implicitly without explicit extraction or annotation.

Graph Inputs and Format. To limit prompt size and reduce irrelevant context, we extract a localized subgraph centered around the changed node rather than using the full BPMN model. This subgraph is defined through a two-phase traversal: a forward search from the changed node, its immediate predecessor, and its successor, stopping at task nodes or start/end events; and a backward search from each of these, again terminating at task or event nodes. A symmetric backward-then-forward traversal is also performed to capture additional dependencies. The union of all reachable task and event nodes defines the subgraph focus. From this set, we construct forward and backward adjacency lists that preserve the actual control-flow structure, including links to gateways even if those were not reached during traversal. Gateways are represented using structured objects that encode their control-flow semantics (e.g., parallel or exclusive branching), preserving the executable logic of the process model. The resulting structure is encoded as JSON and introduced to the model through an accompanying textual explanation of the format. This enables the model to interpret the control-flow graph consistently and reason over its structure in relation to the description.

Example. To illustrate this approach, we use a simplified university enrollment process where a new task, `Assign Case Officer`, is inserted after the application review. Figure 3 shows the BPMN models before and after this change.

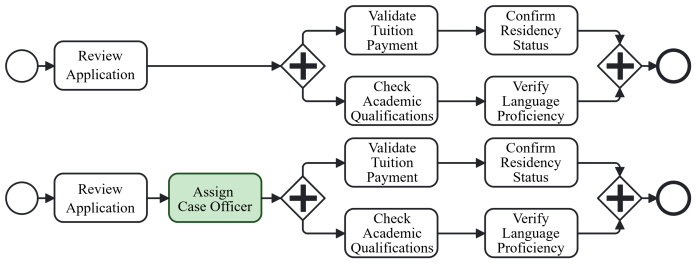

Fig. 3: BPMN models before and after inserting the `Assign Case Officer` task.

The complete prompt for this transformation is shown in Table 1, illustrating all components of the prompting strategy.

Table 1: Prompt for inserting the task `Assign Case Officer`.

Component	Prompt Content
Role Framing	`You are an expert in BPMN process modeling.`
Change Description	`In the modified version, the new task 'Assign Case Officer' has been added.`

Graph Format	`The process flow graphs are provided as JSON adjacency lists in two forms:`
	`- Forward adjacency: each node lists its possible next steps (successors).`
	`- Backward adjacency: each node lists its prerequisites (predecessors).`
	`Nodes may be connected by direct task names, or by objects indicating gateways or control flow, e.g.:`
	`{ "type": "parallel", "branches": [ ... ] } for parallel execution,`
	`{ "type": "exclusive", "branches": [ ... ] } for exclusive (XOR) choices.`
	`Interpret these structures according to their BPMN semantics.`

Update Instructions	`You are provided with:`
	`- An initial process description.`
	`- Two versions of the process flow graph (original and modified), each encoded as both a forward adjacency list and a backward adjacency list.`
	`Your task:`
	`- Update the process description so that it accurately and naturally reflects the addition and integration of the new task.`
	`- Carefully analyze the adjacency lists to determine how the new task affects the sequence, available choices, conditions, or outcomes in the process.`
	`- Use the forward adjacency lists to understand and describe the main process flow.`
	`- Use the backward adjacency lists to check prerequisites and ensure consistency.`
	`- Adapt the description in the style of the given process description, making the change as precise, short, and minimal as possible.`
	`- Ensure each dependency, condition, or sequence is described only once, and avoid repeating the same requirement in different ways. Combine all prerequisites for each step into a single statement.`
	`Answer only with the full, updated process description.`

Full-Context Editing	`The original description is: The process begins when the university reviews a student's application. The workflow then proceeds in parallel along two branches. In one branch, the student's academic qualifications are checked and their language proficiency is verified. In the other branch, the tuition payment is validated and the student's residency status is confirmed.`

Graph Inputs	`The initial process flow graph (forward adjacency): {`
	`'Review Application': [{ 'type': 'parallel', 'branches': ['Check Academic Qualifications', 'Validate Tuition Payment'] }],`
	`'Check Academic Qualifications': ['Verify Language Proficiency'],`
	`...}`
	`The initial process flow graph (backward adjacency): {`
	`'Review Application': ['Start Event'],`
	`'Check Academic Qualifications': ['Review Application'],`
	`...}`
	`The new process flow graph (forward adjacency): {`
	`'Review Application': ['Assign Case Officer'],`
	`'Assign Case Officer': [{ 'type': 'parallel', 'branches': ['Check Academic Qualifications', 'Validate Tuition Payment'] }],`
	`...}`
	`The new process flow graph (backward adjacency): {`
	`'Assign Case Officer': ['Review Application'],`
	`'Validate Tuition Payment': ['Assign Case Officer'],`
	`...}`

Applying the prompt yields the following output:

```
The process begins when the university reviews a student's application.
Next, a case officer is assigned to the application. The workflow then
proceeds in parallel along two branches. [...]
```

5 Tool and Experiments

5.1 Tool

To validate the proposed approach, we developed a prototype system that automatically updates textual process descriptions when a BPMN model changes. The tool takes two BPMN 2.0 XML models (original and modified) together with the initial description, and produces an updated text reflecting the structural differences. The transformation pipeline combines model parsing, structural comparison, and prompt-based text rewriting using GPT-4.1. Depending on the models' properties, it selects either a Longest Common Execution Subsequence algorithm or a beam search strategy to compute the transformation path. Each step generates a targeted prompt to ensure minimal deviation from the original wording unless structural edits are necessary.

The implementation comprises approximately 7k lines of Java code. The tool is available both as a standalone command-line application and as an integrated module within an existing web platform for BPMN generation from natural language descriptions [20]. The web interface, built with HTML, CSS, JavaScript, and `bpmn.io`, allows users to edit diagrams interactively and triggers updates with a single action. A Node.js backend relays the models and description to the Java engine, which returns the revised text for display.

The current prototype requires semantically valid BPMN models that include parallel and exclusive gateways, loops, a single start and end event, and uniquely named tasks.

5.2 Experiments

We evaluated our approach on a total of 50 BPMN models, derived from two representative base processes: one *unbalanced* and one *balanced*. Both models were adapted from existing processes in the literature [11, 5] and slightly modified for evaluation. The unbalanced model contains 16 tasks, 3 parallel gateways with nesting, and 2 exclusive gateways with nesting, while the balanced model contains 18 tasks, 1 parallel gateways, and 3 exclusive gateways with nesting. For each, we applied five structural transformations at each transformation length (from one to five edits), resulting in 25 test cases per category.

Transformations included task, sequence flow, and gateway additions or deletions, as well as task movements and gateway type changes. Each modified BPMN was paired with its original textual description to serve as input for the update task. We compared our method to a baseline using direct prompting with GPT-4.1. In the baseline, the model was given the role of a BPMN expert, and asked to revise the original description based on the modified and original BPMN models (provided in XML), making only minimal, style-preserving changes.

Output quality was evaluated manually along two dimensions. Semantic correctness was assessed by verifying whether the updated description accurately reflected the modified control flow; scores were binary (1 for correct, 0 for incorrect). Stylistic consistency was rated on a 0–2 scale, based on how well the output matched the tone, phrasing, and structure of the original text, where higher values indicate better alignment. Because both methods rely on GPT and generally produce fluent text, we applied a deliberately critical lens to judge whether edits were minimal and stylistically coherent. For each transformation length (1 to 5), scores were averaged across all corresponding test cases in both the balanced and unbalanced categories. Since semantic correctness reflects whether the generated description remains faithful to the process logic, it is considered the primary evaluation criterion.

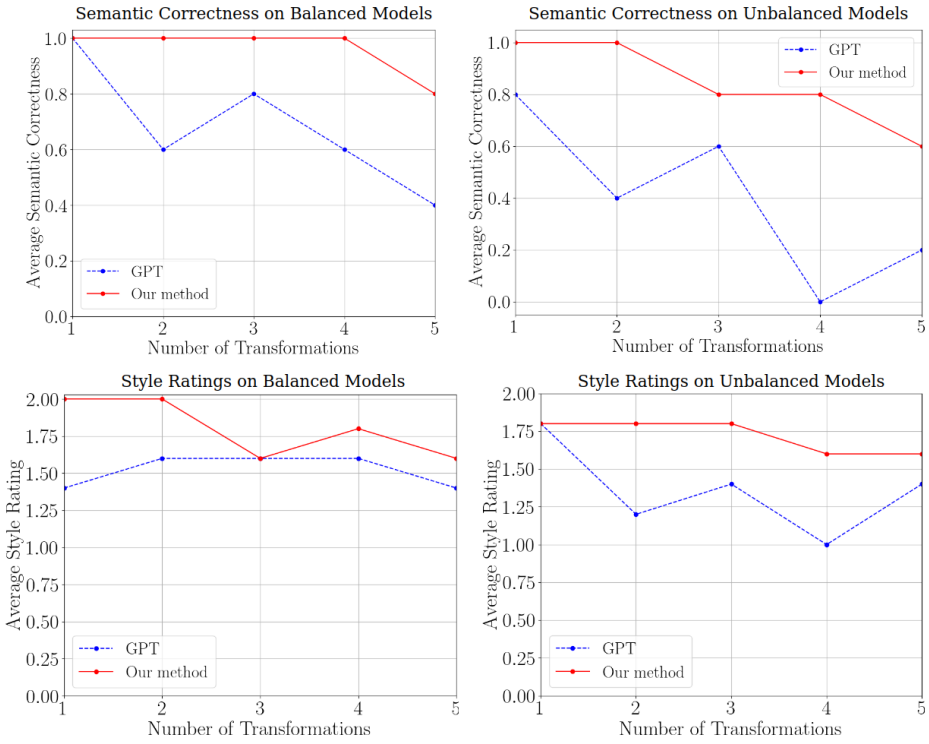

Fig. 4: Average semantic and stylistic scores by number of transformations.

Results and Discussion. Figure 4 shows average semantic and stylistic scores across transformation lengths for both model categories. Overall, our method consistently outperformed the GPT baseline in both metrics. In terms of semantic correctness, our method maintained high scores across all settings, with only a modest decline as the number of edits increased. For balanced models, performance remained particularly strong, likely due to their regular structure and clearer execution semantics. Unbalanced models showed slightly lower accuracy, but our method remained efficient even under more complex transformations.

In contrast, the GPT baseline degraded noticeably as transformations accumulated, especially for unbalanced models. Without intermediate guidance or structure-aware prompting, GPT frequently overlooked changes or misinterpreted updated control flow, resulting in omissions, misplaced tasks, or semantically incorrect descriptions. These issues became more common with larger edits. Our incremental synchronization strategy avoided such drift by isolating and addressing each structural change in sequence.

Stylistic consistency scores were generally high for both methods, as expected from GPT-based generation. However, our method achieved slightly higher ratings overall. The baseline more often introduced stylistic shifts not present in the original text, such as inserting generic phrases like "then the process branches" or adopting a more explanatory tone. Our method more reliably preserved the original style, although occasional artifacts appeared when the same task was revised multiple times, sometimes leading to repetition or awkward reordering. These differences, while subtle, were observed consistently and contributed to the improved stylistic performance of our approach.

Finally, it is worth noticing that GPT was used with its *temperature* parameter set to 0, ensuring determinism between identical runs.

Execution Time Analysis. To assess practical viability, we measured execution times across all test cases, distinguishing between the *search time* (i.e., model differencing), *prompting time* (LLM invocation), and the overall *total time*. Figure 5 presents the average execution times for balanced and unbalanced models, grouped by transformation length.

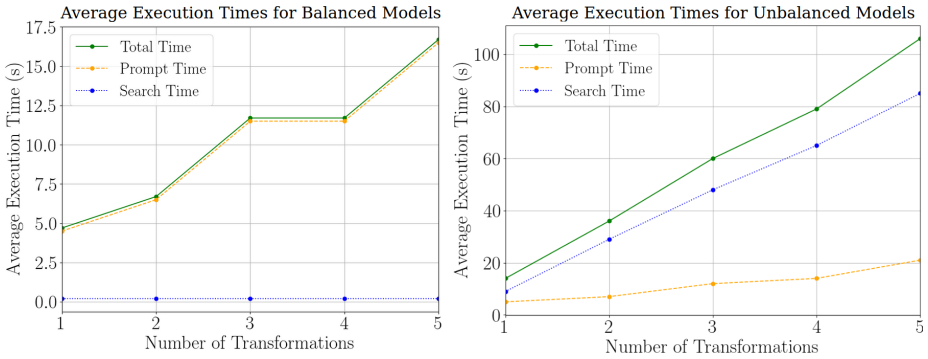

Fig. 5: Average Execution Times by Number of Transformations

For balanced models, search times remained negligible across all transformation lengths, as expected from the deterministic LCES algorithm. Most of the total time was spent on prompting, which grew roughly linearly with the number of transformations—an intuitive result given that each transformation triggers a separate prompt. In contrast, unbalanced models showed significantly higher execution times overall. Here, the search time (due to beam search) clearly dominated the total runtime and increased steadily with transformation length. Nevertheless, growth remained approximately linear, and even the most complex

scenarios (involving five transformations) were completed in under 120 seconds on average. These results highlight two key findings: (1) the LCES-based synchronization for balanced processes is extremely efficient, and (2) while beam search for unbalanced models is computationally more expensive, it consistently converges and remains tractable even for complex updates. This confirms the approach's feasibility for interactive settings and its efficiency under structural variability.

Threats to Validity. Although our evaluation setup is systematic and representative, it is not exhaustive. The selected base processes and transformation types capture a broad range of realistic structural edits, but do not cover all possible combinations. Moreover, the effectiveness of both our method and the baseline depend on the clarity and specificity of the original descriptions. Ambiguous or underspecified input may reduce output quality, particularly with regard to semantic correctness. That said, the test cases were carefully designed to reflect typical evolution scenarios, and the consistency of results across both balanced and unbalanced models supports the general usability of the findings.

6 Related Work

Several works aim at measuring the degree of similarity of business process models [8, 18, 7, 21]. [8] uses causal footprints as an abstract representation of the behaviour captured by a process model. Then, given two footprints, the similarity is computed using the vector space model approach from information filtering and retrieval. [18] relies on Petri nets and propose to compare process models by analysing event logs with typical execution sequences. [7] tackles the problem of retrieving process models from a repository that most closely resemble a given process model. [21] propose two measures of similarity between two versions of a BPMN process. The first one relies on the syntactic descriptions of the processes considered as input, whereas the second one focuses on their semantic models.

Prompt-based strategies have become increasingly central for adapting large language models (LLMs) to structured inputs such as code, tables, and diagrams. Approaches like Code2Seq [2] encode abstract syntax trees as paths for neural summarization, while few-shot prompting [17] shows that pretrained LLMs can generalize over knowledge graphs using carefully crafted textual prompts. More recently, multimodal pipelines such as GenFlowchart [3] have combined OCR-derived structure with prompt engineering to generate fluent descriptions of visual diagrams. These systems are typically optimized for single-pass generation over static inputs.

Graph-to-text generation produces fluent descriptions from structured graphs such as knowledge bases. Structure-aware Transformers [24] incorporate relative positional encodings to improve relational coherence but assume that the complete graph is available at generation time. Few-shot prompting [17] and instruction-tuned models like GraphGPT [22] demonstrate that LLMs can generalize structural knowledge in zero-shot settings. Surveys by Yuan and Färber [23] highlight that most systems focus on producing new text from scratch rather than maintaining continuity across revisions.

More domain-specific work has addressed BPMN and UML verbalization. For BPMN, Leopold et al. [15, 16] pioneered pipelines translating process fragments into sentences, while Azevedo et al. [4] proposed synchronisation frameworks linking diagrams and text but still regenerating full descriptions after each change. The Process-to-Text framework [10] produces high-level summaries of mined processes but does not aim for maintainable documentation. In UML, the authors of [6, 19] developed grammar-driven approaches that ensure traceability but rely on controlled vocabularies and lack support for incremental updates.

Work explicitly addressing incremental updates is limited. Zhu et al. [25] generate abstract Business Process Text Sketches from Conditional Process Trees using large language models, primarily to bootstrap datasets rather than maintain authored documentation; each change yields a new summary without preserving prior phrasing or structure. Azevedo et al.'s bidirectional framework [4] allows incremental edits from text to model but regenerates full descriptions when models change, limiting continuity. The Process-to-Text framework [10] supports time-sensitive summarization of evolving process logs but does not track sentence-level alignment in curated documentation.

Incremental, style-preserving documentation thus remains an open challenge. This paper addresses it by combining symbolic diffs with targeted LLM prompts to enable minimal, localized revisions to maintain coherence as models evolve.

7 Conclusion and Future Work

This work addressed the challenge of maintaining alignment between textual process descriptions and evolving BPMN models. As business processes change through insertions, deletions, or structural refinements, updating documentation accurately while preserving style and validated language becomes critical. We introduced a structured methodology for deriving transformation steps—a minimal sequence of edits mapping the differences between process models—and developed two complementary strategies to compute them: the Longest Common Execution Subsequence (LCES) for balanced acyclic models and a heuristic Beam Search approach for more complex structures. A GPT-based prompting framework incrementally applies each edit, enabling targeted updates rather than wholesale regeneration. Experiments showed that this approach produces semantically correct, stylistically coherent updates while significantly reducing manual effort.

Future work could enhance the system along several dimensions. Fine-tuning the language model on real-world process documentation may improve fluency, terminology, and the ability to accurately reflect control-flow semantics. Richer heuristics—such as graph embeddings or BPMN-specific similarity measures—could help the Beam Search to more effectively identify meaningful transformations. Additionally, supporting group-level edits over entire branches or subprocesses and recognizing common BPMN patterns like loops and decision points could enable more concise transformation sequences and tailored prompting strategies.

References

1. Van der Aa, H., Leopold, H., Reijers, H.A.: Detecting inconsistencies between process models and textual descriptions. In: Proc. of BPM'15 (2015)
2. Alon, U., Brody, S., Levy, O., Yahav, E.: code2seq: Generating sequences from structured representations of code (2019)
3. Arbaz, A., Fan, H., Ding, J., Qiu, M., Feng, Y.: Genflowchart: Parsing and understanding flowchart using generative AI. In: Knowl. Sci., Engin. Manag. (2024)
4. Azevedo, L.G., de Almeida Rodrigues, R., Revoredo, K.: BPMN model and text instructions automatic synchronization. In: Proc. of ICEIS'18 (2018)
5. Bellan, P., et al.: Pet: An annotated dataset for process extraction from natural language text tasks. In: Business Process Management Workshops (2023)
6. Burden, H., Heldal, R.: Natural language generation from class diagrams. In: Proc. of MoDeVVa'11 (2011)
7. Dijkman, R.M., Dumas, M., van Dongen, B.F., Käärik, R., Mendling, J.: Similarity of business process models: Metrics and evaluation. Inf. Syst. **36**(2), 498–516 (2011)
8. van Dongen, B.F., Dijkman, R.M., Mendling, J.: Measuring Similarity between Business Process Models. In: Proc. of CAISE'08. pp. 450–464 (2008)
9. Dumas, M., La Rosa, M., Mendling, J., Reijers, H.A., et al.: Fundamentals of business process management. Springer (2013)
10. Fontenla-Seco, Y., Lama, M., Bugarín, A.: Process-to-text: A framework for the quantitative description of processes in natural language. In: Proc. of TAILOR'20
11. Friedrich, F.: Automated Generation of Business Process Models from Natural Language Input. Master's thesis (2010)
12. Hirschberg, D.S.: A linear space algorithm for computing maximal common subsequences. Communications of the ACM (1975)
13. ISO/IEC: International Standard 19510, Information technology – Business Process Model and Notation (2013)
14. Kong, A., Zhao, S., Chen, H., Li, Q., Qin, Y., Sun, R., Zhou, X., Wang, E., Dong, X.: Better zero-shot reasoning with role-play prompting (2024)
15. Leopold, H., Mendling, J., Polyvyanyy, A.: Generating natural language texts from business process models. In: Advanced Information Systems Engineering (2012)
16. Leopold, H., Mendling, J., Polyvyanyy, A.: Supporting process model validation through natural language generation. Transactions on Software Engineering (2014)
17. Li, J., Tang, T., Zhao, W.X., Wei, Z., Yuan, N.J., Wen, J.R.: Few-shot knowledge graph-to-text generation with pretrained language models (2021)
18. de Medeiros, A.K.A., van der Aalst, W.M.P., Weijters, A.J.M.M.: Quantifying Process Equivalence Based on Observed Behavior. Data Knowl. Eng. (2008)
19. Meziane, F., Athanasakis, N., Ananiadou, S.: Generating natural language specifications from UML class diagrams. Requirements Engineering (2008)
20. Nivon, Q., Salaün, G.: Automated generation of bpmn processes from textual requirements. In: Proc. of ICSOC'24 (2024)
21. Salaün, G.: Quantifying the Similarity of BPMN Processes. In: Proc. of APSEC'22. pp. 377–386. IEEE (2022)
22. Tang, J., et al.: Graphgpt: Graph instruction tuning for large language models. In: Proc. of SIGIR'24 (2024)
23. Yuan, S., Färber, M.: Evaluating generative models for graph-to-text generation (2023)
24. Zhao, F., Zou, H., Yan, C.: Structure-aware knowledge graph-to-text generation with planning selection and similarity distinction. In: Proc. of EMNLP'23 (2023)
25. Zhu, R., Hu, Q., Li, W., Xiao, H., Wang, C., Zhou, Z.: Business process text sketch automation generation using large language model (2023)