Berti *et al. Process Science* (2025) 2:26
https://doi.org/10.1007/s44311-025-00034-4

Process Science

RESEARCH Open Access

Specializing large language models for process modeling via reinforcement learning with verifiable and universal rewards

Alessandro Berti[1*], Xiaoting Wang[1], Humam Kourani[1,2] and Wil M. P. Van der Aalst[1,2]

*Correspondence:
Alessandro Berti
a.berti@pads.rwth-aachen.de
[1]RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany
[2]Fraunhofer Institute for Applied Information Technology FIT, Schloss Birlinghoven, 53757 Sankt Augustin, Germany

Abstract

Process models are central artefacts in business process management: they drive analysis, automation, simulation, and compliance checking. Yet creating and maintaining high-quality models is labor-intensive and requires expertise in both the domain and formal notations such as Petri nets or BPMN. At the same time, many organisations already document their processes informally through textual work instructions, guidelines, and tickets. Automatically *generating* formal process models from such natural-language descriptions would therefore accelerate modeling, keep model repositories aligned with documentation, and provide stronger support for downstream uses such as conformance checking and digital twins. In this work, we study *process model generation* in a narrow, technical sense: given a textual process description, the model must produce a complete, executable process model. This generative capability can then be used both to propose models from scratch and as a building block for process modeling assistance and model completion. Large Language Models (LLMs) pretrained on generic text often struggle with this task, producing syntactically invalid or behaviorally incorrect process models. To address this limitation, we apply Reinforcement Learning (RL) to specialize a pretrained LLM specifically for process model generation. Our RL approach combines automatically verifiable rewards, based on structural checks and behavioral footprints, with universal judgments provided by an LLM-as-a-Judge. We created a dataset of 1312 textual process descriptions with corresponding reference models to support Supervised Fine-Tuning and RL. Experiments demonstrate that RL significantly reduces invalid model generations, improves behavioral correctness, and allows control over model complexity. On the ProMoAI benchmark, the resulting checkpoint approaches the performance of state-of-the-art proprietary models while producing fewer invalid generations.

Keywords Reinforcement learning, Process modeling, Process model generation, POWL, LLM-as-a-Judge, GRPO

Springer

Introduction

Process models are a cornerstone of business process management: they are used to analyse and optimize operations, enact workflows in information systems, simulate alternative scenarios, and check compliance against regulations. In practice, however, obtaining and maintaining such models is costly. Building a formal model typically requires multiple iterations between domain experts and modeling specialists, and model repositories quickly become outdated as processes evolve. At the same time, organisations already possess rich natural-language artefacts (standard operating procedures, work instructions, ticket templates, and guidelines) that describe how work is actually carried out.

This gap motivates a family of tasks that bridge informal descriptions and formal models. In this paper we focus on *process model generation*: given a textual process description, the goal is to automatically produce a complete, executable process model that captures the described behavior. Such a generator is useful on its own, for example, to bootstrap a model that a human modeler can refine, or to keep models aligned with textual documentation, and it also forms the core of more interactive use cases, such as *process modeling assistance* (suggesting fragments or design alternatives) and *process model completion* (extending a partially specified model). Throughout the paper we therefore use "process model generation" in this precise sense, while noting that the specialized model we obtain can be embedded into assistance and completion workflows.

Large Language Models (LLMs), despite their impressive general-purpose capabilities, exhibit significant shortcomings when applied directly to such process modeling tasks. Process modeling requires generating structured, semantically precise models that faithfully represent concurrency, choices, loops, and complex dependencies. Generic LLMs, trained on broad text corpora, often generate syntactically invalid or behaviorally incorrect models due to the lack of specific knowledge about formal modeling constructs (Berti et al. 2024; Rebmann et al. 2024). Supervised Fine-Tuning (SFT) on text–model pairs can partially mitigate these issues by encouraging imitation of provided examples, but it remains limited because it optimizes similarity to given references rather than directly targeting correctness or semantic fidelity.

Reinforcement Learning (RL) represents a promising paradigm for overcoming these limitations by directly optimizing a model's output based on clearly defined task-specific objectives (Bai et al. 2022). RL provides a mechanism to "specialize" pretrained LLMs by offering immediate, actionable feedback derived from verifiable constraints. In process model generation, such feedback can include automated structural checks and behavioral evaluations that measure model correctness and adherence to desired semantics.

In this paper, we explore RL-based specialization of a pretrained LLM for generating process models expressed in the Partially Ordered Workflow Language (POWL) (Kourani and van Zelst 2023; Kourani et al. 2025). We emphasize that POWL is *not* our end-user target notation; rather, we adopt it as a programmatically checkable *intermediate* representation that can be easily translated into Petri nets and BPMN models. Petri nets and BPMN are the two notations most widely used in business process management, and this choice ensures that our outputs remain readily consumable by mainstream BPM tooling while still enabling precise, automatic reward signals during training. Concretely, we build a dataset of 1312 textual process descriptions with corresponding reference models and introduce a comprehensive set of RL reward functions

combining automatically verifiable feedback (structural and behavioral footprints) with universal judgments provided by an LLM acting as a judge. Using Group Relative Policy Optimization (GRPO) (Shao et al. 2024), we systematically evaluate how RL improves generation quality and controls model complexity.

Our experiments show that RL markedly reduces invalid generations, stabilizes behavioral correctness, and enables targeted complexity adjustments. Evaluations on the ProMoAI benchmark (Kourani et al. 2024a) confirm that our RL-trained checkpoint achieves performance close to state-of-the-art models, such as GPT-4o, while producing fewer invalid generations. All experiments and training scripts are publicly available at https://github.com/fit-alessandro-berti/rl-exps.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 introduces the foundational concepts. Section 4 describes our approach. Section 5 presents the implementation and evaluation. Section 6 discusses limitations and future work. Finally, Sect. 7 concludes the paper.

Related work

This section reviews prior work relevant to our approach, organized into three key areas. Section 2.1 discusses applications of LLMs to process mining and modeling tasks. Section 2.2 examines the use of RL in process mining contexts. Finally, Sect. 2.3 explores RL techniques applied to LLMs, focusing on their alignment and specialization.

Applications of LLMs to process mining and process modeling

Fine-tuning has recently been applied to adapt general-purpose LLMs for semantics-aware process mining tasks such as anomaly detection, next activity prediction, and process discovery. These studies show that pretrained models underperform out of the box but become effective once adapted to process semantics (Rebmann et al. 2024). Other works demonstrate that LLMs can be trained for next event prediction directly from raw event logs, bypassing complex preprocessing while still producing structured outputs (Brennig et al. 2024). Semantic enrichment of event data via LLM-based object state extraction has also been shown to boost predictive monitoring accuracy (Yuan et al. 2024).

Beyond predictive settings, LLMs have been used to bridge natural language and formal models. For example, one line of research leverages textual domain knowledge to guide process discovery, producing models more consistent with expert rules (Norouzifar et al. 2024). Another line uses guideline texts to automatically extract formal rules for conformance checking in healthcare (Leonardi et al. 2025). Finally, LLM-based automatic model generation from text has been proposed, with iterative verification protocols ensuring syntactic validity (Kourani et al. 2024b). These approaches confirm that structured, verifiable outputs are essential for reliable process modeling from natural language.

Reinforcement learning in process mining contexts

Reinforcement learning (RL) has been introduced for multiple process mining and management tasks. One stream applies RL to online process discovery, enabling adaptive models under concept drift (Cai et al. 2024). Resource allocation has been optimized through RL-based agents that outperform heuristics across composite processes

(Middelhuis et al. 2025). In predictive monitoring, RL has been used to handle uncertainty and provide anticipatory interventions (Bousdekis et al. 2023). For prescriptive monitoring, RL has been combined with causal inference to decide *when* interventions are worthwhile (Bozorgi et al. 2023), and more recent work incorporates resource constraints and uncertainty-aware policies (Shoush and Dumas 2023). Offline RL frameworks such as FORLAPS further improve efficiency and effectiveness on historical logs (Abbasi et al. 2025).

Domain-specific applications extend this perspective: healthcare studies show RL policies can align with practitioners while improving outcomes (Hundogan et al. 2025), while anomaly detection approaches integrate deep RL with variational autoencoders to detect weakly supervised anomalies (El-Aziz et al. 2023). These efforts generally employ conformance signals, environment responses, or weak supervision as rewards. By contrast, the present work explores rewards grounded in structural validity, behavioral similarity, and preference judgments over LLM outputs.

Reinforcement learning on LLMs

RL has become the backbone of modern LLM alignment and capability improvement. We summarize three themes that motivate our approach: (i) SFT vs. RL, (ii) reward modeling and feedback sources, and (iii) GRPO as a policy optimizer.

SFT versus RL for language models. Supervised Fine-Tuning (SFT) teaches a model to imitate high-quality references token by token (Ouyang et al. 2022). SFT is simple and data-efficient, transferring style and safety from curated examples, but it optimizes for matching references rather than succeeding on end tasks and struggles to trade off multiple desiderata (Zhang et al. 2023). *Reinforcement learning* (RL) treats the LLM as a policy that samples responses which are then scored and reinforced according to their quality, allowing optimization directly against task-specific objectives (Bai et al. 2022).

Preference learning and reward models. Central to RL on LLMs is the *reward model*, typically trained from human or AI preferences to map a prompt–response pair to a scalar score. Reinforcement Learning from Human Feedback (RLHF) established the paradigm for aligning models to human-preferred behavior (Lee 2024). Reinforcement Learning from AI Feedback (RLAIF) scales this further by using an LLM-as-a-Judge to create preference labels at lower cost, often matching RLHF quality (Lee et al. 2024).

Policy optimization with GRPO. Among practical on-policy methods, *Group Relative Policy Optimization* (GRPO) dispenses with a learned critic by using a small *group* of samples per prompt and a group mean reward as the baseline; each sample's advantage is $A_i = r_i - \bar{r}$ (Shao et al. 2024). This yields stable updates with minimal overhead and pairs well with a KL regularizer toward a reference policy to preserve fluency.

Feedback types: verifiable vs. universal. A practical question is how to define rewards. *Verifiable* rewards are objective and programmatically checkable (Dou et al. 2024) (e.g., passing unit tests, satisfying schemas, exact numeric answers, compiling structured artifacts). *Universal* rewards capture broader qualities—helpfulness, coherence, safety—that are meaningful but not easily reduced to code; these can be implemented with *LLM-as-a-Judge*, where a strong evaluator returns scores or preferences that serve as rewards or supervision for a reward model (Yuan et al. 2024). In practice, effective systems combine both, using verifiable checks to guarantee correctness and judge signals to steer toward human-preferred behavior beyond SFT (Gu et al. 2024).

Emergent reasoning via RL. The recent "DeepSeek moment" highlights that RL can induce nontrivial reasoning behaviors. The DeepSeek-R1 pipeline first trained a model primarily with RL, yielding emergent strategies such as self-correction; a small SFT stage improved fluency, and RL then resumed with GRPO, reaching performance competitive with leading proprietary models (DeepSeek-Ai et al. 2025). This suggests a general recipe: combine verifiable task signals with scalable evaluator feedback and an optimizer to specialize models for complex domains.

Positioning of this work. We adopt these lessons in a process-modeling setting: rewards combine automatically verifiable signals (structural checks and behavioral footprints) with an LLM-as-a-Judge, optimized using GRPO. Compared to prior RL uses in process mining (resource allocation, monitoring, discovery), our focus is on *LLM policy specialization* with *verifiable program targets*, aligning with recent large-scale LLM alignment trends while grounding feedback in domain-specific checks.

Preliminaries

This section introduces the foundational concepts used throughout the paper. Section 3.1 defines the Partially Ordered Workflow Language (POWL) models, which serve as our intermediate representation for process modeling. Section 3.1 describes the footprint abstraction, capturing direct-succession and potential concurrency relations critical for behavioral evaluation.

POWL models

A *partially ordered workflow language* (POWL) model is a graph-based representation of process behavior that combines a strict partial order with control-flow operators for exclusive choice and looping (Kourani and van Zelst 2023; Kourani et al. 2025). Let Σ be a finite set of visible activity labels and let $\tau \notin \Sigma$ denote a silent activity. A POWL model M is defined inductively as follows. An activity $a \in \Sigma \cup \{\tau\}$ is a model. If $M_1, \ldots, M_k$ are models, then $\mathrm{XOR}(M_1, \ldots, M_k)$ is a model capturing exclusive choice among the submodels. If B and R are models, then $\mathrm{LOOP}(B, R)$ is a model describing a while-style loop that executes B at least once and, after each completion of B, either exits the loop or executes R followed by another iteration of B. Finally, if $N = M_1, \ldots, M_k$ is a finite set of models and $\prec \subseteq N \times N$ is a strict partial order (irreflexive and transitive), then $\mathrm{PO}(N, \prec)$ is a model where $\prec$ constrains execution order: for $(M_i, M_j) \in \prec$, all executions of M_j must be preceded by the completion of M_i, whereas pairs of nodes that are not related by $\prec$ may proceed concurrently. This strict-partial-order backbone yields compact representations of concurrency, while the XOR and LOOP operators express branching and repetition, respectively. In our implementation we use `pm4py` (Berti et al. 2023), where atomic activities are instances of `Transition` (with `SilentTransition` for τ-steps), operator nodes are represented by `OperatorPOWL` with `XOR` or `LOOP`, and the partial-order container is `StrictPartialOrder` that fixes the set of child nodes and stores ordering constraints via `add_edge`. We include the small Python snippet in Fig. 1 not to teach an API, but to make explicit that POWL models are *programmatically constructible* from text: a short, executable script deterministically builds a model that we can verify and translate. This foreshadows our later methodology, where an LLM will *generate such code* from natural-language descriptions; the snippet therefore serves as the concrete target artifact whose syntax can be checked and whose

```python
import pm4py
from pm4py.objects.powl.obj import StrictPartialOrder, OperatorPOWL, Transition,
    SilentTransition
from pm4py.objects.process_tree.obj import Operator
# Activities
register = Transition(label="register request")
reinitiate = Transition(label="reinitiate request")
examine_casually = Transition(label="examine casually")
examine_thoroughly = Transition(label="examine thoroughly")
check_ticket = Transition(label="check ticket")
decide = Transition(label="decide")
reject = Transition(label="reject request")
pay = Transition(label="pay compensation")

# Examination XOR choice
examine_choice = OperatorPOWL(operator=Operator.XOR, children=[examine_casually,
    examine_thoroughly])

# Examination subprocess
examine_subproc = StrictPartialOrder(nodes=[examine_choice, check_ticket, decide])
examine_subproc.add_edge(examine_choice, decide)
examine_subproc.add_edge(check_ticket, decide)

# Loop: body = examination subprocess, redo = reinitiate
loop = OperatorPOWL(operator=Operator.LOOP, children=[examine_subproc, reinitiate])

# After loop: XOR between reject or pay
decision_choice = OperatorPOWL(operator=Operator.XOR, children=[reject, pay])

# Root partial order
root = StrictPartialOrder(nodes=[register, loop, decision_choice])
root.add_edge(register, loop)
root.add_edge(loop, decision_choice)

# Saves a visualization of the POWL model
pm4py.save_vis_powl(root, "powl_ru.pdf")

# Converts the POWL model to a Petri net
net, im, fm = pm4py.convert_to_petri_net(root)
pm4py.save_vis_petri_net(net, im, fm, "petri_ru.pdf")
```

Fig. 1 Python code in pm4py (Berti et al. 2023) to generate a POWL model and convert it to a Petri net

behavior can be evaluated automatically (and converted to Petri nets/BPMN). The running example in Fig. 1 constructs a model in which the registration must precede an examination loop and the loop must precede a final XOR decision; the corresponding visualization and a behaviorally equivalent Petri net are shown in Fig. 2.

Worked example (cf. Figs. 1 and 2). The example instantiates a POWL model with eight visible activities: *register request, examine casually, examine thoroughly, check ticket, decide, reinitiate request, reject request,* and *pay compensation*. Formally, the model is built bottom-up as

$$
\begin{aligned}
E &= \mathrm{XOR}(\text{examine casually}, \text{examine thoroughly}),\\
B &= \mathrm{PO}\big(\{E, \text{check ticket}, \text{decide}\}, \{(E, \text{decide}), (\text{check ticket}, \text{decide})\}\big),\\
L &= \mathrm{LOOP}\big(B, \text{reinitiate request}\big),\\
D &= \mathrm{XOR}(\text{reject request}, \text{pay compensation}),\\
M &= \mathrm{PO}\big(\{\text{register request}, L, D\}, \{(\text{register request}, L), (L, D)\}\big).
\end{aligned}
$$

Intuitively: after *register request,* the process enters a looped examination subprocess. In each iteration, the examination (either *examine casually* or *examine thoroughly*) and *check ticket* are *unordered*—they may proceed in either order or concurrently—but both must complete before *decide*. After each pass, the loop either terminates or performs *reinitiate request* before repeating, thus ensuring the body executes at least once and that repetition is governed by the explicit `LOOP` operator rather than by cycles in the partial order. When the loop finishes, a final XOR chooses between *reject request* and *pay compensation*. This structure is constructed in Fig. 1 using `OperatorPOWL` for the XOR/LOOP nodes and `StrictPartialOrder` for the precedence constraints (here, both E and check ticket precede decide while remaining mutually unordered); its visualization and the behaviorally equivalent Petri net produced by `pm4py.convert_to_petri_net` appear in the top and bottom panels of Fig. 2, respectively.

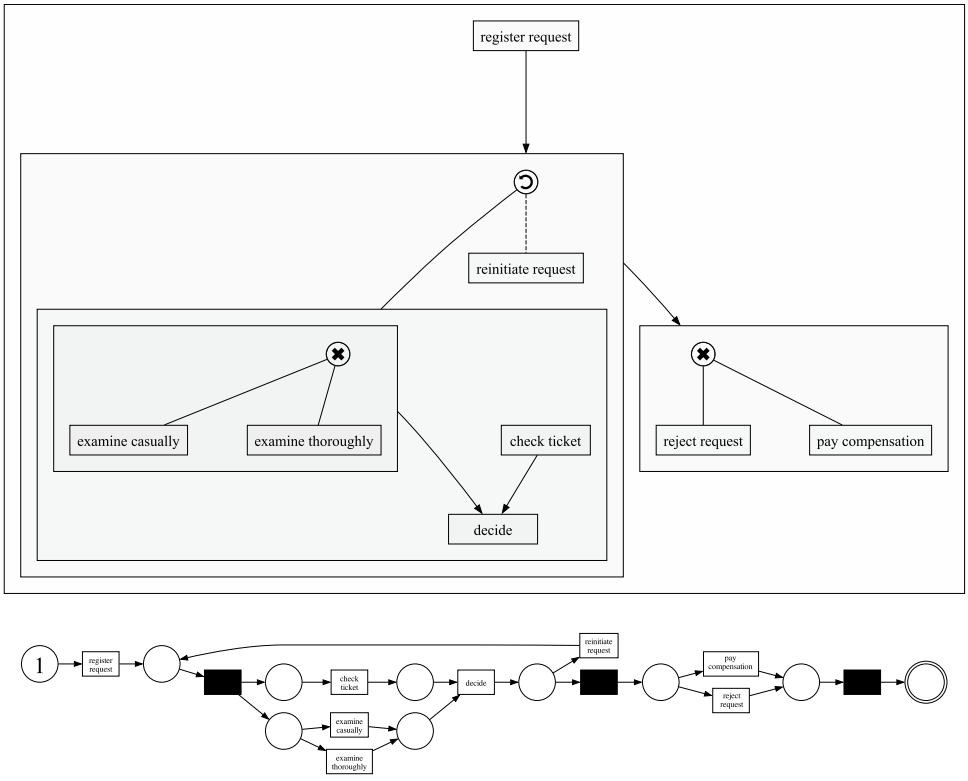


Fig. 2 Visualization of the POWL model and the corresponding Petri net

Footprints of POWL models

Footprints are a compact, relation-based summary of a process model's *observable behavior*. Instead of enumerating full traces, a footprint records—for each pair of activities—whether one can directly follow the other in some execution (direct succession) and whether the two can appear in either order across executions (potential concurrency). This abstraction is model-agnostic (usable for POWL, Petri nets, or BPMN), inexpensive to compute, and robust to behavior-preserving rewritings. In our setting, footprints serve two key purposes: (i) a verifiable target for behavioral evaluation and RL rewards, and (ii) a basis for selection and diagnostics at inference time.

For a POWL model M, let

$$\mathsf{footprints}(M) = \big(\mathsf{Seq}(M),\ \mathsf{Par}(M)\big).$$

Here, $\mathsf{Seq} \subseteq \Sigma \times \Sigma$ collects all activity pairs that can occur in direct succession in at least one execution of M, while $\mathsf{Par} \subseteq \Sigma \times \Sigma$ records potential concurrency: $(a, b) \in \mathsf{Par}$ iff both orders $a \to b$ and $b \to a$ are possible across different executions.

For the running example built in Fig. 1, the complete relations are:

$$\begin{aligned}\mathsf{Seq}(M_{\text{run}}) = \{&(\text{reinitiate request}, \text{examine thoroughly}), (\text{reinitiate request}, \text{examine casually}),\\ &(\text{decide}, \text{reject request}), (\text{check ticket}, \text{decide}), (\text{decide}, \text{pay compensation}),\\ &(\text{examine casually}, \text{decide}), (\text{decide}, \text{reinitiate request}), (\text{register request}, \text{check ticket}),\\ &(\text{register request}, \text{examine thoroughly}), (\text{register request}, \text{examine casually}),\\ &(\text{reinitiate request}, \text{check ticket}), (\text{examine thoroughly}, \text{decide})\},\end{aligned}$$

$$\begin{aligned}\mathsf{Par}(M_{\text{run}}) = \{&(\text{check ticket}, \text{examine casually}), (\text{examine casually}, \text{check ticket}),\\ &(\text{examine thoroughly}, \text{check ticket}), (\text{check ticket}, \text{examine thoroughly})\}.\end{aligned}$$

Approach

As motivated in Sect. 1, we must translate a textual process description into an *executable* and *behaviorally faithful* process model. We specialize a pretrained LLM with *reinforcement learning* (RL), using *POWL* as a verifiable, intermediate program target, and *Group Relative Policy Optimization* (GRPO) to optimize rewards that combine *verifiable* checks (syntax, operator typing, footprints agreement) and *universal* preferences (LLM-as-a-Judge).

We proceed in six steps. We (i) formalize the problem and objectives (Sect. 4.1); (ii) justify the output space and prompting (Sect. 4.2); (iii) describe the optimization rule (GRPO; Sect. 4.3); (iv) define the reward signals (Sect. 4.4); (v) assemble these signals into a concise training curriculum (Sect. 4.5); and (vi) close with the test-time procedure (Sect. 4.6).

The overall workflow of our reinforcement-learning framework is summarized in Fig. 3, which illustrates how textual process descriptions are translated into executable POWL models and optimized through GRPO using both verifiable and universal rewards.

Problem and Objectives

Building on the preliminaries (Sect. 3), we now pin down the task and success criteria.

Task. Given a natural-language specification x, the policy must produce a short Python program y that executes to a POWL model $M = \mathrm{eval}(y)$.

Objectives. We score y (hence M) along three axes:

(D1) Structural validity. A completion is structurally valid iff, when executed in a sandbox:

S1. the code parses (valid AST) and uses only allowed POWL/utility imports;
S2. it constructs a single, top-level POWL object `root` via official constructors (`Transition`, `SilentTransition`, `OperatorPOWL`, `StrictPartialOrder`);
S3. operator arities/types are correct (e.g., `LOOP` has $(\mathrm{body}, \mathrm{redo})$; `XOR` has ≥ 2 children);
S4. the strict partial order is irreflexive, asymmetric, and *acyclic* (repetition only via `LOOP`);
S5. activity labels are strings; silent steps use the designated symbol; no inconsistent object reuse;
S6. no side effects beyond model construction.

We use a predicate $\mathrm{valid}_{\mathrm{struct}}(y) \in \{0, 1\}$ and a soft *code-quality* score $s_{\mathrm{code}} \in [0, 1]$ summarizing S1–S6.

(D2) Behavioral fidelity. With footprints $\mathsf{footprints}(M) = (\mathsf{Seq}(M), \mathsf{Par}(M))$ (Sect. 3), and a reference model M_{ref} when available, we measure similarity via the Jaccard index:

$$\mathsf{sim}(M, M_{\mathrm{ref}}) = \frac{|\mathsf{Seq}(M) \cap \mathsf{Seq}(M_{\mathrm{ref}})| + |\mathsf{Par}(M) \cap \mathsf{Par}(M_{\mathrm{ref}})|}{|\mathsf{Seq}(M) \cup \mathsf{Seq}(M_{\mathrm{ref}})| + |\mathsf{Par}(M) \cup \mathsf{Par}(M_{\mathrm{ref}})|}.$$

(D3) Model complexity. We use $\mathrm{FP}(M) = |\mathsf{Seq}(M)| + |\mathsf{Par}(M)|$ as a proxy for structural richness to avoid under-/over-structuring.

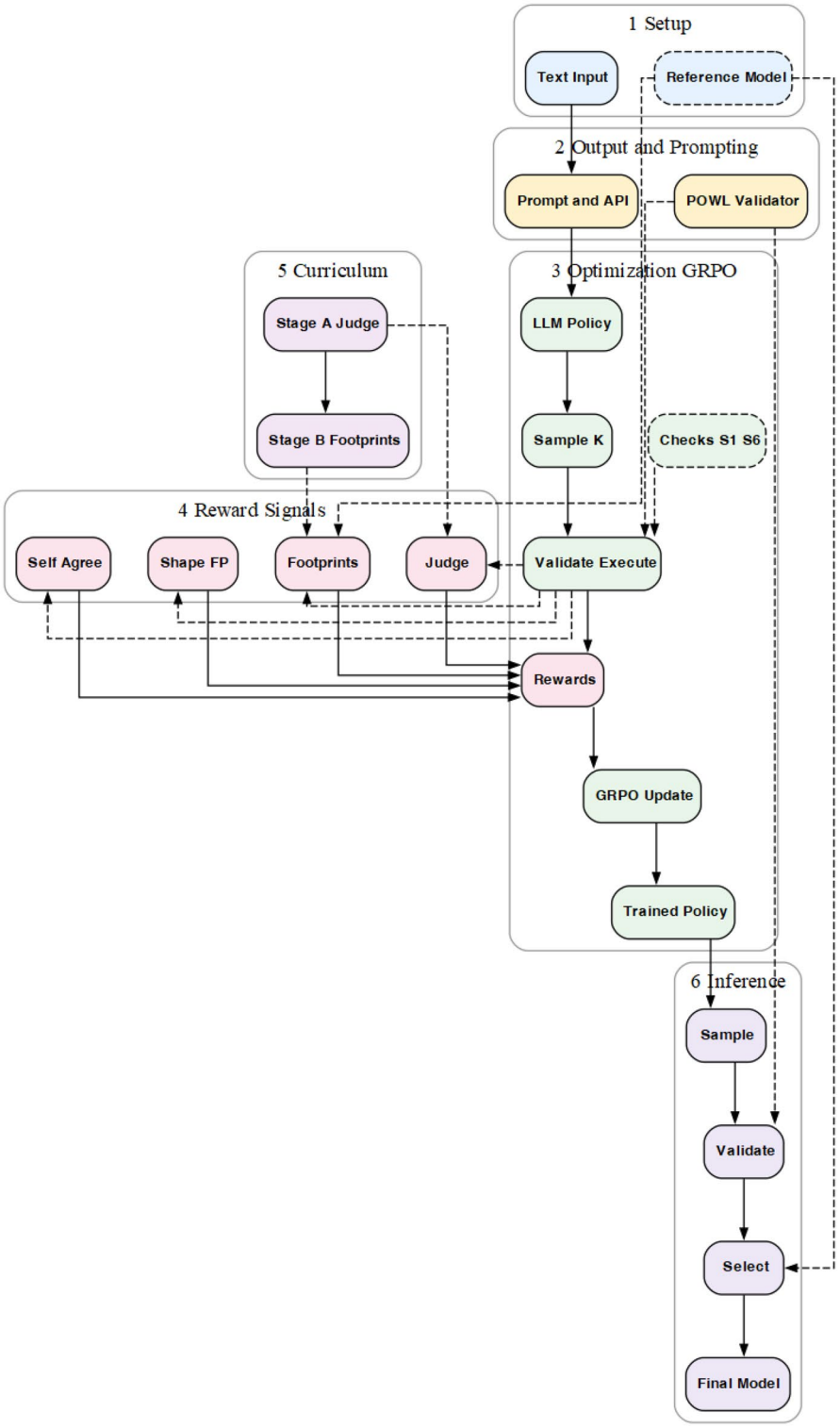


Fig. 3 Overview of the proposed RL approach for process model generation. The pipeline starts from textual process descriptions and optional reference models (1). Prompts are constructed and mapped to executable POWL programs (2). During training (3), the LLM policy generates multiple candidates, which are validated, scored by universal (judge-based) and verifiable (footprints, self-agreement, shaping) rewards (4), and updated via Group Relative Policy Optimization (GRPO). The curriculum (5) alternates between judge-based and footprints-based stages to improve validity and behavioral fidelity. At inference time (6), the trained policy samples and validates multiple candidates, selecting the best process model according to structural and behavioral criteria

Output representation and prompting

With the objectives set, we choose an output space that makes them *verifiable*.

Choice of output space. The policy emits concise Python that constructs POWL models using `pm4py`. This enables: (i) strong structural checks (S1–S6), (ii) execution to recover M for behavioral analysis, and (iii) reuse across notations (POWL → Petri nets/BPMN) without changing training.

Prompting. Prompts include the textual specification x and a minimal reminder of the POWL API and naming discipline for activities. Figure 4 shows the provided textual prompt.

Optimization with GRPO

GRPO is the approach we use to improve the model's output quality. For each textual specification, we ask the model to produce a small *group* of alternative programs. We then score each program using the rewards described in Sect. 4.4. Programs that score *better than the group's average* are encouraged, while those that score *worse than the average* are discouraged. Repeating this process across many prompts gradually shifts the model toward outputs that are consistently preferred according to our rewards. Figure 5 illustrates one GRPO step.

How one learning step works.

- *Sample a group.* For a given prompt, generate a handful of candidate programs (we use a small number to keep training fast).
- *Validate first.* Run the structural validators introduced earlier; invalid programs are marked as such and do not proceed to expensive checks.

```
Generate a POWL model for the following process, saving the final result in the variable '
    root'.
A partially ordered workflow language (POWL) is a partially ordered graph representation
    of a process, extended with control-flow operators for modeling choice and loop
    structures. There are four types of POWL models:
- an activity (identified by its label, e.g., 'M' identifies the activity M). Silent
    activities with empty labels (tau labels) are also supported.
- a choice of other POWL models (exclusive choice: X(A, B)).
- a loop node (* (A, B)): execute A, then choose to exit or execute B then A again,
    repeated until exit.
- a partial order: PO=(nodes={...}, order={...}), where order is a set of source-->target
    dependencies; unconnected nodes are concurrent.

Example 1: PO=(nodes={NODE1, NODE2}, order={})
Example 2: PO=(nodes={NODE1, NODE2}, order={NODE1-->NODE2})
Example 3: PO=(nodes={NODE1, NODE2, NODE3, X(NODE4, NODE5)}, order={NODE1-->NODE2, NODE1
    -->X(NODE4, NODE5), NODE2-->X(NODE4, NODE5)})

POWL classes in pm4py.objects.powl.obj:
- SilentTransition(): silent transition
- Transition(label): labeled transition
- StrictPartialOrder(nodes=[...]) with .add_edge(src, tgt)
- OperatorPOWL(operator=Operator.XOR or Operator.LOOP, children=[...])

Example code:
import pm4py
from pm4py.objects.powl.obj import StrictPartialOrder, OperatorPOWL, Transition,
    SilentTransition
from pm4py.objects.process_tree.obj import Operator
A = Transition(label='A')
B = Transition(label='B')
C = Transition(label='C')
skip = SilentTransition()
loop = OperatorPOWL(operator=Operator.LOOP, children=[A, B])
xor = OperatorPOWL(operator=Operator.XOR, children=[C, skip])
root = StrictPartialOrder(nodes=[loop, xor])
root.add_edge(loop, xor)

Now, generate the POWL model for the process below.
DESCRIPTION: {description}
ACTIVITIES (use these exactly, same names): {activities}

Respond with valid Python code only, defining 'root'.
```

Fig. 4 Prompt to generate a POWL model from a process description

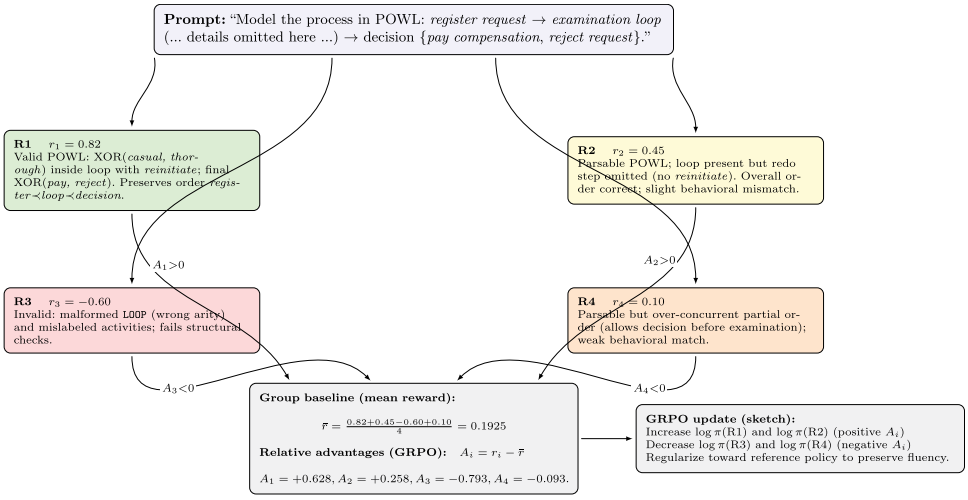


Fig. 5 Illustrative GRPO step: a prompt yields four sampled responses with scalar rewards in $[-1, 1]$. The group mean $\bar{r}$ defines a relative baseline; advantages $A_i = r_i - \bar{r}$ drive per-sample up/down updates

- *Score each candidate.* Apply the active reward (e.g., judge scores or footprints agreement). All scores are mapped onto the same, bounded scale so they are comparable.
- *Compare to the group average.* Compute the group's average score. Any candidate above this average is treated as a positive example for the current prompt; any candidate below it is treated as a negative example.
- *Nudge the policy.* Increase the model's tendency to produce above-average candidates and decrease it for below-average ones.

Why "group-relative" helps. Absolute reward scales can vary across prompts (some descriptions are inherently harder than others). By comparing candidates only *within* the same prompt, GRPO avoids chasing raw numbers that are not comparable. The model simply learns: "for this kind of prompt, prefer the variants that tended to be better".

Keeping learning stable. We adopt a few guardrails so the model improves without drifting or overfitting:

- *Stay close to a fluent reference.* We gently regularize the policy toward a fixed reference model. This keeps formatting, syntax, and general writing style stable while still allowing targeted improvements.
- *Normalize and clip scores.* Before using scores for learning, we normalize them within the group and clip extremes. This prevents a single outlier from dominating an update and makes training less sensitive to noisy judgments.
- *Be fair to short and long programs.* We make sure longer code does not automatically get more credit just because it has more tokens. Each candidate influences learning to roughly the same degree.

Handling invalid or degenerate outputs.

- *Strong penalty, useful signal.* If a program fails validation, it receives the lowest score and remains in the group. This sharpens the contrast with valid peers and teaches the model which patterns to avoid.

– *Skip hopeless groups.* If *all* candidates in a group are invalid, we skip the update for that prompt. This prevents the model from learning from pure noise and becomes rare after the warm start.

Encouraging diversity without chaos. Training relies on seeing both good and bad alternatives for the same prompt. We therefore sample with moderate randomness so groups contain diverse candidates. At the same time, we use strict stop conditions and length limits: truncated or trailing commentary is treated as invalid, keeping the signal focused on proper program generation.

Interaction with our rewards. GRPO is agnostic to *which* reward is active. When the judge reward is active, the model learns to satisfy the prompt more completely and coherently. When the footprints reward is active, the model learns to match the required behavior. Optional stages (self-agreement or complexity shaping) plug in seamlessly: the procedure stays the same; only the scoring function changes.

Why this optimizer fits our setting. GRPO is robust to shifting reward scales and naturally aligned with program generation where validity matters. It turns every prompt into a self-contained lesson: "among these alternatives, prefer the better ones". Combined with our validators and rewards, this produces steady gains in structural reliability and behavioral fidelity without introducing additional modeling complexity.

Reward signals (verifiable and universal)

With the optimizer fixed in Sect. 4.3, we define the signals it uses.

Universal reward: LLM-as-a-Judge. An evaluator LLM inspects the prompt and K candidates and returns per-candidate grades in $[-1, 1]$ capturing correctness, completeness, and adherence to x. The detailed prompt used for the LLM-as-a-Judge is shown in Fig. 6.

```
You are an expert in process modeling. Your task is to evaluate Python code snippets that
     generate POWL models based on a given prompt.

The original prompt was:
---
{{ORIGINAL_PROMPT}}
---

I have received 4 responses. Please evaluate each response based on its correctness,
     completeness, and adherence to the prompt requirements.
Provide a grade for each response on a scale from -1.0 to 1.0, where:
- 1.0: The generated code is perfect and accurately models the process description.
- 0.1 to 0.9: The code is acceptable but may have minor errors or inaccuracies.
- 0.0: The code is syntactically correct but functionally wrong or completely misses the
     point.
- -1.0: The code is terrible, syntactically incorrect, or completely irrelevant.

Here are the responses to grade:

--- RESPONSE 1 ---
{{COMPLETION_1}}

--- RESPONSE 2 ---
{{COMPLETION_2}}

--- RESPONSE 3 ---
{{COMPLETION_3}}

--- RESPONSE 4 ---
{{COMPLETION_4}}

Please provide your evaluation in a single JSON object. The JSON should have a single key
     "grades", which is a list of four floating-point numbers corresponding to each
     response (in order). The number of grades must be exactly 4.

Example format:
{
"grades": [0.8, -0.5, 1.0, 0.4]
}

Now, provide the JSON output for the responses above.
```

Fig. 6 Judge prompt used to score the $K=4$ sampled POWL programs for a given textual specification. The evaluator returns a JSON object with a single key `"grades"` whose four entries lie in $[-1, 1]$ (one per response); these grades serve as the universal reward signal in GRPO

Verifiable reward: footprints agreement (simple behavioral). Given M_{ref}, we reward footprints similarity via a monotone map $g : [0, 1] \to [-1, 1]$:

$$r_{\text{simple}}(M) = g(\mathsf{sim}(M, M_{\text{ref}})) \,, \tag{1}$$

optionally adding small bonuses for $\mathsf{valid}_{\text{struct}}$ and activity coverage.

Verifiable reward: self-agreement (reference-free). Without references, encourage consensus among the K candidates:

$$r_{\text{self}}(M_i) = \frac{1}{K-1} \sum_{j \neq i} \mathsf{sim}(M_i, M_j). \tag{2}$$

Verifiable reward: footprint shaping (complexity control). Let $h : \mathbb{N} \to (0, 1)$ be increasing and saturating. Then

$$r_{\pm\text{foot}}(M) = \begin{cases} h(\text{FP}(M)) & \text{encourage richer behavior,} \\ 1 - h(\text{FP}(M)) & \text{encourage simpler behavior.} \end{cases} \tag{3}$$

Verifiable reward: composite partial credit. Blend coverage, code quality, and behavior:

$$r_{\text{complex}} = \lambda_a \frac{|A_{\text{exp}} \cap A_{\text{gen}}|}{|A_{\text{exp}} \cup A_{\text{gen}}|} + \lambda_c \, s_{\text{code}} + \lambda_b \, \mathsf{sim}(M, M_{\text{ref}}), \tag{4}$$

with nonnegative weights $\lambda_a + \lambda_b + \lambda_c = 1$. In the composite reward function, A_{exp} represents the expected set of activity labels from the reference process description, while A_{gen} denotes the set of activity labels generated in the POWL model, with their Jaccard similarity measuring activity coverage to ensure the model includes all required activities without extras.

Training Curriculum

Program synthesis from text exhibits a pronounced *reward sparsity* at the beginning of training: many samples are syntactically invalid or semantically unusable, so verifiable rewards cannot even be computed. Our curriculum schedules *one* reward at a time to progressively increase reward density and tighten the optimization target. The stages move from (i) broad, dense guidance (judge) that quickly suppresses invalid code and aligns coarse semantics, to (ii) precise, verifiable signals (footprints) that consolidate the intended behavior, with short *optional* branches to improve robustness or control complexity. The optimizer (GRPO) and validators (S1–S6; see Sect. 4.1) remain unchanged across stages.

Optional warm start (SFT). Although *not required* by our approach, we can precede RL with a brief SFT pass on (x, y_{ref}) pairs. The goal is *not* to maximize imitation but to:

(a) raise the on–policy *parse rate* so that a meaningful fraction of sampled programs pass S1–S6;
(b) calibrate the model to the POWL API and naming discipline (shorter exploration horizon, lower variance);

Empirically, this increases reward density for the subsequent RL stages and improves sample efficiency. SFT can be shortened or skipped; the rest of the curriculum remains valid.

Stage A — Closing the validity gap (Judge → GRPO). *Active reward:* universal, LLM-as-a-Judge (Sect. 4.4).

Objective: sharply reduce structural invalidity and align coarse process semantics with the prompt.

Why this works. Judge scores provide *dense*, prompt-relative feedback even when exact behavioral signals are unavailable. In GRPO, invalid or off-format completions receive consistently lower grades than valid ones within the same prompt group, yielding strong negative advantages and thus rapid elimination of systemic errors (e.g., malformed `LOOP`, wrong `XOR` arities, cycles in the partial order). At the same time, the rubric rewards basic control–flow structure and activity coverage, nudging the policy toward the right *shape* of solutions.

Why we do this first. Footprints-based rewards require executing a valid POWL model to obtain Seq/Par; when the valid-sample rate is low, footprints signals are sparse and noisy. Stage A raises the valid-sample fraction and stabilizes code formatting so Stage B can operate on a dense set of verifiable candidates.

Stage B — Consolidating behavior (Footprints → GRPO). *Active reward:* simple behavioral reward r_{simple} in (1).

Objective: maximize agreement of generated footprints with the target behavior; stabilize control-flow semantics.

Why this works. Once the policy reliably emits valid programs, footprints provide *objective, programmatically checkable* supervision on the two relations we care about most: direct succession and potential concurrency (Sect. 3.1). Optimizing r_{simple} aligns the partial-order backbone and operator placement with the reference semantics. Because the reward is verifiable and bounded, GRPO updates become low-variance; the KL term prevents drift in surface form while allowing structural corrections.

Effect in practice. Stage B preserves the low invalid rate achieved in Stage A while (i) increasing footprints similarity, (ii) narrowing the spread of behaviors across samples (fewer near-miss variants), and (iii) reducing residual structural pathologies that escape judge heuristics.

Optional short branches (plug-ins). These can be inserted after Stage B (or briefly interleaved) without altering the core pipeline.

(i) *Reference-free robustness via self-agreement.*

Active reward: r_{self} in (2).

When/why: Use when references are unavailable or noisy. Maximizing average pairwise similarity among the K candidates acts like a consensus regularizer: it suppresses heavy-tail, idiosyncratic structures and improves stability under sampling. Expect narrower variance and milder extremes in footprints counts, with minimal change in medians.

(ii) *Complexity shaping.*

Active reward: $r_{\pm\text{foot}}$ in (3).

When/why: Use to *increase* structural richness (encourage parallelism/branching) when the model under-structures, or to *decrease* it when the model over-structures. This exposes a controllable "complexity knob"; expect predictable shifts in $\mathrm{FP}(M)$ (median and max) with a possible accuracy trade-off if pushed aggressively.

(iii) *Composite partial credit.*

Active reward: r_{complex} in (4).

When/why: Use when references are incomplete or activity labels are inconsistent. Blending activity coverage, code-quality, and footprints yields graded feedback that remains informative even for partially correct candidates. This is useful as a brief "bridge" stage before or after Stage B in low-quality data regimes.

(iv) *Judge polish (optional).*

A short return to the judge reward after Stage B can improve prompt adherence phrasing and minor stylistic conventions without altering behavior, thanks to the KL regularizer and the validators.

The curriculum first *densifies* feedback (judge) to close the validity gap, then *anchors* learning to verifiable semantics (footprints) to consolidate behavior, and finally uses short, optional branches to trade off robustness and complexity as needed. All stages use the same GRPO approach and validators; only the active reward changes.

Inference and selection

At test time we use the same routine as in Sect. 5. Given a new textual description, we produce one POWL model by sampling a small set of candidates and selecting the best according to the same checks and summaries used in evaluation.

Procedure (also used in evaluation).

1. **Generate K candidates.** Sample K short Python programs from the trained policy (we use $K=4$ in all experiments).
2. **Validate structure.** Apply the validators S1–S6 from Sect. 4.1. Candidates failing any check are discarded and counted as invalid in our reported metrics.
3. **Summarize behavior.** Execute each valid program to obtain a POWL model M_i and extract its footprints $\big(\mathsf{Seq}(M_i), \mathsf{Par}(M_i)\big)$ as in Sect. 3.1.
4. **Score and select.**
   - *With a reference model* M_{ref}: select $\arg\max_i \mathsf{sim}(M_i, M_{\text{ref}})$, i.e., the candidate with the highest footprints similarity (Jaccard) defined in Sect. 4.1.
   - *Without a reference*: prefer (i) higher structural code-quality, (ii) full activity coverage, and (iii) stronger consensus with peers (average footprints similarity to the other candidates).
   - *Ties*: break in favor of the simpler model (fewest footprints $|\mathsf{Seq}|+|\mathsf{Par}|$).

The selected result is the POWL model (and its tiny construction program), which can be visualized or converted to Petri nets/BPMN via `pm4py`. In the evaluation (Sect. 5) we always sample $K=4$ candidates per prompt: we report counts of invalids (S1–S6), summarize rewards and footprint sizes over the valid ones (Table 1), and, for ProMoAI, show min/mean/max of the replay F-score across the same four candidates (Table 2). When a single model is required (deployment use case), we return the candidate selected by the rule above.

Table 1 Training variants and aggregate statistics per run on the 312-description test split. *Neg. (−1)* counts completions assigned the lowest reward; *Non-Neg.* counts completions with reward > -1. Rewards and footprints are summarized over valid completions

Model	Parent	Neg. (−1)	Non-Neg.	Min R	Med. R	Max R	Min FP	Med. FP	Max FP
Base (Step 0)	—	889	359	−0.50	0.56	1.00	3	24	281
SFT	Base	328	920	−0.50	0.62	1.00	0	18	289
Step 1 (GRPO, LLM-as-a-Judge)	Base	152	1096	0.43	0.65	1.00	12	14	226
Step 2 (GRPO, simple reward)	Step 1	4	1244	0.51	0.65	1.00	14	15	199
Step 3 (GRPO, self-reward)	Step 2	2	1246	0.51	0.65	1.00	14	15	49
Step 4 (GRPO cont. on Step 2, simple)	Step 2	0	1248	0.51	0.65	1.00	14	15	76
Step 5 (GRPO cont. on Step 2, Judge)	Step 2	2	1246	0.51	0.65	1.00	14	15	113
Step 6 (GRPO, footprints↑)	Step 2	3	1245	0.20	0.52	0.91	14	240	380
Step 7 (GRPO, footprints↓)	Step 2	1	1247	0.51	0.65	1.00	14	15	20
Step 8 (GRPO cont. on Step 1, Judge)	Step 1	130	1118	−0.50	0.64	1.00	0	15	289
Step 9 (GRPO, complex reward)	Base	831	417	0.40	0.51	0.91	2	30	292

Implementation and evaluation

After formalizing the task, outputs, optimizer, rewards, curriculum, and inference in Sect. 4, we now *operationalize* that plan: we describe the corpus and splits, show how the reward signals from Sect. 4.4 are *instantiated*, realize the curriculum from Sect. 4.5 as concrete checkpoints, and evaluate the resulting models with the selection procedure aligned to Sect. 4.6. We report aggregate results, training dynamics, and an external benchmark.

Corpus and Splits

To apply the rewards in Sect. 4.4 and train per Sect. 4.5, we need paired text–model examples and held-out descriptions. We first summarize the dataset; in Sect. 5.2 we explain how the rewards are *instantiated* on this data.

Corpus. We use a collection of **1312** process descriptions paired with POWL reference models (publicly available at https://github.com/fit-alessandro-berti/pmodel-collection). Descriptions explicitly list activities and routing constraints (choices, loops, partial orders). References are executable POWL programs that adhere to the constructors referenced in Sect. 4.2.

Quality and validity checks. Each reference model passes structural checks that mirror the validators summarized in Sect. 4.1 (S1–S6), with a specific emphasis on the strict-partial-order backbone being a *DAG* (no self loops, no two-cycles, and no longer cycles; loops only via `LOOP`). This ensures rewards based on execution and footprints are well-defined.

Splits. We partition the set into **1,000** training descriptions (for SFT/RL) and **312** held-out descriptions (for evaluation only). All evaluation results in this section are on the 312-description test split.

Reward instantiation

The *kinds* of rewards are defined in Sect. 4.4. Here we specify how we *instantiate* them during training and evaluation, without repeating definitions or equations; Sect. 5.3 then places these signals into a concrete training schedule.

Table 2 ProMoAI results across four models (labels 01–20). Each row aggregates four samples per description (minimum/mean/maximum token-based replay F-score)

Label	Qwen2.5–Coder–3B			Qwen2.5–Coder–14B			Qwen2.5–Coder–32B			RL Step 2 (ours)		
	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max	Min	Mean	Max
01	0.000	0.000	0.000	0.000	0.596	0.846	0.000	0.449	0.899	0.000	0.421	0.899
02	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.224	0.894	0.000	0.355	0.674
03	0.000	0.000	0.000	0.000	0.184	0.736	0.000	0.608	0.884	0.351	0.454	0.489
04	0.000	0.000	0.000	0.000	0.619	0.871	0.000	0.469	0.940	0.848	0.865	0.871
05	0.000	0.000	0.000	0.000	0.225	0.899	0.000	0.570	0.827	0.624	0.779	0.862
06	0.000	0.000	0.000	0.000	0.428	0.876	0.000	0.413	0.927	0.483	0.619	0.687
07	0.000	0.000	0.000	0.000	0.651	0.899	0.000	0.602	0.856	0.721	0.763	0.783
08	0.000	0.000	0.000	0.000	0.221	0.886	0.000	0.209	0.836	0.000	0.226	0.585
09	0.000	0.000	0.000	0.000	0.195	0.781	0.000	0.000	0.000	0.000	0.632	0.880
10	0.000	0.000	0.000	0.000	0.522	0.765	0.000	0.456	0.918	0.918	0.918	0.918
11	0.000	0.000	0.000	0.000	0.348	0.768	0.000	0.000	0.000	0.349	0.505	0.617
12	0.000	0.000	0.000	0.000	0.425	0.868	0.000	0.671	0.894	0.000	0.563	0.820
13	0.000	0.000	0.000	0.000	0.219	0.877	0.000	0.448	0.913	0.606	0.769	0.828
14	0.000	0.000	0.000	0.000	0.209	0.837	0.000	0.413	0.837	0.738	0.763	0.793
15	0.000	0.000	0.000	0.000	0.624	0.919	0.919	0.919	0.919	0.732	0.772	0.833
16	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.306	0.821	0.261	0.455	0.570
17	0.000	0.000	0.000	0.000	0.222	0.629	0.000	0.000	0.000	0.000	0.058	0.232
18	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.420	0.839	0.189	0.425	0.743
19	0.000	0.000	0.000	0.000	0.441	0.936	0.000	0.443	0.894	0.611	0.704	0.804
20	0.000	0.000	0.000	0.000	0.206	0.824	0.000	0.245	0.978	0.530	0.635	0.787
Avg.	**0.000**	**0.000**	**0.000**	**0.000**	**0.317**	**0.711**	**0.046**	**0.393**	**0.754**	**0.398**	**0.584**	**0.734**

Common setup. For each prompt we sample a **group of** $K=4$ candidate programs (as in Sect. 4.3). All candidates are filtered by the same validators (S1–S6 from Sect. 4.1); only valid ones are executed to obtain M for footprints-based scoring. All rewards are normalized to a common $[-1, 1]$ range and clipped if needed before GRPO.

Universal (judge) reward. An evaluator LLM grades each of the K candidates against the prompt under a concise rubric (correctness, completeness, adherence). The grader returns one scalar per candidate in $[-1, 1]$. We do not rely on the grader for syntax safety; validators already gate execution.

Simple behavioral reward. Given a reference M_{ref} for the prompt, we compute footprints similarity $\mathsf{sim}(M, M_{\text{ref}})$ and map it to $[-1, 1]$ via a monotone normalization. Small, bounded bonuses may be added for structural validity and activity coverage. (See Sect. 4.4 for the definition; we avoid duplicating equations here.)

Self-agreement reward. When reference-free, we reward consensus across the K valid candidates via average pairwise footprints similarity. This furnishes a usable training signal even without M_{ref}.

Footprint shaping. We shape complexity using the number of footprints $\text{FP}(M)$, with a smooth increasing squashing function to keep scores bounded, encouraging either richer or simpler models depending on the stage goal.

Composite partial credit. To grant graded feedback when candidates are imperfect, we blend activity coverage, a code-quality summary of S1–S6, and footprints agreement, using nonnegative weights summing to one. No new symbols are introduced beyond those already defined in Sect. 4.4.

Curriculum Realization and Checkpoints

With rewards instantiated, we now *realize* the training plan from Sect. 4.5. We specify the order of stages and the checkpoints used in evaluation; Sect. 5.4 will analyze the outcomes.

Base and SFT. We start from a base code-oriented LLM and apply Supervised Fine-Tuning on the 1,000 training pairs to stabilize syntax and teach the POWL "shape" (Sect. 4.2).

GRPO stages. We then run short GRPO phases, *activating one reward at a time*, as outlined in Sect. 4.5:

1. **Stage A (Judge → GRPO)**: close the validity gap using the universal judge signal.
2. **Stage B (Footprints → GRPO)**: consolidate behavioral fidelity using the simple footprints reward.
3. **Optional branches:** apply self-agreement (robustness), footprint shaping (complexity control), or composite partial credit (noisy references).

Checkpoints and DAG. The concrete realization (node names match the "Model" column in Table 1) is shown in Fig. 7. The main path is *Base (Step 0)* → *Step 1 (Judge)* → *Step 2 (Simple)*, with branches for Steps 3–7 and an alternative continuation (Step 8). Step 9 starts directly from Base with the composite reward. Each GRPO stage uses the same $K=4$, validators S1–S6, and a light KL regularizer to the reference policy.

Evaluation protocol. For each checkpoint we evaluate on the 312-description test split by sampling 4 candidates per prompt (1,248 completions per checkpoint). We report: (i) *Invalid count* (violating S1–S6), (ii) *Non-negative* completions under the active reward,

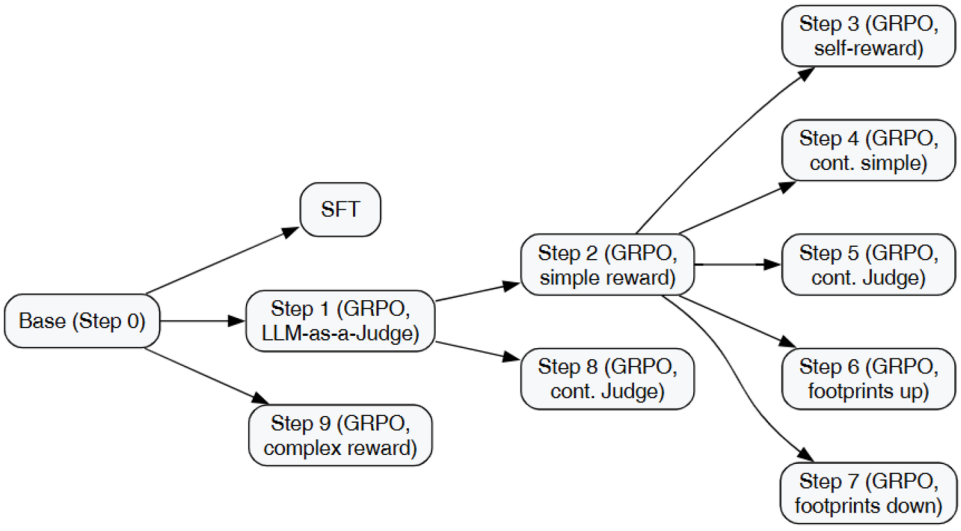


Fig. 7 Dag of training checkpoints and parent–child relations (labels match Table 1). The main path implements the two-stage curriculum from Sect. 4.5; branches add robustness or shape complexity

and (iii) the *min/median/max* of both the reward and the footprints count $\mathrm{FP}(M)$ among valid completions. Aggregates are in Table 1.

Results and Discussion

Having realized the curriculum from Sect. 4.5 and evaluated per Sect. 5.3, we now interpret the results; Sect. 5.5 will examine training dynamics, and Sect. 5.6 will validate on an external benchmark.

From instability to reliability. The base model succeeds sporadically (only $359/1248$ non-negative). SFT (Sect. 4.5, warm start) substantially improves syntactic stability and structural consistency. Stage A (Judge→GRPO) further reduces outright failures and tightens model shape (median footprints ≈14). Stage B (Footprints→GRPO) almost eliminates invalid completions while preserving median reward, indicating that verifiable behavioral feedback has "locked in" the intended control flow.

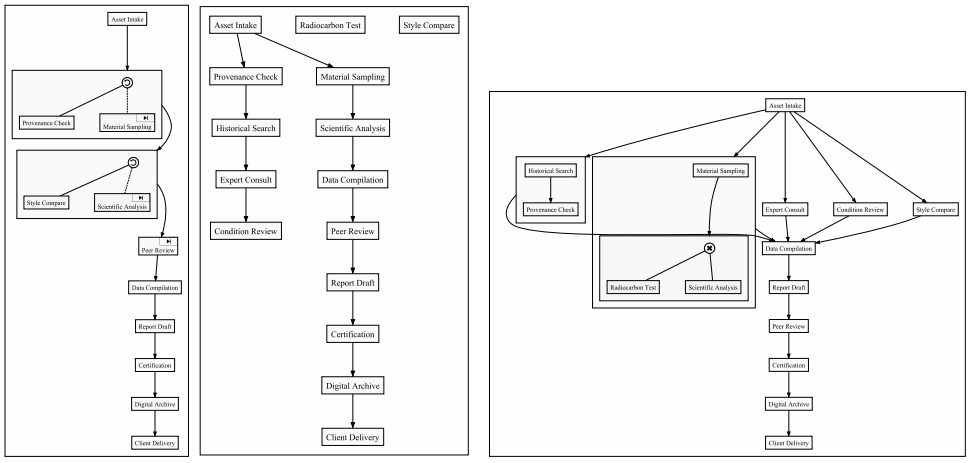
Robustness and tail suppression. Self-agreement (Step 3) maintains near-perfect validity while suppressing heavy-tail complexity (max footprints from 199 to 49), consistent with the consensus effect anticipated in Sect. 4.4.

Complexity control. Footprint shaping behaves as designed: $+\mathrm{foot}$ (Step 6) drastically increases structural richness (median footprints from 15 to 240) at some cost to reward, whereas $-\mathrm{foot}$ (Step 7) caps complexity (max 20) while keeping reward distribution comparable to Step 2. This exposes a practical "complexity knob".

Alternative continuations and ablation. Continuing the judge signal directly (Step 8) remains strong but less stable than the Stage B branch. Training solely with the composite reward from Base (Step 9) fails to bootstrap structure, underscoring the value of the staged plan in Sect. 4.5.

Qualitative comparison. Figure 8 illustrate that the RL checkpoint from Stage B better reflects the intended concurrency and downstream reporting order in a complex process description[1], whereas the baseline collapses distinct strands and misorders downstream

[1] DESCRIPTION: Detailed verification and authentication of antique assets for high-value collectors and museums, incorporating provenance research, material analysis, expert consultation, and condition assessment. The process



(a) Baseline. (b) RL checkpoint (Stage B). (c) Reference (generated by o4-mini).

Fig. 8 POWL comparison on a multi-strand investigation process. The RL model preserves intended concurrency and downstream ordering; the baseline serializes strands and introduces misordering

activities. This matches the goals of Sect. 4.1 (behavioral fidelity) and the choices in Sect. 4.2 (partial-order backbone plus operators).

Training dynamics

Beyond end-state metrics, we examine *how* quickly the staged plan converges.

Invalid rate and reward median. Figure 9 tracks (i) the number of structurally invalid generations and (ii) the median reward among valid completions over optimization steps. Stage A (judge) rapidly collapses invalids and raises the median reward to the target plateau; Stage B (footprints) drives invalids to near zero while maintaining the plateau.

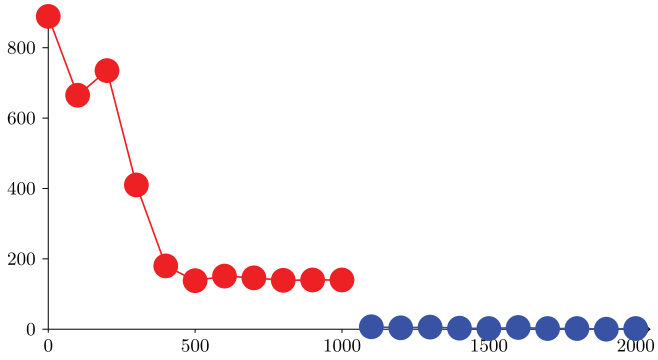
Validation on ProMoAI

We evaluate four models on the ProMoAI benchmark (Kourani et al. 2024a), a suite of 20 textual process descriptions paired with ground-truth logs and reference models: three unmodified baselines (Qwen2.5–Coder–3B/14B/32B) and our RL–trained checkpoint from *Step 2*. For each description we sample four POWL candidates and compute the token-based replay F-score; Table 2 reports, per description (labels 01–20), the minimum/mean/maximum across the four samples for each model.

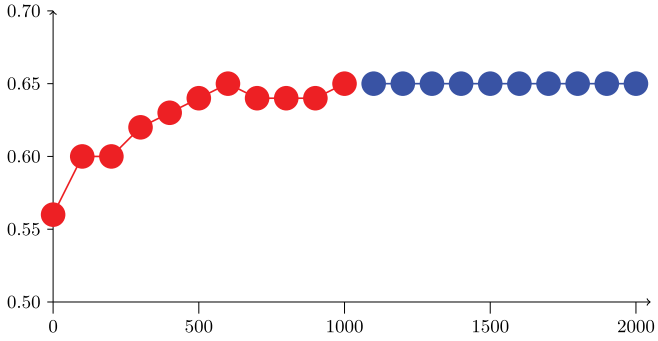
Three observations stand out. *First*, RL specialization yields the strongest average quality and robustness: our checkpoint attains macro averages of **min** = 0.398, **mean** = 0.584, and *best-of-4* **max**= 0.734, while solving *all* descriptions in this set (every row has a non-zero max). By contrast, Qwen2.5–Coder–3B remains at zeros throughout; 14B and 32B reach macro means of 0.317 and 0.393, respectively.

Second, larger baselines exhibit strong *peaks* but weak *stability*.

begins with initial asset intake, followed by multi-layered investigation including historical document cross-checking, scientific testing such as radiocarbon dating or spectroscopy, and comparative stylistic evaluation. Findings are then compiled into a detailed authentication report, which undergoes peer review by external specialists. The final step involves certification issuance and digital archiving of the asset's verified profile. ACTIVITIES: Asset Intake, Provenance Check, Material Sampling, Radiocarbon Test, Style Compare, Historical Search, Expert Consult, Condition Review, Scientific Analysis, Data Compilation, Peer Review, Report Draft, Certification, Digital Archive, Client Delivery.



(a) Structurally invalid generations over training (lower is better). Red: Stage A (judge). Blue: Stage B (footprints).



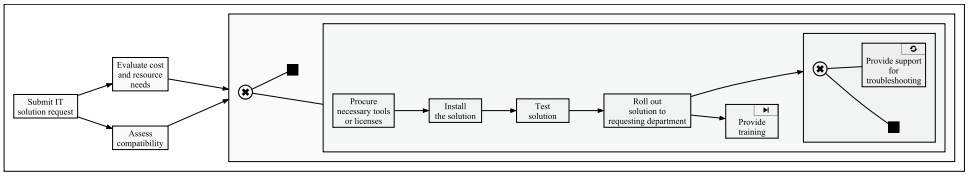
(b) Median reward among valid generations (higher is better). Stage A quickly reaches the plateau; Stage B maintains it while eliminating residual invalids.

Fig. 9 Training speed and convergence across the two RL stages

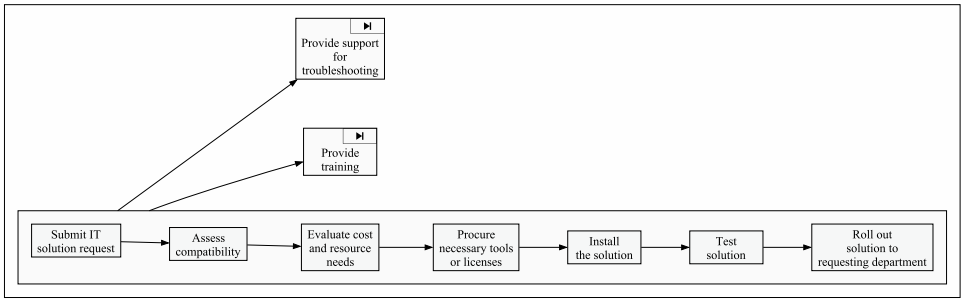
Qwen2.5–Coder–32B achieves the highest average best-of-4 (*max* = 0.754) yet has a very low average worst-of-4 (*min* = 0.046) and leaves three descriptions with all-zero runs; 14B shows a similar pattern (*max* = 0.711, *min* = 0.000; three all-zero rows). In contrast, our checkpoint combines a competitive *max* with a much higher *min* (0.398 on average), indicating substantially fewer catastrophic failures.

Third, RL improves *consistency*. For nine descriptions (labels 4, 5, 7, 10, 13–15, 19, 20) the *worst* of our four samples already exceeds 0.50, whereas neither 14B nor 32B achieves this for more than one label. Modest sampling brings tangible gains: for our checkpoint the average gap between best-of-4 and mean-of-4 is ≈0.15 (e.g., labels 1, 2, 8, 18 show improvements of 0.32–0.48). Overall, the updated results confirm that RL with verifiable behavioral rewards delivers higher average quality *and* markedly better reliability than strong, unspecialized code models.

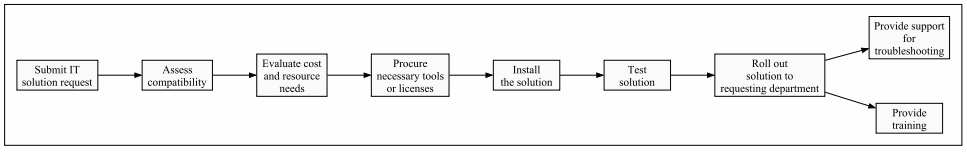
Comparison with proprietary models (GPT–4o and GPT–4o-mini). Under the same evaluation protocol (four samples per description; token-based replay F-score), GPT–4o averages around **0.74** on ProMoAI, while GPT–4o-mini achieves **0.72**. Our checkpoint's *best-of-4* macro average (**0.734**) effectively matches the reported GPT–4o average, despite using a much smaller open model. This supports the claim that RL specialization closes most of the gap to larger proprietary systems while retaining markedly higher reliability than unspecialized open baselines.



(a) Hand-crafted ground-truth POWL model



(b) POWL model generated by GPT-4o



(c) POWL model generated by our RL-trained checkpoint

Fig. 10 Comparison of POWL models for ProMoAI process description 10 (it solution request). The hand-crafted ground-truth model (top) is compared with models automatically generated from the same textual description by GPT-4o (middle) and by our RL-trained checkpoint (bottom)

For ProMoAI process description 10[2], all three POWL models shown in Fig. 10 capture the intended linear backbone of the IT solution request process: an employee submits a request, IT assesses compatibility, evaluates cost and resource needs, procures the necessary tools or licenses, installs and tests the solution, and finally rolls it out to the requesting department. The models generated by GPT-4o and our RL-trained checkpoint are similar to each other; in both, the activities "Provide support for troubleshooting" and "Provide training" can occur multiple times. This example illustrates how RL specialization can preserve the strong backbone produced by a powerful proprietary model.

Discussion

Our study shows that reinforcement learning provides a practical path to make large language models effective process modelers. The core advantage is that RL lets us optimize *behavior* rather than imitation alone. In Sect. 5.3 and Sect. 5.5 we observed that once the model is moved into a regime where a reasonable share of completions can be parsed, group-relative policy optimization rapidly reduces structurally invalid generations and stabilizes behavioral quality. Universal feedback from an LLM-as-a-Judge supplies nuanced guidance when textual specifications admit multiple faithful realizations,

[2] The process starts when an employee or department submits a request for an IT solution, such as new software or hardware. IT assesses the request, checking for compatibility with existing systems and evaluating cost and resource needs. If approved, IT procures the necessary tools or licenses, installs the solution, and tests it in a controlled environment. After successful testing, the new solution is rolled out to the requesting department. Training may be provided, and IT support remains available for troubleshooting.

while verifiable, footprints-based signals (Sect. 5.2) anchor learning to precise control-flow relations and protect against drift. The self-reward variant is particularly appealing for scalability: it supplies a usable signal even without references, encouraging solutions that are both correct and representative among peer samples. Shaping the number of footprints further demonstrates that RL can control structural richness, steering the policy toward simpler or more expressive models on demand. Finally, when evaluated on the ProMoAI benchmark (Sect. 5.6), the checkpoint trained with simple verifiable rewards achieves best-of-*4* performance close to GPT-4o while exhibiting fewer failed generations than strong proprietary baselines[3], underscoring the value of domain-specific reward design.

Limitations

Despite these advantages, several limitations remain. Reward quality is the central bottleneck. Judge-based rewards can encode bias (Szymanski et al. 2025), be sensitive to prompt framing, and are susceptible to reward hacking (Zhao et al 2025); verifiable rewards based on footprints, while precise, only reflect directly-follows and potential concurrency and may miss higher-order semantics. Our complex reward partially bridges this gap but still measures surrogates of modeling quality rather than full semantic equivalence. From a data perspective, the corpus of reference models used for SFT and some RL signals originates from LLM-generated designs, which may carry systematic biases from the generator and limit diversity. Computationally, while GRPO converged quickly once rewards were informative, training remains sensitive to the balance between the universal and verifiable components and to the regularization that keeps the policy near a fluent reference; aggressive shaping toward high footprint counts illustrated a trade-off between expressiveness and fidelity. Finally, our experiments target POWL; the extent to which the learned policy transfers to other notations (e.g., BPMN, process trees) or to settings with richer resource and data perspectives is not yet established.

Future work

These limitations suggest several directions for future work. Broadening the training distribution with human-authored models and additional benchmarks will be important to probe generalization. Finally, integrating automatic repair loops, by detecting and fixing local structural issues before scoring, may further reduce failure rates and make the training signal denser, enabling larger-scale runs with stronger base models.

Conclusion

In this paper, we demonstrated that reinforcement learning effectively specializes a pretrained large language model for process modeling tasks, significantly outperforming generic pretrained models and Supervised Fine-Tuning approaches. By combining verifiable feedback, based on structural validity and behavioral footprints, with universal judgments provided by an LLM-as-a-Judge, our RL approach substantially reduces invalid generations and improves behavioral accuracy of generated POWL models. Experiments on a dedicated corpus and the ProMoAI benchmark show that our RL-specialized

[3] 10 failed generations of our RL checkpoints against 25 failures of GPT-4o.

checkpoint attains results close to bigger proprietary models, while exhibiting fewer generation failures and providing control over model complexity. Future research should focus on incorporating richer semantic checks, structured decoding methods ensuring validity by construction, and extending evaluations across diverse modeling notations and complex process domains.

Acknowledgements
The authors gratefully acknowledge the computing time provided to them at the NHR Center NHR4CES at RWTH Aachen University (project number p0024464). This is funded by the Federal Ministry of Education and Research, and the state governments participating on the basis of the resolutions of the GWK for national high performance computing at universities (https://www.nhr-verein.de/unsere-partner).

Author contributions
A.B. wrote the main manuscript text. X.W. identified Group Relative Policy Optimization (GRPO) as a suitable algorithm for process mining and process modeling use cases. H.K. provided the foundational framework to generate sound models via the POWL formalism. W.M.P.v.d.A. provided useful suggestions that improved the study. All authors reviewed the manuscript.

Funding
Open Access funding enabled and organized by Projekt DEAL. The authors gratefully acknowledge the German Federal Ministry of Education and Research (BMBF) and the state government of North Rhine-Westphalia for supporting this work as part of the NHR funding.

Data availability
No datasets were generated or analysed during the current study.

Declarations

Ethical approval
This study does not involve human or animal subjects. Therefore, ethical approval is not applicable.

Competing interests
The authors declare no competing interests.

Received: 18 September 2025 / Accepted: 8 December 2025
Published online: 18 December 2025

References

Abbasi M, Khadivi M, Ahang M, Lasserre P, Lucet Y, Najjaran H (2025). An innovative data-driven and adaptive reinforcement learning approach for context-aware prescriptive process monitoring. arXiv preprint arXiv:2501.10543
Bai Y, Jones A, Ndousse K et al (2022) : training a helpful and harmless assistant with reinforcement learning from human feedback. CoRR abs/2204.05862
Berti A, Kourani H, van der Aalst WMP (2024) Pm-llm-benchmark: evaluating large language models on process mining tasks. In: ICPM workshops. Lecture notes in business information processing, vol 533. Springer, pp 610–623
Berti A, van Zelst SJ, Schuster D (2023) Pm4py: a process mining library for python. Softw Impacts 17:100556
Bousdekis A, Kerasiotis A, Kotsias S, Theodoropoulou G, Miaoulis G, Ghazanfarpour D (2023) Modelling and predictive monitoring of business processes under uncertainty with reinforcement learning. Sensors 23(15):6931
Bozorgi ZD, Dumas M, Rosa ML, Polyvyanyy A, Shoush M, Teinemaa I (2023) Learning when to treat business processes: prescriptive process monitoring with causal inference and reinforcement learning. In: CAiSE. Lecture notes in computer science, vol 13901. Springer, pp 364–380
Brennig K, Kaltenpoth S, Müller O (2024) Straight outta logs: can large language models overcome preprocessing in next event prediction? In: Business process management workshops. Lecture notes in business information processing, vol 534. Springer, pp 197–208
Cai R, Zheng C, Wang J, Li D, Wang C, Li B (2024) Reinforcement learning-based streaming process discovery under concept drift. In: CAiSE. Lecture notes in computer science, vol 14663. Springer, pp 55–70
DeepSeek-Ai, Guo D, Yang D et al (2025) : deepseek-r1: incentivizing reasoning capability in llms via reinforcement learning. CoRR abs/2501.12948
Dou S, Liu Y, Jia H et al (2024): Stepcoder: improving code generation with reinforcement learning from compiler feedback. In: ACL (1). Association for Computational Linguistics, pp 4571–4585
El-Aziz EA, Fathalla R, Shaheen M (2023) Deep reinforcement learning for data-efficient weakly supervised business process anomaly detection. J Big Data 10(1)
Gu J, Jiang X, Shi Z et al (2024): a survey on llm-as-a-judge. CoRR abs/2411.15594
Hundogan OA, Verhoef BJ, Theeven P, Reijers HA, Lu X (2025) Reinforcement learning for optimizing responses in care processes. Data Knowl Eng 157:102412
Kourani H, Berti A, Schuster D, van der Aalst WMP (2024a) Evaluating large language models on business process modeling: framework, benchmark, and self-improvement analysis. CoRR abs/2412.00023
Kourani H, Berti A, Schuster D, van der Aalst WMP (2024b) Process modeling with large language models. In: BPMDS/EMMSAD@ CAiSE. Lecture notes in business information processing, vol 511. Springer, pp 229–244

Kourani H, van Zelst SJ (2023) POWL: partially ordered workflow language. In: BPM. Lecture notes in computer science, vol 14159. Springer, pp 92–108
Kourani H, van Zelst SJ, Schuster D, van der Aalst WMP (2025) Discovering partially ordered workflow models. Inf Syst 128:102493
Lee H, Phatale S et al H.M.(2024) RLAIF vs. RLHF: scaling reinforcement learning from human feedback with AI feedback. ICML. OpenReview.net
Lee J (2024) Instructpatentgpt: training patent language models to follow instructions with human feedback. CoRR abs/2406.16897
Leonardi G, Montani S, Striani M (2025) Exploiting llms for supporting conformance checking on medical processes. In: AIME (2). Lecture notes in computer science, vol 15735. Springer, pp 224–229
Middelhuis J, Bianco RL, Sherzer E, Bukhsh Z, Adan I, Dijkman RM (2025) Learning policies for resource allocation in business processes. Inf Syst 128:102492
Norouzifar A, Kourani H, Dees M, van der Aalst WMP (2024) Bridging domain knowledge and process discovery using large language models. In: Business process management workshops. Lecture notes in business information processing, vol 534. Springer, pp 44–56
Ouyang L, Wu J, Jiang X et al (2022) D.A.: training language models to follow instructions with human feedback. NeurIPS
Rebmann A, Schmidt FD, Glavas G, van der Aa H (2024) Evaluating the ability of llms to solve semantics-aware process mining tasks. In: ICPM. IEEE, pp 9–16
Shao Z, Wang P, Zhu Q et al (2024) R.X.: Deepseekmath: pushing the limits of mathematical reasoning in open language models. CoRR abs/2402.03300
Shoush M, Dumas M (2023) Prescriptive process monitoring under resource constraints: a reinforcement learning approach. CoRR abs/2307.06564
Szymanski A, Ziems N, Eicher-Miller HA et al (2025) T.J.L.: limitations of the llm-as-a-judge approach for evaluating LLM outputs in expert knowledge tasks. In: IUI. ACM, pp 952–966
Yuan J, Grigori D, van der Aa H (2024) Enhancing predictive process monitoring using semantic information. In: ICPM workshops. Lecture notes in business information processing, vol 533. Springer, pp 293–305
Yuan W, Pang RY, Cho K et al (2024) X.L.: self-rewarding language models. In: ICML. OpenReview.net
Zhang S, Dong L, Li X et al (2023) S.Z.: instruction tuning for large language models: a survey. CoRR abs/2308.10792
Zhao Y, Liu H, Yu D et al (2025) S.Y.K.: one token to fool LLM-as-a-Judge. CoRR abs/2507.08794

Publisher's Note