applied sciences

MDPI

Article

Reusable Neural Guidance for Petri Net Alignments with Replay Verification

Alessandro Berti [1,*] and Wil M. P. van der Aalst [1,2]

[1] Process and Data Science (PADS), RWTH Aachen University, 52074 Aachen, Germany; wvdaalst@pads.rwth-aachen.de
[2] Celonis, 80333 Munich, Germany
* Correspondence: a.berti@pads.rwth-aachen.de

Featured Application

We present a reusable neural scorer for candidate alignments, with executability checked by replay and a verified upper bound on the optimal alignment cost.

Abstract

Alignments explain deviations between an event trace and a process model, but difficult traces can make exact computation slow. This paper studies whether one pretrained neural scorer can be reused to propose alignments for different Petri nets and activity vocabularies. LARA (Learned Alignment with Replay Assurance) encodes the Petri net and the complete trace, scores possible moves, and constructs a candidate using a decoder that tracks the marking. Independent replay checks executability and computes the candidate cost. Optional exact search establishes optimality or supplies a replacement. We explain the neural machinery, synthetic training procedure, and verification contract. Across three training seeds, greedy decoding produces 90.8% optimal candidates on the 512-row synthetic test set (sample standard deviation: 0.3 percentage points), which can be compared with 86.9% without guidance. All candidates pass replay. An optional single-log repair procedure raises the reference checkpoint's rate from 91.0% to 97.3%. The largest learned benefit concerns duplicate activity labels. Tests exclude training-isomorphic graphs, include cyclic structures, and evaluate held-out cases from three public logs. They reveal inconsistent structural gains and no real-log quality advantage from learned rankings. Common-boundary timings compare per-trace methods, while established approximations remain stronger than the greedy candidate policy. Certified mode runs exact search on every trace. The results support a limited role for reusable scoring, with guarantees supplied by process semantics.

Keywords: process mining; conformance checking; alignments; Petri nets; neural networks; reusable neural scoring; replay verification

Check for updates

Academic Editors: Julius Nganji and Kingsley Okoye

Received: 17 September 2026
Revised: 29 September 2026
Accepted: 4 October 2026
Published: 7 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

1. Introduction

Conformance checking is useful because a process model and an event log describe different things. The model states which executions are allowed; the log records what was observed. A difference between them may reveal an exception, an incomplete recording, or a model that needs revision. An alignment makes this difference concrete by relating each recorded event to an execution of the model and identifying the extra or missing behavior

needed to reconcile the two [1]. Its value lies in both a deviation cost and an explanation that can be replayed on the model.

For example, suppose that two transitions have the same activity label, but lead to different branches. The next event alone cannot identify the appropriate transition. Later events may reveal which branch gives the best explanation. An exact aligner explores alternatives and returns a minimum-cost explanation. This is an attractive contract, but exploration becomes expensive when many choices interact through concurrency, repetition, or invisible routing. Even when most traces are inexpensive, a few difficult variants can delay a complete analysis. Faster first results could make conformance checking more responsive, provided their meaning is clear.

This paper investigates learned guidance for that first result. A neural network can learn recurring relationships between process structure, observed events, and good alignment choices. Once trained, it can assign scores to moves on a new Petri net-and-trace pair. The difficulty is that a high score does not prove that a move is enabled, that a complete execution will reach the final marking, or that the resulting cost is minimal. Consequently, replacing formal searches with a predictor would change the meaning of the answer unless those obligations were handled explicitly.

LARA, short for *Learned Alignment with Replay Assurance*, assigns these obligations to different components, as outlined in Figure 1. The neural network proposes preferences. A decoder follows Petri net firing rules while constructing a candidate. An independent replay routine checks the complete candidate and computes its cost. An optional exact aligner then determines whether that candidate is optimal and replaces it if necessary. The basic principle is simple: a prediction may influence which explanation is tried first, while process semantics determines which claims can accompany it.

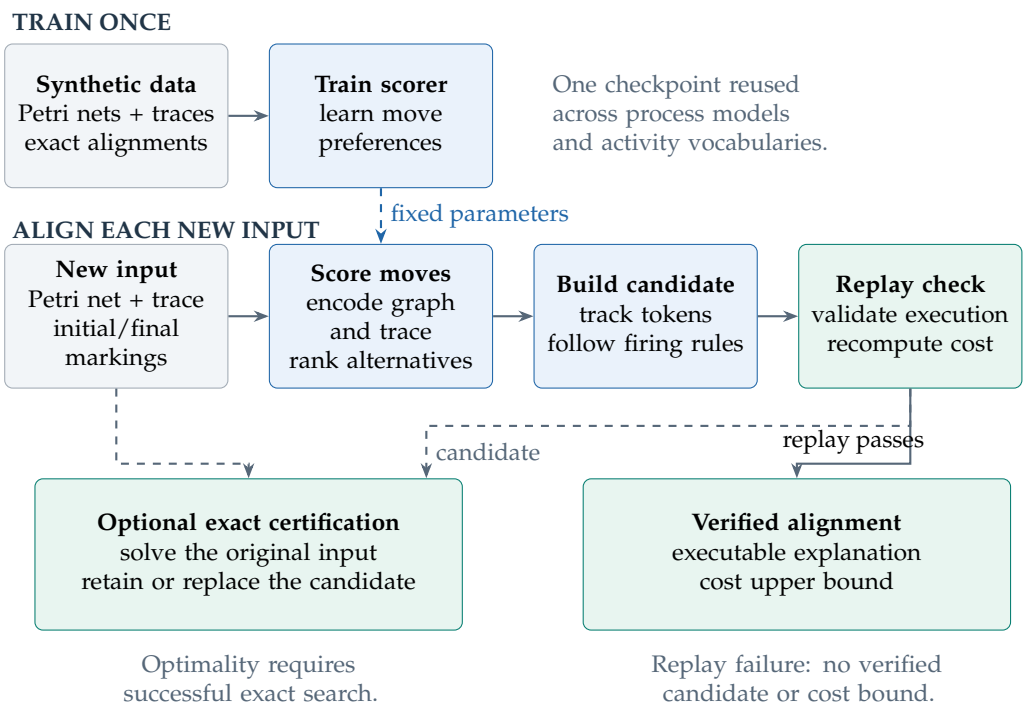


Figure 1. Overview of the LARA method. Synthetic inputs and exact alignments train a reusable neural scorer. For each new Petri net and trace, learned preferences guide candidate construction, and independent replay checks the result. Successful replay yields an executable alignment and a cost upper bound. Optional exact search solves the original input independently, retaining an optimal candidate or returning a replacement. Without a completed exact result, optimality remains unconfirmed. Blue denotes scoring and candidate construction; green denotes verification and exact certification; gray denotes inputs and training data.

The reuse question is central. Training a separate neural network for every process would create a substantial preparation burden. We therefore train one scorer on abstract synthetic process structures and reuse its parameters on other inputs. Activity names identify equal labels; the neural network does not interpret their business meaning. We describe LARA as a *reusable pretrained scorer*. Its single task, small corpus, and limited structural diversity do not support calling it a foundation model. The broader pretraining literature provides context for the reuse question [2,3].

The research contributions are **(C1)** a reusable scorer for Petri nets and traces that accepts variable size inputs and new activity vocabularies; **(C2)** an explicit verification contract that separates candidate construction, replay validity, and exact optimality; **(C3)** a synthetic supervision procedure that pairs equivalent process representations with shared observations and exact alignment targets; and **(C4)** an empirical study that isolates the contribution of learned guidance and measures its limitations against established alternatives. These are contributions to the design and analysis of a learned candidate pipeline, not claims of superior approximation quality or broad structural generalization. Section 2.7 details their scope.

The evaluation addresses three research questions, with their evidence specified in Section 2.6:

RQ1. Candidate validity and quality: How often does LARA produce a complete legal alignment, how often is its cost optimal, and how large are the remaining cost gaps?

RQ2. Contribution of machine learning: When do learned move rankings improve the same decoder compared with deterministic structural choices?

RQ3. Reuse and practical efficiency: With one fixed checkpoint, how do candidate quality and running time vary across representations, scales, and real event logs, and how do they compare with established approximation methods?

The empirical conclusion is similarly specific. Learned scores improve the same constrained decoder from 86.9% to 91.0% optimal candidates on the synthetic test split, with the strongest gain for duplicate label choices. All evaluated candidates pass replay. However, exact search has lower recorded latency on small inputs, and LARA ranks fifth of the six methods by optimal candidate rate in the comparison reported in Section 7.6. The four search or compression approximations all exceed its 91.0% rate. Across three training seeds, the mean controlled-test optimality is 90.8% with a sample standard deviation of 0.3 percentage points. Optional single-log repair raises the reference result to 97.3%. These gains do not establish broad superiority: the structural tests show inconsistent benefits from guidance, and held-out cases from three real logs show none. Section 7.9 reports these checks and timings with common processing boundaries. Certified mode runs an exact search on every trace; the proportion of candidates that are retained is therefore a quality measure, not a reduction in exact computation. These findings motivate a careful account of where machine learning contributes and where the decoder or exact backend does most of the work.

The rest of the paper is organized as follows. Section 2 introduces the formal and machine learning foundations and details the research questions and contributions. Section 3 reviews related work. Section 4 presents the LARA method, and Section 5 describes synthetic data generation and training. Section 6 explains the implementation and interactive workbench. Section 7 reports the experiments, Section 8 discusses the findings and limitations, and Section 9 concludes the paper.

2. Preliminaries, Research Questions, and Contributions

The method combines executable process semantics with a statistical predictor. To make the interface between them precise, this section first defines the alignment problem

and then introduces the small set of machine learning concepts used by LARA. The emphasis is on what each concept permits us to claim. These definitions also establish the research questions and the evidence needed to answer them.

2.1. Traces, Variants, and Labeled Petri Nets

A trace records the order of activities observed for one case. Let $\mathcal{A}$ be an activity alphabet, and let $\sigma = \langle a_1, \ldots, a_n \rangle \in \mathcal{A}^*$ be a finite trace. Its length is $|\sigma| = n$; the empty trace is ϵ. An event log $\mathcal{E}$ is a multiset of traces. A *variant* is a distinct activity sequence, and its frequency is the number of cases with that sequence. LARA studies control flow: timestamps, resources, and other event attributes do not enter the scorer or alignment cost.

We represent a labeled Petri net as $SN = (P, T, F, \lambda, m_0, m_f)$. Here, P is a finite set of places, T is a disjoint finite set of transitions, and $F \subseteq (P \times T) \cup (T \times P)$ is the arc relation. We use ordinary arcs with a weight of one in the formal exposition. The labeling function $\lambda : T \to \mathcal{A} \cup \{\tau\}$ assigns a visible activity or the invisible label $\tau \notin \mathcal{A}$. A marking $m : P \to \mathbb{N}$ assigns tokens to places, with $\mathbb{N}$ including zero. The initial and final markings are m_0 and m_f; $[p]$ denotes the marking with one token at p and zero elsewhere.

For $t \in T$, its preset ${}^\bullet t = \{p : (p, t) \in F\}$ contains its input places, and its postset $t^\bullet = \{p : (t, p) \in F\}$ contains its output places. The transition is enabled at m when $m(p) \geq 1$ for every $p \in {}^\bullet t$. Firing it produces the marking in Equation (1)

$$m'(p) = m(p) - \mathbf{1}[p \in {}^\bullet t] + \mathbf{1}[p \in t^\bullet], \qquad p \in P, \tag{1}$$

written $m \xrightarrow{t} m'$. In words, firing consumes a token from every input and produces a token in every output. An invisible transition follows the same firing rule without representing a recorded activity; firing can leave a marking unchanged, e.g., for a self loop. For a finite transition sequence $\rho \in T^*$, $m \xrightarrow{\rho} m'$ means successive enabled firings, with $m \xrightarrow{\epsilon} m$. A *complete firing sequence* starts at m_0 and reaches exactly m_f. We assume that at least one such sequence exists. This assumption ensures that an alignment can always be constructed by using log moves for the trace and model moves for complete model execution.

Distinct transitions may carry the same label. We therefore distinguish the identifier t from its visible label $\lambda(t)$. Figure 2 illustrates why: t_1 and t_2 both explain an event labeled b, but only t_1 directly leads to c. Matching the label alone does not identify a model execution. Invisible transitions create a related issue: an event may become executable only after unobserved routing steps.

The *complete visible language* $\mathcal{L}(SN)$ contains the label sequences of complete firing sequences after removing all τ labels. Two Petri nets are language-equivalent if their complete visible languages coincide. This is a behavioral relation, not a claim that their graphs or internal choices are identical. An *isomorphic renaming* is stronger: it changes identifiers through bijections on places and transitions while preserving arcs, labels, and markings. We use both relations to test whether candidate quality depends on a particular representation.

Process trees provide one source of synthetic Petri nets. A visible leaf represents an activity; sequence executes children in order, exclusive choice (XOR) selects one child, and parallel composition (AND) permits their interleavings. A loop permits repetition. The training generator uses sequence, XOR, and AND; the separate stress generator also includes loops. A fixed compiler converts these operators to Petri net fragments. We call its output *canonical block* only to name this particular compilation, without asserting a universal normal form.

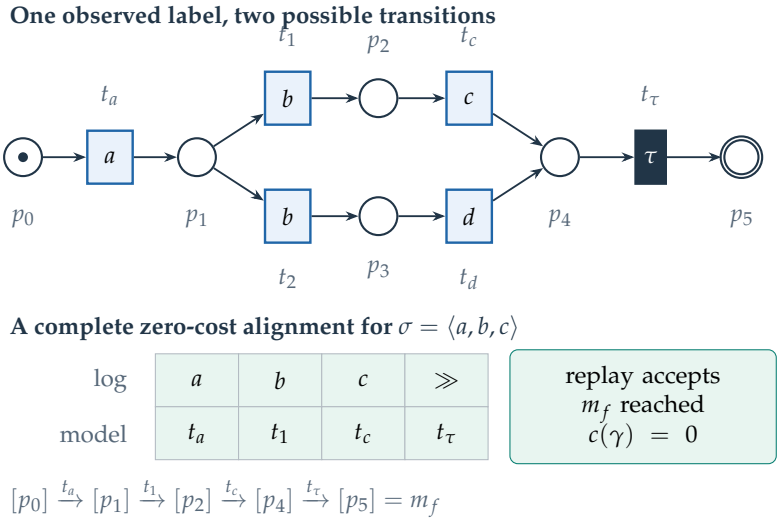


Figure 2. A running example of duplicate label ambiguity. The token begins at p_0 and must finish at p_5. Both t_1 and t_2 are labeled b, but their continuations differ. For $\langle a, b, c \rangle$, the upper branch gives a zero-cost alignment. The final invisible transition must still fire even though it consumes no events. Transition identifiers, activity labels, and alignment skips have separate roles.

2.2. Alignments: Executability and Deviation Cost

An alignment is needed because a trace may omit a model activity or contain an activity that the model cannot execute at that point. Let $\gg \notin \mathcal{A} \cup T \cup \{\tau\}$ denote a skip on one side of an alignment. The permitted moves and their costs are shown in Table 1, where $a \in \mathcal{A}$ and $t \in T$. A synchronous move consumes one observed event and fires a transition with that label. A log move consumes an event without changing the marking. A model move fires a transition without consuming an event. The pair $(\gg, \gg)$ is excluded because it explains nothing.

Table 1. Move vocabulary and the fixed costs used in this study. An invisible model move is an actual firing, whereas a skip means that nothing happens on that side.

Move	Form	Meaning	Cost
Synchronous	$(a, t), \lambda(t) = a$	The event agrees with a model firing.	0
Log	$(a, \gg)$	The event has no corresponding firing.	1
Visible model	$(\gg, t), \lambda(t) \neq \tau$	A modeled activity is absent from the trace.	1
Invisible model	$(\gg, t), \lambda(t) = \tau$	Unobserved routing advances the model.	0

For a finite sequence γ of permitted moves, its log projection $\pi_L(\gamma)$ removes the model components and skips; its model projection $\pi_T(\gamma)$ removes the log components and skips but retains all transition identities, including invisible transitions. Equation (2) states the projection conditions for a *legal* alignment:

$$\pi_L(\gamma) = \sigma \quad \text{and} \quad m_0 \xrightarrow{\pi_T(\gamma)} m_f, \tag{2}$$

and every synchronous move has matching labels. Thus, a legal alignment reconstructs the observed sequence exactly and describes a complete executable model path. Merely firing enabled transitions is insufficient: a sequence can stop in the wrong marking.

Let $c(\gamma)$ be the sum of move costs. Equation (3) defines the optimal alignment cost:

$$\delta(\sigma, SN) = \min_{\gamma \text{ legal for } (\sigma, SN)} c(\gamma). \tag{3}$$

The feasible set is nonempty according to the complete firing sequence assumption, and its costs are nonnegative integers, so the minimum is finite and attained even if there are infinitely many legal alignments. Under these costs, δ is the minimum number of visible insertions and deletions needed to explain the trace through a complete model execution. It does not measure monetary or business loss. Replacing one visible activity with another is represented by one log move and one visible model move, giving a cost of two; the optimal alignment may instead use a different model execution. Other cost policies may be appropriate in an application, but the empirical results here concern the policy in Table 1.

An optimum need not be unique. Independent activities can be ordered differently, and invisible routes may differ without changing the visible cost. Consequently, matching the exact teacher's move sequence is a stricter diagnostic than achieving the same cost. We will report both, but we use cost equality for optimality. For a candidate verified by replay $\widehat{\gamma}$, Equation (4) defines its *gap*:

$$g = c(\widehat{\gamma}) - \delta(\sigma, SN) \geq 0. \tag{4}$$

If an exact search does not finish, the gap is generally unknown. A returned legal candidate still supplies $0 \leq \delta \leq c(\widehat{\gamma})$. In particular, a legal zero-cost candidate already implies $\delta = 0$ and $g = 0$.

2.3. Exact Search, Replay, and Certificates

The shortest-path view explains why finding an alignment and checking one have different computational demands ([1], Chapter 7). A search state can be represented as (i, m), where $i \in \{0, \ldots, n\}$ events have been consumed and m is the current marking. Synchronous and log moves require $i < n$, consume a_{i+1}, and advance i by one. Synchronous and model moves require an enabled transition and update m by Equation (1); a synchronous move additionally requires its label to equal a_{i+1}. The search starts at $(0, m_0)$ and ends at (n, m_f). Equivalently, one constructs a synchronous product of the Petri net of the process and a linear Petri net for the trace. The path cost is the alignment cost. A finite Petri net can have infinitely many reachable markings. Existence of an optimum therefore does not in and of itself guarantee termination of this search, particularly in the presence of unbounded zero-cost invisible behavior.

A* prioritizes states by the sum of the cost already incurred and an estimate of the remaining cost. An estimate is *admissible* when it never overestimates the true minimum remaining cost. Marking equation relaxations obtain useful lower bounds by requiring token balance while relaxing parts of the firing order constraints [4]. They can guide an exact search because their relationship to the original problem is formally justified. LARA's neural scores have no such property and are never substituted for admissible bounds.

Replay solves a simpler task: given a proposed sequence, check each move in order. It confirms the labels, enabledness, complete trace consumption, and final marking and then sums the stated costs. It does not search for an alternative. The amount of work follows the supplied sequence and its incident arcs rather than the full space of alternative executions. This makes replay a practical check even when finding the best sequence is difficult.

There are therefore three distinct levels of evidence. A raw proposal may fail replay. A proposal that passes replay is an executable explanation with a verified upper bound. Finally, if an exact backend proves that its cost equals δ, it is an optimal explanation. Nonnegative costs also imply that any legal zero-cost alignment is mathematically optimal; the certification implementation nevertheless calls the exact backend in certified mode. We keep this implementation fact separate from the mathematical observation.

2.4. Machine Learning Preliminaries

The neural component learns recurring patterns from solved alignment problems. In *supervised machine learning*, each input is paired with a desired output, or *target*. Here, the input is a Petri net-and-trace pair, and an exact aligner supplies the targets; this supervising algorithm is called the *teacher*. A neural network is a numerical function f_θ whose parameters θ comprise adjustable weights and biases. Training changes them to reduce a *loss*, a numerical penalty for disagreement with targets. LeCun et al. [5] provide an overview of learned representations and their training. After training, *inference* evaluates the function with fixed parameters. One evaluation is a *forward pass*; a saved parameter set and its architecture settings form a *checkpoint*.

A *feature* is a numerical description of an input property, such as the number of arcs incident to a place. A *tensor* is a numerical array: a scalar, vector, and matrix have zero, one, and two indexing axes, respectively. Its *shape* gives the size of each axis. An *embedding* is a learned vector representation; an embedding table associates each discrete label ID with one such vector. An *encoder* transforms input features and embeddings into representations that include context. Their coordinates need not have individually interpretable meanings. Sharing parameters across objects of each type permits variable size inputs while keeping the parameter count fixed.

Neural computations are arranged in *layers*. An affine map $x \mapsto Ax + b$ combines coordinates using learned weights A and a bias b; a linear map omits the bias. A nonlinear *activation* lets stacked layers represent functions that a single affine map cannot express. LARA uses the Gaussian error linear unit (GELU), a smooth activation [6]. A *feedforward* neural network applies such maps and activations in sequence. Its intermediate vectors are *hidden representations*; depth counts layers and width describes the number of coordinates in a representation. An *output head* maps a shared representation to a particular prediction, such as a move score.

A *graph neural network* updates representations using graph structure. In *message passing*, a node aggregates information from its neighbors; typed messages use different transformations for different edge relations [7]. Thus, a place feeding a transition can be distinguished from a place receiving its output. Aggregation by a sum or mean is also called *pooling*. A weighted mean lets different objects contribute unequally. *Concatenation*, used when combining graph and trace summaries, instead joins vectors end to end.

Equation (5) defines *attention*, which combines information using weights computed from the input [8]:

$$\mathrm{Att}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V. \tag{5}$$

Rows of Q are *queries*, representing objects seeking information; rows of K are *keys*, against which they are compared; rows of V are the *values* to combine. Learned projections produce these arrays. The query and key width are d_k, and the factor $\sqrt{d_k}$ controls dot product magnitudes. Softmax acts row by row, with $\mathrm{softmax}(z)_j = e^{z_j} / \sum_k e^{z_k}$. Its nonnegative entries sum to one, making each output a weighted combination of values.

In *self attention*, queries, keys, and values come from the same collection. In *cross attention*, queries come from one collection and keys and values from another, such as events querying transitions. *Multihead attention* (MHA) combines several attention computations with distinct projections. These attention heads gather context; they are different from the prediction heads that output move scores. A *transformer block* combines attention with a feedforward network [8]. Its *residual connections* add an update to an earlier representation. *Layer normalization* (LN) centers and rescales each representation using its own coordinate mean and variance, followed by learned scales and offsets [9].

A *positional embedding* records an event's position so that attention can use trace order. A *mask* excludes selected entries from a calculation. A causal mask blocks access to later events; LARA omits it because the complete trace is available. Its label compatibility mask excludes transitions whose labels do not match an event. Neither context exchange nor this label mask checks whether a transition is enabled: the candidate decoder uses the current Petri net marking for that purpose.

A raw prediction score is a *logit*. Softmax converts competing logits into a distribution for a categorical choice, such as selecting one transition. For a binary decision, the *sigmoid* $1/(1+e^{-z})$ converts one logit z to a number between zero and one. Such normalization does not establish *calibration*: predictions assigned a probability of 0.8 would be calibrated if the stated event occurred approximately 80% of the time among them [10]. LARA uses logits as rankings, not probabilities of complete alignment optimality.

Cross entropy compares a target distribution with predicted probabilities. For a target that assigns all mass to one class, also called a *one-hot target*, the loss is the negative logarithm of the probability assigned to that class. Binary cross entropy (BCE) applies this comparison to a yes/no target. *Label smoothing* replaces a fully concentrated target with a less extreme distribution [11]; Section 5.8 specifies LARA's categorical and binary variants. *Regression* instead predicts a numerical target, such as a cost. The smooth-L1 loss used here penalizes small errors quadratically and larger errors linearly, limiting the influence of large deviations.

An *auxiliary objective* trains an additional output alongside the main prediction, usually through shared representations. In LARA, a *router* assigns each event or transition a distribution over internal groups, called *latent regions*. Latent means that these groups have no directly supplied region labels. Entropy measures how spread out an assignment is; low entropy favors concentration, while a balancing term discourages assigning everything to one region. Some auxiliary heads use the *softplus* activation, $\log(1+e^{x})$, to produce positive values. Positivity alone does not make a predicted endpoint a valid cost bound. Section 5.9 explains which auxiliary outputs receive supervision and which remain unused.

Data have three distinct roles. The *training split* updates parameters; the *validation split* selects settings and checkpoints; the *test split* assesses the selected model on held-out inputs, meaning inputs reserved from fitting and selection. *Generalization* is successful prediction on such new inputs. *Overfitting* occurs when a model fits peculiarities of training examples that do not carry over. Architecture choices and training settings, such as embedding width and learning rate, are *hyperparameters*, distinct from the weights learned by optimization. An *epoch* is a full training pass; a *batch* or mini-batch groups examples for one parameter update.

Backpropagation applies the chain rule through neural computations to obtain gradients, which describe how parameter changes affect the loss. An *optimizer* uses these gradients to update parameters; the *learning rate*, or step size, controls the update scale. AdamW adapts updates using running averages of gradients and squared gradients, and separately shrinks weights through *weight decay* [12]. *Gradient accumulation* combines contributions from separately processed examples before an update. *Gradient clipping* rescales a gradient whose Euclidean length exceeds a threshold, limiting extreme updates. These operations train the scorer; they do not differentiate through LARA's discrete candidate search.

Regularization aims to improve generalization by restricting or perturbing fitting. Dropout randomly suppresses some neural activations during training and is disabled during evaluation [13]. *Data augmentation* creates further training views while preserving the intended targets; LARA consistently remaps label IDs in a Petri net and its trace. A learning rate schedule reduces the step size when validation loss stops improving. *Early stopping* ends training after insufficient improvement; its *patience* counts the tolerated epochs, and the improvement threshold defines what counts as progress. A period without

such progress is a *plateau*. Selecting a checkpoint by validation loss keeps test outcomes outside this decision.

An *ablation* removes a component while retaining the surrounding procedure. A *baseline* is a reference method against which performance is compared. In a paired comparison, the alternatives process the same inputs. LARA's guided/unguided comparison therefore tests the contribution of learned rankings while preserving the decoder, move costs, and replay conditions.

2.5. Pretraining, Reuse, and Distribution Shift

Reusable models matter here because a new process should not require another exact labeled training campaign before candidate generation can begin. *Pretraining* fits the scorer on many synthetic Petri net and trace pairs. *Fine tuning* would subsequently update its weights on a target process. LARA's transfer experiments use the pretrained checkpoint without fine tuning, which we call zero shot application with respect to that target process. It still receives the target Petri net and trace as inputs.

Foundation models are commonly associated with broad pretraining and adaptation to a range of downstream tasks [2]. Process specific proposals consider reusable models for event data and process representations [3]. LARA adopts parameter reuse within one alignment task; we do not classify it as a foundation model. Its approximately 1.7 million parameters, small synthetic corpus, fixed costs, and limited training motifs do not demonstrate broad process mining competence. It is also not a language model: a label such as "approve application" is treated as an equality identifier, without tokenizing or interpreting its words.

Distribution shift occurs when deployment inputs differ from training inputs. Relevant shifts include longer traces, larger Petri nets, new structural motifs, or different label vocabularies. An exact semantics check continues to have the same meaning under such shifts, but learned rankings may become worse. Evaluating reuse therefore requires separate measurements of candidate legality, optimality, and latency. Acceptance by replay demonstrates successful execution of the candidate, not statistical generalization of the scorer.

2.6. Research Questions

The definitions above make it possible to ask questions that distinguish the predictor's value from the guarantees of its wrapper. The study addresses three research questions:

RQ1. Candidate validity and quality: How often does candidate generation produce a complete legal alignment, how often is its cost optimal, and how large are its remaining cost gaps? Replay acceptance, exact cost agreement, and gap distributions provide the evidence in Sections 7.2 and 7.8.

RQ2. Contribution of machine learning: When do learned rankings improve the same decoder compared with deterministic structural choices? A paired ablation and representation-specific analysis isolate effects on duplicate labels and identifier sensitivity in Sections 7.3 and 7.4.

RQ3. Reuse and practical efficiency: How do quality and running time change across representations, scales, real event logs, and established approximation methods? Sections 7.4–7.7 assess these changes while keeping the pretrained parameters fixed.

These questions do not assume that a neural candidate is preferable to an exact search. A useful answer may identify inputs for which its inference cost buys little or no improvement. Nor does a high retained-candidate rate answer whether certification is faster: that requires measuring a solver that actually uses the candidate to reduce exact work.

2.7. Contributions and Scope

The contributions are organized around an implementable alignment pipeline and evidence about its limits:

C1. A reusable scorer for Petri nets and traces: One neural network for graphs and sequences predicts transition-aware move preferences for variable size inputs and previously unseen label vocabularies, without training a neural network per target Petri net (Section 4.2). Accepting such inputs does not establish accurate rankings on unseen structural classes.

C2. An explicit verification contract: The scorer is integrated with decoding that tracks the marking, independent replay, and exact-cost comparison. The resulting output states distinguish a proposal, a verified upper bound, a certified candidate, and an exact replacement (Section 4.5). This contract can also wrap candidates produced without a neural network.

C3. A controlled synthetic supervision procedure: Behavior families pair different representations of the same visible behavior, reuse corrupted observations across the pair, and obtain concrete transition targets from exact alignments (Section 5).

C4. An empirical attribution and transfer study: The evaluation compares guided and unguided decoding, representation pairs, larger synthetic inputs, existing approximations, and real event logs. It reports where machine learning helps, where it has no measured effect, and where competing methods perform better (Section 7).

The work concerns completed control flow traces on labeled Petri nets with specified initial and final markings. It does not establish guarantees for data aware or object centric alignments, online prefix prediction, arbitrary cost policies, or learned lower bounds. The implementation described in Section 6 makes these contributions inspectable through command line experiments and an interactive workbench.

3. Related Work

Several research traditions address the cost of conformance checking, and machine learning enters this landscape with different possible roles. To locate LARA precisely, this section compares methods by the object they return, the source of their guarantees, and what they reuse across problems. This distinction matters more than whether a method is described broadly as fast, approximate, or intelligent. Table 2 summarizes the main relationships.

Table 2. Positioning by output and reuse. Guarantees for a particular approximation depend on its algorithm and configuration; the table does not attribute one common guarantee to the entire family.

Approach	Main Object	Reused Information	Relation to LARA
Exact alignment	Minimum-cost executable alignment	Model structure and search heuristics	Supplies teacher and certificate.
Decomposition	Local analyses and recomposed results	Structural fragments and their interfaces	Formal fragments differ from latent regions.
Search or compression approximation	Candidate alignment or approximation	Bounded context, repeated structure	Direct candidate generation competitors.
Subset/edit distance	Conformance estimates and bounds	Representative model executions	Reuses behavior for a process.
Predictive monitoring	Future event, time, or outcome	Learned event representations	Different target and information horizon.
LARA	Candidate verified by replay; optional exact result	One pretrained Petri net and trace scorer	Machine learning ranks; semantics checks.

3.1. From Token Replay to Optimal Alignment

Conformance checking connects event data to executable models through techniques including token replay and alignment ([1], Chapter 4). Token replay diagnoses discrepancies through missing and remaining tokens. Such diagnostics and a globally minimum-cost alignment are different output objects.

Cost-based alignment expresses deviations through synchronous, log, and model moves [14]. The broader conformance-checking framework treats the resulting correspondence as both a measurement tool and a diagnostic explanation [1]. LARA preserves this transition-aware alignment object. Its verifier does not insert tokens to make a proposed firing possible: it checks whether the supplied model projection really executes from the initial to the final marking. This is why replay in our pipeline is a feasibility check on a candidate, rather than a substitute fitness metric.

3.2. Exact Search and Structural Decomposition

Building on the shortest-path formulation in Section 2.3, marking-equation and extended marking-equation methods accelerate searches using formally justified estimates of remaining cost [4]. More recent exact systems, including REACH [15], continue to improve search efficiency through algorithmic treatment of explored states and model information. These are important alternatives whenever the requirement is a certified optimum. LARA uses an exact backend without modifying its search procedure, so its results should not be interpreted as a comparison against every state of the art exact implementation.

The practical motivation for approximate candidates depends on the input and implementation. Our small input timings show that exact search can be faster than neural inference and candidate construction (Section 7.9). We therefore assess running time empirically alongside candidate quality.

The survey by Genga and Winter [16] identifies decomposition as a separate strategy: smaller model segments reduce the search space, yielding partial alignments or a global alignment obtained by recomposition. This distinction helps delimit LARA's auxiliary latent regions. They have no corresponding decomposition theorem and do not generate recomposable local alignments. Although the code includes router and local head terminology, the evaluated decoder constructs one global candidate from move scores.

3.3. Approximation by Restricted Search, Compression, and Reuse

Approximation methods exploit different sources of regularity. Fixed horizon alignment combines an executable prefix with a marking equation estimate of the remaining behavior before committing [17]. Discounted alignment changes the influence of later deviations through a discounted objective [18]. Sliding window methods divide long traces into smaller search problems and retain selected boundary candidates, using global structural information about the remaining trace to rank them [19]. These methods reduce search effort through local commitments or a modified objective, which can yield a higher cost under the original objective. LARA also makes bounded search decisions; its learned ranking uses the complete trace as context.

Compression is another route. Tandem repeat methods exploit repeated trace segments to reduce the alignment problem before reconstructing a result [20]. They require no neural training and can be strong choices when traces contain suitable repetition. We include a tandem repeat implementation in the candidate quality comparison in Section 7.6.

Subset selection and edit distance approximation estimate conformance values and provide upper and lower bounds using a subset of model behavior [21]. This behavior can be obtained by aligning selected log traces or by simulating the model. The reusable object is therefore a collection of representative model executions, whereas LARA reuses

neural network parameters learned over synthetic Petri nets. The PM4Py variant evaluated here additionally retains complete transition sequences to reconstruct executable candidate alignments, including invisible transitions. Representative executions must cover relevant behavior, while a pretrained scorer must generalize to the target structure and observation patterns. Our subset comparison uses the concrete configuration in Section 7.6; it does not establish a best possible performance for all subset sizes or representative selection policies.

Padró and Carmona [22] combine relaxation labeling on a partial order representation with local optimal search to obtain executable, possibly suboptimal alignments. This is a particularly relevant precedent for separating a correspondence heuristic from completion under process constraints. LARA instead learns graph and trace scores across generated instances, then uses bounded marking search and independent replay. Executability checks are valuable for both approaches; they are not a novelty exclusive to neural methods.

3.4. Neural Networks for Event Data and Process Structure

Neural process mining research provides useful representations but often solves a different task. Recurrent models carry a hidden state from one event to the next; predictive monitoring uses this state to estimate future events, timestamps, or remaining execution time from prefixes [23]. ProcessTransformer applies transformer-based sequence modeling to predictive monitoring [24]. Such predictions do not, by themselves, identify an executable minimum deviation explanation against a given Petri net. LARA has the complete trace available and returns an alignment whose transition identities can be replayed.

Machine learning on graphs brings model structure into the numerical representation. Sommers et al. [25] trained a process discovery technique on synthetic log/model pairs and used graph neural networks to produce Petri nets. This established a useful connection to synthetic supervision and machine learning that transfers across process structures. Its target is a discovered process model, while LARA assumes the model is supplied and learns preferences for explaining a trace against it. Message passing frameworks [7] and attention [8] supply general computational building blocks; the alignment-specific design lies in input compatibility, transition-aware targets, constrained decoding, and assurance.

The survey and research agenda developed by Genga and Winter [16] examines artificial intelligence in conformance checking and identifies further opportunities. In this setting, it is useful to distinguish predicting a scalar conformance score, producing a diagnostic object, and proving a property of that object. LARA addresses the second task through learned proposal and constrained construction, while assigning the proof obligations to replay and exact search. It does not infer that successful prediction alone provides conformance assurance.

3.5. Reusable Models and Search Guidance

The foundation model literature studies pretraining for broad reuse. Bommasani et al. [2] discuss this across tasks and modalities, while Rizk et al. [3] consider business process data. LARA addresses the narrower question of reusing one scorer across alignment inputs. Its evidence concerns one task, synthetic motifs, and three target logs. We therefore use "reusable pretrained scorer" for LARA and reserve "foundation model" for the broader research context.

More broadly, machine learning for combinatorial optimization studies how learned decisions can complement established solvers [26]. Neural A* [27], for example, coupled learned guidance with a differentiable search procedure in path planning. The common motivation is to learn useful preferences over expensive choices. LARA's integration is simpler: it does not differentiate through A*, alter the exact objective, or establish admissibility of learned values. It learns a first candidate and checks it afterward. A

future integration that uses a verified incumbent within exact search would be a different algorithm requiring its own evaluation.

This comparison sets the empirical expectation. LARA should be assessed against exact searches, against the same decoder without machine learning, and against existing approximations. Its contribution cannot be established from a high legality rate alone, because semantics enforces the acceptance condition. The key evidence is whether learned rankings improve useful candidate quality enough to justify their inference cost, and how that evidence changes across representations and datasets.

4. Method: From a Petri Net and Trace to a Verified Alignment

Given a labeled Petri net, its initial and final markings, and a complete event trace, LARA proposes an alignment and checks what can be claimed about it. The contribution is a reusable neural network embedded in a pipeline whose correctness comes from explicit process semantics. One checkpoint is trained on synthetic Petri nets and traces with exact alignment targets, then reused across process models and activity vocabularies without training a new neural network for each target process. Section 5 describes this offline preparation.

At inference time, a graph encoder represents the places, transitions, and arcs, while a trace encoder reads the complete event sequence. Exchanging context between these representations allows later events to inform choices among transitions with the same label. A single forward pass assigns preferences to synchronous and model moves. A decoder then constructs a candidate by firing enabled transitions, using bounded searches for intervening model moves and maintaining the current marking. The scores guide these choices; label equality and token availability determine which moves are permitted.

Independent replay checks every firing, complete trace consumption, and the final marking before reporting a candidate as executable and its cost as an upper bound. Optional exact search establishes the minimum cost, retaining an optimal candidate or supplying an exact replacement. If exact search does not finish, a legal candidate keeps its verified cost but receives no optimality certificate. Figure 3 summarizes these stages. The five subsections below specify each stage's input, processing, and output; the exact backend performs its own search and does not use the candidate to accelerate certification.

4.1. Input Representation

Input. The representation stage receives the labeled Petri net $SN = (P, T, F, \lambda, m_0, m_f)$, the complete trace σ, and the move costs from Table 1. The numbers of places, transitions, and events may vary between inputs.

Processing. Equation (6) collects the numerical arrays that preserve the distinctions relevant to alignment:

$$X = (X_P, X_T, E, r, z_T, z_\sigma, C). \tag{6}$$

Table 3 summarizes these objects. At the software interface, activity labels are strings, invisible labels are represented by `None`, and place and transition identifiers are unique strings within each node type. Activity labels and transition identifiers must avoid PM4Py's reserved skip string ». Place and transition identifiers are sorted to define tensor indices; their strings are not embedded as semantic features. This convention makes the mapping between scores and concrete transitions recoverable during decoding.

Let $d^-(v)$ and $d^+(v)$ be the incoming and outgoing arc counts of node v. Equations (7) and (8) define the place and transition feature rows:

$$x_p = (m_0(p), m_f(p), d^-(p), d^+(p), d^-(p) + d^+(p)), \tag{7}$$

$$x_t = (\mathbf{1}[\lambda(t) = \tau], d^-(t), d^+(t), c_M(t), c_S(t), s_t), \tag{8}$$

where $c_M(t) = \mathbf{1}[\lambda(t) \neq \tau]$, $c_S(t) = 0$, and $s_t = \mathbf{1}[{}^\bullet t \cap t^\bullet \neq \emptyset]$. The zero synchronous cost feature is also stored for invisible transitions, although they cannot synchronize with events. Initial and final markings tell the neural network where execution starts and where it should end. Intermediate markings are not input features: they are maintained later by the decoder.

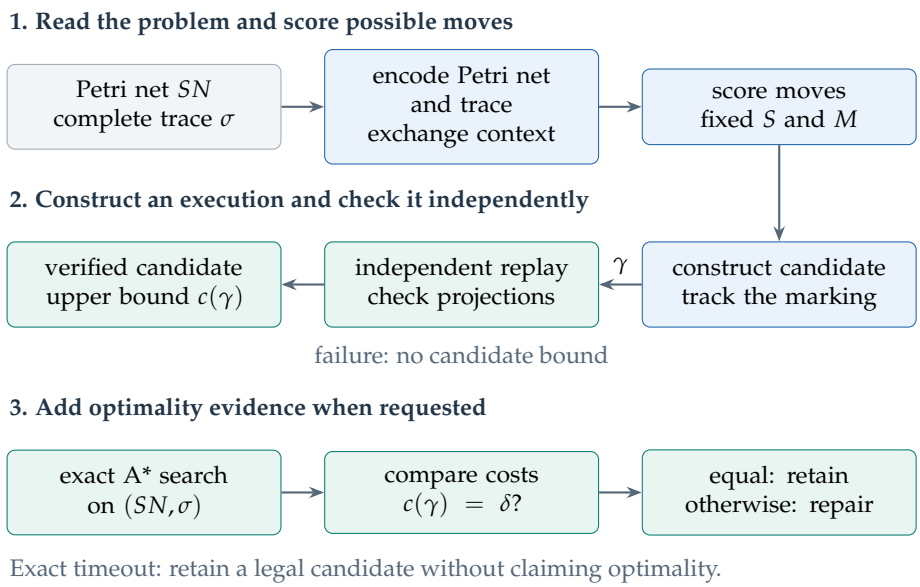


Figure 3. The inference pipeline and its sources of evidence. Blue components produce preferences and a candidate; green components check the candidate or establish an optimum. The upper-bound output is available after replay. Exact search is an additional computation, and a timeout does not turn an unchecked optimum into a certificate. The trained checkpoint is reused across Petri-net-and-trace pairs.

Table 3. Neural inputs. A tensor is a numerical array; its dimensions indicate which process objects index its entries. The teacher alignment is absent from this list.

Object	Shape	Contents and Purpose
X_P	$\lvert P\rvert \times 5$	Initial/final tokens and place degrees.
X_T	$\lvert T\rvert \times 6$	Visibility, transition degrees, costs, and shared input/output context.
E, r	$2 \times 2\lvert F\rvert; 2\lvert F\rvert$	Directed edge endpoints and four relation types.
z_T	$\lvert T\rvert$	Integer label IDs for transitions.
z_σ	n	Integer label IDs for observed events.
C	$n \times \lvert T\rvert$	Exact event and transition label compatibility.

Graph nodes are places followed by transitions. Every original arc is represented in both directions, with four types: place-to-transition consumption, transition-to-place production, reverse consumption, and reverse production. These reverse edges transmit information about a transition's continuation back toward it. In the running example, the two b transitions have the same immediate input place; information from their different output branches helps distinguish them. The encoder reads the graph directly and does not enumerate reachable markings.

To handle labels outside the training vocabulary, Equation (9) converts a visible label to an integer:

$$h(a) = 1 + \big(\mathrm{int}(\mathrm{BLAKE2b}_8(\mathrm{UTF8}(a))) \bmod 8191\big), \qquad h(\tau) = 0. \tag{9}$$

The eight-byte digest is interpreted as an unsigned big-endian integer. The resulting ID selects one of 8192 learned embedding rows. Nearby IDs do not indicate similar activities. This bounded representation permits a fixed checkpoint to accept arbitrary label strings, but collisions are possible: two different strings can select the same row.

Equation (10) defines the compatibility matrix from the exact label relation:

$$C_{ij} = \mathbf{1}[a_i = \lambda(t_j) \neq \tau]. \tag{10}$$

It is computed from the original labels, not their hashes. A hash collision can affect a preference but cannot authorize a false synchronous match. Conversely, $C_{ij} = 1$ does not imply that t_j is enabled. Label compatibility is checked in the score matrix; token availability is checked in the decoder. Keeping these conditions separate prevents statistical encoding choices from changing the move vocabulary.

The runtime can inspect integer arc weights, but the neural features above do not explicitly encode those weights. Our formal and empirical claims therefore concern the ordinary Petri net setting used by the study. Passing a richer Petri net object through a software interface would not by itself establish learned generalization to that setting.

Output. The stage returns the tuple X in Equation (6), together with the ordered identifiers that link tensor rows to concrete places and transitions. The neural network receives structural features and label encodings; the decoder retains the original labels and Petri net semantics for exact checks.

4.2. Encoding the Graph and the Complete Trace

Input. The graph and trace encoders receive the feature arrays, typed edges, and label IDs in X, along with the fixed checkpoint parameters. Their purpose is to represent context that a local label match cannot see. The embedding dimension is $d = 96$.

Processing. Learned affine projections of x_p and x_t initialize graph vectors. A node-type embedding distinguishes places from transitions; transitions also receive their label embedding. Equation (11) gives these initial vectors in compact notation:

$$h_p^{(0)} = A_P x_p + b_P + e_P, \qquad h_t^{(0)} = A_T x_t + b_T + e_T + e_{h(\lambda(t))}. \tag{11}$$

The symbols $A_P, A_T, b_P, b_T, e_P, e_T$ and the label table e are learned parameters. They are shared by all input Petri nets. The model therefore learns transformations of structure and labels rather than a separate vector for each named transition in one process.

Two graph blocks combine global self attention with typed local messages. The columns of E store edge endpoints and the corresponding entries of r store their types. For a node v, let $\mathcal{N}_v = \{(E_{1k}, r_k) : E_{2k} = v,\ 1 \leq k \leq 2|F|\}$. Writing H for the matrix of current node vectors, Equation (12) defines local messages, Equation (13) combines them with attention, and Equation (14) applies the feedforward update:

$$\mu_v = \frac{\sum_{(u,r)\in\mathcal{N}_v} W_r h_u}{\max(1, |\mathcal{N}_v|)}, \tag{12}$$

$$\widetilde{H} = \mathrm{LN}\big(H + \mathrm{Drop}(\mathrm{MHA}(H, H, H) + \mu)\big), \tag{13}$$

$$H' = \mathrm{LN}\big(\widetilde{H} + \mathrm{Drop}(\mathrm{FF}(\widetilde{H}))\big). \tag{14}$$

Each W_r is specific to an edge type, and each block has its own learned parameters. MHA is attention with four heads, based on Equation (5); FF expands the vector to $4d$ coordinates, applies a smooth nonlinear activation (GELU) and dropout, and projects it back to d. The attention modules also apply dropout to attention weights. All dropout is active during training only. A node without incident edges has $\mu_v = 0$; an empty graph bypasses the

blocks. The average in Equation (12) limits the scale of a message aggregate at nodes of high degree. Self attention adds access to the whole graph, while typed messages retain an explicit local connection to arc direction and role.

The trace encoder starts with $e_{h(a_i)} + e^{\text{pos}}_{\min(i-1,4095)}$, adding a learned position vector to each event's label vector. Positions matter because $\langle a, b, c\rangle$ and $\langle c, b, a\rangle$ have different alignment problems despite containing the same labels. A transformer with one layer and four attention heads processes these event vectors. It uses no causal mask: the complete trace, including its suffix, is already available. LARA is therefore an offline alignment method, not a next event predictor restricted to a running prefix. The position table has 4096 rows indexed from zero; the implementation clamps later positions to its final row. The reported traces are much shorter than this limit. An empty trace bypasses this transformer and yields an event matrix with zero rows.

Two cross attention operations then connect graph and trace. Let G_T be the graph's transition vectors and H_E the trace vectors. When $n > 0$ and $|T| > 0$, Equations (15) and (16) give the final event and transition vectors:

$$V = \mathrm{LN}\big(H_E + \mathrm{MHA}(H_E, G_T, G_T)\big), \tag{15}$$

$$U = \mathrm{LN}\big(G_T + \mathrm{MHA}(G_T, H_E, H_E)\big). \tag{16}$$

The two attention modules and their layer normalizations have separate parameters; both contexts are calculated from the vectors before cross attention. If either collection is empty, both cross attention operations and their normalizations are skipped, so $V = H_E$ and $U = G_T$. An event vector can reflect which transitions are relevant in this Petri net, and a transition can reflect which events occur in this trace. These numerical associations explain how the neural network can use the suffix c to prefer one b transition; they do not guarantee that it learns the appropriate association on every input.

Output. The encoders return contextual event vectors $V \in \mathbb{R}^{n\times d}$ and transition vectors $U \in \mathbb{R}^{|T|\times d}$, whose rows are v_i and u_j. These vectors summarize the supplied Petri net and complete trace for the scoring stage; they do not represent the decoder's changing marking.

4.3. Move Scores and Their Limits

Input. The scoring stage receives the contextual event vectors V, transition vectors U, and exact label compatibility matrix C.

Processing. Three output heads convert these vectors into preferences for the move vocabulary. Let $\widetilde{V} = \mathrm{Drop}(V)$ and $\widetilde{U} = \mathrm{Drop}(U)$ be the vectors after the scoring heads' dropout, with rows $\widetilde{v}_i, \widetilde{u}_j$; these equal v_i, u_j at inference. With learned parameters W, w_M, w_L, b_M, b_L, Equation (17) gives synchronous scores and Equation (18) gives model and log scores:

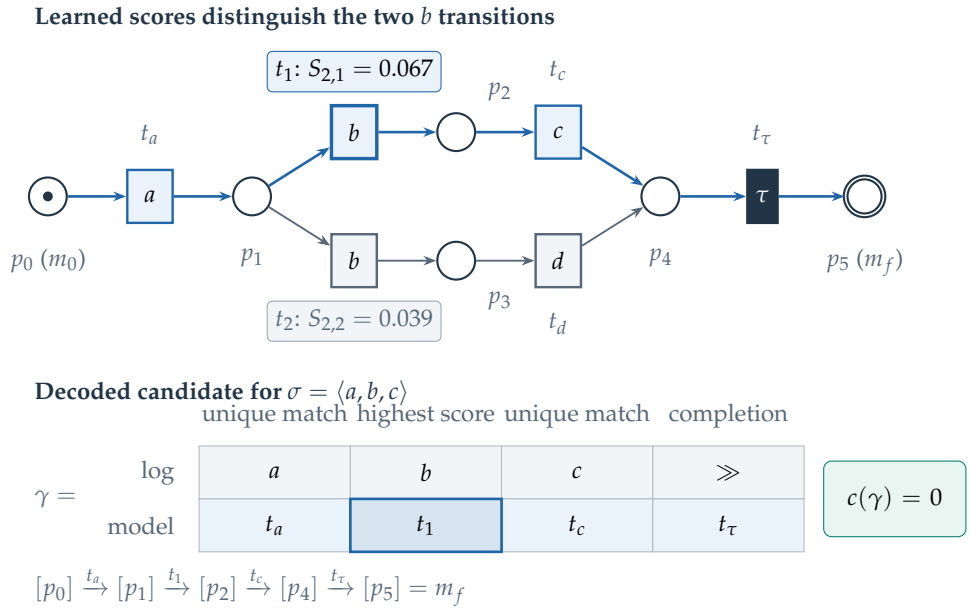
$$S_{ij} = \begin{cases} (W\widetilde{v}_i)^\top \widetilde{u}_j/\sqrt{d}, & C_{ij} = 1, \\ -10^9, & C_{ij} = 0, \end{cases} \tag{17}$$

$$M_j = w_M^\top \widetilde{u}_j + b_M, \qquad \ell_i = w_L^\top \widetilde{v}_i + b_L. \tag{18}$$

The matrix $S \in \mathbb{R}^{n\times|T|}$ ranks matching transitions for each event. The vector $M \in \mathbb{R}^{|T|}$ ranks model moves. The auxiliary vector $\ell \in \mathbb{R}^n$ is trained to indicate log moves but is not used by the default greedy decoder. The finite value -10^9 suppresses incompatible scores in the evaluated floating point computation, but does not remove them from the real-valued softmax denominator. Section 5.8 explains this distinction for Equation (26). The decoder explicitly restricts synchronous choices to enabled transitions with the original matching label, independently of score magnitudes.

A single forward pass calculates these arrays once per Petri net and trace pair. They depend on the complete input, including initial and final markings, but remain fixed as decoding proceeds. In particular, M_j has no event position index, and neither S nor M is recomputed after a mistaken commitment. This saves repeated neural evaluation but limits adaptation to the decoder's current state. Scores rank alternatives; move costs remain those in Table 1.

For the running example, Figure 4 connects the reference checkpoint's complete move scores to the drawn Petri net and decoded alignment. Once t_a has fired, both t_1 and t_2 are enabled and compatible with event b. The score of t_1 is slightly higher, so the decoder takes the branch that can subsequently explain c. The magnitude difference is a preference, not an estimate of a cost saving or a confidence interval. When a unique matching transition is enabled, the synchronous score is not needed at all.



Output	t_1	t_2	t_a	t_c	t_d	t_τ
$S_{1,:}$: event a	×	×	0.353	×	×	×
$S_{2,:}$: event b	0.067	0.039	×	×	×	×
$S_{3,:}$: event c	×	×	×	0.876	×	×
M	−1.357	−1.401	−1.030	−1.830	−1.837	2.519

$\ell = (-1.682, -1.618, -1.527)$ (unused by the decoder); × denotes the finite mask value -10^9.

Figure 4. Complete move scores for the drawn Petri net and trace, rounded to three decimals (reference epoch-49 checkpoint, dropout disabled). Columns follow sorted transition identifiers. The higher synchronous score selects t_1 for b, shown by the highlighted path and alignment cell. The decoder then explains c and completes through t_τ at cost zero. All synchronous, model, and auxiliary log scores are included.

At fixed depth and embedding width, graph self attention has quadratic dependence on the node count $|P| + |T|$, trace self attention on n, and cross attention on $n|T|$. Typed messages additionally traverse the encoded arcs. Avoiding a reachability graph therefore does not make the scorer linear in Petri net size. The parameter count is fixed, but inference cost still grows with the input.

Output. The decoder receives the fixed arrays S and M as rankings of synchronous and model alternatives. The log scores ℓ are unused by the default greedy decoder, but order the alternatives in the optional repair procedure below. None of these arrays replaces the move costs or certifies a complete alignment.

4.4. Candidate Decoding with the Current Marking

Input. The decoder receives SN, σ, the fixed score arrays S, M, and two finite non-negative integer search limits: a depth d_p for enabling the next event and a depth d_f for reaching the final marking after all events. The defaults are $d_p = 8$ and $d_f = 32$; the stress and real log experiments use $d_f = 64$.

Processing. Starting from m_0, the decoder turns preferences into a concrete execution by consulting the current marking at every step. Compatible transition labels alone do not enforce the model. Algorithm 1 specifies how immediate synchronization, bounded model searches, and log moves are combined.

Algorithm 1 Construct a candidate with bounded model move searches

Require: $SN, \sigma = \langle a_1, \ldots, a_n \rangle$, fixed scores S, M, depths d_p, d_f
Ensure: Proposed move sequence γ, awaiting independent replay

```
m ← m_0; γ ← ⟨ ⟩
for i = 1, ..., n do
    B ← {t ∈ T : t enabled at m, λ(t) = a_i}
    if B = ∅ then
        BFS from m, depth at most d_p, to a marking enabling label a_i
        if a path is found then
            Append its model moves to γ and fire them on m
            Recompute B at m
        end if
    end if
    if B ≠ ∅ then
        t_j ← arg max_{t_j ∈ B} S_ij                    ▷ unique match needs no ranking
        Append (a_i, t_j) to γ; fire t_j on m
    else
        Append (a_i, ≫) to γ                            ▷ marking unchanged
    end if
end for
BFS from m, depth at most d_f, to m_f
Append and fire the completion path if found (empty if already at m_f)
return γ for independent replay, including when completion failed
```

Each breadth-first search explores paths by the number of transition firings. Within an expansion, invisible transitions precede visible transitions, and transitions in the same group are ordered by decreasing M_j. Equal scores are resolved by ascending transition identifier for synchronous choices and descending identifier for model search. Each search starts a fresh visited marking set and stops at the first goal marking found. It does not minimize deviation cost: a short path containing a visible move may be considered before a longer path containing only zero-cost invisible moves. Finite depth and finite T ensure termination of each bounded search. The depth limits bound individual paths, without imposing a separate budget on explored markings; branching can still make search expensive.

If no bounded path enables the event, the decoder makes a log move. If a matching transition is already enabled, it synchronizes immediately. This policy never explicitly compares the cost of skipping that event with the downstream consequences of synchronizing it. It also never revises earlier choices. This is a structural limitation: even perfect ranking among matching transitions cannot select a log move while a matching transition or bounded enabling path is available. Better training alone cannot remove this restriction. Section 7.8 gives a concrete counterexample.

In unguided mode, all learned scores are replaced by zero, and the forward pass is skipped. The same firing rules, depth limits, invisible first policy, and identifier tie rules remain. The difference between guided and unguided modes therefore concerns ranking,

not relaxed legality. Identifier order is deterministic but has no business meaning, which motivates the renaming experiment.

We additionally evaluate an optional single-log repair procedure. It first constructs and replays the greedy candidate. Among event positions synchronized by that candidate, it tries at most eight alternatives, each forcing one position to be a log move and rerunning the same greedy construction for the complete trace. Guided repair orders these positions by decreasing ℓ_i, breaking ties by position; unguided repair uses increasing position. Positions already skipped by the base candidate need no trial because forcing the same log move reproduces its path. All trials reuse the original forward pass and its fixed scores. Only a complete, replay-accepted candidate of strictly lower cost replaces a legal incumbent; a legal trial may also replace an incomplete base candidate. When the procedure finishes with a legal incumbent, its cost cannot exceed that of a legal base candidate. A wall-time interruption can still prevent a result. The eight-trial limit bounds the number of alternative candidates, not the width of each marking search. This heuristic permits a log move even when synchronization is available, but does not enumerate combinations of forced log positions or guarantee an optimum. Section 7.9 evaluates it separately from Algorithm 1.

Output. The decoder returns a proposed move sequence $\widehat{\gamma}$ with concrete transition identities. It consumes the trace but may fail to reach m_f when the bounded completion search is unsuccessful. The sequence therefore goes to independent replay before it can be reported as a verified alignment.

4.5. Replay Verification and Exact Certification

Input. The verification stage receives the proposed sequence $\widehat{\gamma}$, the original Petri net and trace, the move costs, and the requested mode, FAST or CERTIFIED. Verification is needed even though decoding fires enabled transitions: bounded completion can fail, and independent checking also detects malformed candidates.

Processing. Replay reconstructs the marking sequence from m_0, validates each transition identity and synchronous label, checks that the log projection is exactly σ, and requires the final marking to equal m_f. Only then is the summed cost advertised as an upper bound. A failed proposal can be retained for diagnostics, but it is not a verified alignment result.

The assurance contract follows immediately from the alignment definition.

Proposition 1 (Verified candidate and cost equality)**.** *If replay accepts $\widehat{\gamma}$, then $\delta(\sigma, SN) \leq c(\widehat{\gamma})$. If an exact aligner establishes $\delta(\sigma, SN)$ for the same trace, marked Petri net, and move costs, and $c(\widehat{\gamma}) = \delta(\sigma, SN)$, then $\widehat{\gamma}$ is optimal, irrespective of whether its move sequence equals the exact aligner's witness.*

Proof. Replay acceptance establishes that $\widehat{\gamma}$ belongs to the feasible set in Equation (3). Its cost is therefore at least the minimum over that set. Equality means that it attains the minimum. No property of the neural scores is needed. □

In FAST mode, the system returns the candidate, replay status, and verified cost when available. In CERTIFIED mode, it additionally calls PM4Py's state equation A* aligner. The wrapper explicitly supplies all three cost functions from Table 1, including zero cost for invisible model moves, and compares replayed integer costs without rescaling. If an optimal exact result is available, it retains a legal candidate with the same cost; otherwise it returns the exact alignment as a replacement, with replay checks on that result. Replaying the exact witness establishes its legality; the claim that its cost is minimal relies on the exact backend. If exact search times out or fails, a legal candidate remains an upper bound and the wrapper leaves its certification flag unset. The mathematical optimality of a legal zero-cost candidate still follows from nonnegative costs, as explained in Section 2.3.

The exact backend starts its own search. It does not consume the candidate as an incumbent solution, prune states using its cost, or use learned scores in its heuristic. Consequently, a candidate that is already optimal still incurs exact work in certified mode. The wrapper separates latency to a usable candidate from latency to a certificate; it does not yet accelerate the certificate itself. These same steps can be applied to a candidate produced by a classical approximation.

Output. The result records the replay status, any verified candidate and cost, and any exact result or failure status. Table 4 maps these outcomes to the claims they support. A proposal, a verified upper bound, and a certified optimum remain distinct outputs.

Table 4. Result states reported by the wrapper. Its certification flag requires successful exact computation.

State	Evidence Obtained	Claim Available
Rejected proposal	Replay fails or no proposal is produced	No candidate upper bound.
Verified candidate	Replay succeeds	Executable explanation; $\delta \leq c(\widehat{\gamma})$.
Certified candidate	Replay succeeds; exact optimum has equal cost	Candidate attains the minimum cost.
Exact replacement	Exact optimum available; candidate rejected or has larger cost	Returned exact alignment is optimal.
Exact timeout	No completed exact result	Only the candidate's existing replay evidence, if there is any.

5. Data Generation and Training

Training a reusable scorer requires examples that separate process behavior from the way a Petri net happens to represent it. It also requires trustworthy targets: synthetic corruption tells us how an observation was produced, but not necessarily its cheapest explanation. This section describes the complete path from a generated behavior family to input tensors, exact alignment targets, and parameter updates. All generation and loss details needed to understand the reference model are integrated here.

5.1. Why Generate Behavior Families?

An isolated pair of a random Petri net and a noisy trace would teach the neural network about one representation at a time. It would give much less control over whether a performance change comes from different behavior or merely different routing. A *behavior family*, defined in Equation (19), groups two Petri nets with the same complete visible language and two shared observations:

$$\mathcal{B} = (SN_1, SN_2, \sigma_1, \sigma_2), \qquad \mathcal{L}(SN_1) = \mathcal{L}(SN_2). \tag{19}$$

Each family expands into four supervised inputs, (SN_r, σ_k) for $r, k \in \{1, 2\}$. Both representations see exactly the same observations. Figure 5 shows this expansion and the checks that precede writing the rows.

The entire family belongs to one split before expansion. This avoids placing two views of the same generated family in training and test. It does not make all visible languages, motifs, or Petri net topologies disjoint across splits: independently generated families can repeat the controlled structures. The test split measures new family instances within this generator, not transfer to wholly unseen motif classes.

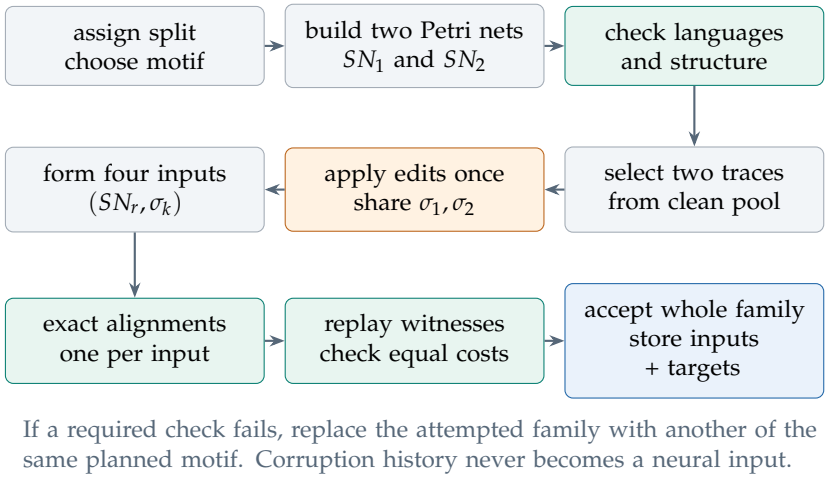


Figure 5. Data generation as a sequence of explicit checks. One family supplies two behaviorally equivalent representations and two observations shared across them. Each of the four rows receives its own exact witness. A family is written only after all rows pass, keeping both the split assignment and the planned motif balance intact. Green denotes verification and exact supervision, amber denotes trace corruption, blue denotes accepted training examples, and gray denotes data preparation.

Before generating Petri nets, a deterministic quota assigns the requested number of families to four equally weighted motifs. Minimum family counts are reserved per motif, remaining slots are distributed by the weights, and the plan is shuffled with a split specific seed. If a generation attempt fails, another family index is tried for the same planned motif. This prevents rejection from silently shifting the corpus toward easier model classes. The global reference seed is 13; separate seeds derived from the split and family index control structure, representation, and corruption.

5.2. Four Motifs and Their Representation Pairs

The motifs target distinct reasons why a label sequence may be difficult to align. Table 5 specifies their visible core languages. The three controlled motifs append two fresh activities shared by both representations. If their labels are u, v, the complete language is $\{wuv : w \in \mathcal{L}_{\text{core}}\}$. Fresh labels prevent the suffix from introducing another core ambiguity. The implementation selects the first unused activity names: the duplicate/silent and M motifs append D, E, while the concurrency motif appends C, D.

Table 5. Motif design. The listed languages omit the shared suffix for the three controlled motifs. "Canonical" identifies the generator's fixed block compiler.

Motif	Core Behavior	Representation Pair
Ordinary tree	Sampled tree language	Canonical block Petri net and isomorphic identifier renaming.
Duplicate/silent	AB, AC	Two A transitions that select a branch; one A followed by invisible routing.
Concurrency	AB, BA	True parallel execution; explicit choice between the two sequences.
M pattern	B, AC, CA	XOR of B and parallel A, C; non-free-choice M Petri net.

5.2.1. Ordinary Trees and Identifier Renaming

This pair tests whether a candidate depends on incidental naming. A random process tree has 4 to 12 visible leaf occurrences and depth at most six. Sequence, XOR, and AND have selection probabilities 0.4, 0.3, and 0.3. The alphabet size is approximately 70% of the visible leaf count, so different leaves can carry the same label. The compiler maps leaves to visible transitions, sequences to linked fragments, XOR branches to shared entry and exit places, and AND to invisible split/join routing.

The second Petri net is an isomorphic copy with renamed place and transition identifiers. Labels, arcs, markings, and weights remain unchanged under the corresponding bijections. Its firing possibilities and optimal costs are therefore unchanged. Sorting the new identifiers can, however, change how the unguided decoder breaks a tie. This pair distinguishes an improvement in handling identifiers from an improvement on a new process behavior.

5.2.2. Duplicate Prefix and Silent Routing

This pair isolates a choice that may have to be made before its distinguishing event arrives. In the duplicate prefix Petri net, two enabled transitions are both labeled A. One leads to B and the other to C. Firing A therefore commits to a branch. In the silent routing Petri net, there is one shared A transition followed by two invisible alternatives that enable B or C. Both Petri nets generate AB or AC before the common suffix, but the visible ambiguity occurs at a different point. Figure 6 shows both cores.

For an observation beginning with A, C, selecting the appropriate duplicate A requires looking beyond the current event. The silent routing representation can postpone its branch selection until trying to enable C. If guidance helps mainly on the first Petri net, that is evidence about early ambiguity rather than a universal advantage over the same decoder.

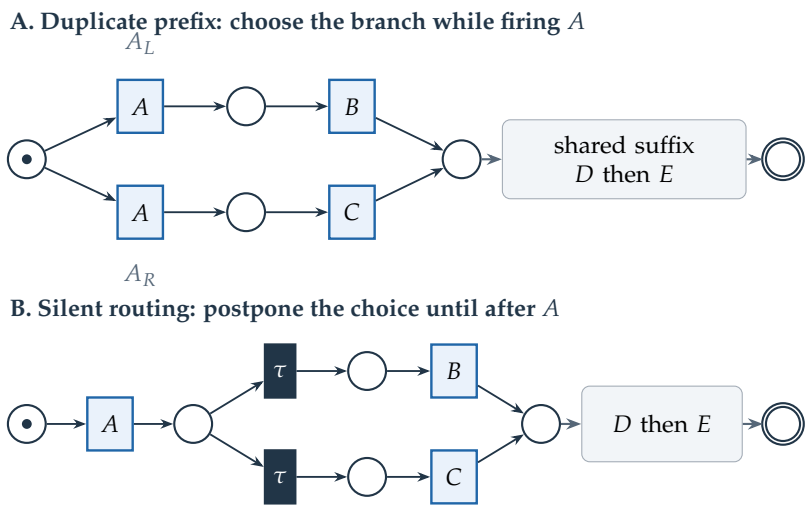


Figure 6. Equal visible languages with different decision points. (**A**): choosing a concrete A transition commits to B or C. (**B**): a shared A precedes invisible branch routing. The appended suffix D, E is identical in both Petri nets. The visible language is $\{ABDE, ACDE\}$, although transition identities and silent moves differ.

5.2.3. Parallelism and Explicit Interleaving

This pair tests a change in the representation of concurrency. One Petri net uses an invisible split to enable A and B in parallel, followed by a join. The other uses an XOR between the sequences AB and BA. Their complete visible traces coincide, but their causal structures differ: simultaneous availability in one representation becomes a choice between sequential executions in the other. The equivalence is therefore trace language equivalence,

not equivalence of partial order behavior. This distinction matters when interpreting representation agreement as evidence of transfer.

5.2.4. Block Structure and the Non-Free-Choice M Pattern

This pair tests a conflict between transitions with different input places. The block representation is $\mathrm{XOR}(B, \mathrm{AND}(A, C))$. In the M representation, an invisible split places one token in each of two places. A consumes the first, C consumes the second, and B consumes both. Therefore B conflicts with both A and C, while A and C can occur in either order. The presets of conflicting transitions are unequal, giving the non-free-choice structure. Appropriate output places and a join complete the A, C branch. Figure 7 shows the token level construction together with the parallelism pair.

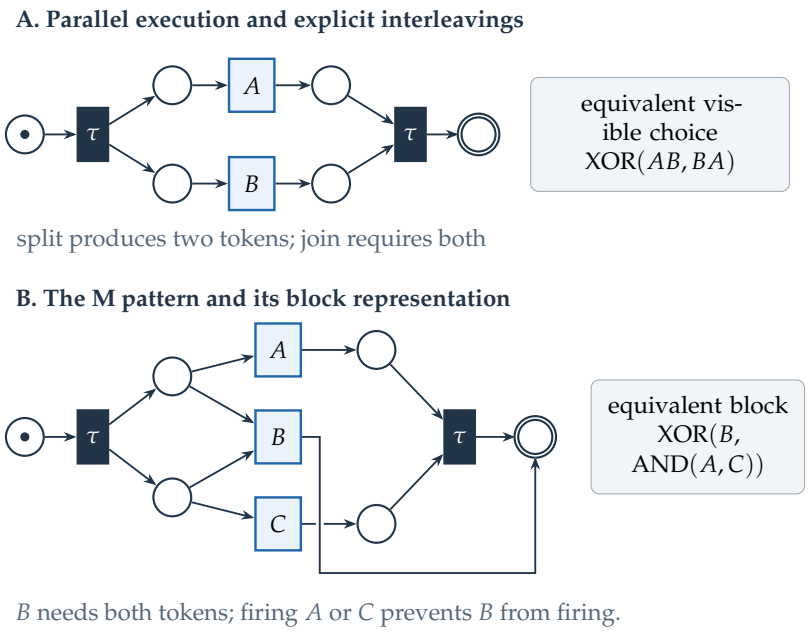


Figure 7. Two additional changes of representation. The parallel core and the explicit interleaving expression both admit AB and BA. In the M core, B consumes both central tokens, while A and C consume one each and later join; its visible language is $\{B, AC, CA\}$. The right hand expressions denote the corresponding block structures. Shared fresh suffixes are omitted.

5.3. Checking Behavioral Equivalence and Usable Structure

The representation checks protect the interpretation of a family. If the two Petri nets admitted different visible traces, a cost difference could be semantically correct rather than a representation effect. The generator enumerates complete visible traces using search records (m, w, k): a marking, the visible word so far, and the number of firings. Distinct words reaching the same marking must remain distinct records. Default limits are 5000 explored records, 10,000 complete traces, visible length 20, and at most $4 \cdot 20 + |T|$ firings per path. Reaching a limit while further exploration is needed makes the enumeration incomplete. For the generated representations, no transition is enabled at m_f, so stopping there cannot omit longer complete executions. This stopping rule would need reconsideration for arbitrary nets that can leave and revisit m_f. A comparison is labeled *exact* only if both enumerations finish without truncation and their languages agree. Matching sets found within bounds are not proof of full language equality.

Generation rejects mismatching enumerated languages or an empty enumerated complete language. A separate bounded marking exploration must witness the final marking and at least one firing of every transition. A transition is dead if it can never fire from a reachable marking; failure to observe a firing in a truncated search does not prove deadness.

These are conservative acceptance filters, and even witnessing the final marking and every transition does not establish that every reachable marking can complete, as workflow soundness requires. The default configuration additionally rejects bounded equivalence checks in the training split. All accepted rows in the reference training, validation, and test corpus have exact language certificates within the configured bounds.

The cost comparison used later has a formal rationale. Let $d_{\mathrm{ID}}(\sigma, w)$ be insertion/deletion distance, with unit cost for each inserted or deleted visible activity. Equation (20) relates this distance to alignment cost under Table 1:

$$\delta(\sigma, SN) = \min_{w \in \mathcal{L}(SN)} d_{\mathrm{ID}}(\sigma, w). \tag{20}$$

Every legal alignment selects a complete visible model word and a corresponding sequence of insertions, deletions, and matches. Conversely, such an edit sequence can be combined with a firing sequence for that word, inserting its zero-cost invisible moves. Thus language equivalent Petri nets have equal optimal visible deviation costs, even though their optimal transition sequences may differ. This equality would require reexamination if costs depended on transition identity or charged invisible routing.

5.4. Producing Observations by Trace Corruption

Corruption creates deviations without changing the reference process. For each accepted family, the enumerated language is sorted. If it contains more than 16 words, the family random generator selects 16 indices and restores them to language order. Otherwise the whole language is retained. The first two entries of this pool supply clean traces, wrapping around when only one entry is available. These choices and their corruption are deterministic given the configured seeds.

Each observation is kept clean with a probability of 0.2. Otherwise, it receives one, two, or three edits with probabilities of 0.5, 0.3, and 0.2. Table 6 lists the operation weights. Operations are drawn again after each edit and act on the current trace. We write X for the outside label whose literal implementation string is `__OOD__`; it is absent from the family alphabet. Inapplicable operations, such as a swap on a one-event trace, fall back to insertion of this same X. Traces are capped at a length of 128. The realized frequencies can therefore differ from the nominal weights, especially for short observations.

Table 6. Trace corruption operations. Examples start from $ABDE$ and illustrate individual edits; multi edit traces apply several operations in sequence. Weights are conditional on drawing an edit operation.

Operation	Weight	Effect	Example
Delete	0.15	Remove an event.	ADE
Insert	0.15	Insert a label from the family alphabet.	$ABBDE$
Substitute	0.15	Replace an event label.	$ACDE$
Swap	0.15	Exchange neighboring events.	$BADE$
Repeat	0.15	Duplicate one event.	$ABDEE$
Prefix truncation	0.10	Remove a beginning segment.	DE
Suffix truncation	0.10	Remove an ending segment.	AB
Outside insertion	0.05	Insert the outside label X.	$ABXDE$

The same corrupted observation is then paired with both Petri nets. This is essential to the comparison: applying different noise to the two representations would confound behavior with observation quality. The corruption history is metadata, not a neural input and not the alignment target. An edit can produce another fitting trace; several edits can cancel or admit an explanation cheaper than undoing them. For example, substituting C for

B in $ABDE$ produces $ACDE$, which still fits the duplicate/silent family. Exact alignment is therefore performed after corruption.

5.5. Exact Supervision and Family Level Acceptance

Each expanded row is sent to the PM4Py state equation A* teacher with transition-aware output. The reference generation imposes no per trace timeout. Results are cached within a split using a signature of the Petri net, the observed sequence, and the cost model version. Caching reduces repeated teacher work without changing the targets. A different representation still requires its own concrete transition witness.

Every exact witness is replayed independently. The initializer requires an optimal result, checks all moves and both projections, and verifies that the paired Petri nets give equal optimal costs for the same observation when their language equivalence was exact. Rows are buffered until the entire family passes; a failure replaces the family rather than writing only its successful rows. For the reference corpus, the manifest records 25 generation-time `ValueError` attempts that were replaced. It records no exact alignment, replay, or paired cost rejection. These are observations about this run, not a guarantee that future configurations cannot fail those gates.

From one optimal witness γ^*, the code derives three move targets. The synchronous target y_i is the zero-based index of the transition used for event i, or -1 when there is no active synchronous target. The binary target q_i is one exactly when the event is a log move. The binary target b_j is one when transition t_j occurs at least once as a model move anywhere in γ^*. Invisible model moves are included in b. This last target does not record the position or multiplicity of model moves, matching the deliberately coarse vector M.

Teacher identity is not unique when several alignments are optimal. The generator uses one particular witness. For the duplicate prefix motif, it further masks all synchronous targets when the observation contains both distinguishing labels B, C or neither. This is the implemented heuristic for avoiding ambiguous identity supervision, not a general theorem characterizing all unique optima. Log, model, and cost targets remain active, and the witness itself remains in the stored row.

5.6. A Complete Worked Training Row

A concrete row helps separate input information from the answer supplied during training. Consider stored sample `train-000048`. Starting from $ABDE$, the generator swaps the final events and substitutes C for B, yielding $\sigma = \langle A, C, E, D \rangle$. Its duplicate prefix Petri net is the upper representation of Figure 6, followed by the suffix. Write A_L, A_R for its two A transitions, with A_L leading to B and A_R to C.

The place order is $(p_D, p_E, p_0, p_f, p_L, p_R)$ and the transition order is (D, E, A_L, A_R, B, C). The core ends at p_f, followed by D to p_D and E to p_E; the final marking is $[p_E]$. Equation (21) gives the numerical feature matrices in these orders:

$$X_P = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 2 & 2 \\ 0 & 0 & 2 & 1 & 3 \\ 0 & 0 & 1 & 1 & 2 \\ 0 & 0 & 1 & 1 & 2 \end{bmatrix}, \qquad X_T = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}. \tag{21}$$

Every transition has the same local feature row, yet the graph connections and labels differ. This demonstrates why local degrees alone cannot resolve the duplicate choice.

The graph uses place indices $0, \ldots, 5$ and transition node indices $6, \ldots, 11$. Table 7 specifies all edges compactly. For each triple (p, t, q), insert edges $(p, t), (t, p), (t, q), (q, t)$

with types $(0, 2, 1, 3)$. Thus E has 24 columns and r repeats this four type sequence six times. This explicitly recovers the arc information that distinguishes the otherwise equal rows of X_T.

Table 7. Full edge specification for the worked input. The readable transition name identifies a row; the three integers are graph node indices in Equation (21).

Transition	Input Place p	Transition Node t	Output Place q
D	3	6	0
E	0	7	1
A_L	2	8	4
A_R	2	9	5
B	4	10	3
C	5	11	3

Equation (22) gives the label IDs, and Equation (23) gives the exact compatibility matrix:

$$z_T = (2587, 4475, 8075, 8075, 6408, 5878),$$
$$z_\sigma = (8075, 5878, 4475, 2587), \tag{22}$$
$$C = \begin{bmatrix} 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}. \tag{23}$$

The first event can match either A transition. The third event is compatible with E, even though E is not enabled initially. The separate position indices $(0, 1, 2, 3)$ are added by the trace encoder. Equations (21)–(23) and Table 7 specify every component of X; no clean trace or exact cost is among these inputs.

Equation (24) gives the stored teacher witness:

$$\gamma^* = \langle (A, A_R), (C, C), (E, \gg), (D, D), (\gg, E) \rangle, \qquad \delta = 2. \tag{24}$$

It takes the branch consistent with C, skips the early E, synchronizes D, and finishes with a model move E. Another optimum could insert D before synchronizing E and then skip the final observed D. The chosen witness gives the targets in Equation (25):

$$y = (3, 5, -1, 0), \qquad q = (0, 0, 1, 0), \qquad b = (0, 1, 0, 0, 0, 0). \tag{25}$$

These arrays and δ are the supervision. In particular, target $y_1 = 3$ names A_R in the stored zero-based transition order; it is not activity ID 3. This distinction is important when implementing transition-aware machine learning.

5.7. Training Time Relabeling and Corpus Composition

Label relabeling discourages the scorer from solving controlled examples by memorizing a small set of synthetic IDs. On every training visit, probability 0.5 triggers a consistent random remapping of distinct nonzero IDs in the Petri net and trace to distinct values in $\{1, \ldots, 8191\}$. The graph, exact compatibility mask, and targets remain unchanged. For example, mapping the IDs for A, B, C, D, E to $101, 205, 309, 413, 517$ would give $z'_T = (413, 517, 101, 101, 205, 309)$ and $z'_\sigma = (101, 309, 517, 413)$. This example illustrates an augmentation, not a recorded random draw.

ID zero stays reserved for invisible transitions, and existing hash collisions remain collisions. Because embeddings differ across IDs, this augmentation encourages robustness

but does not prove invariance under every label renaming. Validation and test use their original hashed IDs, and dropout is disabled there. The reference split sizes appear in Table 8. The test traces average 3.6 events and their exact costs range from zero to seven, so this is a small controlled corpus rather than a comprehensive sample of industrial process complexity.

Table 8. Reference corpus with seed 13. A family contributes four rows and remains inside one split. All four motifs are balanced within each split.

Split	Families	Petri Net and Trace Rows	Families per Motif
Training	512	2048	128
Validation	128	512	32
Test	128	512	32

5.8. Training the Move Scorer

The loss expresses which aspects of an exact witness the neural network should imitate. It does not differentiate through Petri net execution or prove the predicted sequence correct. The training objective assumes $|T| > 0$, as in the generated families, while allowing $n = 0$. Let $I = \{i : y_i \geq 0\}$ and $J_i = \{j : C_{ij} = 1\}$. When y_i is used as a matrix column below, its stored zero-based index is converted to the corresponding column. Each active teacher target belongs to J_i, so $J_i \neq \emptyset$ for $i \in I$. The implementation applies log-softmax to the full score row after the finite masking in Equation (17); Equation (26) states the corresponding probabilities over all transition columns:

$$p_{ij} = \frac{e^{S_{ij}}}{\sum_{k=1}^{|T|} e^{S_{ik}}}, \qquad j \in \{1, \ldots, |T|\}, \quad i \in I. \tag{26}$$

Cross entropy penalizes putting little mass on the teacher transition. Smoothing allocates target mass only to compatible transitions. With $\epsilon_s = 0.05$, Equation (27) defines the smoothed loss:

$$\mathcal{L}_{\text{sync}} = -\frac{1}{|I|} \sum_{i \in I} \left[(1 - \epsilon_s) \log p_{i,y_i} + \frac{\epsilon_s}{|J_i|} \sum_{j \in J_i} \log p_{ij} \right]. \tag{27}$$

Equation (27) assigns the teacher mass $1 - \epsilon_s + \epsilon_s / |J_i|$ and every other compatible transition mass $\epsilon_s / |J_i|$. Incompatible transitions receive no target mass. In the worked row, the two A transitions receive 0.025 and 0.975. The implementation evaluates `log_softmax` directly using a stable log-sum-exp calculation; it does not compute the raw exponentials in Equation (26). With a finite mask, incompatible columns still have positive probability in exact real arithmetic. When their stabilized exponential contributions underflow to zero, the computation is numerically equivalent to normalization over J_i; a single compatible transition then contributes zero synchronous loss. This is not an identity for arbitrary scores: a compatible logit below -10^9 could let masked columns dominate the denominator. If I is empty, the loss is zero. Events with $J_i = \emptyset$, including X, have no active synchronous target and are excluded before normalization. The decoder separately checks original labels and enabledness, so finite masking is never the legality guarantee.

For binary decisions, define the binary cross entropy $\text{BCE}(z, u) = \log(1 + e^z) - uz$, evaluated directly from logit z with a numerically stable implementation. It compares the sigmoid probability with target u. A positive target encourages a high score and a

negative target a low score. Binary smoothing maps target zero to 0.03 and target one to 0.97. Equation (28) defines the other move losses:

$$\mathcal{L}_{\text{log}} = \frac{1}{n}\sum_i \text{BCE}(\ell_i, 0.94q_i + 0.03), \qquad \mathcal{L}_{\text{model}} = \frac{1}{|T|}\sum_j \text{BCE}(M_j, 0.94b_j + 0.03). \tag{28}$$

An empty output contributes zero. This objective imitates one exact witness; it does not optimize the cost of the decoded candidate. Synchronous loss trains rankings among compatible transitions. Model loss trains whether a transition appears anywhere as a model move, without its position, multiplicity, or current marking. Log loss trains ℓ_i, which the default greedy decoder ignores, although its gradients affect shared encoders. The optional repair procedure uses this head to order forced-log trials. To isolate the effect of log supervision on default decoding, Section 7.9 compares two matched training seeds with its loss coefficient set to one or zero; the remaining losses and architecture are unchanged. Greedy decoding still gives synchronization and enabling paths priority over log moves, so reducing the training losses need not reduce candidate cost. Neither the ablation nor repair constitutes direct optimization of decoded cost.

5.9. Auxiliary Cost and Region Objectives

The reference architecture also contains auxiliary region and cost heads. Their role must be explained because they contribute to the training objective, although they do not produce local executable alignments or participate in certification. A learned router applies dropout followed by separate affine maps for transitions and events, then a row-wise softmax over $R = 6$ latent regions. The resulting assignments are denoted Q^T_{jr} and Q^E_{ir}. "Region" means an internal grouping in representation space; it is not a verified subnet decomposition.

Equation (29) defines normalized weighted means for each region:

$$\bar{u}_r = \frac{\sum_j Q^T_{jr} u_j}{\max(10^{-6}, \sum_j Q^T_{jr})}, \qquad \bar{v}_r = \frac{\sum_i Q^E_{ir} v_i}{\max(10^{-6}, \sum_i Q^E_{ir})}. \tag{29}$$

A missing event collection gives $\bar{v}_r = 0$. A feedforward neural network transforms the concatenated summaries through affine maps of widths $2d \to d \to d$, each followed by GELU and training-only dropout, to obtain h_r. Its weights are shared across regions. Two heads produce $\widehat{l}_r = \text{softplus}(w_l^\top h_r + b_l)$ and $\widehat{u}_r = \widehat{l}_r + \text{softplus}(w_u^\top h_r + b_u)$, where $\text{softplus}(x) = \log(1 + e^x)$. This enforces nonnegative ordered endpoints. It does not enforce that the true alignment cost lies between them. Equation (30) supervises only the sum of the larger predicted endpoints:

$$\widehat{c} = \sum_r \widehat{u}_r, \qquad \mathcal{L}_{\text{cost}} = H_1(\widehat{c} - \delta), \qquad H_1(e) = \begin{cases} e^2/2, & |e| < 1, \\ |e| - 1/2, & |e| \geq 1. \end{cases} \tag{30}$$

The smooth-L1 penalty is quadratic near the target and grows linearly for larger errors. The identifiers `local_lower_bounds` and `local_upper_bounds` are retained for compatibility with the reference checkpoint and interface. In mathematical terms they are only ordered cost predictions. Neither $\sum_r \widehat{l}_r \leq \delta$ nor $\delta \leq \sum_r \widehat{u}_r$ is guaranteed, and the interval has no measured coverage. The smaller endpoint has no separate target, although it receives gradients through $\widehat{u}_r$. No endpoint is used to prune search, accept a candidate, or certify optimality. The valid upper bound is the replayed cost $c(\widehat{\gamma})$; exact search supplies its own justified lower bounds. The architecture also exposes a vector of "sketch" logits and a nonnegative "uncertainty" scalar per region. These are reserved auxiliary outputs: no

target gives the sketch entries an alignment meaning, and no calibration validates the uncertainty scalar. Neither has a direct supervised loss or enters the decoder.

Three regularizers influence the latent grouping. Let $U_p = \{j : (p, t_j) \in F \text{ or } (t_j, p) \in F\}$ be the set of distinct transition indices incident to place p, ignoring arc direction. Equation (31) averages the boundary penalty over unordered pairs of distinct indices:

$$\mathcal{L}_{\text{boundary}} = \frac{\sum_{p \in P} \sum_{\substack{j,k \in U_p \\ j<k}} (1 - \sum_r Q^T_{jr} Q^T_{kr})}{\sum_{p \in P} \binom{|U_p|}{2}}, \tag{31}$$

set to zero when the denominator is zero. It favors assigning transitions sharing a place to the same region. A pair sharing more than one place is counted for each shared place. Define the mean load $a_r = (\sum_j Q^T_{jr} + \sum_i Q^E_{ir})/(|T| + n)$ and the numerically stabilized entropy $H_\eta(q) = -\sum_r q_r \log \max(\eta, q_r)$ with $\eta = 10^{-8}$. Equations (32) and (33) define the balance and entropy penalties:

$$\mathcal{L}_{\text{balance}} = \frac{1}{R} \sum_r (a_r - 1/R)^2, \tag{32}$$

$$\mathcal{L}_{\text{entropy}} = \begin{cases} \frac{1}{2}\left(\frac{1}{|T|} \sum_j H_\eta(Q^T_j) + \frac{1}{n} \sum_i H_\eta(Q^E_i) \right), & n > 0, \\ \frac{1}{|T|} \sum_j H_\eta(Q^T_j), & n = 0. \end{cases} \tag{33}$$

For an empty trace, the load also uses transitions only. For a nonempty trace, balance weights every event and transition equally, whereas entropy averages the two collections separately and gives each collection half the weight. Minimizing the positive entropy term encourages concentrated assignments, while balance discourages using a single region for everything. These competing preferences shape representations, not verified process fragments. No ablation in this study isolates the benefit of these auxiliary terms.

5.10. Optimization and Checkpoint Selection

Equation (34) combines the losses with the reference run weights:

$$\begin{aligned} \mathcal{L} = {} & \mathcal{L}_{\text{sync}} + \mathcal{L}_{\text{log}} + \mathcal{L}_{\text{model}} + 0.03\mathcal{L}_{\text{cost}} \\ & + 0.01\mathcal{L}_{\text{boundary}} + 0.01\mathcal{L}_{\text{balance}} + 0.001\mathcal{L}_{\text{entropy}}. \end{aligned} \tag{34}$$

Each component is first reduced within one input pair; the resulting losses are averaged over 16 pairs per optimizer step. The implementation processes variable size pairs separately and accumulates gradients. AdamW [12] updates the parameters with initial step size 5×10^{-4} and weight decay 3×10^{-3}. Gradient norms are clipped at one. Dropout probability is 0.30.

Validation loss controls step size reduction by factor 0.5 after two plateau epochs, with a minimum step size of 10^{-5}. Early stopping has patience six and requires an improvement of at least 5×10^{-4}. The reference run uses training seed 13, executes 50 epochs, and selects epoch 49. Runs for seeds 17 and 29 use the same corpus, architecture, optimizer settings, validation selection, and maximum of 50 epochs. Each seed also has a matched run without the log loss. Each run selects its own checkpoint using its corresponding validation objective; test costs are never used for selection. The runs for seeds 17 and 29 use a different CPU and thread configuration, documented in Section 7.9. The mean epoch duration of the reference run is approximately 41 s on the reported CPU setup. Figure 8 shows the full recorded loss curves. Lower validation than training loss is plausible because validation disables dropout and random label remapping; the two losses are evaluated under different perturbation conditions.

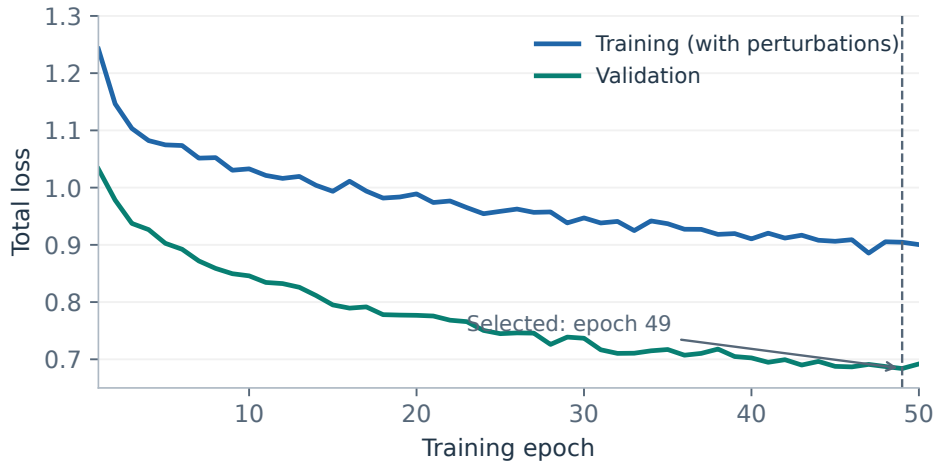


Figure 8. Recorded total training and validation losses. The dashed line identifies the checkpoint selected at epoch 49. These are the weighted objectives in Equation (34), not alignment costs or certificate rates. Training includes dropout and random label remapping; validation does not.

Replay legality and exact optimality are evaluated outside this differentiable objective. The neural network learns to imitate aspects of exact solutions; the semantic checks remain explicit computations at use time. This is the reason a poor prediction may reduce candidate quality without invalidating the meaning of a successful replay or an exact certificate.

6. Implementation and Interactive Use

An alignment method becomes useful to an analyst only when its explanations and evidence can be inspected. This section describes how the mathematical stages are implemented, how the workbench exposes their different outcomes, and how experiments can be reproduced. The interface treats a candidate and its later exact comparison as separate results so that a fast first explanation is never silently presented as an optimum.

6.1. Software Components and Interfaces

The application is publicly available at https://lara-pm.app (accessed on 29 September 2026), and the implementation is available at https://github.com/fit-alessandro-berti/lara-align (accessed on 29 September 2026). The training data are available at https://www.alessandroberti.it/data_lara_latest.tar.gz (accessed on 29 September 2026), and the trained checkpoint is available at https://www.alessandroberti.it/checkpoint_lara_latest.tar.gz (accessed on 29 September 2026). It uses PyTorch for neural computation and PM4Py for Petri net and event log objects, process discovery, and exact alignment [28]. The package provides Python and command line interfaces; the workbench uses Streamlit. Table 9 maps the conceptual stages to their implementation modules. This separation allows candidate generation experiments to reuse the same independent verifier.

An alignment move stores the observed label, transition name, and transition label where applicable. This avoids ambiguity when several transitions share a label. Proposal results retain the candidate, replay report, timings for feature construction/inference/decoding/verification, and optional score diagnostics. Certification results retain the exact witness, exact cost, elapsed time, and status. The candidate is preserved when a better exact alignment becomes available, allowing direct inspection of the gap.

The ordinary alignment entry point supports fast and certified modes. Separate proposal and certification calls support progressive presentation in the workbench. If proposal construction raises an exception, it returns an explicit failure; the workbench can still request exact computation through its separate certification operation. This distinction matters because the convenience alignment call does not automatically recover every

possible software failure. The "anytime" mode in the Python enumeration uses the exact comparison path; it is not a learned branch and bound algorithm with a stream of improving lower bounds.

Table 9. Implementation responsibilities in the `lara_align` package. A module's output is not given stronger guarantees than the checks it performs.

Module	Responsibility
`features.py`	Builds typed graph arrays, label IDs, and exact compatibility.
`model.py`	Encodes inputs and returns learned move scores and auxiliary outputs.
`decode.py`	Maintains markings, chooses synchronous transitions, and searches for model move paths.
`verify.py`	Replays complete move sequences, checks projections, and recomputes costs.
`exact.py`	Invokes PM4Py's exact backend and retains concrete transition identities.
`certifier.py`	Exposes proposal and exact comparison operations with explicit statuses.
`families.py`, `data.py`	Generate behavior families and store inputs, witnesses, and metadata.
`training.py`	Extracts supervision and computes the full training objective.

For decoding, the runtime precomputes each transition's preset, postset, label group, and score lookup. Local marking search then operates on sparse token dictionaries and a visited marking set. These implementation choices reduce overhead without changing the firing semantics. Each proposal constructs its input features and runtime; the reported timing does not assume a globally cached neural representation for a target Petri net. The checkpoint is loaded once before processing a sequence of traces.

6.2. Preparing a Log and Reference Model

The Setup page accepts XES event logs, including compressed files, and Petri nets in PNML format. An analyst may provide an independent reference model or discover a model from the uploaded log using Inductive Miner. The latter option is labeled explicitly: checking a model discovered from the same data is a different analysis from checking compliance with an independently specified process. The discovery noise threshold controls whether infrequent behavior is filtered.

The page summarizes the log and Petri net, lists activity labels and duplicate label groups, and groups cases into variants. Grouping is valid for this control flow setting because cases with the same activity sequence have the same alignment problem against a fixed Petri net and cost policy. It reduces repeated computation while retaining a mapping back to case identifiers. The analyst chooses which variants to process, and the workbench reports the corresponding case coverage. Unprocessed variants are not counted as successful alignments.

6.3. Progressive Results and Trace Inspection

The Live Run page offers fast candidates, progressive certification, and an exact baseline. Figure 9 shows its progress indicators and variant results. In candidate modes it first produces and independently verifies the proposal for a variant. The result can be displayed immediately, while exact work is a subsequent stage when selected. Statuses distinguish a ready candidate, an illegal candidate, exact computation, certification, repair, timeout, and failure. A timeout retains the evidence already obtained; it does not fill in an unknown optimum.

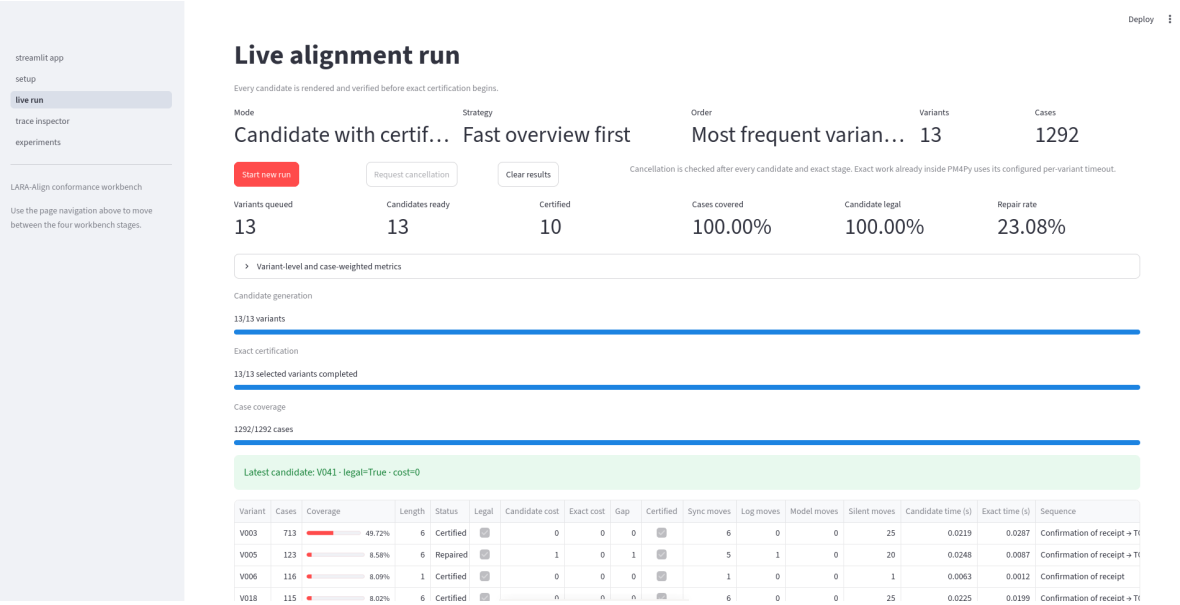


Figure 9. The Live Run page of the LARA workbench. Progress indicators distinguish candidate generation, exact certification, and case coverage. The variant table keeps candidate legality, candidate and exact costs, gaps, certification status, and timings visible as separate fields. The displayed values describe the illustrated session. Only part of the interface and variant table is shown.

The Trace Inspector aligns candidate and exact explanations to common observed event positions, as shown in Figure 10. It exposes concrete transition identities, which is especially useful for duplicate labels, and allows the model projection to be replayed through precomputed marking snapshots. An analyst can therefore see both where an event was skipped and which enabled transition the decoder chose. Neural score diagnostics are supplementary preferences. They are not labeled as causal explanations or calibrated certainty about correctness.

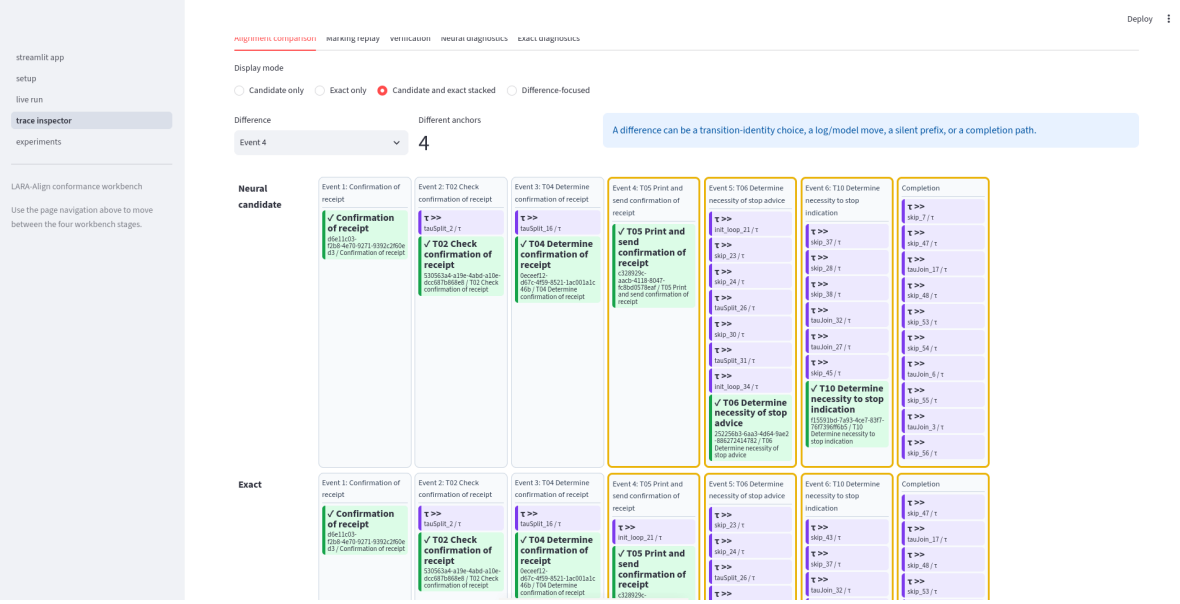


Figure 10. The Trace Inspector with candidate and exact alignments stacked at common event positions. Green cards show synchronous moves, purple cards show invisible model moves, and outlined columns mark differences between the explanations. Concrete transition identities and the separate completion column expose choices that a comparison of activity labels alone would hide. The remaining exact moves are available by scrolling or export.

Results can be exported as variant level and case level CSV, detailed JSON, execution logs, and per alignment move tables. Candidate cost, exact cost, gap, legality, and certification remain separate fields. This makes subsequent analysis auditable: a CSV consumer can select only certified optima or deliberately study the latency and quality of verified candidates. The optional experiment dashboard reads stored metric files for training curve and benchmark inspection.

6.4. Reproducibility and Experiment Artifacts

The journal source includes the reference synthetic corpus, generation configuration, per epoch metrics, and numerical benchmark records. The pretrained checkpoint is distributed separately through the download linked in the Data Availability Statement; its recorded checksum identifies the model used here. The script `init_data.py` generates the behavior families; `train_model.py` trains and saves checkpoints. Evaluation scripts cover the fixed synthetic test split, scaling inputs, PM4Py approximations, and real logs. The included result data and figure scripts allow the plots to be rebuilt without rerunning training.

Reproducing candidate quality figures requires the recorded checkpoint and matching inputs. Reproducing timings additionally requires the hardware, software, thread settings, and measurement boundaries in Section 7.1. Stored result files contain the recorded measurements; rebuilding plots from them does not execute runtime benchmarks. The delivery includes commands for both operations and keeps them separate.

The experiments in Section 7.9 provide separate training histories, checkpoint hashes, input manifests, per-input move witnesses, and repeated timing records. The four checkpoints trained with seeds 17 and 29 are stored in distinct local experiment folders and are not publicly distributed. Included witnesses can be independently replayed without these weights; retraining is needed to reproduce these models. The public seed-13 checkpoint is the reference model.

The workbench is an implementation artifact, not a user study result. Its progressive workflow illustrates how to expose the method's guarantees in practice, but the experiments below measure alignment outputs and algorithmic timings. They do not measure analyst productivity, user satisfaction, or deployment performance under concurrent load.

7. Experimental Evaluation

The evaluation asks whether the candidate is useful before exact repair and whether machine learning improves that candidate enough to justify its cost. We therefore measure the proposed explanation separately from the exact result. The experiments progress from controlled test families to larger synthetic Petri nets, competing approximations, and real event logs. This order separates the effects of semantic constraints, learned ranking, representation, and input scale.

7.1. Protocol, Metrics, and Comparison Boundaries

The reference experiments use an Intel Core i7-1355U CPU and the approximately 1.7-million parameter checkpoint selected at epoch 49. Computation is CPU only, with PyTorch reporting ten intra operation and ten inter operation threads; trace calls are sequential. The checkpoint is loaded once before timing. The synthetic test set has 512 rows from 128 generated families, with 64 rows for each motif/representation cell. The scoring parameters stay fixed throughout the reference test, scaling, and real log experiments. Section 7.9 reports training runs with seeds 17 and 29 and controlled evaluations.

The primary metrics are replay acceptance, exact cost agreement, and the cost gap in Equation (4). We also report full transition-aware move sequence agreement with the

teacher and agreement after replacing transition identities with labels. These sequence measures diagnose imitation of one witness; neither is required for an optimum when several witnesses share its cost. For representation pairs, cost agreement measures whether the same observation receives equal candidate costs on the two equivalent Petri nets.

The guidance ablation preserves the decoder, costs, depth bounds, and test inputs. It disables the neural forward pass and uses deterministic orderings in place of learned rankings. This is a strong baseline because enabledness and bounded model path search already resolve many cases. It directly answers RQ2; comparison with exact search alone would not isolate the contribution of machine learning.

The timers in the reference experiments measure different operations. LARA fast mode includes feature extraction, runtime construction, neural inference when enabled, decoding, and replay. The exact timer includes cost parameter construction, heuristic initialization, search, witness parsing, and independent replay inside the exact wrapper. The approximation benchmark times the PM4Py alignment call and excludes independent parsing and replay. Checkpoint loading, log loading, and model discovery are outside these per trace timers. Thus the medians do not provide a common end to end comparison across all methods. LARA and exact search also return different guarantees: a verified candidate versus a proven optimum. We do not infer a general runtime ranking or a certified alignment speedup from these records. Section 7.9 reports a controlled timing comparison with common start and end points, warmup, and repeated measurements. Candidate costs and coverage remain comparable because they are checked afterward under the same unit visible deviation policy.

The default test decoder uses a catch-up depth of eight and a completion depth of 32. Scaling and real log scripts use a completion depth of 64 while keeping the catch-up depth at eight. Exact timeouts in the stress benchmark last 30 s, and approximation calls receive five seconds. A timeout leaves the exact cost unknown; the corresponding candidate is included in coverage statistics but excluded from cost gap and optimality calculations that require the optimum. The tables state these denominators explicitly.

For representation specific guidance differences, a family bootstrap constructs 10,000 resampled datasets by drawing whole families with replacement, using seed 13 to make the random draws reproducible. The 2.5th and 97.5th percentiles of the resampled differences define the reported 95% intervals. A cell contains 32 families with two observations each. Resampling families respects the dependence between observations and representations. For the aggregate guided versus unguided comparison, we additionally resample 32 whole families within each of the four motifs on every draw. Each family retains its two observations and both representations, preserving the balanced design and pairing. We again use 10,000 percentile resamples with seed 13. Both analyses condition on the single epoch-49 checkpoint trained with seed 13. They quantify sampling within this generator, not variability from initialization, training order, checkpoint selection, or deployment domains. The training runs in Section 7.9 separately assess variation across three training seeds; bootstrap intervals still condition on the evaluated checkpoints.

7.2. Candidate Quality on the Synthetic Test Split

The first experiment answers whether the fast pipeline produces complete executable explanations and how close they are to the optimum (RQ1). All 512 candidates pass independent replay. Of these, 466 have the exact optimal cost, giving 91.0% optimality. Table 10 reports the remaining diagnostics and Figure 11 shows the full gap distribution.

The mean gap is small because most candidates are optimal and the remaining gaps are between one and four. Sequence agreement is substantially lower than cost agreement. This is expected when several legal explanations are optimal and the teacher stores one of

them. The important distinction for the assurance contract is that replay proves feasibility while exact cost comparison proves optimality, regardless of identity agreement.

Table 10. Candidate results on the 512-row synthetic test split. Exact costs come from the teacher/reference alignment; candidate metrics precede any exact repair.

Metric	Result
Candidates verified by replay	512/512 (100.0%)
Candidates with optimal cost	466/512 (91.0%)
Full transition-aware move sequence match	313/512 (61.1%)
Full label level move sequence match	320/512 (62.5%)
Mean/median cost gap	0.180/0
90th/95th percentile cost gap	0/2
Maximum cost gap	4

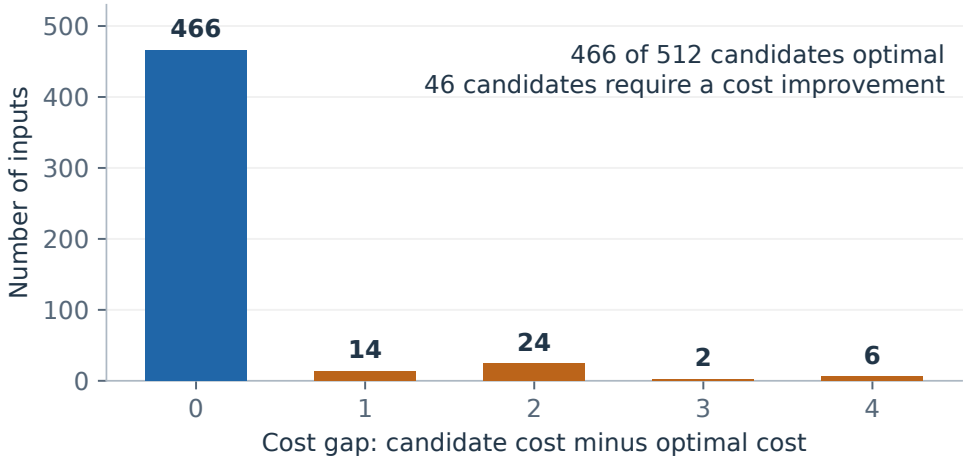


Figure 11. All candidate gaps on the test split. Counts sum to 512; 466 candidates have gap zero and 46 have positive gaps. A low mean gap does not make a particular unchecked candidate optimal.

When certified mode is evaluated on these inputs, it invokes exact search on all 512. The 466 optimal candidates can be confirmed and retained; the remaining 46 are replaced by exact alignments. The 91.0% value therefore counts successful proposals, not traces on which exact search was avoided. This experiment establishes useful candidate quality within the generator, while leaving end to end certification acceleration unproven.

7.3. What Does Learned Guidance Add?

The ablation isolates learned ranking from the model semantics. Unguided decoding also returns 512 legal candidates and reaches optimal cost on 445 of them, or 86.9%. Adding the learned scorer yields 21 more optimal candidates, an increase of 4.10 percentage points, with a 95% paired family-bootstrap interval of $[2.54, 5.66]$ points. The mean gap decreases from 0.258 to 0.180, a reduction of 0.078 with an interval of $[0.051, 0.105]$. These intervals concern the fixed checkpoint and the stratified generator sample, as specified in Section 7.1; they do not establish that retraining would reproduce the gain. Transition-aware sequence agreement rises from 57.8% to 61.1% and label-level agreement from 58.8% to 62.5%.

These aggregate gains are real for the fixed test set but do not imply that machine learning is responsible for most successful alignments. The unguided decoder already synchronizes uniquely enabled matches and searches for short model move paths. Many inputs offer little meaningful ranking choice. To answer where the improvement comes from, Table 11 separates the eight representation cells and Figure 12 shows the corresponding differences.

Table 11. Optimality by motif and representation. Each cell contains 64 inputs. G and U identify guided and unguided decoding; all outputs pass replay. Mean gaps are reported for guided candidates.

Motif	Representation	G Optimal Count (%)	U Optimal Count (%)	G Mean Gap
Ordinary tree	Canonical block	54/64 (84.4%)	55/64 (85.9%)	0.344
Ordinary tree	Isomorphic rename	54/64 (84.4%)	49/64 (76.6%)	0.344
Duplicate/silent	Duplicate prefix	61/64 (95.3%)	44/64 (68.8%)	0.094
Duplicate/silent	Silent routing	63/64 (98.4%)	63/64 (98.4%)	0.031
Concurrency	Parallel	58/64 (90.6%)	58/64 (90.6%)	0.219
Concurrency	Interleaving	58/64 (90.6%)	58/64 (90.6%)	0.219
M pattern	Canonical block	59/64 (92.2%)	59/64 (92.2%)	0.094
M pattern	Non-free choice	59/64 (92.2%)	59/64 (92.2%)	0.094

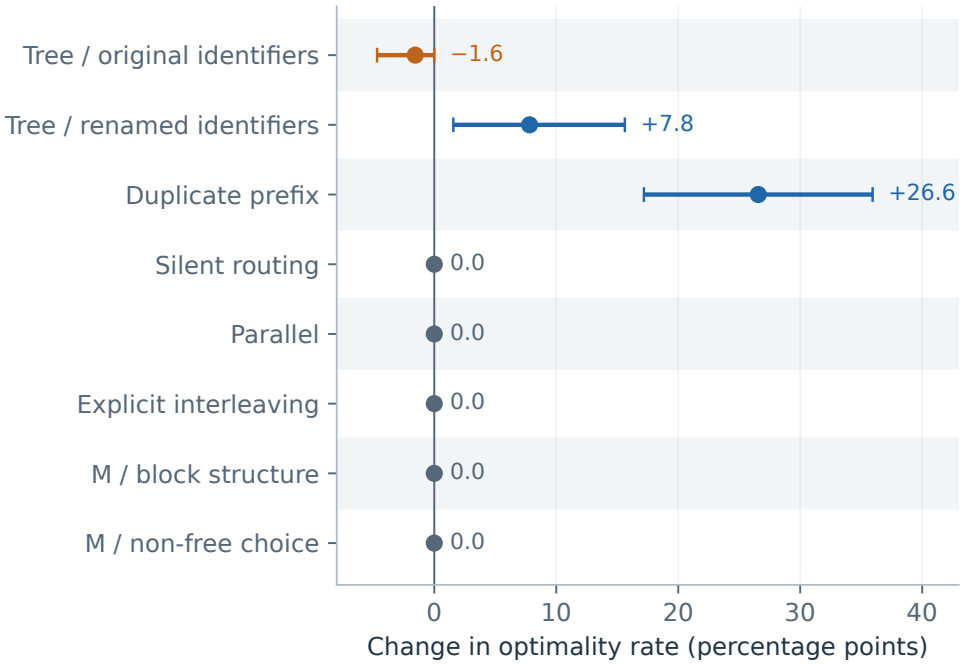


Figure 12. Change in optimality rate when the fixed scorer is enabled. Positive values favor guidance. Intervals are 95% percentile intervals from 10,000 family level bootstrap resamples; the zero width intervals indicate identical sampled success differences, not certainty about other generators. The strongest gain is concentrated in duplicate prefix choices.

The duplicate prefix cell improves from 68.8% to 95.3%, a gain of 26.6 points, with a family bootstrap interval of [17.2, 35.9] points. This is the case in which the trace suffix can distinguish two currently enabled transitions with the same label. Silent routing already attains 98.4% without machine learning: its bounded search can postpone the choice until the distinguishing event. Guided and unguided costs coincide on the parallel/interleaving and M cells. RQ2 therefore has a localized answer: the measured gain is strongest where the decoder must commit before structural enabledness resolves the ambiguity.

7.4. Sensitivity to Equivalent Representations

Equivalent models let us examine reuse without changing the visible alignment problem (RQ3). Table 12 compares candidate costs across each representation pair for the same observation. The ordinary tree pair is particularly informative because the two Petri nets differ only in identifiers.

Guided decoding reaches 84.4% optimality on both canonical and renamed trees, with identical paired costs. Unguided decoding falls from 85.9% to 76.6% after renaming. Guid-

ance is thus slightly worse on canonical identifiers, by -1.6 points with interval $[-4.7, 0.0]$, but better on the renamed identifiers by $+7.8$ points with interval $[1.6, 15.6]$. These signs have a common explanation: a deterministic name order happens to be favorable on one numbering and less favorable on the other. The learned scorer maintains the same quality across this tested renaming.

Table 12. Candidate cost agreement across paired representations. Each row compares 64 observation pairs. Exact costs agree in every pair. Candidate cost equality may include two equally suboptimal results.

Equivalent Representation Pair	Guided Agreement	Unguided Agreement
Canonical tree/isomorphic rename	100.0%	90.6%
Duplicate prefix/silent routing	96.9%	70.3%
Parallel/explicit interleaving	100.0%	100.0%
Block M/non-free-choice M	100.0%	100.0%
All 256 observation pairs	99.2%	90.2%

Across all representation pairs, agreement rises from 90.2% to 99.2%. This is evidence of stability for the tested transformations, not proof of invariance under every isomorphism or activity relabeling. Ties, numerical behavior, and changed label embeddings can still affect choices. All four motif classes and both representations of each class occur during training. Family separation therefore tests new instances, and paired cost agreement tests stability under the listed transformations. Neither is a held-out motif or compiler experiment. The stress generator adds loops and larger nets, but the declining quality and small guidance effect do not establish broad learned structural generalization.

7.5. Scaling: Candidate Latency and Quality Together

The scaling experiment tests whether the candidate remains useful when inputs move beyond the small training distribution. A separate generator produces random block structured Petri nets with sequence, choice, parallelism, loops, duplicate labels, and invisible routing. The nominal sizes are 5, 10, 20, and 40 visible leaf occurrences. An alphabet fraction of 0.7 permits repeated labels; loop playout uses redo probability 0.3 and at most two redos. Deviations are injected at rates 0.15 and 0.35, with ten inputs per configuration. These rates are parameters of the stress generator, not the clean/noisy mixture in Section 5.4. Table 13 reports the sizes, quality measures, and timings for all eight configurations.

Table 13. Scaling results. Size is the requested visible leaf count and $\overline{|T|}$ the rounded mean total transition count. Every configuration has ten legal candidates. n_δ is the number with an exact reference; optimality and mean gap use that denominator. Time columns are median milliseconds per trace; exact timeouts are recorded at 30 s.

Size	Deviation	$\overline{\lvert T\rvert}$	n_δ	Optimal	Mean Gap	Fast ms	Exact ms
5	0.15	7	10	90.0%	0.2	5.5	1.3
5	0.35	6	10	60.0%	1.6	4.6	1.3
10	0.15	13	10	80.0%	1.2	5.0	1.8
10	0.35	13	10	50.0%	1.7	5.3	2.8
20	0.15	28	10	60.0%	3.6	6.9	2.9
20	0.35	26	10	60.0%	3.1	6.0	11.7
40	0.15	55	10	30.0%	8.1	8.7	63.9
40	0.35	54	9	44.4%	11.7	8.6	24.7

Figure 13 shows the recorded latencies across the grid. LARA's guided median ranges from 4.6 to 8.7 ms. The recorded exact median is lower on the small configurations and higher at size 20 with deviation 0.35 and at size 40 for both rates. One exact call times out at size 40 and deviation 0.35. Both timers include replay, but they compare candidate generation with exact optimization (Section 7.1). They do not establish a speedup for returning the same certified result. Certified mode performs exact work in addition to candidate generation.

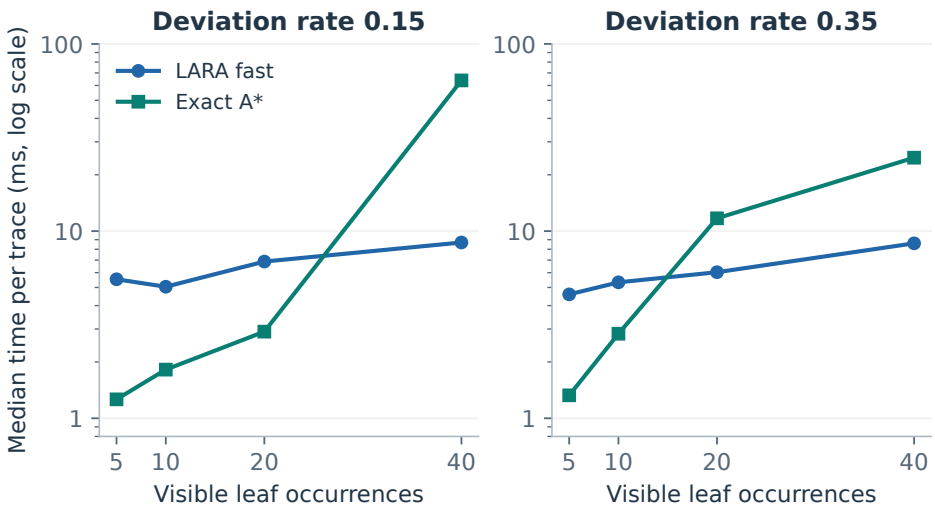


Figure 13. Recorded median latencies on the scaling grid. The vertical axis is logarithmic; the horizontal axis is the number of visible leaf occurrences, not distinct activity labels. Exact search is faster on smaller cells, while its median is larger on several larger cells. These observations do not establish an asymptotic runtime law. A* denotes the A-star search algorithm.

Figure 14 shows the accompanying decline in quality: at size 40 only 30.0% or 44.4% of comparable candidates are optimal, and mean gaps reach 8.1 or 11.7. Replay remains successful on all 80 candidates, demonstrating executability despite this loss of quality. Unguided decoding takes approximately 0.2 to 1.2 ms across the grid and is faster than the exact medians throughout. Guidance contributes little on this stress distribution. The practical question is therefore whether better rankings justify the neural inference cost, not simply whether a neural forward pass is faster than a difficult exact search.

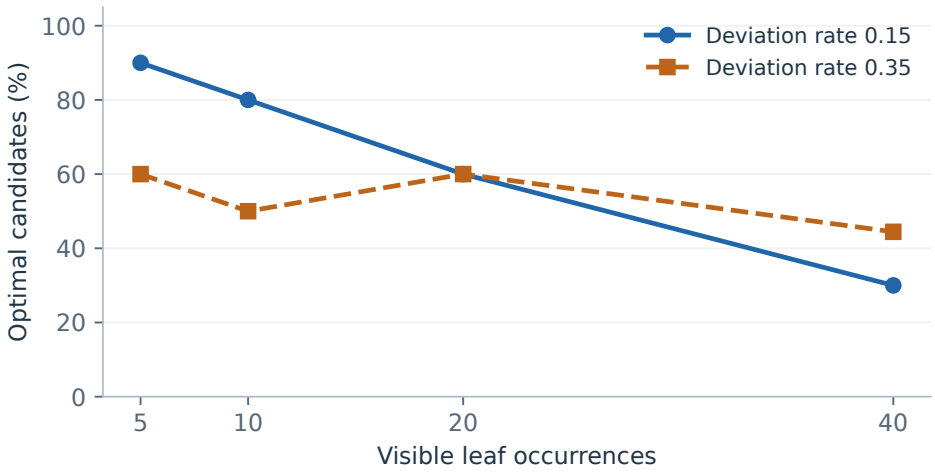


Figure 14. The quality cost of the latency advantage in Figure 13. Optimality declines on larger stress inputs. Each point uses ten exact references except size 40/deviation 0.35, which uses nine. Small cell sizes and changing random Petri nets explain why the curves need not be monotone.

7.6. Comparison with Established Approximations

Competing candidate generators provide a more demanding test than exact search alone. The benchmark uses PM4Py 2.7.23.2 implementations of discounted A* [18], tandem repeats [20], sliding windows [19], fixed horizon search [17], and subset/edit distance approximation [21]. Sliding windows use length 20 and at most five candidates; the fixed horizon is four; the configured expansion budget is 100,000 where supported. The subset size is one within each model variant group and is evaluated only on the test corpus. These are concrete settings, not tuned best case envelopes for each method.

All returned alignments are parsed into transition-aware moves, replayed independently, and rescored with the same costs. This is particularly important for discounted objectives: their internally reported objective need not equal the ordinary visible deviation count. Table 14 reports the test comparison, and Table 15 reports stress results from the approximation benchmark run.

Table 14. Candidate quality on the synthetic test split. All six methods provide 512 legal outputs in this experiment, so the optimality denominator is 512 for every row. A* is the algorithm name (A-star).

Method	Optimal Candidates	Mean Gap
LARA guided	466/512 (91.0%)	0.180
Discounted A*	490/512 (95.7%)	0.090
Tandem repeats	502/512 (98.0%)	0.039
Sliding window	512/512 (100.0%)	0.000
Fixed horizon	502/512 (98.0%)	0.047
Subset/edit distance	360/512 (70.3%)	0.637

Table 15. Stress comparison over 80 inputs. Coverage counts legal outputs out of all 80; n_δ counts legal outputs with an exact reference. Optimality and mean gap are conditional on those n_δ inputs. Time is the median over all calls, including unsuccessful calls. LARA includes replay; PM4Py timings exclude parsing and replay. These times are not a common end to end comparison. Different coverage denominators also limit direct quality comparisons. A* is the algorithm name (A-star).

Method	Coverage	n_δ	Optimal	Mean Gap	Median ms
LARA guided	80/80	79	59.5%	3.797	7.69
Discounted A*	80/80	79	75.9%	1.468	1.17
Tandem repeats	78/80	78	93.6%	0.128	1.37
Sliding window	79/80	79	100.0%	0.000	1.27
Fixed horizon	76/80	76	82.9%	2.263	82.37

Every tested Petri net search/compression approximation exceeds LARA's optimality rate on the small corpus. LARA exceeds the particular subset/edit distance configuration. All test traces fit within one sliding window of 20 events, so the perfect result in that row does not exercise the approximation caused by crossing a window boundary. The comparison illustrates why a small benchmark can be useful for diagnosis without being a sufficient stress test.

The stress comparison again favors established approximations in quality. Discounted A* has the same full legal coverage as LARA, a higher exact cost rate, and a smaller mean gap. Its lower recorded median covers fewer processing stages, so it is not used to claim an end to end timing advantage. Tandem repeats and sliding windows are even closer to optimal where they return comparable results. Their incomplete coverage must remain visible: a perfect conditional optimum rate is not a result on every attempted input. LARA's full coverage is useful, but it is not unique to the learned method on this grid.

The small and stress comparisons rule out a claim that the learned greedy decoder is the best general approximation. They instead identify a reusable scoring mechanism with a measured local advantage over its own unguided baseline. Any later claim of competitive overall approximation quality requires a stronger decoder and renewed comparison using common timing boundaries and coverage accounting.

7.7. Application to Public Event Logs Without Fine Tuning

Real log inputs test the use of one checkpoint with different vocabularies, model structures, and trace lengths. We use a building permit receipt log [29], containing 1434 cases, 116 variants, and 27 activity labels, and the repository's 100-case sample of the road traffic fines log [30], containing ten variants and ten labels. The road traffic results refer to that sample, not the complete public log. The longest variants have 25 and nine events, respectively.

For each log, PM4Py discovers Petri nets using Inductive Miner [31] at noise threshold 0.0 and its infrequent behavior variant [32] at threshold 0.5. Models are discovered from the same log that is subsequently aligned, so this experiment evaluates algorithmic transfer, not independent normative compliance. The positive noise threshold removes infrequent behavior and creates additional deviations against the resulting model. The checkpoint is unchanged; new label strings enter through hashing without any language interpretation or fine tuning.

Table 16 distinguishes variants from cases. If variant v occurs f_v times, Equation (35) defines case weighted optimality:

$$\frac{\sum_v f_v \mathbf{1}[g_v = 0]}{\sum_v f_v}, \tag{35}$$

whereas variant optimality weights each distinct sequence once. The numerators make this distinction concrete. For road traffic, the optimal variant/case counts are 10/10 and 100/100 at noise 0.0, and 7/10 and 94/100 at 0.5. For receipt, they are 58/116 and 1185/1434 at 0.0, and 64/116 and 1070/1434 at 0.5. Frequent variants are easier for this decoder; misses are concentrated in less frequent variants. For the fitting receipt Petri net, all exact costs are zero, but only half of the candidate variants attain zero. A legal, positive cost candidate on a fitting trace is therefore a concrete example of why replay alone is insufficient to infer nonconformance.

Table 16. Real log candidate quality. All variants pass replay. Petri net size gives places and visible/invisible transitions. Guided and unguided decoding have identical quality values in all four configurations. Mean and maximum gaps are over variants.

Log	Noise Threshold	Petri Net Size	Optimal (%)		Cost Gap
		$\lvert P\rvert/\lvert T_v\rvert + \lvert T_\tau\rvert$	Variants	Cases	Mean/Max
Road traffic	0.0	$15/10 + 10$	100.0	100.0	0/0
Road traffic	0.5	$13/10 + 7$	70.0	94.0	0.50/3
Receipt	0.0	$45/27 + 47$	50.0	82.6	0.76/5
Receipt	0.5	$25/24 + 19$	55.2	74.6	0.82/7

Table 17 records higher guided candidate medians than exact search on road traffic and lower ones on receipt, for candidate generation and exact optimization, respectively (Section 7.1). Unguided decoding has lower recorded times and exactly the same cost on every tested variant. These discovered nets have no duplicate visible labels, so they do not test the ambiguity for which the synthetic study finds the clearest learned benefit. This is a

limited test of applying the same checkpoint: two logs, two discovery thresholds, and no independent reference models. The four configurations are not four independent domains. They show executable outputs with new labels and model sizes, but no real log quality benefit from learning and no demonstrated operational speedup. Section 7.9 evaluates held-out cases from three logs with separate discovery cases. Independent normative models and real logs with duplicate labels remain outside the evidence.

Table 17. Recorded median time per variant in milliseconds. Both candidate and exact timers include replay, but they return different guarantees (Section 7.1). These medians do not establish full log or certified alignment speedups. Including the unguided decoder makes clear that the real log candidate advantage is not evidence of a learned quality benefit. A* is the algorithm name (A-star).

Log	Noise	Guided Fast	Unguided Fast	Exact A*
Road traffic	0.0	5.39	0.56	1.92
Road traffic	0.5	5.22	0.48	1.56
Receipt	0.0	16.05	6.01	31.04
Receipt	0.5	6.65	0.61	12.72

7.8. A Failure That Explains the Remaining Gaps

Aggregate rates show how often the decoder fails to be optimal; the worked failure in Figure 15 explains why. Consider test observation $\langle A, D, B, C, C, D \rangle$ in the parallel/interleaving family. Its process permits A, B in either order, then C, D. An optimal alignment skips the early D and one repeated C, giving cost two. LARA instead inserts model moves B, C so that it can synchronize the early D. It reaches the final marking while four observed events remain, forcing four log moves. Its total cost is six, hence gap four.

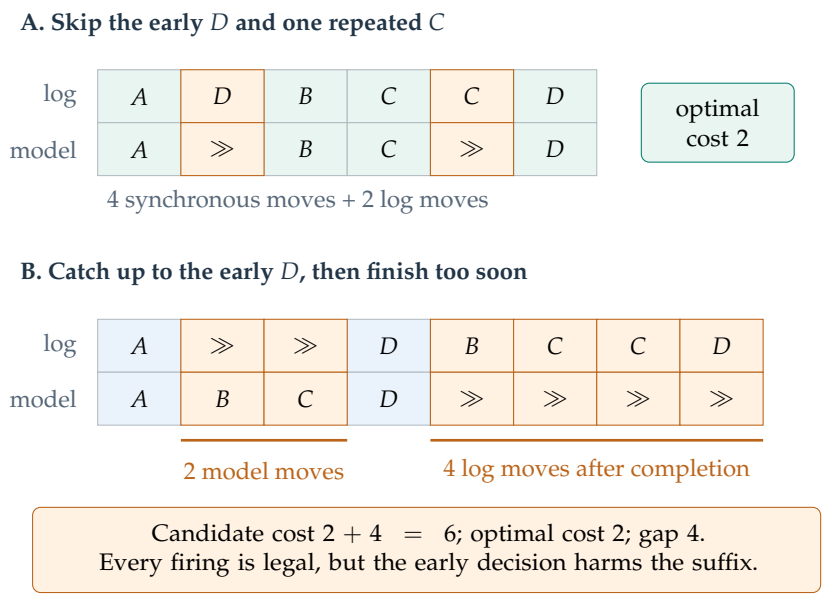


Figure 15. A legal but suboptimal explanation of $\langle A, D, B, C, C, D \rangle$. Each visible deviation is amber. Green cells denote synchronous moves in the optimal explanation; blue cells denote synchronous moves in the candidate. Invisible routing is omitted, and model rows display activity labels for readability. The upper explanation costs two; the lower costs six. The early attempt to synchronize D causes both the two inserted model activities and the four subsequent skipped events.

The error is a decoder commitment, not a replay failure. The default greedy decoder tries a bounded model path before considering a log move and never compares that path with skipping the early D. The trained log score is unused, and earlier choices cannot be revised. At the early D, the bounded path through B, C is available, so this policy cannot

choose the optimal log move regardless of how high its trained log score is. The error can persist even with perfect synchronous rankings. The optional repair procedure explicitly tries a log move at such positions; the next experiment measures its effect.

7.9. Retraining, Transfer, and Timing

The following experiments assess retraining variability, log supervision, candidate repair, structural transfer, held-out real log cases, and timing under common processing boundaries. Inputs and evaluation budgets are fixed before checkpoint evaluation. No test result controls checkpoint selection or a subsequent training run. Computation uses an Intel Xeon Platinum 8468 CPU, Python 3.12.13, PyTorch 2.10.0, and PM4Py 2.7.23.6, with one intra operation and one inter operation thread. The seed-13 reference run uses an Intel Core i7-1355U CPU, ten intra operation and ten inter operation threads, and PM4Py 2.7.23.2. Quality evaluations allow five seconds per candidate and 30 s for exact search, with a 31-s external cutoff for the latter. Prefix depth is eight; completion depth is 32 on the 512-row test set and 64 on the structural and real log inputs. Every reported candidate cost is independently replayed.

7.9.1. Retraining, Log-Loss Ablation, and Candidate Repair

Full-objective runs for seeds 17 and 29 follow Section 5.10. Table 18 reports their selected epochs and results alongside seed 13 and the optional repair decoder. The unguided control needs no training. The same 512 test rows are used in every comparison. Variation across the three full-objective checkpoints summarizes these particular runs; the small seed count and differences in execution environment limit population-level conclusions.

Table 18. Training seed variation and optional repair on the 512-row synthetic test set. Every condition returns 512 legal candidates; all 512 exact references are available. Epoch denotes the validation-selected checkpoint.

Decoder	Seed	Epoch	Optimal Count	Optimal (%)	Mean Gap
Greedy, unguided	n/a	n/a	445	86.91	0.258
Greedy, guided	13	49	466	91.02	0.180
Greedy, guided	17	27	463	90.43	0.188
Greedy, guided	29	48	465	90.82	0.184
Repair, unguided	n/a	n/a	479	93.55	0.111
Repair, guided	13	49	498	97.27	0.039

The full-objective checkpoints produce 466, 463, and 465 optimal candidates for seeds 13, 17, and 29. Their mean optimality is 90.76% with sample standard deviation 0.30 percentage points; mean gap is 0.184 with sample standard deviation 0.004. All three improve on the 445 optimal unguided candidates. The gains are 4.10, 3.52, and 3.91 points. Using the same stratified paired family bootstrap as Section 7.1, their respective 95% intervals are $[2.54, 5.66]$, $[1.95, 5.08]$, and $[2.54, 5.27]$ points. These intervals describe generator sampling for each fixed fitted model, while the standard deviation describes the three observed training runs. They should not be interpreted as interchangeable uncertainty measures.

For each of seeds 17 and 29, a matched run sets only the log-loss coefficient to zero. Architecture, remaining losses, data, seed, and optimizer settings are unchanged. Each run retains its own validation selection, so the intervention includes its effect on the learning-rate schedule and selected epoch. Table 19 reports default greedy decoding for both

conditions. This tests whether supervising the unused head helps the shared representation; it does not isolate each gradient pathway or redesign the loss around decoded cost.

Table 19. Matched log-loss ablation with default greedy decoding on 512 test rows. Full retains log supervision; no log sets only its coefficient to zero. All candidates pass replay. The difference is full minus no log in percentage points.

Seed	Epoch Full	Epoch No Log	Optimal Full	Optimal No Log	Difference (Points)
17	27	50	463	465	−0.39
29	48	48	465	467	−0.39

Removing log supervision yields 465 rather than 463 optima for seed 17 and 467 rather than 465 for seed 29. The full-minus-ablated difference is −0.39 points for each seed, with paired family-bootstrap intervals $[-1.56, 0.78]$ and $[-1.17, 0.00]$. These small differences do not establish a benefit from log supervision for the default decoder. The full objective defines the reference condition; the ablation and repair experiments separately assess log supervision and the use of log scores during decoding.

Using the seed-13 scores, single-log repair raises optimal candidates from 466 to 498, or from 91.02% to 97.27%, while the mean gap falls from 0.180 to 0.039. Unguided repair improves from 445 to 479 optimal candidates, or from 86.91% to 93.55%, with mean gap 0.111. Thus additional candidate exploration helps substantially even without learning. Guided and unguided repair differ in both move rankings and trial order, so their difference cannot be attributed solely to the log head. The failure in Figure 15 is repaired from cost six to the optimum of two. The 6.25-point improvement over guided greedy decoding has a paired family-bootstrap interval of $[3.52, 9.38]$ points, conditional on this checkpoint and test generator. The repaired method misses 14 optima and remains a finite heuristic. Its additional computation is included in the timing comparison below.

7.9.2. Structures Excluded from Training and a Second Compiler

We generate 64 acyclic process trees and 64 trees with a loop at the root, using seed 703. Each class balances nominal visible sizes 6, 10, 14, and 18; sequence, choice, and parallel operators have weights 0.4, 0.3, and 0.3. The loop class adds a single visible redo leaf. An alphabet fraction of 0.7 permits repeated labels. Each tree supplies a clean playout and a corrupted observation at deviation rate 0.25; loop playout uses redo probability 0.5 and at most two redos. The same two observations are paired across the LARA block compiler and PM4Py's process-tree compiler, yielding 512 inputs.

Before accepting a tree, we exclude either compilation if its directed Petri-net graph is isomorphic to a training graph. The comparison preserves place/visible-transition/silent-transition types, arc weights, and initial/final tokens, while ignoring identifiers and activity strings. A structural hash filters candidates for full graph isomorphism checking. Two acyclic proposals are rejected. All training graphs are checked to be acyclic; both compilations of every loop input are checked to be cyclic. The loop class is therefore absent from training, and the PM4Py compiler provides an additional representation test. This does not exclude repeated visible languages or claim novelty relative to the validation split. Exact costs agree across all 256 paired observations; every clean observation has exact cost zero.

Table 20 shows complete legal and exact coverage, but limited transfer of the learned advantage. On acyclic inputs, the three greedy checkpoints attain 78.1% to 81.2% optimality with the LARA compiler and 80.5% to 82.0% with PM4Py; unguided decoding attains 76.6% and 82.8%, respectively. On loops, all three checkpoints fall below their unguided control under both compilers. For seed 13, mean gap on loops is 4.156 with the LARA

compiler and 4.797 with PM4Py; guided repair reduces these to 1.781 and 2.375. Repair improves candidates, but neither accepting new structures nor preserving their exact paired costs demonstrates a general learned advantage. The unfavorable cyclic result is evidence against that stronger claim.

Table 20. Optimal candidate rates (%) on structures excluded from training. Each row has 128 inputs from 64 trees and complete candidate/exact coverage. The three seeds use greedy decoding; guided repair uses seed 13.

Class, Compiler	Unguided	Seed 13	Seed 17	Seed 29	Repair Unguided	Repair Guided
Acyclic, LARA	76.6	81.2	81.2	78.1	82.0	86.7
Acyclic, PM4Py	82.8	82.0	82.0	80.5	87.5	87.5
Loop, LARA	54.7	52.3	51.6	52.3	64.1	65.6
Loop, PM4Py	55.5	53.9	51.6	53.9	65.6	68.0

7.9.3. Held-Out Cases from Three Public Logs

The held-out study uses receipt, the road traffic sample, and the Sepsis Cases event log [33], which contains 1050 cases, 15,214 events, and 16 activity labels. Only activity sequences enter the experiment. For each log, seed 37 fixes a 70% discovery and 30% evaluation split by case before discovery. The resulting discovery/evaluation counts are 70/30 for road traffic, 1003/431 for receipt, and 735/315 for sepsis. Inductive Miner uses only discovery cases at thresholds 0.0 and 0.5. All evaluation variants are retained: 6, 48, and 272 per threshold, respectively. Case sets are disjoint, but identical activity variants may occur in both sets. No target log is used for neural training or selection.

Table 21 reports variant and case counts. Every greedy cost for all three training seeds equals the unguided cost on the same input. Guided repair improves receipt at threshold 0.5 from 33 to 39 optimal variants and from 398 to 421 optimal cases, exactly matching unguided repair. On sepsis at threshold 0.5, guided repair gives 267 optimal variants and 301 optimal cases, while unguided repair gives 269 variants and 303 cases. Other conditions have unchanged optimal counts under repair, although receipt at threshold 0.0 has a smaller mean gap. Thus the extra exploration helps some cases, but learning supplies no real-log quality advantage in this study.

Table 21. Held-out real-log cases. Fractions report optimal counts over all evaluated variants or cases. Greedy columns use seed 13. Repair columns are variant counts; guided repair also uses seed 13. All 652 variant/model inputs have exact references and legal candidates.

Log	Threshold	Greedy Variants	Greedy Cases	Repair Guided Variants	Repair Unguided Variants
Road traffic	0.0	6/6	30/30	6/6	6/6
Road traffic	0.5	6/6	30/30	6/6	6/6
Receipt	0.0	40/48	423/431	40/48	40/48
Receipt	0.5	33/48	398/431	39/48	39/48
Sepsis	0.0	254/272	297/315	254/272	254/272
Sepsis	0.5	263/272	297/315	267/272	269/272

This experiment separates model discovery from evaluated cases across three domains. It does not supply independent normative process models, a temporal split, or operational deployment evidence. The road traffic sample is small. None of the six discovered nets has duplicate visible labels, so this test lacks the real-world ambiguity with the

clearest synthetic guidance benefit. Thresholds are repeated conditions within a log, not independent domains.

7.9.4. Timing with Common Processing Boundaries

A common timing harness evaluates eight methods on the 512-row synthetic test set without concurrent training or quality evaluation jobs. Each call starts with the same in-memory Petri net and trace and ends after parsing, when required, and final independent replay. Method-specific parameter construction, feature extraction, neural inference, candidate construction or search, and existing internal checks are included. Checkpoint loading, log loading, and model discovery are excluded for every method. Three warmup inputs per method precede five repetitions; input and method order are shuffled with seed 902. Every call has the same five-second wall-time budget. Approximation parameters match Section 7.6. The batch-based subset approximation is excluded because its unit of computation is a group of traces; the timing comparison concerns per-trace methods only.

For each input and method, we first take the median of its five calls. Table 22 reports the median and interquartile range of those 512 input medians, plus legal coverage and optimality over all 2560 attempted calls per method. The interquartile range describes differences across inputs, not a confidence interval. All 20,480 calls finish with legal outputs. Guided greedy decoding takes a median 2.060 ms, guided repair 2.325 ms, exact A* 0.664 ms, and sliding windows 0.330 ms. Unguided greedy decoding takes 0.307 ms. Thus the learned candidate is slower than exact search on these small inputs; repair adds a modest median cost while improving quality. Sliding windows are optimal throughout this test because its short traces fit inside one window. Fixed-horizon search is slower at 4.225 ms but has a higher optimal rate than guided repair. These measurements support a runtime comparison on this host and test distribution with equal outer boundaries. Candidate generation and exact search provide different guarantees, and the experiment establishes no reduction in certified-mode computation or deployment latency.

Table 22. Common-boundary CPU timings, including final independent replay. Each method has 512 inputs and five calls per input. Time summaries aggregate the per-input medians. Optimality uses all attempted calls as denominator. A* is the algorithm name (A-star).

Method	Legal Calls	Optimal (%)	Median (ms)	Interquartile Range (ms)
Greedy, unguided	2560/2560	86.91	0.307	[0.275, 0.344]
Greedy, guided	2560/2560	91.02	2.060	[2.006, 2.118]
Repair, guided	2560/2560	97.27	2.325	[2.203, 2.467]
Exact A*	2560/2560	100.00	0.664	[0.579, 0.778]
Discounted A*	2560/2560	95.70	0.414	[0.364, 0.471]
Tandem repeats	2560/2560	98.05	0.338	[0.277, 0.417]
Sliding window	2560/2560	100.00	0.330	[0.269, 0.418]
Fixed horizon	2560/2560	98.05	4.225	[3.247, 5.248]

8. Discussion and Limitations

The results are most useful when they guide a concrete choice about when to generate a candidate and what to infer from it. This section connects the three research questions to that choice, distinguishes the semantic contract from empirical performance, and identifies the changes needed for stronger claims. It also records limits of the evidence so that the observed transfer is not mistaken for general purpose alignment competence.

8.1. Interpreting the Three Research Questions

For RQ1, the experiments show that the decoder and replay pipeline produced legal candidates on all evaluated inputs, with 91.0% optimality on the controlled test split. Optional single-log repair raises the reference checkpoint's rate to 97.3%, while leaving 14 suboptimal candidates. The formal guarantee is conditional: when replay accepts, the candidate is a complete executable explanation and its cost is an upper bound. The observed 100% acceptance rate does not prove that the bounded decoder always reaches the final marking. Larger gaps on stress inputs and the worked failure show that a reliable legality check can coexist with a weak candidate policy.

For RQ2, machine learning has a measurable but concentrated effect. The duplicate prefix improvement demonstrates that global trace context can help decide among equal labels before the decoder reaches their distinguishing continuation. The identifier renaming experiment shows more stable candidate costs than the name ordered baseline. The remaining motifs show no cost improvement. The evidence therefore supports using learned rankings for selected ambiguities, rather than attributing the overall success rate to the neural network.

For RQ3, fixed checkpoints accept new Petri net sizes and label vocabularies without fine tuning. The structural transfer experiment excludes training-isomorphic graphs and tests a cyclic class absent from training under two compilers. It establishes measured behavior on those structural shifts, but the inconsistent guidance gains do not support broad learned superiority. Candidate quality degrades on larger stress inputs, and real log results do not show an advantage from learned rankings. Table 22 supplies comparable per-trace timings on the controlled test set; it does not convert a feasible candidate into an exact result or demonstrate faster certification. Reuse of parameters, successful replay, and improved learned decision quality remain separate empirical claims.

8.2. What a Verified Candidate Can and Cannot Tell an Analyst

A verified candidate is useful because it is an executable explanation of the observed case. It can support visualization, an initial inspection of deviations, and a bound on the best achievable cost. Its deviation locations should nevertheless be read in the light of its optimization status. The fitting receipt model makes this concrete: a positive candidate cost does not prove that the trace deviates from the model, because a zero-cost alignment may exist. If the goal is to assert the minimum deviation count, exact evidence remains necessary.

Likewise, equal optimal costs do not identify a unique diagnostic story. Different optimal alignments may allocate deviations or synchronize independent activities differently. Certification establishes minimal cost under the chosen policy, not the uniquely correct account of what happened in the organization. Business interpretation still depends on the model, event data quality, and the suitability of the cost function.

Progressive presentation is a practical application of these distinctions. A workbench can show a candidate and its replay status immediately and then add exact comparison when available. The implementation supports that sequence. A deployment policy that starts exact search first and invokes candidate generation only for slow cases could reduce unnecessary inference, but this policy has not been evaluated. The threshold would depend on model structure, hardware, expected quality, and the analyst's need for a certificate.

8.3. Threats to Validity

Training distribution. The corpus is small and intentionally structured. It contains four motif types, three of which use fixed controlled constructions. Family level splitting prevents separation of a generated family's views, but it does not exclude repeated visible

languages or motifs across splits. The structural test explicitly excludes graphs isomorphic to training inputs and holds out a cyclic class, with a second compiler. Both compilers receive generated process trees, and the test spans only 128 trees. It is not a sample of unrestricted industrial Petri nets. Training on abstract labels also leaves business semantics, resources, and time outside the study.

Teacher and objective. Exact targets are verified for the supplied cost model, but only one optimal witness is retained. Sequence agreement therefore mixes candidate quality with witness choice. The masking rule for duplicate prefix targets is motif specific. The matched loss ablation measures the effect of supervising the log head on greedy decoding for two seeds; optional repair uses that head to order alternatives. Neither experiment establishes an objective aligned with minimum decoded cost, and the auxiliary region losses remain unablated. Model scores are global transition preferences rather than marking- and position-specific action values.

Statistical coverage. The full objective is evaluated with three training seeds; the loss ablation has two matched pairs. Reporting all selected checkpoints and their sample standard deviation reveals variation in these runs, but three seeds cannot tightly characterize arbitrary retraining. The seed-13 run uses a different execution environment from the runs for seeds 17 and 29. Family-bootstrap intervals condition on each fitted checkpoint and generator; they are not intervals over training seeds or deployment domains. The stress cells contain only ten inputs. The held-out real log study covers three logs, including one restricted road traffic sample, with a single case split per log. Perfect replay coverage in these collections should not be generalized to arbitrary systems.

Comparison protocol. The evaluation uses one exact backend and concrete approximation configurations. Other exact implementations, window sizes, or representative subsets may change the results. The timers in the reference experiments cover different operations and provide descriptive measurements. The common timing harness includes method-specific preparation, conversion, and replay, with warmup, repeated calls, and full coverage accounting. Its comparison is limited to per-trace implementations on the controlled test set; the batch subset method and real-log evaluations are outside that protocol. Internal checks remain part of each implementation's measured cost. Different hardware, software, concurrency, or target processes may change the runtime ordering.

Real log interpretation. The models in the reference real log experiment use the same cases for discovery and alignment. The held-out study discovers models from separate cases and evaluates all held-out variants, but uses the same source log and may share variants across the case split. This tests transfer to unseen cases, not compliance with an independent organizational specification. Variant and case weighting answer different questions. Both are reported, and the limited road traffic sample and absence of duplicate visible labels remain explicit.

Semantic and computational scope. The evaluated setting uses ordinary control flow Petri nets and fixed unit visible deviation costs. Neural features omit explicit arc weights, current decoder markings, and data guards. The scorer uses dense attention, and the decoder's depth bounds do not bound the width of marking exploration. Neither component has a demonstrated constant latency guarantee. The guarantee that survives these implementation limits is the conditional replay/certification contract.

8.4. Methodological Extensions Suggested by the Evidence

The optional repair procedure addresses the failure in Figure 15 by explicitly testing log alternatives, and its measured gains justify further investigation of candidate exploration. Its trials each force only one log position; the remaining continuation is greedy. A decoder could instead rank synchronous, log, and model alternatives jointly and retain

several partial continuations. Recomputing scores from the current marking or predicting model preferences per event position could help it respond to earlier choices. These extensions would need separate evaluation against unguided search and strong classical approximations, including their extra computation.

A separate extension concerns certification. A candidate verified by replay supplies a feasible incumbent cost. An exact search algorithm can potentially use that cost for pruning while relying on admissible lower bounds for correctness. Such an integration would need to expose the relevant solver interface, preserve exactness, and demonstrate reduced total work. Merely comparing a candidate with an independently computed optimum, as done here, cannot support a certification speedup claim.

Training could also focus on the decisions that the active decoder actually makes, collect multiple optimal witnesses, or penalize downstream candidate gaps more directly. Additional structural diversity and deliberate tests of hash relabeling would strengthen evidence for reuse. The auxiliary regions would require independent validation before being treated as meaningful process decompositions. These are research directions motivated by observed limitations, not capabilities of the evaluated checkpoint.

9. Conclusions

The aim of this study is to find a useful role for reusable machine learning in alignment-based conformance checking while keeping the meaning of an explanation explicit. LARA combines a pretrained scorer for graphs and traces with a decoder that tracks the marking, independent replay, and optional exact comparison. The scorer proposes move preferences; replay establishes executability and an upper bound; successful exact computation establishes the optimum or supplies a replacement.

Across three training seeds, guided greedy decoding attains a mean optimality of 90.76% with a sample standard deviation of 0.30 percentage points on the controlled synthetic test set, which can be compared with 86.91% without guidance. The reference checkpoint gives 466 rather than 445 optima, and optional single-log repair raises this to 498 of 512. The matched loss ablation provides no evidence that supervising the unused log head helps greedy decoding. Tests excluding training-isomorphic graphs and including cyclic models under two compilers reveal inconsistent learned gains. Held-out cases from three real logs show no quality advantage from guidance. All evaluated candidates pass replay, but these coverage observations do not prove completeness of bounded decoding. The reference greedy policy ranks fifth among six methods in Table 14; common-boundary timings also show that it is slower than an exact search on the small test set. Certified mode runs an exact search for every trace.

The resulting contribution is a transparent, reproducible study of learned candidate scoring within an alignment pipeline whose guarantees come from process semantics. The most promising next steps are making stronger decisions between competing move types, conducting broader tests of structural transfer, and using verified incumbents to reduce exact search work. Their value must be demonstrated in candidate quality and total computation, with the same explicit separation between a plausible prediction and a proven alignment property.

Author Contributions: Software, A.B.; writing (review and editing), A.B. and W.M.P.v.d.A. All authors have read and agreed to the published version of the manuscript.

Funding: Funded by the European Union. This work has received funding from the European High Performance Computing Joint Undertaking (JU) and from the German Federal Ministry of Research, Technology and Space (BMFTR), the Ministry of Culture and Science of North Rhine-Westphalia

(MKW NRW), and the Hessian Ministry of Science and Research, Arts and Culture (HMWK) under grant agreement No. 101250682.

Institutional Review Board Statement: Not applicable. This study uses synthetic traces and previously published event logs.

Informed Consent Statement: Not applicable.

Data Availability Statement: The application is publicly available at https://lara-pm.app (accessed on 29 September 2026). The implementation and experiment scripts are available at https://github.com/fit-alessandro-berti/lara-align (accessed on 29 September 2026). The `ft-journal` branch includes the reference synthetic corpus, numerical result records, and figure sources in `paper_applsci`. The pretrained checkpoint is available at https://www.alessandroberti.it/checkpoint_lara_latest.tar.gz (accessed on 29 September 2026); the journal source records its SHA-256 checksum. The main text in Section 6 provides both data and trained checkpoint download links. The public event log sources are cited in Section 7.7.

Acknowledgments: Generative AI tools (OpenAI Codex) supported software implementation and manuscript generation. The authors take responsibility for the content.

Conflicts of Interest: The author W.M.P.v.d.A. was employed by the company Celonis. The remaining author declares that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

A*	Heuristic shortest-path search
BFS	Breadth-first search
CPU	Central processing unit
LARA	Learned Alignment with Replay Assurance
PNML	Petri Net Markup Language
RQ	Research question
XES	Extensible Event Stream

References

1. Carmona, J.; van Dongen, B.; Solti, A.; Weidlich, M. *Conformance Checking: Relating Processes and Models*; Springer: Cham, Switzerland, 2018. [CrossRef]
2. Bommasani, R.; Hudson, D.A.; Adeli, E.; Altman, R.; Arora, S.; von Arx, S.; Bernstein, M.S.; Bohg, J.; Bosselut, A.; Brunskill, E.; et al. On the Opportunities and Risks of Foundation Models. *arXiv* **2021**, arXiv:2108.07258. [CrossRef]
3. Rizk, Y.; Venkateswaran, P.; Isahagian, V.; Narcomey, A.; Muthusamy, V. A Case for Business Process-Specific Foundation Models. In *Business Process Management Workshops*; Lecture Notes in Business Information Processing; Springer: Cham, Switzerland, 2024; Volume 492, pp. 44–56. [CrossRef]
4. van Dongen, B.F. Efficiently Computing Alignments: Using the Extended Marking Equation. In *Business Process Management*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2018; Volume 11080, pp. 197–214. [CrossRef]
5. LeCun, Y.; Bengio, Y.; Hinton, G. Deep learning. *Nature* **2015**, *521*, 436–444. [CrossRef] [PubMed]
6. Hendrycks, D.; Gimpel, K. Gaussian Error Linear Units (GELUs). *arXiv* **2016**, arXiv:1606.08415. [CrossRef]
7. Gilmer, J.; Schoenholz, S.S.; Riley, P.F.; Vinyals, O.; Dahl, G.E. Neural Message Passing for Quantum Chemistry. In *Proceedings of the 34th International Conference on Machine Learning*; PMLR (Proceedings of Machine Learning Research): Cambridge, MA, USA, 2017; Volume 70, pp. 1263–1272. Available online: https://proceedings.mlr.press/v70/gilmer17a.html (accessed on 11 September 2026).
8. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is All you Need. In *Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2017; Volume 30. Available online: https://proceedings.neurips.cc/paper_files/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html (accessed on 11 September 2026).
9. Ba, J.L.; Kiros, J.R.; Hinton, G.E. Layer Normalization. *arXiv* **2016**, arXiv:1607.06450. [CrossRef]

10. Guo, C.; Pleiss, G.; Sun, Y.; Weinberger, K.Q. On Calibration of Modern Neural Networks. In *Proceedings of the 34th International Conference on Machine Learning*; PMLR (Proceedings of Machine Learning Research): Cambridge, MA, USA, 2017; Volume 70, pp. 1321–1330. Available online: https://proceedings.mlr.press/v70/guo17a.html (accessed on 14 September 2026).
11. Szegedy, C.; Vanhoucke, V.; Ioffe, S.; Shlens, J.; Wojna, Z. Rethinking the Inception Architecture for Computer Vision. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*; IEEE: New York, NY, USA, 2016; pp. 2818–2826. [CrossRef]
12. Loshchilov, I.; Hutter, F. Decoupled Weight Decay Regularization. In Proceedings of the 7th International Conference on Learning Representations, New Orleans, LA, USA, 6–9 May 2019. Available online: https://openreview.net/forum?id=Bkg6RiCqY7 (accessed on 11 September 2026).
13. Srivastava, N.; Hinton, G.; Krizhevsky, A.; Sutskever, I.; Salakhutdinov, R. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *J. Mach. Learn. Res.* **2014**, *15*, 1929–1958. Available online: https://jmlr.org/papers/v15/srivastava14a.html (accessed on 14 September 2026).
14. Adriansyah, A.; van Dongen, B.F.; van der Aalst, W.M.P. Conformance Checking Using Cost-Based Fitness Analysis. In Proceedings of the 2011 IEEE 15th International Enterprise Distributed Object Computing Conference, Helsinki, Finland, 29 August–2 September 2011; pp. 55–64. [CrossRef]
15. Casas-Ramos, J.; Mucientes, M.; Lama, M. REACH: Researching Efficient Alignment-Based Conformance Checking. *Expert Syst. Appl.* **2024**, *241*, 122467. [CrossRef]
16. Genga, L.; Winter, K. Artificial intelligence in conformance checking: State of the art and research agenda. *Process Sci.* **2025**, *2*, 9. [CrossRef]
17. van Dongen, B.; Carmona, J.; Chatain, T.; Taymouri, F. Aligning Modeled and Observed Behavior: A Compromise Between Computation Complexity and Quality. In *Advanced Information Systems Engineering*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2017; Volume 10253, pp. 94–109. [CrossRef]
18. Boltenhagen, M.; Chatain, T.; Carmona, J. A Discounted Cost Function for Fast Alignments of Business Processes. In *Business Process Management*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2021; Volume 12875, pp. 252–269. [CrossRef]
19. Bogdanov, E.; Cohen, I.; Gal, A. A Scalable and Near-Optimal Conformance Checking Approach for Long Traces. *arXiv* **2024**, arXiv:2406.05439. [CrossRef]
20. Reißner, D.; Armas-Cervantes, A.; La Rosa, M. Efficient Conformance Checking Using Approximate Alignment Computation with Tandem Repeats. *arXiv* **2020**, arXiv:2004.01781. [CrossRef]
21. Fani Sani, M.; van Zelst, S.J.; van der Aalst, W.M.P. Conformance Checking Approximation Using Subset Selection and Edit Distance. In *Advanced Information Systems Engineering*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2020; Volume 12127, pp. 234–251. [CrossRef]
22. Padró, L.; Carmona, J. Computation of Alignments of Business Processes through Relaxation Labeling and Local Optimal Search. *Inf. Syst.* **2022**, *104*, 101703. [CrossRef]
23. Tax, N.; Verenich, I.; La Rosa, M.; Dumas, M. Predictive Business Process Monitoring with LSTM Neural Networks. In *Advanced Information Systems Engineering*; Lecture Notes in Computer Science; Springer: Cham, Switzerland, 2017; Volume 10253, pp. 477–492. [CrossRef]
24. Bukhsh, Z.A.; Saeed, A.; Dijkman, R.M. ProcessTransformer: Predictive Business Process Monitoring with Transformer Network. *arXiv* **2021**, arXiv:2104.00721. [CrossRef]
25. Sommers, D.; Menkovski, V.; Fahland, D. Process Discovery Using Graph Neural Networks. In *Proceedings of the 2021 3rd International Conference on Process Mining (ICPM)*; IEEE: New York, NY, USA, 2021; pp. 40–47. [CrossRef]
26. Bengio, Y.; Lodi, A.; Prouvost, A. Machine Learning for Combinatorial Optimization: A Methodological Tour d'Horizon. *Eur. J. Oper. Res.* **2021**, *290*, 405–421. [CrossRef]
27. Yonetani, R.; Taniai, T.; Barekatain, M.; Nishimura, M.; Kanezaki, A. Path Planning Using Neural A* Search. In *Proceedings of the 38th International Conference on Machine Learning*; PMLR (Proceedings of Machine Learning Research): Cambridge, MA, USA, 2021; Volume 139, pp. 12029–12039. Available online: https://proceedings.mlr.press/v139/yonetani21a.html (accessed on 11 September 2026).
28. Berti, A.; van Zelst, S.; Schuster, D. PM4Py: A Process Mining Library for Python. *Softw. Impacts* **2023**, *17*, 100556. [CrossRef]
29. Buijs, J. *Receipt Phase of an Environmental Permit Application Process (WABO), CoSeLoG Project*; Dataset; Eindhoven University of Technology: Eindhoven, The Netherlands, 2014. [CrossRef] [PubMed]
30. de Leoni, M.; Mannhardt, F. *Road Traffic Fine Management Process*; Dataset; Eindhoven University of Technology: Eindhoven, The Netherlands, 2015. [CrossRef] [PubMed]
31. Leemans, S.J.J.; Fahland, D.; van der Aalst, W.M.P. Discovering Block-Structured Process Models from Event Logs - A Constructive Approach. In *Application and Theory of Petri Nets and Concurrency*; Lecture Notes in Computer Science; Springer: Berlin/Heidelberg, Germany, 2013; Volume 7927, pp. 311–329. [CrossRef]

32. Leemans, S.J.J.; Fahland, D.; van der Aalst, W.M.P. Discovering Block-Structured Process Models from Event Logs Containing Infrequent Behaviour. In *Business Process Management Workshops*; Lecture Notes in Business Information Processing; Springer: Cham, Switzerland, 2014; Volume 171, pp. 66–78. [CrossRef]
33. Mannhardt, F. *Sepsis Cases—Event Log*; Dataset, Version 1; via 4TU.ResearchData; Eindhoven University of Technology: Eindhoven, The Netherlands, 2016. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.