

009

Petri nets and process discovery

A field reference to the models, algorithms, quality measures and benchmarks behind discovering a process from its event log

AUTHORS

Ali Akram · Bahaulddin Shammery · Yassine Say · Mohammed Hashmi

/ EXECUTIVE SUMMARY

PURPOSE	Organisations record what their systems do in event logs, and process discovery turns those logs into a model of how the work really flows. This reference explains the ideas a reader needs to follow current work in the field.
CONTENTS	It starts with event logs, Petri nets, workflow nets and the soundness property that marks a model as safe to use. It then explains how the main discovery algorithms work, from the Alpha miner to the optimisation-based miners of 2023 to 2026. It closes with how a discovered model is scored, how the field benchmarks its methods, and the problems that remain open.
MAIN POINTS	Petri nets are both a diagram and a matrix, so a model can be drawn for people and analysed by a computer. The inductive miner family guarantees a sound model by construction, and the strongest recent open methods build on its recursion. On the Process Discovery Contest 2024 data, the authors of the best openly published Petri-net method report 88.24 percent, against about 70 percent for the 2013 inductive miner [1]. The last broad comparison of methods on real logs dates from 2019 [2].

Most organisations cannot fully describe the processes they run, although their information systems record what actually happened as an instance identifier, an activity name and a timestamp, repeated across millions of rows. *Process mining* is the discipline that turns those records into models. Its central task is *process discovery*, which means building, from such a record (an event log), a model that explains the observed behaviour [3].

Petri nets [4], [5] are a widely used formal model for this work, because they can be both drawn and computed with. A practitioner can read a net as a diagram of boxes, circles and arrows. The same net is also a matrix, over which one can prove conservation laws, find repeating behaviour and decide whether the model is correct. A Petri net can show two activities running at the same time and can then be used to prove properties of that behaviour, for example that both branches always finish.

This reference covers the ideas that current discovery work depends on, from event logs and Petri nets to the benchmarks used to compare discovery methods. Formal definitions sit in the appendix, and the text explains each one in words.

This section gives a short plain-language tour of the ideas the rest of this reference uses. All definitions are standard, and we checked each one against its primary source. A source is cited where a formulation belongs to one author, while a definition treated as common knowledge carries no citation.

Event logs

An event log records the executions of a process. Each execution, such as one order, one permit application or one repair job, is a *process instance*, and the log stores it as a *trace*, the ordered sequence of activities recorded for that instance. The same activity names recur across traces, and discovery works from those recurring sequences.

Nets, markings, and firing

A Petri net has two kinds of node. *Places*, drawn as circles, hold work waiting at some stage. *Transitions*, drawn as boxes, are the steps of the process. Arrows only ever join a place to a transition or a transition to a place. The places with an arrow into a transition are its inputs, and the places it points to are its outputs. Each transition carries an activity name, or it is *silent*, which means it moves work along but leaves no record in the log. All nets in this reference give every arrow a weight of one. The formal statements are Definitions A.1 and A.2.

The state of a net, called its *marking*, is the number of *tokens* sitting in each place. A transition may fire when each of its input places holds at least one token. Firing takes one token from each input place and puts one token in each output place (Definition A.3).

Concurrency follows directly from this rule, since if tokens sit in two places that feed two different transitions, both can fire, in either order, and neither blocks the other. Synchronisation works the other way round, since a transition with two input places fires only once both hold tokens. Choice arises when one token enables two transitions and only one of them can take it.

Workflow nets and soundness

A *workflow net* (WF-net) is a Petri net with one start place and one end place, in which every node lies on some path from start to end (Definition A.4). An instance begins as a single token in the start place and should finish as a single token in the end place. Most Petri-net-based discovery algorithms return a workflow net, and its test of correctness is soundness.

Figure 1 shows a small workflow net for an order process. It contains a choice between rejecting, accepting and rechecking an order, a rework loop back to the check, and two steps that run in parallel before shipping.

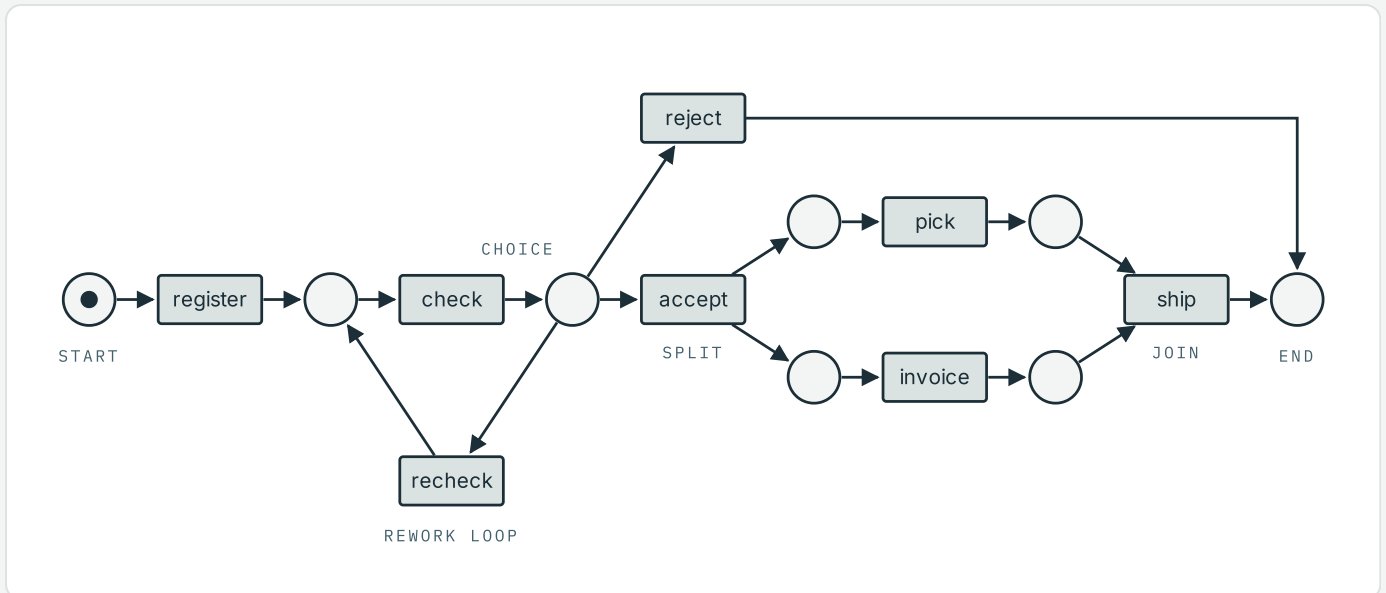


FIGURE 1. A workflow net for an order process. Circles are places, boxes are transitions, and the token in the start place is one process instance waiting to begin.

A workflow net is *sound* when every instance can always still finish, finishing leaves nothing behind, and every step in the model can actually happen (Definition A.5). Soundness is decidable, since it is equivalent to two properties of the net with an extra step joining end back to start. In that extended net, every step can always fire again later, and no place collects an unlimited number of tokens [6]. This is how soundness is checked in practice [7].

The incidence matrix and the state equation

Questions about a net’s structure can be answered with linear algebra over a single table of numbers, the *incidence matrix*, which has one row per place and one column per transition. Each entry records how many tokens a transition adds to a place, minus how many it removes (Definition A.6).

Each column therefore shows the net change in tokens caused by firing one transition. A transition that moves a token from one place to another has a column with a single -1 and a single $+1$. A transition that splits work into two parallel branches has -1 in its input place and $+1$ in each of two output places, so a program can find the parallel split by reading the numbers.

The *state equation* follows from these columns. Count how many times each transition fires in some run. The state at the end then equals the state at the start plus the matrix applied to those counts (Definition A.7).

A necessary condition only. Every reachable state satisfies the state equation, although some states that satisfy it cannot actually be reached. A set of non-negative whole-number firing counts can satisfy the equation even though no actual run fires the transitions in any workable order. States reached only on paper in this way are called *spurious* markings. Linear algebra therefore narrows the set of states worth checking, and each remaining candidate is confirmed by exploring the actual runs of the net.

Invariants, conservation and cycles

Two kinds of invariant come from the incidence matrix. A *place invariant* (P-invariant) is a weighting of the places whose weighted token total no firing can change (Definition A.8). A *transition invariant* (T-invariant) is a set of firing counts which, if the transitions can actually fire that many times from the current state, returns the net exactly to that state (Definition A.9).

Each has a business meaning, and Section 5 uses both for structural diagnosis. A P-invariant is a *conservation law*, which certifies that work neither vanishes nor duplicates, so instances do not leak out of the process. A T-invariant describes a set of firings that returns the net to where it started, which in most business processes means a loop,

often rework such as a document rejected and resubmitted or a check that fails and is repeated. Computing T-invariants therefore lists the candidate rework loops of a process as a linear-algebra problem.

Process trees

Block-structured discovery first builds a process tree and only then turns it into a net. A *process tree* has activities (or a silent step) at its leaves. Each inner node combines its children in one of four ways, namely in sequence, as an exclusive choice, in parallel, or as a loop with a body and redo parts (Definition A.10).

Every process tree translates to a sound WF-net, because each of the four ways of combining has a fixed net fragment and putting fragments together keeps the WF-net shape, so soundness follows by induction. An algorithm that returns a process tree therefore returns a sound model *by construction*, with no separate check needed.

The directly-follows graph

The *directly-follows graph* of an event log has one node per activity. It has an arrow from one activity to another whenever the second immediately follows the first somewhere in the log. It also records which activities start and which end a trace (Definition A.11).

The graph takes one pass over the log to build, and it is the input to every algorithm in the inductive and heuristic families. It records only which activity immediately followed which, so it loses dependencies between activities further apart in a trace, and Section 7 returns to this under incompleteness and representational bias.

Figure 2 follows a log of two traces through the whole pipeline, from the log to its directly-follows graph, a cut, a process tree and finally a Petri net.

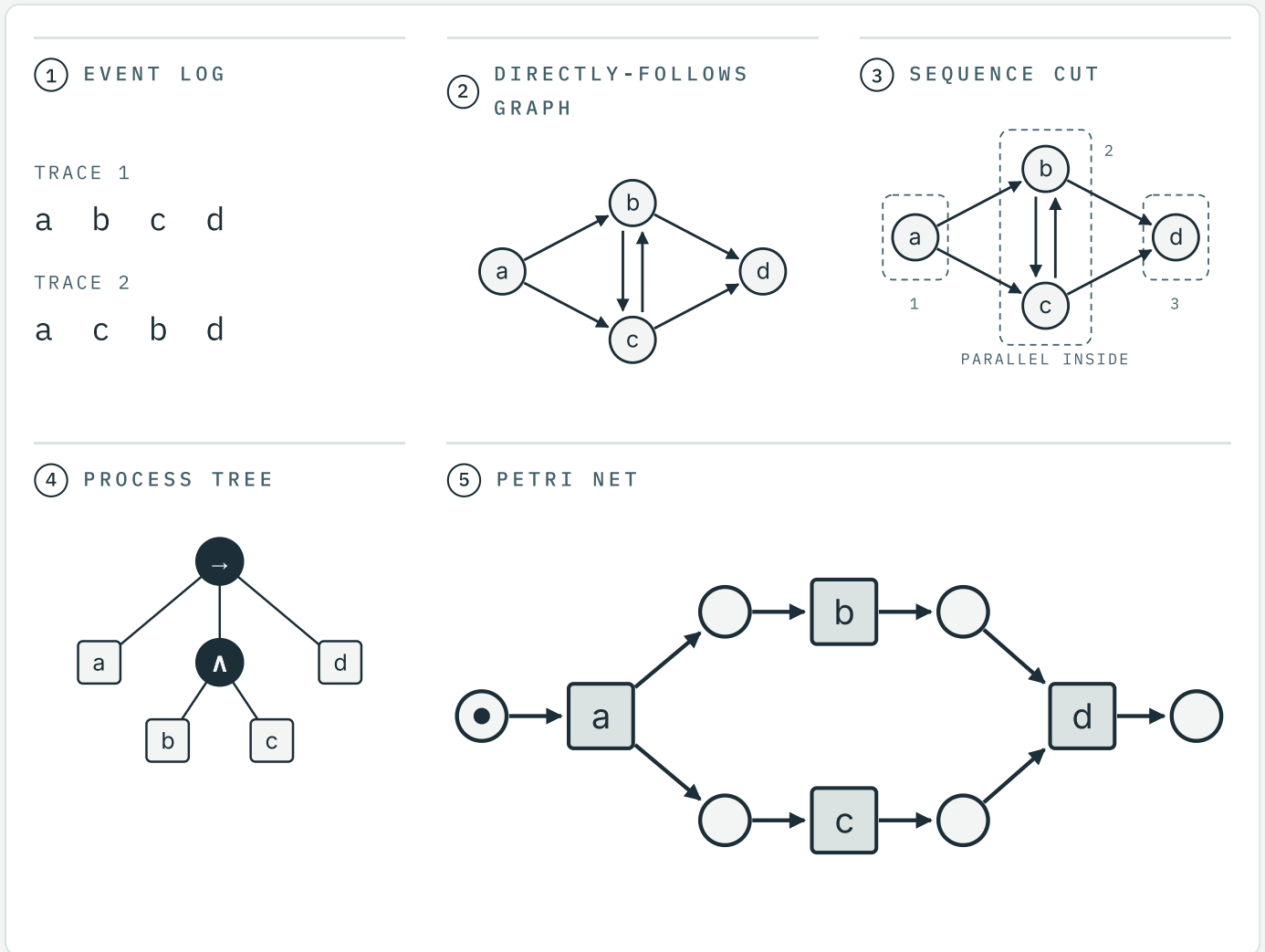


FIGURE 2. From a two-trace log to a Petri net. The graph has arrows both ways between b and c, a sequence cut splits it into three numbered groups, and a parallel cut inside group 2 gives the tree $\rightarrow (a, \wedge(b, c), d)$ and its net.

/ 3 THE CLASSES OF NET THAT DISCOVERY TARGETS

Petri nets form a family of classes, in which restricting a net’s shape makes it easier to analyse and extending it with data or time lets it describe more while making analysis harder. Discovery works with a small part of that family, and Table 1 lists the classes that the discovery algorithms in Section 4 rely on.

TABLE 1. The net classes used in process discovery.

CLASS	ALLOWS	GIVES UP	ROLE IN DISCOVERY
State machine	choice	concurrency	sequential processes
Marked graph	concurrency	choice	fixed pipelines
Free-choice	choice and concurrency	tangled choices	common discovery target
Workflow net	one start, one end	several start or end places	output of most tools
Block-structured	nested blocks	some sound nets	inductive miner output

Restrictions on shape. In an ordinary net all tokens look the same, and three subclasses at this level are defined by shape alone. A net is a *state machine* when every transition has exactly one input and one output place, $|\bullet t| = |t^\bullet| = 1$. Such nets express choice but never concurrency, so an instance sits in exactly one place at a time. A net is a *marked graph* when every place has exactly one input and one output transition, $|\bullet p| = |p^\bullet| = 1$, which expresses concurrency but never choice. A net is *free-choice* when any two transitions that compete for a token always compete for exactly the same tokens, so a choice is never tangled up with waiting for other branches

(Definition A.12). Free-choice nets allow both choice and concurrency, and important questions about them, including whether a free-choice workflow net is sound, can be decided in polynomial time [8], [6]. For that reason they, and the block-structured nets inside them, are the practical target of process discovery. Murata [5] calls the condition used here *extended* free-choice, and texts differ on the name.

Workflow nets. Workflow nets are the class built for business processes. They have one source, one sink, and every node on a path between the two. Soundness (Definition A.5) is the test of correctness.

Block-structured nets. A block-structured net is the translation of a process tree, so every choice, parallel section and loop opens and closes as one nested block, and such nets are sound by construction. Some sound nets cannot be written this way, for example ones with dependencies that cross block boundaries.

Richer classes attach data, time or several object types to tokens. The discovery methods in this reference work on control flow alone, so those classes are left aside.

/ 4 PROCESS DISCOVERY ALGORITHMS

Discovery takes a log $L \in \mathcal{B}(\Lambda^*)$ and returns a model. Algorithms differ in what they guarantee, in what they assume about noise, and in what they do when the log is ambiguous. This section takes each family in turn, starting with the Alpha miner, and notes for each the weakness in earlier methods that it addresses.

The Alpha miner

The first discovery algorithm [9] reads four relations between pairs of activities from the directly-follows graph (DFG) $G(L)$, which records which activity was seen immediately after which. The four relations are direct succession, causality, parallelism and independence, and the algorithm builds a net from them. It is provably correct on a restricted class, but in practice it is fragile in three ways. A single deviant trace can change an inferred relation and with it the whole structure. Loops of length one and two are mishandled, and the output need not be sound. Today the algorithm is used mainly for teaching and is rarely applied to production logs.

The heuristic miner

Weijters and van der Aalst [10] weighted the directly-follows relation by *frequency*, so that discovery could cope with the noise found in real logs. An edge is trusted only when its relative frequency exceeds a threshold, so rare behaviour is treated as noise and dropped. This gives stable results on real logs and handles loops well. It gives up formal guarantees in exchange, since the output is typically a causal net with no promise of soundness.

The inductive family

The inductive miner [11] replaced local net construction with a global, recursive split of the log. It proceeds in five steps:

- 1 Compute $G(L)$ (Definition A.11).
- 2 Find a *cut*, meaning a partition $\Sigma_1, \dots, \Sigma_n$ of the activities that matches one of four operator patterns.
- 3 Split L into sublogs according to the cut type, and recurse on each.
- 4 Stop at the base condition of a single activity, or the empty trace, yielding a leaf or τ .
- 5 If no cut exists, apply a *fall-through*, ultimately the “flower” model that permits any ordering of the remaining activities. The original paper specifies the flower model in full.

Finding cuts is the core of the method. Each of the four cuts is a test on the arrows of $G(L)$ that run between the groups of activities, and each pattern of arrows points to one way of combining the groups. Figure 3 draws the four patterns on small graphs, each with the tree operator it produces.

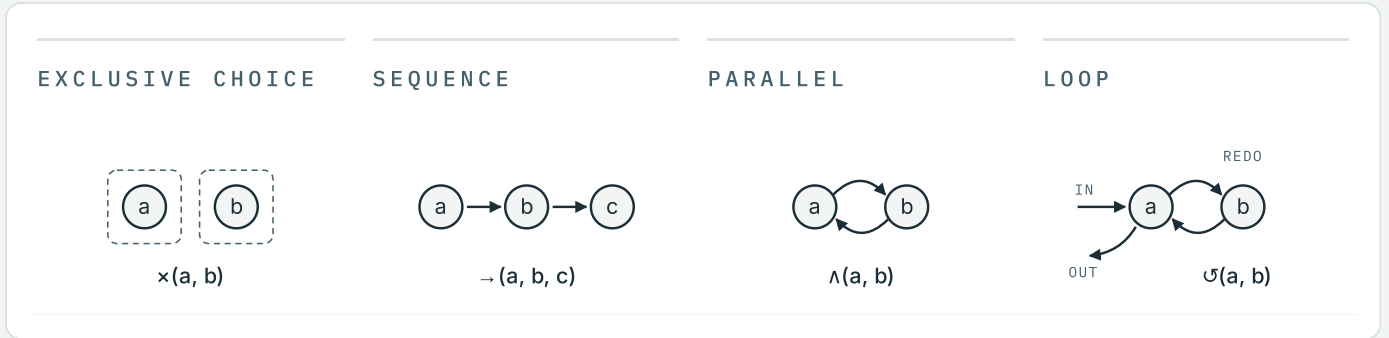


FIGURE 3. The four cut patterns on small directly-follows graphs, with the process-tree operator each one produces. In the loop, an instance enters and leaves through the body a , and b leads back to a .

An *exclusive-choice* cut is one where no arrow runs between any two groups (Definition A.13, Equation (11)). The branches never touch, so exactly one of them is taken in each instance.

A *sequence* cut is one where arrows lead only one way, from earlier groups to later ones (Definition A.14). The test applies to paths through the graph as a whole. Every activity in an earlier group can reach every activity in a later group by some chain of arrows, and nothing can get back. The condition is on paths, so a missing direct arrow does not break it, and a single arrow between groups does not by itself establish the cut. The original definition adds a further condition on how the end activities of one group connect to the start activities of the next [11], and Definition A.14 states the two path conditions only.

A *parallel* cut is one where every activity in one group has arrows both to and from every activity in every other group, because branches that run at the same time are seen interleaved in both orders (Definition A.15). Each group must also contain a start activity and an end activity, so that each branch can begin and finish on its own.

A *loop* cut is one where a main body holds every start and end activity, and the other groups are redo parts whose arrows lead back to the start of the body (Definition A.16). Arrows leave the body only from its end activities, and arrows from a redo part re-enter only at start activities. The redo parts do not connect to each other, and they all join the body in the same way.

Every run of the inductive miner returns a process tree, and every process tree translates into a sound workflow net, so the discovered model is sound without any separate check (Proposition A.17). None of the other classical families in this section (Alpha, heuristic, region-based and Split miner) offers this guarantee. Among newer methods, miners built on the Partially Ordered Workflow Language (POWL), described below, and some learned miners also guarantee soundness. Soundness by construction makes the inductive miner a natural starting point for a refinement loop, since soundness holds along the whole run whenever every state of the loop is a process tree.

Leemans et al. [12] added the frequency-filtering variant known as inductive miner infrequent (IMf), which removes infrequent edges from $G(L)$ before looking for cuts, and Leemans et al. [13] addressed incomplete logs.

Region-theoretic and ILP miners

A separate lineage computes the net with the smallest behaviour that still contains the log, using the theory of regions and integer linear programming (ILP) [14], [15], [16]. Fitness is high by construction, and three costs follow from that construction. First, in their basic form these miners tolerate no noise, because every observed trace must be permitted, so a single error becomes structure in the model. Second, they scale poorly, and third, they produce overfitted models. Schröder et al. [17] report ILP-based miners producing models “almost nine times larger” than inductive models on the Business Process Intelligence (BPI) Challenge 2020 event logs. The eST-Miner line [18], [19] refines this branch by adding discovery of long-term dependencies with guarantees, together with filtering of infrequent behaviour. Its successor eST² also takes partially ordered input, on the argument that the strict time order of timestamps misrepresents which events actually caused which.

Split miner

Augusto et al. [20] filter $G(L)$, detect concurrency explicitly, and build a Business Process Model and Notation (BPMN) model. The method achieves a strong balance of fitness and precision at high speed without insisting on soundness, and it remains a standard baseline.

Optimisation-based miners (2023 to 2026)

Recent work keeps the inductive recursion and replaces rule-of-thumb cut selection with explicit optimisation. On the contest scores its authors report (Figure 4), this line accounts for the rise from about 70 percent for IMf to 88.24 percent for OptIMlIst.

AIM. van Detten et al. [21] propose the Approximate Inductive Miner (AIM). It applies k -means clustering to estimated relations to group activities for each operator, which reduces an exponential cut search to a cost linear in the parameter space. It also folds the suggestion of filter parameters into the recursion itself. On 18 real-life logs it reports mean fitness of 78 percent, mean precision of 88 percent and mean model size of 51 nodes. It is Pareto-optimal on 17 of 18 logs, attains the highest precision on 11 of 18 and returns the smallest model on 16 of 18. Its built-in parameter suggestion completes in 30 minutes against 5 days for exhaustive hyperparameter search. On 480 synthetic Process Discovery Contest (PDC) logs it reports structure-detection accuracy of 86 percent. One reported limitation is aggressive filtering, since on some logs roughly half the activities are discarded [17].

OptIMlIst. Schröder et al. [1], extending Schröder et al. [17], introduce OptIMlIst (Optimization Inductive Miner Infrequent Incomplete eSTimation), which replaces cut selection with a family of integer linear programs, one base ILP per operator $\{\rightarrow, \times, \wedge, \circ\}$. Each program finds the split of activities into groups that comes closest to a valid cut. Closeness is measured as the total frequency of the observed arrows that break the operator's pattern, and the program minimises that total (Equation (12)). Each operator defines a broken arrow in its own terms. A backward arrow breaks $\rightarrow$, and any crossing arrow breaks $\times$. For $\wedge$ a *missing* reverse arrow counts as a break, and for $\circ$ a break is an arrow that ignores the body and redo structure. Filtered ILPs may also drop a single disruptive activity when a local fitness and precision estimate justifies the loss of information. This avoids the problems that fixed frequency thresholds cause when results do not change steadily as the threshold moves. The method is *parameter-free*, and it also detects τ -constructs for optional and loop behaviour. As a member of the inductive family, it returns a plain sound workflow net.

POWL and beyond process trees. Kourani et al. [22] extend POWL with choice graphs. These allow decisions that are not block-structured, and cyclic flows, while keeping soundness. Of the methods from 2023 to 2026 covered here, it goes furthest in relaxing the limits that process trees place on what a model can express, while still guaranteeing soundness.

Rule-guided discovery. The rule-based inductive miner (IMr) [23], [24] discards candidate cuts that break supplied declarative rules and selects the lowest-cost survivor, so that expert knowledge of how the process should run can constrain discovery. Norouzifar et al. [25] build on it, using large language models to turn domain knowledge into such rules.

Learned discovery. Sommers et al. [26] train graph neural networks (GNNs) to perform discovery, so the algorithm is learned from example logs and their models. The network decides which candidate places survive and passes behavioural information across the graph. Trained on 2,000 block-structured models, it produces sound nets of arbitrary structure, free of the block-structure restriction, at accuracy and simplicity reported as comparable to the state of the art.

Where the frontier sits

Table 2 gives the results of the Process Discovery Contest 2024 [27]. It is the most recent objective comparison among many participants, scored on 288 synthetic logs with known ground truth (Section 6 gives the scoring rule).

TABLE 2. Process Discovery Contest 2024 results on 288 training logs, in percent [27].

SUBMISSION	AUTHOR (S)	SCORE
FuzzyLogAbstraction (Pyrrhus)	Christfort, Xu, Slaats	90.86
UiPath_PDC2024_v1_config06	Brons	89.46
DisCoveR (BitDisCoveRJS)	Christfort, Slaats	79.89
DisCoveRJS-Filtering-Composite	Christfort, Slaats	79.77
DisCoveRJS-Filtering-Violation	Christfort, Slaats	78.83
2023.2.a.DisCoveRJS	Christfort, Slaats	75.12
FuzzyDisCover-FuzzyClas	Christfort, Slaats	69.64
BitDisCoveRJS-Filtering	Christfort, Slaats	69.62
BonitaMiner	Eldin et al.	36.80

The winning entry, Pyrrhus, recorded a positive accuracy of 90.94 percent and a negative accuracy of 90.80 percent. It comes from the authors of the DisCoveR entries, whose miner discovers declarative models in the form of Dynamic Condition Response (DCR) graphs [28], and the runner-up is an industrial entry from UiPath. The PDC 2025 call for participation cites 91.8 percent for the same submission, which is its score on the 2025 data [29]. Among openly published Petri-net algorithms, Schröder et al. [1] report OptIMlIst at 88.24 percent on the same dataset, against roughly 83 percent for AIM and roughly 70 percent for the original inductive miner and IMf. By its authors' figures, OptIMlIst therefore scores about 18 percentage points above the 2013 inductive miner. The PDC score is classification accuracy, so it does not say whether the gain comes from fitness or precision. Figure 4 places the open algorithms and the contest entries on one scale, so it compares Petri-net miners with declarative miners.

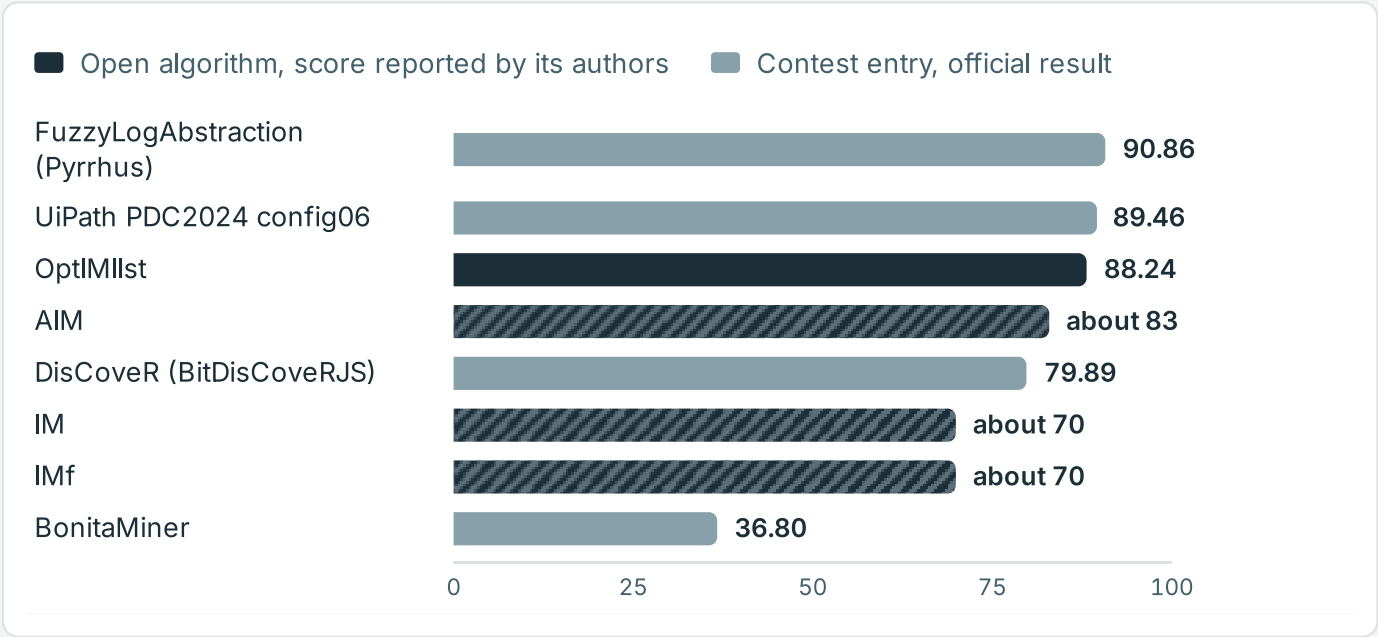


FIGURE 4. Process Discovery Contest 2024 scores on 288 synthetic logs, in percent. Contest entries are official results, open algorithms are scores reported by the OptIMlIst authors, and striped bars are approximate values read from their comparison.

/ 5 CONFORMANCE CHECKING AND STRUCTURAL DIAGNOSIS

A discovered net is useful only if one can measure how well it describes what really happens and ask what its structure implies. Diagnosis has three layers, namely comparison against the log, scoring, and questions put to the structure itself. Comparison and scoring supply an objective that a refinement method can improve, and the structural questions supply its constraints. This section takes each of the three layers in turn.

Alignments

An alignment lines up one recorded trace, step by step, against one complete run of the model. Where the two agree, the pair is a synchronous move. A log move marks a step the log took that the model does not allow, and a model move marks a step the model requires that the log left out (Definition A.19).

Each move carries a penalty, and by convention a synchronous move costs 0, while log moves and visible model moves cost 1. A silent model move, a step the model takes without any visible activity, usually costs 0 or a very small amount, and since the choice affects every fitness figure built on the alignment, a report should state which was used. An *optimal* alignment is the one with the lowest total penalty, found by shortest-path search (typically A*) over a combined model of log and net [30], [31].

Each log move and model move is tied to a position in a trace, so an alignment shows where the model and the log disagree as well as how often. Automated repair and refinement methods use that position to decide which part of the model to change.

The four quality measures

Fitness asks whether the model can replay what happened. It scores the total alignment penalty of the log against the worst case, so a model that replays every trace exactly scores 1 (Definition A.20).

Precision covers the opposite failure, a model that allows behaviour the log never shows. At each point a trace reaches, it compares the steps the model would permit with the steps the log actually took there. Permitted steps that the log never took are called escaping edges, and the more of them there are, the lower the precision (Definition A.21). Every trace counts, including repeats, so frequently visited points weigh more and this weighting is already part of the measure [32], [33]. Entropy-based alternatives with monotonicity guarantees, meaning the score moves in the expected direction as a model is made more or less permissive, are given by Polyvyanny et al. [34].

A model *generalises* well when it still fits traces it has not seen, and it is *simple* when a person can read it. Neither measure has a standard formula, and model size is the usual structural stand-in for simplicity. Following Buijs et al. [35], [36], all four measures must be reported together. Fitness and precision are usually combined into one number, their harmonic mean, which is the standard F_1 statistic (Definition A.22).

Two meanings of F-score. The F-score of fitness and precision (Definition A.22) is a different quantity from the “F-score” of the Process Discovery Contest. The contest score is the harmonic mean of how accurately a model sorts test traces into positive and negative classes (Section 6). Both are called F-score in the literature, so a careful report states which of the two it means.

Structural diagnosis

The algebra of Section 2 turns five questions that a consultant would otherwise answer by inspection into computations, listed below.

- **Reachability bounds.** The state equation (7) gives a necessary condition for reaching a state by solving an integer linear system. As the note on spurious solutions in Section 2 explains, a positive answer must still be confirmed.
- **Leaks and duplication.** P-invariants (8) are conservation laws. A P-invariant certifies that the weighted token count over its places never changes. If such invariants cover every place, no work can vanish or be duplicated anywhere.
- **Rework.** T-invariants (9) list the sets of firings which, if they can actually be scheduled, bring the net back to its starting marking. Finding candidate rework loops therefore reduces to computing the minimal non-negative T-invariants, a task standard invariant algorithms handle.
- **Dead ends.** Violations of (3) identify reachable states from which an instance can never finish, which in business terms is a case the model would leave stuck.

- **Bottlenecks.** With timestamps attached, waiting time builds up on the places between transitions, so the spread of tokens over places shows where delay occurs.

All of these checks run without human input, yet in the usual workflow an analyst reads the results and decides whether to mine again.

/ 6 BENCHMARKS AND EVALUATION

Evaluation rests on three sources of evidence:

- synthetic logs with a known true model,
- public logs from real operations,
- comparative benchmarks.

This section takes each in turn.

Ground truth from the Process Discovery Contest

Organised by Eric Verbeek under the IEEE Task Force on Process Mining, the PDC is the closest the field comes to an objective scoreboard, because the true model is known and kept hidden from entrants. van Dongen and Verbeek [37] review it alongside the BPI Challenges, and each edition's logs are archived as a citable dataset (for example [38] for the 2020 edition). Scoring takes three steps for each training log. The first step discovers a model from the training log. The second uses that model to judge, for each trace of a held-out test log, whether it fits the model better than the matching trace of a *base* log. The third computes

$$F = \frac{2 \cdot P \cdot N}{P + N}, \quad (1)$$

with P positive accuracy and N negative accuracy, averaged over training logs. The test models and logs vary along seven dimensions:

- long-term dependencies,
- loops (none, simple or complex),
- OR constructs,
- routing constructs,
- optional tasks,
- duplicate tasks,
- log quality.

Editions with published datasets are 2016, 2017, and 2019 to 2025, and there was no 2018 edition. At the time of writing, the 2026 edition had not been announced. PDC 2023 used 384 training logs and 96 test and base pairs. PDC 2024 and 2025 used 288 training logs and 96 pairs, at 1,000 traces per log, with a 10-minute compute limit per log in 2025. The PDC 2024 dataset is archived at 4TU.ResearchData [39]. It holds 288 training, 96 test, 96 base and 96 ground-truth logs in eXtensible Event Stream (XES) format, plus 96 workflow nets in Petri Net Markup Language (PNML).

Two caveats apply to any paper that uses PDC scores. First, the logs are *synthetic*, so they lack the heavy-tailed spread of variants, the hundreds of labels and the unfinished instances of real data. Second, the metric (1) rewards *classification* accuracy alone and says nothing about whether the model is readable, so a model can score well and remain unusable.

Real logs

The 4TU.ResearchData repository is the standard home of the public real-life logs. They include the annual BPI Challenge logs [40], the Road Traffic Fine Management Process [41], Sepsis Cases [42] and the receipt-phase log of an environmental permit application process, known as WABO [43]. These logs have the properties synthetic

data lacks, although none of them comes with a known true model. Fitness and precision can still be measured, but *correctness* cannot be assessed.

The benchmark gap

Augusto et al. [2] evaluated discovery methods across 24 real-life logs (12 public, 12 proprietary) on 9 quality metrics, and it remains the largest published comparison of its kind. Because it dates from 2019, it covers none of the newer methods AIM, OptIMist, eST²-Miner, POWL 2.0 or the GNN miner. We searched the literature to September 2026 and found no comparison of that scale, with many discovery algorithms run on many logs and a citable results table. The substitutes are the PDC results tables and the single-paper comparisons in Schröder et al. [1] and van Detten et al. [21].

This gap affects every method in this reference. Claims that “method X is state of the art” currently rest either on a synthetic contest or on comparisons the authors ran themselves. This reference argues that a modern successor to Augusto et al. [2] is among the most useful contributions open to the field, because it would let claims about new algorithms be checked on shared real-life logs.

/ 7 OPEN PROBLEMS

This section takes each open problem in turn, from noise in the log to representational bias.

Noise and infrequent behaviour. Real logs contain rare, malformed and exceptional traces. Keeping these traces makes the model larger and less precise, while filtering them out can remove behaviour that is rare but genuine. Every algorithm sets this threshold somewhere, and none yet sets it reliably across different logs.

Incompleteness. A log holds only the behaviour that occurred during the period it covers, so discovery must generalise beyond the observed traces without inventing behaviour. Guarantees such as the rediscoverability theorem of Leemans et al. [11] hold only if the log is directly-follows complete, meaning every pair of activities that can follow each other does so somewhere in the log. That precondition is rarely checked in practice. When it does not hold, what looks like a failure of the miner can be a sampling problem that more traces would remove.

The trade-off between accuracy and generality. A model that replays as much of the log as possible tends to permit much else as well. A model limited to exactly what was seen rejects valid variants. Such a model scores 1 on both fitness and precision, although it generalises poorly and is hard to read. In practice the trade-off therefore runs between fitness and precision on one side and generalisation and simplicity on the other. We found no current algorithm that presents this choice to the analyst in terms they can interpret.

Complexity and overfitting. High-fitness methods can return tangled “spaghetti” nets that are accurate and unusable. Most process models are read by a person who has to act on them, so a model that scores well and cannot be read is of little practical use. Since simplicity has no standard measure (Section 5), the problem is also hard to state formally.

Scalability, and where the cost actually is. Industrial logs can reach millions of events and hundreds of activities. Most of the computing time goes into evaluating models, since an optimal alignment is found by a search whose cost can grow exponentially with the size of the trace and the model [31]. A method that scores many candidate models, such as a refinement loop, therefore spends most of its time on scoring, and a faster scoring step shortens the whole run more than a faster miner would.

Concept drift. Processes change, so a model discovered from last year’s log may misrepresent this year’s process, yet discovery treats the log as fixed. Sato et al. [44] survey how drift is detected and handled and name the obstacles that persist. The field “still primarily focuses on offline analysis”, and assessment is hampered by “the lack of common evaluation protocol, datasets, and metrics”. Recent work applies machine learning to drift detection [45]. Drift is also the practical argument for treating a discovered model as something to maintain, rechecked against new data long after delivery.

Representational bias. Block-structured discovery can return only models that a process tree can express, so long-term dependencies that span several levels of nesting are systematically left out. Norouzifar et al. [25] report

this limitation explicitly for IMr, and POWL 2.0 [22] and the eST-Miner line [18] attack it from opposite directions. Any refinement loop confined to process trees inherits this bias.

/ 8 CONCLUSION

Process discovery turns the event logs that organisations already keep into models of how their work runs, and Petri nets make those models both readable and open to analysis. The inductive family made soundness a property of the algorithm, and the optimisation-based miners of 2023 to 2026 have raised its accuracy on synthetic benchmarks. How these methods compare on real logs is still open, since the last broad comparison dates from 2019, and Section 7 sets out the problems that remain.

This list collects the formal definitions and results the reference refers to.

Petri nets, workflow nets and their algebra

DEFINITION A.1 (Preset, postset) For a net with flow relation F and $x \in P \cup T$, $\bullet x = \{y \mid (y, x) \in F\}$ and $x^\bullet = \{y \mid (x, y) \in F\}$.

DEFINITION A.2 (Labelled Petri net) A labelled Petri net is a tuple $N = (P, T, F, W, \Lambda, \ell)$ where P is a finite set of *places*, T a finite set of *transitions* with $P \cap T = \emptyset$, $F \subseteq (P \times T) \cup (T \times P)$ the *flow relation*, $W : F \rightarrow \mathbb{N}^+$ an arc-weight function, Λ a set of activity labels, and $\ell : T \rightarrow \Lambda \cup \{\tau\}$ a labelling function. Here $W(x, y) = 0$ for $(x, y) \notin F$, and all nets in this reference are ordinary, i.e. $W \equiv 1$, so the firing rule below is stated in its unit-weight form. A transition t with $\ell(t) = \tau$ is *silent*, in that it participates in the dynamics but emits no observable event.

DEFINITION A.3 (Marking, enabling, firing) A *marking* is a multiset $M \in \mathcal{B}(P)$, where $M(p)$ is the number of tokens in place p . Transition t is *enabled* in M , written $M[t]$, iff $\forall p \in \bullet t : M(p) \geq 1$. Firing t yields the marking

$$M' = (M \setminus \bullet t) \uplus t^\bullet, \quad (2)$$

written $M[t]M'$. Here $R(N, M)$ denotes the set of markings reachable from M by any firing sequence, and $\text{lang}(N)$ the set of observable label sequences of complete executions.

DEFINITION A.4 (Workflow net) A labelled Petri net N is a *workflow net* (WF-net) iff it has a unique source place i with $\bullet i = \emptyset$, a unique sink place o with $o^\bullet = \emptyset$, and every node lies on a path from i to o [46].

DEFINITION A.5 (Soundness) A WF-net N with initial marking $[i]$ and final marking $[o]$ is *sound* iff

$$\text{(option to complete)} \quad \forall M \in R(N, [i]) : [o] \in R(N, M), \quad (3)$$

$$\text{(proper completion)} \quad \forall M \in R(N, [i]) : M \geq [o] \Rightarrow M = [o], \quad (4)$$

$$\text{(no dead transitions)} \quad \forall t \in T \exists M \in R(N, [i]) : M[t]. \quad (5)$$

DEFINITION A.6 (Incidence matrix) For a net with $|P| = m$ places and $|T| = n$ transitions, the incidence matrix $C \in \mathbb{Z}^{m \times n}$ has entries

$$C[p, t] = \underbrace{W(t, p)}_{\text{tokens produced in } p} - \underbrace{W(p, t)}_{\text{tokens consumed from } p}, \quad (6)$$

where W gives arc weights (and 0 for absent arcs).

DEFINITION A.7 (State equation) Let $x \in \mathbb{N}^n$ be a *firing vector* (Parikh vector), $x[t]$ counting firings of t . If M' is reachable from M by a firing sequence with Parikh vector x , then

$$M' = M + Cx. \quad (7)$$

DEFINITION A.8 (P-invariant) A *place invariant* (P-invariant, S-invariant) is a vector $y \in \mathbb{N}^m$, $y \neq \mathbf{0}$, with

$$y^\top C = \mathbf{0}^\top, \quad \text{whence} \quad y^\top M' = y^\top M \quad \forall M' \in R(N, M). \quad (8)$$

DEFINITION A.9 (T-invariant) A *transition invariant* (T-invariant) is a vector $x \in \mathbb{N}^n$, $x \neq \mathbf{0}$, with

$$Cx = \mathbf{0}. \quad (9)$$

DEFINITION A.10 (Process tree) A process tree over an alphabet Λ is defined inductively. Any $a \in \Lambda$ is a tree (a leaf), τ is a tree (the silent leaf), and if $T_1, \dots, T_k$ are trees and $\oplus \in \{\rightarrow, \times, \wedge, \odot\}$ then $\oplus(T_1, \dots, T_k)$ is a tree. The operators denote sequence ($\rightarrow$), exclusive choice ($\times$), interleaved parallelism ($\wedge$) and loop ($\odot$, first child the body, remaining children the redo parts).

DEFINITION A.11 (Directly-follows graph) Let $L \in \mathcal{B}(\Lambda^*)$ be an event log. Its directly-follows graph $G(L) = (\Lambda_L, \rightarrow_L, S_L, E_L)$ has a node per activity occurring in L , an edge $a \rightarrow_L b$ iff some trace of L contains $\dots a b \dots$, start activities $S_L =$ first elements of traces, and end activities $E_L =$ last elements. Here $a \rightsquigarrow b$ stands for the existence of a directed path from a to b in $G(L)$.

Petri net classes and discovery algorithms

DEFINITION A.12 (Free-choice net) A net is *free-choice* iff

$$\forall t_1, t_2 \in T : \bullet t_1 \cap \bullet t_2 \neq \emptyset \implies \bullet t_1 = \bullet t_2, \quad (10)$$

i.e. whenever two transitions compete for a token they compete for exactly the same tokens, so the choice between them is never entangled with synchronisation.

DEFINITION A.13 (Exclusive-choice cut) $\Sigma_1 \dots \Sigma_n$ is an exclusive-choice cut iff

$$\forall i \neq j, \forall a_i \in \Sigma_i, a_j \in \Sigma_j : (a_i, a_j) \notin \rightarrow_L. \quad (11)$$

DEFINITION A.14 (Sequence cut) $\Sigma_1 \dots \Sigma_n$, ordered, is a sequence cut iff for all $i < j$ every $a_i \in \Sigma_i, a_j \in \Sigma_j$ satisfy $a_i \rightsquigarrow a_j$ and $a_j \not\rightsquigarrow a_i$.

DEFINITION A.15 (Parallel cut) $\Sigma_1 \dots \Sigma_n$ is a parallel cut iff every part contains a start activity and an end activity, and for all $i \neq j$ and all $a_i \in \Sigma_i, a_j \in \Sigma_j$ both $(a_i, a_j) \in \rightarrow_L$ and $(a_j, a_i) \in \rightarrow_L$.

DEFINITION A.16 (Loop cut) Σ_1 (the body) together with $\Sigma_2, \dots, \Sigma_n$ (the redo parts) is a loop cut iff all start and end activities lie in Σ_1 and no edges connect distinct redo parts. Every edge from a redo part into the body must end at a start activity, and every edge from the body into a redo part must leave from an end activity. The connectivity of those entry and exit edges must be uniform across the redo parts.

PROPOSITION A.17 (Soundness by construction, Leemans et al. [11]) Every run of the inductive miner returns a process tree, and every process tree translates to a sound workflow net. Hence the discovered model is sound without any verification step.

DEFINITION A.18 (OptIMILst cut objective) For a candidate operator the ILP assigns each activity to a part so as to minimise the total weight of edges violating that operator's pattern,

$$\min_{\text{assignment}} \sum_{(a,b) \in \rightarrow_L} w(a,b) \cdot \mathbf{1}[(a,b) \text{ violates the operator's edge pattern}], \quad (12)$$

where $w(a,b)$ is the observed edge frequency.

Conformance checking

DEFINITION A.19 (Moves and alignments) Let $\sigma \in \Lambda^*$ be a trace and N a model. An *alignment* of σ with N is a sequence of pairs (x, y) with $x \in \Lambda \cup \{\perp\}$ and $y \in T \cup \{\perp\}$ such that projecting the first components (dropping $\perp$) yields σ and projecting the second yields a complete firing sequence of N . A pair is a *synchronous move* if $x = \ell(y)$. It is a *log move* if $y = \perp$, meaning the log did something the model disallows, and a *model move* if $x = \perp$, meaning the model requires a step the log omits.

DEFINITION A.20 (Alignment-based fitness, after Carmona et al. [31])

$$fitness(L, N) = 1 - \frac{\sum_{\sigma \in L} cost(\sigma, N)}{\sum_{\sigma \in L} cost(\sigma, \langle \rangle) + |L| \times \min_{x \in N} cost(\langle \rangle, x)}, \quad (13)$$

where $cost(\sigma, \langle \rangle)$ aligns σ against the empty execution (all log moves) and $\min_{x \in N} cost(\langle \rangle, x)$ is the cheapest complete model execution aligned against the empty trace (all model moves).

DEFINITION A.21 (Escaping-edges precision) Let $AT(s)$ be the transitions the model allows at prefix-automaton state s and $RT(s)$ those the log actually took there, with $EE(s) = AT(s) \setminus RT(s)$ the *escaping edges*. Then

$$precision(L, N) = 1 - \frac{\sum_{i=1}^{|L|} \sum_{j=1}^{|\sigma_i|+1} |EE(s_{ij})|}{\sum_{i=1}^{|L|} \sum_{j=1}^{|\sigma_i|+1} |AT(s_{ij})|}. \quad (14)$$

DEFINITION A.22 (F-score of fitness and precision)

$$F_1 = \frac{2 \cdot fitness \cdot precision}{fitness + precision}. \quad (15)$$

/ ACRONYMS

AIM	Approximate Inductive Miner
BPI	Business Process Intelligence
BPMN	Business Process Model and Notation
DCR	Dynamic Condition Response (graph)
DFG	Directly-follows graph
GNN	Graph neural network
ILP	Integer linear programming
IM	Inductive miner
IM _f	Inductive miner, infrequent behaviour variant
IM _r	Inductive miner with rules
PDC	Process Discovery Contest
PNML	Petri Net Markup Language
POWL	Partially Ordered Workflow Language
WF-net	Workflow net
XES	eXtensible Event Stream

/ REFERENCES

- [1] C. Schröder, J. N. van Detten, and S. J. J. Leemans, "Locally optimized inductive process discovery," *Process Sci.*, vol. 3, no. 1, Art. no. 18, 2026, doi: 10.1007/s44311-026-00051-x.
- [2] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo, "Automated discovery of process models from event logs: Review and benchmark," *IEEE Trans. Knowl. Data Eng.*, vol. 31, no. 4, pp. 686-705, 2019, doi: 10.1109/TKDE.2018.2841877.
- [3] W. M. P. van der Aalst, *Process Mining: Data Science in Action*, 2nd ed. Berlin, Germany: Springer, 2016, doi: 10.1007/978-3-662-49851-4.
- [4] C. A. Petri, "Kommunikation mit Automaten," Ph.D. dissertation, Rheinisch-Westfälisches Institut für Instrumentelle Mathematik, Univ. Bonn, Bonn, Germany, 1962. English translation: "Communication with automata," Griffiss Air Force Base, NY, USA, Tech. Rep. RADC-TR-65-377, vol. 1, 1966.
- [5] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. IEEE*, vol. 77, no. 4, pp. 541-580, 1989, doi: 10.1109/5.24143.
- [6] W. M. P. van der Aalst, K. M. van Hee, A. H. M. ter Hofstede, N. Sidorova, H. M. W. Verbeek, M. Voorhoeve, and M. T. Wynn, "Soundness of workflow nets: Classification, decidability, and analysis," *Formal Aspects Comput.*, vol. 23, no. 3, pp. 333-363, 2011, doi: 10.1007/s00165-010-0161-4.
- [7] H. M. W. Verbeek, T. Basten, and W. M. P. van der Aalst, "Diagnosing workflow processes using Woflan," *Comput. J.*, vol. 44, no. 4, pp. 246-279, 2001, doi: 10.1093/comjnl/44.4.246.
- [8] J. Desel and J. Esparza, *Free Choice Petri Nets*, Cambridge Tracts in Theoretical Computer Science, vol. 40. Cambridge, U.K.: Cambridge Univ. Press, 1995, doi: 10.1017/CBO9780511526558.
- [9] W. M. P. van der Aalst, A. J. M. M. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 9, pp. 1128-1142, 2004, doi: 10.1109/TKDE.2004.47.
- [10] A. J. M. M. Weijters and W. M. P. van der Aalst, "Rediscovering workflow models from event-based data using little thumb," *Integr. Comput.-Aided Eng.*, vol. 10, no. 2, pp. 151-162, 2003, doi: 10.3233/ICA-2003-10205.
- [11] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from event logs: A constructive approach," in *Application and Theory of Petri Nets and Concurrency (PETRI NETS 2013)*, Lecture Notes in Computer Science, vol. 7927. Berlin, Germany: Springer, 2013, pp. 311-329, doi: 10.1007/978-3-642-38697-8_17.
- [12] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from event logs containing infrequent behaviour," in *Business Process Management Workshops (BPM 2013)*, Lecture Notes in Business Information Processing, vol. 171. Cham, Switzerland: Springer, 2014, pp. 66-78, doi: 10.1007/978-3-319-06257-0_6.
- [13] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst, "Discovering block-structured process models from incomplete event logs," in *Application and Theory of Petri Nets and Concurrency (PETRI NETS 2014)*, Lecture Notes in Computer Science, vol. 8489. Cham, Switzerland: Springer, 2014, pp. 91-110, doi: 10.1007/978-3-319-07734-5_6.
- [14] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik, "Process discovery using integer linear programming," in *Applications and Theory of Petri Nets (PETRI NETS 2008)*, Lecture Notes in Computer Science, vol. 5062. Berlin, Germany: Springer, 2008, pp. 368-387, doi: 10.1007/978-3-540-68746-7_24.
- [15] J. M. E. M. van der Werf, B. F. van Dongen, C. A. J. Hurkens, and A. Serebrenik, "Process discovery using integer linear programming," *Fundam. Inform.*, vol. 94, no. 3-4, pp. 387-412, 2009, doi: 10.3233/FI-2009-136.
- [16] R. Bergenthum, J. Desel, R. Lorenz, and S. Mauser, "Process mining based on regions of languages," in *Business Process Management (BPM 2007)*, Lecture Notes in Computer Science, vol. 4714. Berlin, Germany: Springer, 2007, pp. 375-383, doi: 10.1007/978-3-540-75183-0_27.
- [17] C. Schröder, J. N. van Detten, and S. J. J. Leemans, "Locally optimized process tree discovery," in *Process Mining Workshops (ICPM 2024)*, Lecture Notes in Business Information Processing, vol. 533. Cham, Switzerland: Springer, 2025, pp. 389-401, doi: 10.1007/978-3-031-82225-4_29.
- [18] L. L. Mannel, W. M. P. van der Aalst, L. Bernardinello, J. Kleijn, and L. Petrucci, "Discovering process models with long-term dependencies while providing guarantees and filtering infrequent behavior patterns," *Fundam. Inform.*, 2024, doi: 10.3233/FI-242168.
- [19] S. Folz-Weinstein, C. Rennert, L. L. Mannel, R. Bergenthum, and W. M. P. van der Aalst, "eST² miner: Process discovery based on firing partial orders," in *Advanced Information Systems Engineering (CAiSE 2025)*, Lecture Notes in Computer Science, vol. 15702. Cham, Switzerland: Springer, 2025, pp. 59-75, doi: 10.1007/978-3-031-94571-7_4.
- [20] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, and A. Polyvyanyy, "Split miner: Automated discovery of accurate and simple business process models from event logs," *Knowl. Inf. Syst.*, vol. 59, no. 2, pp. 251-284, 2019, doi: 10.1007/s10115-018-1214-x.
- [21] J. N. van Detten, P. Schumacher, and S. J. J. Leemans, "An approximate inductive miner," in *Proc. 5th Int. Conf. Process Mining (ICPM)*, 2023, pp. 129-136, doi: 10.1109/ICPM60904.2023.10271971.
- [22] H. Kourani, G. Park, and W. M. P. van der Aalst, "A discovery technique for expressive yet sound process models," *Process Sci.*, vol. 3, Art. no. 14, 2026, doi: 10.1007/s44311-026-00046-8.
- [23] A. Norouzifar, M. Dees, and W. M. P. van der Aalst, "Imposing rules in process discovery: An inductive mining approach," in *Research Challenges in Information Science (RCIS 2024)*, Lecture Notes in Business Information Processing, vol. 513. Cham, Switzerland: Springer, 2024, pp. 220-236, doi: 10.1007/978-3-031-59465-6_14.
- [24] A. Norouzifar, M. Dees, and W. M. P. van der Aalst, "Rule-guided process discovery," *Data Knowl. Eng.*, vol. 161, Art. no. 102508, 2026, doi: 10.1016/j.datak.2025.102508.

- [25] A. Norouzifar, H. Kourani, M. Dees, and W. M. P. van der Aalst, "Integrating domain knowledge into process discovery using large language models," 2025, *arXiv:2510.07161*. [Online]. Available: <https://arxiv.org/abs/2510.07161>
- [26] D. Sommers, V. Menkovski, and D. Fahland, "Supervised learning of process discovery techniques using graph neural networks," *Inf. Syst.*, vol. 115, Art. no. 102209, 2023, doi: 10.1016/j.is.2023.102209.
- [27] International Conference on Process Mining (ICPM), "Process Discovery Contest 2024: Results," 2024. [Online]. Available: <https://icpmconference.org/2024/process-discovery-contest>
- [28] C. O. Back, T. Slaats, T. T. Hildebrandt, and M. Marquard, "DisCover: Accurate and efficient discovery of declarative process models," *Int. J. Softw. Tools Technol. Transf.*, 2021, doi: 10.1007/s10009-021-00616-0.
- [29] International Conference on Process Mining (ICPM), "Call for Process Discovery Contest 2025," 2025. [Online]. Available: <https://icpmconference.org/2025/process-discovery-contest/>
- [30] A. Adriansyah, B. F. van Dongen, and W. M. P. van der Aalst, "Conformance checking using cost-based fitness analysis," in *Proc. IEEE 15th Int. Enterprise Distributed Object Computing Conf. (EDOC)*, 2011, pp. 55-64, doi: 10.1109/EDOC.2011.12.
- [31] J. Carmona, B. van Dongen, A. Solti, and M. Weidlich, *Conformance Checking: Relating Processes and Models*. Cham, Switzerland: Springer, 2018, doi: 10.1007/978-3-319-99414-7.
- [32] J. Muñoz-Gama and J. Carmona, "A fresh look at precision in process conformance," in *Business Process Management (BPM 2010)*, Lecture Notes in Computer Science, vol. 6336. Berlin, Germany: Springer, 2010, pp. 211-226, doi: 10.1007/978-3-642-15618-2_16.
- [33] N. Tax, X. Lu, N. Sidorova, D. Fahland, and W. M. P. van der Aalst, "The imprecisions of precision measures in process mining," *Inf. Process. Lett.*, vol. 135, pp. 1-8, 2018, doi: 10.1016/j.ipl.2018.01.013.
- [34] A. Polyvyanny, A. Solti, M. Weidlich, C. Di Ciccio, and J. Mendling, "Monotone precision and recall measures for comparing executions and specifications of dynamic systems," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 3, pp. 1-41, 2020, doi: 10.1145/3387909.
- [35] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "On the role of fitness, precision, generalization and simplicity in process discovery," in *On the Move to Meaningful Internet Systems: OTM 2012*, Lecture Notes in Computer Science, vol. 7565. Berlin, Germany: Springer, 2012, pp. 305-322, doi: 10.1007/978-3-642-33606-5_19.
- [36] J. C. A. M. Buijs, B. F. van Dongen, and W. M. P. van der Aalst, "Quality dimensions in process discovery: The importance of fitness, precision, generalization and simplicity," *Int. J. Cooperative Inf. Syst.*, vol. 23, no. 1, Art. no. 1440001, 2014, doi: 10.1142/S0218843014400012.
- [37] B. F. van Dongen and E. Verbeek, "Challenges and contests in process mining," in *Mining a Scientist's Process: Essays Dedicated to Wil van der Aalst on the Occasion of His 60th Birthday*, Lecture Notes in Computer Science, vol. 16480. Cham, Switzerland: Springer, 2026, pp. 333-348, doi: 10.1007/978-3-032-17618-9_24.
- [38] E. Verbeek, "Process Discovery Contest 2020," dataset, 2021.
- [39] E. Verbeek, "Process Discovery Contest 2024," dataset, 4TU.ResearchData, 2024, doi: 10.4121/3cfcdbb7-c909-4f60-8bec-62c780598047.v1.
- [40] B. van Dongen, "BPI Challenge 2017," dataset, 4TU.ResearchData, 2017, doi: 10.4121/uuid:5f3067df-f10b-45da-b98b-86ae4c7a310b.
- [41] M. de Leoni and F. Mannhardt, "Road traffic fine management process," dataset, 4TU.ResearchData, 2015, doi: 10.4121/uuid:270fd440-1057-4fb9-89a9-b699b47990f5.
- [42] F. Mannhardt, "Sepsis cases: Event log," dataset, 4TU.ResearchData, 2016, doi: 10.4121/uuid:915d2bfb-7e84-49ad-a286-dc35f063a460.
- [43] J. Buijs, "Receipt phase of an environmental permit application process (WABO), CoSeLoG project," dataset, 4TU.ResearchData, 2014. Version 2 (2022), doi: 10.4121/12709127.
- [44] D. M. V. Sato, S. C. De Freitas, J. P. Barddal, and E. E. Scalabrin, "A survey on concept drift in process mining," *ACM Comput. Surv.*, vol. 54, no. 9, pp. 1-38, 2021, doi: 10.1145/3472752.
- [45] A. Kraus and H. van der Aa, "Machine learning-based detection of concept drift in business processes," *Process Sci.*, vol. 2, 2025.
- [46] W. M. P. van der Aalst, "The application of Petri nets to workflow management," *J. Circuits Syst. Comput.*, vol. 8, no. 1, pp. 21-66, 1998, doi: 10.1142/S0218126698000043.

AGENT 01

Ali Akram

FOUNDING ENGINEER

Ali is a founding engineer at 009, where he leads process discovery and built the Petri net layer of the Agent Twin, the part that turns an institution's event logs into a working model of how its operation actually runs. He has spent seven years building machine learning and AI systems. Before 009 he was founding AI engineer at Focal, a San Francisco startup building AI tools for financial advisors, and before that he built LLM systems for stock analysis at TransCrypts. At Cambridge Cognition he built speech models that detect signs of cognitive impairment, and co-authored a study of automated speaker verification in Alzheimer's disease clinical trials, published by the Association for Computational Linguistics in 2023. He holds a degree in Systems Design Engineering from the University of Waterloo.

AGENT 02

Bahaulddin Shammary

FOUNDING ENGINEER

Bahaulddin is a founding engineer at 009, where he builds the verification and governance behind the Agent Twin. Before 009 he was a Lead AI Engineer at Watad, building AridOS, a multi-agent system for construction procurement and credit. Before that he led the production hardening of a multi-agent platform for network operations at OpenText and was a Senior AI Research Scientist in Dell's Office of the CTO, where he filed multiple patents on AI deployment and security. He worked with the Open Source Security Foundation, part of the Linux Foundation, to publish its guide to securing machine learning pipelines. He also co-authored research published by the Association for Computational Linguistics on Arabic dialect intent detection, raising accuracy on Tunisian Arabic from 43 to 94 percent. He holds an MSc in Artificial Intelligence from Munster Technological University, Ireland.

AGENT 03

Yassine Say

FOUNDING ENGINEER

Yassine is a founding engineer at 009, where he leads the work on graph neural networks for process discovery. Before 009 he was a Lead AI Engineer at Watad, where he built the layer that registers, permissions, and orchestrates AI agents across AridOS. Before that he led the team behind Rooceptionist at Vendoroo, a voice agent that handled half a million calls for more than 330 property management companies and resolved about 70 percent of them with no human involved. He spent six years at Dell Technologies, rising from intern to Data Science and Machine Learning Engineer Advisor. He holds a master's in artificial intelligence from Mohammed V University in Rabat.

AGENT 04

Mohammed Hashmi

CHIEF TECHNOLOGY OFFICER

Mohammed is the Chief Technology Officer of 009. Before 009 he was Chief Product Officer at Watad, where he was on the founding team and led the build of AridOS, an agentic operating system for the construction industry in the Gulf. He spent a decade as an engineer at some of the world's most demanding technology companies. At Apple he worked in the Special Projects Group and then as an iPhone systems engineer in Cupertino. He went on to systems engineering at Uber, R&D at Videri, and technical program management at Lyft. He is also a technology partner at Flow Ventures and an angel investor in Riyadh. He holds a degree in Electrical and Computer Engineering from the University of Waterloo, where he won a President's Scholarship.

009

/ ABOUT 009

009 builds an Agent Twin, a working model of an institution that shadows the real operation before anything touches production. The twin maps the operation in read-only shadow mode and returns a verified baseline of cost, cycle time, and error rate. Agents then move from the twin into production through a measured rollout, and every action stays inside the Command Center, which rests on five controls: an audit trail, approval gates, rollback, a kill switch, and an agent registry.

Request a Recon briefing.

The numbers are yours whether or not you proceed.

009LABS.AI