Why Is This on My Bill? Documentation-Grounded Rule Learning for Itemized Bill Explanation

Jason Lefever*
Drexel University
USA
jtl86@drexel.edu

Muhammed Numair Mansur
Amazon Web Services
Germany
numairm@amazon.de

John Konstantinides
Amazon Web Services
USA
johnknst-aws@amazon.com

Martin Schäf
Amazon Web Services
USA
schaef@amazon.com

Willem Visser
Amazon Web Services
USA
vissie@amazon.com

Abstract

Complex itemized bills from services such as cloud computing, healthcare, or telecommunications can be difficult to understand because the pricing terms that determine *what* a customer pays are straightforward, but the eligibility rules that determine *whether* a given line item, a single charge for a product and quantity, qualifies for a given price are scattered across user guides, service terms, and fine print. No single document describes them all, and without them, it is difficult to explain why a specific charge appears on a bill.

We present a neuro-symbolic tool that learns these eligibility rules automatically from billing data and documentation. Pricing terms are evaluated against line items to produce labelled data indicating whether each pricing term was eligible for a given line item. An LLM agent then uses this labelled data, a knowledge base of public documentation, and a small set of rule-evaluation tools to generate rules that are *accurate*: no rule incorrectly labels a line item as ineligible for a pricing term, and *explainable*: every rule is backed by a passage from the knowledge base.

We showcase the tool on the AWS billing system, evaluating millions of line items across hundreds of internal accounts. The tool learns a small set of rules that, together with the pricing terms, explain the majority of observed discrepancies, and we measure documentation coverage, how rules generalize across sample sizes, and the impact of ML pre-seeding on discovery cost.

CCS Concepts

• **Software and its engineering → Software testing and debugging**.

*Work performed while at Amazon Web Services.

ASE '26, Munich, Germany

ACM ISBN 978-1-4503-XXXX-X/2026/10
https://doi.org/XXXXXXX.XXXXXXX

Keywords

Test Oracles, Billing Systems, Eligibility Conditions, LLM Agents, Documentation Coverage

ACM Reference Format:
Jason Lefever, Muhammed Numair Mansur, John Konstantinides, Martin Schäf, and Willem Visser. 2026. Why Is This on My Bill? Documentation-Grounded Rule Learning for Itemized Bill Explanation. In *Proceedings of the 41st IEEE/ACM International Conference on Automated Software Engineering (ASE '26), October 12–16, 2026, Munich, Germany*. ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

Itemized bills from services such as cloud computing, healthcare, or telecommunications contain line items: each maps a product and a quantity to a price and an amount, recording what was consumed and what was charged. For enterprise clients, such bills can contain billions of line items. The pricing terms that determine *what* a customer pays are explicit and well-defined. But the *eligibility rules* that determine *whether* a particular line item qualifies for a given price are not. Precedence rules between overlapping discount programmes, service exclusions, commitment exhaustion, and precision thresholds are scattered across user guides, contractual fine print, implementation details, and many other factors can affect if a line item is eligible for a pricing term. A customer or support agent who wants to understand a bill must assemble these fragments from multiple sources to explain why certain pricing terms apply.

Approach. We present a tool that closes this explainability gap by evaluating pricing terms against line item data and then learning the eligibility rules that explain why certain line items are eligible or ineligible for a given pricing term. The tool runs in two stages. The first, deterministic stage takes line items and pricing terms as input and produces a dynamic traces file: one row per eligible (line item, pricing term) pair, labelled either *pass* (the line item is eligible for the pricing term, and the price or discount matches) or *discrepancy* (the line item is eligible but price or discount doesn't match). We further extend the dynamic traces with some computed fields, such as the total discount, to simplify expressing and learning rules. Dynamic trace rows are the object of study for the second stage.

The second stage is an agentic learner: an LLM equipped with a small set of tools that can evaluate candidate rules against the

dynamic traces, sample rows, query a documentation corpus, and retrieve passages that justify a rule. The learner iteratively proposes, refines, and grounds these *eligibility rules*, which are conditions over the fields of dynamic traces, that suppress discrepancies that would otherwise be reported. Each rule must be *accurate* (match zero pass rows) on the training data set and, wherever possible, *grounded* in a documentation passage that explains why the billing system behaves the way it does. A machine-learning-based preseeder can warm-start the agent with candidate predicates, but the agent handles the rest.

Grounding is what distinguishes meaningful eligibility conditions from rules that are merely accurate over the observed trace: a pure-ML learner would happily propose *"if no discount was applied to this line item, suppress the discrepancy"*. This rule is trivially valid, but vacuous as an explanation of billing and dangerous if deployed. Rejecting such rules requires consulting documentation, which is why the learner is an agent and not a classifier. The output is a rule set that, together with the pricing terms, explains the billing data with zero false positives relative to the observed passes.

Applications. The learned rules enable three capabilities:

- *Bill explanation.* Every line item can be traced to a pricing term, an eligibility condition, and a documentation citation.
- *Documentation monitoring.* Ungrounded rules are documentation gaps; rules whose grounding passage has changed signal drift.
- *Regression testing.* Line items unexplained by any known rule surface billing-system changes for investigation.

Industrial showcase. We demonstrate the tool on the AWS billing system, evaluating millions of line items across hundreds of internal accounts. The tool learns a small set of eligibility rules that explain the majority of discrepancies with zero false positives. We measure documentation coverage, evaluate how rules generalize across sample sizes, and report cost-reduction strategies.

Contributions.

(1) We formalize the *eligibility gap* and characterize the eligibility conditions that arise in a production billing system (§2).
(2) We present a neuro-symbolic agent that learns eligibility rules from billing data, validates their accuracy against observed passes, and grounds them in documentation (§3–§4).
(3) We report our experience of explaining line items of internal AWS accounts, including documentation coverage, generalization across sample sizes, and engineering lessons (§5).

2 End-to-End Example

We walk through the complete process of evaluating pricing terms against line items, learning eligibility rules that explain discrepancies, and grounding these rules in documentation using a publicly documented pricing scenario from AWS.[1]

Two discount pricing terms. Amazon EC2 charges for compute resources on a pay-per-use basis at published *On-Demand rates*. Customers who commit to sustained usage can subscribe to pricing terms that reduce these rates. We explain the two relevant to our example.

Reserved Instances (RI). A Reserved Instance is a one or three-year commitment to a *specific* virtual machine configuration (instance type, operating system, and region). In exchange, the customer receives a discounted rate for that configuration compared to On-Demand pricing. An RI is not a physical machine; it is a billing discount that automatically applies to any running instance that matches the reserved attributes.

Savings Plans (SP). A Savings Plan is a one or three-year commitment to spend a fixed dollar amount per hour on compute usage (e.g., $18.20/hour). Unlike RIs, Savings Plans apply across instance types, regions, and even across services (virtual machines, containers, serverless functions). The billing system applies the Savings Plan rate to eligible usage, starting with the usage that yields the highest savings percentage, until the hourly commitment is exhausted. Remaining usage is charged at On-Demand rates.

We express these *pricing term* as a combination of an expected rate or discount, and the set of conditions under which they apply to a line item, such as date range, product (here EC2), and, if necessary accumulated usage. For our example customer:

- **RI term**: "`r5.4xlarge` Linux in us-east-1 at $0.60/hr" (On-Demand rate: $1.00/hr). This term applies to at most two instances (the customer purchased two RIs).
- **SP term**: "Compute Savings Plan, $18.20/hour commitment, SP rate $0.70/hr for `r5.4xlarge`." This term applies to any SP-eligible compute usage until the commitment is exhausted.

Each pricing term has a product condition, an expectation, and optionally a *volume condition* that limits how much usage the term can cover:

- *RI term*: if the SKU matches the reserved configuration, expect a discount child with rate = $0.60. Volume condition: at most 2 instance-hours per clock-hour (the customer purchased two RIs).
- *SP term*: if the SKU is SP-eligible compute, expect a discount child with rate = $0.70. Volume condition: cumulative SP-discounted spend $\leq$ $18.20 per hour (the commitment amount).

Volume conditions are evaluated *after* all line items are processed: the evaluator accumulates usage across line items and checks whether the capacity was exceeded. A discrepancy on a line item whose pricing term's volume condition is exhausted can be ignored because the term simply ran out of capacity.

Line Items. In a given hour, the customer runs four `r5.4xlarge` instances. The billing system applies discounts in order: RIs first, then Savings Plans. Each usage charge produces a *parent line item* at the On-Demand rate. When a discount applies, the billing system creates a *child line item* linked to the parent. The parent records the product, quantity, and On-Demand rate; the child records the discount rate (or discount percentage) and a negative charge that offsets the parent's charge so that the net billed amount reflects the discounted rate.

To check if a pricing applies to a parent line item, we first check if the product condition matches that of the parent line item (and the date condition if we have one). Then, we check if there is a

[1] Adapted from Scenario 4 at https://docs.aws.amazon.com/savingsplans/latest/userguide/sp-applying.html. Rates are illustrative.

Table 1: Four parent line items (On-Demand rate $1.00/hr) and their discount children, evaluated against the SP pricing term. Instances #1 and #2 are covered by the RI; instance #3 is covered by the SP; instance #4 exceeds the SP commitment. A child records its own rate and a negative charge (−$(on-demand − child rate)) that offsets the parent's on-demand charge.

Parent	Child	Rate	Charge	vs. RI	vs. SP
#1	RI_DISCOUNT	$0.60	−$0.40	Pass	Discrepancy
#2	RI_DISCOUNT	$0.60	−$0.40	Pass	Discrepancy
#3	SP_DISCOUNT	$0.70	−$0.30	Disc.	Pass
#4	(none)	—	—	Disc.	Discrepancy

child line item that matches the rate or discount of our pricing term. We mark the line item as a *pass* if there is a match, otherwise, we mark it as a *discrepancy*. If the parent line item has any child that cannot be explained with our pricing terms, we also mark that as a *discrepancy*.

Table 1 shows this evaluation. The SP term's product condition matches all four parent line items; they are all SP-eligible `r5.4xlarge` instances. The SP term expects each parent to carry a discount child of type `SP_DISCOUNT` whose rate equals $0.70. Only instance #3's child satisfies this expectation; its $0.30 negative charge reduces the net hourly cost to the SP rate. The other three are discrepancies: instances #1 and #2 carry an `RI_DISCOUNT` child with a different rate and offset, and instance #4 carries no discount child at all, so the parent is billed at the full On-Demand rate.

All three discrepancies reflect *correct* billing:

- Instances #1 and #2 received the RI rate instead, because RIs take precedence over Savings Plans.
- Instance #4 was charged at On-Demand because the SP's hourly commitment was already exhausted by instance #3 and other workloads (Fargate, Lambda) in the same hour.

But the SP pricing term does not mention RI precedence. From the term's perspective, instances #1 and #2 are unexplained. Instance #4, however, is handled by the term's volume condition: after aggregating all SP-discounted usage in the hour, the evaluator determines that the $18.20 commitment was exhausted, and automatically relabels the discrepancy as a pass—no learned rule is needed.

From Discrepancies to Grounded Rules. After the volume condition step relabels instance #4, two discrepancies remain: instances #1 and #2. They are the input to the learning stage. The evaluator has written a labelled dynamic traces T containing all values of line item and pricing term in a flat row; the two remaining discrepancies sit next to their matched SP pricing term, while the pass on instance #3 and all the other rows the evaluator produced are also present in T.

An agentic learner then operates on T. Equipped with tools that evaluate candidate rules against T, sample rows, query a documentation corpus, and retrieve passages that support a rule, it proceeds roughly as follows. It first notices that both remaining discrepancies share a distinctive feature: both have a child of type `RI_DISCOUNT`. This feature does not appear on the pass row for instance #3. It proposes the candidate rule *"if a line item has a child of type* `RI_DISCOUNT`*, suppress the SP pricing term's discrepancy"*, validates it by checking that no pass row in T matches, and queries the Savings Plans User Guide for a passage that justifies it. The guide states *"Savings Plans apply to your usage after the Amazon EC2 Reserved Instances (RI) are applied"*[2], which directly supports the rule. The agent commits the rule, now both *accurate* (zero matching passes) and *grounded* (cited passage). Had no passage supported it, the rule would still be returned but flagged as an unexplained documentation gap.

Rule `ri-precedence-over-sp` (the contribution)

$$\varphi = \texttt{child.RI\_DISCOUNT.exists} = \texttt{true} \wedge \texttt{term.discountProgram} = \texttt{SavingsPlan}$$

Action: suppress discrepancy. **Grounded in:** AWS SP User Guide— *"Savings Plans apply after Reserved Instances are applied."*

Such a rule is not stated in any single pricing term: it is *learned* from labelled evaluation data, *validated* against observed passes, and *grounded* in public documentation. Producing these rules end-to-end is the contribution of this paper.

We discuss later how our agentic learner can be *pre-warmed* with candidate rules using traditional machine learning techniques to reduce the cost and time of rule discovery, however, we still need the agentic loop for grounding rules in documentation to weed out overfitted candidates.

With this rule and the earlier volume-condition step, every line item in the example is explained: instance #3 passes directly; instances #1 and #2 are explained by RI precedence, with a citation; instance #4 is relabelled a pass because the SP commitment was exhausted. The following sections formalize each step: §3 defines pricing terms, line items, and the trace file T, and §4 describes the agentic learner in detail.

3 Pricing Terms, Line Items, and Rules

This section defines line items and pricing terms, formalizes how we evaluate pricing terms against line items, what a discrepancy is, and how the labelled trace file produced by the evaluation drives rule learning. A short rule grammar at the end of the section is the target language of the learner in §4.

Line Items and Child Modifiers. A *line item* is a single billing record. We follow the FOCUS [15] schema, an open standard for cloud cost and usage data, but the approach works with any tabular format. Each line item has 40+ fields; the top half of Figure 1 shows the subset relevant to this paper.

A billing event is represented as one *parent* line item together with zero or more *child* modifiers linked to it. The parent records the base charge (at the public rate). Each child records a modifier: a *percentage discount* (e.g. a 5% cross-service discount) or a *rate override* (e.g. a Savings Plan or Reserved Instance benefit). The child's `typeCode` identifies the modifier class (e.g. `RI_DISCOUNT`,

[2] https://docs.aws.amazon.com/savingsplans/latest/userguide/sp-applying.html

type	product	sku	rate	usage	amount	...
Usage	EC2	HQR7...	1.00	24	24.00	...
RI	EC2	HQR7..	—	—	−9.60	...
DI	EC2	HQR7..	—	—	−0.72	...

label	li.product	li.amount	t.type	t.value	c.RI	c.DI	...
✓	EC2	24.00	Disc	5.0	✓	✓	...
✗	EC2	24.00	Rate	0.70	✓	✓	...

Figure 1: Top: a billing line item (parent) with two child modifiers (Reserved Instance benefit and a percentage discount), shown with a subset of columns. Bottom: two dynamic trace rows derived from the same line item paired with two different pricing terms. The first term (a 5% discount) passes; the second (a Savings Plan rate of $0.70) is a discrepancy because the RI was applied instead. Prefixes: `li` = line item, `t` = pricing term, `c` = child summary.

SP_DISCOUNT); its amountBeforeTax is the signed adjustment. A parent may have multiple children when several modifiers stack.

Pricing Terms. A *pricing term* prescribes how usage of a given set of products should be charged. It consists of a *product condition* (which SKUs the term covers), an optional *date condition*, an optional *volume condition* (bounding cumulative usage), and an *expectation*. Conditions are first-order Boolean formulas over line-item fields. Expectations take one of two forms:

- Rate(r): the net rate after discount equals r;
- Discount(d): a child reflects a d% reduction.

Tiered terms additionally keep links to their neighbouring tiers. For our showcase, pricing terms can be extracted directly using the AWS Pricing client; other cloud providers expose similar APIs that return pricing terms in machine-readable form.

We say a pricing term *matches* a line item if its product and date conditions are satisfied by the line item's product and date (volume conditions are evaluated post-hoc, see below). When a term matches and a child of the line item satisfies the term's expectation, the pair is a *pass* and the line item's usage is accumulated toward the term's volume condition. When the term matches but no child satisfies its expectation, or when a child exists that no matching term explains, the pair is a *discrepancy*. Every (line item, pricing term) observation is recorded as a dynamic trace: a flat key-value record containing the line-item fields, the pricing-term fields, and a handful of computed fields that the learning stage can reuse without rederiving them.

Dynamic traces. A *dynamic trace* is the flat key-value record written for a single (line item, pricing term) observation, together with some pre-computed fields, such as the final amount of a line item after applying all its children, that simplify downstream rule learning. The format is borrowed from the Daikon invariant detector [14]; although our approach does not require Daikon, the format is convenient. The bottom half of Figure 1 shows two example rows. Fields span four namespaces:

- `lineItem.*`: raw fields from the billing record (product code, SKU, rate, usage type, amount);
- `term.*`: the matched pricing term's expectation type, value, and discount-programme classification;
- `computed.*`: derived values such as `netAmount` (parent amount plus all children) and `expectedDiscountAmount`;
- `child.T.*`: per-type summaries for each child modifier type T (exists, count, discount factor, amount).

The evaluator also records *sibling context*: how many other pricing terms passed or showed a discrepancy on the same line item, and whether any satisfied term's discount exceeds the current term's expectation. This context enables rules that reason about interactions between pricing terms (e.g. "a larger discount was applied by a higher-priority term").

This flat structure is a deliberate design choice: precomputing values like the total discount, or Boolean fields recording the existence of a given child type, lets the learning stage express rules over simple predicates rather than having to rederive this information from scratch inside each rule.

All dynamic traces are written to a single labelled file, with a `label` column distinguishing passes from discrepancies. This file is the labelled data consumed by the learning stage (§4).

Rule: `ri-discount-precedence-over-sp`

```
child.RI_DISCOUNT.exists = true ∧ term.discountProgram = SavingsPlan
```

"Savings Plans apply after Reserved Instances are applied."[3]

Figure 2: The eligibility rule from the motivating example (§2), expressed in the rule grammar. The rule suppresses discrepancies of the SP pricing term on line items that already received an RI discount. The quoted documentation grounds the rule.

Eligibility Rules. An *eligibility rule* is a named condition over dynamic trace fields. Rules are first-order conditions over expressions, using the same grammar as pricing-term conditions: atoms, set membership, Boolean connectives, and arithmetic:

$$\varphi \;::=\; f = v \;\mid\; f \in V \;\mid\; e_1 \bowtie e_2 \;\mid\; \varphi \wedge \varphi \;\mid\; \varphi \vee \varphi \;\mid\; \neg\varphi$$

where f is a field name, v a constant, V a set, $\bowtie \in \{<, \leq, >, \geq, =, \neq\}$, and e is a numeric expression over field values, constants, and arithmetic operators ($+, -, \times, /, |\cdot|$).

Figure 2 shows the rule from the motivating example. Other rules can contain linear inequalities; for instance, the *precision-threshold* rule suppresses discrepancies when the expected discount amount falls below the billing system's rounding precision:

$$\texttt{term.expectation.type} = \texttt{Discount} \;\wedge\; |\texttt{computed.expectedDiscountAmount}| < 10^{-10}$$

A rule is *accurate* with respect to a trace file T if it matches zero pass rows in T, i.e. it never suppresses a pricing term that correctly matches a child line item. Our tool drops inaccurate rules, because retaining them would lead to incorrect explanations for why a pricing term did or did not apply to a line item.

[3] https://docs.aws.amazon.com/savingsplans/latest/userguide/sp-applying.html

Algorithm 1 EVALUATE: pricing terms against line items.

Require: Line items L, pricing terms O
Ensure: Labelled dynamic trace file T (one row per (line item, pricing term) observation; each row has a `label` column in {*pass*, *discrepancy*})

$T \leftarrow \emptyset$
for each $\ell \in L$ **do**
  $M \leftarrow \{o \in O \mid prod(o)(\ell) \wedge date(o)(\ell)\}$
  $S \leftarrow \emptyset$; $E \leftarrow \emptyset$ ▷ satisfied terms; claimed children
  for each $o \in M$ **do** ▷ *match:* at most one child per term
    for each $c \in children(\ell) \cup \{\ell\}$ with $c \notin E$ **do**
      if $exp(o)$ is satisfied by c **then**
        S += $\{o\} \cup tierGroup(o)$; E += $\{c\}$
        accumulate ℓ's usage toward $vol(o)$
        break
      end if
    end for
  end for
  for each $o \in M$ without an unsatisfied predecessor **do** ▷ *emit:* pricing-term rows
    T += {DTRACE(ℓ, o, S, M) with label ($o \in S$? *pass* : *discrepancy*)}
  end for
  for each $c \in children(\ell) \setminus E$ with an expectation **do** ▷ *emit:* unexplained-child rows
    T += {DTRACE$(\ell, \perp, S, M)$ with c and label *discrepancy*}
  end for
end for
Post-processing: for every pricing term o with a volume condition whose accumulated usage exceeds $vol(o)$, relabel all discrepancy rows for o in T as *pass*
return T

3.1 Evaluation Algorithm

Algorithm 1 takes line items L and pricing terms O and returns a single labelled dynamic trace file T: each row is one (line item, pricing term) observation, and the `label` column marks it as a *pass* or a *discrepancy*. The algorithm proceeds in three steps per line item. The first step walks the matching pricing terms and greedily pairs each with at most one child that satisfies its expectation; a match marks the term *satisfied* and propagates to the entire tier group, and the child is *claimed* so that stacked modifiers are attributed one-to-one. The second step emits a row for every matching pricing term whose predecessor (if any) was already satisfied. Satisfied terms become pass rows, unsatisfied terms become discrepancy rows. The third step emits a discrepancy row for any child that was never claimed but carries its own expectation, covering the case "a child that cannot be explained by a pricing term". Finally, post-processing relabels discrepancies whose pricing term's volume condition was exhausted: these reflect legitimate out-of-capacity usage and must not appear as discrepancies, nor as negative training data for rule learning.

3.2 Applying Eligibility Rules

Algorithm 2 takes the dynamic trace file from Algorithm 1 and an eligibility rule set R, and computes per-rule statistics $\sigma(r)$ together with the discrepancies left unexplained. For each rule r, the algorithm counts the pass rows and discrepancy rows it matches:

Algorithm 2 APPLYRULES: eligibility rules against a dynamic trace file.

Require: Dynamic trace file T, eligibility rules R
Ensure: Per-rule stats $\sigma : R \to \mathbb{N}^2$ giving (*matched discrepancies*, *matched passes*), and the set of reported discrepancies $D^\star$

$\sigma(r) \leftarrow (0, 0)$ for all $r \in R$
$D^\star \leftarrow \emptyset$
for each row $\tau \in T$ **do**
  $match \leftarrow \{r \in R \mid \varphi_r(\tau)\}$
  for $r \in match$ **do**
    if τ.`label` = *discrepancy* **then** $\sigma(r)$.discrepancy += 1
    else $\sigma(r)$.pass += 1
    end if
  end for
  if τ.`label` = *discrepancy* $\wedge$ $match = \emptyset$ **then** $D^\star$ += $\{\tau\}$
  end if
end for
return $(\sigma, D^\star)$

Algorithm 3 LEARNRULES: driver for agentic rule discovery.

Require: Dynamic trace file T, documentation corpus $\mathcal{D}$, round budget K, optional pre-seed learner $\mathcal{L}$
Ensure: Eligibility rule set R (accurate w.r.t. T, each rule grounded in $\mathcal{D}$ when possible)

$R \leftarrow \mathcal{L}(T)$ if $\mathcal{L}$ is provided, else $\emptyset$ ▷ optional ML pre-seeding
for $k \leftarrow 1$ to K **do**
  $R' \leftarrow$ AGENT$(T, R, \mathcal{D})$ ▷ Figure 3
  if $R' = R$ **then break**
  end if ▷ fixed point
  $R \leftarrow R'$
end for
return $\{r \in R \mid r$ is accurate w.r.t. $T\}$

r is *accurate with respect to* T iff $\sigma(r)$.pass = 0, and its *coverage* on T is $\sigma(r)$.discrepancy. A discrepancy is reported in $D^\star$ only if no rule explains it; any caller that wants to treat inaccurate rules as "not suppressing" can simply filter R before calling Algorithm 2. The two algorithms share no state: Algorithm 1 is the expensive scan of billing data (once per billing period, cached as T), whereas Algorithm 2 is a cheap filter that can be re-run many times as the rule set evolves.

4 Agentic Learning of Eligibility Rules

Given the labelled dynamic trace file T produced by Algorithm 1, the goal is to find a set of rules R such that every $r \in R$ is accurate with respect to T (matches zero pass rows) and the combined rule set maximizes coverage of the discrepancy rows. We cast this as an *agentic* task: an LLM with a small set of tools iteratively proposes, validates, and grounds rules until no further progress is possible. Algorithm 3 is the outer driver; Figure 3 shows what happens inside a single invocation of the agent.

Rule grammar. A rule φ is a first-order logic formula over the dynamic trace field names, as described in the previous section. To ensure the agent can quickly validate, evaluate, and share generated

Inside AGENT(T, R, $\mathcal{D}$)

Tools exposed to the LLM:

- APPLY_RULES(R') $\mapsto$ $(\sigma, D^\star)$ (Algorithm 2)
- PROFILE(*field*) data summaries, error-only values
- SAMPLE(*filter*) $\mapsto$ rows of T
- LOOKUP(*query*) retrieve from $\mathcal{D}$
- GROUND(r) $\mapsto$ passages in $\mathcal{D}$ supporting r

Typical trajectory (abbreviated; ReAct-style [32]):

```
thought  Which discrepancies are still unexplained?
act      apply_rules(R)
obs      D* = 412 rows; two clusters visible.
thought  Cluster 1 always has a child RI_DISCOUNT.
act      sample({child.RI_DISCOUNT.exists, label=discrepancy})
obs      87 rows; SP pricing term on every one.
thought  Propose r: suppress SP pricing term when RI child
exists.
act      apply_rules(R ∪ {r})
obs      σ(r) = (87, 0) − accurate, covers 87 discrepancies.
thought  Is r grounded?
act      ground(r)
obs      "Savings Plans apply after Reserved Instances."
thought  r is accurate and grounded. Commit and continue.
.
:     (repeat for other clusters)
return    R ∪ {r,...}, each rule annotated with its grounding
```

Figure 3: One invocation of AGENT from Algorithm 3. The agent is a Kiro instance with tool access; it is prompted to extend or refine R, and may revise or discard any rule using APPLY_RULES to re-score after each edit. Rules that cannot be grounded in $\mathcal{D}$ are flagged as documentation gaps but still returned. The agent terminates the round when it reports no further progress.

rules, and obtain fast feedback about syntax errors, we use a shared schema file, and lightweight tools to parse and test generated rules. Accuracy and coverage are exactly as returned by Algorithm 2.

Pre-seeding with ML learners. The optional pre-learner $\mathcal{L}$ in Algorithm 3 supplies candidate predicates without any LLM calls. We run two decision tree learners on T: a CART tree [10] using Gini impurity, and a hybrid tree we call BINC4.5 [10, 26] using C4.5-style gain ratio with error-based pruning and CART-style binary partitioning of categorical values. Both use iterative peeling: fit a tree, extract root-to-leaf paths that predict discrepancy with precision ≥ 0.9, remove the covered discrepancies, and repeat. The binary partitioning produces set-membership conditions directly. Features with more than 20 distinct values are excluded before tree construction to bound the cost of evaluating binary partitions of the values. The candidates are accurate by construction (checked against the pass rows of T) but ungrounded; the agent decides which to keep, rewrite, or discard based on documentation evidence.

On our data, BINC4.5 consistently produces simpler candidates than CART. The feature space is dominated by low-cardinality categorical attributes, where gain ratio—designed to penalize high-cardinality splits [26]—selects more informative first splits than Gini impurity. Across 11 batches, BINC4.5 found depth-1 rules in 6 where CART required depth-2 or depth-3 conjunctions for the same coverage and precision. Simpler candidates are easier for the agent to ground: a single set-membership condition maps to one documentation question, whereas a conjunction generates several. We run both learners and let the agent select from the union. A promising future direction is replacing greedy induction with optimal classification trees [1, 9], which formulate tree construction as a mixed-integer program and could be adapted to directly maximize coverage subject to a zero-false-positive constraint.

Tool back-ends. PROFILE returns categorical distributions and numeric summaries per field, plus an `errorOnlyValues` section listing values that appear only on discrepancy rows which serve as candidates for rule atoms. LOOKUP and GROUND are backed by a pre-built summary of the user-provided reference resources (documentation pages, wiki articles, user guides, contracts): each resource is chunked into passages and indexed in a vector store. Generating the summary once, in an independent LLM invocation, reduced per-round agent cost by 76% and wall-clock time by 81% relative to letting the agent read raw resources.

Rule-to-question decomposition for grounding. Embedding a formal rule condition directly as a retrieval query performs poorly: predicate vocabulary (field names, operators, thresholds) differs from documentation vocabulary (natural language, business concepts). GROUND therefore first asks the LLM to produce the natural-language questions that a resource would have to answer in order to justify the rule. For example, a rule suppressing an SP discrepancy when an RI child exists generates *"Which discount type is applied first when both RI and SP are available?"* Each question is retrieved independently; a second pass checks whether the union of retrieved passages answers all of them. A rule is *grounded* if every question is answered by at least one passage, *ungrounded* otherwise. The generated questions are kept with the rule: they tell the documentation owner exactly what the resource should say.

To avoid rule duplication, we use an internal partial order over rules: $r_1 \sqsubset r_2$ when r_1's discrepancies are a strict subset of r_2's. This lets the agent skip narrower rules once a broader one is grounded, which cuts tool-call counts without changing the returned rule set.

Why an agent, rather than a classifier over T? Accuracy over T is a strictly weaker property than "describes an eligibility condition". Because every pass row in T has, by construction, a matching child, any rule whose condition fires only on line items without such a child is automatically accurate. The extreme case being *"if the line item has no children, suppress the discrepancy"* would satisfy APPLY_RULES with zero false positives and suppress most of the discrepancy rows in T, but it describes no billing mechanism and would silently hide every future regression in which a discount was dropped. Rules keyed on a specific `account_id` set, or on an arbitrary identifier that happens to cluster with the discrepancies, are lower-contrast versions of the same failure: accurate on the training data and vacuous about billing.

A classifier has no principled mechanism to reject these rules; it only sees labels. The agent does: for any candidate it decomposes the rule into the natural-language questions a documentation passage would have to answer in order to justify it (§4), and rejects the rule when no passage answers them. The criterion is semantic, and it maps directly to what the user actually wants: rules that describe the eligibility conditions the billing system is supposed to implement. Our ML configuration indeed proposes exactly this

kind of overfitted rule (an accurate but account-specific pattern, discussed in §5.3); grounding is how we catch it.

Output of the tool. A rule that is still inaccurate when Algorithm 3 returns is dropped from the output; APPLY_RULES makes this check cheap. A rule that is accurate but ungrounded is returned and flagged for human review: either the documentation is incomplete (an eligibility condition that is not described in any provided resource) or the billing system is behaving unexpectedly, and the rule is classifying the anomaly. Because accuracy is defined relative to the observed T, the agent is re-run on an updated T whenever the corpus changes; this either confirms the existing rules or flags the ones that need refinement.

5 Evaluation

This section evaluates the tool on AWS billing data of internal accounts as an *industrial showcase*, i.e, how the approach behaves inside our organization, not a controlled experiment for generalization. We answer four questions: **RQ1 (Effectiveness)**: can the tool learn a set of eligibility rules that explains the observed discrepancies? **RQ2 (Grounding)**: what fraction of learned rules can be grounded in documentation, and what do the ungrounded ones indicate? **RQ3 (Size generalization)**: do rules learned on a small sample continue to hold on larger slices of the corpus? **RQ4 (Seeding)**: does ML pre-seeding reduce discovery cost without sacrificing rule quality?

5.1 Setup

The corpus is a 2026-04 snapshot of 1,695,467 discrepancies and 5,402,991 verified passes produced by running Algorithm 1 on the line items and pricing terms of 100 internal service accounts that are eligible for various discount pricing terms (no customer data). We stratified-sample three sizes: 10k, 50k, and 200k line items, balanced across the four discrepancy strata (the different types of pricing terms), targeting a 1:1 discrepancy-to-pass ratio (e.g. 4999 errors and 4999 passes at N=10k).

For each size we run two configurations of Algorithm 3. *Cold*: $\mathcal{L} = \emptyset$; the agent starts from the labelled sample and the documentation corpus. *Warm*: $\mathcal{L}$ is a pair of ML learners, whose candidate predicates are filtered for accuracy and handed to the agent as a head start. Both configurations share the same agent (Claude Opus 4 via Kiro CLI 2.0), the same prompt structure (hard constraints, grounding guidance, stopping criterion), the same accuracy check, and the same documentation corpus (11 internal wiki pages, 7 public AWS docs). The only difference is whether the agent starts with ML candidates or an empty rule set. Every (config, size) combination is repeated 3 times with different seeds. A post-processing step removes subsumed rules (rules whose error set is a strict subset of another rule's).

RQ1, RQ2, and RQ3 are answered from the cold configuration. RQ4 (§5.5) introduces the warm configuration.

Discount programmes. The accounts in our corpus are enrolled in several overlapping discount programmes. *Percentage discounts*: cross-service (CSD) or service-specific (SSD) grant a fixed percentage reduction in exchange for a spending commitment or upfront payment [5, 6]. *Private Rate Cards* (PRC) are negotiated per-SKU rates that replace the public rate entirely, as offered through marketplace custom pricing [4]. A single line item can be eligible for multiple programmes; the billing system applies them in a precedence order that is not fully described in any single document. The eligibility rules the tool discovers express which programme took precedence and why.

Table 2: Cold-start results per N across 3 runs. "Rules": accurate, non-subsumed rules returned. "Coverage": percentage of discrepancy rows suppressed by at least one rule. "Grounded": rules with a supporting documentation passage. "Credits": LLM credits per run. "Time": wall-clock seconds.

N	Rules	Coverage	Grounded	Credits	Time (s)
10k	7–8	98.0–99.9%	5–7	11.3–16.2	822–1,089
50k	6–11	99.8–99.9%	6–9	13.9–19.0	1,173–1,964
200k	8–9	97.8–99.8%	6–8	16.0–16.7	3,640–4,176

5.2 RQ1: Effectiveness and Non-Redundancy

Table 2 summarizes the nine cold runs. Every proposed rule is accurate. Zero pass-row matches under re-validation, and non-subsumed after the deduplication step. A single run returns 6 to 11 rules and suppresses 97.8–99.9% of the sample's discrepancies; the average across the nine runs is 99.4%. LLM spend stays within a narrow band (11.3–16.2–16.0–16.7 credits) and is essentially flat in sample size: the agent issues roughly the same number of propose/-validate/ground rounds at 200k as at 10k. Wall-clock time grows with N because the agent spends most of each round reading and profiling the labelled data.

Non-redundancy. For each run we classify every ordered pair of rules by how their matched-discrepancy sets relate: *disjoint* (no shared rows), *subsumed* (one is a strict subset of the other), or *overlapping* (shared rows, neither contains the other). Across the 267 pairs in the nine cold runs, 260 (97%) are disjoint and 7 (3%) overlap without containment. Zero pairs are subsumed. The post-processing step already removed them. The agent finds distinct eligibility conditions with high reliability.

5.3 RQ2: Grounding in Documentation

Table 3 lists all 15 rule categories with subsumption relationships. To construct this table we evaluated every rule from all nine cold runs against a shared anchor sample (200k, seed #1) and clustered by Jaccard similarity ≥ 0.8 on matched-discrepancy sets. Subsumption is a *cross-run* property: within any single run, the deduplication step already removed subsumed rules (Table 2 reports the post-dedup count). But when rules from different runs are evaluated on the same anchor, a broad rule discovered in one run can subsume a narrower rule discovered in another, the agent expresses the same eligibility gap at different levels of generality depending on the sample it sees.

The eleven non-indented rows are *independent*: removing any one reduces coverage. The four indented rows are strict subsets of their parent and add no marginal coverage. The table also illustrates *generality and groundability are in tension.*

Table 3: Rule categories discovered across the nine cold runs, ordered by peak discrepancies suppressed on the 200k anchor sample. "Runs": in how many of the 9 runs the category appears. "Grounded": rule instances with a verbatim documentation passage vs. total instances. Indented rows (⊂) are subsumed by the parent, their error set is a strict subset, and add no marginal coverage. Categories are clustered by Jaccard similarity ≥ 0.8 on matched-discrepancy sets.

#	Category	Runs	Discrepancies	Grounded	Short explanation
1	SP discount present, no CSD child	9/9	85,198	2/9	Savings Plan discount child present; no CSD/SSD child generated.
	⊂ SP covers charge (Other prog.)	1/9	1,282	1/1	Savings Plan discount on "Other" programme; no CSD/SSD child.
2	PRC supersedes discount	9/9	9,091	9/9	PRC present; discount amount does not match oracle expectation.
	⊂ PRC on private pricing	1/9	1,346	1/1	PRC on private-pricing line supersedes CSD/SSD.
	⊂ PRC + SP supersede discount	1/9	951	0/1	Both PRC and SP present; no CSD/SSD child.
3	Discount-child amount mismatch (sibling)	8/9	2,329	9/9	Discount child present but amount differs; a sibling oracle was satisfied.
	⊂ Discount child, no sibling passed	1/9	543	1/1	Discount child present; no sibling oracle passed.
4	Savings Plan supersedes CSD/SSD	6/9	1,684	6/6	Savings Plan discount child present; no CSD/SSD child.
5	PRC usage-amount mismatch	2/9	886	2/2	PRC present; usage-weighted amount differs from oracle.
6	Precision threshold	9/9	256	9/9	Expected discount $< 10^{-10}$; below rounding precision.
7	SSD supersedes CSD within discount	5/9	209	5/5	Service-Specific overrides Cross-Service on same line item.
8	UBD not eligible for discount	8/9	126	5/8	Usage-Based Discount present; distinct mechanism.
9	Internal-only rate cards	6/9	52	6/6	Rate cards for internal accounts; not modelled by the pricing system.
10	SP alters discount percentage	2/9	30	2/2	SP and discount coexist; SP alters rate via composition rule.
11	Credit-only, no discount child	4/9	8	4/4	Only credit children; no discount child to match oracle.

Rule 1 ("SP discount present, no discount child") is the largest rule, covering 85,198 discrepancies, but it grounds in only 2 of 9 runs. Its condition, "Savings Plan discount child exists and no CSD/SSD child exists", is a billing *symptom* that can arise from many different causes, and no single documentation passage explains all of them. By contrast, the independent rules that cover *other* reasons for a missing discount child, e.g. usage-based discounts (5/8 grounded), internal-only rate cards (6/6), credit-only (4/4), each name a specific mechanism and can be anchored in a specific document. A deployment that cares about explainability would prefer the narrower, groundable rules; a deployment that cares only about coverage would keep the broad rule.

Rules about Usage-Based Discount (UBD, Rule 8) fail to ground because the billing system calls the mechanism *Usage-Based Discount*, but the public documentation refers to it as a *bundled discount*[4]. The agent's retrieval misses the synonym. This could be addressed by enriching the grounding prompt with known synonyms.

All other rules could be grounded in the provided documentation.

Of the 73 rules returned across the nine cold runs, 62 (85%) are grounded in a verbatim documentation passage. The remaining 11 (15%) are flagged *quote-missing*. Eight of these are vocabulary mismatches (Rule 1's broad condition does not map to a single documentation passage) and three are synonym gaps (Rule 8's UBD mechanism, discussed above). No overfitted or vacuous rules survived in the nine cold runs. We revisit this under RQ4 where the ML pre-seeder *does* produce candidates that the accuracy filter rejects.

5.4 RQ3: Size Generalization

Since every sample is drawn from the same corpus, we can ask whether rules learned at a small size continue to be both accurate and high-coverage on a larger sample. Table 4 reports both numbers.

[4] https://docs.aws.amazon.com/cur/latest/userguide/discount-details.html

Table 4: Cross-size evaluation per seed. Each cell shows coverage (%) on the test sample using rules learned on the training sample. Numbers after "/" count pass rows incorrectly matched (accuracy violations when evaluated on a sample the rules were not trained on).

Train	Random Seed #1			Random Seed #2			Random Seed #3		
	10k	50k	200k	10k	50k	200k	10k	50k	200k
10k	99.9	99.8	99.8/8	98.0	97.9	98.0/13	99.8	99.8/8	99.8/26
50k	99.9	99.9	99.8/11	99.8	99.8	99.8/12	99.9	99.9	99.9/6
200k	99.9	99.8	99.8	99.7	99.6	99.7	97.9	97.8	97.8

Coverage is stable. A rule set learned at 10k covers 97.9–99.8% of discrepancies at 200k, comparable to what the same rules achieve on their training sample. The agent's rules generalize: concepts that discriminate discrepancies from passes at 10k continue to do so at 200k.

Accuracy occasionally breaks at larger sizes. Six of nine cross-size evaluations show a small number of pass rows (6–26) matched by a rule that was accurate on its training sample. The largest violation, seed #3's 10k rules leaking 26 passes (0.03%) at 200k, concerns rules whose conditions are accurate on the stratified 10k slice but not on the broader distribution. These are not bugs in the tool: they are empirical evidence that our stratification is too narrow. When a rule needs to be promoted for production use, it must be re-validated against a larger sample before shipping.

5.5 RQ4: Impact of ML Pre-Seeding

The warm configuration differs from cold only in that the agent starts with ML-mined candidate predicates (filtered for accuracy before the agent sees them). Both configurations share the same

Table 5: Warm-start results per N. The warm agent starts with ML candidates (filtered for accuracy), then runs the same residual-discovery loop as cold. Columns match Table 2.

N	Rules	Coverage	Grounded	Credits	Time (s)
10k	3–7	99.9%	3–7	13.2–17.4	814–992
50k	6–8	98.0–99.9%	6–7	16.4–19.2	1,675–1,907
200k	11	99.9%	6	13.7	4,001

prompt structure, grounding guidance, accuracy check, and stopping criterion (≥1% residual reduction). Table 5 reports the seven completed warm runs.

Warm converges faster at comparable quality. Across the seven warm runs, total Kiro agent cost is 114.5 credits (vs. 144.2 for the nine cold runs) and total wall time is 3.4 hours (vs. 5.4 hours for cold). Per-run, warm is roughly 20% cheaper in credits and 37% faster in wall-clock time. Coverage and accuracy are comparable: warm achieves 98.0–99.9% coverage with zero pass leaks, matching cold's 97.8–99.9%. The ML candidates give the agent a head start that saves roughly one exploration round; both configurations then converge to the same fixed point.

The accuracy filter catches leaky ML candidates. The ML learners use a 90% precision threshold internally and can emit rules that leak passes. The warm agent's first action is to filter the candidate set against the sample's pass rows; 1–3 candidates per run are dropped this way. This is the accuracy-is-a-weaker-property argument from §4 made concrete: the ML stage produces accurate-sounding rules that the symbolic check immediately rejects, and the agent never spends LLM budget on them.

Operational recommendation. The two configurations reach the same quality; warm gets there faster. A production deployment should use warm when a nightly pipeline budget constrains wall-time, and cold when simplicity matters (no ML infrastructure needed). Both configurations share the accuracy check and deduplication step, so their outputs are directly comparable.

5.6 Threats to Validity

As an industrial showcase, we do not attempt to generalize beyond the AWS billing domain. Results depend on the model (Claude Opus 4), the Kiro CLI harness, the prompt, and the grounding corpus. All four evolve. We mitigate with 3 seeds per configuration and by re-validating every rule against the full pass set rather than the sampled subset. The accuracy check assumes the pass set is correctly labelled: pricing-term compilation is deterministic, and a manual review of 200 sampled passes found no labelling errors; if an oracle is wrong, the check conservatively flags the rule as an unexplained discrepancy rather than suppressing it.

The cross-size accuracy violations in RQ3 (Table 4) show that accuracy on a small stratified sample does not guarantee accuracy on a larger sample. In practice our deployment re-validates every rule against the full pass set before production use, which catches the small-sample leaks; the main threat is that our initial stratification strata (four discount-programme categories) may not capture all the dimensions along which eligibility conditions vary. The predicate grammar is single-line-item; cross-line-item conditions (volume tiers spanning charges, cross-hour commitment tracking) require grammar extensions we leave for future work.

6 Related Work

The oracle problem. Barr et al. [8] survey the test oracle problem. In our domain the oracles themselves are given (derived from pricing terms), so the challenge is not generating oracles but learning the eligibility conditions they leave implicit. TOGA [13] and TOGLL [18] use neural methods and LLMs to generate test oracles from code context; Hossain et al. [19] found that TOGA produces false positives 47% of the time.

Invariant detection and learning from examples. Daikon [14] discovers likely program invariants from execution traces using statistical analysis over observed values. Our bootstrapping adapts this idea to billing data, treating discrepancies and passes as the two partitions. The ICE framework [16] learns invariants from positive examples, negative examples, and implication counterexamples. Our setting provides natural ICE instances (passes, discrepancies, and counterexample feedback from the validator). MELT [27] mines lightweight code-transformation rules from pull-request histories, treating each PR as a labelled example of a rewrite; we mine eligibility rules from the analogous label, passes vs. discrepancies, on billing traces. Recent work on LLM-based invariant generation [30] uses language models to propose loop invariants with counterexample guidance; structurally similar to our iterative loop, applied to program verification rather than oracle refinement. CEGAR [11] iteratively refines an abstract model by analysing spurious counterexamples; our methodology follows this structure.

Model learning and black-box checking. Angluin's L^* algorithm [7] learns a minimal DFA from membership and equivalence queries. Peled et al. [25] combine L^* with model checking to verify black-box systems against temporal specifications, learning a model and checking it against a property in an interleaved loop. Vaandrager [31] surveys applications of model learning to protocol conformance testing, where learned automata are compared against RFC specifications. Meinke and Sindhu [24] apply learning-based testing to reactive systems, incrementally building Kripke structures from observations. Our work shares the learn-and-check structure: we learn eligibility rules (analogous to a model) and check them against documentation (analogous to a specification). The key difference is that our "model" is a set of predicate rules rather than a state machine, and our "specification" is natural-language documentation rather than a formal property.

Grey-box and code-augmented analysis. Grey-box testing combines black-box techniques with partial access to system internals [17]. DSD-Crasher [12] combines Daikon invariants with symbolic execution. These approaches require deep tooling integration. CodaMosa [23] uses LLMs to generate test inputs for hard-to-cover branches; our work applies a similar idea to rule discovery rather than input generation.

LLM-based specification mining. RBCTest [20] uses LLMs to mine oracles from API response bodies. Kamath et al. [21] leverage LLMs to propose program-verification artifacts that are then discharged

by a symbolic back-end. The same two-tier propose-and-check pattern we instantiate with a documentation grounding step. DiffSpec [28] drives an LLM with natural-language specifications to produce differential tests for eBPF runtimes and Wasm validators; our agent similarly consumes natural-language documentation, but targets rule grounding rather than test generation. Our agent refines *existing* oracles rather than mining from scratch, benefiting from the structure provided by pricing terms.

LLM-based testing at industry scale. Meta's TestGen-LLM [2] uses LLMs to generate unit tests that improve code coverage, deployed at scale on production code; its generate-and-filter pipeline mirrors our accuracy and grounding checks. The accompanying Assured LLMSE position [3] frames this filtering explicitly as the way to deploy LLM-generated software artefacts at scale. Ansible Lightspeed [29] is a companion example of a production LLM code-generation service. Konstantinou et al. [22] show that LLM-generated test oracles tend to capture *actual* rather than *expected* behavior. These systems generate test inputs or assertions for general-purpose code; our system learns eligibility conditions for a domain-specific oracle, with documentation grounding as the additional filter that distinguishes *expected* from merely *accurate-on-the-training-data*.

Code-documentation inconsistency. Zhou et al. [33] detect directive defects in API documentation by comparing code behavior against documentation claims. Our documentation grounding step inverts this direction: we start from learned rules (derived from system behavior) and search for documentation that explains them. Ungrounded rules are documentation gaps, i.e., places where the system's behavior is not described.

7 Conclusion

We presented a neuro-symbolic tool that learns the implicit eligibility rules governing complex itemized bills and grounds each learned rule in public documentation. Once learned, these eligibility rules function as a light-weight abstraction of the actual billing logic and can support several use cases. We use the system to answer common questions, such as *"why did I not receive this discount on my line item?"*, and more general questions like *"Can I stack an RI discount with a Savings Plan?"*. This can be used to accelerate/automate customer support tickets and incident responses.

We also use our system as a regression detection tool: Our billing system is large, and code changes frequently. Our system tries to explain every single line item produced in pre-prod stages. If a new unexplained discrepancy emerges, we have to decide if this is a planned behavior change, if our rules need refinement, or if a code change introduced unexpected behavior.

Finally, we can use the grounding of eligibility rules to test our documentation. If documentation or rules change, we can easily check if all rules are still explainable. We can even use the eligibility rules to synthesize a compact explanation of the (observed) logic of the billing system.

Two observations stand out from our showcase. First, grounding learned rules in public documentation is an effective filter: it rejects accurate-but-vacuous candidates that a pure classifier would have retained, and it produces rules that generalize beyond the training sample. Second, delegating the learning loop to an agent kept our implementation effort minimal. For a pipeline that runs daily on the order of hours, the agent's cost is small relative to the engineering work it replaces.

Broader applicability. The methodology applies wherever pricing terms are explicit, but eligibility conditions are implicit: insurance claims processing, healthcare billing, telecommunications invoicing, and tax computation. The inputs are domain-independent: pricing terms, line items, and a documentation corpus.

Future work. Three directions follow directly from the evaluation. Extending the predicate grammar to cross-line-item conditions (volume tiers spanning charges, cross-hour commitment tracking) would cover eligibility rules the current grammar cannot express. A held-out evaluation across disjoint account sets would stress-test the generalization we measured within a shared pool in RQ3. And embedding the learned rules in a customer-facing support agent, where symbolic rules select the relevant documentation passage and an LLM produces a natural-language explanation of a specific charge, is the payoff the tool was designed to enable.

Disclosure of Generative AI Use

Generative AI is both the subject and an instrument of this work. The rule-learning agent we present is built on Claude Opus 4 via the Kiro CLI harness, as described in §4 and §5.1. The same tools were used heavily in preparing this manuscript: LLM assistants (Claude, via Kiro CLI) drafted and revised text, tables, and figures, and assisted in analyzing and summarizing the experimental logs. All content was reviewed and edited by the authors, who take full responsibility for it; all technical claims, numerical results, and bibliographic references were verified by the authors against primary sources.

Data Availability Statement

All tools and models used in this paper are publicly available: the C4.5 tree learner is open source, and the Kiro CLI agent framework and Claude Opus 4 are commercially available off-the-shelf components. Since our approach operates on billing data, we cannot share concrete line items, even for internal accounts. However, a reader can download line items for their own account via the billing API of their preferred cloud provider in the open-source FOCUS format [15] and apply the methodology directly.

Our approach uses three prompts: one to prepare the knowledge base (which we cannot share because it references internals of our billing system), one to learn eligibility rules (which can be recreated from the algorithm and grammar described in this paper), and one to ground the discovered rules in documentation (which reads the produced rules and compares them against public documentation). We are confident that a motivated reader can reproduce our methodology, but since the underlying agent framework and model will evolve outside of our control, exact reproduction of our specific numerical results is unlikely.

References

[1] Sina Aghaei, Andrés Gómez, and Phebe Vayanos. 2025. Strong Optimal Classification Trees. *Operations Research* 73, 4 (2025), 2223–2241. doi:10.1287/opre.2021.0034

[2] Nadia Alshahwan, Jubin Chheda, Anastasia Finogenova, Beliz Gokkaya, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Automated Unit Test Improvement using Large Language Models at Meta. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering (FSE Companion)*. ACM, 185–196. doi:10.1145/3663529.3663839

[3] Nadia Alshahwan, Mark Harman, Inna Harper, Alexandru Marginean, Shubho Sengupta, and Eddy Wang. 2024. Assured LLM-Based Software Engineering. arXiv preprint arXiv:2402.04380. arXiv:2402.04380 [cs.SE] doi:10.48550/arXiv.2402.04380

[4] Amazon Web Services. 2026. Custom Terms and Pricing — AWS Marketplace. https://aws.amazon.com/marketplace/features/custom-terms-pricing/ Accessed 2026-04-27.

[5] Amazon Web Services. 2026. Enterprise Pricing — AWS Pricing. https://aws.amazon.com/pricing/enterprise/ Accessed 2026-04-27.

[6] Amazon Web Services. 2026. Savings Plans — Amazon Web Services. https://aws.amazon.com/savingsplans/ Accessed 2026-04-27.

[7] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. *Information and Computation* 75, 2 (1987), 87–106. doi:10.1016/0890-5401(87)90052-6

[8] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525. doi:10.1109/TSE.2014.2372785

[9] Dimitris Bertsimas and Jack Dunn. 2017. Optimal Classification Trees. *Machine Learning* 106, 7 (2017), 1039–1082. doi:10.1007/s10994-017-5633-9

[10] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.

[11] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. 2000. Counterexample-Guided Abstraction Refinement. In *CAV (LNCS, Vol. 1855)*. Springer, 154–169. doi:10.1007/10722167_15

[12] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. 2008. DSD-Crasher: A Hybrid Analysis Tool for Bug Finding. *ACM Transactions on Software Engineering and Methodology* 17, 2 (2008), 1–37. doi:10.1145/1348250.1348254

[13] Elizabeth Dinella, Gabriel Ryan, Todd Mytkowicz, and Shuvendu K. Lahiri. 2022. TOGA: A Neural Method for Test Oracle Generation. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22)*. ACM, Pittsburgh, PA, USA, 2130–2141. doi:10.1145/3510003.3510141

[14] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. 2001. Dynamically Discovering Likely Program Invariants to Support Program Evolution. *IEEE Trans. Software Eng.* 27, 2 (2001), 99–123. doi:10.1109/32.908957

[15] FinOps Foundation. 2024. FinOps FOCUS Specification. https://focus.finops.org/

[16] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. 2014. ICE: A Robust Framework for Learning Invariants. In *CAV (LNCS, Vol. 8559)*. Springer, 69–87. doi:10.1007/978-3-319-08867-9_5

[17] Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Commun. ACM* 55, 3 (2012), 40–44. doi:10.1145/2093548.2093564

[18] Soneya Binta Hossain and Matthew B. Dwyer. 2025. TOGLL: Correct and Strong Test Oracle Generation with LLMs. In *Proceedings of the 47th IEEE/ACM International Conference on Software Engineering (ICSE) (ICSE '25)*. IEEE, 1475–1487. doi:10.1109/ICSE55347.2025.00098

[19] Soneya Binta Hossain, Antonio Filieri, Matthew B. Dwyer, Sebastian Elbaum, and Willem Visser. 2023. Neural-Based Test Oracle Generation: A Large-Scale Evaluation and Lessons Learned. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 120–132. doi:10.1145/3611643.3616265

[20] Hieu Huynh, Tri Le, Tu Nguyen, Viet Nguyen, Vu Nguyen, and Tien N. Nguyen. 2026. RBCTest: Leveraging LLMs to Mine and Verify Oracles of API Response Bodies for RESTful API Testing. In *Proc. IEEE/ACM International Conference on Software Engineering (ICSE)*. doi:10.48550/arXiv.2504.17287

[21] Adharsh Kamath, J. Nausheen Mohammed, Aditya Senthilnathan, Saikat Chakraborty, Pantazis Deligiannis, Shuvendu K. Lahiri, Akash Lal, Aseem Rastogi, Subhajit Roy, and Rahul Sharma. 2024. Leveraging LLMs for Program Verification. In *Formal Methods in Computer-Aided Design (FMCAD)*. doi:10.34727/2024/isbn.978-3-85448-065-5_16

[22] Michael Konstantinou, Renzo Degiovanni, and Mike Papadakis. 2024. Do LLMs generate test oracles that capture the actual or the expected program behaviour? *arXiv preprint arXiv:2410.21136* (2024). doi:10.48550/arXiv.2410.21136

[23] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. 2023. CodaMosa: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. 919–931. doi:10.1109/ICSE48619.2023.00085

[24] Karl Meinke and Muddassar A. Sindhu. 2011. Incremental Learning-Based Testing for Reactive Systems. In *Tests and Proofs (TAP) (LNCS)*. Springer, 134–151. doi:10.1007/978-3-642-21768-5_11

[25] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. 1999. Black Box Checking. In *Proceedings of the Joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE/PSTV)*. Kluwer, 225–240. doi:10.1007/978-0-387-35578-8_13

[26] J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.

[27] Daniel Ramos, Hailie Mitchell, Inês Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. 2023. MELT: Mining Effective Lightweight Transformations from Pull Requests. In *Proceedings of the 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1516–1528. doi:10.1109/ASE56229.2023.00117

[28] Nikitha Rao, Elizabeth Gilbert, Harrison Green, Tahina Ramananandro, Nikhil Swamy, Claire Le Goues, and Sarah Fakhoury. 2024. DiffSpec: Differential Testing with LLMs using Natural Language Specifications and Code Artifacts. (2024). arXiv:2410.04249 [cs.SE] doi:10.48550/arXiv.2410.04249

[29] Priyam Sahoo, Saurabh Pujar, Ganesh Nalawade, Richard Gebhardt, Louis Mandel, and Luca Buratti. 2024. Ansible Lightspeed: A Code Generation Service for IT Automation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '24)*. ACM, 2148–2158. doi:10.1145/3691620.3695277

[30] Yuheng Su, Tianjun Bu, Qiusong Yang, Yiwei Ci, and Enyuan Tian. 2026. CIll: CTI-Guided Invariant Generation via LLMs for Model Checking. *arXiv preprint arXiv:2602.23389* (2026). doi:10.48550/arXiv.2602.23389

[31] Frits Vaandrager. 2017. Model Learning. *Commun. ACM* 60, 2 (2017), 86–95. doi:10.1145/2967606

[32] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*. doi:10.48550/arXiv.2210.03629

[33] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald C. Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE, 27–37. doi:10.1109/ICSE.2017.11