Natural-Language Policies to Executable Decisions: An Interpretable Large Language Model Framework

Ziqiang Zhang, Jing Ma, Zilong Wang, Jiayuan Chen, Yi Qiao, Yu He, Wei Zhang, Dai Cheng, Xiaoyu Shen*
Ningbo Institute of Digital Twin, Eastern Institute of Technology, Ningbo
zqzhang@idt.eitech.edu.cn xyshen@eitech.edu.cn

Abstract

Pricing automation in large-scale tourism is challenging because travel orders are highly unstructured, while pricing policies are complex, rapidly evolving, and inherently open-ended. Traditional rule engines are brittle and costly to maintain, whereas unconstrained LLM agents lack the reliability and auditability required for financial decisions. We present a production-grade LLM-powered pricing system with a strict decision boundary: LLMs perform structured extraction and bounded policy/path selection, while all numeric pricing, including total-price computation, is executed deterministically. Policies are compiled into interpretable condition trees, enabling open-ended support for new clauses and evolving rules without code changes, while exposing auditable artifacts for human-in-the-loop control. Periodic fine-tuning on logged traces further improves tree induction and path matching. Deployed at a municipal state-owned tourism enterprise across 7 scenic sites and 12 business categories with 1,500+ operators and 1,000+ active policies, the system processed 3,960 orders in six months, reduced the order management team from 15–20 to 3, and cut per-order handling time from $\sim$10 minutes to $<$ 2 minutes.

1 Introduction

Accurate and timely processing of travel orders is the operational core of modern tourism services. A typical order describes a group's itinerary, including scenic spots, travel dates, agent name, etc, which must be transformed into structured facts and matched with complex repository of pricing policies (Kaushik et al., 2017). While automating this workflow is essential for operational scalability, it remains remarkably difficult in practice due to the unstructured nature of travel requests and the intricate, ever-evolving pricing policies (Zhou et al., 2025; Liu et al., 2025).

In real-world production settings, travel orders exhibit extreme heterogeneity. Requests range from semi-structured digital forms to casual instant-messaging text or even handwritten notices. Critical information is often implicit or ambiguous: scenic spots may be referred to by informal abbreviations, and travel dates frequently rely on background context rather than explicit mention. These factors transform simple data parsing into a high-order information-understanding task that requires reasoning well beyond surface-level patterns (Xu et al., 2019; Mathew et al., 2020).

This input complexity is compounded by the fundamentally *open-ended* nature of pricing policies (Hendrycks et al., 2021). Determining the applicability of a policy requires reasoning over a vast array of interacting conditions, such as seasonal windows, group size thresholds, and bespoke contractual clauses, distributed across heterogeneous policy documents. Because new policies routinely introduce novel conditions that were never anticipated at system design time, traditional software-based solutions that rely on manually encoding logic into structured databases and rule engines (Desmond et al., 2022) are inherently brittle. Each policy update often necessitates synchronized changes to backend schemas, rule code, and user interfaces, creating prohibitive maintenance costs and making city-scale deployment unscalable.

Recent advances in large language models (LLMs) offer a flexible alternative for reasoning over natural language (Su et al., 2022; Achiam et al., 2023; Liu et al., 2024; Xu et al., 2025; Ding et al., 2026). In principle, LLMs can parse informal orders, interpret policy text, and perform cross-document reasoning without rigid schemas. In practice, however, fully autonomous LLMs are ill-suited for high-stakes applications. They often lack stability, provide limited interpretability, and offer no clear mechanism for non-technical users to inspect or correct intermediate reasoning steps,

*Corresponding author.

which is an essential requirement for pricing and auditability (Agarwal et al., 2024).

In this work, we present a production-grade, LLM-powered pricing system that reconciles the linguistic flexibility of large language models with the strict reliability requirements of real-world financial operations. Our central insight is that LLMs should not replace pricing systems, but instead operate within a carefully defined and interpretable decision boundary. The system is designed around two core principles: (1) *Open-ended condition support*: Instead of hard-coding logic, we leverage LLMs to match order-side facts against policy-side condition trees. This design naturally supports an evolving rule space, enabling the system to handle arbitrary ad-hoc rules and seasonal exceptions without code changes or model retraining (Gao et al., 2022; Chen et al., 2022). (2) *End-to-end interpretability*: The system automatically extracts logic from policy documents and organizes it into explicit *condition trees*. These trees preserve the semantics of natural language while exposing their logical structure, allowing tourism managers to review and maintain rules without programming knowledge (Xiong et al., 2024; Wang et al., 2025). Crucially, all numeric computation is handled by a deterministic engine. LLMs are restricted to structured extraction and discrete decision selection. This separation provides the reliability of traditional software systems while retaining the reasoning flexibility of modern language models.

We deployed the system at a municipal, state-owned tourism enterprise operating a city-wide platform spanning seven major scenic areas and twelve business categories, with 1,500+ operators. Over the first six months after rollout, the system processed 3,960 orders, reduced the order-management team from 15–20 to 3, and cut per-order handling time from $\sim$10 minutes to < 2 minutes. The deployment externalizes pricing governance into auditable artifacts, policy-induced condition trees, and logged decision traces, making disagreements actionable via bounded human confirmation and overrides. These traces enable systematic error diagnosis and iterative improvements, without changing numeric execution. An internal survey reports improved communication/efficiency (96.92%) and positive feedback on usability and analyzable rules (76.92%).

While our deployment focuses on tourism, the challenges of open-ended policy logic, informal inputs, strict correctness requirements, and human-centered governance are common across many domains such as insurance underwriting, compliance checking, and contract execution. By eliminating rigid rule engineering while preserving human agency, our system offers a practical path toward scalable, trustworthy AI in policy-driven industries.

2 Background

The Pricing Lifecycle in Tourism Operational pricing in modern tourism follows a recurring tripartite lifecycle: (i) *policy onboarding*, (ii) *order intake and quotation*, and (iii) *settlement and verification*. The process begins when business managers publish pricing policies—documents (often PDFs or spreadsheets) that define the rules for various customer segments and travel seasons. Once published, tour coordinators receive travel orders and must match them against the active policy repository to generate a quote. Finally, the calculated price must be verified against the original policy to ensure auditability and financial compliance.

Open-Ended Complexity of Pricing Policies A core challenge in this domain is that pricing conditions are fundamentally *open-ended*. Policies are defined over *resources* (atomic sellable units like a cable car ride) and *products* (bundles of resources). However, the conditions governing their price, ranging from specific age brackets and group size thresholds to complex seasonal overlaps and ad-hoc contractual clauses, are virtually unlimited. New policies frequently introduce entirely novel logic that was not anticipated during initial system design. This evolving complexity means that "pricing logic" is not a static set of parameters, but a growing library of natural-language rules that must be interpreted in context.

Informality and Mismatch in Travel Orders The intake side of the lifecycle is equally challenging due to the extreme informality of travel orders. In our deployment, 97% of orders (based on 2025H2 production logs) are received as instant-messaging screenshots rather than structured digital forms. These requests mix travel dates, destinations, and casual notes without a fixed template. Furthermore, a "resource mismatch" is common: the itinerary described in the initial text often deviates from the final executed plan. Nearly half of our production cases require manual edits to the item list before a quotation can be finalized, necessitating a system that can tolerate incomplete

descriptions and mid-process human correction.

The Inadaptability of Traditional Rule Engines Traditional software solutions are ill-equipped for this environment because they rely on rigid, predefined schemas. Since pricing conditions are unlimited, adding a new policy often requires more than just data entry; it necessitates a coordinated update of the *database structure* (to store new attributes), the *rule logic* (to handle new predicates), and the *input UI* (to allow users to select these new options). This "hard-coding" cycle creates a massive maintenance burden, as software developers must constantly translate natural-language nuances into executable code. For large tourism sites with hundreds of evolving policies, the delay and cost of these manual system updates make traditional rule engines fundamentally unscalable.

Implications for System Design These constraints dictate that a production system cannot rely on a "closed-world" assumption. Instead, the system must: (i) decouple policy logic from the underlying software schema to handle an open-ended condition space, and (ii) externalize its reasoning into *interpretable intermediate artifacts* that preserve natural-language semantics and treat human intervention as a first-class operation.

3 Methodology

We design an auditable pricing pipeline that operationalizes informal travel orders and natural-language pricing policies under strict governance constraints. The central design principle is a *strict LLMs decision boundary*: LLMs are restricted to structured information extraction and discrete decision selection, while all numeric calculations are performed by a deterministic engine. This boundary structurally eliminates numeric hallucination, makes remaining uncertainty explicit, and enables rapid human verification. The executable tree schema and the validator suite (Appendix A), a realistic tree instance (Appendix B), and end-to-end UI traces that externalize L1–L3 confirmations and overrides (Appendix C).Before publication, every induced tree must pass an explicit validator suite (Table 5), and any failure is blocked from entering execution and routed to manual correction. The system is decomposed into two major components: a *policy onboarding module* (stages *A0–A3*) and a *price calculation module* (stages *B0–D*). To ensure reliability, we incorporate a tiered human-in-the-loop override process (*L1–L3*) that allows operators to intervene at distinct levels of the reasoning chain. These identifiers correspond to the architectural components illustrated in Figure 1.

3.1 Policy Onboarding and Tree Induction

The policy processing module is the "offline" stage where policy documents are transformed into *condition trees*. A condition tree acts as the sole executable interface for pricing a specific product. In this structure, each internal node represents a natural-language applicability condition, while each root-to-leaf path corresponds to exactly one *price specification*. This *one-path–one-price-spec* invariant guarantees that once a path is selected, the deterministic calculation. The induction of these trees follows a four-step pipeline:

A0: Text extraction. We first convert policy documents into text via OCR or table extraction, producing a normalized textual input. OCR is treated here as an upstream text-acquisition step rather than a solved reasoning component. Raw OCR output never directly becomes executable pricing logic: downstream validator checks, editable intermediate artifacts, and operator confirmation/overrides bound residual extraction risk before execution.

A1: Identify resources. The LLM identifies referenced resources/products in the policy text and grounds them into the catalog R, outputting a set of `resource_ids`. Catalog grounding acts as a hard constraint that prevents out-of-catalog entities from entering the executable interface.The grounding catalog R is maintained as a first-class operator artifact (Fig. 6) and serves as the only executable namespace for resources.

A2: Extract nodes. For each grounded entity, the LLM extracts three types of information: (i) policy metadata, such as customer scope and validity windows; (ii) the natural-language conditions that define the logic of the policy; and (iii) the leaf-level *price specifications*. Crucially, the LLM treats numeric values as static fields to be stored, rather than performing any calculations at this stage.

A3: Assemble & validate. The extracted clauses are assembled into a condition tree while enforcing the *one-path–one-price-spec* invariant. A validator suite then checks the tree for schema completeness, unit compatibility, and numeric sanity (e.g., non-negativity). The system also performs conflict detection to ensure that no two paths under the same customer scope and validity window overlap. Only validated trees are published to the *Tree Store*;

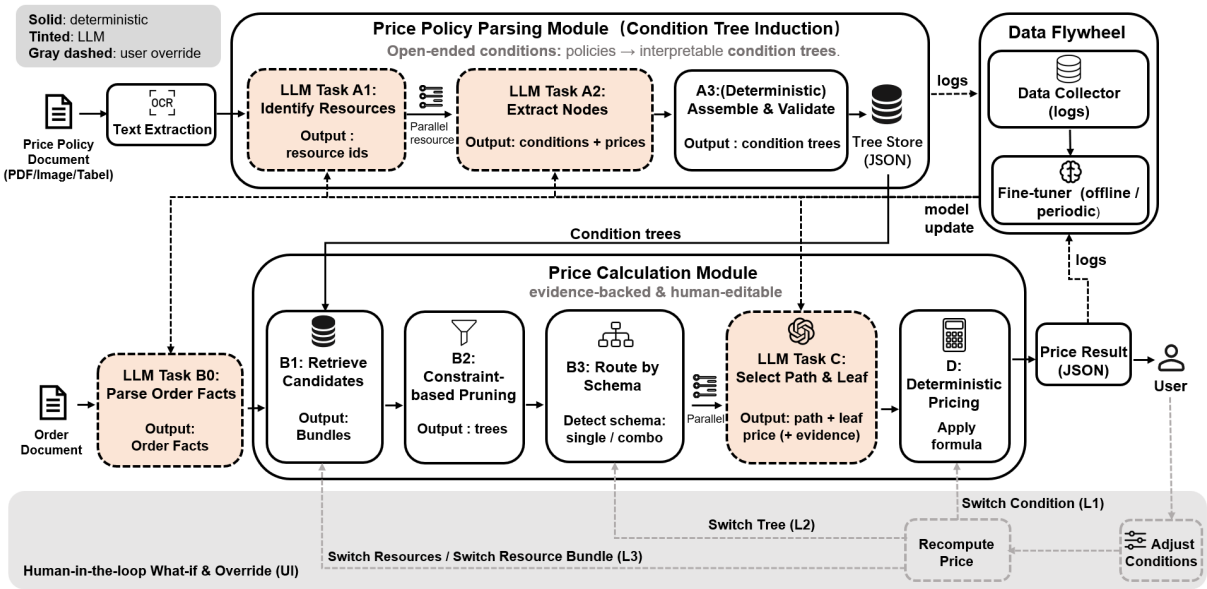

Figure 1: System overview. The policy parsing module induces interpretable condition trees from pricing documents, and the price calculation module routes orders to candidate trees, verifies applicability, and computes final prices with deterministic execution and human-in-the-loop overrides.

those that fail are routed to humans for correction. Induced trees are reviewed and edited in an operator console before publication (Fig. 7), making extraction errors observable and correctable. The complete executable tree schema (including leaf-level pricing specs, applicability constraints, and UI metadata for rendering/review) is provided in Appendix A with the full field list in Table 4.

Accordingly, policy onboarding should be understood as a validator-gated, human-confirmed publication workflow rather than raw autonomous tree generation. The production target is the operator-confirmed executable tree published to the Tree Store, not the unreviewed LLM draft.

3.2 Order Parsing and Price Calculation

Once policies are onboarded as condition trees, the price calculation module processes incoming orders through the following stages:

B0: Order Parsing and Fact Extraction Incoming orders (typically screenshots) are parsed by an LLM into *OrderFacts* (*B0*). This structured record contains non-resource fields, such as travel dates, group size, and customer identity, and grounded resource mentions.OrderFacts are surfaced in the operator console as an editable, structured form to correct missing/ambiguous fields before downstream routing and execution (Fig. 9).

B1: Candidate Bundle Enumeration Since orders may mention individual resources that are sold

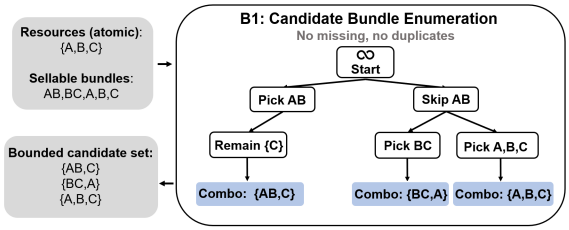

Figure 2: **B1: Candidate bundle enumeration.** Given atomic resource mentions and a catalog of sellable bundles, we deterministically enumerate all feasible bundle decompositions that exactly exhaust the requested resources for downstream tree routing and path selection.

as *products* (bundles), the system must determine the best way to group these resources. We use a deterministic DFS backtracking algorithm to enumerate all feasible bundle combinations that exactly exhaust the requested resources; Figure 2 gives a toy illustration.

B2–B3: Retrieval and Routing For each candidate bundle/product enumerated in B1, we retrieve its associated condition trees from the *Tree Store* and deterministically filter/prune them using policy metadata constraints, including: (i) overlap between the order date range and the policy validity window, and (ii) match between the order-side customer scope and the policy applicability scope. After obtaining a bounded set of viable trees per candidate bundle/product, we further route candidates deterministically by a predefined pricing schema

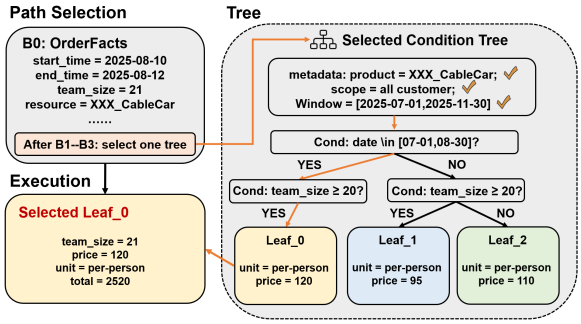

Figure 3: **Toy walkthrough of tree-guided pricing.** The LLM extracts *OrderFacts* (B0). B1–B3 deterministically route the order to a *bounded* policy tree via metadata filtering. The LLM then performs *discrete* root-to-leaf path selection (C) and returns a leaf-level *PriceSpec*, while the deterministic engine computes the final amount from validated quantities (D). A production-faithful condition tree (with metadata constraints and operator-visible traces) is provided in Appendix B.

(e.g., *single-item* vs. *bundle* structures) to select the appropriate matching policy trees.

C–D: Path Selection and Execution As shown in Figure 3, in the final reasoning step, an LLM analyzes the filtered trees and selects the root-to-leaf path whose conditions are satisfied by the *OrderFacts* (*C*). To ensure transparency, the model provides evidence pointers to specific spans in the order text. Once a path is confirmed, the deterministic engine executes the final pricing (*D*) by applying the quantity normalization and units defined in the selected *PriceSpec*. The PriceResult view externalizes the selected candidate tree (L2) and matched path/leaf (L1) together with resource-level breakdown and recomputable totals, enabling auditable confirmation and bounded overrides within the exposed candidate set (Fig. 12).

3.3 Governance and Self-Evolving

Override Protocol To maintain zero-tolerance for numeric errors, we expose the system's reasoning through three override levels: (1) *L1 (Path Override):* The operator selects a different branch within a fixed tree, correcting LLM errors of natural-language conditions (Fig. 12). (2) *L2 (Tree Override):* The operator switches between valid policy trees, typically used to resolve overlapping policy conflicts (Fig. 12). (3) *L3 (Composition Override):* The operator corrects the underlying data, either by editing the resource list (*L3-a*) to fix extraction errors or by switching the bundle decomposition (*L3-b*) to reflect changes in the actual itinerary (figs. 9 to 11).

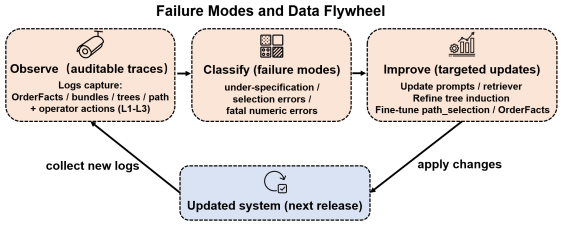

Figure 4: **Failure-mode taxonomy and data flywheel.**

Failure Modes and Data Flywheel Production failures are categorized into three types: *under-specification* (incomplete order info), *selection errors* (incorrect path choice), and *fatal numeric errors*. As shown in Figure 4, our architecture is designed to make the first two types recoverable via the *L1–L3* protocols, while the third is prevented by our strict decision boundary. Finally, we maintain a *production data flywheel* by logging raw model outputs alongside human-confirmed overrides. This allows for iterative improvement of extraction and selection prompts without risking the stability of the deterministic execution logic.

4 Experiment

We deploy our system on a city-level state-owned tourism enterprise. This enterprise is responsible for managing seven major scenic areas across the city, encompassing 12 diverse tourism business categories across land, water, and air dimensions (including accommodation, dining, car rental, attraction visits, shopping, entertainment, tour guide services, performances, cruise ships, conferencing, cable cars, drones, etc.), with over 1,500 operational staff, complex organizational structure, and diverse role distributions. We evaluate our system on production logs from 2025H2 (Jul–Dec 2025). The dataset comprises 3,960 orders, of which 3,842 (97.0%) are instant-messaging screenshots processed via OCR and 118 (3.0%) are Word uploads. We focus on screenshot orders as the dominant deployment modality. Table 1 summarizes order structural complexity: over 70% of orders involve multiple resources, and 25% contain bundle products, motivating the deterministic composition enumeration in stage B1.

Evaluation Protocol We organize evaluation as a decision-boundary funnel that progressively isolates each decision level, so that downstream metrics reflect only the residual decision. In Track 1, we quantify operator workload on all 3,842 screen-

Category	Count	%
Single-resource orders	1,178	29.7
Multi-resource orders	2,782	70.3
2 resources	1,858	46.9
3+ resources	924	23.4
Orders with bundle products	990	25.0

Table 1: Order structural complexity (2025H2, $n = 3{,}960$).

shot orders. We distinguish two mutually exclusive L3 interventions: L3-a (resource edit) addresses mismatch between the extracted resource set and the actually executed itinerary, while L3-b (bundle/product switch) addresses composition mismatch when the resource set is already consistent. After removing all L3 cases, we measure L2 (tree switching) on the remaining orders. In Track 2, we isolate within-tree path selection (L1) and pricing correctness. From the 1,785 orders that survive L2 and L3 filtering, we further restrict to an information-consistent subset where critical pricing fields (travel dates and team size) are consistent with logged outcomes, yielding 1,349 orders covering 2,598 matched product instances. All correctness metrics are reported on this subset. We report three metrics aligned with production requirements. First, workload is measured by the frequency of L3-a, L3-b, and L2 interventions across the full screenshot set. Second, unit-price accuracy is the fraction of product instances whose predicted unit price matches any logged ground-truth price under the same order-product key. Third, the fatal numeric error rate captures any output with invalid units, aggregation mismatch, or prices unsupported by a verified leaf specification.

Our formal experiments focus on downstream deployment reliability after publication, because unpublished drafts never enter execution. We therefore do not report a standalone offline accuracy metric for raw tree induction in this paper. Instead, onboarding quality is controlled by catalog grounding, the A3 validator suite, and operator review before publication, while logged published trees and operator corrections are used for periodic improvement of A1–A2.

Results Table 2 presents the complete decision-boundary funnel. On the full 3,842 screenshot orders, 1,842 (47.9%) require L3-a resource edits and 108 (2.8%) require L3-b bundle/product switching. The high L3-a rate reflects a known operational reality: coordinators frequently omit or revise resources after submission, so this is not a system error but a characteristic of the domain workflow. Among the 1,892 orders requiring no composition-level intervention, only 107 (5.7%) require L2 tree switching, indicating that metadata-based routing resolves the vast majority of policy-selection decisions automatically. On the 1,349 information-consistent orders (2,598 product instances), the system achieves zero fatal numeric errors, confirming that the strict LLM decision boundary—where the model selects discrete paths rather than generating prices—structurally eliminates numeric hallucination. Unit-price accuracy reaches 85.2%; the remaining 14.8% are L1 path-selection errors that operators can recover via a single UI click rather than requiring any change to the execution layer.

Decision boundary	Count	Rate
Track 1: Production workload (n=3,842)		
L3-a resource edits	1,842	47.9%
L3-b bundle/product switch	108	2.8%
No L3 intervention	1,892	—
L2 tree switching	107	5.7%
No L2/L3 intervention	1,785	—
Track 2: Correctness (n=2,598 product inst.)		
Unit correctness	2,598 / 2,598	100%
Unit-price accuracy (L1)	2,214 / 2,598	85.2%
Fatal numeric error	0 / 2,598	0%

Table 2: Decision-boundary funnel. Track 1 reports operator workload on the full screenshot set; Track 2 reports correctness on the information-consistent subset where only L1 remains.

Metric	Before	After
Policy onboarding time	∼20 min	∼1 min
Order processing time	manual	∼2 min
Fatal numeric errors	—	0%
Developer involvement	required	not required

Table 3: Operational impact before and after deployment.

Operational Impact Table 3 summarizes key deployment metrics from the same 2025H2 six-month deployment window. The most significant operational gain is the elimination of developer involvement in policy onboarding. Previously, each new pricing policy required an expert developer to translate natural-language conditions into executable rules, taking approximately twenty minutes per policy. With our system, non-technical tourism managers can review and publish condition trees directly, reducing onboarding time to roughly one minute and eliminating the communi-

cation overhead between business and engineering teams. Order processing time is also reduced from approximately ten minutes to under two minutes on average, covering the full pipeline from OCR extraction through pricing execution and operator confirmation.

5 Conclusion

We present a production LLM framework that converts natural-language pricing policies into executable, interpretable condition trees and processes informal travel orders through a deterministic pricing pipeline. By strictly separating LLM-based discrete selection from deterministic numeric execution, the system achieves zero fatal pricing errors across thousands of production orders, while reducing policy onboarding time from twenty minutes to one minute without developer involvement. In a six-month deployment at a municipal, state-owned tourism enterprise, the system processed 3,960 orders and delivered substantial operational gains, demonstrating that LLMs can be safely integrated into high-stakes decision workflows when paired with structured, auditable reasoning interfaces. The framework generalizes to other policy-driven domains where interpretability, reliability, and non-expert usability are equally critical.

Limitations

Our current deployment relies on a manual grouping strategy that merges policies with similar customer scope and validity patterns into shared condition trees, in order to limit the number of trees per product. As the system runs longer and policies accumulate, this grouping may degrade, increasing matching ambiguity and operator burden. Developing automatic tree merging and splitting strategies is a natural direction for future work.

Nearly half of production orders (47.9%) require L3-a resource edits, reflecting the domain-specific gap between the resources stated in the order and the itinerary actually executed. While the system surfaces this mismatch for operator correction, reducing this rate—through richer order input formats or multi-turn clarification with coordinators—would further lower the human workload.

Our correctness metrics in Track 2 are computed on an information-consistent subset of 1,349 orders where composition and tree selection are pre-resolved. End-to-end accuracy on the full order set, including undecidable cases where the input is genuinely ambiguous, remains difficult to evaluate without additional ground-truth annotations. Furthermore, all experiments are conducted at a single tourism deployment site. Although the architecture is designed to be domain-agnostic, empirical validation in other policy-driven domains such as insurance or regulatory compliance is needed to confirm generalizability.

In addition, we do not yet provide an independently annotated offline benchmark for standalone evaluation of raw tree induction. Although published tree versions and operator edits are logged for iterative improvement, constructing a rigorous benchmark for the onboarding stage remains future work.

References

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, and 1 others. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

Chirag Agarwal, Sree Harsha Tanneru, and Himabindu Lakkaraju. 2024. Faithfulness vs. plausibility: On the (un)reliability of explanations from large language models. *ArXiv*, abs/2402.04614.

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. 2022. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Trans. Mach. Learn. Res.*, 2023.

Michael Desmond, Evelyn Duesterwald, Vatche Isahagian, and Vinod Muthusamy. 2022. A no-code low-code paradigm for authoring business automations using natural language. *ArXiv*, abs/2207.10648.

Longwei Ding, Anhao Zhao, Fanghua Ye, Ziyang Chen, and Xiaoyu Shen. 2026. From llms to lrms: Rethinking pruning for reasoning-centric models. *arXiv preprint arXiv:2601.18091*.

Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. 2022. Pal: Program-aided language models. *ArXiv*, abs/2211.10435.

Dan Hendrycks, Collin Burns, Anya Chen, and Spencer Ball. 2021. Cuad: An expert-annotated nlp dataset for legal contract review. *ArXiv*, abs/2103.06268.

Divyansh Kaushik, Shashank Gupta, Chakradhar Raju, R. A. Dias, and Sanjib Ghosh. 2017. Making travel smarter: Extracting travel information from email itineraries using named entity recognition. In *Recent Advances in Natural Language Processing*.

Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, and 1 others. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437*.

Yunuo Liu, D. Zhu, Zena Al-Khalili, Dai Cheng, Yanjun Chen, Dietrich Klakow, Wei Zhang, and Xiaoyu Shen. 2025. Pricinglogic: Evaluating llms reasoning on complex tourism pricing tasks. *ArXiv*, abs/2510.12409.

Minesh Mathew, Dimosthenis Karatzas, R. Manmatha, and C. V. Jawahar. 2020. Docvqa: A dataset for vqa on document images. *2021 IEEE Winter Conference on Applications of Computer Vision (WACV)*, pages 2199–2208.

Hui Su, Xiao Zhou, Houjin Yu, Xiaoyu Shen, Yuwen Chen, Zilin Zhu, Yang Yu, and Jie Zhou. 2022. Welm: A well-read pre-trained language model for chinese. *arXiv preprint arXiv:2209.10372*.

Yafang Wang, Yangjie Tian, Xiaoyu Shen, Gaoyang Zhang, Jiaze Sun, He Zhang, Ruohua Xu, and Feng Zhao. 2025. Fault2flow: An alphaevolve-optimized human-in-the-loop multi-agent system for fault-to-workflow automation. *arXiv preprint arXiv:2511.12916*.

Sichao Xiong, Yigit Ihlamur, Fuat Alican, and Aaron Ontoyin Yin. 2024. Gptree: Towards explainable decision-making via llm-powered decision trees. *arXiv preprint arXiv:2411.08257*.

Fengli Xu, Qianyue Hao, Zefang Zong, Jingwei Wang, Yunke Zhang, Jingyi Wang, Xiaochong Lan, Jiahui Gong, Tianjian Ouyang, Fanjin Meng, and 1 others. 2025. Towards large reasoning models: A survey of reinforced reasoning with large language models. *arXiv preprint arXiv:2501.09686*.

Yiheng Xu, Minghao Li, Lei Cui, Shaohan Huang, Furu Wei, and Ming Zhou. 2019. Layoutlm: Pre-training of text and layout for document image understanding. *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*.

Ruiwen Zhou, Wenyue Hua, Liangming Pan, Sitao Cheng, Xiaobao Wu, En Yu, and William Yang Wang. 2025. Rulearena: A benchmark for rule-guided reasoning with llms in real-world scenarios. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 550–572.

A Executable Policy Schema

A.1 Full Field List

Table 4 lists all fields used in the condition-tree node representation. Each node carries identification and type information; leaf nodes additionally store pricing specifications. Non-leaf nodes encode branching conditions in natural language via `fieldValue`, while the tree structure is maintained through `children` references.

Field	Type	Description
`id`	int	Unique node identifier
`fieldName`	str	Node category (e.g., product type, condition)
`fieldValue`	str	Category value (product name or condition clause)
`children`	array	Child nodes; empty array for leaves
`isLeafNode`	bool	Whether this node is a leaf
Leaf-only fields (pricing specification)		
`price`	number	Unit price
`unit`	str/null	Billing unit (e.g., per person, per group)
Constraint and UI fields		
`limitValue`	object/null	Structured applicability constraints
`limit`	str	Reserved for future use
`expanded`	bool	UI expansion state (default: true)

Table 4: Full schema of condition-tree nodes. Fields are grouped by function: identification and structure (top), leaf pricing specification (middle), and constraints/UI metadata (bottom).

A.2 Constraint Structure (`limitValue`)

The `limitValue` field encodes applicability constraints as a nested boolean expression. The top level specifies a logical relation (AND/OR) over one or more *groups*. Each group in turn carries its own relation and a list of atomic conditions, where each atomic condition is a triple of (type, operator, value). For example, a constraint requiring "group size ≥ 20 AND travel duration ≤ 3 days" is represented as a single group with AND relation containing two atomic conditions. This two-level nesting (groups of conditions) is expressive enough to cover all policy constraints encountered in our production deployment while remaining simple to validate and render in the operator UI.

A.3 Validator Codes

During policy onboarding, every induced condition tree must pass a suite of validators before being published to the Tree Store. Table 5 lists the validator codes and their purposes. Trees that fail any validator are routed to manual review rather than silently published, ensuring that only structurally sound and semantically consistent trees enter the executable pipeline.

Validator	Purpose
`schema_complete`	All required fields present; tree is well-formed
`unit_compat`	Billing units are consistent and compatible across sibling leaves
`numeric_sanity`	Prices are non-negative and within plausible bounds
`conflict_detect`	No contradictory rules under overlapping metadata (scope, validity)
`service_name`	Resource/service names resolve to valid catalog entries
`restrict_format`	`limitValue` structure conforms to the two-level boolean schema

Table 5: Validators applied during policy onboarding. A tree is published only if all validators pass.

B Condition Tree Example

Figure 5 illustrates a simplified condition tree induced from a real pricing policy. The root node identifies the product; internal nodes represent natural-language conditions (seasonality and group-size thresholds); each leaf stores an executable price specification. During order processing, the system traverses from root to leaf by evaluating conditions against extracted OrderFacts, and the selected leaf's price specification is passed to deterministic execution. Operators can inspect this tree in the UI and override the selected path if needed.

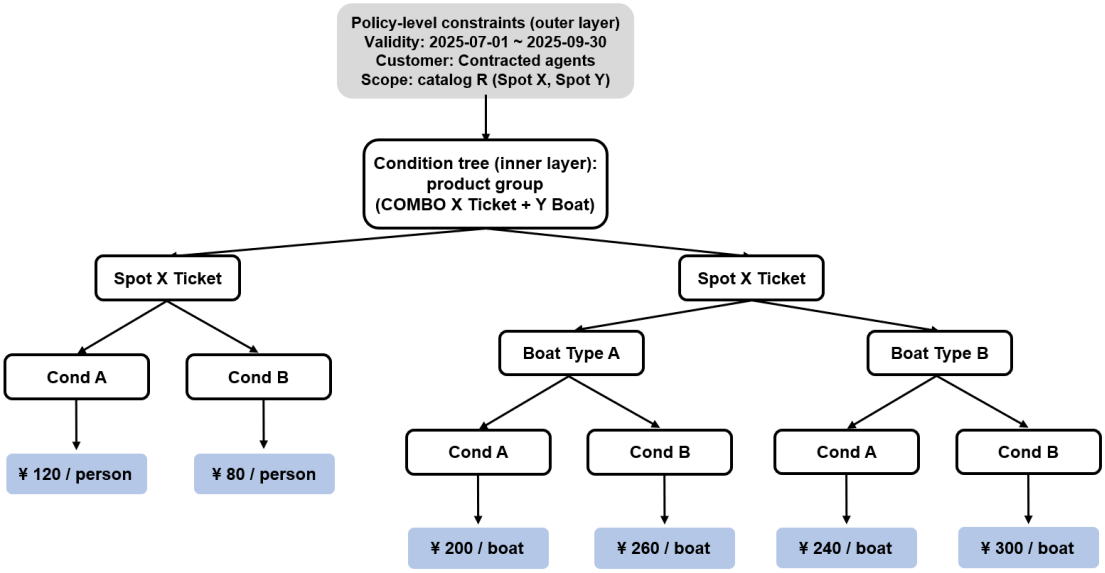

Figure 5: A realistic induced policy structure with an outer metadata layer and an arbitrarily nested condition tree. The inner tree branches into two resource-specific subtrees (Spot X Ticket and Spot Y Boat).

C Operator UI for Auditable Overrides

This appendix presents the operator-facing UI that externalizes intermediate artifacts and supports bounded, auditable overrides. We group screenshots by the two core modules: (i) *price policy parsing* (A1–A3), which grounds executable entities in the catalog R and induces/reviews condition trees before publication; and (ii) *price calculation*, which exposes candidate trees and paths for L1/L2 confirmation, and supports localized L3 overrides without changing deterministic numeric execution.

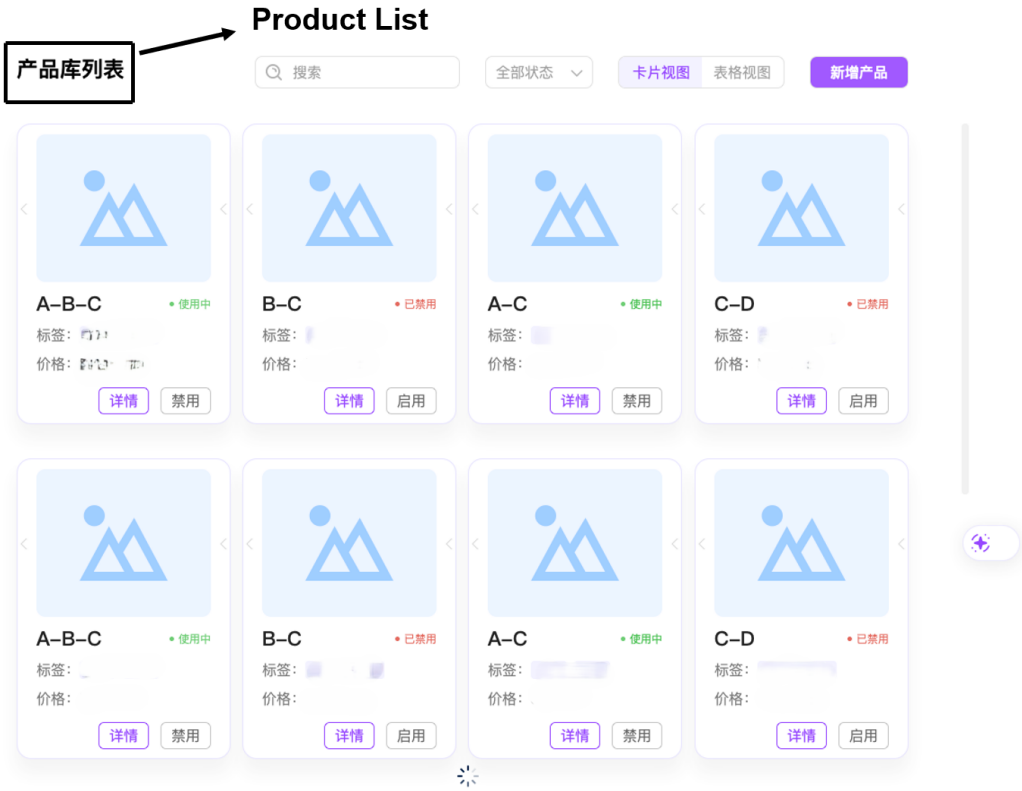

Figure 6: Catalog management UI defining the resource space R (A1).

Figure 7: Policy parsing/editing UI for inducing and reviewing condition trees before publication (A2–A3).

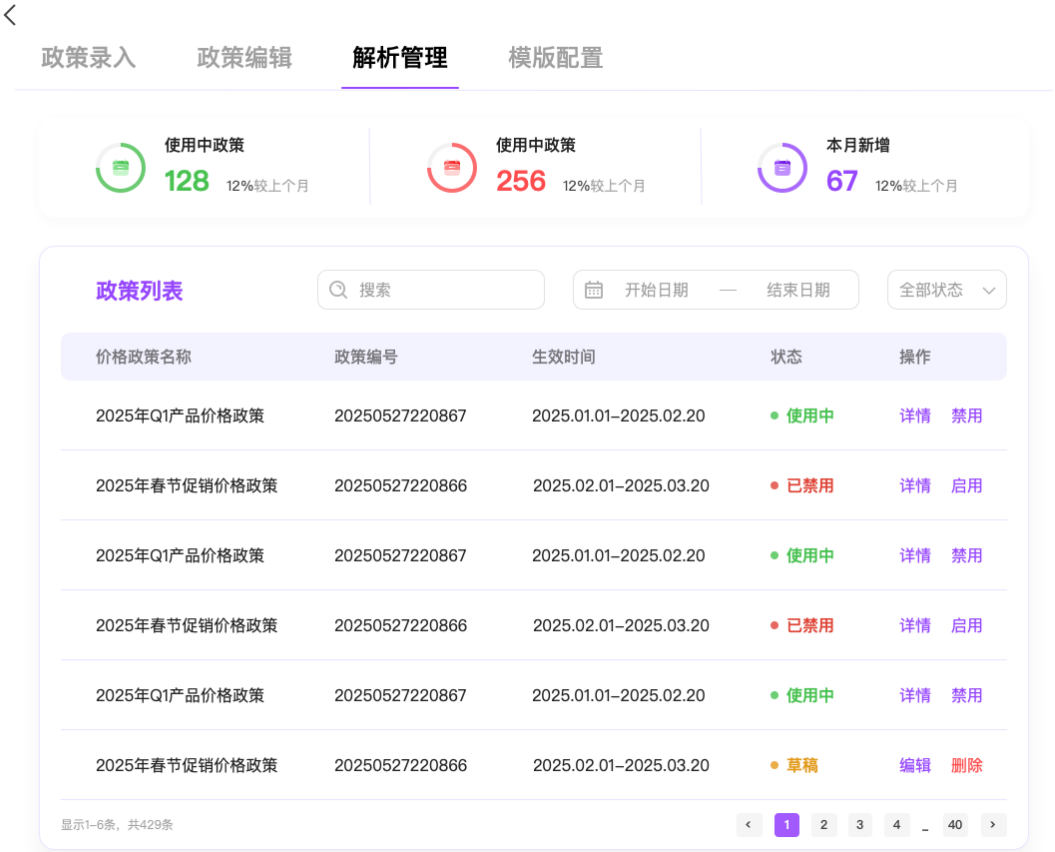

Figure 8: Policy management UI for validity windows and status (metadata routing).

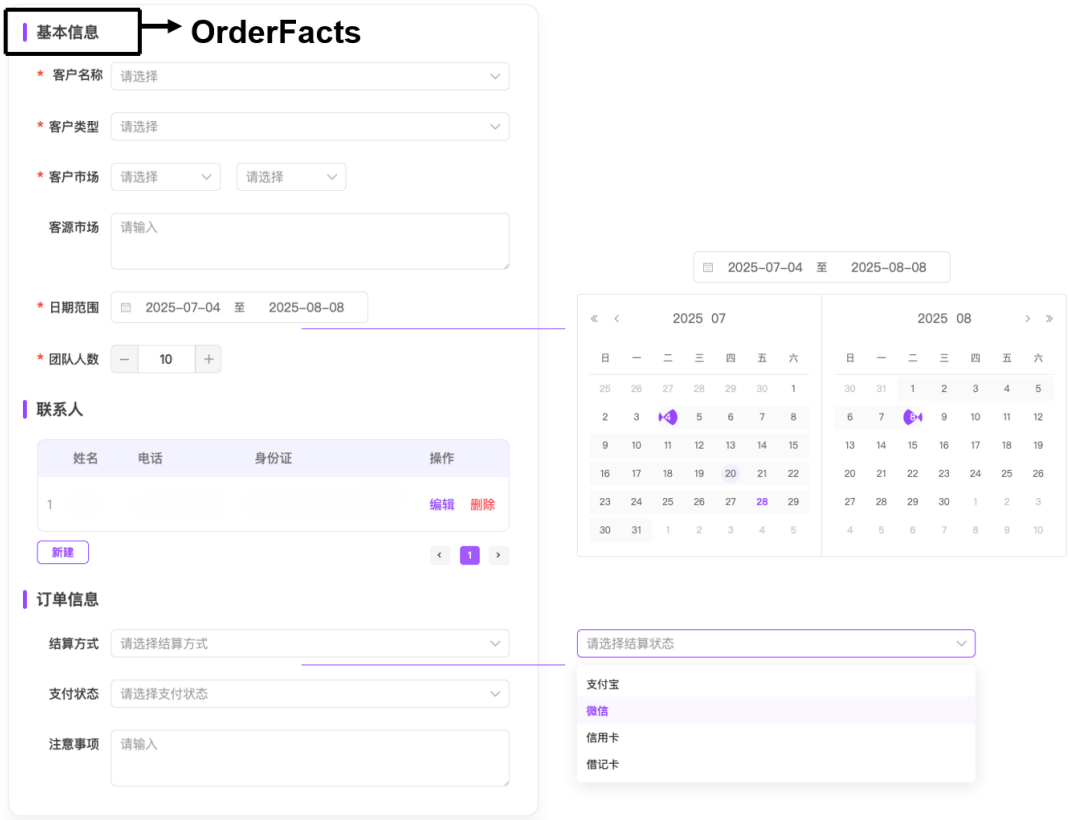

Figure 9: OrderFacts panel for order intake. Extracted structured fields are fully visible to operators and can be edited before downstream bundle enumeration and policy matching (B0).

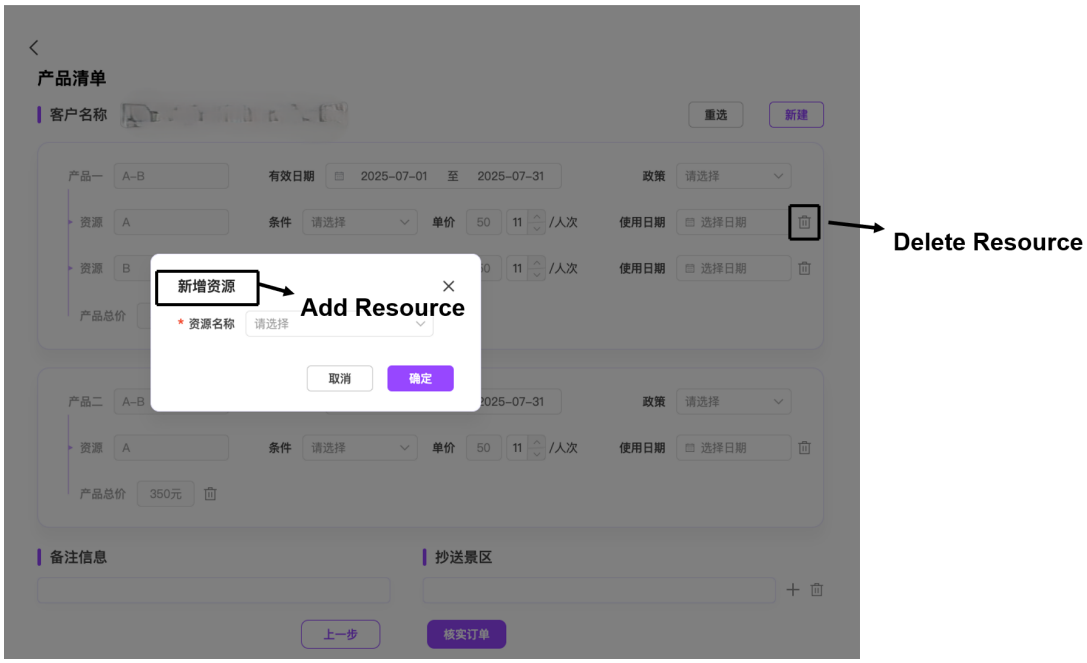

Figure 10: Add/remove resource UI (L3-a resource edit).

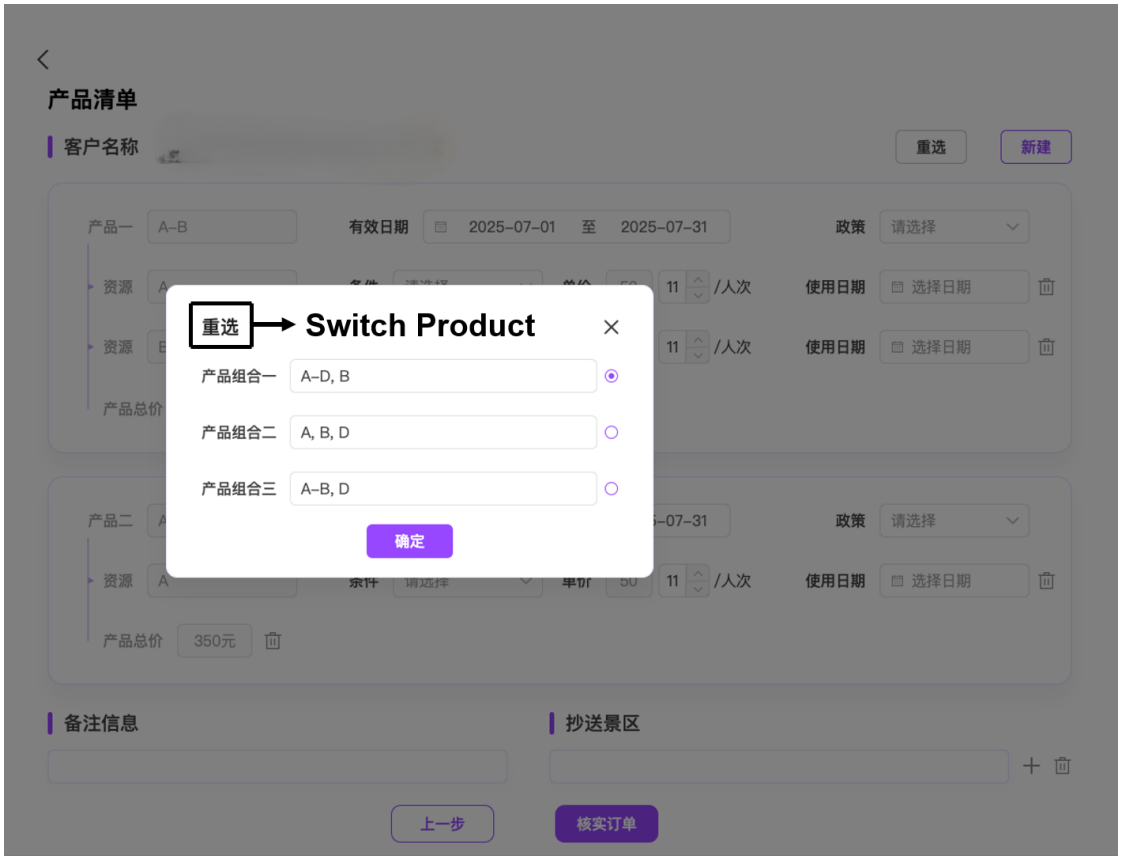

Figure 11: Bundle/product switching UI (L3-b composition override).

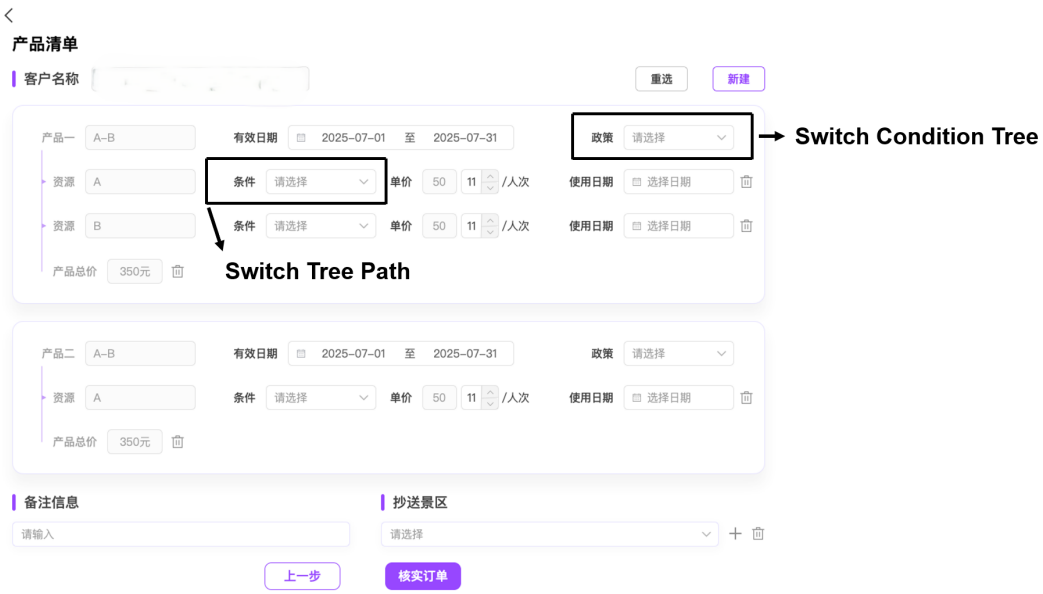

Figure 12: PriceResult view. Each item shows the selected tree (L2) and matched path/leaf (L1) with resource-level breakdown and recomputable totals; operators can switch the candidate tree (L2) or candidate path (L1) within the exposed set.