MIND: MARKET INTERPRETATION DSL FOR UNIFIED MARKET DESIGN AND SIMULATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Market mechanisms such as auctions and matchings coordinate supply and demand at scale, yet their implementations remain locked in rigid procedural code that hinders iteration and auditing. We introduce the Market Interpretation DSL (MIND), a typed language and toolchain for declarative market specification to achieve unified market design and simulation. MIND comprises (i) a core grammar with a phased Intermediate Representation (IR) and economic safety checks, (ii) a natural language assistant that translates descriptions into DSL with automated diagnostics and safe rewrites, and (iii) rule-based simulation and convex optimization backends. Using synthetic specifications generated across 87 domains with held-out validation, our fine-tuned Llama-3-8B assistant achieves 96.33% semantic correctness, measured as IR equivalence to gold programs, surpassing few-shot GPT-4o at 91.41%. Across second-price auctions, multi-stage auctions, and matching markets, MIND reduces specification complexity by approximately 79% in lines of code compared to Python implementations. In a preregistered within-subjects study with 17 participants, mechanism modifications were completed 4 to 10 times faster using MIND. Code, dataset, and models will be released upon acceptance.

1 INTRODUCTION

Market mechanisms such as auctions and matching markets form the backbone of modern economics, digital platforms, and decentralized systems. They coordinate supply and demand, reduce transaction costs, and enable efficient allocation of scarce resources (Milgrom, 2004; Roth, 2018; Milgrom, 2021). Despite this centrality, practical modeling and implementation remain cumbersome. Most platforms and simulators still hard-code allocation rules, matching logic, and pricing routines into procedural code, creating a lossy translation from policy to code (Calheiros et al., 2011; Byrd et al., 2019). This limits experimentation and complicates verification of market properties.

Beyond performance, platform operators must ensure transparent rules and reproducible outcomes for regulatory compliance, requiring a chain from human-readable policies to executable logic with audit trails. Decentralized trading systems raise the bar further: mechanism logic executes on-chain and requires formal checks for correctness and economic safety (d'Eon et al., 2024; Bouaicha et al., 2025). The core limitation is the absence of a unified interface that bridges conceptual specifications to deterministic implementations with support for governance, testing, and audit.

Recent LLM-based approaches to mechanism automation yield non-deterministic outputs and brittle patches, making debugging difficult where fairness and correctness are paramount. They also struggle to bridge underspecified natural language and verbose implementations, frequently omitting crucial details such as reserve prices, tie-breaking rules, or budget constraints. Moreover, the primary users are economists and policy analysts who possess domain expertise but typically lack programming skills. While GUI-based tools exist, they cannot express conditional constraints or multi-stage interactions, reducing mechanisms to rigid templates.

We advocate a domain-specific language paired with natural language translation that separates authoring, validation, and execution. A compact, typed DSL makes specifications legible, enables static checks for economic consistency, and supports deterministic compilation to multiple backends for cross-validation. This creates a governance surface where specifications carry provenance, version identifiers, and audit traces, while validators enforce safety gates before deployment. The

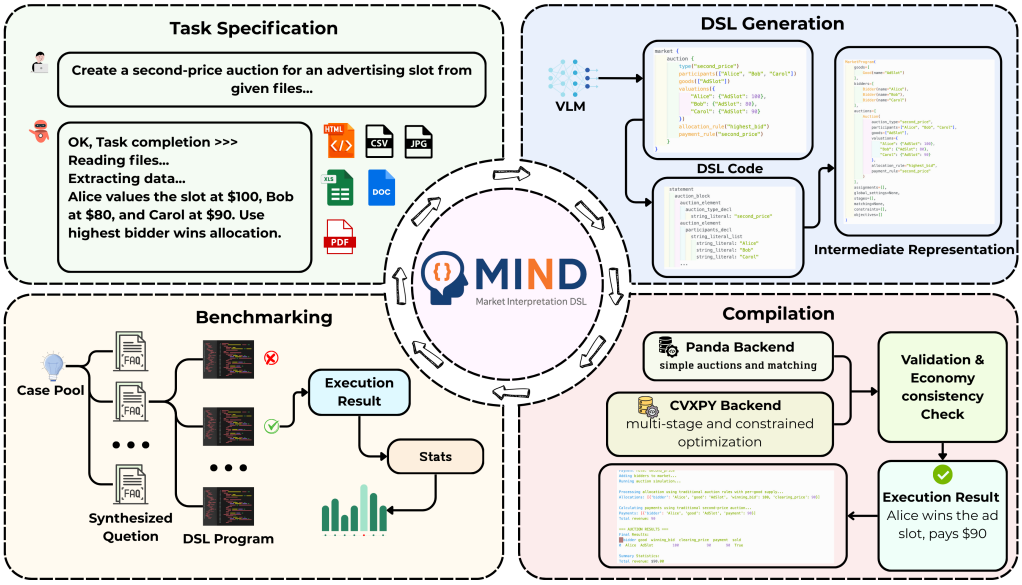

Figure 1: The whole workflow of our system. First, the Completeness Agent helps users with complete task descriptions. Then, the Copilot generates the MIND programs. Finally, the compiler executes the programs with appropriate backends to produce the final results.

system serves both audiences: domain experts use natural language to generate initial specifications, while power users can directly edit DSL programs. In MIND, semantic correctness is evaluated as Intermediate Representation (IR) equivalence to reference programs, and specification complexity is measured in AST nodes.

In this paper, we present MIND (Market Interpretation DSL), a comprehensive and extensible language and toolchain for specifying, validating, and executing market mechanisms. MIND defines a grammar of objects, actions, and types, compiled into a phased IR which is automatically validated and compiled into executable simulations, thereby decoupling specification from execution. The system includes a natural language assistant that translates descriptions into DSL programs, provides diagnostics, and applies safe rewrites (Zhang et al., 2023). Two backends support rule-based simulation and convex optimization. Each specification is a versioned artifact with machine-checkable metadata. To further enhance usability, we develop a Copilot system that translates natural-language descriptions into DSL programs, offers warnings and suggestions, and applies safe auto-fixes (Zhang et al., 2023).

We evaluate MIND along three axes. First, a workflow study shows a 79% reduction in specification complexity versus Python. Second, a natural language to DSL study finds that a fine-tuned Llama-3-8B model achieves 96.33% semantic correctness. Third, case studies demonstrate auditable policy updates through validator reports and change logs.

Our contributions are threefold:

- We introduce MIND, a domain-specific language with formal grammar and a phased IR that bridges natural language to executable simulations while creating auditable specifications.
- We develop an execution framework with two backends that support rule-based simulation and convex optimization from unified specifications, enabling deterministic cross-validation for compliance.
- We build a natural language to DSL translation system, achieving 96.33% semantic correctness across 87 domains, serving both non-programmers and power users.

2 Related Work

Market mechanism DSLs. Prior mechanism modeling in economics has largely relied on general-purpose programming or specialized simulators, making specification and validation cumbersome. Recent work explores domain-specific languages to capture auction rules, negotiation

games, or fair division protocols in symbolic form (Hoseindoost et al., 2024; De Jonge & Zhang, 2021; Bertram et al., 2023). CoorERE (Hoseindoost et al., 2024) provides an executable DSL for auction-based coordination in crisis response, reducing development effort by nearly half, but it addresses single-item auctions without cross-mechanism support. GDL has been repurposed as a unifying description language for negotiation domains (De Jonge & Zhang, 2021), enabling generic solvers, but it lacks intermediate representations with economic validation. Slice (Bertram et al., 2023) defines a DSL for fair division protocols with automated envy-freeness verification, yet remains limited to division problems without auction or matching support. These DSLs improve mechanism specification but are scoped to individual subdomains and do not provide staged validation, two execution backends, or governance artifacts that MIND includes.

LLMs in mechanism design. The rise of large language models has motivated new approaches to automating specification and simulation. Recent studies use LLMs to generate valuations, bidding policies, and to propose new auction formats (Duetting et al., 2024; Sun et al., 2024; Dubey et al., 2024b; Shah et al., 2025). LaMP-Val (Sun et al., 2024) uses GPT-4 to infer personalized valuations from text and fine-tunes smaller models as strategic agents. Dubey et al. (Dubey et al., 2024b) and Duetting et al. (Duetting et al., 2024) examine auctions where advertisers bid for influence over LLM outputs, proposing incentive-compatible rules for token-level allocation. Shah et al. (Shah et al., 2025) show GPT-4 agents can reproduce human-like bidding behaviors, suggesting LLMs can serve as synthetic participants. These approaches demonstrate potential for synthesis and simulation but operate without typed specifications, deterministic compilation, or audit trails. They generate code directly without an intermediate representation, making systematic verification and governance difficult. They often lack empirical validation of generated mechanisms against ground truth specifications.

Unified frameworks and positioning. Prior DSLs achieve domain-specific expressiveness and LLM approaches enable automation, yet the literature remains fragmented: CoorERE focuses on crisis response, Slice on fair division, and LLM methods typically lack formal specifications. Technical barriers to unification include incompatible type systems across auction and matching domains, the absence of staged validation for economic properties, and limited support for governance requirements such as provenance tracking and policy diffs. MIND addresses these gaps through a unified grammar spanning auctions, matchings, and exchanges; an intermediate representation with three-stage validation (parsing, typing, economic consistency); two execution backends for simulation and optimization that scale to thousands of participants; natural language translation achieving 96.33% semantic correctness on 87 domains; and governance artifacts including versioning, validator reports, and audit logs. This combination links formal specification, property verification, and agent-based evaluation in a single reproducible workflow. Our evaluation shows it reduces specification complexity by 79% while maintaining semantic accuracy comparable to hand-written implementations.

3 METHOD

As illustrated in Figure 1, our system provides an end-to-end pipeline for generating, validating, and simulating MIND, starting from a user specification. The architecture is composed of several key parts: (1) a symbolic DSL for formal representation (Shi et al., 2024; Borum & Seidl, 2022), (2) an Intermediate Representation (IR) with a robust validation system, (3) a two-backend framework for code generation, and (4) an AI-powered toolchain including a dataset generation pipeline, a completeness agent, and a fine-tuned Copilot.

3.1 MARKET INTERPRETATION DSL

The foundation of our system is Market Interpretation DSL (MIND), a formal language designed for the specification of market rules. The language's grammar is built on a clear separation of core concepts: (i) **Objects** are entities that constitute a market, such as auction, participants, goods, and matching; (ii) **Actions** are operations that define the market's behavior, such as specifying the auction type, defining valuations, or setting constraints; (iii) **Types** are specific variants of objects and actions, like `type("second_price")` or `type("first_price")`. Some DSL examples are shown in Figure 2.

AUCTION DSL SAMPLE

```
market {
    auction {
        type("second_price")
        participants(["Alice", "Bob", "Carol"])
        goods(["AdSlot"])
        valuations({
            "Alice": {"AdSlot": 100},
            "Bob": {"AdSlot": 80},
            "Carol": {"AdSlot": 90}
        })
        allocation_rule("highest_bid")
        payment_rule("second_price")
    }
}
```

BIPARTITE MATCHING DSL SAMPLE

```
market {
    matching {
        type("bipartite")
        participants(["Alice", "Bob", "Carol", "General", "Cardiac", "Neuro"])
        compatibility_graph({
            "Alice": ["General", "Cardiac"],
            "Bob": ["Cardiac", "Neuro"],
            "Carol": ["General", "Neuro"],
            "General": ["Alice", "Carol"],
            "Cardiac": ["Alice", "Bob"],
            "Neuro": ["Bob", "Carol"]
        })
        matching_rule("stable_matching")
    }
}
```

Figure 2: Two MIND specifications. Left: second-price auction where `type` specifies auction type, `participants` lists bidders, `goods` declares the item, `valuations` gives each bidder's valuation, `allocation_rule()` assigns to the highest bidder, and `payment_rule()` charges the second-highest bid. Right: simple matching market specification.

3.2 Intermediate Representation (IR) and Validation

A challenge in designing a language with multiple execution targets is preventing our language parser from getting entangled with the specific details of every execution backend (Pandas, CVXPY, etc.). This creates a brittle, unscalable system where adding a new backend or modifying the DSL syntax would require cascading changes across the entire codebase.

To solve this problem, we introduce an Intermediate Representation (IR) (Lattner et al., 2021) as a critical abstraction layer. The IR is a typed abstract syntax tree (AST) over market constructs (e.g., `AuctionNode`, `ConstraintNode`). The parser translates DSL source into IR only; code generators read IR only. This separation ensures modularity and maintainability.

To ensure that any market specified in the DSL is not just syntactically correct, but also semantically and economically sound, the IR undergoes a rigorous validation process before generating code. This process consists of three phases: parsing, typing, and economic consistency. Three main validators run in order on the IR:

1. **CoreMarketValidator:** Performs fundamental checks, ensuring names are unique, references are valid, valuations align with participants, and auction rules are recognized.
2. **StageAndMatchingValidator:** If the design uses stages or matching, this validator runs to perform checks on global settings and validate the structure of these advanced components.
3. **AdvancedOptimizationValidator:** If the design includes constraints or objectives, this validator checks that their types are recognized and parameters are valid (e.g., non-negative budgets).

Each validator consumes an IR snapshot and emits a `ValidationReport` with typed findings (`error`, `warning`, `autofixable`). The Autofixer applies only safe rewrites; if a required rule cannot be inferred, it emits a blocking error rather than altering semantics. We persist the spec hash, validator report identifier, and compile artifact path with the run logs to enable exact reconstruction during audit. All experiments log these identifiers, allowing any reported result to be traced to its exact specification and validator state.

3.3 Two-Backend Code Generation

In practice, one execution engine cannot serve all market designs well. Simple single-shot auctions and matching markets benefit from fast, table-driven simulation, while constrained or combinatorial designs need solver-grade optimization. To handle both without exposing backend complexity to users, we compile the same backend-independent IR into different execution targets via `MarketCompiler`.

Backend routing. `MarketCompiler` selects a backend by inspecting IR features: designs without explicit objectives or global constraints are routed to simulation; designs that declare objectives

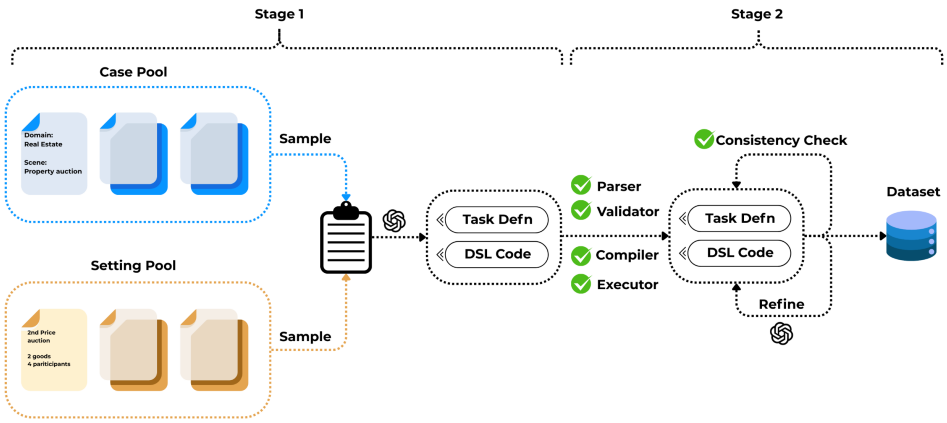

Figure 3: Market Mechanism Dataset pipeline with three phases: data generation, augmentation, and filtering.

or feasibility constraints are routed to optimization. Routing is a pure function of IR features and contains no heuristic randomness.

- **Pandas Backend:** A lightweight, simulation-based backend for standard auctions and matching markets, optimized for speed and simplicity.
- **CVXPY Backend:** An optimization-based backend using the CVXPY library (Diamond & Boyd, 2016), automatically selected for scenarios involving constraints (e.g., budget balance) or objectives (e.g., maximizing revenue).

Backends implement the same IR semantics; observable outcomes (allocations, payments, feasibility flags) must agree for identical IR inputs. This two-backend approach lets users scale from simple simulations to constrained optimizations without changing the DSL specification, while preserving consistent semantics across backends.

3.4 Dataset Generation

Supervised fine-tuning of a specialized Copilot requires a large-scale, high-quality dataset of (Natural Language Description, DSL) pairs. To the best of our knowledge, there is no such dataset for the task of translating natural language specifications into a formal DSL for market mechanisms. To address this, we developed an automated pipeline (Ratner et al., 2017; Northcutt et al., 2021) to use LLMs to generate synthetic data as illustrated in Figure 3. The process begins by programmatically generating diverse prompts for a generator LLM (GPT-4o (Hurst et al., 2024)). To achieve this, we predefine over 800 possible market use cases within 87 domains. For each use case, we randomly sample settings to generate prompts with the formal DSL grammar and in-context examples.

To ensure correctness, every DSL program undergoes a rigorous 4-stage validation pipeline (parsing, typing, economic consistency, execution): it must successfully parse against the grammar, pass semantic and economic validation, compile to an executable backend, and execute without runtime errors. This multi-stage process guarantees that every DSL sample is syntactically, semantically, and functionally correct.

After guaranteeing code correctness, we refine the corresponding natural language descriptions. Each validated DSL program is passed to an LLM to generate a more detailed and complete description. As a final quality control step, another verifier LLM performs a description-DSL consistency check, confirming semantic alignment between the enhanced natural language description and the DSL code. Only pairs passing this final verification are included in the dataset.

3.4.1 Human Audit Protocol

To ensure the quality of our automated pipeline, we perform a manual human audit. We drew a simple random sample of 100 (description, DSL) pairs from the final, post-filter dataset after the 4-stage validation and description-DSL consistency check, stratified by domain and mechanism type. Two independent raters not involved in data generation evaluated each pair on: (i) syntactic correctness (DSL parses under the grammar), (ii) semantic alignment (IR equivalence of the compiled DSL to the behavior described), and (iii) functional executability (successful backend compilation and execution). Raters were blinded to generator identity and pipeline metadata during evaluation.

A pair passes only if all three criteria are satisfied. Disagreements were resolved by adjudication with a third reviewer. The audit confirms a very high accuracy rate (98/100) under the criteria in Section 3.4.1. Audit sheets record the sampled dataset indices and corresponding spec hashes to support reconstruction.

3.5 Completeness Agent

To ensure the generated DSL program fully aligns with users' expectations, we need descriptions with sufficient detail. Since users rarely provide complete descriptions initially, we developed a Completeness Agent (Yao et al., 2023; Shinn et al., 2023) as a pre-processor to help users provide enough information for the Copilot.

The agent operates through a multi-node scheme defining key elements to capture. For each node, it extracts information from users' prompts and populates the scheme with required and optional fields. If all required fields are fulfilled, the node completes and the agent proceeds. Otherwise, it requests missing critical information. After passing all nodes, the agent outputs a *completion schema* and a *complete task specification* that are passed to the Copilot. The completion schema maps directly to the Copilot input fields (participants, goods, constraints, objectives), ensuring the prompt is structurally complete before translation. This significantly increases the likelihood of generating a valid and executable DSL program on the first attempt.

3.6 Copilot Finetuning

The core of our natural language interface is the Copilot, an AI assistant fine-tuned for NL-to-DSL translation. We used a Llama-3-8B-Instruct model (Dubey et al., 2024a) as our base and applied LoRA (Hu et al., 2022) with rank $r = 32$ for efficient training. The model was trained on our curated dataset using a standard supervised fine-tuning (SFT) objective to maximize the conditional probability of generating the ground-truth DSL program given the natural language description. The training loss is:

$$\mathcal{L}(\theta) = -\sum_{(X_i, Y_i) \in \mathcal{D}} \log P(Y_i | X_i; \theta) \tag{1}$$

Here, $\mathcal{D}$ is the set of (description X_i, DSL sequence Y_i) pairs; Y_i is tokenized as a left-to-right sequence for teacher forcing under the SFT objective in Eq. 1.

4 Experiments

To validate our system, we designed two primary experiments. First, we evaluate the Market Interpretation DSL itself by comparing its workflow and expressiveness against standard procedural programming approaches for market design. Second, we quantitatively evaluate the performance of our fine-tuned AI Copilot in translating natural language specifications into valid and semantically correct DSL code.

4.1 DSL Workflow Evaluation

The primary motivation for creating a Domain-Specific Language is to accelerate development, reduce errors, and improve clarity over general-purpose programming. This experiment quantifies these benefits by comparing the implementation of common market design tasks in MIND versus alternative workflows.

4.1.1 Methodology and Benchmark Tasks

We selected three representative market design problems of increasing complexity to serve as benchmarks:

- **Task 1: Standard Second-Price Auction:** A canonical sealed-bid auction where the highest bidder wins but pays the price of the second-highest bid.
- **Task 2: Multi-Stage Auction with Reserve Price:** A sequential process where unsold goods from an initial auction are re-auctioned in a subsequent stage.
- **Task 3: Compatibility Matching Market:** A two-sided matching market where participants can only be matched if they are compatible.

We implemented each task using three distinct approaches:

Table 1: A comparison of implementation workflows for common market design tasks. Our declarative approach, MIND, consistently requires the least specification effort, offers the highest clarity for verification, and provides the greatest flexibility for experimentation.

Task	Approach	Specification Complexity	Readability & Verifiability	Flexibility (Effort to Modify)
Second-Price Auction	MIND	∼10 lines of DSL	High: Declarative economic syntax.	Trivial: Change a single keyword.
	Python	∼40-60 lines of code	Low: Core logic is embedded in code.	Moderate: Requires rewriting functions.
	AnyLogic	∼15-20 graphical steps	Medium: Logic is distributed across agents.	High: Requires reconfiguration.
Multi-Stage Auction	MIND	∼20-25 lines of DSL	High: Staging logic is explicit and easy to follow.	Trivial: Modify a self-contained stage block.
	Python	∼100-120 lines of code	Low: State management is complex and error-prone.	High: Requires significant refactoring of the main control flow.
	AnyLogic	∼25-35 graphical steps	Low: Managing agent state across stages is hard.	High: Requires a full redesign of the simulation flowchart.
Matching Market	MIND	∼15 lines of DSL	High: The compatibility graph is a direct input.	Low: Change the data directly.
	Python (w/ NetworkX)	∼60-80 lines of code	Medium: Requires graph library expertise to understand.	Moderate: Requires implementing a different matching algorithm.
	AnyLogic	∼35-45 graphical steps	Medium: Requires defining custom agent interaction rules.	High: Requires creating new agent protocols.

- **MIND:** Using our proposed symbolic language to declaratively specify market rules.
- **General-Purpose Language (Python):** Writing procedural code from scratch using standard Python libraries (NumPy, Pandas) to define data structures, implement allocation and payment logic, and run the simulation.
- **General Simulation Platform (AnyLogic):** Using a multi-method simulation software (Borshchev & Filippov, 2004)to model the market through visual, agent-based modeling with graphical interfaces and Java code.

Our evaluation compares these approaches based on three key criteria: Specification Complexity, Readability & Verifiability, and Flexibility. We define these based on established concepts in software engineering. Specification Complexity refers to the effort required to define the mechanism, measured by the number of distinct modeling steps and Source Lines of Code (LoC) Molnar & Motogna (2020). Readability & Verifiability is how easy the implementation can be audited against its theoretical design, a crucial aspect of model correctness Alawad et al. (2019). Finally, Flexibility measures the effort required to modify an existing mechanism, such as changing an auction's pricing rule, which is a key indicator of software maintainability Ardito et al. (2020).

4.1.2 Results and Analysis

The results of our workflow comparison, summarized in Table 1, demonstrate the significant advantages of the DSL-based approach. For all tasks, MIND provided the most concise, readable, and adaptable method for specifying the market. Its declarative syntax allows designers to focus on the economic rules rather than the implementation details of simulation logic. In contrast, the Python approach required significant boilerplate code and embedded the core mechanism logic within procedural control flow, making it difficult to verify and modify. While powerful, AnyLogic introduced a high degree of complexity and a steep learning curve, making it not that suited for the rapid prototyping of mechanism rules, which is a primary goal of our system. In this case, we can confirm that MIND successfully bridges the gap between the conceptual design of a market and its executable simulation.

4.2 COPILOT GENERATION EVALUATION

This experiment evaluates the ability of our AI Copilot to automatically generate high-quality Market Interpretation DSL code from natural language descriptions.

4.2.1 EXPERIMENTAL SETUP

We evaluated our fine-tuned MarketCopilot (Llama-3-8B + LoRA) against several baseline models on a held-out test set of 300 examples across 87 domains, ensuring no overlap with training data beyond a 0.85 cosine similarity threshold computed on TF-IDF representations. We additionally exclude near-duplicates by AST hash to prevent leakage. The training set consisted of 11,000 examples, with 10% used for validation during hyperparameter tuning.

To provide a robust comparison, baseline models were evaluated in a few-shot setting with the formal DSL grammar specification and 4 examples of (NL, DSL) pairs along with the task description. In contrast, our MarketCopilot operates zero-shot, taking only the natural language task description as input.

4.2.2 EVALUATION PIPELINE AND METRICS

We employed a rigorous multi-stage pipeline to assess correctness:

1. **Grammar Validation:** Each output is parsed using our Lark EBNF grammar. We measure Parse Success Rate as the percentage of syntactically valid programs.
2. **Semantic Validation:** Syntactically correct programs undergo three checks:
   - *Validator check:* Tests logical consistency using the three-phase validation (parsing, typing, economic consistency)
   - *Compiler check:* Verifies code generation to the two execution backends
   - *IR Semantic Equivalence:* Compares the generated IR to ground truth using graph isomorphism on the AST
3. **Execution Validation:** Programs are executed on 300 test scenarios to verify they produce correct market outcomes (allocations, payments, feasibility). Scenarios mirror the functional spec used in the workflow study.

Our primary metric is **End-to-End Correctness**: the percentage of generations that pass all validation stages and are semantically equivalent to the reference solution. End-to-End Correctness equals the proportion of generations that pass grammar, validation, compilation to both backends, IR semantic equivalence, and execution checks.

4.2.3 RESULTS AND DISCUSSION

Table 2 presents performance metrics. Results are averaged over 5 seeds with fixed prompts; decoding settings are held constant across methods where applicable. Statistical significance was assessed using bootstrap resampling with 1000 iterations.

Table 2: Performance of the AI Copilot against baseline LLMs on the NL-to-DSL generation task. Scores are percentages (%). Our method is highlighted in bold.

Model	Parse Success	Validation + Compilation Success	IR Equivalence
Proprietary LLMs			
GPT-4o-mini	97.01	95.68	90.42
GPT-4o (Hurst et al., 2024)	97.67	97.34	91.41
Open-Source LLMs			
Llama-3-8B	80.76	73.71	60.89
Qwen3-Coder-30B(Yang et al., 2025)	95.51	94.55	81.73
Our Method (Llama-3 + LoRA)	**100.00**	**100.00**	**96.33**

Our fine-tuned model achieves the highest scores across all metrics, with 96.33% end-to-end correctness significantly outperforming the best baseline GPT-4o at 91.41%. Error analysis reveals that baselines frequently fail on: (i) constraint specification, (ii) multi-stage coordination, and (iii) payment rule semantics. Error categories follow our audit taxonomy: SYNTAX, IR_MISMATCH, ECON_CONSISTENCY, EXECUTION, DESC_ALIGN. Their high parse success but lower semantic correctness indicates they generate syntactically plausible but semantically incorrect programs.

5 Ablation Studies

To quantify the impact of our data curation process, we conduct ablation studies on progressively less-filtered versions of our training dataset: (1) **Parse-Only**: filtered only for syntactic correctness; (2) **No Execute Check**: filtered through compilation (Parse → Validator → Compiler) but without execution-time validation; (3) **No LLM Check**: passes full 4-stage validation (Parse → Validator → Compiler → Execute) but lacks the description-DSL consistency verification. We train separate MarketCopilot models on each dataset variant using identical architectures, training budgets, and hyperparameters.

Table 3: Ablation study results demonstrating the impact of each data curation stage. All models trained with identical architectures and budgets.

Model	**Parse Success** (%)	**Validation + Compilation** Success (%)	**IR Equivalence** (%)	**ΔIR vs Previous** (pp)
Parse-Only	98.33	81.33	66.33	–
w/o Execute Check	98.67	92.33	68.67	+2.34
w/o LLM Check	100.00	99.33	72.33	+3.66
Full Pipeline	**100.00**	**100.00**	**96.33**	**+24.00**

Table 3 demonstrates that each curation stage contributes significantly to final performance. While parse success remains uniformly high (>98%) across all variants—indicating that learning basic DSL syntax is straightforward—the gaps emerge in semantic correctness. Validation and compilation success improves from 81.33% to 100% as filtering stages are added, with the execution check contributing 7.00 percentage points and validator checks contributing 11.00 percentage points from the Parse-Only baseline.

Most critically, IR equivalence shows dramatic improvement: from 66.33% (Parse-Only) to 96.33% (Full Pipeline), a total gain of 30.00 percentage points. The description-DSL consistency check alone contributes 24.00 percentage points (72.33% to 96.33%), highlighting that alignment between natural language and formal specifications is crucial for semantic correctness. Without this final verification, models generate syntactically valid but semantically incorrect programs—they learn surface patterns rather than the underlying mapping between economic concepts and their formal representations.

These results validate our design choice to prioritize data quality over quantity. Training on carefully curated examples produces models that understand the semantic correspondence between natural language descriptions and market mechanisms, rather than merely mimicking syntactic patterns.

6 Conclusion and Future Work

We present Market Interpretation DSL (MIND), a symbolic language and AI-powered toolchain bridging economic design and executable implementation. MIND combines a declarative DSL with phased validation, a dual-backend framework for simulation and optimization, and a natural-language-to-DSL translator, reducing specification complexity by 79% and achieving 96.33% semantic correctness. Experiments across auctions, multi-stage markets, and matching mechanisms demonstrate improved verifiability, with governance artifacts—spec hashes, validator reports, and audit logs—ensuring traceability for compliance and on-chain deployment. By decoupling authoring, validation, and execution, MIND facilitates designers' focus on economic properties, accelerating applications in domains such as spectrum allocation and carbon credits.

There remain opportunities to further enhance MIND, also our future work. MIND can be extended to support combinatorial auctions, iterative mechanisms, and multi-turn refinement, which could broaden applicability to more complex markets and further facilitate users. We are also working on scaling the two-backend architecture to millions of participants to enable deployment in large operational settings such as retail electricity markets. Additionally, we plan to incorporate stochastic valuations and Bayesian games to expand MIND's modeling scope and support richer economic analysis. With these future efforts, we hope MIND can further facilitate users in exploring and implementing market designs.

Reproducibility statement

Our work is committed to the principles of open and reproducible research. To this end, all code, datasets, and experimental configurations will be made publicly available upon acceptance of this paper.

References

Huda Alawad, Ramesh Panta, Minhaz Zibran, and Mohammad Amin Al Islam. An empirical study of the relationships between code readability and software complexity. In *2019 IEEE 27th International Conference on Program Comprehension (ICPC)*, pp. 108–118. IEEE, 2019. doi: 10.1109/ICPC.2019.00023.

Luca Ardito, Rosella Coppola, Lorenzo Barbato, and Daniele Verga. A tool-based perspective on software code maintainability metrics: A systematic literature review. *Software: Practice and Experience*, 50(12):2203–2230, 2020. doi: 10.1002/spe.2876.

N Bertram, A Levinson, and J Hsu. Cutting the cake: a language for fair division. corr abs/2304.04642 (2023), 2023.

Andrei Borshchev and Alexei Filippov. From system dynamics and discrete event to practical agent based modeling: reasons, techniques, tools. 2004.

Holger Stadel Borum and Christoph Seidl. Survey of established practices in the life cycle of domain-specific languages. In *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems*, pp. 266–277, 2022.

MA Bouaicha, G Destefanis, T Montanaro, N Lasla, and L Patrono. Shill bidding prevention in decentralized auctions using smart contracts. *Information Sciences*, pp. 122374, 2025.

David Byrd, Maria Hybinette, and Tucker Hybinette Balch. Abides: Towards high-fidelity market simulation for ai research. *arXiv preprint arXiv:1904.12066*, 2019. URL `https://arxiv.org/abs/1904.12066`.

Rodrigo N. Calheiros, Rajiv Ranjan, Anton Beloglazov, César A. F. De Rose, and Rajkumar Buyya. Cloudsim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011. doi: 10.1002/spe.995. URL `https://onlinelibrary.wiley.com/doi/10.1002/spe.995`.

Dave De Jonge and Dongmo Zhang. Gdl as a unifying domain description language for declarative automated negotiation. *Autonomous Agents and Multi-Agent Systems*, 35(1):13, 2021.

Greg d'Eon, Neil Newman, and Kevin Leyton-Brown. Understanding iterative combinatorial auction designs via multi-agent reinforcement learning. In *Proceedings of the 25th ACM Conference on Economics and Computation*, pp. 1102–1130, 2024.

Steven Diamond and Stephen Boyd. Cvxpy: A python-embedded modeling language for convex optimization. *Journal of Machine Learning Research*, 17(83):1–5, 2016.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv e-prints*, pp. arXiv–2407, 2024a.

Avinava Dubey, Zhe Feng, Rahul Kidambi, Aranyak Mehta, and Di Wang. Auctions with llm summaries. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pp. 713–722, 2024b.

Paul Duetting, Vahab Mirrokni, Renato Paes Leme, Haifeng Xu, and Song Zuo. Mechanism design for large language models. In *Proceedings of the ACM Web Conference 2024*, pp. 144–155, 2024.

Samaneh Hoseindoost, Afsaneh Fatemi, and Bahman Zamani. An executable domain-specific modeling language for simulating organizational auction-based coordination strategies for crisis response. *Simulation Modelling Practice and Theory*, 131:102880, 2024.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. Gpt-4o system card. *arXiv:2410.21276*, 2024.

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14. IEEE, 2021.

Paul R. Milgrom. *Putting Auction Theory to Work*. Cambridge University Press, Cambridge, 2004. doi: 10.1017/CBO9780511813825. URL https://www.cambridge.org/core/books/putting-auction-theory-to-work/63FA2A2D332E9E3A3238B6D5B650F2A5.

Paul R. Milgrom. Auction research evolving: Theorems and market designs. *American Economic Review*, 111(5):1383–1405, 2021. doi: 10.1257/aer.111.5.1383. URL https://www.aeaweb.org/articles?id=10.1257/aer.111.5.1383.

Andreea Molnar and Simona Motogna. Longitudinal evaluation of open-source software maintainability. *arXiv preprint arXiv:2003.00447*, 2020. URL https://arxiv.org/abs/2003.00447.

Curtis Northcutt, Lu Jiang, and Isaac Chuang. Confident learning: Estimating uncertainty in dataset labels. *Journal of Artificial Intelligence Research*, 70:1373–1411, 2021.

Alexander Ratner, Stephen H Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. Snorkel: Rapid training data creation with weak supervision. In *Proceedings of the VLDB endowment. International conference on very large data bases*, volume 11, pp. 269, 2017.

Alvin E. Roth. Marketplaces, markets, and market design. *American Economic Review*, 108 (7):1609–1658, 2018. doi: 10.1257/aer.108.7.1609. URL https://www.aeaweb.org/articles?id=10.1257/aer.108.7.1609.

Anand Shah, Kehang Zhu, Yanchen Jiang, Jeffrey G Wang, Arif K Dayi, John J Horton, and David C Parkes. Learning from synthetic labs: Language models as auction participants. *arXiv preprint arXiv:2507.09083*, 2025.

Yu-Zhe Shi, Haofei Hou, Zhangqian Bi, Fanxu Meng, Xiang Wei, Lecheng Ruan, and Qining Wang. Autodsl: Automated domain-specific language design for structural representation of procedures with constraints. *arXiv preprint arXiv:2406.12324*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

Jie Sun, Tianyu Zhang, Houcheng Jiang, Kexin Huang, Chi Luo, Junkang Wu, Jiancan Wu, An Zhang, and Xiang Wang. Large language models empower personalized valuation in auction. *arXiv preprint arXiv:2410.15817*, 2024.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69, 2023.

USE OF LLMS IN OUR WORK

We used large language models (LLMs) in four ways: (i) manuscript polishing—to improve grammar, clarity, and flow without altering substantive claims; (ii) literature triage—to surface potentially relevant papers; (iii) data creation—to synthesize a portion of our NL→DSL pairs (see Sec. 3.4); and (iv) *prompt design*—to iterate on task instructions and few-shot exemplars. All LLM outputs affecting results were reviewed by authors for accuracy, and dataset items were validated with our parser/validator pipeline and spot-audited by humans.

A MARKET INTERPRETATION DSL COMPONENTS

The core building blocks and their overall structure are:

```
market {
  global_settings { ... }
  auction { ... } // repeatable
  stage(name="...") { ... } // optional, repeatable
  matching { ... } // optional
  constraints { ... } // optional
  objectives { ... } // optional
}
```

Block Descriptions

- **market**: Top-level container for a complete market specification.
- **global_settings** (optional): Global parameters (e.g., units, supply, defaults).
- **auction** (repeatable): Auction mechanism definition (type, participants, goods, valuations, rules).
- **stage** (optional, repeatable): Multi-stage orchestration with its own auction and options.
- **matching** (optional): Matching market (type, participants, compatibility, rule).
- **constraints** (optional): Feasibility/policy conditions; simple or parameterized forms.
- **objectives** (optional): Optimization goals used by solver backends.

B FORMAL GRAMMAR (EBNF)

```
program            : "market" "{" market_block* "}" ;

market_block       : global_settings_block
                   | auction_block
                   | stage_block
                   | matching_block
                   | constraints_block
                   | objectives_block ;

global_settings_block
                   : "global_settings" "{" gs_element* "}" ;
gs_element         : currency_decl
                   | supply_decl
                   | reserve_price_decl ;

auction_block      : "auction" "{" auction_element* "}" ;
auction_element    : auction_type_decl
                   | participants_decl
                   | goods_decl
                   | valuations_decl
                   | allocation_rule_decl
                   | payment_rule_decl ;

stage_block        : "stage" "(" "name" "=" string_literal ")"
                     "{" stage_element* "}" ;
stage_element      : auction_block
```

```
                    | reauction_decl ;
reauction_decl      : "reauction" "(" "unsold_goods" "=" string_literal ","
                                  "auction_type"  "=" string_literal ")" ;

matching_block      : "matching" "{" matching_element* "}" ;
matching_element    : matching_type_decl
                    | participants_decl
                    | compatibility_graph_decl
                    | matching_rule_decl ;

constraints_block   : "constraints" "{" constraint_entry_list "}" ;
objectives_block    : "objectives"  "{" objective_entry_list  "}" ;

participants_decl   : "participants" "(" string_literal_list ")" ;
goods_decl          : "goods"        "(" string_literal_list ")" ;
valuations_decl     : "valuations"   "(" valuation_entry_list ")" ;

string_literal_list: "[" (string_literal ("," string_literal)*)? "]" ;
valuation_entry_list
                    : "{" (valuation_entry ("," valuation_entry)*)? "}" ;
valuation_entry     : string_literal ":" "{" good_value ("," good_value)* "}" ;
good_value          : string_literal ":" number ;

auction_type_decl   : "type"           "(" string_literal ")" ;
allocation_rule_decl
                    : "allocation_rule" "(" string_literal ")" ;
payment_rule_decl   : "payment_rule"   "(" string_literal ")" ;

matching_type_decl : "type"           "(" string_literal ")" ;
matching_rule_decl : "matching_rule"  "(" string_literal ")" ;
compatibility_graph_decl
                    : "compatibility_graph" "(" compatibility_entry_list ")" ;
compatibility_entry_list
                    : "{" (compatibility_entry ("," compatibility_entry)*)? "}" ;
compatibility_entry: string_literal ":" "[" (string_literal ("," string_literal)*)? "]" ;

constraint_entry_list
                    : (constraint_param_entry | string_literal)
                      ("," (constraint_param_entry | string_literal))* ;
constraint_param_entry
                    : identifier "(" (parameter_assignment
                                      ("," parameter_assignment)*)? ")" ;
parameter_assignment
                    : identifier "=" value ;

objective_entry_list
                    : (string_literal ("," string_literal)*)? ;

string_literal      : ESCAPED_STRING ;
identifier          : /[A-Za-z_][A-Za-z0-9_]*/ ;
number              : SIGNED_NUMBER ;
value               : number | string_literal | boolean ;
boolean             : "true" | "false" ;
```

C VALIDATION

What is verified

- Names and References: unique goods/participants; auctions reference declared participants/-goods.
- Valuations Consistency: keys match auction participants; goods in valuations are declared; sparse entries → warnings.
- Rules Recognition: auction types, allocation/payment rules recognized or mapped from common aliases.
- Stage/Matching (if present): global settings sanity; stage naming; reauction fields; matching type/rule; graph nodes exist; symmetry warnings.
- Constraints/Objectives (if present): types recognized; basic parameter sanity; objective conflict warnings.

Simple Validation Algorithm

```
Input             : MarketProgram
Output            : ValidationReport (errors, warnings, suggestions); IR may be autofixed

1) Basic field checks (hard errors)
   - Good.name not empty; reserve_price >= 0
   - Bidder.name not empty; budget >= 0
   - Auction.auction_type not empty
   - Assignment fields not empty; bid_price >= 0

2) Core checks (always)
   - Unique names; auctions reference existing participants/goods
   - valuations match auction participants/goods; sparse -> WARN, mismatches -> ERROR
   - auction_type, allocation_rule, payment_rule:
     - map known aliases
     - unknown -> ERROR; some types partially implemented -> WARN

3) Stage/Matching checks (only if present)
   - Global settings: supply/reserve defaults; negatives -> ERROR, missing -> WARN
   - Stages: unique, named, each has an auction
   - Reauction: needs unsold_goods and auction_type; validate type; loose goods check
   - Matching: normalize matching_type; unknown -> WARN + default to bipartite
     - participants non-empty, no duplicates
     - compatibility_graph nodes exist; symmetry missing -> WARN
     - matching_rule: missing -> default stable_matching (WARN); unknown -> ERROR

4) Advanced checks (only if constraints/objectives present)
   - Constraints: type recognized; params sane (e.g., budgets >= 0, caps >= 0)
   - Objectives: normalize; unknown -> ERROR; conflicting goals -> WARN

5) Autofix safe defaults
   - Missing/unknown allocation_rule -> highest_bid
   - Missing/unknown payment_rule   -> second_price
   - Global supply missing/invalid  -> 1
   - Missing reserve_price (global/good) -> 0.0
   - All fixes logged as suggestions

6) Return report; program contains applied defaults where safe
```

D BACKEND SELECTION

Heuristic (implemented in `MarketCompiler`)

- Pure matching or simple Phase-1 auctions → Pandas (NetworkX for matching).
- Multi-stage without optimization features → Pandas + Prefect orchestration.
- Combinatorial auctions or constraints/objectives → CVXPY optimization.

Mapping

IR features	Selected backend
auction only, valuations, simple rules	Pandas
matching (bipartite/stable)	Pandas + NetworkX
stage flow (no constraints/objectives)	Pandas + Prefect
constraints/objectives present	CVXPY
combinatorial auction	CVXPY

E DATASET GENERATION PIPELINE

Overview

1. Sample a use case (914 total) and random market settings; assemble 4-shot prompt + grammar (markdown + EBNF).
2. Generate (GPT-4o-mini) brief description + DSL.
3. Filter with 4-stage validation (Parser → Validator → Compiler → Execute); keep only programs that pass all stages.

4. Enhance description (GPT-4o-mini) by extracting all facts from DSL; replace description text only.
5. Consistency check (GPT-4o): "YES/NO" whether description matches DSL; keep YES, drop NO.

Prompt for data generation

````
You are an expert Market Mechanism DSL generator. I will provide you with:
- The grammar of the MarketMechanismDSL,
- A few example DSL programs,
- A target use case,
- And a set of market settings.

Your task is to generate a complete MarketMechanismDSL program that
fits the given scenario and settings. Strictly follow the provided
grammar and take inspiration from the examples. Use the canonical
constructs: participants([...]), goods([...]), valuations({ ... }),
and valid allocation_rule/payment_rule names. Do not invent syntax
not present in the grammar.

Please output your response in EXACTLY the following format and nothing else:

Description:

```markdown
1-3 sentences written from the user's perspective describing the market
```

DSL code:

```dsl
<your complete MarketMechanismDSL program here>
```

---

Inputs:

DSL Grammar:
```markdown
{grammar}
```

DSL Program Examples:
```dsl
{examples}
```

Use Case and Settings:
- Domain: {domain}
- Scenario: {scenario}
- Settings: {settings}
````

Prompt for description completion (from DSL)

````
You are a precise technical writer. Given a MarketMechanismDSL program,
write a COMPLETE, human-readable task description that includes ALL facts
present in the DSL (participants, goods, valuations, auction/matching type,
allocation/payment/matching rules, key settings). Do NOT hallucinate new
entities or numbers. Use clear, concise prose (4-8 sentences).

Output EXACTLY in this format:

Description:

```markdown
<concise but complete description, entirely derived from the DSL>
```

---

Inputs:

DSL Program:
````

```
```dsl
{dsl_code}
```

Optional Examples (style guide only):
```markdown
{examples}
```
```

F Completeness Agent Workflow

Node-by-node extraction Given a natural-language description, the agent queries nodes in order and marks each as *Enough/Not Enough*:

1. Auction (type, participants, goods, valuations, rules).
2. Global settings (currency, supply, reserve_price).
3. Stages (name, auction, optional reauction).
4. Matching (type, participants, compatibility_graph, rule).
5. Constraints (simple/parameterized).
6. Objectives (list).

If Not Enough, it asks for the missing facts before proceeding. Finally it merges fragments into a normalized schema and renders minimal DSL.

G End-to-End Example (Simple Auction)

G.1 Natural Language Input

Step 1: Natural Language

```
"Create a second-price auction for three participants competing for an advertising slot.
Alice values the slot at $100, Bob at $80,
and Carol at $90. Use highest bidder wins allocation."
```

G.2 Copilot Generation (NL → DSL)

Step 2: Generated DSL

```
market {
  auction {
    type("second_price")
    participants(["Alice", "Bob", "Carol"])
    goods(["AdSlot"])
    valuations({
      "Alice": {"AdSlot": 100},
      "Bob": {"AdSlot": 80},
      "Carol": {"AdSlot": 90}
    })
    allocation_rule("highest_bid")
    payment_rule("second_price")
  }
}
```

G.3 Parser (DSL → Parse Tree)

Step 3: Lark Parse Tree (excerpt)

```
program
  statement
    auction_block
      auction_element
        auction_type_decl
          string_literal: "second_price"
      auction_element
        participants_decl
          string_literal_list
            string_literal: "Alice"
            string_literal: "Bob"
            string_literal: "Carol"
      ...
```

G.4 Transformer (Parse Tree → IR)

Step 4: Intermediate Representation (MarketProgram)

```
MarketProgram(
  goods=[Good(name="AdSlot")],
  bidders=[Bidder(name="Alice"), Bidder(name="Bob"), Bidder(name="Carol")],
  auctions=[Auction(
    auction_type="second_price",
    participants=["Alice","Bob","Carol"],
    goods=["AdSlot"],
    valuations={"Alice":{"AdSlot":100},"Bob":{"AdSlot":80},"Carol":{"AdSlot":90}},
    allocation_rule="highest_bid",
    payment_rule="second_price"
  )],
  assignments=[], global_settings=None, stages=[],
  matching=None, constraints=[], objectives=[]
)
```

G.5 Validation (IR Semantic Check)

Step 5: Validation

```
program.validate_comprehensive() successful
- All participants referenced in valuations
- All goods have valid valuations
- Auction type and payment rule compatible
- Unique participant and good names verified
```

G.6 Compilation and Backend Selection

Step 6: Compiler Analysis

```
Analyzing DSL features...
- Single auction block detected
- No constraints or objectives
- No multi-stage features
- No matching markets

Backend Selected: Pandas
Reasoning: Simple auction, simulation-based approach sufficient
```

G.7 Execution (Generated Code Excerpt)

Step 7: Pandas Backend Code (excerpt)

```
# Allocation: highest_bid rule
allocations = {}
for good in goods:
    good_bids = bids_df[bids_df['good'] == good].copy()
    if not good_bids.empty:
        winner = good_bids.loc[good_bids['valuation'].idxmax()]
        allocations[good] = winner['participant']

# Payment: second_price rule
payments = {}
for good, winner in allocations.items():
    good_bids = bids_df[bids_df['good'] == good].copy()
    sorted_bids = good_bids.sort_values('valuation', ascending=False)
    if len(sorted_bids) >= 2:
        second_highest = sorted_bids.iloc[1]['valuation']
        payments[winner] = payments.get(winner, 0) + second_highest
    else:
        payments[winner] = payments.get(winner, 0) + sorted_bids.iloc[0]['valuation']
```

G.8 Results

Step 8: Console Output

```
=== AUCTION RESULTS ===
Allocations:
  AdSlot: Alice
Payments:
  Alice: $90
Artifacts:
  - auction_results.csv
```