Intelligent Equality Saturation: From Hierarchical Prospection to a Minimal Implementation

Anonymous Author(s)

***Abstract*—Equality saturation avoids premature commitment to a single, sub-optimal rewrite path in broad compilation problems, yet its practical adoption is limited by uncontrolled e-graph growth under finite time and memory budgets. Managing this growth requires strategies that decide which rewrites may interact, how search should be phased, and when intermediate e-graph states should be simplified. Large language model agents are attractive for this design task, but direct low-level control of backend rewrite code or per-rewrite decisions is unstable, expensive, and difficult to validate.**

This paper presents `EggMind` and makes two contributions. First, `EggMind` proposes two-level hierarchical prospection as an explicit strategy-boundary view for intelligent EqSat: phase-level adaptation should operate through bounded functors and strategy DSL constructs, while case-level transfer should reuse strategy artifacts rather than raw e-graph state or open-ended conversation history. Second, `EggMind` narrows that broad view into a minimal work-in-progress implementation target: offline synthesis of reusable, inspectable EqSat strategies. In this formulation, related cases, rewrite rules, a cost model, and a selected functor interface are used to synthesize a strategy offline. The resulting strategy is then reused online by a trusted EqSat runtime. A preliminary vectorization case study suggests that this narrowed strategy-synthesis formulation is executable, measurable, and useful for improving the resource-quality trade-off under bounded search.

I. Introduction

Equality saturation (EqSat) avoids the early binding decisions made by traditional compilers by exploring a vast space of equivalent programs simultaneously within the e-graph data structure. Supported by high-performance engines such as `egg` [9] and `egglog` [14], EqSat has become an increasingly popular paradigm in domains such as vectorization [7], [8], logic synthesis [2], [11], and high-level synthesis [4], [10].

However, in practical deployments, the e-graph scale and algorithmic complexity of the blind EqSat process remain the primary bottlenecks to obtaining optimal solutions within reasonable time and resource constraints. This persists despite recent advancements in e-graph extraction [1], [12], which aim to accelerate the selection of final solutions but cannot mitigate the underlying state-space explosion. Current approaches manage such complexity with expert-tuned, domain-specific, static strategies [7], but these strategies lack runtime self-adaptation and generality to new problems.

Adopting LLMs is a natural way to automatically compose effective EqSat strategies for given problems, inspired by the success of LLM-driven compiler optimization [3], [6]. Recently, ASPEN [13] has demonstrated the feasibility of using LLM agents to automate rule proposal and selection for RTL optimization, leveraging real-world PPA feedback to guide the saturation and extraction process. However, these approaches typically operate on a specific domain or rely on design-specific rule discovery, and the key question remains: how to apply LLM techniques to general EqSat strategy problems across different phases and problem scales. Naively asking LLM agents to make all EqSat decisions consumes many tokens and can lead to suboptimal solutions and explosive e-graph growth.

To address these challenges, we present `EggMind`, a framework that enables scalable EqSat via *two-level hierarchical prospection*. First, at the *phase level*, `EggMind` treats EqSat as a runtime control loop, where an LLM evolves compositions of high-level functors (e.g., rule generators and schedulers) based on e-graph signals rather than issuing per-rewrite decisions. Second, at the *case level*, it transfers successful strategy patterns from tractable seed cases to complex instances to bootstrap exploration. This design is supported by an inspectable strategy DSL that represents schedules, rule groups, and phase structure as reusable artifacts. Ultimately, `EggMind` provides a scalable, token-efficient paradigm that tames e-graph explosion and transfers optimization expertise across problem scales.

This paper then narrows the broad `EggMind` vision into a minimal implementation target. Instead of requiring online LLM control in every production EqSat run, we focus on offline synthesis of reusable, inspectable strategies from representative cases. The resulting strategy is validated and executed by a trusted EqSat runtime and reused on new cases from the same family.

This paper contributes:

- a two-level hierarchical prospection framing for scalable EqSat
- a narrowing of case-level learning into offline reusable strategy synthesis
- an explicit strategy-synthesis harness over selected EqSat control functors
- a preliminary vectorization case study suggesting that synthesized strategies can improve the EqSat resource-quality trade-off.

II. Background

A. E-Graph and Equality Saturation

An e-graph is a compact data structure that represents large sets of equivalent expressions by grouping them into equivalence classes (e-classes) of alternative e-nodes. Equality saturation (EqSat) starts from an e-graph built from the initial

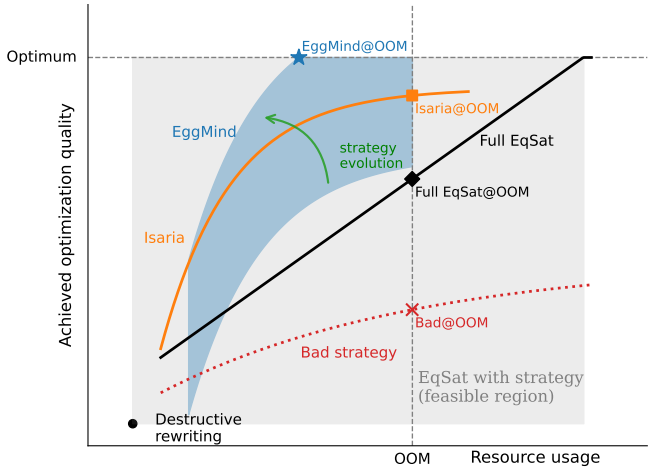

Fig. 1. Optimization spectrum for rewriting techniques.

program, and repeatedly applies rewrite rules to add e-nodes and merge e-classes, monotonically growing the e-graph until a fixed point or a resource limit is reached, after which an extraction phase selects the best representative expression given the cost function [9], [14]. By separating exploration from selection, EqSat avoids committing too early, but in practice interacting rules can grow the e-graph rapidly and make saturation expensive, motivating strategy control for scalability [7], [9].

B. A Spectrum of Rewriting Techniques

We analyze existing rewriting techniques through a unified lens of how they trade off optimization quality against finite time and resource budgets. We compare traditional destructive rewriting (e.g., MLIR passes [5]), classic equality saturation, strategy-driven EqSat exemplified by Isaria [7], and the broader `EggMind` design in Fig. 1. The x-axis is resource usage (e.g., memory), and the y-axis is achieved optimization quality. The horizontal dashed line denotes the best achievable solution in practice, while finite budgets truncate attainable outcomes, as indicated by the vertical dashed OOM line.

Destructive rewriting appears as a low-resource, low-quality point in the lower-left corner. E-graph-based approaches trace curves because achieved solution quality varies continuously with available budgets. Classic, unguided EqSat can in principle reach the theoretical optimum given sufficient resources, but under realistic constraints it often fails to saturate. Introducing a strategy constrains the rewrite search and restricts attainable outcomes to a feasible region (the shaded rectangle), where solution quality is bounded between the destructive baseline and the theoretical optimum, and resource consumption is bounded between the destructive baseline and full saturation. Different strategies trace different curves within this region: a well-designed, expert-tuned strategy [7] attains substantially higher quality under the same budget (orange curve) but requires domain expertise and transfers poorly across workloads, while a poor strategy may even underperform exhausted EqSat (red curve). The broader `EggMind` design aims to occupy and expand the upper portion of this feasible region by synthesizing and evolving rewriting strategies. Leveraging execution traces and an LLM-based strategist, it dynamically modifies the rewriting strategies both online and across instances. In Fig. 1, this behavior is depicted as an evolving blue envelope. The blue star marks the expected `EggMind` operating point under the OOM budget, which approaches expert-level performance while respecting lower time and resource constraints.

TABLE I
FUNCTORS IN AN EGGMIND PHASE ITERATION.

#	Functor	Role
1	`PREPHASE`	Iteration hook: initialize context or state.
2	`EVAL`	Global termination decision.
3	`STEP`	Iteration control decision.
4	`SCHEDULE`	Rewrite rule selection and ordering.
5	`UPDATE_RULESET`	Active ruleset adaptation.
6	`INSPECT`	E-graph state query.
7	`ANALYZE`	Derived metric or property computation.
8	`SIMPLIFY`	E-graph growth control.
9	`SOLVE`	Solution extraction decision.
10	`POSTPHASE`	Iteration hook: finalize records or state.

III. Abstractions for LLM-Based Strategies

`EggMind` is designed around the principle that effective equality saturation requires explicit, programmable control over rewriting behavior. To this end, `EggMind` provides two complementary components: runtime-programmable *functors* and an embedded strategy DSL. These components decouple optimization strategy from the underlying EqSat engine, transforming EqSat into a manageable control loop that LLMs can evolve.

A. Functor Mechanism

`EggMind` introduces *functors* as the foundational abstraction for runtime control in EqSat-based optimization. A functor encapsulates a constrained control routine that observes the current e-graph state, runtime context, or phase-local records and emits concrete control actions. By externalizing these decisions, `EggMind` transforms the standard rewriting loop into an explicit, programmable environment. As summarized in Table I, `EggMind` exposes ten specific control points that structure each runtime iteration. These points allow EqSat strategies to react to lightweight runtime signals, such as e-graph growth statistics and cost-improvement trends, enabling adaptive strategy adjustment through LLM guidance at runtime.

B. Strategy DSL

To orchestrate EqSat-based optimization actions into reusable and auditable policies, `EggMind` uses a lightweight strategy DSL that LLMs can compose and evolve. Fig. 2 presents the DSL's declarative three-layer hierarchy.

Layer 1 (Units) defines the fundamental *optimization atoms*. It covers pattern variables ($?x$), term templates (p),

Layer 1: Units (Optimization Atoms)

$?x \in$	PatVars	(Pattern variables, Holes)
$p ::=$	term$(?x, \dots)$	(Term templates)
$r ::=$	$p_1 \to p_2$ $[g]$	(Rewrite rules with optional guard)
$rs ::=$	$\{r_1, r_2, \dots\}$	(Named rulesets)
$m ::=$	cost_fn(e)	(Extraction metrics)

Layer 2: Commands (Rewrite Algebra)

$c ::=$	`Saturate`(rs)	(Apply until fixpoint)
	$\mid$ `Repeat`(n, c)	(Bounded iteration)
	$\mid$ `Seq`$(c_1, \dots)$	(Sequential composition)
	$\mid$ `Union`$(c_1, \dots)$	(Parallel merge)

Layer 3: Plans (Orchestrated Workflows)

$ph ::=$	Phase(id, c, n)	(Named stage with iteration limit)
$pl ::=$	Plan$(ph, \dots)$	(Concatenated phases)
	$\mid$ Plan$(\dots, H)$	(With init/cleanup hooks)

Fig. 2. Formal syntax of the strategy DSL's three-layered hierarchy.

and rewrite rules (r) that are grouped into named **rulesets** (rs). These rulesets form the domain-specific vocabulary of optimization knowledge, while extraction metrics (m) guide the final selection of optimal programs from the e-graph.

Layer 2 (Commands) constitutes a *rewrite algebra* that specifies how these units are applied. By using combinators such as `Saturate`(rs) for fixpoint application, `Repeat`(n, c) for bounded loops, and `Seq` or `Union` for composition, the DSL treats scheduling as a first-class algebraic object.

Layer 3 (Plans) orchestrates these commands into *structured workflows*. A **Phase** (ph) encapsulates a command with a strict iteration limit, and phases are then concatenated into a **Plan** (pl). This layer also provides hooks (H) for initialization and cleanup, ensuring that complex optimization pipelines remain modular and reproducible.

Because strategies are represented as first-class data rather than imperative glue code, the DSL provides a constrained, schema-like target for LLM-assisted synthesis. In the `EggMind` framework, the LLM generates compact strategy plans that are statically validated for structural well-formedness and resource bounds before execution. By delegating low-level e-graph management to the DSL runtime and limiting the LLM's authority to the policy layers, the DSL prevents the LLM from making detailed e-graph action decisions and lets it concentrate on strategy evolution.

IV. Two-Level Hierarchical Prospection

`EggMind` proposes *two-level hierarchical prospection* for leveraging LLMs to achieve both strong scalability and high solution quality. The system reasons about the next phases within one run and about future cases across a related workload. As illustrated in Fig. 3, the two levels are the *phase level* and the *case level*. The `EggMind` runtime starts from a cold-start strategy and refines its logic across tasks and related cases, converging toward a well-optimized strategy.

A. Phase Level: Intra-Case Adaptation

Phase-level intelligence enables *intra-case adaptation*, transforming equality saturation from an open-loop, exhaustive search into a responsive, closed-loop control system. This level governs how `EggMind` dynamically adjusts its behavior *during* a single optimization run, using real-time evidence to refine its strategy. As established in Section III-A, each control point is represented by a functor, which may be either fixed or LLM-evolved. Phase-level intelligence emerges primarily from the latter: LLM-based functors can update their internal policies online, guided by runtime feedback rather than static pre-programming.

A core design principle of `EggMind` is bounded authority: the LLM is strictly prohibited from modifying the underlying e-graph engine or injecting unverified rewrite rules, operating instead at the strategy layer where it proposes updates to functor configurations, such as ruleset activation or pruning thresholds, exclusively at phase boundaries. As depicted in Fig. 3, an **LLM-based strategist** resides within the optimization pipeline of every case, acting as a high-level controller that evolves functor behaviors by consuming lightweight runtime statistics such as e-graph size, cost-reduction trends, and e-graph analysis. By confining the LLM's agency to policy orchestration while delegating execution to the formal DSL runtime, `EggMind` achieves a robust synergy between expressive, adaptive search control and the rigorous determinism required for backend correctness.

B. Case Level: Inter-Case Learning

`EggMind` further introduces a case level that enables experience reuse across related EqSat instances. Rather than treating each optimization task as an isolated execution, `EggMind` models optimization as a population of structurally similar cases and uses case-level intelligence in two complementary ways: *Strategy induction* for cold start and *Batch learning and transfer* for runtime. The former provides a strong initial policy by synthesizing reusable DSL strategies from tractable seed cases, while the latter continuously refines and specializes those policies through aggregated evidence from larger case batches. This division makes the workflow explicit: initialize strategy quality early, then improve robustness and domain fit through iterative cross-case feedback.

a) Strategy induction.: For the "cold start" of a new domain, `EggMind` offers two initialization paths: an expert-driven specification that leverages predefined domain templates, and strategy induction from small seed cases using an LLM. While the former relies on manual expertise, the latter exemplifies `EggMind`'s case-level intelligence by distilling optimization experiences from tractable instances into reusable strategies. The process starts from baseline EqSat runs on small instances and uses the explanation facilities of the `egg` backend [9] to obtain structured proof traces, exposing which rule interactions are responsible for high-quality solutions. The LLM then acts as a tool-mediated strategist: it queries rule-space statistics, proposes candidate rulesets, and constructs the structural skeleton of phase strategies. `EggMind` uses

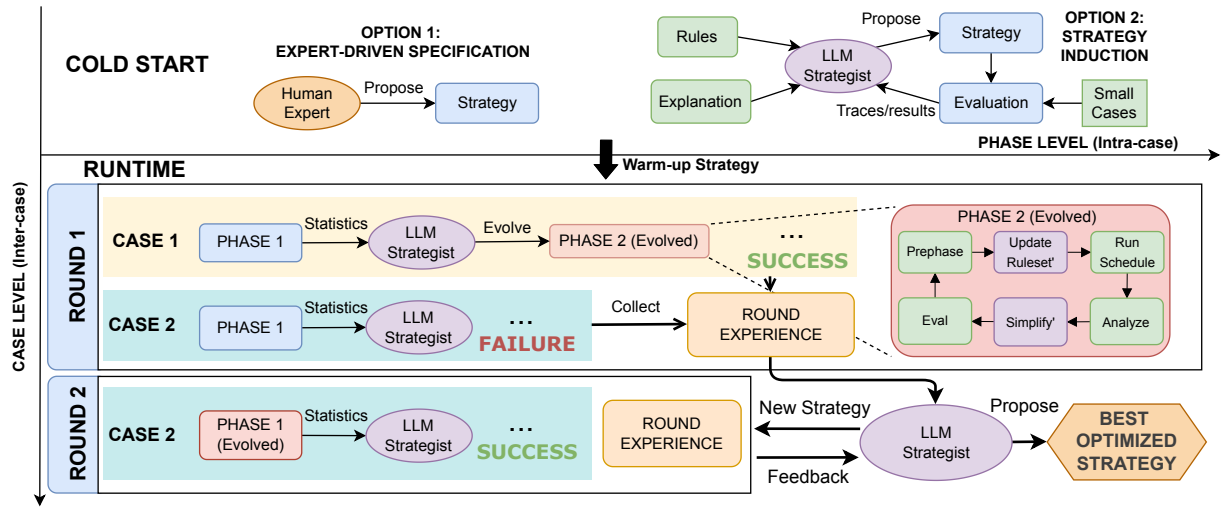

Fig. 3. Two-level hierarchical prospection. The phase level controls how one EqSat run is organized, while the case level transfers experience across related cases.

a hybrid synthesis workflow that separates structural design from numerical tuning: the LLM determines policy structure, including phase order, ruleset predicates, and loop shape, while an autotuning loop searches iteration limits and loop counts. In this loop, the `Evaluation` tool is applied to the small seed cases to filter candidate strategies before transfer: each candidate must satisfy a hard cost constraint (final cost no worse than the baseline) and is further ranked by runtime and memory. The selected policy, which performs best, is then emitted as a typed and executable warm-up strategy artifact that can be directly reused on larger cases.

b) Batch learning and transfer.: Beyond cold start, `EggMind` performs iterative refinement through batch execution over related instances, using a shared container called "Round Experience" that records per-case outcomes, schedule choices, and resource-quality tradeoffs. As results accumulate, the system retrieves similar solved cases using lightweight case features and transfers strategy-level decisions (e.g., phase structure, ruleset selection bias, and pruning aggressiveness) to ongoing or failed runs. Failed cases can then be retried with learned policy variants and adjusted limits, while consistently ineffective policies are pruned from the candidate pool. This workflow specializes strategies to a target domain without manual retuning and keeps transfer lightweight because it reuses control policies rather than e-graph states. In effect, case-level learning turns isolated EqSat runs into a feedback process that compounds experience across the batch.

The rest of this paper narrows the broad two-level design. We keep the case-level goal of transfer, but realize it as offline synthesis of one reusable strategy artifact rather than full online batch learning.

V. Minimal Implementation: Strategy Synthesis

The broad case level can be stated as a learning problem: how should experience from solved EqSat instances help solve future ones? In full generality, this question includes representation learning over programs, online scheduling, failure recovery, and persistent memory across workloads. Such a formulation is expressive, but it is also difficult to implement, evaluate, and validate. We therefore treat reusable strategy synthesis as the minimal implementation step that makes the case level of hierarchical prospection concrete.

Given a family of related EqSat cases, a fixed rewrite vocabulary, a cost model, and a selected subset of control functors, the goal is to synthesize an explicit strategy offline from representative cases and reuse it online on new cases from the same family. This reduction deliberately excludes several broader forms of intelligence: the LLM is not placed inside the production EqSat loop, the runtime does not maintain open-ended conversational memory across all workloads, and the strategy search does not modify the trusted rewrite semantics.

A. Selected Control Functors

The reduction also narrows the phase level. Rather than allowing an agent to manipulate every runtime decision, we focus on functors that define the high-level shape of search. `SCHEDULE` and `UPDATE_RULESET` determine which rule groups are exposed in each phase and in what order. `STEP` and `EVAL` define bounded repetition, stopping conditions, and resource budgets. `SIMPLIFY` controls where the e-graph is contracted between phases. `SOLVE` determines how the run produces an output. In the preliminary setting in this paper, this means extracting one expression under a cost model, while future settings may emit candidate sets for downstream evaluation.

Other functors are still useful, but they play a different role in this narrowed formulation. `INSPECT` and `ANALYZE` can provide observations to the synthesis harness, such as e-graph size, phase progress, timeout behavior, or cost improvement. They are evidence channels rather than online LLM actions.

`PREPHASE` and `POSTPHASE` mostly provide bookkeeping around phase-local records. This separation keeps the agent's authority at the strategy level: it can choose and parameterize a small number of control functors, while the backend remains responsible for executing sound rewrites.

B. Problem Formulation

The synthesis problem receives a family of related EqSat cases, a rewrite vocabulary, a cost model, and the selected functor interface. The cases need not be identical, but they should share enough structure that a strategy learned from representative instances can plausibly transfer to held-out ones. The rewrite vocabulary defines the legal transformations. The cost model defines what it means to improve a program.

The output is a reusable strategy specification. At a minimum, such a strategy should describe ruleset groupings, phase order, bounded repetition, simplification locations, and resource budgets. It should be explicit enough to inspect and edit, and structured enough for static checks such as bounded loops and references to known rewrite rules. One practical way to realize such strategies is through a small strategy language over the selected functors, but the narrowed formulation is not tied to a particular DSL implementation.

C. Agentic Synthesis Loop

Fig. 4 sketches the agentic synthesis loop implied by this formulation. Candidate strategies are generated, executed by the EqSat backend, and compared using resource-quality outcomes such as final cost, runtime, peak memory, timeout, or out-of-memory behavior. The selected strategy is then reused online on new cases from the same family. This separation amortizes the cost of agentic design and avoids placing an LLM in the inner loop of every production optimization run.

The loop can be decomposed into a small set of conceptual agent and tool roles. The *Strategist* decides what aspect of the strategy should be improved next, such as phase order, simplification placement, loop budgets, or the split between expansive and optimizing rewrites. The *Generator* proposes candidate strategies constrained by the strategy interface. The *Evaluator* lowers a candidate strategy to backend execution and reports outcomes. The *Inspector* and other tools provide per-phase evidence, such as rule-application traces, e-graph growth, timeout behavior, and cost-improvement trends.

This loop depends on bounded authority. Agents should not mutate the e-graph directly, inject unchecked rewrites, or bypass resource limits. They produce strategy specifications that a runtime validates before execution. The agent searches the design space, while the backend remains responsible for sound rewrite application and extraction. This boundary also makes failures legible: a timeout, memory blowup, or quality loss can be attributed to phase structure, ruleset interaction, or simplification choices rather than opaque generated backend code. The reusable unit is therefore a concrete strategy rather than an implicit conversational history or a collection of low-level backend scripts.

VI. Preliminary Case Study

To ground the formulation, we conducted a small preliminary study on a vectorization benchmark family used in prior EqSat work [7], [8]. The goal is not to evaluate a complete optimizer. Instead, the study asks whether an explicit strategy is a useful boundary for expressing EqSat control decisions and observing their resource-quality consequences.

The probe uses a small subset of convolution and matrix-multiplication cases with a fixed rewrite vocabulary and latency-oriented cost model. Candidate strategies describe phased schedules over rulesets, bounded repetition, and simplification points. We intentionally report only a compact subset of cases, with abstract labels in the figure, so that the evidence supports the paper's modeling claim rather than becoming a benchmark table.

Fig. 5 shows extracted cost for the subset, normalized to unguided EqSat. In these examples, the strategy-level schedule either lowers the extracted cost relative to the manual schedule or preserves its quality. Resource behavior is also informative: synthesized strategies are not uniformly the fastest choice on every small case, but they avoid the long searches observed for the manual schedule on larger examples in the subset. Their measured wall-clock times range from a few seconds to under five minutes, and their peak memory stays in a narrow band of roughly 0.53–0.64 GB. We use these time and memory observations only as evidence that strategy-level control can make search behavior inspectable.

The qualitative difference was that expansive rewrites were not allowed to interact freely with all later optimizations. Instead, they were placed in bounded phases with simplification between search steps. This preliminary result should be interpreted cautiously: it does not establish generality across domains or replace a full evaluation. It does, however, support the central modeling claim that reusable strategies provide a useful granularity for proposing, measuring, and revising EqSat search behavior.

VII. Discussion and Future Work

The broader implication is that hierarchical prospection for compiler and architecture optimization should not be framed as unrestricted code generation. For complex optimization systems, the important design question is often where to place the agent boundary. In EqSat, the natural boundary is the strategy layer: it is high-level enough for language models to reason about and low-level enough to affect resource behavior.

This perspective also clarifies what should be evaluated in future work. A useful agentic optimizer should not only find a good result on one instance. It should produce a strategy that is inspectable, reusable across a related family, and robust under resource budgets. The strategy should also make failures legible. If a strategy times out, exceeds memory, or loses quality, the failure should be attributable to phase structure, ruleset interaction, or simplification choices rather than to opaque generated backend code.

The current formulation also makes a simplifying assumption about evidence. It fits direct EqSat workflows where

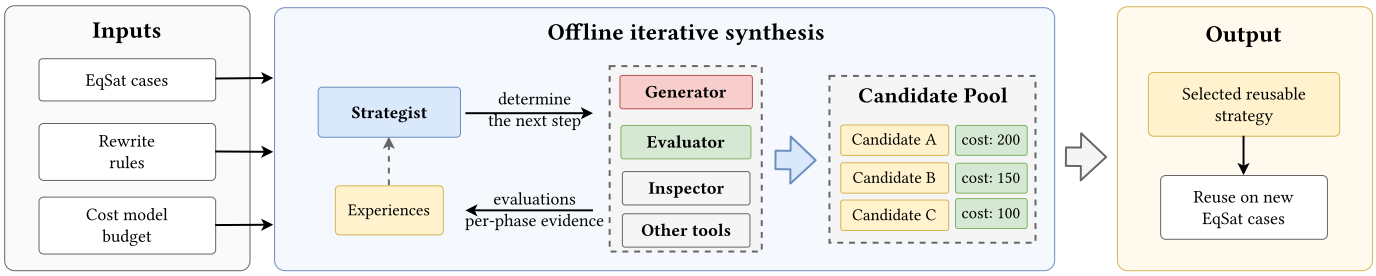

Fig. 4. Agentic synthesis harness for reusable EqSat strategy design. Agents operate over structured strategy specifications, while a trusted EqSat harness validates and executes candidates and returns resource-quality observations, keeping agent authority bounded.

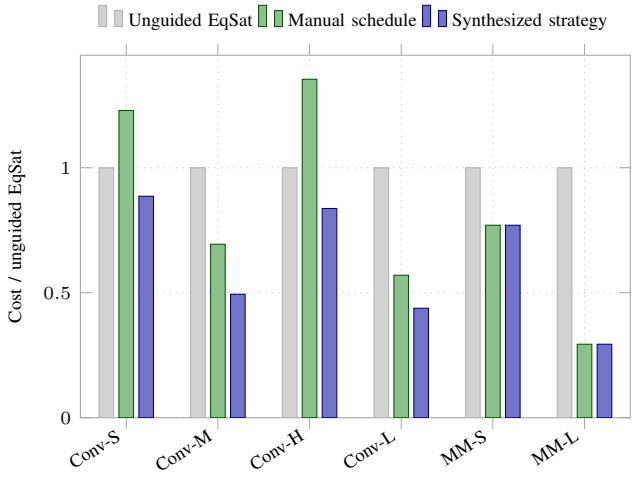

Fig. 5. Preliminary vectorization results for selected benchmark families, reporting extracted cost normalized to unguided EqSat (lower is better).

the backend extracts one best expression and compares its cost. Many architecture and compiler optimization workflows are broader than this. Trinity- or ISAMORE-style systems, for example, may produce multiple candidates and rely on downstream tools to evaluate feasibility, quality, performance, or Pareto trade-offs. Extending strategy synthesis to such settings would require evidence beyond a single extracted cost: candidate-set diversity, the rate of useful candidates, downstream evaluation cost, and the quality of the resulting frontier may all become part of the feedback signal. This is a natural next step because the same strategy boundary can describe not only how to reach one low-cost extraction, but also how to guide search toward candidate sets that are valuable to later tools.

Another extension is to let agents help expand the rewrite space itself. In this paper, strategies operate over a fixed rewrite vocabulary. Future systems could allow an LLM to propose candidate rewrites, rewrite schemas, or rule families, while keeping semantic validation outside the model. Trusted equivalence checkers, domain-specific legality tests, or synthesis-based validators would decide which proposals are admitted. This preserves the bounded-authority principle: the LLM expands the design space, but the system verifies rules before they can affect optimization. Under this view, rule proposal and strategy synthesis can co-evolve without asking the model to be the source of correctness.

More broadly, future work includes richer evidence extraction from successful runs, tractability-aware guidance for large rewrite spaces, broader evaluation across compiler and hardware optimization domains, and stronger validation of strategies before execution. Another important direction is to understand how general such strategies can be: some workloads may require highly domain-specific schedules, while others may share reusable patterns across kernels, circuits, or tensor graphs.

VIII. Conclusion

We argue that the right role for LLM agents in equality saturation is not low-level rewrite control, but strategy-space modeling over reusable, inspectable strategies. Hierarchical prospection separates phase-level control within a run from case-level reuse across related instances. By narrowing the case level into offline reusable strategy synthesis over selected control functors, we obtain a minimal work-in-progress implementation without requiring open-ended runtime agent control. Preliminary vectorization evidence suggests that this strategy-synthesis step is promising and motivates deeper study of strategy-centered agentic design for scalable optimization systems.

References

[1] Y. Cai, K. Yang, C. Deng, C. Yu, and Z. Zhang, "Smoothe: Differentiable e-graph extraction," *Proceedings of the ACM on Programming Languages*, 2025.

[2] C. Chen, G. Hu, C. Yu, Y. Ma, and H. Zheng, "E-morphic: Scalable equality saturation for structural exploration in logic synthesis," in *Proceedings of the 62nd ACM/IEEE Design Automation Conference*, 2025, pp. 1–7.

[3] H. Chen, A. Novikov, N. Vu, H. Alam, Z. Zhang, A. Grossman, and A. Yazdanbakhsh, "Magellan: Autonomous discovery of novel compiler optimization heuristics with alphaevolve," 2026.

[4] J. Cheng, S. Coward, L. Chelini, R. Barbalho, and T. Drane, "Seer: Super-optimization explorer for high-level synthesis using e-graph rewriting," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024, pp. 1029–1044.

[5] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization*, 2021, pp. 2–14.

[6] A. Novikov, N. Vu, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. R. Ruiz, A. Mehrabian, M. P. Kumar, A. See, S. Chaudhuri, G. Holland, A. Davies, S. Nowozin, P. Kohli, and M. Balog, "Alphaevolve: A coding agent for scientific and algorithmic discovery," 2025.

[7] S. Thomas and J. Bornholt, "Automatic generation of vectorizing compilers for customizable digital signal processors," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2024, pp. 19–34.

[8] A. VanHattum, R. Nigam, V. T. Lee, J. Bornholt, and A. Sampson, "Vectorization for digital signal processors via equality saturation," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 874–886.

[9] M. Willsey, C. Nandi, Y. R. Wang, O. Flatt, Z. Tatlock, and P. Panchekha, "egg: Fast and extensible equality saturation," *Proceedings of the ACM on Programming Languages*, vol. 5, no. POPL, pp. 1–29, 2021.

[10] Y. Xiao, Y. Zou, and Y. Liang, "Skyegg: Joint implementation selection and scheduling for hardware synthesis using e-graphs," 2025.

[11] J. Yin, Z. Song, C. Chen, Q. Hu, and C. Yu, "Boole: Exact symbolic reasoning via boolean equality saturation," *arXiv preprint arXiv:2504.05577*, 2025.

[12] J. Yin, Z. Song, C. Chen, Y. Cai, Z. Zhang, and C. Yu, "E-boost: Boosted e-graph extraction with adaptive heuristics and exact solving," 2025.

[13] N. Zhang, C. Deng, J. M. Kuehn, C.-T. Ho, C. Yu, Z. Zhang, and H. Ren, "Aspen: Llm-guided e-graph rewriting for rtl datapath optimization," *2025 ACM/IEEE 7th Symposium on Machine Learning for CAD*, pp. 1–9, 2025.

[14] Y. Zhang, Y. R. Wang, O. Flatt, D. Cao, P. Zucker, E. Rosenthal, Z. Tatlock, and M. Willsey, "Better together: Unifying datalog and equality saturation," 2023.