TABRSI: A RECURSIVE SELF-IMPROVING MULTI-AGENT SYSTEM FOR TABLE UNDERSTANDING AND REASONING

Anonymous authors
Paper under double-blind review

ABSTRACT

Recursive Self-Improvement (RSI) systems, which recursively modify their own code to enhance their capabilities and empirically validate changes using their own benchmarks, have proven effective for developing powerful coding agents and AI scientific discovery systems. However, whether similar paradigms generalize to other domains remains largely unexplored. We study this question in table understanding and reasoning: a challenging domain that requires models to jointly resolve structural table references, reconstruct hierarchical structures, perform precise computation, and produce strictly formatted outputs. To our knowledge, this is the first application of a RSI paradigm to table understanding and reasoning. We first show that Gödel-machine-inspired RSI paradigms, specifically the Darwin Gödel Machine (DGM) and the Huxley Gödel Machine (HGM), do transfer to this domain and yield large gains on several benchmarks. They are not a silver bullet, however: applied as monolithic solvers, they do not consistently outperform CoT, because a solver that only rewrites its own code cannot recover structure that its input never carried, nor choose a different reasoning modality per sub-task. We therefore propose a modular multi-agent pipeline comprising a self-evolving solver agent complemented by specialized supporting roles, while dynamically interleaving CoT and PoT across each role. We term our framework ***TabRSI***. TabRSI jointly searches over pipeline topology, role-specific code, and reasoning modality. Across seven benchmarks and three LLM backbones, TabRSI improves on CoT prompting on every benchmark, by up to $+155\%$.

1 INTRODUCTION

Recent work on recursive self-improvement (RSI) envisions an agent that rewrites its own code, evaluates the rewrite against a benchmark it can run itself, and keeps what measurably helps. Such a paradigm has already produced striking results in program synthesis and software engineering (Zhang et al., 2025; Wang et al., 2026; Romera-Paredes et al., 2024; Novikov et al., 2025; Lehman et al., 2023). What these successes share is a domain in which every candidate is an executable program and every evaluation is an exact, cheap, automatic test. Whether the same machinery transfers to other domains remains largely unexamined. Table understanding and reasoning is a natural place to ask, and one where such an agent ought, on the face of it, to work well. Across tabular benchmarks (Li et al., 2026; Wu et al., 2025; Grijalba et al., 2024) it has been reported repeatedly that writing and executing code is the effective strategy. Applying a self-improving coding agent to tabular reasoning is therefore the natural next step that has not been attempted before.

Applying RSI directly, however, is not straightforward all the time for two reasons. First, code is not uniformly the better modality for every sub-task in table understanding and reasoning. Second, table reasoning is not a single ability but a composition of several, such as parsing and representing the table with hierarchical information, together with reasoning across its structural semantics, metadata, and cross-table relations. Figure 1 illustrates the parsing half of the problem on hierarchical table QA in HiTab (Cheng et al., 2022). Asked *what percentage of Quebec population spoke French in 2011? (answer: 85.4)*, the coding agent returns the same wrong answer even after applying self-improvement loops. Critical metadata that captures the table's hierarchical structure was destroyed by the LLM's token serialization step, and the coding agent cannot give the correct answer once

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

(a) In the source file: two header trees

```
top_root
 |- 2011                          col 0
 `- 2036 (9 scenarios)        cols 1-9

left_root
 `- percent
    |- canada                    row 0
    |  |- english                 row 1
    |  `- french                  row 2
    |- quebec                     row 5
    |  |- english                 row 6
    |  `- french                  row 7
    `- canada outside quebec      10-14
```

$\Longrightarrow$ default table→text serialization

(b) In the solver's prompt: one flat grid

row	% of population	2011	*ref.*	*low*	*high*
0	*(blank)*				
1	english	75.4	77.8	77.7	77.8
2	french	22.9	20.4	20.9	20.2
5	*(blank)*				
6	english	13.6	16.9	16.4	17.2
7	french	85.4	82.4	83.0	82.0
10–14	*third group elided (also a* `french` *row)*				

(c) What a correct PARSER puts back
`row 7` = `percent` > `quebec` > `french` = 85.4
worth HITAB $0.574 \rightarrow 0.811$ over any single LLM alone.

Figure 1: **The information the LLM needs is destroyed even before its prompt exists. (a)** HiTab internally stores the table as two header *trees*: the gold cell is a path on each axis, `percent` > `quebec` > `french` at `2011` = 85.4, and every region group repeats the same leaf labels. **(b)** When an LLM is given (a) as input, it serializes the table into a flat grid and completely loses this critical metadata, resulting in the wrong answer regardless of how powerful the LLM is; no LLM can recover the path. **(c)** A well-designed PARSER can process the source and put the correct solution path back (i.e. $0.556 \rightarrow 0.811$ over solver-only evolution).

that information is lost. This is why simply rewriting the solver cannot address every sub-task of tabular understanding and reasoning. One can of course prepend the header tree by hand, but that is a parser, written once for one benchmark, and it does not scale.

Towards enabling more structured and powerful table reasoning, in this paper we propose a new tabular RSI framework, **TabRSI**, which keeps the Gödel-machine principle at the core, supplied by other specialized roles tailored to the tabular understanding and reasoning domain. Our TabRSI framework consists of a self-improving coding agent, the SOLVER, recursively rewriting itself and retaining only empirically validated edits as its core component, but it is also embedded in an ecosystem of other specialized roles. TabRSI tackles the problem through five modular roles: a PARSER that reconstructs table structures; a PLANNER that interprets questions through entity resolution or schema-linking; a SOLVER that answers directly or executes computations; a VERIFIER that validates entailment or recomputes operations; and a FORMATTER that deterministically coerces outputs into requested formats. TabRSI is, to our knowledge, the first application of recursive self-improvement to table understanding and reasoning, and our results show that the paradigm is effective in this domain: a self-improving solver remains at the core of every deployed configuration, and the search is what decides, per task, whether a stronger solver or an additional role is required.

The TabRSI framework optimizes performance by searching jointly across three orthogonal axes: active pipeline topology, role-specific code implementation, and reasoning modality (PoT vs. CoT) chosen independently per role. These three axes are what let TabRSI tackle the intricacy and unique nature of tabular understanding and reasoning problems, overcoming the intrinsic limitation of a monolithic agent, and they address the two obstacles above in turn. Because modality is chosen per role, nothing has to commit globally to code or to prose, so a sub-task that rewards code execution and one that rewards prose processing can be served by the same pipeline. Because topology is searched, a competence the solver itself can never structurally acquire — say, reconstructing header-tree metadata in a table — is supplied by a role (e.g., the PARSER) that does have the right access, instead of wasting valuable self-improving cycles that are never going to be effective. And because the coding agent itself still self-improves, TabRSI keeps the mechanism that makes RSI effective wherever rewriting code is genuinely the right lever.

We evaluate on seven structured-table benchmarks spanning hierarchical, long-context, numerical, and multi-table reasoning, under three LLM backbones. Monolithic DGM and HGM agents are not reliably better than CoT prompting in the table domain; TabRSI is, improving on a chain-of-thought baseline on all seven benchmarks, by $+11\%$ to $+155\%$ on DeepSeek-v4-flash and $+6\%$ to $+143\%$ on Qwen3.8-27B. The limitation is therefore not RSI itself, but the narrowness of what a monolithic RSI agent is allowed to search over. Unlike in coding benchmarks, the effect of different search rules such as greedy or Clade-Metaproductivity is minimal compared with that of introducing multiple specialized sub-agents: the two search rules differ by margins at or below run-to-run variation, while the gap between a role team and a monolithic solver is an order of magnitude larger.

2 Related Work

Self-improving and evolutionary code agents. The concept of self-improving and evolving computational machines has deep roots in biology and automata theory, dating back well before the large language model (LLM) era. Early foundational explorations range from universal self-reproducing automata (von Neumann & Burks, 1966) and genetic algorithms (Holland, 1992) to Gödel Machines (Schmidhuber, 2006), which theoretically formalize fully self-referential problem solvers capable of provably optimal self-rewrites. Found on these ideas, recent work on recursive self-improvement uses LLMs to iteratively inspect, modify, and evaluate their own code (Tian et al., 2024). The Darwin Gödel Machine (DGM) (Zhang et al., 2025) maintains an open-ended archive of self-modifying coding agents through greedy empirical selection, while the Huxley-Gödel Machine (HGM) (Wang et al., 2026) introduces Clade-Metaproductivity (CMP), which credits an ancestor with the best score achieved by any agent in its descendant clade. This idea is closely related to broader approaches that use LLMs as mutation operators, including Evolution through Large Models (ELM) (Lehman et al., 2023), FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Novikov et al., 2025), and Eureka (Ma et al., 2024). These methods use LLMs to propose code, algorithms, or reward functions that are evaluated automatically. While these works have made substantial progress in developing self-improving systems, they have largely focused on coding and program synthesis, leaving open whether the same evolutionary mechanisms are effective for structured tabular reasoning, where performance depends on how well the system can leverage structured data, decomposes reasoning, and coordinates multiple computational steps.

Multi-agent role decomposition for reasoning. Decomposing complex multi-step reasoning into specialized functional roles, such as planners, solvers, critics, and verifiers, is widely recognized to improve LLM performance by structuring intermediate cognition (Wang et al., 2023; Shinn et al., 2023; Du et al., 2024; Hong et al., 2024). However, existing architectures almost universally treat these roles as fixed natural language prompts engineered by the researcher, enforcing rigid task decompositions that cannot adapt to the underlying failure modes of a domain. In this paper, we frame multi-agent coordination as a modular pipeline in which each role is instantiated as an independently evolvable code file, allowing the role implementations, pipeline structure, and reasoning modalities to evolve rather than relying on static prompts.

Structured-table reasoning. Structured-table reasoning is a critical capability for real-world intelligence, with its applications ranging from business and financial auditing to scientific discovery (Cheng et al., 2022; Zhao et al., 2022; Grijalba et al., 2024). Unlike unstructured natural language, tabular reasoning requires navigating complex two-dimensional geometries, resolving multi-hierarchical header trees, grounding ambiguous natural language questions in relational schemas, and performing exact numeric calculations where generative hallucinations prove fatal (Chen et al., 2023; Cheng et al., 2023). Existing table-reasoning frameworks primarily rely on static prompt paradigms, such as comparing textual Chain-of-Thought (CoT) Wei et al. (2022) with executable Program-of-Thought (PoT) Chen et al. (2023) or text-to-SQL generation (Chen et al., 2023; Grijalba et al., 2024), or deploy fixed neural-symbolic architectures with hardcoded semantic parsers and execution pipelines (Cheng et al., 2023). In this work, we introduce self-improving LLM-based agents to this task to automatically discover and refine effective reasoning pipelines. Rather than relying on static prompts or fixed architectures, we model table reasoning as a modular pipeline of specialized roles, whose implementations and interactions can evolve based on task performance.

3 Preliminaries

We first introduce two recent Gödel-machine-based self-improving frameworks this work extends, and then describe how we apply them as the solver of a table-reasoning agent.

Darwin Gödel Machine (DGM) and Huxley Gödel Machine (HGM). Both of these methods are self-improving coding agents whose own source code is constantly being optimized. Whereas Schmidhuber's original Gödel machine required a *proof* that a rewrite is beneficial, both of these Gödel machines replace that proof with measurement: a rewrite that is kept if it scores better on a benchmark the agent can run itself, which is what makes them practical as coding agents.

DGM (Zhang et al., 2025) maintains an archive of agents, organized as a tree of self-modifications. At each step it samples an agent from the archive, asks a frozen foundation model to write an improved version of that agent's code, evaluates the child on a coding benchmark, and adds it to the archive. Because the archive is kept rather than collapsed to a single incumbent, several lineages can be explored in parallel, and the search is driven by each agent's *own* measured performance. On SWE-bench this takes a coding agent from 20% to 50%.

HGM (Wang et al., 2026) keeps the same target, essentially one evolving agent script, and changes only *which* node the search expands next. Its starting observation is a mismatch: an agent's own benchmark score is a poor predictor of how productive its *descendants* will be, so expanding whatever currently scores highest discards stepping stones that would have paid off a few edits later. HGM therefore scores a node by *Clade-Metaproductivity* (`CMP`) that aggregates the performance of the descendants it has actually produced so far, and expands the lineages whose clade looks most productive, trading that against how little each node has been tried. On coding benchmarks, specifically SWE-bench Verified and Polyglot, HGM reaches higher performance than prior self-improving agents, including DGM, at lower compute cost. The difference between the two is therefore not what they modify but what signal drives the search: DGM expands on an agent's own score, HGM on its clade's. Everything else such as the frozen proposer, the empirical accept test, the archive is shared by both, which is what lets us state the two as a single procedure with the selection rule as a switch: Algorithm 1 (Appendix A) gives that procedure, with greedy keeping a single incumbent that a child replaces only when it scores better, and `CMP` keeping every child in the archive and selecting the next node by clade score plus an exploration bonus.

Coding agents for table understanding and reasoning. We apply both DGM and HGM in their original form they were designed for, but now in the context of table understanding and reasoning as it follows. The evolving script *is* the coding agent SOLVER, the main coding agent that reads a serialized table and a question and emits an answer. The meta-agent is a frozen LLM that proposes rewrites of the coding agent's script conditioned on a sample of its recent failures; a rewrite is accepted on the benchmark's own scorer over a held-out training sample, and the agent we finally deploy is the archive node that scores best on a disjoint validation split, so a benchmark is either improved or left exactly at its seed. Only the benchmark and the scorer change relative to the coding setting. This is the direct transfer we evaluate in §5.2, and its limits, which includes a solver that rewrites itself cannot recover information its input never carried, nor switch reasoning modality per sub-task, are what motivate the role team in TabRSI.

4 TABRSI

In the DGM and HGM workflows, a single monolithic SOLVER must do every job inside one evolving script: read the question, interpret the table, compute an answer, format it as the scorer expects, and do whatever else the task demands. A mistake in any one of those jobs is often unfixable within the same script that is also doing the arithmetic. TabRSI-DGM and TabRSI-HGM extend DGM and HGM by replacing that single script with a *team* $T = (\mathcal{R}, c, m)$ built around the SOLVER, in which every role is independently evolvable by the same search machinery, so the search ranges over the team rather than over the SOLVER alone.

The five roles $\mathcal{R} = \{\text{PARSER}, \text{PLANNER}, \text{SOLVER}, \text{VERIFIER}, \text{FORMATTER}\}$ each have their own mutable code file: the PARSER enriches the question with a computed table digest before anything else runs, the PLANNER prepends a hint about how to approach the question, the SOLVER is the always-on worker, the VERIFIER recomputes and repairs the SOLVER's answer against the table, and the FORMATTER coerces the answer into the shape the scorer expects. The SOLVER always runs, while the other four are optional, switched on and off by the bits $c \in \{0,1\}^4$. Every role also carries a reasoning modality, collected in $m \in \{\text{CoT}, \text{PoT}\}^{|\mathcal{R}|}$ and motivated at the end of this section. The team then runs left to right,

$$T(q) = \big(\text{FORMATTER}^{c}_{m_5} \circ \text{VERIFIER}^{c}_{m_4} \circ \text{SOLVER}_{m_3} \circ \text{PLANNER}^{c}_{m_2} \circ \text{PARSER}^{c}_{m_1}\big)(q), \qquad (1)$$

where a superscripted role runs only if its bit in c is on, a role in CoT reasons in language while one in PoT emits code that is executed, and the VERIFIER and FORMATTER always see the *original* question rather than a version an earlier role has rewritten.

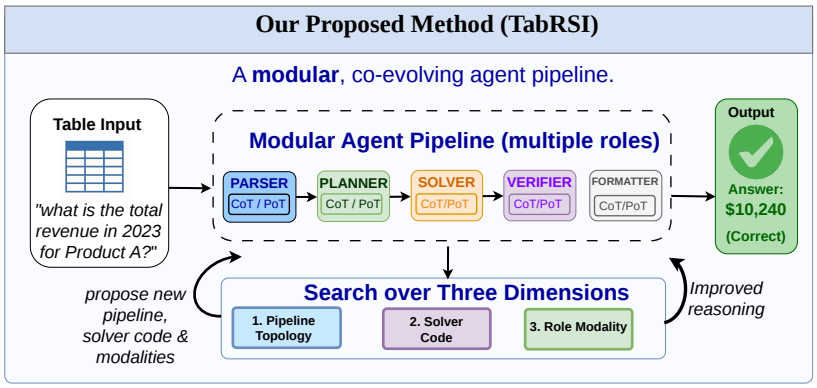

Figure 2: An overview of TabRSI. Unlike existing RSI methods that iteratively refine a single agent, TabRSI uses a modular pipeline with specialized roles and jointly searches over pipeline topology, code, and CoT/PoT modality.

All four systems, DGM, HGM, TabRSI-DGM and TabRSI-HGM, run the same evolution procedure for the core SOLVER. In each of them, the solver-evolution step of Algorithm 1 (Appendix A) proposes a rewrite and accepts or rejects it: how the next coding-agent node is selected (*greedy*, which always takes the current best, or *CMP*). Greedy is a different rule rather than `CMP` with its exploration turned off: it keeps a single incumbent that a child replaces only when it scores better, whereas `CMP` keeps every child in the archive, and the constant C only scales how far `CMP` strays from the best clade. In the case of TabRSI variants, there are some additional loops to potentially change the whole *team* in addition to the SOLVER's code, as described next. Figure 2 gives a high-level architectural view of how the TabRSI framework works.

Team update and self-evolving mechanism for TabRSI-DGM and TabRSI-HGM. Three operators edit a team, kept deliberately separate so their effects can be measured apart: `topo` changes *which* roles run, `mutR` changes *what* a role says, and `mutM` changes *how* it says it. `topo` flips one optional role's on/off bit, with no code change and no LLM call. `mutR` shows the meta-agent one role's code together with the *team's* recent failures and asks for a rewrite, $\text{code}'_r \leftarrow M(\text{code}_r, \text{fails}(T))$; rewriting the VERIFIER or FORMATTER also switches it on, so the edit is actually used. `mutM` flips exactly one role between COT and POT and re-seeds only that role, leaving every other role's already-evolved code untouched. The self-improving solver of §3 is not replaced by any of this: the SOLVER is always on and always mutable, and DGM's solver-only search is exactly the special case where the team is {SOLVER} and only `mutR` is available.

Algorithm 2 (Appendix A) gives the TabRSI-HGM loop in full; it repeats four steps until the budget is spent. **(1) Select.** Pick the team to expand by its `CMP` score plus an exploration bonus, so a lineage is worth expanding either because it has already produced a strong result or because it has barely been tried. This is the only place exploration enters: the constant C scales it, and at $C{=}0$ the rule expands the best clade with no exploration at all. **(2) Propose.** Draw one of the three operators from disjoint slices of a single random number, so each is sampled at its own fixed rate: with probability p a `topo` flip, which turns one optional role on or off with no LLM call; with probability q a `mutM` flip, which switches one role between prose and code and re-seeds that role alone; otherwise a `mutR` rewrite, in which the meta-agent rewrites one role's code from the team's recent failures. Exactly one role changes, and the child is discarded if it fails to compile or breaks the interface the pipeline calls. **(3) Evaluate.** Score the child on the training sample, and perform a validation pass only when training improves, which keeps the judge cost bounded; a modality flip must additionally clear a margin δ or it is dropped, since `mutM` re-seeds a role and can otherwise trade an evolved role for a fresh one on noise. **(4) Insert.** Add the child to the archive *whether or not it is better* and refresh `CMP` along the path back to the root, so a team that looks unpromising now is still credited if a descendant later succeeds. At the end the deployed agent is whichever archive node scores best on validation, and because the untouched seed is always in the archive, a benchmark is either improved or left exactly where it started.

Table 1: Summary of the performance of the selected solution by our method against the CoT baseline. As indicated, the recursive self-improvement solution introduced by our method outperforms CoT in most cases across different datasets and LLM backbones.

	DeepSeek-v4-flash		Qwen3.8-27B		Qwen3.5-9B	
Benchmark	CoT	TabRSI (Best)	CoT	TabRSI (Best)	CoT	TabRSI (Best)
DATABENCH	0.328	**0.838**	0.346	**0.841**	0.258	**0.782**
HITAB	0.574	**0.811**	0.663	**0.840**	0.620	**0.833**
MULTIHIERTT	0.576	**0.637**	0.601	**0.664**	0.125	**0.250**
MMTU	0.563	**0.652**	0.569	**0.698**	0.156	**0.241**
TREB	0.320	**0.441**	0.416	**0.610**	0.148	**0.273**
TABLEBENCH	0.412	**0.537**	0.475	**0.584**	0.236	**0.319**
LONGTABLEBENCH	0.585	**0.650**	0.647	**0.669**	0.188	**0.228**

Why modality is a per-role coordinate. The `mutM` operator exists because CoT (Wang et al., 2023) and PoT (Chen et al., 2023) have complementary strengths on tables: prose carries qualitative and multi-step reasoning, executed code carries precise computation. Applying DGM or HGM directly fixes the SOLVER to whichever style its seed code happened to use, and that choice matters: replacing the SOLVER with a code-based reasoner is a large gain on some datasets and a loss on others, so no single modality is universally right. Routing each *question* to a modality is the obvious alternative but is brittle, since one choice must be right about the whole pipeline at once. We therefore attach modality to each *role*, which is what m in Eq. 1 carries: the VERIFIER can recompute in code while the SOLVER reasons in prose.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETUP

We evaluate our method across three LLM backbones and seven datasets. We study three LLMs as the SOLVER: DeepSeek-v4-flash (Xu et al., 2026), Qwen3.5-9B (Team, 2026) and Qwen3.8-27B (Qwen Team, 2026). For DeepSeek-v4-flash, the same model serves as the *meta-agent* (proposing new code) and *judge* (grading answers), while DeepSeek-v4-flash serves these roles for both Qwen backbones. We evaluate each method across seven structured-table benchmarks spanning dataframe QA (DATABENCH Grijalba et al. (2024)), hierarchical tables (HITAB Cheng et al. (2022)), multi-hierarchical financial QA (MULTIHIERTT (Zhao et al., 2022)), mixed table understanding (MMTU (Xing et al., 2026)), table reasoning (TREB (Li et al., 2026)), table benchmarking (TABLEBENCH (Wu et al., 2025)), and long-context table QA (LONGTABLEBENCH (Li et al., 2025)). We use each benchmark's own native metric: computed-value accuracy for DATABENCH, numeric-aware whole-word accuracy for HITAB, QA accuracy for MULTIHIERTT, the per-task metrics for MMTU, macro ROUGE-L over the twenty table tasks for TREB, and count-weighted category accuracy for TABLEBENCH.

5.2 RESULTS

We compare our method, TabRSI-DGM and TabRSI-HGM, as well as the original DGM and HGM, against CoT prompting and organize our results as seven research questions (RQs).

RQ1. Does RSI work for tabular understanding and reasoning tasks? In particular, does TabRSI outperform the CoT baseline? Table 1 shows that TabRSI consistently outperforms the CoT baseline across all three evaluated LLM backbones and all seven datasets, demonstrating the effectiveness of our framework, and of RSI as a paradigm for tabular understanding and reasoning. The margins are large: the relative gain of the selected solution over CoT ranges from $+11\%$ to $+155\%$ on DeepSeek-v4-flash and from $+6\%$ to $+143\%$ on Qwen3.8-27B (last column of Tables 2 and 3). For simplicity, the full per-method results for Qwen3.5-9B are reported only in Appendix D. These results also support the view that combining a strong RSI paradigm with a correct harness designed for tabular understanding and reasoning is the pathway to tackling the intricacy of this problem domain.

Table 2: Results obtained by using DeepSeek-v4-flash as the LLM backbone, strict scoring throughout (MMTU under its official per-subtask metrics, TREB under official ROUGE-L). The last column is the relative gain of the selected solution over the CoT baseline. Judge-scored counterparts for every method are in Table 9.

Benchmark	CoT	DGM	TabRSI -DGM	HGM	TabRSI -HGM	Δ vs. CoT
DATABENCH	0.328	0.828	0.824	0.824	**0.838**	+155%
HITAB	0.574	0.556	**0.811**	0.575	0.804	+41%
MULTIHIERTT	0.576	0.570	**0.637**	0.560	0.621	+11%
MMTU	0.563	0.547	0.577	0.582	**0.652**	+16%
TREB	0.320	0.404	**0.441**	0.408	0.433	+38%
TABLEBENCH	0.412	0.509	**0.537**	0.466	0.529	+30%
LONGTABLEBENCH	0.585	**0.655**	0.584	0.619	0.650	+12%

Table 3: Results obtained by using Qwen3.8-27B as the LLM backbone, strict scoring throughout. The last column is the relative gain of the selected solution over the CoT baseline. Judge-scored counterparts for every method are in Table 19.

Benchmark	CoT	DGM	TabRSI -DGM	HGM	TabRSI -HGM	Δ vs. CoT
DATABENCH	0.346	0.834	0.828	0.825	**0.841**	+143%
HITAB	0.663	0.657	0.839	0.668	**0.840**	+27%
MULTIHIERTT	0.601	0.602	0.657	0.605	**0.664**	+10%
MMTU	0.569	0.674	**0.698**	0.671	0.696	+23%
TREB	0.416	0.467	0.595	0.580	**0.610**	+47%
TABLEBENCH	0.475	0.548	**0.584**	0.554	0.577	+23%
LONGTABLEBENCH	0.647	**0.687**	0.669	0.641	0.629	+6%

RQ2. Does TabRSI perform better than the original Gödel-machine-based coding agents, DGM and HGM? Tables 2, 3 and 17 show TabRSI's over-performance compared to DGM and HGM: on every dataset except LONGTABLEBENCH, the best-performing method is either TabRSI-DGM or TabRSI-HGM. Adding specialized roles to a solo solver helps on almost every benchmark and on all three backbones, for both DGM and HGM, but by different degrees. It is more decisive on HITAB, MULTIHIERTT, MMTU and TABLEBENCH, and marginal on DATABENCH, where the solo code solver already reaches within 0.010 of the best score any method attains, and on TREB. LONGTABLEBENCH is the single exception, and we believe this is because its problems are closer to data engineering and data science workflow automation, which a traditional LLM solver already handles well.

RQ3. Is the CMP-credited search rule proposed by HGM consistently more effective than the greedy search strategy of DGM for tabular understanding and reasoning? Unlike pure coding cases, our results indicate that the CMP-credited search rule of HGM is not a clear winner over the greedy search strategy of DGM for tabular understanding and reasoning, either for the solo methods or for their TabRSI counterparts. Counting head-to-head wins over the seven benchmarks in Tables 2, 3 and 17, the solo methods split 4/3 in favor of DGM on DeepSeek-v4-flash, 3/4 in favor of HGM on Qwen3.8-27B and 2/5 on Qwen3.5-9B; the TabRSI variants split the same way on each backbone. Which rule wins therefore depends on the benchmark and the backbone rather than on the rule itself, and the gaps are small: across the 42 head-to-head comparisons, the median is 0.015 and two thirds are at or below 0.03, the range of run-to-run variation reported in Appendix B.

This behavior contrasts with the results reported for coding agents, where HGM's CMP-based search has been shown to outperform DGM on coding benchmarks (Wang et al., 2026). This difference sheds light on the distinction between the coding domain for which these methods were proposed and the tabular understanding and reasoning setting studied here.

RQ4. When does the role team (TabRSI) help and which roles help over solo solver evolution? Roles and solver evolution act as *substitutes*, not complements: the team pays exactly where the solver structurally cannot reach the fix, and adds almost nothing where it can. Three mechanisms account for the wins, and Appendix B attributes every clean one to them.

Evolution Budget	Solver	+PARSER	Δ (rel.)
0 (seed)	0.6503	0.8346	+0.184
10	0.6212	0.8403	+0.219
20	0.6402	0.8283	+0.188

Table 4: HITAB, the same DGM ladder scored twice at each budget point: solver alone, and the identical solver behind the un-evolved seed PARSER (n=1,584 per method per point).

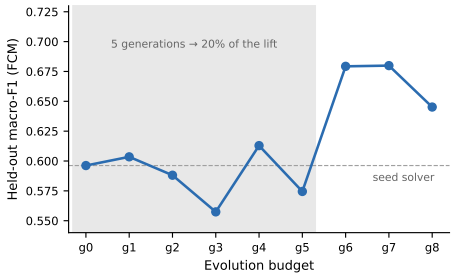

Figure 3: The influence of additional evolution budget on performance for DGM.

(i) Restoring the table structure representation the solver never sees. Both hierarchical benchmarks turn on this mechanism: DGM and HGM sit flat against CoT and only a PARSER lifts them. HITAB gains +0.255/+0.229 for the two families on DeepSeek-v4-flash and +0.182/+0.172 on Qwen3.8-27B; MULTIHIERTT gains +0.067/+0.061 and +0.055/+0.059. The effect is four times larger on HITAB, where the answer is a header path that flattening removes outright (Figure 1). **(ii) Recomputing a checkable answer.** A VERIFIER gives the most reproducible effect in the paper. We observe big performance boost on DeepSeek-v4-flash; MMTU Arithmetic-Relationship $0.516 \rightarrow 0.856$, Table-Locate-by-Row-Col $0.720 \rightarrow 0.970$, Table-needle-in-a-haystack $0.715 \rightarrow 0.992$ and TREB Distribution-Testing $0.934 \rightarrow 0.980$, from an already strong solver. Similarly, we observe a similar strong performance improvement on Qwen3.8-27B, Arithmetic-Relationship $0.519 \rightarrow 0.865$, as an example. Lu et al. (2025) report that LLM verification pays off most on mathematical and logical tasks, and that its benefit shrinks as the verifier comes to resemble the solver. Both halves fit what we observe. Every cell above is computational in nature with an answer the table fixes exactly, and our VERIFIER is not a second opinion drawn from the solver's own distribution. This implies that it is a separately evolved role that recomputes the answer from the table instead of re-judging it in prose. Our modality ablation isolates exactly that distinction since an executable verifier helps while a prose one does not (Table 5). **(iii) Repairing a broken output format.** A FORMATTER moves MMTU semantic-join $0.028 \rightarrow 0.872$ on DeepSeek-v4-flash, the largest single number in this paper, but only $0.812 \rightarrow 0.819$ on Qwen3.8-27B, inside that backbone's 0.023 jitter floor. This is essentially an unscaffolded method already scores 0.779 where DeepSeek's scored 0.028. A role pays only against such deficit when it is actually present which is why this mechanism alone does not transfer all the time.

We observe that across all 106 (cell, family) comparisons on DeepSeek-v4-flash, roles help in 37, tie in 63 and hurt in 6, every loss belonging to DGM, which applies its team unconditionally where HGM gates per subtask. On Qwen3.8-27B, we obseve that 11 of the 18 cells carrying a role help and none hurt. Roles buy nothing when executable code already reaches the answer. DATABENCH's methods, for instance, span 0.014 and 0.016 on the two strict-scored backbones inside its ± 0.010 jitter floor, and LONGTABLEBENCH is the one benchmark where the team is a net cost. Qwen3.8-27B's MMTU decomposition puts a number on the split (Table 25), because of the $+0.1289$ total over CoT, the scaffold is worth 81% while the role team 19% and the search rule 2%. Roles are real but concentrated and therefore not a blanket improvement.

RQ5. Does spending more evolution budget consistently improve final performance? Our results show that a larger evolution budget can generally provide gains, but performance does not improve monotonically, and evolution cannot replace appropriate role modalities.

To study the effect of evolution budget on final performance, we vary the solver's budget on the Fuzzy Conditional Manipulation subtask of LONGTABLEBENCH. As shown in Figure 3, a DGM solver without evolution achieves a held-out macro-F1 of 0.596 and improves modestly in the early stages: after 4 evolution steps, only about 20% of the total gain is achieved, but the peak performance is only reached at a budget of 7.

To further examine the effect of evolution budget and the importance of specialized roles, we apply the same DGM evolution to the preprocessing task of HITAB. Interestingly, even after 10 or 20 evolution steps, solver evolution yields no notable performance gain. However, introducing a specialized parser substantially improves performance even without evolution, yielding a 0.184 gain

Table 5: Results of six CoT/PoT compositions on DATABENCH.

Solver	Verifier	Acc.
CoT	none	0.312
CoT	CoT	0.316
CoT	PoT	**0.836**
PoT	none	0.828
PoT	CoT	0.558
PoT	PoT	**0.840**

Table 6: TABLEBENCH, PoT before and after evolution against the un-evolved CoT baseline.

Category	PoT seed	CoT seed	PoT evolved
NumericalReasoning	0.4367	0.7975	0.7722
DataAnalysis	0.2066	0.3573	**0.4512**

over the original solver-only baseline. Evolution up to a budget of 10 further improves performance to 0.6212, while increasing the budget to 20 yields no additional gain. This pattern arises because the seed solver's shortcomings on HITAB's preprocessing task are structural rather than a matter of search: the preprocessing keeps leaf-only headers, so the parent-child grouping the PARSER restores is simply absent from the solver's input, and no amount of code evolution recovers it. Evolution therefore helps when a deficit is computational and better code can address it, and not when the deficit is informational.

RQ6. Does the role modality matter beyond the presence of the roles? Results in Table 5 show that choosing the appropriate role modality can be more important than the mere inclusion of roles.

To study the influence of the CoT/PoT choice per role, we evaluate TabRSI-DGM on DATABENCH across solver and verifier modalities (Table 5). The modality matters more than the presence of the role: a CoT verifier adds nothing to a CoT solver (0.312 to 0.316) and actively hurts a PoT solver (0.828 to 0.558), whereas a PoT verifier lifts both (0.836 and 0.840). A role with the wrong modality is therefore worse than no role at all, which is why TabRSI searches over modalities instead of fixing one in advance.

RQ7. Can evolution overturn the modality it started with? Our results indicate that this depends on the task and setting.

In this RQ, we investigate whether evolution can overcome the limitations of an initially suboptimal modality choice. To this end, we study two subtasks from TABLEBENCH, with results shown in Table 6. In both subtasks, CoT substantially outperforms PoT, making CoT the preferred modality at initialization. Interestingly, in the DATAANALYSIS subtask, evolution improves the PoT solver beyond the performance of the CoT seed solver, showing that evolution can, in some cases, overcome the limitations of an initially suboptimal modality choice. However, this is not consistently the case. On the NUMERICALREASONING subtask, even after evolution, the PoT solver fails to surpass the performance of the CoT seed solver. These results further highlight the complementary importance of recursive self-improvement and searching over execution modalities as two key components of our approach to tabular understanding and reasoning.

6 CONCLUSION

We introduced TABRSI, a recursive self-improving multi-agent framework for structured table understanding and reasoning that extends the Gödel Machine paradigm beyond monolithic solver evolution. By jointly searching over pipeline topology, role-specific implementations, and reasoning modalities, TABRSI can allocate different capabilities to specialized roles while retaining recursive self-improvement at its core. Across seven benchmarks and three LLM backbones, our results show that this broader search space consistently improves over standard CoT prompting in most settings, while also revealing why direct applications of DGM and HGM do not reliably transfer to table reasoning. In particular, performance gains arise primarily when the searched roles or evolving programs address the actual source of a task's errors, whereas the choice between greedy and CMP-based search has a comparatively smaller effect. These findings suggest that effective recursive self-improvement in table understanding and reasoning depends less on how the search is run than on what the search is allowed to change: the structure of the pipeline, the capabilities assigned to each role, and the reasoning modality each role uses.

REFERENCES

Wenhu Chen, Xueguang Ma, Xinyi Wang, and William W. Cohen. Program of thoughts prompting: Disentangling computation from reasoning for numerical reasoning tasks. *Transactions on Machine Learning Research (TMLR)*, 2023.

Zhoujun Cheng, Haoyu Dong, Zhiruo Wang, Ran Jia, Jiaqi Guo, Yan Gao, Shi Han, Jian-Guang Lou, and Dongmei Zhang. HiTab: A hierarchical table dataset for question answering and natural language generation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1094–1110, 2022.

Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. Binding language models in symbolic languages. In *International Conference on Learning Representations (ICLR)*, 2023.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, pp. 11733–11763, 2024.

Jorge Osés Grijalba, L Alfonso Urena Lopez, Eugenio Martínez-Cámara, and Jose Camacho-Collados. Question answering over tabular data with databench: A large-scale empirical evaluation of llms. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pp. 13471–13488, 2024.

John H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. MIT Press, 1992.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenglin Chen, Jie Zeng, Xiangru Tang, Yuhang Shi, Xiao Tang, and Chenglin Wu. MetaGPT: Meta programming for a multi-agent collaborative framework. In *International Conference on Learning Representations (ICLR)*, 2024.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O Stanley. Evolution through large models. In *Handbook of evolutionary machine learning*, pp. 331–366. Springer, 2023.

Ce Li, Xiaofan Liu, Zhiyan Song, Ce Chi, Boshen Shi, Chen Zhao, Guanguang Chang, Zhendong Wang, Kexin Yang, Xing Wang, et al. Treb: A comprehensive benchmark for evaluating table reasoning capabilities of large language models. In *Proceedings of the 49th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 3267–3275, 2026.

Liyao Li, Jiaming Tian, Hao Chen, Wentao Ye, Chao Ye, Haobo Wang, Ningtao Wang, Xing Fu, Gang Chen, and Junbo Zhao. Longtablebench: Benchmarking long-context table reasoning across real-world formats and domains. In *EMNLP (Findings)*, pp. 11927–11965, 2025.

Jack Lu, Ryan Teehan, Jinran Jin, and Mengye Ren. When does verification pay off? a closer look at LLMs as solution verifiers. *arXiv preprint arXiv:2512.02304*, 2025.

Yecheng Jason Ma, William Liang, Guanzhi Wang, De-An Huang, Osbert Bastani, Dinesh Jayaraman, Yuke Zhu, Jim Fan, et al. Eureka: Human-level reward design via coding large language models. In *International conference on learning Representations*, volume 2024, pp. 26516–26560, 2024.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Mohammadamin Barekatain, Francisco J. R. Ruiz, others, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026. URL `https://qwen.ai/blog?id=qwen3.8`.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Jürgen Schmidhuber. Gödel machines: Fully self-referential optimal universal self-improvers. In *Artificial General Intelligence*, pp. 199–226. Springer, 2006.

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

Qwen Team. Qwen3. 5-omni technical report. *arXiv preprint arXiv:2604.15804*, 2026.

Ye Tian, Baolin Peng, Linfeng Song, Lifeng Jin, Dian Yu, Lei Han, Haitao Mi, and Dong Yu. Toward self-improvement of llms via imagination, searching, and criticizing. *Advances in Neural Information Processing Systems*, 37:52723–52748, 2024.

John von Neumann and Arthur W. Burks. *Theory of Self-Reproducing Automata*. University of Illinois Press, 1966.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 2609–2634, 2023.

Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-g" odel machine: Human-level coding agent development by an approximation of the optimal self-improving machine. In *International Conference on Learning Representations*, volume 2026, pp. 80356–80386, 2026.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Xianjie Wu, Jian Yang, Linzheng Chai, Ge Zhang, Jiaheng Liu, Xeron Du, Di Liang, Daixin Shu, Xianfu Cheng, Tianzhen Sun, et al. Tablebench: A comprehensive and complex benchmark for table question answering. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pp. 25497–25506, 2025.

Junjie Xing, Yeye He, Mengyu Zhou, Haoyu Dong, Shi Han, Lingjiao Chen, Dongmei Zhang, Surajit Chaudhuri, and HV Jagadish. Mmtu: A massive multi-task table understanding and reasoning benchmark. *Advances in Neural Information Processing Systems*, 38, 2026.

Anyi Xu, Bangcai Lin, Bing Xue, Bingxuan Wang, Bingzheng Xu, Bochao Wu, Bowei Zhang, Chaofan Lin, Chen Dong, Chenchen Ling, et al. Deepseek-v4: Towards highly efficient million-token context intelligence. *arXiv preprint arXiv:2606.19348*, 2026.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Gödel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

Yilun Zhao, Yunxiang Li, Chenying Li, and Rui Zhang. MultiHiertt: Numerical reasoning over multi hierarchical tabular and textual data. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6588–6600, 2022.

APPENDIX A FULL ALGORITHMS

Algorithm 1 EVOLVE(select, space; budget G)

$\mathcal{A} \leftarrow \{c_0\}$; incumbent $\leftarrow c_0$ ▷ seed $c_0 = (\text{seed-solver}, r{=}\mathbf{0})$
for $g = 1$ to G **do**
  if select = greedy **then** $u \leftarrow$ incumbent
  else if select = CMP **then** $u \leftarrow \arg\max_{n \in \mathcal{A}} \left[\mathrm{CMP}(n) + C\sqrt{\ln(\sum \mathrm{visits})/\mathrm{visits}(n)}\right]$; $\mathrm{visits}(u) {+}{=} 1$
  end if
  if space = solo **then** $c' \leftarrow M.\mathsf{rewrite}(u.\textsc{Solver} \mid \mathrm{fails}(u))$
  else if space = flow **then** op $\leftarrow$ `topo` w.p. p else `mutR`; $c' \leftarrow \mathrm{op}(u)$
  end if
  if $\neg\,\mathsf{valid}(c')$ **then continue**
  end if
  $\tau \leftarrow \widehat{\mathrm{acc}}_{\text{train}}(c')$ ▷ val computed on improvement
  if select = greedy **then if** $\tau > \widehat{\mathrm{acc}}_{\text{train}}(\text{incumbent})$ **then** incumbent $\leftarrow c'$
  else if select = CMP **then** attach c' as a child of u in $\mathcal{A}$ ▷ kept even if worse
  end if
end for
return $\hat{a} = \arg\max_{a \in \mathcal{A} \cup \{c_0\}} \hat{g}(a)$ ▷ seed-floored deploy

Algorithm 2 TabRSI-HGM: co-evolve roles + composition + modality under CMP

Require: seed solver s_0, roles $\mathcal{R}$, budget G, exploration C, rates p (`topo`), q (`mutM`), modality margin δ
1: $T_0 \leftarrow (s_0,\ c{=}\mathbf{0},\ m{=}m_0)$ ▷ SOLVER-only seed team; m_0 = seed modality prior
2: $u_0 \leftarrow \textsc{NewNode}(T_0)$; $\mathrm{val}(u_0) \leftarrow \widehat{\mathrm{acc}}_V(T_0)$; $\mathrm{CMP}(u_0) \leftarrow \mathrm{val}(u_0)$; $\mathcal{A} \leftarrow \{u_0\}$
3: **for** $g = 1$ to G **do**
  (1) Select a node by UCT over CMP.
4: $N \leftarrow \sum_{u \in \mathcal{A}} \mathrm{visits}(u)$
5: $u^\star \leftarrow \arg\max_{u \in \mathcal{A}} \left[\mathrm{CMP}(u) + C\sqrt{\ln(N+1)/\mathrm{visits}(u)}\right]$; $\mathrm{visits}(u^\star)$ +$= 1$
  (2) Propose a child — `topo` (which roles), `mutM` (how one runs), `mutR` (what it says).
6: $z \leftarrow \mathrm{rand}()$
7: **if** $z < p$ **then** ▷ `topo`: composition, no LLM call
8: $c' \leftarrow$ flip one non-solver bit of $T_{u^\star}.c$; $T' \leftarrow (T_{u^\star}.s,\ c',\ T_{u^\star}.m)$
9: **else if** $z < p + q$ **then** ▷ `mutM`: CoT⇄PoT, Eq. 1
10: pick a role r with its bit ON; $m' \leftarrow T_{u^\star}.m$ with $m'_r \leftarrow \neg m_r$
11: $\mathrm{code}'_r \leftarrow M\big(\mathrm{seed}_{r,m'_r},\ \mathrm{fails}(T_{u^\star})\big)$; $T' \leftarrow T_{u^\star}$ with r re-seeded, $m \leftarrow m'$ ▷ every other role keeps its evolved code
12: **else** ▷ `mutR`: role-code rewrite
13: pick a role $r \in \mathcal{R}$; $\mathrm{code}'_r \leftarrow M\big(\mathrm{code}_r,\ \mathrm{fails}(T_{u^\star})\big)$
14: $T' \leftarrow T_{u^\star}$ with role r's code replaced; **if** $r \in \{\textsc{Ver}, \textsc{Fmt}\}$ **then** set its bit ON
15: **end if**
16: **if** $\neg\big(\mathsf{compiles} \wedge \mathsf{has_class}_r \wedge \mathsf{has_method}_r\big)$ **then continue**
17: **end if**
  (3) Evaluate the team (pipeline forward pass, Eq. 1).
18: $\tau \leftarrow \widehat{\mathrm{acc}}_{\text{train}}(T')$; **if** τ improves **then** $\mathrm{val}(T') \leftarrow \widehat{\mathrm{acc}}_V(T')$ **else** $\mathrm{val}(T') \leftarrow \tau$
19: **if** $z \in [p, p{+}q)$ **and** $\mathrm{val}(T') \le \mathrm{val}(T_{u^\star}) + \delta$ **then continue** ▷ margin gate, §4
20: **end if**
  (4) Insert and propagate CMP up the lineage.
21: $v \leftarrow \textsc{NewNode}(T')$ as a child of $u^\star$; $\mathrm{val}(v), \mathrm{CMP}(v) \leftarrow \mathrm{val}(T')$; $\mathcal{A} \leftarrow \mathcal{A} \cup \{v\}$
22: **for** w on the path $v \to u_0$ **do** $\mathrm{CMP}(w) \leftarrow \max\big(\mathrm{CMP}(w),\ \mathrm{val}(v)\big)$
23: **end for**
24: **end for**
25: **return** $\hat{a} \leftarrow \arg\max_{u \in \mathcal{A} \cup \{u_0\}} \mathrm{val}(u)$ ▷ seed-floor: $\hat{a}$ is val-no-worse than SOLVER-only

Section 3 introduces the unifying algorithm that covers DGM, HGM and their TabRSI variants; Algorithm 1 formalizes it. Algorithm 2 then gives TabRSI-HGM in full, the system of which the other three are restrictions, including the modality operator `mutM` and its margin gate. Line 5 is the only place exploration enters: a node is worth expanding if its lineage already contains something strong *or* it simply has not been visited much – the property a purely greedy rule lacks. Lines 6–17

propose one of three edits, drawn from disjoint slices of a single random draw: a one-bit composition change (cheap, no LLM call), a one-role modality flip that re-seeds only that role and leaves every other role's evolved code untouched, or an LLM rewrite of one role conditioned on the team's failures. Rewriting the VERIFIER/FORMATTER turns it on so the edit is actually used, and any candidate that fails to compile or breaks the expected interface is thrown out for free. Line 18 scores the child by running the full pipeline (Eq. 1) on the training sample, only paying for a validation pass when training improves, to keep the LLM-judge cost bounded. Unlike greedy search, the child is *always* kept in the archive (lines 21–23), and `CMP` is refreshed along the path back to the root in time proportional to the tree's depth. Line 25 deploys whichever node – including the untouched seed – scores best on validation, which is exactly what makes the seed-floor guarantee below hold.

APPENDIX B WHERE THE ROLE TEAM WORKED, AND WHY

Rather than reprint every per-subtask number here (the complete tables are in Appendix C), we collect the clearest cases where the role team beat *its own solo method* — the same search family, the same evolved solver, the same held-out rows — since the mechanism behind each is worth stating once, plainly. The tables below are highlights, not the full set; the paragraph after them gives the counts, so the reader can see what fraction of the evidence they represent.

Table 7: Role-attributable gains on **DeepSeek-v4-flash**: the role-team method minus its own solo method, within one search family, on the same held-out rows. MMTU rows are judge-scored (the judge grades the accuracy subtasks; the F1 subtasks keep strict F1, since a judge cannot represent F1 over set-valued answers), everything else strict.

Benchmark	Task / cell	family	solo	TabRSI	roleset
*(1) The answer was right and the **format** was rejected* — FORMATTER					
MMTU	semantic-join	HGM	0.028	**0.872**	`fmt`
MMTU	Table-Fact-Verification	DGM	0.527	**0.907**	blanket
MMTU	Entity-Matching	DGM	0.850	**0.986**	blanket
TABLEBENCH	NumericalReasoning×TCoT	HGM	0.468	**0.620**	`plan+verify+fmt`
*(2) The answer was **recomputable*** — VERIFIER					
MMTU	Arithmetic-Relationship	HGM	0.516	**0.856**	`verify`
MMTU	Table-Locate-by-Row-Col	HGM	0.720	**0.970**	`verify`
MMTU	Table-needle-in-a-haystack	DGM	0.715	**0.992**	blanket
TREB	Distribution-Testing	HGM	0.934	**0.980**	`verify`
TREB	Distribution-Testing	DGM	0.397	**0.936**	blanket
*(3) The **representation** was lossy* — PARSER					
HITAB	(single task)	DGM	0.556	**0.811**	`parse`-led
MULTIHIERTT	(single task)	DGM	0.570	**0.637**	`parse`-led
MMTU	Columns-property-anotation	DGM	0.228	**0.476**	blanket
*(4) The solver **could not be evolved at all** — roles as the only lever*					
TREB	Title-Naming	HGM	0.578	**0.883**	`parse+plan+fmt`
TABLEBENCH	Visualization×TCoT	HGM	0.550	**0.750**	`parse+plan+verify+fmt`
TABLEBENCH	FactChecking×SCoT	DGM	0.605	**0.842**	blanket
TABLEBENCH	DataAnalysis×TCoT	DGM	0.260	**0.336**	`plan`

How representative these highlights are. Across the three multi-cell benchmarks on DeepSeek-v4-flash there are 106 (cell, family) comparisons of a role-team method against its own solo method. Roles **help** in 37, are a **tie** in 63, and **hurt** in 6, for a summed delta of $+5.7$. The split by benchmark is MMTU $17/20/5$, TREB $8/31/1$ and TABLEBENCH $12/12/0$. Outside MMTU the role team costs a cell exactly once, on TREB Correlation-Analysis ($0.599 \rightarrow 0.583$ at $n{=}25$). All five MMTU losses belong to DGM, which applies `parse+plan+verify+fmt` to every subtask unconditionally, and four are severe — equi-join-detect $0.676 \rightarrow 0.130$, Error-Detect $0.230 \rightarrow 0.000$, Functional-Dependency $0.728 \rightarrow 0.457$, Data-transform-reshape $0.525 \rightarrow 0.371$. HGM gates per subtask and ships roles-off on exactly those, which is the concrete reason its macro leads. On Qwen3.8-27B, counting only cells that carry a role at all, roles help in 5 of 12 MMTU cells and 6 of 6 TABLEBENCH cells, and hurt in none.

Table 8: The same accounting on **Qwen3.8-27B**, restricted to cells that actually carry a role. The measured jitter floor on MMTU is 0.023 — the largest solo-versus-roles difference across the subtasks where the two methods are the *identical* configuration, so a difference below it is measurement noise alone. On HiTab and MultiHiertt the methods come from separate runs, each of which also scored the unevolved seed; those replicates span 0.010 and 0.006.

Benchmark	Task / cell	family	solo	TabRSI	roleset
Recomputable — Verifier					
MMTU	Arithmetic-Relationship	both	0.519	**0.865**	`verify+fmt`
MMTU	List-to-table	both	0.567	**0.667**	`plan+verify+fmt`
Lossy representation — Parser					
HiTab	(single task)	DGM	0.657	**0.839**	`parse`-led
HiTab	(single task)	HGM	0.668	**0.840**	`parse`-led
MultiHiertt	(single task)	HGM	0.605	**0.664**	`parse`-led
MMTU	header-value-matching	HGM	0.746	**0.773**	`parse+fmt`
TableBench	DataAnalysis×PoT	HGM	0.337	**0.359**	`parse`
Roles as the only lever, and whole-benchmark aggregate					
TableBench	Visualization×SCoT	DGM	0.500	**0.700**	`parse+plan+verify+fmt`
TableBench	NumericalReasoning×SCoT	HGM	0.639	**0.677**	`fmt`
TReB	20 tasks, micro ROUGE-L	DGM	0.467	**0.595**	mixed, per task
*Where it did **not** transfer*					
MMTU	semantic-join	HGM	0.812	0.819	`fmt`, below floor
DataBench	(single task)	DGM	0.842	0.828	roles declined
LongTableBench	(whole benchmark)	DGM	0.687	0.669	roles cost

(1) Formatting, not reasoning — and it does not transfer. On DeepSeek-v4-flash, MMTU semantic-join moves $0.028 \rightarrow 0.872$ from the Formatter alone: the solver already found the right pairs, and the official exact-match metric was rejecting the envelope they arrived in. This is the largest single number in the paper, and it is, honestly, a presentation fix — which is exactly why we also report the LLM-judge column throughout §5.2. It is also the clearest case of a gain that exists only because a *particular* backbone had a particular defect: on Qwen3.8-27B the same subtask with the same `fmt` roleset moves $0.812 \rightarrow 0.819$, inside the 0.023 jitter floor, because that backbone's unscaffolded method already scores 0.779 there against DeepSeek's 0.028. There was no envelope left to fix.

(2) Verification is the most transferable role. It is the one mechanism that reproduces across backbones at a similar magnitude: MMTU Arithmetic-Relationship gains $+0.340$ on DeepSeek-v4-flash and $+0.346$ on Qwen3.8-27B, in both cases because the answer is *recomputable* from the table, so a program-aided second opinion can overrule a prose guess. It is also not simply a stand-in for a better solver — on TReB Distribution-Testing the solver had already reached 0.934 and the Verifier still added $+0.046$.

(3) Rebuilding a lossy representation. HiTab and MultiHiertt are hierarchical tables that get flattened into plain text upstream; the Parser restores the structure the flattening destroyed. This is the second mechanism that transfers cleanly: HiTab gains $+0.255$ on DeepSeek-v4-flash and $+0.182$/$+0.172$ on Qwen3.8-27B's two families, against cross-run seed replicates spanning only 0.010. Both benchmarks gain from roles and gain nothing, or slightly less than nothing, from solver evolution alone — the sharpest contrast in the paper. No amount of rewriting the solver recovers information that was never in its input.

(4) Roles as the only available lever. On some cells, several generations of solver mutation find nothing at all — the search keeps rejecting every proposed rewrite — and the role team is the only thing that moves the number. TableBench's TCoT cells on DeepSeek-v4-flash are the clearest instance: Visualization and DataAnalysis both sit at their vanilla score across every solo method, and only the role team moves them. TReB Title-Naming is the same shape with a larger margin, $0.578 \rightarrow 0.883$ under `parse+plan+fmt`.

The counter-rule: roles substitute for solver strength, they do not add to it. Every clean role win above sits on a solver that could not reach the answer by itself. Where the solver already

computes the right answer, roles buy nothing and the deployment floor correctly declines them: on Qwen3.8-27B's DATABENCH the code solver reaches 0.842 and the role-team method 0.828, and on LONGTABLEBENCH the role-team method is *below* its solo method in both families ($0.687 \rightarrow 0.669$ and $0.641 \rightarrow 0.629$). LONGTABLEBENCH is the one benchmark in this study where the role team is a net cost, and §5.2 reports it as such rather than averaging it away.

APPENDIX C FULL RESULTS ON DEEPSEEK-V4-FLASH

This appendix gives the complete per-subtask numbers on DeepSeek-v4-flash behind the headline rows of Table 2, one table per benchmark, every method in every table. Methods are labelled as the following: **CoT** is a single unscaffolded call with CoT prompting, **DGM** and **HGM** are the two published monolithic self-improving agents, and **TabRSI-DGM** and **TabRSI-HGM** are the corresponding role-team methods.

Three points are important when interpreting these results. First, when a method is not selected by the deployment floor, it copies the method immediately below it, so its delta is exactly 0.000 by construction rather than due to sampling noise. This accounts for the flat TREB rows across the five columns. Second, temperature-0 evaluation remains nondeterministic (§5.1); consequently, results with $n \lesssim 40$ should be interpreted as directional, while our main claims are based on the macro rows. Finally, DGM applies the same role team to every subtask, whereas HGM selects the role team separately for each subtask. This distinction explains why TabRSI-DGM produces both the largest individual gains and all of the severe MMTU collapses.

Table 9: All seven benchmarks at method level on DeepSeek-v4-flash, under both scorers — the roll-up the per-benchmark tables below expand. Metrics are benchmark-native and comparable *within* a row, never across rows. TREB carries no judge column by design: it is reported on its official ROUGE-L, and re-scoring it with a judge would replace the official instrument rather than check it. On MMTU and LONGTABLEBENCH the judge is applied only where F1 degenerates to exact match; rows with set-valued gold keep strict F1, since a judge returns one holistic score per row and cannot represent F1 over a set.

Benchmark	scorer	n	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
DATABENCH	strict	500	0.328	0.828	0.824	0.824	**0.838**
	LLM judge	500	0.348	0.860	0.860	0.852	**0.862**
HITAB	strict	1,584	0.574	0.556	**0.811**	0.575	0.804
	LLM judge	1,584	0.563	0.553	**0.796**	0.564	0.787
MULTIHIERTT	strict	696	0.576	0.570	**0.637**	0.560	0.621
	LLM judge	696	0.532	0.520	0.579	0.539	**0.608**
LONGTABLEBENCH	strict	1,391	0.585	**0.655**	0.584	0.619	0.650
	LLM judge	1,391	0.604	**0.669**	0.601	0.630	0.664
TREB	ROUGE-L	1,961	0.320	0.404	**0.441**	0.408	0.433
MMTU	strict	17,511	0.563	0.547	0.577	0.582	**0.652**
	LLM judge	17,511	0.521	0.519	0.588	0.540	**0.612**
TABLEBENCH	strict	353	0.412	0.509	**0.537**	0.466	0.529
	LLM judge	353	0.626	0.625	**0.645**	0.546	0.644

Table 10: MMTU on DeepSeek-v4-flash: all 21 subtasks, strict scoring with each subtask's official metric, 17,511 held-out rows. Sorted by the TabRSI-HGM gain over CoT.

Subtask	n	metric	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
semantic-join	70	F1	0.028	0.047	0.822	0.028	**0.872**
Arithmetic-Relationship	464	F1	0.503	0.499	**0.862**	0.516	0.856
Table-needle-in-a-haystack	599	acc	0.694	0.701	0.987	**1.000**	**1.000**
Table-Locate-by-Row-Col	600	acc	0.720	0.712	**0.980**	0.722	0.970
Functional-Dependency	148	F1	0.732	0.728	0.457	**0.784**	**0.784**
Columns-property-anotation	521	acc	0.428	0.417	0.422	**0.463**	**0.463**
List-to-table	600	acc	0.553	0.560	**0.607**	0.583	0.583
Column-type-annotation	526	acc	0.593	0.591	0.586	**0.622**	**0.622**
equi-join-detect	364	F1	0.671	0.676	0.130	**0.691**	**0.691**
Data-transform-reshape	120	acc	0.517	0.433	0.358	0.483	**0.533**
Schema-Matching	419	F1	0.750	0.749	0.748	**0.764**	**0.764**
String-Relationship	409	F1	0.158	0.000	0.000	**0.167**	**0.167**
Data-Imputation	1464	acc	0.613	0.616	0.571	**0.617**	**0.617**
Entity-Matching	3719	acc	0.972	0.971	**0.976**	0.973	0.973
header-value-matching	545	acc	0.671	0.672	**0.676**	0.671	0.671
Formula-prediction-context	3113	acc	0.060	**0.061**	0.060	0.060	0.060
Cell-entity-annotation	534	acc	**0.762**	0.727	0.747	**0.762**	**0.762**
Table-QA	1283	acc	**0.603**	0.546	0.555	0.602	0.602
Table-Fact-Verification	558	acc	**0.876**	0.866	0.873	0.871	0.871
semantic-transform	71	acc	0.689	0.682	**0.702**	0.682	0.682
Error-Detect	1384	F1	**0.232**	0.230	0.000	0.156	0.156
MACRO	**17,511**		**0.563**	**0.547**	**0.577**	**0.582**	**0.652**

Table 11: TREB on DeepSeek-v4-flash: all 20 table tasks, official ROUGE-L, 1,961 held-out rows. Sorted by the best method's gain over CoT.

Task	n	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
Table Distribution Testing	197	0.209	0.397	0.936	0.934	**0.980**
Table Title Naming	200	0.446	**0.883**	**0.883**	0.578	**0.883**
Table Correlation Analysis	25	0.181	0.536	0.536	**0.599**	0.583
Table Query	200	0.174	**0.493**	**0.493**	0.456	0.456
Table Selection	200	0.579	**0.765**	**0.765**	0.579	0.579
Table Outlier Detection	17	0.268	0.268	**0.410**	0.283	0.322
Table Summary	200	0.230	0.343	0.343	0.291	**0.368**
Table Column Naming	148	0.134	**0.236**	**0.236**	0.185	0.185
Table Retrieval	200	0.565	0.560	0.560	**0.654**	**0.654**
Table Domain-specific Operations	95	0.344	0.344	**0.389**	0.344	0.387
Table Hypothesis Testing	17	0.160	**0.188**	**0.188**	0.152	0.152
Table Fact Checking	200	**0.970**	**0.970**	**0.970**	**0.970**	**0.970**
Multi-step Correlation Analysis	11	**0.287**	**0.287**	**0.287**	**0.287**	**0.287**
Multi-step Hypothesis Testing	9	**0.361**	**0.361**	**0.361**	**0.361**	**0.361**
Multi-step Operations	9	**0.177**	**0.177**	**0.177**	**0.177**	**0.177**
Table General Operations	200	**0.322**	0.278	0.278	0.314	0.314
Multi-step Fact Checking	12	**0.750**	**0.750**	**0.750**	**0.750**	**0.750**
Multi-step Retrieval	11	**0.048**	**0.048**	**0.048**	**0.048**	**0.048**
Table Plausibility Verification	6	**0.201**	**0.201**	**0.201**	**0.201**	**0.201**
Multi-step Conditional Calculation	4	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
MACRO (ROUGE-L)	**1,961**	**0.320**	**0.404**	**0.441**	**0.408**	**0.433**

Table 12: TABLEBENCH on DeepSeek-v4-flash: all 12 cells (4 categories $\times$ 3 instruction paradigms), strict scoring. The deployed system gates one paradigm per category.

Category	paradigm	n	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
FactChecking	TCoT	38	0.789	0.816	0.816	**0.842**	**0.842**
FactChecking	PoT	38	0.684	0.763	0.763	0.684	**0.816**
FactChecking	SCoT	38	0.605	0.605	**0.842**	0.763	0.763
NumericalReasoning	TCoT	158	0.411	0.411	0.595	0.468	**0.620**
NumericalReasoning	PoT	158	0.424	**0.627**	**0.627**	0.532	0.532
NumericalReasoning	SCoT	158	0.456	0.462	0.462	0.462	**0.582**
DataAnalysis	TCoT	137	0.260	0.260	**0.336**	0.260	0.318
DataAnalysis	PoT	137	**0.193**	**0.193**	**0.193**	**0.193**	**0.193**
DataAnalysis	SCoT	137	0.136	0.203	**0.309**	0.268	0.268
Visualization	TCoT	20	0.550	0.550	0.650	0.550	**0.750**
Visualization	PoT	20	0.650	0.700	0.700	0.650	**0.750**
Visualization	SCoT	20	0.450	**0.550**	**0.550**	0.450	0.500

Table 13: LONGTABLEBENCH on DeepSeek-v4-flash, 1,391 held-out rows (of 1,771; the other 380 are the train and validation rows the search selected on). *strict* is the benchmark's own macro-F1 with the extraction repaired; *LLM judge* is applied per row by gold shape — strict F1 is kept where the gold is multi-item or a dict, and the judge grades the rows where F1 degenerates to exact match. This is the one benchmark where a solo method wins outright.

Method	strict	LLM judge
CoT	0.585	0.604
DGM	**0.655**	**0.669**
TabRSI-DGM	0.584	0.601
HGM	0.619	0.630
TabRSI-HGM	0.650	0.664

Table 14: LONGTABLEBENCH broken down by length regime and turn type, DeepSeek-v4-flash, judge scorer on the 1,391 held-out rows. The aggregate in Table 13 is the row-count-weighted mean of these six cells. The 8k bin carries 62% of the rows, which is why a benchmark-level tie can hide a large effect at `inf`: DGM beats CoT by $+0.27$ on `inf/multi` and *loses* to it by 0.04 on `8k/multi`.

Method	8k/single	8k/multi	32k/single	32k/multi	inf/single	inf/multi
CoT	0.624	**0.705**	0.506	0.636	0.369	0.302
seed (code/PoT)	0.672	0.613	**0.677**	0.688	0.464	0.560
DGM	**0.796**	0.670	0.675	**0.707**	0.533	**0.570**
TabRSI-DGM	0.707	0.598	0.677	0.623	0.435	0.511
HGM	0.770	0.634	0.675	0.619	0.498	0.516
TabRSI-HGM	0.788	0.676	0.671	0.675	**0.539**	0.539
n	155	712	137	149	87	151

Table 15: The three single-task benchmarks on DeepSeek-v4-flash at method level, with the LLM judge's 95% confidence interval. HITAB, MULTIHIERTT and DATABENCH declare no subtasks, so for these three the method-level figure *is* the full breakdown; HITAB has a nominal tier field but 1,571 of its 1,584 rows sit in one tier. DATABENCH's answer-type split (Table 16) is the one further cut any of them supports. MULTIHIERTT is the one benchmark where the two scorers disagree on which method is best; every CI overlaps there, so we read it as a tie.

Benchmark	Method	strict	judge	judge 95% CI
HITAB (n=1,584)	CoT	0.574	0.563	[0.538, 0.587]
	DGM	0.556	0.553	[0.528, 0.577]
	TabRSI-DGM	**0.811**	**0.796**	[0.776, 0.815]
	HGM	0.575	0.564	[0.539, 0.588]
	TabRSI-HGM	0.804	0.787	[0.766, 0.806]
MULTIHIERTT (n=696)	CoT	0.576	0.532	[0.495, 0.568]
	DGM	0.570	0.520	[0.483, 0.557]
	TabRSI-DGM	**0.637**	0.579	[0.542, 0.615]
	HGM	0.560	0.539	[0.502, 0.576]
	TabRSI-HGM	0.621	**0.608**	[0.571, 0.643]
DATABENCH (n=500)	CoT	0.328	0.348	[0.308, 0.391]
	seed (code/PoT)	**0.830**	**0.868**	[0.836, 0.895]
	DGM	0.828	0.860	[0.827, 0.888]
	TabRSI-DGM	0.824	0.860	[0.827, 0.888]
	HGM	0.824	0.852	[0.818, 0.880]
	TabRSI-HGM	**0.838**	**0.862**	[0.829, 0.889]

Table 16: DATABENCH by answer type on DeepSeek-v4-flash (n=500). *greedy*=one sample, *best-of-8*=execution-consensus over eight, *pass@8*=oracle. List answers carry the remaining headroom.

Answer type	n	greedy	best-of-8	pass@8
boolean	124	0.927	0.911	0.952
number	110	0.791	0.800	0.836
category	83	0.735	0.795	0.843
list[category]	86	0.651	0.721	0.767
list[number]	97	0.557	0.711	0.794
All	**500**	**0.746**	**0.796**	**0.846**

APPENDIX D QWEN CROSS-BACKBONE REPLICATION

This appendix gives the full detail behind RQ1, RQ2, RQ4, and RQ6 in §5.2: a seven-benchmark replication on Qwen3.5-9B, and a replication on Qwen3.8-27B (Table 3), with a deeper, MMTU-specific dive into per-subtask detail and the LLM judge below. Both backbones use DeepSeek-v4-flash as the fixed meta-agent and judge, so any difference from the main-text numbers is attributable to the backbone under test, not to a moving teacher or a moving instrument.

D.1 QWEN3.5-9B, ALL SEVEN BENCHMARKS

The full results of experiments with Qwen3.5-9B using the strict evaluation native to each dataset, as well as the results using LLM as a judge are provided in Tables 17 and 18 respectively. These results also follow the same findings from the other two LLM backbones, which were discussed in RQ1 and RQ2 in Section 5.

In both tables, we observe that TabRSI consistently outperforms the CoT baseline, while the exisitng RSI solutions, DGM and HGM show inferior performance to CoT in some cases. For instance, when using the strict scoring, on HITAB, both DGM and HGM preform weaker than CoT, while their TabRSI variants either match or outperform it. These findings, as well as the results obtained by the other studied backbones indicate the strength of the modular co-evolution as well role modality search offered by TabRSI, which leads to consistent improves in structured tabular understanding and reasoning.

Table 17: Results obtained by using Qwen3.5-9B as the LLM backbone with strict evaluation. The last column is the relative gain of the selected solution over the CoT baseline.

Benchmark	CoT	DGM	TabRSI -DGM	HGM	TabRSI -HGM	Ours (Best)	Δ vs. CoT
DATABENCH	0.258	0.658	**0.782**	0.676	0.712	**0.782**	+203%
HITAB	0.620	0.587	**0.833**	0.567	0.620	**0.833**	+34%
MULTIHIERTT	0.125	0.140	0.150	0.195	**0.250**	**0.250**	+100%
MMTU	0.156	0.230	0.239	0.235	**0.241**	**0.241**	+54%
TREB	0.148	0.166	0.189	0.256	**0.273**	**0.273**	+84%
TABLEBENCH	0.236	0.255	0.304	0.284	**0.319**	**0.319**	+35%
LONGTABLEBENCH	0.188	0.228	0.223	0.204	**0.228**	**0.228**	+21%

Table 18: Results obtained by using Qwen3.5-9B as the LLM backbone with LLM as a judge evaluation. The last column is the relative gain of the selected solution over the CoT baseline.

Benchmark	CoT	DGM	TabRSI -DGM	HGM	TabRSI -HGM	Ours (Best)	Δ vs. CoT
DATABENCH	0.246	**0.816**	**0.816**	0.742	0.746	**0.816**	+232%
HITAB	0.567	0.620	**0.787**	0.573	0.587	**0.787**	+39%
MULTIHIERTT	0.190	0.225	0.235	0.190	**0.345**	**0.345**	+82%
MMTU	0.148	0.213	**0.220**	0.178	0.194	**0.220**	+49%
TREB	0.333	0.423	**0.468**	0.420	0.443	**0.468**	+41%
TABLEBENCH	0.248	0.210	0.183	0.202	**0.254**	**0.254**	+2%
LONGTABLEBENCH	0.196	0.214	0.304	0.228	**0.362**	**0.362**	+85%

D.2 QWEN3.8-27B, ALL SEVEN BENCHMARKS, MMTU IN DEPTH

Table 3 in §5.2 summarizes the deployed methods across all seven benchmarks. Here, we provide a more detailed breakdown: first, a method-level summary under both scorers (Table 19), followed by benchmark-level results for datasets with finer-grained annotations — MMTU by subtask, TREB by task, TABLEBENCH by cell, and LONGTABLEBENCH by length regime and turn type. HITAB, MULTIHIERTT, and DATABENCH do not define subtasks, so their roll-up rows correspond to the complete results. All methods are evaluated by both scorers: the judge re-scores the same persisted predictions without re-running the solver, ensuring that differences between scorers reflect the scoring procedure rather than resampling.

Table 19: Method-level results on all seven benchmarks with Qwen3.8-27B, evaluated under both scorers. On MMTU, the judge scores the 14 accuracy subtasks, while the 7 F1 subtasks retain strict F1, as F1 over set-valued answers cannot be represented by a judge score. For LONGTABLEBENCH, the judge assigns partial credit by rescaling the model's 0–5 verdict, so its judge score is not directly comparable to the DeepSeek-v4-flash result in Table 9; comparisons between backbones should instead use strict scoring. For TABLEBENCH, the two rows use different populations: strict scoring averages over all 9 cells (945 rows), whereas the judge covers only the 6 non-`viz` cells (885 rows), since `viz` requires chart execution. The strict score on these same 6 cells is also reported to provide a like-for-like comparison with the judge score.

Benchmark	scorer	n	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
DATABENCH	strict	500	0.346	0.834	0.828	0.825	**0.841**
	LLM judge	500	0.352	0.862	0.852	0.853	**0.864**
HITAB	strict	1,584	0.663	0.657	0.839	0.668	**0.840**
	LLM judge	1,584	0.645	0.635	**0.810**	0.631	**0.810**
MULTIHIERTT	strict	696	0.601	0.602	0.657	0.605	**0.664**
	LLM judge	696	0.556	0.550	0.602	0.556	**0.612**
LONGTABLEBENCH	strict	1,391	0.647	**0.687**	0.669	0.641	0.629
	LLM judge	1,391	0.674	**0.714**	0.701	0.662	0.645
TREB	ROUGE-L	1,961	0.416	0.467	0.595	0.580	**0.610**
	LLM judge	1,961	0.576	0.614	0.637	0.662	**0.680**
MMTU	strict	630	0.569	0.674	**0.698**	0.671	0.696
	LLM judge	630	0.489	0.669	**0.688**	0.661	0.682
TABLEBENCH	strict (9 cells)	945	0.475	0.548	**0.584**	0.554	0.577
	LLM judge (6 cells)	885	0.549	**0.612**	0.605	0.583	0.590
	strict, same 6 cells	885	*0.412*	*0.516*	*0.521*	*0.476*	*0.486*

The same ordering holds under strict scoring: CoT $<$ HGM $<$ DGM $<$ TabRSI-HGM $<$ TabRSI-DGM.

The results suggest that most of the gains in TabRSI come from the evolved pipeline, while the specialized roles provide a smaller but consistent improvement. In contrast, the effect of the search rule is too small to distinguish from evaluation variability in this experiment. The search returned the unevolved seed on 19 of 21 subtasks, and the two search variants differ by only 0.0022, compared with a ± 0.048 two-sigma interval estimated from 15 repeated runs. Thus, these results do not provide evidence that the choice of search rule, rather than the pipeline structure, is responsible for the observed gains.

Arithmetic-Relationship reproduces the DeepSeek result on the same subtask with the same `verify+fmt` team, making it the most consistent effect across all backbones we evaluated. The other four gated adoptions fall within the observed evaluation noise. This is largely because the original gate was evaluated with a bug that has since been fixed: `semantic-join` was previously limited by an insufficient per-example compute budget on about 22% of its rows. Fixing this issue improved the gate's baseline and substantially reduced the apparent gaps.

Where the evolving pipeline is neutral or harmful. On Functional-Dependency, CoT (0.805) beats every agentic method (best 0.765, -0.039) – the only subtask on this backbone where adding the pipeline is a net loss rather than a tie. On equi-join-detect, CoT (0.629) ties all four agentic methods (0.623–0.637). On Formula-prediction-context, every method, including CoT, is floor-bound at 0.100, so this subtask does not inform any comparison in either direction.

Table 20: MMTU on Qwen3.8-27B: all 22 stored subtasks, strict scoring, 30 rows each. *roleset* is what the deployment floor gated on for that subtask; `none` means the solo and TabRSI methods are the identical configuration there, so their difference is the jitter floor (0.023, the largest such difference observed) rather than an effect.

Subtask	roleset	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
Arithmetic-Relationship	`verify+fmt`	0.431	0.519	**0.865**	0.519	**0.865**
List-to-table	`plan+verify+fmt`	0.533	0.567	**0.667**	0.567	**0.667**
Functional-Dependency	`none`	**0.805**	0.741	0.764	0.765	0.733
header-value-matching	`parse+fmt`	0.519	0.723	0.737	0.746	**0.773**
semantic-join	`fmt`	0.779	0.795	0.803	0.812	**0.819**
semantic-transform	`none`	0.450	0.520	0.527	0.523	**0.530**
equi-join-detect	`none`	0.629	0.623	0.628	0.636	**0.637**
Cell-entity-annotation	`none`	**0.667**	**0.667**	**0.667**	**0.667**	**0.667**
Column-type-annotation	`none`	0.700	**0.800**	**0.800**	**0.800**	**0.800**
Columns-property-anotation	`none`	0.267	**0.433**	**0.433**	0.333	0.367
Data-Imputation	`none`	0.367	**0.400**	**0.400**	**0.400**	**0.400**
Data-transform-reshape	`none`	0.467	**0.733**	**0.733**	**0.733**	**0.733**
Entity-Matching	`none`	**1.000**	**1.000**	**1.000**	**1.000**	**1.000**
Error-Detect	`none`	0.250	**1.000**	**1.000**	**1.000**	**1.000**
Formula-prediction-context	`none`	**0.100**	**0.100**	**0.100**	**0.100**	**0.100**
NL2SQL	`none`	**0.000**	**0.000**	**0.000**	**0.000**	**0.000**
Schema-Matching	`fmt`	0.762	**0.775**	**0.775**	0.762	0.762
String-Relationship	`fmt`	0.233	**0.267**	**0.267**	**0.267**	**0.267**
Table-Fact-Verification	`none`	0.867	**0.967**	**0.967**	**0.967**	**0.967**
Table-Locate-by-Row-Col	`none`	**0.833**	**0.833**	**0.833**	0.800	**0.833**
Table-QA	`none`	0.567	**0.700**	**0.700**	**0.700**	**0.700**
Table-needle-in-a-haystack	`none`	0.733	**1.000**	**1.000**	**1.000**	**1.000**
MACRO (21 scorable)		**0.5694**	**0.6744**	**0.6984**	**0.6713**	**0.6961**

Table 21: TREB on Qwen3.8-27B: all 20 table tasks, official ROUGE-L, 1,961 held-out rows. Sorted by the TabRSI-DGM gain over its own solo method.

Task	n	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
Table_Distribution_Testing	197	0.005	0.373	**0.970**	0.911	0.955
Table_Title_Naming	200	0.497	0.497	0.830	0.826	**0.861**
Table_Query	200	0.130	0.213	0.466	0.476	**0.566**
Table_Summary	200	0.337	0.337	0.417	0.337	**0.426**
Table_Outlier_Detection	17	0.275	0.323	0.381	0.267	**0.451**
Table_Correlation_Analysis	25	0.261	0.567	0.572	0.261	**0.577**
Multi-step_Conditional_Calculation	4	**0.500**	**0.500**	**0.500**	**0.500**	**0.500**
Multi-step_Correlation_Analysis	11	**0.273**	**0.273**	**0.273**	**0.273**	**0.273**
Multi-step_Fact_Checking	12	**0.583**	**0.583**	**0.583**	**0.583**	**0.583**
Multi-step_Hypothesis_Testing	9	**0.351**	**0.351**	**0.351**	**0.351**	**0.351**
Multi-step_Operations	9	**0.111**	**0.111**	**0.111**	**0.111**	**0.111**
Multi-step_Retrieval	11	**0.272**	**0.272**	**0.272**	**0.272**	**0.272**
Table_Column_Naming	148	**0.222**	**0.222**	**0.222**	0.169	0.169
Table_Domain-specific_Operations	95	**0.529**	**0.529**	**0.529**	**0.529**	0.488
Table_Fact_Checking	200	**0.975**	**0.975**	**0.975**	**0.975**	**0.975**
Table_General_Operations	200	0.345	0.343	0.343	**0.411**	**0.411**
Table_Hypothesis_Testing	17	0.132	0.262	0.262	**0.272**	**0.272**
Table_Plausibility_Verification	6	**0.255**	**0.255**	**0.255**	**0.255**	**0.255**
Table_Retrieval	200	**0.598**	**0.598**	**0.598**	**0.598**	**0.598**
Table_Selection	200	**0.606**	**0.606**	**0.606**	**0.606**	**0.606**
MICRO (row-weighted)	**1,961**	**0.4160**	**0.4666**	**0.5951**	**0.5797**	**0.6096**

Table 22: TABLEBENCH on Qwen3.8-27B: all 9 cells (3 categories × 3 paradigms), strict. *roleset* is TabRSI-DGM's; a dash means that method was not independently measured and inherits the method below it.

Cell	n	roleset	CoT	DGM	TabRSI-DGM	HGM	TabRSI-HGM
da_pot	137	—	0.207	**0.451**	**0.451**	0.337	0.359
da_scot	137	—	**0.275**	**0.275**	**0.275**	**0.275**	**0.275**
da_tcot	137	—	**0.357**	**0.357**	**0.357**	**0.357**	**0.357**
nr_pot	158	—	0.437	**0.772**	**0.772**	0.677	0.677
nr_scot	158	fmt	0.601	0.677	**0.703**	0.639	0.677
nr_tcot	158	—	**0.797**	**0.797**	**0.797**	**0.797**	**0.797**
viz_pot	20	—	**0.700**	**0.700**	**0.700**	**0.700**	**0.700**
viz_scot	20	parse+plan+verify+fmt	0.500	0.500	**0.700**	0.500	0.650
viz_tcot	20	verify+fmt	0.400	0.400	0.500	**0.700**	**0.700**
MACRO (9 cells)	**945**		**0.4749**	**0.5478**	**0.5839**	**0.5536**	**0.5770**

Table 23: LONGTABLEBENCH on Qwen3.8-27B by length regime and turn type, repaired strict, 1,391 held-out rows, all six methods re-scored in one pass. This is the one benchmark where the role team is a net cost on this backbone.

Method	8k/single	8k/multi	32k/single	32k/multi	inf/single	inf/multi	overall
CoT	0.767	0.656	0.707	0.669	0.522	0.481	0.6474
seed (code/PoT)	0.760	0.657	0.721	0.685	0.521	0.466	0.6485
DGM	0.778	0.718	0.715	0.678	0.516	0.531	0.6873
TabRSI-DGM	0.784	0.687	0.705	0.676	0.499	0.528	0.6693
HGM	0.793	0.661	0.698	0.598	0.519	0.446	0.6406
TabRSI-HGM	0.731	0.636	0.713	0.655	0.467	0.479	0.6286

Table 24: MMTU, Qwen3.8-27B subject, DeepSeek-v4-flash meta-agent and judge, macro over 21 scorable subtasks (NL2SQL excluded, no official scorer). ‡ the HGM judge column is partial: only 8 of 14 accuracy subtasks had surviving judged predictions, so the rest silently fall back to strict; not comparable cell-for-cell with DGM's complete column.

Method	strict	LLM judge
vanilla	0.5694	0.4892
HGM	0.6713	0.6608‡
DGM	0.6744	0.6691
TabRSI-HGM	0.6961	0.6817‡
TabRSI-DGM	**0.6984**	**0.6882**

Table 25: The total TabRSI-DGM−vanilla gain of $+0.1289$, split into the three things that could explain it.

component	size	share
scaffold (vanilla → solo)	$+0.1049$	81%
roles (solo → TabRSI)	$+0.0240$	19%
search (DGM vs. HGM)	$+0.0022$	2%

Table 26: Only 6 of 21 subtasks adopted any roleset; of those, only two clear the measured ± 0.048 noise floor (bold).

subtask	roleset	gate Δ	measured Δ
Arithmetic-Relationship	verify+fmt	$+0.374$	$\mathbf{+0.346}$
List-to-table	plan+verify+fmt	$+0.100$	$\mathbf{+0.100}$
header-value-matching	parse+fmt	$+0.014$	$+0.027$
semantic-join	fmt	$+0.291$	$+0.006$
Schema-Matching	fmt	$+0.012$	$+0.000$
String-Relationship	fmt	$+0.000$	$+0.000$