

000 ROBOPHD: SELF-IMPROVING TEXT-TO-SQL 001 002 THROUGH AUTONOMOUS AGENT EVOLUTION 003 004

005 **Anonymous authors**

006 Paper under double-blind review

007 008 ABSTRACT 009

010 We present RoboPhD, a system where AI agents autonomously conduct research
011 to improve Text-to-SQL performance. RoboPhD implements a closed-loop evolution
012 cycle with two coordinated components: a SQL Generation agent composed
013 of a database analysis script and SQL generation instructions, and an Evolution
014 agent that designs new versions based on performance feedback. Central to the
015 framework is an ELO-based selection mechanism enabling survival-of-the-fittest
016 dynamics while handling non-transitivity in performance.
017

018 Starting from a naive 70-line baseline, RoboPhD evolves agents through iterative
019 cross-pollination, discovering effective techniques without any external guidance
020 on the Text-to-SQL domain. Our best agent, evolved to 1500 lines over 18 iterations,
021 autonomously discovered strategies such as size-adaptive database analysis
022 that adjusts depth based on schema complexity and SQL generation patterns for
023 column selection, evidence interpretation, and aggregation.
024

025 Evolution provides the largest gains on cheaper models: while we improve by 2.3
026 points over a strong Claude Opus 4.5 naive baseline, we show an improvement
027 of 8.9 points over the weaker Claude Haiku model. This enables "skip a tier"
028 deployment: evolved Haiku exceeds naive Sonnet accuracy, and evolved Sonnet
029 exceeds naive Opus—both at lower cost.
030

031 The full system achieves 71.3% accuracy on the BIRD development set, demon-
032 strating that AI can autonomously build a strong agentic system without human
033 intervention.
034

035 1 INTRODUCTION

036 Text-to-SQL, the task of translating natural language questions into executable SQL queries, remains
037 a fundamental challenge in natural language processing with significant real-world applications. Re-
038 cent advances have shown that large language models (LLMs) can achieve impressive performance
039 on benchmarks like Spider (Yu et al., 2018) and BIRD (Li et al., 2023), but these approaches typi-
040 cally require extensive manual prompt engineering, careful model selection, and domain expertise.
041

042 The ability of AI systems to conduct AI research autonomously has been identified as a crucial cap-
043 ability milestone with profound implications for the pace of AI progress (Steinhardt, 2022; Cotra,
044 2022). Several researchers have argued that AI systems capable of improving themselves or con-
045 ducting research represent a potential discontinuity in technological development (Bostrom, 2014;
046 Yudkowsky, 2008). Recent work has begun to explore this frontier, with systems demonstrating in-
047 creasing autonomy in scientific discovery (Romera-Paredes et al., 2024), machine learning research
048 (Lu et al., 2024), and algorithm development (Fawzi et al., 2022).
049

050 While well short of a full self-improving system, our work demonstrates a concrete instantiation of
051 AI agents autonomously conducting the research cycle of hypothesis generation, experimentation,
052 and iterative refinement to improve performance on text-to-SQL generation.
053

054 RoboPhD's architecture evolves AI agents composed of two artifacts—a database analysis tool and
055 SQL generation instructions—using an architecture that separates offline analysis from online infer-
056 ence. This pattern generalizes to other tasks with similar structure.
057

058 This work makes the following key contributions:
059

- **Autonomous AI Research Framework:** We demonstrate AI agents conducting systematic research on text-to-SQL through a closed-loop evolution cycle, autonomously discovering effective optimization strategies without human intervention
- **ELO-Based Evolutionary Selection:** First application of ELO ratings for evolutionary prompt/agent optimization, effectively handling non-transitivity and asynchronous agent entry
- **Evolution of a complete Text-to-SQL system:** The framework simultaneously evolves a set of Python database analysis tools and SQL-generation instructions to improve end-to-end performance.
- **Inverse Scaling of Evolutionary Benefit:** We show that evolution provides larger gains on cheaper models (+8.9 points for Haiku vs +2.3 for Opus), enabling cost-effective deployment where evolved cheaper models exceed naive expensive models
- **Self-Directed Improvement without Author-Supplied Domain Expertise:** The system improves without author-provided Text-to-SQL knowledge from a trivial baseline; gains arise from latent LLM capabilities and iterative error-driven learning across iterations

2 RELATED WORK

2.1 TEXT-TO-SQL SYSTEMS

The BIRD benchmark (Li et al., 2023) has driven significant advances in Text-to-SQL, with top systems achieving (as of mid-September 2025) 71-77% accuracy through diverse approaches. AskData (Shkpenyuk et al., 2025) achieves 77.14% using GPT-4o with data analysis agents. CHASE-SQL (Pourreza & Rafiei, 2024) reaches 76.02% through multi-path reasoning and preference optimization. XiYan-SQL (Liu et al., 2024) employs multi-scale few-shot learning for 75.63% accuracy. CSC-SQL (Sheng et al., 2025) uses community-driven schema construction achieving 73.67%. Reasoning-SQL (Pourreza et al., 2025) distills capabilities to smaller models reaching 72.78%. OpenSearch-SQL (Xie et al., 2025), OmniSQL (Li et al., 2025a), and GenaSQL (Dönder et al., 2025) all achieve approximately 72% through various architectures. Distillery (Maamari et al., 2024) introduces compositional approaches at 71.83%. CHESS (Talaei et al., 2024) achieves 71.1% using contextual harnessing.

Our work uniquely focuses on *autonomous discovery* of these strategies rather than manual design. While prior systems require human experts to engineer prompts and architectures, RoboPhD discovers optimization techniques autonomously. Note that we have experimented with using all of the above papers (with the exception of the BIRD benchmark paper (Li et al., 2023)) as inputs to the evolution agent in the *research-driven* evolution strategy mentioned in Section 5.2.

Prior work on prompt optimization includes APE (Zhou et al., 2022), OPRO (Yang et al., 2023), DSPy (Khattab et al., 2023), and TextGrad (Yuksekgonul et al., 2025). While these systems optimize prompts, our approach evolves complete agent artifacts including database analysis tools and SQL generation instructions. Crucially, our system conducts research autonomously without human intervention in the optimization loop. One key aspect of our approach is allowing the Evolutionary agent access to the errors made from previous candidate agents. This is similar to the idea of *text gradients* explored in the TextGrad paper (Yuksekgonul et al., 2025), where the LLM produces a (metaphorical) gradient to describe how to address a particular error. In our system, the error gradients are available for the agent to choose how to incorporate them into the next candidate agent.

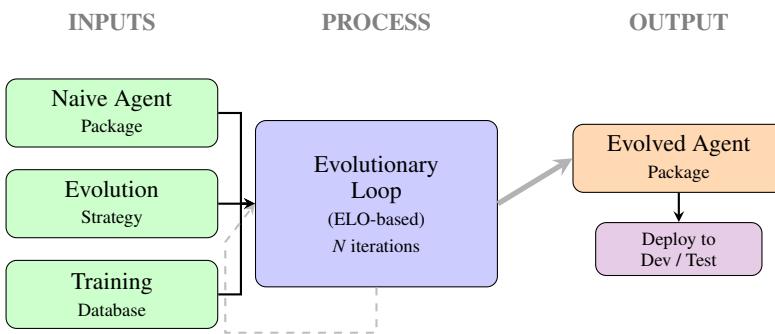
Evolutionary algorithms have been applied to neural architecture search (Pham et al., 2019) and hyperparameter optimization (Loshchilov & Hutter, 2016; Jaderberg et al., 2017). We extend this to evolving complete AI agents with natural language specifications and tool usage, creating a fully autonomous research system.

The ELO rating system (Elo, 1978), originally developed for chess, has found recent applications in AI evaluation. The Chatbot Arena (Zheng et al., 2023) uses ELO ratings to create a dynamic leaderboard of LLMs based on human preferences, demonstrating ELO’s effectiveness for model comparison. While these systems use ELO for passive evaluation and ranking, we introduce ELO as an active selection mechanism for evolutionary optimization (see §3.2.5 for our implementation).

108

3 METHOD

110
 111 Figure 1 shows the high-level architecture of RoboPhD. The system takes three inputs: a simple
 112 *naive agent* package containing minimal Text-to-SQL logic (~70 lines total), an *evolution strategy*
 113 that provides meta-level research guidance, and the BIRD *training database* (69 databases, 6.6K
 114 training questions). Through competitive evolution over N iterations, these simple origins yield a
 115 production-ready agent. Our best agent grew from 50 lines of code and 20 lines of SQL generation
 116 instructions to over 1000 lines of evolved code and 500 lines of generation instructions across 18
 117 evolutionary iterations, achieving 71.3% accuracy on the BIRD development set.



129 Figure 1: RoboPhD system overview. Simple inputs (a naive agent, an evolution strategy, and
 130 training data) feed into an ELO-based evolutionary loop that produces a strong, production-ready
 131 agent for deployment.

132

3.1 CONCEPTUAL OVERVIEW: THE GRADUATE STUDENT METAPHOR

133
 134
 135 RoboPhD operates like a graduate student operating with minimal faculty supervision, focused on
 136 reducing error rate. This paper describes an application to the BIRD benchmark, but the lack of
 137 SQL-specific initial guidance points towards easy transfer to other domains. The evolutionary agent,
 138 here implemented as Claude Code¹, wrapping Claude Sonnet 4.5² or Opus 4.5³, can be interpreted
 139 as a doctoral student attempting to build a complete AI-powered text-to-SQL system, analogous to
 140 prior efforts on the BIRD benchmark.

141 The graduate student (AI Evolutionary Agent) analyzes experimental failures to identify patterns,
 142 forms hypotheses about improvements, and implements these ideas as new system designs. System
 143 designs are tested and refined at least once before being deployed. Each agent it creates consists of a
 144 Python-based database analysis tool and SQL generation instructions for the evaluation model. Note
 145 that, while we have a new “graduate student” with each iteration, the evolutionary agent maintains
 146 context across the entire process of creating an agent package, refining its design and accumulating
 147 expertise with each system test.

148 A core design goal of RoboPhD is *domain independence*: we intentionally avoid injecting author-
 149 crafted Text-to-SQL techniques. The evolution strategy (see Appendix C) provides only meta-level
 150 research direction (“combine the best techniques from top-performing agents”), rather than pre-
 151 scribing scientific content about Text-to-SQL itself. Consequently, measured gains arise from two
 152 sources: (i) the latent capabilities of the underlying LLMs and (ii) experience accumulated over
 153 many iterations as the Evolutionary Agent builds successively stronger agents that it refines and
 154 combines to create even stronger agents.

155 Algorithm 1 presents the complete evolution cycle, showing how strategy selection, agent evolution,
 156 evaluation, and ELO updates work together to drive continuous improvement.

160 ¹<https://claude.com/product/claude-code>

161 ²<https://www.anthropic.com/claude-sonnet-4-5-system-card>

162 ³<https://www.anthropic.com/claude-opus-4-5-system-card>

162 **Algorithm 1** RoboPhD Evolution Cycle

163 1: **Input:** Database pool \mathcal{D} , question pool \mathcal{Q} , iterations N , strategy S , initial agents \mathcal{A}_0
164 2: *// In this paper: $\mathcal{D} = \text{BIRD train (69 databases)}$, $S = \text{cross-pollination}$, $\mathcal{A}_0 = \{\text{naive}\}$*
165 3: **Output:** Final agent rankings and performance history
166 4:
167 5: $\mathcal{A} \leftarrow \mathcal{A}_0$; $\text{competitors} \leftarrow \mathcal{A}_0$
168 6:
169 7: **for** $i = 1$ to N **do**
170 8: *// Evaluation*
171 9: $\text{databases} \leftarrow \text{RandomSample}(\mathcal{D}, 5)$
172 10: **for** each db in databases **do**
173 11: $\text{questions}[db] \leftarrow \text{RandomSample}(\mathcal{Q}[db], 30)$
174 12: **end for**
175 13: **for** each $agent$ in competitors **do**
176 14: $\text{accuracy}[agent] \leftarrow \text{EvaluateOnQuestions}(agent, \text{databases}, \text{questions})$
177 15: **end for**
178 16:
179 17: *// ELO Update via Pairwise Decomposition*
180 18: **for** each pair (a, b) in $\{(1, 2), (1, 3), (2, 3)\}$ **do**
181 19: $\text{score} \leftarrow 1$ if $\text{accuracy}[a] > \text{accuracy}[b]$, 0.5 if equal, else 0
182 20: $\text{UpdateELO}(\text{competitors}[a], \text{competitors}[b], \text{score}, K = 32)$
183 21: **end for**
184 22:
185 23: $\text{winner} \leftarrow \arg \max_{agent \in \text{competitors}} \text{accuracy}[agent]$
186 24:
187 25: *// Prepare next iteration*
188 26: **if** $i < N$ **then**
189 27: $\text{agent}_{\text{new}} \leftarrow \text{EvolveAgent}(\text{competitors}, S, \text{ErrorAnalysis}(i))$
190 28: $\mathcal{A} \leftarrow \mathcal{A} \cup \{\text{agent}_{\text{new}}\}$
191 29:
192 30: *// Tournament Selection for Next Round*
193 31: $\text{competitors}[1] \leftarrow \text{winner}$ *{Current winner}*
194 32: $\text{competitors}[2] \leftarrow \text{agent}_{\text{new}}$ *{Newly evolved}*
195 33: $\text{competitors}[3] \leftarrow \text{RandomTop}(\mathcal{A}, 2)$ *{Random selection from top 2 by ELO}*
196 34: **end if**
197 35: **end for**
198 36: **return** \mathcal{A} with final ELO rankings

3.2 TECHNICAL IMPLEMENTATION

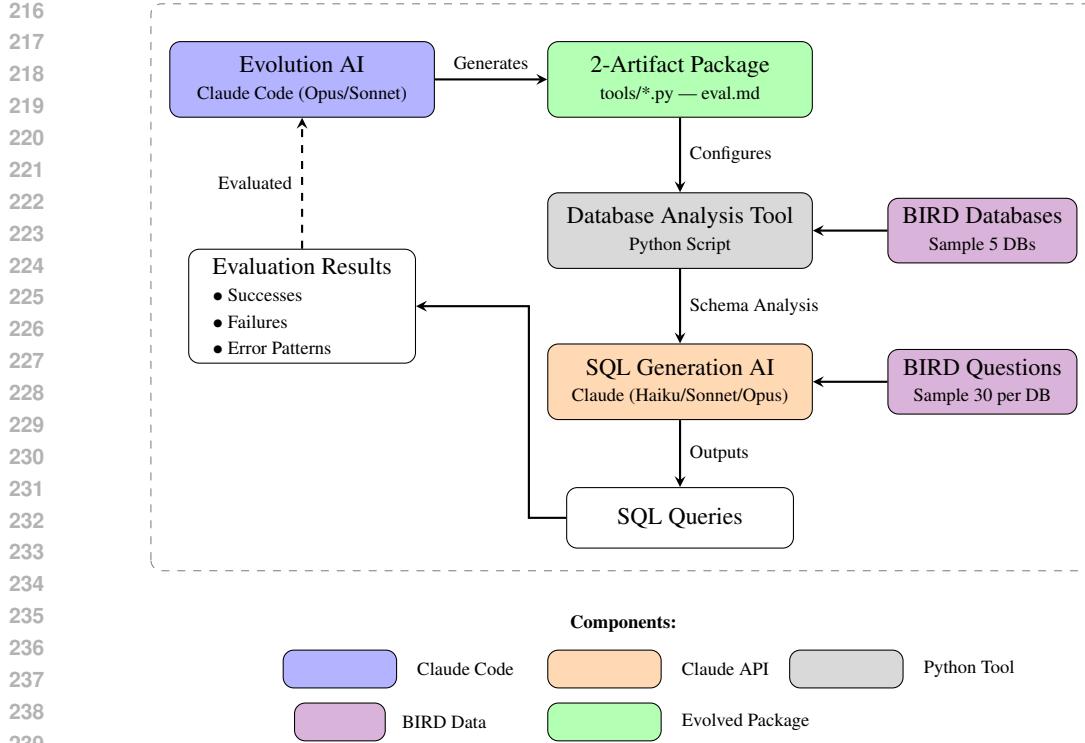
We now detail how RoboPhD achieves accurate SQL generation from simple initial components through an autonomous evolution process.

3.2.1 SYSTEM OVERVIEW: EVOLVING AN AGENT WITH TWO COMPONENTS

RoboPhD’s architecture centers on an *agent* consisting of two components that are evolved together each iteration, as illustrated in Figure 2:

1. **Database Analysis Tool:** A deterministic Python script that analyzes database schemas offline, producing structured documentation of tables, relationships, sample values, and data patterns.
2. **Eval Instructions:** SQL generation guidance that teaches the LLM how to transform natural language questions into precise SQL queries.

These artifacts operate in sequence: the analysis tool examines a database offline, producing structured output that feeds into online SQL generation. The SQL generation model never sees the database schema directly—all understanding must flow through the analysis layer, forcing the evolution process to develop effective communication strategies between its components.



240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269

Figure 2: The RoboPhD evolutionary cycle. The Database Analysis Tool (gray) is a deterministic Python script, not an LLM call, making offline analysis fast (\$0.00) and reproducible.

3.2.2 OFFLINE DATABASE ANALYSIS PHASE

The database analysis phase is conducted without seeing the questions that will be asked. This separation is typical of industrial requirements where extensive database analysis can be performed offline to facilitate SQL generation, which operates under tight time constraints. This pattern generalizes broadly: separating offline analysis from online inference enables thorough preprocessing to support rapid real-time responses.

Tool-Only Execution: Our best-performing approach uses deterministic Python scripts that analyze databases without any LLM calls. The evolved tool receives full access to a SQLite database (schema and row data) and produces structured analysis including (for our better performing agents): complete DDL schema, table relationships with foreign key cardinality, sample values for each column, enumerated values for categorical columns, and data format patterns (dates, currencies, codes).

We experimented with three approaches during system development: (1) LLM-only analysis where an AI agent reasons about the schema, (2) hybrid approaches where an LLM orchestrates Python tools and synthesizes their outputs, and (3) tool-only execution where Python scripts generate complete analysis deterministically. Through competitive evaluation, tool-only execution emerged as superior for two reasons: it is fast and economical (\$0.00 per database vs. \$0.50 for LLM-based analysis), and, perhaps because a deterministic target is easier to optimize in an evolutionary cycle, it yielded our strongest results. The tool-assisted hybrid approach remains as a fallback mechanism if the evolved Python tool fails, but in practice this is relatively rare. For instance, tool failures accounted for only 5.5% of total costs in the evolution of our best-performing system.

3.2.3 ONLINE SQL GENERATION PHASE

The goal of a Text-to-SQL system is to transform natural language questions into executable SQL queries. The SQL generation phase is the end of the process, where the system transforms natural language questions into executable SQL through concatenation of the two evolved artifacts with user inputs:

270

271

Prompt = DatabaseAnalysis \oplus EvalInstructions \oplus Question \oplus Evidence

272

273

where \oplus denotes concatenation. DatabaseAnalysis is the output of the evolved Python tool (Section 3.2.2). EvalInstructions is the evolved SQL generation guidance. The Question is the natural language question as provided by the user (here, BIRD benchmark). Evidence is provided by BIRD and supplies hints about the domain/schema and clarifications for the question.

274

275

276

The Role of Eval Instructions: The `eval_instructions.md` file serves as the bridge between database analysis and SQL generation. While the analysis tool extracts database structure, the eval instructions guide how to transform natural language questions into precise SQL queries. These instructions operate at the final online inference step. Successful eval instructions balance comprehensiveness with precision. Overly lengthy instructions can be counterproductive, as the LLM may have difficulty focusing on the most important components and they may use up too much of Claude’s 200K token limit.

277

278

279

280

281

282

283

Our best-performing agent’s eval instructions (527 lines) include: (1) **Column Selection Discipline:** Detailed rules with examples showing that, for instance, “Which movie has the best rating?” should return only the title, not `SELECT Title, Rating`—a critical requirement where extra columns count as failure. (2) **Evidence Interpretation Patterns:** Seven distinct patterns for parsing BIRD’s evidence field, from column mappings (“full address refers to `street_num`, `street_name`, `city`”) to complex percentage formulas. (3) **String Matching Rules:** Instructions to check the tool’s enum value output for exact case-sensitive matches, with guidance on when to use `LIKE` vs. exact equality. (4) **Percentage Formula Parsing:** Rules distinguishing “percentage of A that are B” from “percentage of B that are A”—the word order determines which count is the numerator vs. denominator.

284

285

286

287

288

289

290

291

292

Agentic Query Result Validation: text-to-SQL systems face two critical failure modes: syntax errors and semantic errors. Syntax errors prevent execution, while semantic errors produce incorrect results. Prior work typically addresses these challenges through multiple generation attempts. For instance, OpenSearch-SQL (Xie et al., 2025) generates multiple candidate queries and uses self-consistency voting. Our system employs an agentic answer evaluation approach:

293

294

295

296

297

Universal Verification: Every generated SQL query undergoes verification where the model reviews its own query execution results in the context of the question and the system prompt with database analysis information. The model then responds with either `CORRECT` to accept those results as the answer or an improved SQL query in light of this new information from the candidate execution result. This self-verification and correction process can iterate up to $k = 2$ times, using progressive temperature ($0.0 \rightarrow 0.2 \rightarrow 0.3$) to encourage exploration on retries. Additionally, if iteration k produces errors or an empty (null) result, the system employs an additional targeted retry with an alert highlighting the SQL error or null result.

298

299

300

301

302

303

304

305

3.2.4 EVOLUTION: CREATING NEW ARTIFACTS

306

307

The Evolution AI creates new artifacts based on comprehensive performance feedback from previous iterations (see Algorithm 1). It receives ELO rankings, accuracy metrics, and detailed error analysis, then generates two artifacts that define a candidate solution: 1) a database analysis tool (`tools/*.py`); and 2) SQL generation instructions (`eval_instructions.md`). The evolution process is guided by a hand-crafted strategy prompt; evolving these strategies themselves is future work (Section 5.2).

308

309

310

311

312

313

Cross-Pollination Strategy: Our primary evolution strategy instructs the Evolution AI to examine top-performing agents and combine their best techniques into a new “super-tool.” The strategy prompt (see Appendix C) emphasizes:

314

315

316

317

- **Identifying complementary strengths:** Which techniques do different agents use effectively? How do they complement each other?
- **Error-driven improvement:** What error patterns appear in the iteration’s `error_analysis_report.md`? How can the new agent address them?
- **Tool-only requirement:** The strategy requires deterministic Python tools rather than LLM-based analysis, ensuring speed and reproducibility.

324 The Evolution AI produces a `reasoning.md` file documenting its analysis before generating artifacts, providing transparency into its design decisions. We experimented with alternative strategies (refinement, research-driven, error-focused) but found cross-pollination with tool-only emphasis consistently produced our strongest agents.

325 **Deep Focus Refinement:** Before a newly-evolved agent is put into the main competitive loop, it
 326 undergoes a multi-round testing process we call “Deep Focus.” After generating an artifact using
 327 the methodology described above, the system tests the newly created agent on the questions and
 328 databases from prior iterations, beginning with the most recent iteration and working backwards,
 329 comparing it with the corresponding agents from those iterations (for instance, it is shown questions
 330 the new agent uniquely answered correctly and uniquely answered incorrectly). After each testing
 331 round, the Evolution AI can refine its artifacts based on this performance feedback. We parame-
 332 terize this with k test rounds (default $k = 1$), where each round tests against one additional prior
 333 iteration. Our experiments with $k \in \{0, 1, 2, 3\}$ found that even a single test round ($k = 1$) provides
 334 meaningful refinement opportunities while keeping evolution time reasonable. Key to Deep Focus
 335 is that all Deep Focus testing and refinement takes place in the same Claude Code session in which
 336 the agent was created, enabling the Evolution AI to maintain continuity of planning and analysis.

340 3.2.5 EVALUATION AND ELO RANKING

341 The system evaluates three agents simultaneously on identical sets of databases and questions, com-
 342 puting overall accuracy using BIRD’s set-based comparison (row order ignored, exact match re-
 343 quired). Each iteration tests agents on randomly selected databases with their associated questions,
 344 providing head-to-head comparisons on the exact same tasks.

345 These three-way competitions are decomposed into three pairwise comparisons. For example, if
 346 agents achieve 65%, 62%, and 62% accuracy, we process three head-to-head results: A beats B
 347 (1-0), A beats C (1-0), and B ties C (0.5-0.5).

348 Maintaining an accurate picture of relative agent strength is critical for the evolutionary algorithm.
 349 This work introduces ELO as a ranking mechanism for evolutionary algorithms, updating scores
 350 using the standard formula ($\Delta_{ELO} = K(S - E)$, $K = 32$). ELO offers key advantages for our
 351 setting:

352 (1) **Asynchronous Competition:** Like chess players who play their first tournament match at dif-
 353 ferent times, agents can join the population at any point while maintaining fair comparisons through
 354 persistent ratings. ELO naturally weights unexpected outcomes more strongly than expected out-
 355 comes—when a low-rated (or newly-created) agent defeats a high-rated one, both scores change
 356 much more than if the higher-rated agent defeats the lower-rated.

357 (2) **Non-transitivity Handling:** ELO naturally accommodates rock-paper-scissors dynamics where
 358 on one sample of databases and questions, Agent A beats B, on another sample B beats C, but on a
 359 third sample C beats A—a situation which is common when different agents have different strengths
 360 and weaknesses.

361 (3) **Task Normalization:** Win/loss treatment normalizes across varying difficulty—although accu-
 362 racy commonly swings from 60% to 80% on different database samples, relative rankings on the
 363 same task remain stable.

364 This competitive framework enables survival-of-the-fittest dynamics while maintaining fairness
 365 across agents with different entry points.

366 3.2.6 EXPERIMENTAL PROTOCOL: SELECTION AND SAMPLING

367 **Agent Selection:** In a standard evolutionary iteration, (see Appendix D for details on exceptions),
 368 RoboPhD tests three agents selected via a priority system: (1) the previous winner, (2) the newly
 369 evolved agent, and (3) a random agent selected from the top 2 performers by ELO (other than the
 370 winner). This balance ensures thorough testing while maintaining the evolutionary pressure.

371 **Database Sampling:** To prevent overfitting and encourage generalizable improvements, each iteration
 372 evaluates agents on only 5 randomly selected databases from the 69-database training set, with
 373 30 randomly sampled questions per database (BIRD supplies a median of 82 questions per database
 374 for a total of 6.6K training questions). This deliberate undersampling ensures that agents never see

378 the complete dataset in any single iteration, forcing the evolution process to discover robust pat-
 379 terns rather than memorizing specific database quirks. The random sampling changes each iteration,
 380 exposing agents to different challenges and preventing specialization on a fixed subset.
 381

382 4 EXPERIMENTS

384 4.1 EXPERIMENTAL SETUP

386 We evaluate on the BIRD benchmark (Li et al., 2023), a large-scale cross-domain dataset with 69
 387 training databases and 11 development databases. Each evolution iteration (using only Haiku 4.5
 388 LLMs for SQL generation) costs approximately \$2 in Claude API for the calls and runs for 22
 389 minutes on average, enabling rapid experimentation cycles. Note that evolution is accomplished via
 390 Claude Code (using either Sonnet or Opus 4.5 LLMs) and does not incur direct costs, but rather
 391 uses up a weekly usage quota. Empirically, each 30-iteration run uses about 12% of the weekly
 392 usage quota of a Claude Max plan.⁴ Computational requirements are modest. Experiments were
 393 conducted on either a MacBook Pro (48GB RAM) or an Azure VM with 8 vCPUs and 32GB RAM.
 394

395 4.2 MAIN RESULTS

397 We demonstrate RoboPhD’s effectiveness by evolving agents on the training set and evaluating them
 398 across three tiers of Anthropic models: Haiku-4.5, Sonnet-4.5, and Opus-4.5. As noted above, the
 399 naive baseline represents the simplest possible agentic design, similar to what someone would use
 400 if they were new to the Text-to-SQL problem, with a 50-line Python tool that only dumps raw DDL
 401 schema statements (see Appendix E for the complete agent). The naive agent represents the starting
 402 point of our evolutionary process as well as our experimental baseline.
 403

404 Table 1: Results on BIRD dev set. Evolution provides larger improvements on cheaper models.

	Opus-4.5		Sonnet-4.5		Haiku-4.5	
	Accuracy	Cost/Query	Accuracy	Cost/Query	Accuracy	Cost/Query
Naive	69.0%	1.61¢	65.7%	0.56¢	57.2%	0.34¢
Best Evolved	71.3%	3.13¢	69.2%	0.87¢	66.1%	0.51¢
Δ	+2.3		+3.5		+8.9	

412 There are three key takeaways from our main results, shown in Table 1:
 413

414 (1) Evolution produced the largest gains on the cheapest models. While the evolved Opus agent
 415 improves by 2.3 points over naive Opus, the evolved Haiku agent gains 8.9 points—nearly four
 416 times the improvement. This inverse relationship between model capability and evolutionary benefit
 417 suggests that more powerful models already capture much of what can be learned through prompting
 418 and tooling, leaving less room for improvement.

419 (2) Relative to the naive baseline, evolution delivers higher accuracy at lower cost. An evolved Haiku
 420 agent (66.1%, 0.51¢/query) exceeds the accuracy of a naive Sonnet agent (65.7%, 0.56¢/query) at
 421 lower cost. Similarly, an evolved Sonnet agent (69.2%, 0.87¢/query) exceeds naive Opus (69.0%,
 422 1.61¢/query) at roughly half the cost. Since evolution is a one-time training cost while inference
 423 is ongoing, organizations can deploy cheaper models with evolved prompts rather than expensive
 424 models with naive prompts.

425 (3) Evolution delivers meaningful improvements across all model tiers, arguing for the broad ap-
 426 plicability of RoboPhD’s approach. Even Opus-4.5, which starts from a strong 69.0% baseline,
 427 benefits from evolution, reaching 71.3%—our highest recorded accuracy on the BIRD dev set. As
 428 of the time of this writing, 71.3% ranks 17th on the BIRD dev set.⁵ We would argue that this is a
 429 strong result to achieve without the benefit of human domain knowledge.

430 ⁴<https://www.claude.com/pricing/max>

431 ⁵<https://bird-bench.github.io/>

432 4.3 EVOLUTION ANALYSIS
433

434 **Champion Agent Design.** Our best-performing agent, `iter18_hybrid_comprehensive_analyzer`,
 435 emerged from *cross-pollination*—in the 18th round of evolution, the Evolution AI synthesized tech-
 436 niques from three parent agents that had each reached 76% training accuracy on the 17th iteration
 437 through different approaches: adaptive context scaling, semantic pattern analysis, and precision
 438 column coverage. Rather than choosing one approach, evolution combined their complementary
 439 strengths into a unified tool. The key design innovation was a size-based feature matrix that adjusts
 440 analysis depth based on database complexity:

443 Feature	444 Small (≤150 cols)	444 Medium (≤300 cols)	444 Large (≤400 cols)	444 Ultra (>400 cols)
445 Sample values/column	446 10	446 5	446 3	446 1
446 Enum value limit	447 All	447 15	447 5	447 0
447 Semantic patterns	448 Full	448 Essential	448 Skip	448 Skip
448 Cross-table validation	449 Full	449 Critical	449 Skip	449 Skip

449 Table 2: Size-adaptive feature matrix. The agent gracefully degrades analysis depth for larger
 450 databases to prevent overflowing the 200K token Claude context limit while maintaining compre-
 451 hensive analysis for typical BIRD databases.

452
 453 This adaptive approach prevents catastrophic failures on large databases (which previously caused
 454 0% accuracy due to context overflow) while maintaining rich analysis for small and medium
 455 databases where detailed semantic understanding improves accuracy.

456 **Database Analysis Script.** The ~1000-line Python script generates a structured 10-section anal-
 457 ysis: (1) complete schema DDL, (2) table overview with row counts, (3) detailed column analysis
 458 with data types and sample values, (4) foreign key relationship map with cardinality, (5) enum-
 459 erated value reference for low-cardinality columns, (6) value ranges for numeric columns, (7) format
 460 detection (dates, currencies, codes), (8) semantic patterns like temporal sequences or hierarchical
 461 relationships, (9) cross-table validation identifying orphaned foreign keys, and (10) query guidance
 462 with common pitfalls for the specific database.

463 **Evolved SQL Instructions.** The eval instructions (527 lines) encode patterns discovered across
 464 18 iterations of evolution, including: precise column selection rules (return only columns explicitly
 465 requested, not those used for filtering or sorting), evidence interpretation patterns (when evidence
 466 says “full address refers to street_num, street_name, city,” return all three columns in that exact
 467 order), and aggregation guidance (apply LIMIT 1 only for superlatives like “best” or “highest,” not
 468 for singular nouns).

470
471 5 DISCUSSION AND FUTURE WORK

472 5.1 PRACTICAL APPLICABILITY

473
 474 RoboPhD produces easily deployable artifacts. The evolved agent consists entirely of a Python
 475 script and a markdown file containing SQL generation instructions—no complex frameworks or
 476 dependencies. At deployment time, the Python script runs once per database as an offline prepro-
 477 cessing step, generating a structured analysis without seeing any user questions. This analysis is
 478 concatenated with the SQL generation instructions to form a system prompt.

479 At runtime, answering a user’s question requires only this system prompt, the question, and any
 480 provided evidence—there is no dependency on Claude Code, the evolutionary infrastructure, or
 481 any of the complexity described in this paper. The only additional logic beyond the prompt is the
 482 universal verification step (Section 3.2.3), which validates query results and optionally retries. This
 483 simplicity enables straightforward integration into production systems.

486
487

5.2 ARCHITECTURAL OPTIONS UNDER INVESTIGATION

488
489
490

RoboPhD’s framework supports several capabilities that we have explored but not included in this paper due to insufficient experimental evidence. In the interest of simplicity, we focus on the cross-pollination strategy with tool-only execution, but note these alternatives for future investigation:

491
492
493
494
495

Meta-evolution. A meta-agent critiques and evolves the evolution strategies themselves, meta-evolving new strategies and scheduling them strategically based on performance trends and review of the reasoning and post-evolution reflection of the evolutionary agents. Our preliminary experiments show promise, but results remain ambiguous—meta-evolution may primarily benefit very lengthy runs exceeding 30 iterations where strategy adaptation becomes more valuable.

496
497
498
499
500

Alternative evolution strategies. We developed several manually-written strategies beyond cross-pollination, including a *research-driven* strategy that instructs the Evolution AI to study academic papers and extract applicable techniques. While the research-driven strategy is intriguing, we view these alternative strategies as likely superseded by meta-evolution, which could discover the most effective strategies automatically.

501
502
503
504
505

LLM-based database analysis. Our current approach is “all-or-nothing”—either a deterministic Python script or full LLM reasoning. A promising middle ground would allow generated Python scripts to invoke the LLM in targeted situations where semantic understanding provides high value, such as inferring implicit relationships or disambiguating column purposes.

506
507

5.3 BROADER APPLICATIONS

508
509
510
511
512

Other Text-to-SQL benchmarks. While we evaluate on BIRD, the RoboPhD framework is benchmark-agnostic. Spider 2.0 (Lei et al., 2024) presents a natural extension, with its enterprise-scale databases and more complex queries requiring multi-step reasoning. BIRD-Critic (Li et al., 2025b) offers another direction, focusing on debugging and fixing incorrect SQL queries rather than generation from scratch.

513
514
515
516
517

Other domains. Given that a key feature of the system is an absence of human-injected text-to-SQL domain knowledge, we would expect RoboPhD to generalize beyond Text-to-SQL. Code generation represents a particularly promising application, where similar evolutionary pressure could discover effective prompting strategies and tool designs for programming tasks.

518
519
520

6 CONCLUSION

521
522
523
524
525

We presented RoboPhD, a system where AI agents autonomously conduct research to improve Text-to-SQL performance. Our system achieves 71.3% accuracy on the BIRD Dev set through systematic evolution, discovering effective strategies without human intervention in the research loop. Like a tireless PhD student, RoboPhD runs experiments continuously, learning from failures and evolving better approaches.

526
527
528
529
530
531
532

While our system operates within the bounded domain of database queries, it demonstrates that AI agents can conduct meaningful research cycles—analyzing failures, forming hypotheses, and iteratively refining solutions—without human intervention. The techniques explored here, including competitive evolution and error-driven learning, may extend to other domains where systematic experimentation is feasible. We open-source our framework to enable the community to explore whether similar approaches can accelerate capability development in domains beyond Text-to-SQL.

533
534

ETHICS STATEMENT

535
536
537

This work involves autonomous AI systems with significant operational freedom, raising important safety and security considerations. We acknowledge several critical risks:

538
539

Execution Safety: RoboPhD operates with Claude Code in an automated mode where generated code is executed without human review. While model alignment has reduced risks significantly, autonomous code execution remains inherently dangerous—the system could theoretically execute

540 destructive commands. We mitigate this through process isolation and filesystem permissions, but
 541 acknowledge this remains an open challenge for autonomous AI systems.
 542

543 **Security Vulnerabilities:** Deploying AI-evolved code introduces novel attack surfaces. The evolved
 544 Python tools process database schemas and could potentially be manipulated by adversarial inputs
 545 to exfiltrate data or execute unintended operations. While these considerations are not a concern for
 546 BIRD’s open-source databases, production deployments would require strict sandboxing, network
 547 isolation, and comprehensive security auditing of the evolved artifacts.
 548

549 **Code Release:** We release our code openly to enable reproducibility and community scrutiny. While
 550 the techniques demonstrated here are domain-specific, we encourage practitioners deploying similar
 551 autonomous code generation systems to implement appropriate safeguards including sandboxing,
 552 least-privilege execution, and security review of generated artifacts.
 553

554 REFERENCES

555 Nick Bostrom. *Superintelligence: Paths, Dangers, Strategies*. Oxford University Press, 2014.

556 Ajeya Cotra. Without specific countermeasures, the easiest path to transformative ai likely leads to
 557 ai takeover. *AI Alignment Forum*, 2022.

558 Ahmet Dönder et al. Genasql: A generative agent for advanced sql generation. *arXiv preprint*
 559 *arXiv:2505.14174*, 2025.

560 Arpad E Elo. *The Rating of Chessplayers, Past and Present*. Arco Publishing, 1978.

561 Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Moham-
 562 madamin Barekatain, et al. Discovering faster matrix multiplication algorithms with reinforce-
 563 ment learning. *Nature*, 610(7930):47–53, 2022.

564 Max Jaderberg, Victor Dalibard, Oriol Osindero, et al. Population based training of neural networks.
 565 *arXiv preprint arXiv:1711.09846*, 2017.

566 Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri
 567 Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, et al. Dspy:
 568 Compiling declarative language model calls into self-improving pipelines. *arXiv preprint*
 569 *arXiv:2310.03714*, 2023.

570 Fangyu Lei, Jixuan Chen, Yuxiao Ye, Ruisheng Cao, Dongchan Shin, Hongjin Su, Zhaoqing Suo,
 571 Hongcheng Gao, Wenjing Hu, Pengcheng Yin, et al. Spider 2.0: Evaluating language models on
 572 real-world enterprise text-to-sql workflows. *arXiv preprint arXiv:2411.07763*, 2024.

573 Hao Li et al. Omnisql: Database systems enhancement with large language models. *arXiv preprint*
 574 *arXiv:2503.02240*, 2025a.

575 Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhu Li, Bowen Li, Bailin Wang, Bowen Qin,
 576 Ruiying Geng, Nan Huo, Xuanhe Zhou, Chenhao Ma, Guoliang Li, Kevin C.C. Chan, Leyang
 577 Fan, Yongji Du, Shaodong Qu, Hongkun Yao, Ming An, Annan Cheng, Gang Zhao, Ziwei Zou,
 578 and Fei Yang. BIRD: Big bench for large-scale database grounded text-to-SQLs. In *Thirty-
 579 seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*,
 580 2023. URL <https://openreview.net/forum?id=qJVWOgJT5H>.

581 Jinyang Li, Xiaolong Li, Ge Qu, Per Jacobsson, Bowen Qin, Binyuan Hui, Shuzheng Si, Nan Huo,
 582 Xiaohan Xu, Yue Zhang, et al. Swe-sql: Illuminating ILM pathways to solve user sql issues in
 583 real-world applications. *arXiv preprint arXiv:2506.18951*, 2025b.

584 Yichen Liu et al. Xiyuan-sql: A multi-scale few-shot learning method for text-to-sql. *arXiv preprint*
 585 *arXiv:2507.04701*, 2024.

586 Ilya Loshchilov and Frank Hutter. Cma-es for hyperparameter optimization of deep neural networks.
 587 *arXiv preprint arXiv:1604.07269*, 2016.

594 Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist:
 595 Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*,
 596 2024.

597 Daniel Maamari et al. Distillery: A compositional framework for text-to-sql. *arXiv preprint*
 598 *arXiv:2408.07702*, 2024.

600 Hieu Pham, Melody Y Guan, Barret Zoph, Quoc V Le, and Jeff Dean. Efficient neural architecture
 601 search via parameter sharing. *ICML*, 2019.

602 Mohammadreza Pourreza and Davood Rafiei. Chase-sql: Multi-path reasoning and preference opti-
 603 mized candidate selection in text-to-sql. *arXiv preprint arXiv:2410.01943*, 2024.

604 Mohammadreza Pourreza et al. Reasoning-sql: Distilling reasoning capabilities from llms to smaller
 605 models for text-to-sql. *arXiv preprint arXiv:2503.23157*, 2025.

606 Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog,
 607 M Pawan Kumar, Emilien Dupont, et al. Mathematical discoveries from program search with
 608 large language models. *Nature*, 625(7995):468–475, 2024.

609 Yunnan Sheng et al. Csc-sql: Community-driven schema construction for text-to-sql. *arXiv preprint*
 610 *arXiv:2505.13271*, 2025.

611 Vladislav Shkapenyuk et al. Askdata: Naturally leveraging data analysis agents for complex text-
 612 to-sql generation. *arXiv preprint arXiv:2505.19988*, 2025.

613 Jacob Steinhardt. Ai capabilities and alignment research. *AI Alignment Forum*, 2022.

614 Shayan Talaei, Mohammadreza Pourreza, and Davood Rafiei. Chess: Contextual harnessing for
 615 efficient sql synthesis. *arXiv preprint arXiv:2405.16755*, 2024.

616 Jianmin Xie et al. Opensearch-sql: Context-aware sql generation with an open-source suite. *arXiv*
 617 *preprint arXiv:2502.14913*, 2025.

618 Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun
 619 Chen. Large language models as optimizers. *arXiv preprint arXiv:2309.03409*, 2023.

620 Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li,
 621 Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex
 622 and cross-domain semantic parsing and text-to-sql task. *EMNLP*, 2018.

623 Eliezer Yudkowsky. Artificial intelligence as a positive and negative factor in global risk. In *Global*
 624 *catastrophic risks*, pp. 308–345, 2008.

625 Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Pan Lu, Zhi Huang, Carlos Guestrin,
 626 and James Zou. Optimizing generative ai by backpropagating language model feedback. *Nature*,
 627 639(8055):609–616, 2025.

628 Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang,
 629 Zi Lin, Zhuohan Li, Dacheng Li, Eric P Xing, Hao Zhang, Joseph E Gonzalez, and Ion Stoica.
 630 Judging llm-as-a-judge with mt-bench and chatbot arena. *NeurIPS*, 2023.

631 Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan,
 632 and Jimmy Ba. Large language models are human-level prompt engineers. *arXiv preprint*
 633 *arXiv:2211.01910*, 2022.

634

635 A LLM USAGE

636

637 **Scope and philosophy.** In keeping with the spirit of this project, we made *extensive* use of large
 638 language models (LLMs) throughout. Our goal was to study whether LLMs can function as au-
 639 tonomous research agents with minimal domain-specific guidance from the authors. In addition to
 640 using Claude Code to help write, enhance, and refactor the code, we used Claude to help write design
 641 documents to brainstorm and clarify new ideas, and Claude helped author parts of the manuscript.
 642 In each case there was human oversight and revision applied, especially for the manuscript itself.

648
649

A.1 ROLES WITHIN THE ROBOPHD SYSTEM

650
651
652
653

Evolution (Section 3.2.6). As discussed in Section 3.1, a core contribution is showing that an LLM can serve in the role of a graduate student. Concretely, we used *Claude Code* running *Opus 4.5* or *Sonnet 4.5* for the Evolution Agent. This agent generated new agent packages (prompts and python tools) across iterations.

654
655
656

SQL Generation (Section 3.2.3). The SQL Generation phase used *Haiku*, *Sonnet*, or *Opus 4.5* via the Claude API (not via *Claude Code*) to translate questions (and evidence) into executable SQL, conditioned on the database analysis and evaluation instructions.

657
658
659
660
661

Minimal domain guidance (Sections 3.1, 3.2). By design, author-provided Text-to-SQL know-how was kept to an absolute minimum. Evolution strategies specify *process* (e.g. cross-pollinate leading agents, analyze errors, etc.) rather than *scientific content*; improvements therefore arise from the models’ latent knowledge and error-driven iteration.

662
663

A.2 LLMs USED AS SOFTWARE-ENGINEERING TOOLS

664
665
666
667
668
669
670
671
672
673
674

In addition, and consistent with the project’s philosophy, we used *Claude Code* (primarily with *Opus* and *Sonnet*) models as a development assistant for the RoboPhD infrastructure. The RoboPhD codebase was almost entirely authored through *Claude Code* sessions, with the human authors providing specifications, reviewing diffs, enforcing style/tests, and making final design decisions. In this context, *Claude Code* functioned as a coding tool under close human supervision. Very little of the final codebase was the result of direct manual edits by the authors, with one exception: the evolution strategy prompts discussed in Section 3.2.4 and shown in Appendix C were primarily hand-written. We see this as consistent with the Graduate Student metaphor laid out in Section 3.1: we, the authors, provide thoughtful strategic guidance (similar to the role of a research advisor), while leaving autonomy to the graduate student to iterate towards a solution. As discussed in Section 5, evolving this strategic guidance is a direction for future research.

675
676

A.3 LLMs USED IN MANUSCRIPT PREPARATION

677
678
679
680
681

In contrast to the codebase, most of the manuscript text was written by the authors. However, portions of this paper were drafted and edited by the authors with assistance from *Claude Code* (*Opus 4.1* and *4.5*), *ChatGPT 5*, and *Gemini 3*. In these cases, the authors determined the structure and arguments. Where LLM-suggested text was used, it was reviewed and, as needed, rewritten by the authors for accuracy and clarity.

682
683

A.4 ATTRIBUTION, AUTHORSHIP, AND ACCOUNTABILITY

684
685
686
687
688

Although LLMs played a significant role in ideation (via the Evolution Agent), engineering assistance, and writing support, all scientific claims, experiments, analyses, and conclusions are the responsibility of the human authors. The models did not have independent access to private test data or to submission systems, and they did not make publication decisions.

689
690

A.5 DATA GOVERNANCE AND SAFEGUARDS

691
692
693
694

LLM prompts and outputs were constrained to benchmark-permitted artifacts (schemas, evidence, and allowed literature). We avoided leaking ground-truth SQL or test questions to the agents; evaluation used standard execution-accuracy protocols. Logs of prompts/outputs for key experiments are retained for audit and reproducibility.

695
696
697

B REPRODUCIBILITY STATEMENT

698
699
700
701

The results in this paper are reproducible using the open source code bundle provided as supplementary material to this manuscript. The entire framework can be run from scratch following instructions in the README, which will execute all of the iterations of evolution, reproducing agents similar to the ones featured in the results and discussion earlier in the paper. In the main body of the paper, we note the model families and modes (*Claude Code Opus/Sonnet 4.5* for evolution and *Claude*

702 *Sonnet 4.5 for evaluation), interface (Claude Code vs. Claude API), and where nondeterminism is*
 703 *relevant, we note seeds and sampling parameters in the released configs. Despite seeds in our code*
 704 *for the randomness that we control, there is some non-determinism that we cannot control in Claude*
 705 *Opus/Sonnet API responses that may lead to minor differences in the results when run from scratch.*
 706 *Given that this is an evolutionary approach over many iterations, small variations in one iteration,*
 707 *may lead to amplified differences in subsequent iterations.*

709 C CROSS-POLLINATION STRATEGY PROMPT

711 This appendix provides excerpts from the cross-pollination evolution strategy that guides the Evolution
 712 Agent. The full prompt (~240 lines) emphasizes process-oriented guidance rather than domain-
 713 specific Text-to-SQL techniques.

715 C.1 CORE OBJECTIVE

717 “*You are creating a new Claude Code agent primarily by combining successful*
 718 *elements from multiple existing agents using the three-artifact architecture.*”

719 “***Note: Although you are primarily using a cross-pollinating approach, you can use a new idea of your own if you think you see an opportunity.*”

721 C.2 CROSS-POLLINATION APPROACH

723 The strategy instructs the Evolution Agent to identify complementary strengths:

725 “*When examining top-performing agents, ask: Which techniques do different agents use effectively? Which approaches consistently achieve the best results? How do different agents’ tools complement each other? What patterns of effectiveness emerge across multiple agents? Which combinations of techniques might create synergies?*”

730 “*Cross-Pollination Approach: 1) Identify Agent A’s most effective techniques. 2) Identify Agent B’s complementary strengths. 3) Identify Agent C’s unique successful approaches. 4) Your tool: Combine these complementary techniques into one comprehensive analyzer.*”

734 C.3 TOOL-ONLY REQUIREMENT

736 The strategy mandates deterministic execution:

738 “*The system supports a **tool-only execution mode** where your Python/shell tool generates a complete analysis file that is directly copied to the agent output, bypassing the AI agent entirely. This is the REQUIRED approach for this cross-pollination strategy.*”

743 C.4 META-LEVEL GUIDANCE

745 The strategy concludes with process-oriented direction:

747 “*Your overall goal: Push accuracy higher through tool-only cross-pollination. You are an expert in the field of Text2SQL. Use your knowledge of the field, your analysis of what is bringing accuracy down with current agents, and your analysis of which tool-based techniques from each agent work best to build an agent package that will achieve higher accuracy than previous agents on a set of databases that you haven’t seen before.*”

752 “*Remember: **Think harder**⁶ than you normally would about this. Review the tools from multiple top-performing agents, identify what makes each one effective, and*

755 ⁶“Think harder” is a Claude Code keyword that triggers extended reasoning. See <https://www.anthropic.com/engineering/clause-code-best-practices>.

756 **combine the best elements into a comprehensive tool-only solution that achieves**
 757 **your goal of pushing accuracy higher.”**

759 These excerpts demonstrate how we provide high-level research methodology—analyze multiple
 760 agents, identify complementary strengths, combine deterministic techniques—without injecting
 761 domain-specific Text-to-SQL knowledge.

763 D AGENT SELECTION PROTOCOL

765 This appendix details exceptions to the standard three-agent selection protocol described in Sec-
 766 tion 3.2.5.

768 D.1 STANDARD SELECTION (ITERATIONS 1–11)

770 Each iteration tests three agents:

- 772 1. **Pending winner(s)**: The winner from the previous iteration. If the previous iteration re-
 773 sulted in a tie, all tied agents are pending winners and receive priority slots.
- 774 2. **Newly evolved agent**: The agent created by the Evolution AI for this iteration.
- 775 3. **ELO-based selection**: Random selection from the top performers by ELO (excluding
 776 pending winners and the new agent).

778 D.2 LATE-STAGE EXPLORATION (ITERATIONS 12–30)

780 In the experiments described in this paper, starting at iteration 12, the system probabilistically sub-
 781 stitutes non-evolution iterations to increase testing coverage of the accumulated agent population:

- 782 • **70% probability**: Standard cross-pollination evolution with 3 agents
- 783 • **15% probability**: “Challenger” iteration—no evolution occurs; instead, 4 agents are
 784 tested, targeting under-tested agents with above average ELO (ELO > 1500, prioritizing
 785 non-pending winners with the fewest prior tests)
- 786 • **15% probability**: “None” iteration—no evolution occurs; 4 agents are tested via standard
 787 ELO-based random selection

789 The increased agent count (4 vs. 3) in non-evolution iterations compensates for the lack of a newly
 790 evolved agent, allowing more thorough evaluation of existing agents. This late-stage exploration
 791 helps identify “hidden gems”—agents with strong potential that received insufficient testing in ear-
 792 lier iterations.

794 E THE NAIVE BASELINE AGENT

797 This appendix presents the complete naive baseline agent referenced in Section 4.2. This agent
 798 represents the simplest possible starting point—prompts that someone new to the Text-to-SQL prob-
 799 lem might write without any domain expertise.

800 E.1 DATABASE ANALYSIS AGENT (AGENT.MD)

802 The database analysis component uses tool-only execution to extract raw schema:

```
804        ---  

  805        name: naive  

  806        description: Baseline agent that outputs raw DDL schema using tool-only execution  

  807        execution_mode: tool_only  

  808        tool_command: python tools/extract_schema.py  

  809        tool_output_file: tool_output/schema.txt  

  810        ---
```

```

810
811 # Naive DDL Extractor (Tool-Only)
812
813 This agent uses deterministic tool-only execution to extract raw database schema.
814
815 ## Process
816
817 1. **Run Schema Extraction Tool**
818     ````bash
819     python tools/extract_schema.py
820     `````
821
822 2. **Read and Output Results**
823     - Read the generated schema from 'tool_output/schema.txt'
824     - Write the complete output to './output/agent_output.txt'
825
826 ## Error Recovery
827
828 If the tool fails:
829
830 1. Check 'database.sqlite' exists
831 2. Verify Python environment has sqlite3 library
832 3. Examine any error messages in 'tool_output/'
833 4. Attempt to run the tool manually to see errors
834 5. Fall back to manual schema extraction using SQLite CLI:
835     ````bash
836     sqlite3 database.sqlite ".schema" > output/agent_output.txt
837     `````
838
839
840 E.2 SQL GENERATION INSTRUCTIONS (EVAL_INSTRUCTIONS.MD)
841
842 The SQL generation instructions are equally minimal:
843
844 # SQL Generation Instructions
845
846 ## Core SQL Requirements
847
848 Generate clean, executable SQL:
849 - No markdown formatting or code blocks
850 - No comments or explanatory text
851 - Only the SQL statement
852
853 ## Evidence Handling
854
855 Evidence is important! When evidence is provided, be sure to follow
856 it very carefully
857
858 ## SQLite Specifics
859
860 - This is a SQLite database. Be sure to use SQLite syntax
861
862 ## Remember
863
864 Keep it simple. Return exactly what's requested. Follow evidence literally.

```

E.3 SCHEMA EXTRACTION TOOL (TOOLS/EXTRACT_SCHEMA.PY)

The naive baseline includes a single 50-line Python script that extracts raw DDL:

```

864 #!/usr/bin/env python3
865 """Baseline schema extractor - extracts raw DDL using sqlite3."""
866
867 import sqlite3
868 import sys
869
870 def extract_schema(db_path: str, output_file: str):
871     """Extract complete database schema as DDL."""
872     try:
873         conn = sqlite3.connect(db_path)
874         cursor = conn.cursor()
875
876         # Get all schema DDL statements (equivalent to .schema command)
877         cursor.execute("""
878             SELECT sql || ';' 
879             FROM sqlite_master
880             WHERE sql IS NOT NULL
881             ORDER BY tbl_name, type DESC, name
882         """)
883
884         schema_statements = cursor.fetchall()
885
886         # Format and write output
887         output_lines = []
888         for (sql,) in schema_statements:
889             output_lines.append(sql)
890
891         schema_output = '\n'.join(output_lines)
892
893         with open(output_file, 'w') as f:
894             f.write(schema_output)
895
896         print(f"Schema extraction complete - wrote {len(schema_statements)} DDL statements")
897         conn.close()
898         return 0
899
900     except Exception as e:
901         print(f"ERROR: {e}", file=sys.stderr)
902         return 1
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917

```