SEGRA: A Structured Experience Guided Reasoning Agent for Property Graph Question Answering

Saiyue Lyu[1]†, Mariam Dundua[2], Vishaal Kapoor[2], Sarthak Ahuja[2], Neda Kordjazi[2], Evren Yortucboylu[2], Harsh Amin[2], Rebecca Steinert[2]
[1]University of British Columbia, [2]Amazon,
saiyue.lyu@ubc.ca, {madundua,vishaalk}@amazon.com
{sarahuja,nedakord,yortuc,haminate,rsteinrt}@amazon.com

Abstract

Enterprise IT support knowledge graphs capture rich relationships among cases, users, devices, symptoms, taxonomic categories, root causes, and historical resolutions. Yet querying them in Gremlin requires knowledge of graph schemas, traversal semantics, edge directionality, and property-graph-specific constraints, making them difficult for non-expert operators to use. We introduce SEGRA, an experience-guided agent for enterprise text-to-Gremlin question answering. SEGRA integrates intent routing, schema- and taxonomy-grounded query generation, multi-shot decomposition, execution-aware verification, and a curriculum-bootstrapped skill library that reuses verified query patterns. On an enterprise IT support benchmark, SEGRA achieves a 7.0× higher mean judge score than backbone-only chain-of-thought prompting. Its skill library further reduces LLM calls by 20% and dollar cost by 18% relative to SEGRA without skills, while preserving answer quality. These results show that schema-grounded agent design and reusable execution experience improve both accuracy and efficiency for enterprise graph QA.

1 Introduction

Graph databases are widely used to model connected data in applications such as fraud detection, knowledge management, recommendation, and enterprise analytics (Guo et al., 2020; Huang et al., 2022; Besta et al., 2023). Unlike relational databases, they represent information as vertices and edges, allowing connected entities to be traversed directly rather than reconstructed through joins (Rodriguez, 2015; Francis et al., 2018). This structure is well suited to enterprise IT support, where cases are linked to users, devices, symptoms, root causes, resolutions, timestamps, and related historical incidents. However, querying such graphs requires fluency in graph query languages, schema labels, edge directionality, and label-versus-property semantics (Angles et al., 2017), creating a barrier for non-expert operators, who must rely on a few specialists to query the graph at production volume.

Large language models offer a natural way to bridge this gap by translating user questions into executable database queries. Text-to-SQL has been extensively studied for relational databases, with mature benchmarks and methods for mapping natural language to SQL (Pourreza & Rafiei, 2023; Chen et al., 2024; Wang et al., 2025). In contrast, text-to-graph-query generation remains less developed. Graph databases introduce challenges beyond SQL, including traversal semantics, schema-specific vertex and edge labels, and label-versus-property distinctions. While Cypher generation has received recent attention (Ozsoy et al., 2025; Lyu et al., 2026), Gremlin generation remains relatively understudied in the academic literature.

Enterprise IT support graphs introduce more challenges beyond standard text-to-database settings. Diagnostic questions often require intermediate operations such as comparisons, set

†Corresponding author. Work done during an internship at Amazon.

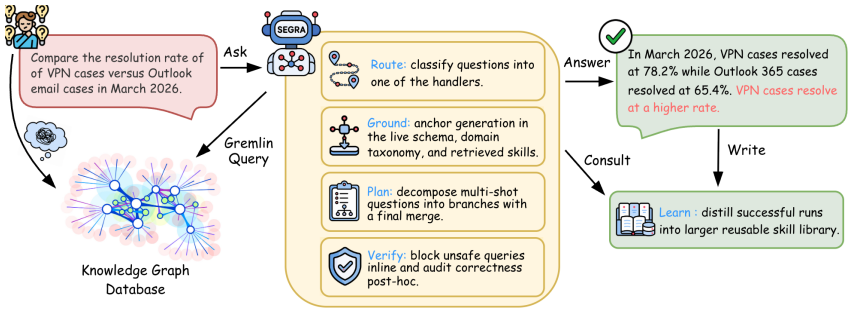

Figure 1: Overview of SEGRA framework. SEGRA takes natural-language questions as input and produces Gremlin queries to further generate natural-language answers.

differences, or top-N aggregation, making single-step Gremlin generation brittle. The model must also align user phrasing with a domain taxonomy of symptoms, causes, categories, and resolutions, rather than guessing schema labels from surface text. In addition, derived similarity edges encode domain-specific relationships whose traversal patterns are easy to misuse without explicit guidance. These challenges make enterprise IT support a demanding testbed for natural-language-to-Gremlin systems.

We propose **SEGRA**, a **S**tructured **E**xperience **G**uided **R**easoning **A**gent for question answering over enterprise property graphs, as illustrated in Figure 1. SEGRA grounds generation in the graph schema and taxonomy, supports similarity queries through dedicated traversal patterns, and decomposes complex questions into verified sub-queries. Because public text-to-Gremlin benchmarks remain limited, SEGRA is designed to accumulate experience from successful executions: verifier-passing executions are distilled into reusable skills that capture effective query patterns and reasoning strategies. These skills guide future queries and further reduce LLM calls and inference cost.

SEGRA is a system-level framework that unifies intent routing, schema-grounded generation, similarity handling, multi-shot planning, execution-aware verification, and cross-query skill accumulation. To the best of our knowledge, it is the first text-to-Gremlin agent to pair reusable execution experience with property-graph-specific correctness checks. Although evaluated on enterprise IT support, the design applies to property-graph interfaces requiring domain grounding, derived relations, and multi-step query composition.

2 Related Work

Natural-language querying of structured data. Text-to-SQL has been extensively studied for relational databases (Yu et al., 2018; Li et al., 2023), while text-to-graph-query generation has received recent attention primarily for Cypher (Ozsoy et al., 2025; Lyu et al., 2026), leaving Gremlin relatively underexplored. KGQA systems combine retrieval and reasoning over entity-relation graphs (Sun et al., 2024; Luo et al., 2024) but target answer retrieval rather than executable queries. SEGRA addresses the less-studied setting of generating executable Gremlin over enterprise property graphs, whose schemas combine domain taxonomies, temporal constraints, and derived similarity edges, and whose queries require traversal operators, edge directionality, aggregation, and execution constraints.

Experience-guided LLM agents. Recent LLM agents improve without weight updates by accumulating reusable experience: Voyager (Wang et al., 2023) stores successful action programs in a skill library, AFlow (Zhang et al., 2025b) searches over workflows of LLM operators, and related systems explore persistent memory and reflection (Shinn et al., 2023; Zhang et al., 2025a). The absence of public Text-to-Gremlin training data makes this direction a natural fit for enterprise property-graph QA: SEGRA reuses execution experience by representing complex questions as workflow DAGs, verifying candidates with deterministic constraints, and distilling verifier-passing runs into reusable skills indexed by question template.

IT support question answering. LLM-based IT support work spans diagnostic dialogue (Kapoor et al., 2026; Ahuja et al., 2026), cloud-incident root-cause analysis (Ahmed et al., 2023; Roy et al., 2024; Chen et al., 2025), and natural-language-to-Gremlin generation (Yun et al., 2025). The closest, EICopilot (Yun et al., 2025), generates a single Gremlin script per question for entity-centric summarization using a static example bank. SEGRA instead targets analytical questions over a historical support graph – aggregations, comparisons, top-N, set differences, and similarity traversal – motivating two design choices: multi-shot decomposition with a final merge, since single-shot generation breaks on compositional questions; and an experience-guided skill library that grows online from verifier-passing runs.

3 Method

We denote the enterprise property graph by G and its live schema by Σ (vertex labels, edge labels, per-label property keys), discovered once at startup. Let $\mathcal{T}$ denote the domain taxonomy of case categories.

As shown in Figure 2, SEGRA processes a natural-language question q in five stages. ① **Route.** An intent classifier assigns q a route label $\rho \in \mathcal{R} = \{\text{DIR}, \text{SIM}, \text{PLAN}\}$, each corresponding to a handler. ② **Ground.** SEGRA grounds generation in Σ and $\mathcal{T}$ via schema discovery, taxonomy lookup, and vertex sampling, replacing model guesses with values from the live graph. The chosen handler also retrieves the top k skills from $\mathcal{S}$ by cosine similarity to q, conditioning generation on past runs. ③ **Plan.** When $\rho = \text{PLAN}$, q is decomposed into a plan $\mathcal{P}$ of parallel and sequential branches with a final merge step. ④ **Verify.** A runtime safety wrapper blocks unsafe Gremlin, an inline plan validator checks $\mathcal{P}$, and a post-hoc property verifier audits captured queries against four similarity-edge invariants. ⑤ **Learn.** Every verifier-passing run is distilled into a skill, regardless of route. The library grows monotonically as evaluation proceeds, so SEGRA conditions later questions on a richer experience archive. The full procedure is formalized in Appendix B.

3.1 Direct and Similarity Handlers

The Direct (H_{DIR}) and Similarity (H_{SIM}) handlers both answer questions in a single LLM tool-use loop that emits one Gremlin query against G. They differ in what the question targets: H_{DIR} filters and aggregates over case properties and the taxonomy (e.g., *"how many VPN cases were resolved last quarter?"*), while H_{SIM} traverses derived similarity edges between cases to retrieve solved history (e.g., *"what resolutions worked for cases similar to V123?"*). Both ground generation in Σ and $\mathcal{T}$, since the model has no view of the live graph and would otherwise have to guess vertex labels and taxonomy values from training-time knowledge. Both also retrieve relevant skills from $\mathcal{S}$ before generation (§3.4), giving the model in-context Gremlin patterns that succeeded on similar past questions.

Direct (H_{DIR}). This handler runs an LLM tool-use loop with five tools. SCHEMA-DISCOVERY retrieves Σ from the live graph G and injects it into the system prompt. TAXONOMY-LOOKUP resolves user phrasing to values in the hierarchical taxonomy $\mathcal{T}$, returning matching entries ranked into *strong* matches (the topic appears as a token in the entry name) and *weak* matches (the topic only appears in the description); strong matches are used directly in `P.within(...)` filters, while weak matches are surfaced with explicit instructions to include them only when the description confirms relevance. VERTEX-SAMPLING returns up to three real vertices of a given label with their property values, enabling the model to verify property formats before composing filters. TEXT-SEARCH runs `TextP.containing(topic)` against the free-text `content` property of `reason`, `symptoms`, and `rootCause` vertices and reports per-label case counts plus three sample texts, recovering questions whose topics lack taxonomy entries. EXECUTE-GREMLIN forwards the LLM-emitted Gremlin to the runtime safety wrapper (§3.3) and then to G, returning results back into the loop.

Similarity (H_{SIM}). This handler runs the base loop with three additional tools that cover two execution patterns: an *anchored walk* from a referenced case, and a *free-text seeded walk*

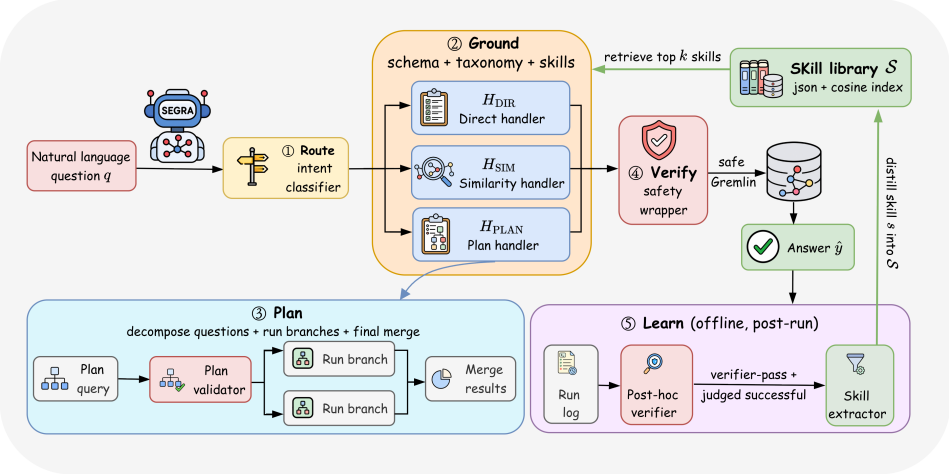

Figure 2: Pipeline of SEGRA framework: route to a handler, ground in the live schema and taxonomy with retrieved skills, optionally plan and execute branches, verify, and distill verifier-passing runs into the skill library that guides future questions.

when no case is referenced. The intent router selects the pattern by checking q for a case-ID, so the handler dispatches to the right tools without runtime disambiguation.

The anchored walk handles questions like *"what is similar to case V123?"*. RESOLVE-CASE verifies the referenced case exists and reports which of its content links (to reason, symptoms, root cause) are populated, so the model picks a feasible similarity edge. SIMILAR-CASES then executes the canonical bidirectional walk (*case* $\rightarrow$ *entity* $\rightarrow$ `bothE(similar*)` $\rightarrow$ *entity* $\rightarrow$ *case*) with a score threshold, avoiding the asymmetric-traversal failure mode where directional `outE` drops a substantial fraction of peers.

The free-text seeded walk handles questions like *"find cases like a VPN authentication failure"*. VECTOR-SEARCH-CASES retrieves cases by free-text similarity in embedding space, providing seeds that the model can optionally chain into SIMILAR-CASES to expand. The post-hoc property verifier (§3.3) audits captured similarity queries from both patterns offline.

3.2 Plan: Multi-shot Decomposition

The Plan handler H_{PLAN} targets questions that no single Gremlin query can answer, such as comparisons (*"compare resolution rate of VPN vs. Outlook"*), set differences, and top-N-then-aggregate queries. It decomposes the question into a structured plan $\mathcal{P}$ of dependent branches, dispatches each branch through the router, and merges branch outputs into a final answer.

Operators. H_{PLAN} exposes three tools to the LLM. PLAN-QUERY produces $\mathcal{P}$, specifying branches with their dependencies, sub-question texts, target handlers $\rho_b \in \mathcal{R} \setminus \{\text{PLAN}\}$, and a merge strategy; the plan is validated inline (§3.3) before execution. RUN-BRANCH dispatches each branch recursively through the router into H_{ρ_b}; independent branches run in parallel. MERGE-RESULTS combines branch outputs to produce the final answer without further graph traversal.

Use of retrieved skills. Like the Direct and Similarity handlers, H_{PLAN} retrieves skills from $\mathcal{S}$ at the Ground stage, but uses them more heavily: retrieved skills surface past plan structures and per-branch Gremlin templates that the planner adapts to the new question. Multi-shot questions benefit most from this guidance, since past worked examples constrain a much larger decision space — where to split the question, what merge strategy applies, and how each branch should be phrased — than single-query routes face.

3.3 Verify: Correctness Mechanisms

SEGRA enforces correctness through three mechanisms at different stages of the pipeline: a runtime sanitiser and a plan validator guard execution inline, while a post-hoc property verifier audits captured queries offline and gates skill extraction.

Runtime safety wrapper. Every Gremlin query emitted by EXECUTE-GREMLIN passes through a sanitiser before reaching G. It strips markdown fences, blocks mutations, rejects full-graph scans, and auto-injects `.limit(`N`)` on non-aggregating queries. We extend the blocklist with bare similarity-edge scans (e.g., `g.E().hasLabel('similarReason')`), which are unsafe on graphs with millions of edges per similarity label. Violations return to the tool-use loop as structured errors for correction.

Inline plan validator. Plans from PLAN-QUERY are validated before execution: each branch must declare a route hint $\rho_b \in \mathcal{R} \setminus \{\text{PLAN}\}$, the merge strategy must come from a fixed vocabulary, the dependency graph must be acyclic, and recursion through RUN-BRANCH is bounded. Invalid plans are rejected, preventing malformed decompositions from reaching G.

Post-hoc property verifier. A regex-based offline checker audits four similarity-edge correctness properties on the captured Gremlin history (full definitions in Appendix C). The verifier walks every captured query regardless of which handler produced it. A run that passes all properties counts as *verifier-pass*, which gates skill extraction (§3.4).

3.4 Learn: Skill Library

The skill library $\mathcal{S}$ persists experience across runs with an in-memory cosine-similarity index over question embeddings. All three handlers (H_{DIR}, H_{SIM}, H_{PLAN}) consult $\mathcal{S}$ during the Ground stage and contribute to it after a verifier-passing run.

Skill schema. A skill records the question template, the route that answered it, and the captured Gremlin templates with entity-specific tokens abstracted; multi-shot skills additionally record the plan structure and merge strategy.

Retrieval. For each question, the library returns the top k skills by cosine similarity to the question embedding. H_{DIR} and H_{SIM} see worked Gremlin templates as in-context examples; H_{PLAN} additionally sees plan structures it can adapt.

Online accumulation. On the shuffled evaluation set, SEGRA processes questions sequentially: each handler retrieves from $\mathcal{S}$, generates, and writes back a new skill when its run is verifier-pass and judged successful. Each question therefore retrieves from a strictly larger library, accumulating experience across the evaluation sequence rather than across separate training and test phases. We isolate this effect with a baseline that runs the same questions in the same order with $\mathcal{S}$ disabled (§5).

4 Experimental Setup

We evaluate SEGRA on an enterprise IT support QA graph using a benchmark of 156 natural-language questions that mirror the analytical and diagnostic queries operators issue in production. The questions are constructed by the authors from recurring enterprise IT support information needs, spanning direct lookup, taxonomy-based aggregation, similarity-driven retrieval, and multi-step diagnostic analysis. Our primary deployment-oriented backbone is a commercially hosted instruction-following LLM in the `Medium` tier, selected to reflect a strong cost–quality tradeoff for practical deployment. To assess whether SEGRA's gains depend on backbone capability, we also evaluate `Small` and `Large` tier backbones under the same protocol. We anonymize the backbone identities to keep the evaluation focused

on the SEGRA framework rather than on benchmarking or ranking specific commercial models.[1] Full details of the graph database and question set are in Section D.

4.1 Systems Comparison

We compare three systems that together stage the contribution of SEGRA into a backbone-only baseline, the architecture without skill memory, and the architecture with skill memory. All three share the same generator LLM, graph access, evaluation order, and validation protocol, and differ only in which mechanisms they expose. ① **CoT.** A single LLM call asked to emit a Gremlin query with chain-of-thought reasoning. No live schema, no tools, no skill library. Establishes the floor: how well a generic LLM can guess at an unseen schema. ② **Few-shot prompting CoT.** A single LLM call with the live schema Σ in the prompt and five worked (question, Gremlin) examples, but no tools, no execution feedback, and no skill library. Tests whether richer prompting alone, without an agent loop, can close the gap to SEGRA. ③ **SEGRA w/o skills.** Full SEGRA architecture (router, planner, handlers, tool surface, post-hoc verifier) with the skill library $\mathcal{S}$ disabled: retrieval returns the empty set, and runs do not write skills back. Every question is dispatched through the same handler graph as SEGRA w/ skills, so this isolates the contribution of $\mathcal{S}$ from the contribution of the architecture itself. ④ **SEGRA w/ skills.** The complete system. The router classifies intent, retrieved skills condition both the planner outer loop and every dispatched handler sub-loop, and verified runs write skills back into $\mathcal{S}$ via Alg. 3. Matches §3 and is the headline system in §5.

4.2 Evaluation Metrics

LLM-as-judge. A judge LLM scores each run on five orthogonal dimensions – *intent fidelity*, *execution validity*, *grounding*, *completeness*, and *usefulness* – each on a 0–4 scale, with the aggregate `overall_score` taken as the floor of the mean. The judge sees the question, the generated Gremlin, the executed result, and the answer, but not which system produced the run. We use OpenAI GPT-5.4 with medium reasoning as the automatic judge. Since the generator backbones (§4) come from a different model provider, this choice avoids the risk of self-preference in evaluation. Full per-dimension definitions and score anchors are in Appendix D.

Cost and efficiency. For each run we log the tool-use *turns* taken by the generator (one turn per LLM call, planner outer loop plus all sub-loops) and the *token totals*: input tokens, output tokens, cache-read input tokens, and cache-creation input tokens. We report mean tokens and mean turns per question for each system, broken down by reasoning shape, so the cost of routing, decomposition, and skill retrieval can be read alongside their accuracy contribution. Multi-shot questions accumulate tokens and turns across the planner outer loop and every branch sub-run; we report the per-question total to keep comparisons end-to-end.

5 Results

This section compares SEGRA with the baselines in §4.1. We report results across `Small`, `Medium`, `Large` generator backbones to test robustness across model tiers.

5.1 Main Results

Table 1 reports mean judge score, mean LLM calls, and mean dollar cost per question. SEGRA changes the performance regime: backbone-only CoT performs poorly on all model tiers, and adding the schema with five worked examples (Few-shot CoT) raises score by 0.16–0.29 points but is still far short of the SEGRA architecture. Even `Large` with Few-shot

[1]The underlying commercial backbone identities are omitted due to business confidentiality constraints. The tier assignments are fixed across all experiments, and all systems are evaluated under the same protocol.

	Small			Medium			Large		
System	Score	Calls	$/q	Score	Calls	$/q	Score	Calls	$/q
CoT	0.12	1.0	0.003	0.29	1.0	0.008	0.31	1.0	0.042
Few-shot CoT	0.41	1.0	0.006	0.50	1.0	0.018	0.47	1.0	0.111
SEGRA w/o $\mathcal{S}$	1.48	12.9	0.118	1.94	11.3	0.307	2.03	8.8	1.498
SEGRA w/ $\mathcal{S}$	**1.67**	**10.2**	**0.095**	**2.03**	**9.0**	**0.253**	**2.06**	**6.5**	**1.074**

Table 1: Mean judge score, mean LLM calls per question, and mean cost per question by system and backbone. SEGRA improves quality substantially, and the skill library further reduces cost/calls.

CoT scores 0.47, below `Small` running SEGRA without skills (1.48). This shows that neither model scale nor richer prompting alone solves enterprise text-to-Gremlin QA; the system needs schema-grounded planning, tool use, execution feedback, and verification.

The agent architecture accounts for the largest quality gain. Moving from Few-shot CoT to SEGRA without skills improves mean score by $+1.07$ on `Small`, $+1.44$ on `Medium`, and $+1.56$ on `Large`. On our primary `Medium` backbone, SEGRA without skills lifts the score from 0.50 to 1.94, bringing the system close to the judge pass threshold. With skills enabled, `Medium` reaches 2.03, crossing the threshold while reducing the average number of LLM calls from 11.3 to 9.0. Few-shot CoT's failure mode is concentrated on multi-shot questions: with no opportunity to decompose, retry, or self-correct. SEGRA's planner and handler loops convert these into branch-and-merge plans with per-branch execution feedback, which is what the architecture lift in the table reflects.

The skill library mainly improves efficiency rather than replacing the architecture. Across all three backbones, adding $\mathcal{S}$ preserves or slightly improves quality while reducing both calls and cost: calls drop by 21% on `Small`, 20% on `Medium`, and 26% on `Large`, with corresponding cost reductions of 19%, 18%, and 28%. Thus, SEGRA's two components play complementary roles: **the agent architecture provides the quality lift, and the skill library recovers a substantial fraction of the tool-use overhead by reusing verified query patterns.**

5.2 Model-Tier Trade-offs

Figure 3 compares quality against runtime cost and LLM calls across model tiers. SEGRA improves the quality–cost trade-off substantially: `Medium` with SEGRA and skills reaches the same mean score as `Large` with SEGRA without skills (2.03), while costing roughly $1/6$ as much. It is also within 0.03 points of the best-scoring configuration, `Large` with skills, at less than $1/4$ of the cost. These results show that system design can partially substitute for model scale. The SEGRA architecture provides the main quality gain, while the skill library improves efficiency by reducing calls and cost without degrading quality. In practical terms, SEGRA makes a mid-tier backbone competitive with a substantially larger model, and skills improve the efficiency of every backbone.

5.3 Human Evaluation

We manually evaluate SEGRA w/ $\mathcal{S}$ with the `Medium` backbone on all 156 benchmark questions. Two annotators independently score each output using the same 0–4 rubric as the LLM judge. Results are reported by handler: direct querying (H_{DIR}, $n = 36$), similarity traversal (H_{SIM}, $n = 20$), and planner-based multi-shot reasoning (H_{PLAN}, $n = 100$). Annotators see the question, generated Gremlin, execution result, and final answer, but not the judge score; LLM-generated Gremlin explanations are used only as annotation aids. Detailed inter-annotator results are included in Section E.1.

Figure 4 shows that both annotators rate SEGRA higher than the automatic judge across all handlers. Averaged human scores exceed judge scores by 0.96 on H_{DIR}, 1.62 on H_{SIM}, and 0.92 on H_{PLAN}. The disagreement is one-sided: among the 35 questions where the

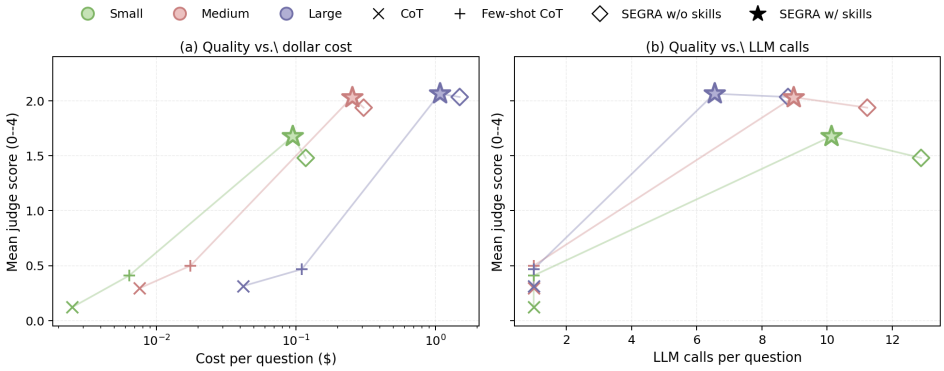

Figure 3: Quality–cost and quality–calls trade-offs. SEGRA architecture lifts every backbone to better score and lower cost and fewer calls.

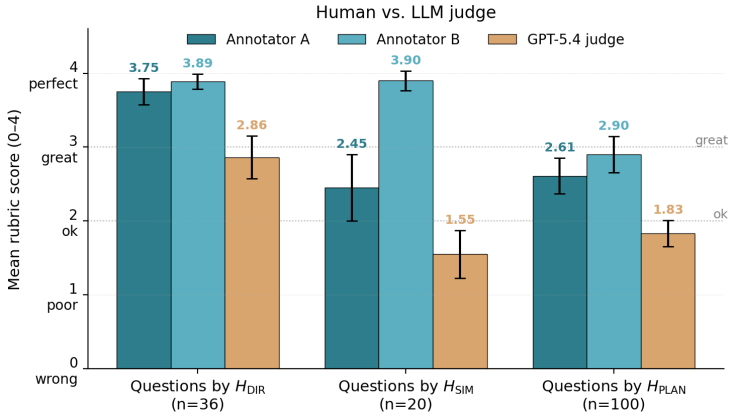

Figure 4: Human and LLM judge mean scores for SEGRA w/ $\mathcal{S}$, stratified by handler. Error bars show 95% confidence intervals.

averaged human score differs from the judge by at least two points, the judge is lower in every case. H_{DIR} is the strongest handler, with over 80% of direct-query outputs rated perfect by both annotators and no poor or wrong cases. Overall, the judge provides a conservative scalable metric, especially for similarity and planner outputs; we therefore interpret judge scores as lower-bound estimates of absolute quality and complement them with a strict count-accuracy analysis in Section A.

6 Conclusion

We presented SEGRA, an experience-guided framework for Gremlin-based question answering over enterprise IT support graphs. SEGRA combines schema- and taxonomy-grounded query generation, similarity-aware traversal handling, planner-based decomposition, execution-aware verification, and a skill library that reuses verified query patterns across questions. Experiments show that SEGRA substantially improves over backbone-only chain-of-thought prompting, while the skill library reduces LLM calls and cost without degrading answer quality. Human evaluation further suggests that automatic judge scores are conservative, especially for planner-based multi-step outputs. Overall, SEGRA shows that structured agent design and reusable execution experience can make natural-language interfaces to property graphs more accurate, efficient, and deployable. The agent design (intent routing, multi-shot decomposition and merge, execution-aware verification, and cross-query skill reuse) is independent of Gremlin and of IT support. The same design therefore transfers to other graph query languages such as Cypher and to other property-graph domains.

Limitations

SEGRA is evaluated on a single enterprise IT support graph, so broader application on other property graphs remains to be validated. The system also assumes access to schema tools, execution feedback, and verifier rules. In longer deployments, the skill library may require maintenance under schema drift. Finally, human evaluation covers only the primary configuration and we use the automatic judge for the full system-by-backbone comparison, leaving broader human evaluation for future work.

References

Toufique Ahmed, Supriyo Ghosh, Chetan Bansal, Thomas Zimmermann, Xuchao Zhang, and Saravan Rajmohan. Recommending root-cause and mitigation steps for cloud incidents using large language models. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1737–1749. IEEE, 2023.

Sarthak Ahuja, Neda Kordjazi, Evren Yortucboylu, Vishaal Kapoor, Mariam Dundua, Yiming Li, Derek Ho, Vaibhavi Padala, Jennifer Whitted, and Rebecca Steinert. Vigil: Towards edge-extended agentic ai for enterprise it support. *arXiv preprint arXiv:2603.16110*, 2026.

Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

Maciej Besta, Robert Gerstenberger, Emanuel Peter, Marc Fischer, Michał Podstawski, Claude Barthels, Gustavo Alonso, and Torsten Hoefler. Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries. *ACM Computing Surveys*, 56(2):1–40, 2023.

Peter Baile Chen, Devin Yang, Weiyue Li, Fabian Wenz, Yi Zhang, Nesime Tatbul, Michael Cafarella, Çağatay Demiralp, and Michael Stonebraker. Beaver: an enterprise benchmark for text-to-sql. *arXiv preprint arXiv:2409.02038*, 2024.

Yinfang Chen, Manish Shetty, Gagan Somashekar, Minghua Ma, Yogesh Simmhan, Jonathan Mace, Chetan Bansal, Rujia Wang, and Saravan Rajmohan. Aiopslab: A holistic framework to evaluate ai agents for enabling autonomous clouds. *Proceedings of Machine Learning and Systems*, 7, 2025.

Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. In *Proceedings of the 2018 international conference on management of data*, pp. 1433–1445, 2018.

Qingyu Guo, Fuzhen Zhuang, Chuan Qin, Hengshu Zhu, Xing Xie, Hui Xiong, and Qing He. A survey on knowledge graph-based recommender systems. *IEEE Transactions on Knowledge and Data Engineering*, 34(8):3549–3568, 2020.

Xuanwen Huang, Yang Yang, Yang Wang, Chunping Wang, Zhisheng Zhang, Jiarong Xu, Lei Chen, and Michalis Vazirgiannis. Dgraph: A large-scale financial dataset for graph anomaly detection. *Advances in Neural Information Processing Systems*, 35:22765–22777, 2022.

Vishaal Kapoor, Mariam Dundua, Sarthak Ahuja, Neda Kordjazi, Evren Yortucboylu, Vaibhavi Padala, Derek Ho, Jennifer Whitted, and Rebecca Steinert. Dqa: Diagnostic question answering for it support. *arXiv preprint arXiv:2604.05350*, 2026.

Jinyang Li, Binyuan Hui, Ge Qu, Jiaxi Yang, Binhua Li, Bowen Li, Bailin Wang, Bowen Qin, Ruiying Geng, Nan Huo, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *Advances in Neural Information Processing Systems*, 36:42330–42357, 2023.

Haoran Luo, E Haihong, Zichen Tang, Shiyao Peng, Yikai Guo, Wentai Zhang, Chenghao Ma, Guanting Dong, Meina Song, Wei Lin, et al. Chatkbqa: A generate-then-retrieve framework for knowledge base question answering with fine-tuned large language models. In *Findings of the association for computational linguistics: ACL 2024*, pp. 2039–2056, 2024.

Songlin Lyu, Lujie Ban, Zihang Wu, Tianqi Luo, Jirong Liu, Chenhao Ma, Yuyu Luo, Nan Tang, Shipeng Qi, Heng Lin, et al. Text2gql-bench: A text to graph query language benchmark [experiment, analysis & benchmark]. *arXiv preprint arXiv:2602.11745*, 2026.

Makbule Gulcin Ozsoy, Leila Messallem, Jon Besga, and Gianandrea Minneci. Text2cypher: Bridging natural language and graph databases. In *Proceedings of the Workshop on Generative AI and Knowledge Graphs (GenAIK)*, pp. 100–108, 2025.

Mohammadreza Pourreza and Davood Rafiei. Din-sql: Decomposed in-context learning of text-to-sql with self-correction. *Advances in neural information processing systems*, 36: 36339–36348, 2023.

Marko A Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th symposium on database programming languages*, pp. 1–10, 2015.

Devjeet Roy, Xuchao Zhang, Rashi Bhave, Chetan Bansal, Pedro Las-Casas, Rodrigo Fonseca, and Saravan Rajmohan. Exploring llm-based agents for root cause analysis. In *Companion proceedings of the 32nd ACM international conference on the foundations of software engineering*, pp. 208–219, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Jiashuo Sun, Chengjin Xu, Lumingyuan Tang, Saizhuo Wang, Chen Lin, Yeyun Gong, Lionel Ni, Heung-Yeung Shum, and Jian Guo. Think-on-graph: Deep and responsible reasoning of large language model on knowledge graph. In *International Conference on Learning Representations*, volume 2024, pp. 3868–3898, 2024.

Bing Wang, Changyu Ren, Jian Yang, Xinnian Liang, Jiaqi Bai, Linzheng Chai, Zhao Yan, Qian-Wen Zhang, Di Yin, Xing Sun, et al. Mac-sql: A multi-agent collaborative framework for text-to-sql. In *Proceedings of the 31st International Conference on Computational Linguistics*, pp. 540–557, 2025.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*, 2023.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pp. 3911–3921, 2018.

Yuhui Yun, Huilong Ye, Jingfeng Deng, Ruojia Li, Xinru Li, Li Li, and Haoyi Xiong. Eicopilot: Search and explore enterprise information over large-scale knowledge graphs with llm-driven agents. In *2025 IEEE International Conference on Big Data (BigData)*, pp. 2689–2696. IEEE, 2025.

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

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, et al. Aflow: Automating agentic workflow generation. In *International Conference on Learning Representations*, volume 2025, pp. 34040–34077, 2025b.

A Strict Count-Accuracy Evaluation

To further study answer quality under deterministic ground truth, we construct a scalar-count analytic subset from the 156-question evaluation benchmark. We include only questions whose gold solution can be represented as a single Gremlin query returning a scalar count. For each included question, a human annotator manually writes the gold Gremlin query, executes it on the graph, and uses the returned count as the reference answer. This filtering excludes display-style questions, list-valued answers, schema-introspection questions, free-form explanations, and multi-shot questions whose gold solution requires decomposition. The resulting subset contains $n = 60$ questions. It is not randomly sampled; it is the eligible subset for which human-written query ground truth and scalar-count comparison are well defined.

For each question q_i, let g_i be the human-written gold Gremlin query and let $\hat{g}_i$ be the query produced by SEGRA. We report two diagnostics. **Query match** measures whether the generated query has the same structure as the human-written gold query after a normalizer $C(\cdot)$ removes known meaning-preserving differences, such as formatting, equivalent interval predicates, symbolic time-window literals, contact–case anchoring variants, and result-preserving deduplication. For example, a generated query using `P.between(a,b)` and a gold query using `P.gte(a).and(P.lt(b))` are treated as matching if the two traversals are otherwise identical. Similarly, harmless differences in whitespace, quote style, or redundant label checks after a fixed vertex anchor are ignored. The metric is exact equality after normalization:

$$\mathrm{Acc}_{\mathrm{query}} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\{C(\hat{g}_i) = C(g_i)\}.$$

Execution match measures whether the generated query returns the same scalar count as the human-written gold query. Let $y_i = \mathrm{Exec}(g_i)$ and $\hat{y}_i = \mathrm{Exec}(\hat{g}_i)$. We count an execution as correct when

$$\mathrm{Acc}_{\mathrm{exec}} = \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}\Big\{\frac{|\hat{y}_i - y_i|}{\max(|y_i|, 1)} \le 0.02\Big\}.$$

We report the average query-match and execution-match accuracy over the 60-question subset. Query match evaluates agreement with the manually written Gremlin structure; execution match evaluates agreement with the human-established scalar answer.

Results. Table 2 reports both metrics for SEGRA w/ $\mathcal{S}$ across the three backbones. `Medium` achieves the highest execution match (54%), while `Large` achieves the highest query match (48%). This shows that structural agreement and answer agreement are related but not identical: a model may generate a different traversal while still computing the correct count.

Backbone	$\mathrm{Acc}_{\mathrm{query}}$	$\mathrm{Acc}_{\mathrm{exec}}$
`Small`	35%	41%
`Medium`	40%	**54**%
`Large`	**48**%	44%

Table 2: Query match and execution match on the $n = 60$ scalar-count analytic subset for SEGRA w/ $\mathcal{S}$.

This subset analysis complements the judge-based and human evaluations by isolating cases where exact scalar-count comparison against human-written Gremlin ground truth is possible. We treat these metrics as diagnostics rather than headline quality measures because they do not capture partial correctness, alternative valid decompositions, or all semantically equivalent Gremlin formulations.

We also reflect the results in Table 2 that the absolute correctness is relatively limited compared to stronger judge and human ratings. We identify three main sources of disagreement: (1) Taxonomy ambiguity. The enterprise taxonomy is large, noisy, and hierarchical. A term in a question may plausibly refer to an issue category, subcategory, issue detail, device

attribute, or textual mention. Queries based on different reasonable taxonomy interpretations can preserve the user's high-level intent but select different populations and return different counts. (2) Temporal and graph-snapshot mismatch. The enterprise graph grows continuously. To make evaluation reproducible, SEGRA is instructed to constrain queries to a fixed time window. Some reference queries omit this window, apply it at a different point in the traversal, or contain answers generated from an earlier graph state. Questions whose temporal scope overlaps or falls outside the fixed window can consequently produce semantically reasonable but numerically different answers. (3) Genuine query-generation errors. Some generated queries omit required taxonomy, text, status, or temporal constraints; traverse the wrong edge or direction; or use incorrect counting, grouping, or deduplication semantics. These are genuine failures rather than alternative interpretations. Thus, the gap should not be interpreted solely as a contradiction between the judge and execution evaluation. The judge assesses whether an answer is semantically appropriate and grounded in its executed result, whereas execution correctness tests agreement with one reference interpretation within a numerical tolerance.

B Full Algorithm

This appendix gives the complete SEGRA evaluation procedure, factored into three algorithms: the outer question-stream loop (Alg. 1), the multi-shot decomposition that runs when the router selects the plan handler (Alg. 2), and the post-run skill update that gates the library on the verifier (Alg. 3). All three operate on the live property graph G with schema Σ, the domain taxonomy $\mathcal{T}$, the routing function R, and the persistent skill library $\mathcal{S}$.

Outer loop (Alg. 1). Questions arrive one at a time in a fixed shuffled order, mirroring how an analyst would issue them in production. For each question q, SEGRA routes via R, retrieves the top k skills from $\mathcal{S}$ by cosine similarity to q, and dispatches to the appropriate handler. Direct and Similarity questions run in a single tool-use loop (H_ρ); plan questions invoke Alg. 2. After the run, UPDATELIBRARY inspects the captured run record and writes a skill back to $\mathcal{S}$ if the verifier accepts it. Each subsequent question therefore retrieves from a strictly larger library; the no-memory baseline disables the write step and feeds the empty $\mathcal{S}_0$ to every question.

Multi-shot decomposition (Alg. 2). The plan handler first emits a plan tree $\mathcal{P}$ with branches, dependencies, target handlers, and a merge strategy, conditioned on the retrieved skills. The plan validator (§3.3) checks $\mathcal{P}$ before any branch executes; on failure, it returns the specific validation error (e.g., a duplicate branch id, an unknown route hint, a cycle in the dependency graph) to the planner LLM as the tool result. The LLM may correct the plan and re-invoke `PLAN-QUERY` on a subsequent turn, subject to the planner's overall turn budget (14 turns shared across plan, dispatch, and merge). Once a valid plan is accepted, the planner dispatches each branch in $\mathcal{P}$ back through the router into H_{ρ_b} with $\rho_b \neq$ PLAN, so a branch is itself a single-handler tool-use loop. In our implementation, branches are dispatched in topological order of their declared dependencies, with independent branches running in parallel. After all branches return, `MERGE-RESULTS` produces the final answer from branch outputs without further graph traversal. The captured Gremlin history of every branch is collected into the run record for the verifier.

Library update (Alg. 3). The post-hoc property verifier V_{prop} (Appendix C) runs over every captured Gremlin query in the run record and reports a verifier-pass flag. A skill is written to $\mathcal{S}$ only when the run is both verifier-pass and judged successful by the LLM judge with score ≥ 2, ensuring that low-quality or structurally invalid runs cannot pollute the library. The skill schema is the same regardless of route (question template, route label, abstracted Gremlin templates), with multi-shot skills additionally recording the plan structure and merge strategy.

Algorithm 1 SEGRA: question evaluation with skill accumulation.

Require: Question stream Q; initial library $\mathcal{S}_0$; router R; live schema Σ; taxonomy $\mathcal{T}$; retrieval budget k.

$\mathcal{S} \leftarrow \mathcal{S}_0$
for $q \in Q$ in fixed order **do**
  $\rho \leftarrow R(q)$
  ▷ intent classification: $\rho \in \{\text{DIR}, \text{SIM}, \text{PLAN}\}$
  $\mathcal{K}_q \leftarrow \text{RETRIEVE}(\mathcal{S}, q, k)$
  ▷ top-k skills by cosine similarity
  if $\rho = \text{PLAN}$ **then**
    $ans, record \leftarrow Alg.\ 2(q, \Sigma, \mathcal{T}, \mathcal{K}_q)$
  else
    $ans, record \leftarrow H_\rho(q, \Sigma, \mathcal{T}, \mathcal{K}_q)$
  ▷ single-handler tool-use loop
  end if
  $\mathcal{S} \leftarrow Alg.\ 3(\mathcal{S}, record)$
end for
return answers and final $\mathcal{S}$

Algorithm 2 PLAN-RUN-MERGE: multi-shot decomposition invoked from Alg. 1 when $\rho = \text{PLAN}$.

Require: q, Σ, $\mathcal{T}$, retrieved skills $\mathcal{K}_q$.

$\mathcal{P} \leftarrow \text{PLAN-QUERY}(q, \Sigma, \mathcal{T}, \mathcal{K}_q)$
  ▷ plan tree with branches and merge m
if plan validator rejects $\mathcal{P}$ **then**
  return validation error as tool result
end if
for branch b in $\mathcal{P}$ **do**
  $r_b \leftarrow H_{\rho_b}(q_b, \Sigma, \mathcal{T}, \mathcal{K}_q)$
  ▷ recursive dispatch via R, $\rho_b \neq \text{PLAN}$
end for
$ans \leftarrow \text{MERGE-RESULTS}(\{r_b\}_b, m)$
  ▷ LLM-only step, no graph traversal
return ans, run record (plan, branch outputs, Gremlin history)

Algorithm 3 UPDATELIBRARY: post-run skill extraction gated by the verifier and the judge.

Require: Library $\mathcal{S}$; run record from Alg. 1 (Gremlin history, plan, branch outputs, final answer).

$pass \leftarrow V_{\text{prop}}(\text{record.gremlin\_history})$
  ▷ Appendix C
$score \leftarrow \text{JUDGE}(\text{record})$
  ▷ LLM judge, score $\in \{0, 1, 2, 3, 4\}$
if $pass$ and $score \geq 2$ **then**
  $s \leftarrow \text{EXTRACTSKILL}(\text{record})$
  ▷ question template, route, abstracted Gremlin; plan + merge if $\rho = \text{PLAN}$
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{s\}$
end if
return $\mathcal{S}$

C Post-Hoc Property Verifier

The verifier in §3.3 audits captured Gremlin queries against four similarity-edge correctness properties. This appendix gives the full definition of each property, the failure mode it prevents, and the regex pattern used to detect violations. Notation: $\mathcal{E}_{\text{sim}} = \{\texttt{similarReason}, \texttt{similarSymptoms}, \texttt{similarRootCause}\}$ is the set of similarity edge labels, and the within-case

pairing Φ maps each similarity edge to the within-case edge that brackets it:

$$\Phi(\texttt{similarReason}) = \texttt{hasReason},$$
$$\Phi(\texttt{similarSymptoms}) = \texttt{exhibits},$$
$$\Phi(\texttt{similarRootCause}) = \texttt{attributedTo}.$$

The verifier walks the captured Gremlin history from each run; a run record is *verifier-pass* iff all four properties hold on every captured query. Properties P_1–P_4 are vacuously PASS on queries that contain no similarity edge step (e.g., Direct route queries that filter by property only).

P1: Window-on-Destination

Rationale. Our evaluation graph is a fixed snapshot of the production support graph with all cases tagged by a `createdAt` timestamp, and we restrict every evaluation query to a fixed experiment window [WIN_LO, WIN_HI] so that ground-truth counts and aggregates are reproducible across runs. The verifier therefore audits where in a similarity walk the window predicate appears: P1 enforces the implementation choice that the window must scope the *peer* cases returned by the similarity walk, not just the anchor case the walk starts from.

Statement. Every query containing a similarity-edge step must apply the time-window predicate `P.between`(WIN_LO, WIN_HI) *after* the post-similarity `.in(`$\Phi(e)$`)` step that returns to a case vertex, not on the anchor.

Failure mode prevented. Applying the window filter on the anchor case (before the similarity hop) restricts the source case to the experiment window, but lets the walk return peers from *any* time period. Conversely, applying it on the destination case restricts the peer set itself, which is the intended scoping for similarity questions over the experiment window.

Detection. Pass iff (a) the query contains `P.between`(WIN_LO, WIN_HI) at least once, and (b) at least one `.in(`$\Phi(e)$`)` step is followed by the window predicate later in the query string. Queries that have no `.in(`$\Phi(e)$`)` step (e.g., walks that end at the entity vertex without returning to a case) are vacuously PASS.

P2: Threshold-on-Edge

Statement. For every similarity-edge step `bothE`(e), `outE`(e), or `inE`(e), with $e \in \mathcal{E}_{\text{sim}}$, the query must apply `.has('similarityScore', P.gte(`θ`))` between the edge step and the next vertex step `.otherV()`, `.inV()`, or `.outV()`.

Failure mode prevented. Without an explicit threshold, similarity walks return all peers regardless of similarity score, which dilutes the result set with low-similarity matches. Applying the threshold *after* the vertex step instead of on the edge filters the wrong object — vertices don't carry the similarity score; the edge does.

Detection. For each similarity-edge step, find the substring between it and the next vertex step. Pass iff every such substring contains `.has('similarityScore', P.gte(...))`. Edge steps with no following vertex step (malformed traversals) are vacuously PASS since they do not reach a peer.

P3: Entity-Bracket

Statement. Every similarity-edge step with edge $e \in \mathcal{E}_{\text{sim}}$ must be preceded by a within-case `.out(`$\Phi(e)$`)` step earlier in the query.

Failure mode prevented. Similarity edges connect content vertices (`reason`, `symptoms`, `rootCause`), not cases directly. Walking from a case to a similarity edge without first traversing the corresponding within-case edge is a structural error: the traversal has no source content vertex on which to attach the similarity step.

Detection. For each occurrence of a similarity edge with label e, scan the query string before the edge step and require that `.out(`$\Phi(e)$`)` appears at least once. Note the pairing is per-label: `similarReason` requires `.out('hasReason')` specifically, not `.out('exhibits')` or `.out('attributedTo')`.

P4: Anchor-by-Id

Statement. Case anchors must be addressed by vertex id, either via `g.V('uuid')` or via `.hasId('uuid')`. The label-and-property form `g.V().has('case', 'id', 'uuid')` addresses an `id` property that does not exist on case vertices and is forbidden.

Failure mode prevented. Cases in the live graph do not carry an `id` property; their identity is the vertex id itself. Filtering on `.has('case', 'id', 'uuid')` matches zero vertices and silently returns an empty result, which the LLM may misinterpret as a real (negative) finding.

Detection. Reject any query containing the literal substring pattern `V().has('case', 'id', ...)`. Other anchoring forms (`g.V('uuid')`, `.hasId`) are not flagged.

Verifier Output

For each captured Gremlin query, the verifier emits a per-property record:[2]

```
{
  "P1_window_on_destination": {"pass": bool, "reason": str | null},
  "P2_threshold_on_edge":     {"pass": bool, "reason": str | null},
  "P3_entity_bracket":        {"pass": bool, "reason": str | null},
  "P4_anchor_by_tilde_id":    {"pass": bool, "reason": str | null}
}
```

A failing property surfaces a structured `reason` string identifying the offending substring or position. The run-level *verifier-pass* flag is the conjunction of all per-query per-property results across the captured Gremlin history.

D Detailed Experimental Setup

Graph Database. SEGRA queries an enterprise IT support property graph, backed by a managed graph database and accessed through Gremlin, at production scale. Each case vertex is connected to free-text content vertices ($v_{c_1}, v_{c_2}, v_{c_3}$) covering incident description, observed behavior, and diagnosed cause), resolution vertices, a multi-level hierarchical taxonomy of case categories, and context vertices capturing the parties involved and device/compliance state. Derived similarity-edge labels $\mathcal{E}_{\text{sim}} = \{e_1, e_2, e_3\}$ connect content vertices that share semantic content, each carrying a continuous similarity score used as a threshold for filtering peers (P2 in Appendix C). Every case carries a creation timestamp, used by the experiment-window predicate for reproducibility (P1). SEGRA only reads from the graph and never modifies it.

Dataset. The benchmark has 166 questions split between a 10-question bootstrap set (used to seed $\mathcal{S}$ before evaluation) and a 156-question evaluation set. The evaluation set is partitioned by reasoning shape:

[2]The field name `P4_anchor_by_tilde_id` refers to Gremlin's `~id` notation for vertex IDs (the meta-id, distinct from a property named `id`); the leading tilde is dropped from the JSON key for compatibility.

- **36 single-shot** questions answered by H_{DIR} – filter-and-count, top-N ranking, and aggregate questions over case properties and the taxonomy.
- **20 similarity-anchored** questions answered by H_{SIM} – cross-case similarity questions over derived `similar*` edges, either anchored to a referenced case or seeded by free-text retrieval.
- **100 multi-shot** questions answered by H_{PLAN} – compositional questions that no single Gremlin query can answer.

The 100 multi-shot questions span 20 categories drawn from a failure-mode analysis of pilot runs. Categories include parallel comparisons (*"compare resolution rate of VPN vs. Outlook"*), top-N analytics with sub-aggregation, set differences across temporal windows, recurrence patterns over taxonomic categories, and similarity-driven aggregations. The category-level distribution drives the curriculum-bootstrap procedure described in §3.4. We also include examples from our dataset in Table 3 with the natural-language question, the Gremlin issued, and the final answer. The Direct example issues a single aggregation query; the Similarity example anchors on a case and walks the bidirectional similarity edge with score and time-window filters; the Multi-shot example is decomposed by the planner into two parallel branches whose results are combined by the merge step.

Example for H_{DIR}
Question: How many cases were created in total? **Gremlin:** `g.V().hasLabel('case').count()` **Answer:** 123,456 cases were created in the experiment window.
Example for H_{SIM}
Question: How many in-window cases share similar symptoms with case `4f4ded41-...` above the production threshold? **Gremlin:** `g.V('4f4ded41-...').hasLabel('case').out('exhibits').bothE('similarSymptoms').has('similarityScore', P.gte(0.675)).otherV().in('exhibits').hasLabel('case').hasId(P.neq('4f4ded41-...')).dedup().count()` **Answer:** 4 in-window cases share similar symptoms at or above the production threshold (≥ 0.675).
Example for H_{PLAN}
Question: Which has more cases – Mac or Windows? **Plan:** `parallel_compare`; two independent branches, merged with `compare_ratios`. **Gremlin 1:** `g.V().hasLabel('case').where(out('hasTelemetry').has('devicePlatform', 'MAC')).count()` $\rightarrow$ 12,345. **Gremlin 2:** `g.V().hasLabel('case').where(out('hasTelemetry').has('devicePlatform', 'WINDOWS')).count()` $\rightarrow$ 123,456. **Answer:** Windows has more cases than Mac.

Table 3: Example questions and Gremlin queries for different handlers.

Judge Rubric. The LLM judge (§4.2) scores each run on five orthogonal dimensions, each on a 0–4 anchor scale, and reports a per-dimension rationale plus an aggregate. The anchor scale is shared across dimensions: **4** = great, fully meets the dimension with no caveat worth flagging; **3** = good, solidly meets, at most a cosmetic nit; **2** = OK, mostly meets with a defensible minor gap; **1** = poor, significant gap that affects the answer's value; **0** = failed, does not meet the dimension at all. The aggregate `overall_score` is the *floor* of the mean of the five scores; rounding down rather than to nearest errs on the conservative side.

1. **Intent fidelity.** Does the executed Gremlin capture what the user actually asked? Looks at vertex labels, filter predicates (issue category/subcategory, time window, status), traversal direction, and whether the right entities are anchored. **4**: traversal precisely matches the question, all filters justified. **3**: matches with one inferred-but-reasonable filter. **2**: right shape but a non-decisive filter is missing or slightly off. **1**: wrong vertex type, wrong direction, or a filter inversion. **0**: traversal answers a fundamentally different question.

2. **Execution validity.** Did the query run cleanly and return data of the expected shape? **4**: ran first try, non-empty data of the expected shape. **3**: ran cleanly with a small retry/relaxation noted in the trace. **2**: ran but result is partial / had to be retried successfully. **1**: errored on first try and recovered with a degraded result, or returned data of the wrong shape that still answers the question. **0**: errored without recovery, returned empty when data was clearly expected, or the shape is unusable. An "honest empty" answer that explains why the data is missing earns at most 2.
3. **Grounding.** Is the natural-language answer faithful to what the query actually returned? This is the hallucination check, independent of whether the query was correct. **4**: every numeric, name, and claim is supported by the result. **3**: supported with a single rounding or wording inconsistency. **2**: one minor unsupported detail, headline still correct. **1**: a key claim is unsupported or contradicts the result. **0**: fabricates entities, counts, or relationships not in the data.
4. **Completeness.** Did the answer address every part of the question, including sub-asks ("compare X and Y", "list and explain", "rank by Z")? **4**: every sub-ask answered with explicit comparisons or explanations. **3**: every sub-ask answered, one is briefer than ideal. **2**: main ask answered, secondary sub-ask glossed. **1**: only one of multiple sub-asks addressed. **0**: answer is on a different topic or refuses without justification. For single-ask questions, the default is 4 unless something obvious is omitted.
5. **Usefulness.** Would an analyst actually use this answer to make a decision? Vague hedging is penalised, while honest fallbacks (e.g., "vector search unavailable, here is the text-search fallback") are rewarded because the analyst can act on them. **4**: directly actionable, no follow-up needed. **3**: actionable with a one-line caveat the answer surfaces. **2**: useful with a meaningful caveat. **1**: ambiguous; analyst would need to re-run the query. **0**: misleading, evasive, or surfaces no actionable signal.

A run is counted as pass when `overall_score` ≥ 2. The judge is instructed to reply with a single JSON object containing the five per-dimension records, the aggregate, and the booleans, with no preamble or code fences.

E Per-Dimension Breakdown

Table 4 reports the mean per-dimension GPT-5.4 judge scores that compose the aggregate `overall_score` reported in §5, for each (system, backbone) cell over the 156-question evaluation set. The five dimensions follow the rubric defined in Appendix D: *Intent* = intent fidelity, *Exec* = execution validity, *Ground* = grounding, *Compl* = completeness, *Use* = usefulness. The aggregate `overall_score` reported in Table 1 is the floor of the per-question mean of these five dimensions, so the per-dimension means here will be slightly higher than the aggregate (mean before floor vs. floor before mean across questions).

The pattern across systems is consistent. CoT and Few-shot CoT score reasonably on *Intent* (the model usually understands what is being asked) but collapse on every other dimension because, with no tools, queries either fail outright or return a degenerate shape. Few-shot CoT lifts *Intent* above CoT by 0.4–0.6 points on every backbone – the worked examples teach traversal patterns – but *Execution* stays under 1.0, confirming that single-call prompting cannot recover from the schema-shaped errors that tool feedback would catch. The SEGRA architecture pulls *Execution*, *Grounding*, and *Completeness* into the 2–3 range, and adding the skill library lifts *Usefulness* (the most synthesis-heavy dimension) by a further 0.02–0.30 points.

E.1 Human-vs-Judge Agreement

The main paper's per-dimension breakdown reports GPT-5.4 judge scores. We additionally collected human annotations on the SEGRA-w/-skills + `Medium` cell to gauge how the automatic judge compares to expert evaluation. Two annotators independently scored all 156 questions on the same 0–4 anchor scale as the LLM judge, but assigned a single *overall* score per question rather than per-dimension scores – so this section is restricted to overall-score agreement and does not extend the per-dimension table above. Annotators

	Small					Medium					Large				
System	Intent	Exec	Ground	Compl	Use	Intent	Exec	Ground	Compl	Use	Intent	Exec	Ground	Compl	Use
CoT	1.29	0.08	1.19	0.06	0.25	1.73	0.21	1.36	0.08	0.22	1.59	0.08	1.54	0.08	0.24
Few-shot CoT	**1.91**	0.64	0.94	0.42	0.32	**2.17**	0.88	1.12	0.36	0.22	**2.08**	0.99	1.05	0.36	0.22
SEGRA w/o $\mathcal{S}$	1.59	2.26	**1.68**	2.54	1.24	1.92	**3.12**	2.06	3.15	1.46	1.91	**3.14**	2.15	3.33	1.74
SEGRA w/ $\mathcal{S}$	1.75	**2.72**	1.63	**2.99**	**1.26**	2.16	3.11	**2.15**	**3.15**	**1.75**	1.99	3.13	**2.20**	**3.41**	**1.76**

Table 4: Per-dimension mean GPT-5.4 judge scores (0–4) over the 156-question evaluation set, by system and backbone. Dimensions are defined in Appendix D. Bold marks the highest-scoring system per (backbone, dimension); SEGRA w/ $\mathcal{S}$ attains the highest score on a majority of cells, with the most consistent gains on *completeness* and *usefulness* (which reward end-to-end synthesis rather than just running a query). Note the contrast on *Intent*: Few-shot CoT slightly exceeds SEGRA on this dimension because both reach a ceiling from prompt-level grounding, but its low *Execution* drags the aggregate down.

Handler	n	Exact	≤ 1 pt	κ_q	ρ	MAD
H_{DIR}	36	80.6%	94.4%	0.13	0.25	0.25
H_{SIM}	20	15.0%	55.0%	0.06	0.35	1.45
H_{PLAN}	100	42.0%	82.0%	0.56	0.50	0.81
All	156	47.4%	81.4%	0.52	0.47	0.76

Table 5: Inter-annotator agreement on SEGRA w/ $\mathcal{S}$, `Medium` backbone. "Exact" is the percentage of questions on which both annotators picked the same integer score; "≤ 1 pt" allows a one-bucket disagreement. κ_q is Cohen's quadratic-weighted kappa, ρ is Spearman rank correlation, and MAD is the mean absolute difference between the two annotators' integer scores. The H_{DIR} κ_q is depressed by a ceiling effect (both annotators rate nearly all outputs ≥ 3).

saw the question, generated Gremlin, executed result, and final answer, but not the judge score.

Inter-annotator agreement. Before comparing humans against the judge, we measure how the two annotators agree with each other on the same 156 questions. Table 5 reports five complementary metrics, since the 0–4 scale is ordinal: exact agreement (both annotators picked the same integer), within-1-point agreement (they differed by at most one bucket), Cohen's quadratic-weighted κ (the standard ordinal-agreement statistic, where ≥ 0.6 is conventionally read as "substantial"), Spearman's ρ (rank correlation, robust to one annotator being uniformly harsher than the other), and mean absolute difference (concrete points-apart). Overall, the annotators agree exactly on 47.4% of questions and within one point on 81.4%, with a quadratic-weighted κ of 0.52 and a Spearman ρ of 0.47 – moderate agreement under standard interpretation, with the bulk of disagreements being one-bucket calls rather than wholesale reversals.

The handler-level breakdown reveals two distinct patterns. On H_{DIR}, both annotators rate nearly every output as great (A-mean 3.75, B-mean 3.89, within-1 94.4%); the κ of 0.13 is misleadingly low because the score distribution has almost no variance for κ to credit. On H_{PLAN}, agreement is moderate-to-substantial ($\kappa = 0.56$, $\rho = 0.50$, within-1 82%) – the largest sub-sample (n=100) and the most representative of overall agreement. On H_{SIM}, agreement is genuinely poor ($\kappa = 0.06$, exact 15%, MAD 1.45): Annotator A is systematically harsh on similarity outputs (mean 2.45) while Annotator B rates almost all 20 similarity outputs as great (mean 3.90).

Table 6: Sensitivity of evaluation to question ordering across five random seeds.

Seed	Mean score (0–3)	≥ 2 rate	≥ 3 rate	Skills at end
0	2.674	96.7%	70.7%	108
1	2.552	91.7%	63.5%	110
2	2.544	95.6%	58.9%	109
3	2.534	89.8%	63.6%	110
4	2.511	92.0%	59.1%	110
Cross-seed	**2.563 ± 0.064**	**93.2% ± 2.9%**	**63.2% ± 4.8%**	**109.4 ± 0.9**

Table 7: Seven-system ladder evaluated by three judge models across five dimensions. **I** = intent fidelity, **E** = execution validity, **G** = grounding, **C** = completeness, and **U** = usefulness.

	Small					Medium					Large				
System	I	E	G	C	U	I	E	G	C	U	I	E	G	C	U
CoT	1.29	0.08	1.19	0.06	0.25	1.73	0.21	1.36	0.08	0.22	1.59	0.08	1.54	0.08	0.24
Few-shot CoT	**1.91**	0.64	0.94	0.42	0.32	**2.17**	0.88	1.12	0.36	0.22	**2.08**	0.99	1.05	0.36	0.22
CoT + tools (ReAct-style)	1.62	2.10	1.51	2.08	1.13	2.04	3.06	2.05	2.84	1.59	2.04	3.11	**2.21**	2.99	**1.82**
SEGRA without planner	1.71	2.19	1.38	2.38	1.22	1.98	3.07	2.09	2.63	1.47	1.78	2.77	2.19	2.76	1.67
SEGRA without skills	1.59	2.26	**1.68**	2.54	1.24	1.90	3.10	2.05	3.13	1.46	1.90	3.12	2.13	3.31	1.73
SEGRA, planner-only retrieval	1.58	2.57	1.52	2.74	**1.26**	2.04	3.07	2.02	3.14	1.62	1.87	3.02	2.25	3.09	1.81
SEGRA with skills	1.74	**2.71**	1.62	**2.97**	1.25	2.16	**3.11**	**2.15**	**3.15**	**1.75**	1.99	**3.13**	2.20	**3.41**	1.76

F Additional Results

F.1 Question Streaming Order

To measure the sensitivity of question order during evaluation, we ran the 100 planner evaluation questions under five random orderings, each initialized with the same 10 bootstrapped skill library. And the results are illustrated in Table 6.

Across the five orderings, the mean score is 2.563 ± 0.064, the fraction of questions scoring at least 2 is 93.2% ± 2.9%, and the fraction scoring 3 is 63.2% ± 4.8%. The variation in mean score is approximately 2% of the full 0–3 scale, indicating limited sensitivity to question order. The learned skill libraries also converge closely. Starting from the same 10 bootstrap skills, each run promotes approximately 100 additional skills and ends with 108–110 skills. The mean pairwise Jaccard similarity is 0.989 when bootstrap skills are included and 0.988 when considering only skills added during evaluation. Overall, these results suggest that both answer quality and the resulting skill library are stable across the tested question orderings. Our current experiments target the intended streaming setting, in which verified experiences can be incorporated as new questions arrive.

F.2 More Baseline Comparison

We expand the evaluation to a seven-system ladder, including prompt-only, few-shot, ReAct-style, no-planner, no-skills, planner-only retrieval, and full SEGRA configurations. The results are illustrated in Table 7.

We observed that (1) adding the schema and examples primarily improves intent recognition, but not execution. Few-shot CoT obtains the highest intent score on each backbone but remains below 1.0 on execution because it cannot inspect the graph, execute queries, or recover from errors. (2) adding ReAct-style tool use produces the largest execution improvement over the prompting-only baselines. This confirms that live schema access and iterative execution feedback are critical for text-to-Gremlin generation. (3) explicit planning most consistently improves completeness: full SEGRA exceeds the no-planner variant by 0.59, 0.52, and 0.65 points on the small, medium, and large backbones, respectively. (4) On the largest backbone, ReAct and full SEGRA perform similarly on execution, while SEGRA achieves substantially higher completeness. This suggests that strong models can perform more implicit planning within a general tool loop, whereas explicit decomposition remains valuable for multi-part questions.