PARSE: LLM Driven Schema Optimization for Reliable Entity Extraction

Anubhav Shrimal*, Aryan Jain*, Soumyajit Chowdhury, Promod Yenigalla

RBS Tech Sciences, Amazon

{shrimaa, arynjn}@amazon.com

*Equal contribution

Abstract

Structured information extraction from unstructured text is critical for emerging Software 3.0 systems where LLM agents autonomously interact with APIs and tools. Recent approaches apply large language models directly to extraction tasks using existing JSON schemas, often with constraint decoding or reinforcement learning approaches to ensure syntactic validity, but treat JSON schemas as static contracts designed for human developers, leading to suboptimal extraction performance, frequent hallucinations, and unreliable agent behavior when schemas contain ambiguous or incomplete specifications. We recognize that JSON schemas themselves are a form of natural language understanding contract that encodes rules, relationships, and expectations about data structure contracts that LLMs should be able to both interpret and systematically improve. Consequently, we develop PARSE (Parameter Automated Refinement and Schema Extraction), a novel system with two synergistic components: ARCHITECT, which autonomously optimizes JSON schemas for LLM consumption while maintaining backward compatibility through RELAY (an integrated code generation system), and SCOPE, which implements reflection-based extraction with combined static and LLM-based guardrails. We evaluate PARSE qualitatively and quantitatively on three datasets including Schema-Guided Dialogue (SGD), Structured Web Data Extraction (SWDE), and internal retail conversation data, and find that it achieves up to 64.7% improvement in extraction accuracy on SWDE with combined framework improvements reaching 10% across models, while reducing extraction errors by 92% within the first retry and and maintaining practical latency.

1 Introduction

The emergence of Software 3.0 represents a fundamental shift from static, form-based applications to dynamic systems where Large Language Model (LLM) agents autonomously interact with APIs and tools to accomplish complex tasks. In this new paradigm, reliable structured information extraction from unstructured text becomes mission-critical—agents must accurately parse natural language requests, extract precise parameters, and invoke the correct tools with valid arguments. Unlike traditional software where human developers handle data transformation complexity, LLM agents must perform this extraction reliably at scale, making the difference between a helpful assistant and a system that fails catastrophically in production.

What makes reliable structured extraction so challenging for LLM agents? Current approaches face a fundamental mismatch: JSON schemas that define expected output structures were designed as contracts between human developers and static systems, not as instructions for LLM agents. These schemas often contain ambiguous descriptions, incomplete validation rules, and structural choices optimized for human readability rather than machine comprehension. When LLM agents attempt extraction using these human-centric schemas, they struggle with unclear entity boundaries, conflicting requirements, and insufficient context about field relationships, leading to frequent hallucinations and schema non-adherence.

Existing work has focused primarily on forcing LLMs to conform to existing schema structures through constraint decoding, reinforcement learning, and self-correction mechanisms. However, these approaches treat schemas as immutable contracts rather than optimizing the structures themselves for LLM comprehension.

We observe that JSON schemas are themselves a form of natural language understanding contract—they encode rules, relationships, and expectations using descriptive text and validation logic that LLMs should be able to both interpret and systematically improve. Rather than viewing

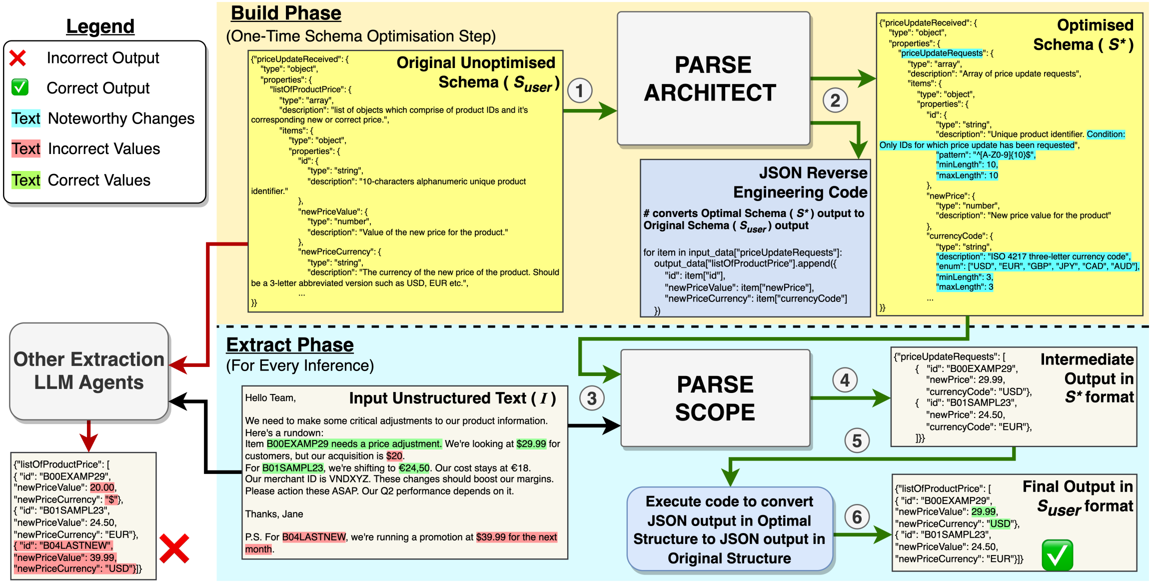

Figure 1: Overview of PARSE framework showing schema optimization and extraction pipeline. The system takes an unoptimized schema and input text (left), processes it through ARCHITECT for schema refinement and SCOPE for extraction (center), producing accurate structured output (right). The diagram highlights how PARSE outperforms other information extraction agents by implementing robust schema optimization and validation. Numbers indicate the sequential flow of operations.

schemas as static artifacts, we can treat them as evolving interfaces optimized specifically for LLM consumption while maintaining backward compatibility with existing systems.

We present PARSE (Parameter Automated Refinement and Schema Extraction), a comprehensive system addressing both sides of the structured extraction challenge through two synergistic components. Together, these components create a virtuous cycle where schema optimization improves extraction performance, and extraction errors inform further schema refinement.

Our main contributions include: (1) **ARCHITECT (Automated Refinement and Conversion Handler for Information Transformation and EnhanCemenT):** A novel automated schema optimization framework that iteratively refines JSON schemas for LLM consumption by analyzing extraction performance, adding detailed entity descriptions and validation rules, and restructuring schemas for clearer representation while maintaining backward compatibility through RELAY, our automated transformation code generator. ARCHITECT includes RELAY (Reverse Engineering Layer for Automated Yoking), which automatically generates Python transformation code to maintain compatibility with original schema formats. (2) **SCOPE (Schema Compliant Organized Pattern Extractor):** A comprehensive reflection-based extraction framework that combines static and LLM-based guardrails to enable systematic error identification and correction, ensuring reliable structured information extraction through sophisticated validation mechanisms. Our evaluation demonstrates that PARSE achieves up to 64.7% improvement in extraction accuracy compared to state-of-the-art baselines on SWDE, with combined framework improvements reaching 10% across SGD, SWDE, and internal retail conversation datasets, representing a paradigm shift toward treating structured extraction as a co-optimization problem between schema design and extraction mechanisms—enabling the reliable LLM agent systems that Software 3.0 applications demand.

2 Related Work

Our work builds upon and extends several key research areas in information extraction, schema optimization, and LLM applications. This section provides a comprehensive review of relevant work across these domains.

LLM-based Structured Extraction and Schema Compliance: Large Language Models have transformed information extraction from discriminative to generative approaches that produce structured outputs directly from unstructured text (Zhang et al., 2025). Two-stage frameworks combining general LLMs with domain-specific refinement show promise for complex extraction tasks (Zhang et al., 2024), but current methods face significant limitations. LLMs struggle with complex schemas, exhibiting substantial performance gaps compared to traditional methods, particularly for nested entity recognition (Deng et al., 2022; Han et al., 2023). Common error patterns include missing spans, incorrect types, and schema non-adherence, with GPT-4 showing an 11.97% invalid response rate for complex extraction tasks (Han et al., 2023). Traditional JSON generation approaches have evolved from simple prompting to sophisticated constraint enforcement. Constraint decoding methods like Outlines guarantee schema compliance through grammar-guided generation but sometimes sacrifice output quality (Lu et al., 2025a; Agarwal et al., 2025). Reinforcement learning approaches achieve high valid JSON rates (98.7% vs. 82.3% baseline) by using schemas as training signals (Agarwal et al., 2025), while the "Thought of Structure" paradigm shows substantial improvements (44.89% gain) by encouraging structural reasoning before generation (Lu et al., 2025b). However, these approaches focus on syntactic validity rather than optimizing schemas for LLM comprehension.

Self-Correction and Agent Frameworks: Self-correction techniques enhance LLM reliability through Chain-of-Thought reasoning, self-verification, and iterative refinement mechanisms (Kumar et al., 2025). The Chain of Self-Correction framework embeds correction as an inherent ability through iterative generation and verification (Gao et al., 2024). Runtime guardrail mechanisms like AgentSpec provide lightweight constraint enforcement through domain-specific languages (Wang et al., 2025), but existing methods primarily target general text generation rather than structured data extraction with strict schema adherence. LLM agents have emerged as critical frameworks for autonomous tool manipulation and API interaction (Luo et al., 2025). The ReAct framework demonstrates synergized reasoning and acting through interleaved traces and actions (Yao et al., 2023), while systems like RestGPT enable direct RESTful API interaction (Luo et al., 2025). Despite advances, agents face reliability challenges including high inference latency, output uncertainty, and security vulnerabilities (Liang and Tong, 2025), with "Agentic ROI" highlighting fundamental trade-offs between value and operational costs (Liu et al., 2025).

While substantial progress exists in LLM-based extraction, JSON schema optimization, and self-correction mechanisms, existing work treats these as independent problems. Current schema optimization focuses on syntactic validity rather than extraction performance, and self-correction mechanisms aren't designed for structured data adherence. Most critically, existing approaches assume static, human-designed schemas rather than exploring schema optimization for LLM consumption in Software 3.0 applications where LLM agents are primary consumers. Our work addresses this gap through an integrated approach combining automated schema optimization specifically for LLM agents with reflection-based guardrails tailored for structured extraction tasks.

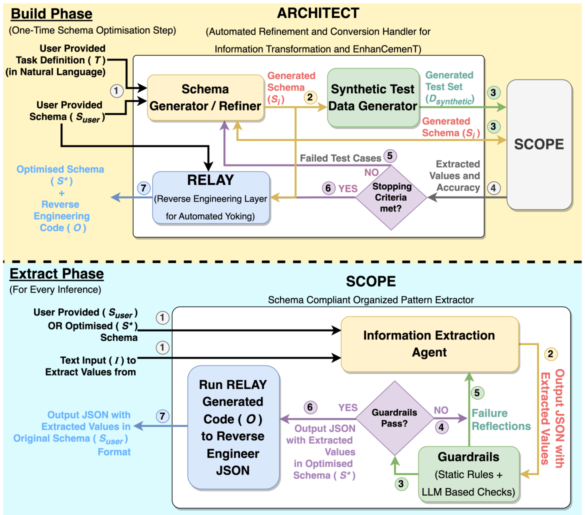

Figure 2: Detailed architecture of PARSE's two main components: ARCHITECT and SCOPE. The Build Phase (top) shows ARCHITECT's workflow for schema optimization through iterative refinement using synthetic test data generation and validation. The Extract Phase (bottom) illustrates SCOPE's extraction pipeline with built-in guardrails and reflection mechanisms for ensuring reliable structured output. Numbers indicate the sequential flow of operations.

3 PARSE: Parameter Automated Refinement and Schema Extraction

PARSE addresses the challenge of reliable structured information extraction from unstructured text through a two-phase approach. Our key insight is that JSON schemas themselves represent natural language contracts that can be optimized for LLM consumption, creating a virtuous cycle where better schemas lead to improved extraction performance. PARSE consists of two primary components operating in distinct phases: (1) **Build Phase:** AR-

CHITECT optimizes JSON schemas, as a one time process, specifically for LLM agent consumption while maintaining backward compatibility through RELAY, (2) **Extract Phase:** SCOPE performs reliable information extraction using reflection-based guardrails over the optimised schemas created and maps it back to original schema through RELAY transformation code generated in Build Phase. Refer Figure 2 for detailed flow.

3.1 ARCHITECT: Automated Refinement and Conversion Handler for Information Transformation and EnhanCemenT

Problem Formulation: Let S_{user} represent a user-provided JSON schema and T represent a natural language task description. Current schemas are designed for human developers and static systems, leading to suboptimal performance when consumed by LLM agents. We formalize the schema optimization problem as:

$$S^* = \arg\max_{S'} \mathcal{P}(S', D_{\text{synthetic}}, M_{\text{llm}})$$

where $\mathcal{P}(S', D_{\text{synthetic}}, M_{\text{llm}})$ represents the extraction performance of LLM model M_{llm} using schema S' on synthetic validation data $D_{\text{synthetic}}$.

Schema Generation and Refinement: ARCHITECT begins by analyzing the original schema structure and generating an optimized version that enhances LLM comprehension through improved descriptions, structural clarity, and additional validation rules. The schema optimization process follows an iterative refinement procedure:

$$S_{i+1} = \text{Refine}(S_i, \mathcal{E}(S_i, D_{\text{synthetic}}), T)$$

where S_i is the schema at iteration i, $\mathcal{E}(\cdot)$ computes extraction errors on test data $D_{\text{synthetic}}$ and could be done via SCOPE or any other information extraction agent, and $\text{Refine}(\cdot)$ represents the LLM-based schema improvement function. The optimization process terminates when extraction accuracy reaches a threshold τ or maximum iterations K are reached.

Synthetic Test Data Generation: To validate schema improvements, ARCHITECT generates synthetic test cases using both the current schema and a seed dataset of real examples to ensure the synthetic data reflects realistic extraction scenarios:

$$D_{\text{synthetic}} = \text{Generate}(S_i, T, D_{\text{seed}}, n_{\text{samples}})$$

where S_i is the current schema iteration, T is the task description, n_{samples} controls the diversity of generated validation examples, and The seed dataset D_{seed} provides crucial context for generating realistic adversarial examples: $D_{\text{seed}} = \{(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)\}$, where each x_j is unstructured text and y_j is the corresponding ground truth extraction following the schema structure.

RELAY (Reverse Engineering Layer for Automated Yoking): To maintain compatibility with existing systems, ARCHITECT has a sub-module called RELAY which generates executable Python code that transforms outputs from the optimized schema S^* back to the original schema format S_{user}. The reverse mapping function ensures semantic preservation: RELAY : $\mathcal{O}(S^*) \to \mathcal{O}(S_{user})$, where $\mathcal{O}(S)$ represents the output space of schema S.

RELAY automatically generates and validates this transformation code during the Build Phase, ensuring semantic preservation through: (1) Automated Python code generation for schema mapping, (2) Sample data pair generation for testing transformations, (3) Iterative refinement until semantic preservation is verified. This ensures that downstream systems can continue using original schema formats while benefiting from ARCHITECT's optimizations.

3.2 SCOPE: Schema Compliant Organized Pattern Extractor

SCOPE implements a systematic validation framework with constrained decoding and static rule checking, enabling agents to self-correct through structured reflection. The extraction process with guardrails follows:

$$\hat{y} = \text{Extract}(x, S^*, G_{\text{static}})$$

where x is input text, S^* is the optimized schema, G_{static} represents static guardrails and validation rules in S^*.

Multi-Stage Validation Process: SCOPE implements a systematic three-stage validation that operate as follows: (1) Missing Attribute Check: Verifies that all required fields specified in the schema are present in the extracted output (2) Grounding Verification: Confirms that extracted values can be found in the original input text (3) Rule Compliance Check: Validates that extracted values adhere to schema constraints such as patterns, length limits, enumerated values, date format validations, and more. Each validation stage returns a status indicator (pass or fail) along with specific error details. When validation fails, SCOPE generates structured reflections that guide the extraction agent toward correct outputs.

4 Discussion

While PARSE focuses on structured extraction from unstructured text for LLM agent systems, the framework's principles extend to related information extraction domains. Named Entity Recognition (NER) (Li et al., 2020; Shrimal et al., 2022), for instance, can be formulated as schema-based extraction where entity types, spans, and attributes are defined in a structured schema. Multi-agent orchestration frameworks for task automation (Shrimal et al., 2024) face similar challenges in ensuring reliable structured outputs, particularly for tool calling where agents must extract precise parameters from natural language to invoke APIs correctly. In such systems, PARSE's components can enhance execution accuracy: ARCHITECT can optimize tool parameter schemas for clearer LLM comprehension, while SCOPE's reflection-based guardrails can validate parameter extraction and recover from common errors like parameter hallucination or incorrect formatting—challenges explicitly identified in multi-agent systems. This positions PARSE as complementary to agent orchestration frameworks, providing the schema optimization and validation layer needed for reliable tool invocation.

The core insight of PARSE—that schemas themselves can be optimized for LLM consumption rather than treated as static contracts—applies broadly across structured extraction tasks. Whether extracting named entities, tool parameters in agent systems, slot values in dialogue systems, or complex nested structures from web data, the challenge remains: how do we design and refine the structural contracts that guide LLM extraction? PARSE's automated schema optimization through ARCHITECT and systematic validation through SCOPE provide a general framework applicable to these diverse scenarios.

5 Experiments

We evaluate PARSE through comprehensive experiments designed to answer three key questions: (1) Does ARCHITECT effectively optimize JSON schemas for LLM consumption? (2) Does SCOPE's reflection-based guardrail system improve extraction reliability? (3) How do these components work together to enable reliable structured information extraction for Software 3.0 applications?

5.1 Experimental Setup

Datasets: We evaluate on three complementary datasets that test different aspects of structured extraction: (1) **Retail-Conv**: A curated internal dataset of 6 diverse retail conversation schemas with 40 samples each (240 total), designed to test extraction from natural business communications. This dataset captures realistic scenarios where customers describe product issues, requests, and updates in conversational language; (2) **Schema-Guided Dialogue (SGD)** (Rastogi et al., 2020): A large-scale dataset of 20,000 annotated task-oriented conversations across 20 domains including banking, events, media, and travel. (3) **Structured Web Data Extraction (SWDE)** (Hao, 2011): A standard benchmark containing 1,600 test samples across 8 verticals (200 test samples per vertical). This dataset tests extraction from semi-structured web content with varied formatting.

Baseline: We implement an extraction agent using best prompting practices and constraint decoding with the original user-provided schemas as our baseline. This baseline uses clear instructions, few-shot examples, and standard JSON formatting requests without any guardrails or schema optimization. We evaluate across five LLM variants to ensure robustness: Claude 3.5/3.7 Sonnet, Claude 3.5 Haiku, Llama 4-Maverick, and DeepSeek-R1-671B, spanning diverse architectures and capabilities.

Metrics: We measure field-level accuracy where all required schema fields must be correctly extracted and properly formatted. This strict metric reflects real-world requirements where partial extraction often provides little value. We also track end-to-end extraction time including all reflection iterations and guardrail checks, providing insight into the practical trade-offs between accuracy and speed. For ARCHITECT evaluation, we analyze schema modification patterns and track accuracy improvements across optimization iterations.

Model	Retail-Conv				SGD Data				SWDE Data			
	Original Schema		ARCHITECT Schema		Original Schema		ARCHITECT Schema		Original Schema		ARCHITECT Schema	
	Baseline Agent	SCOPE	Baseline Agent	SCOPE	Baseline Agent	SCOPE	Baseline Agent	SCOPE	Baseline Agent	SCOPE	Baseline Agent	SCOPE
Claude 3.7 Sonnet	75	88.6	77.1	**90.01**	92.3	94.2	93.1	**94.8**	24.5	89.2	31.38	**93.14**
Claude 3.5 Sonnet V2	75.5	88.6	79.8	**91.7**	90.9	93.9	92.66	**94.1**	25.7	90.8	33.51	**92.99**
Claude 3.5 Haiku	72.5	85	76.4	**87.3**	88.16	91.7	89.09	**93.3**	25	88.3	32.81	**91.67**
Llama 4 Maverick	75.5	84	78.8	**88.4**	83.22	88.9	84.18	**91.44**	21.3	85.1	28.99	**88.32**
DeepSeek-R1	76	91	81.2	**93.7**	87.15	91.22	89.25	**92.01**	19.3	82.7	25.13	**86.16**

Table 1: Accuracy comparison for Baseline LLMs and SCOPE with and without ARCHITECT schemas on Retail-Conv, SGD and SWDE datasets. **Bold** indicates best result per model per dataset, underline indicates second best.

	Retail-Conv	
	ARCHITECT-Claude	ARCHITECT-LLAMA
SCOPE-Claude	**93.7**	91.11
SCOPE- LLAMA	89.1	88.4

Table 2: Comparison of performance when Optimising schema using model X in ARCHITECT then applying the schema in SCOPE with model Y. Claude version used is Sonnet 3.5 V2.

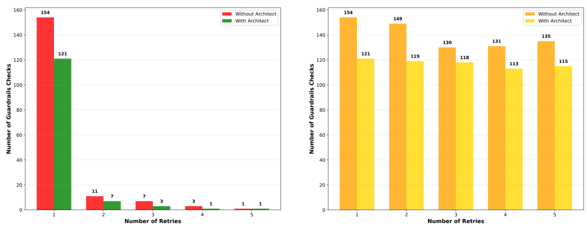

Figure 3: Error recovery with and without reflection and with and without ARCHITECT schemas

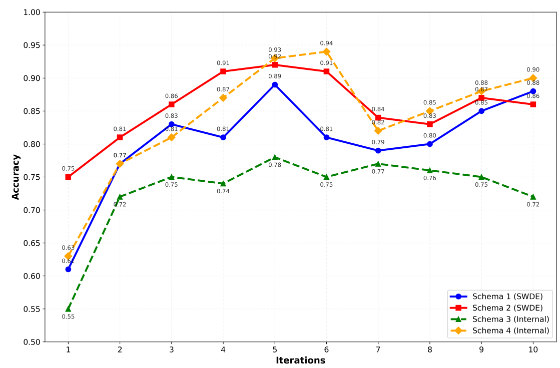

Figure 4: Performance of schemas on validation sets over ARCHITECT optimisation iterations

5.2 Main Results

5.3 Combined System Performance

Performance and Reliability: Table 1 demonstrates that PARSE achieves substantial improvements across all datasets, with the combined ARCHITECT + SCOPE system showing the strongest performance. The improvements remain consistent across different LLMs suggesting our approach makes sophisticated extraction more accessible across model scales.

5.4 ARCHITECT Analysis

Iterative Improvement: ARCHITECT's schema optimization shows consistent improvement over iterations, with most gains achieved within the first 5-6 iterations before dipping again. This suggests the optimization process efficiently identifies and addresses the most critical schema limitations although doing it for larger durations can lead to overfitting.

Schema Modification Patterns: Analysis of ARCHITECT's schema changes (Figure 5) reveals consistent patterns across datasets with: (1) Entity Description Enhancement (34%): Adding detailed descriptions and contextual information (2) Structural Reorganization (55%): Flattening nested structures and clarifying field relationships (3) Validation Rule Addition (0.08%): Implementing format constraints and enumerated values (4) Pattern rules additions (3%): Enforcing strict regex pattern rules to constrain outputs

Generalization Across Models: Table 2 suggests that schemas optimized using one LLM (e.g., Claude 3.5 Sonnet) maintain their performance benefits when applied with different models (e.g., Llama 4), indicating that ARCHITECT identifies model-agnostic schema improvements rather than model-specific optimizations.

5.5 SCOPE Guardrail Effectiveness

Reflection-Based Improvement: SCOPE's multi-stage validation shows clear benefits over simple retry mechanisms. On average, SCOPE reduces extraction errors by 92% within the first retry compared to baseline approach that simply re-prompt on failures (Figure 3). Moreover, ARCHITECT optimised schema's are less prone to have re-tries as shown in the same figure.

Cost-Accuracy Trade-offs: While SCOPE in-

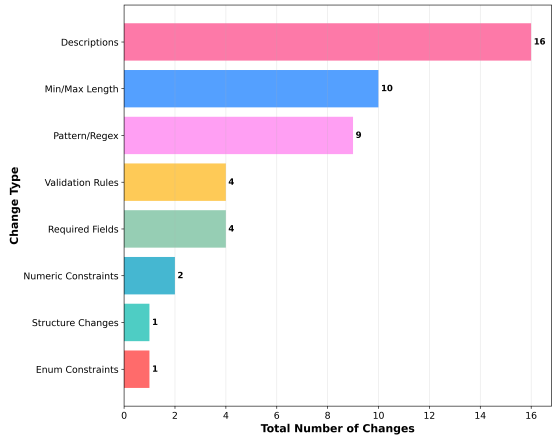

Figure 5: Description of how frequently different types of changes are done through ARCHITECT to optimise a schema

Model	SWDE Data			
	Latency with Original Schema		Latency with ARCHITECT Schema	
	Baseline Agent	SCOPE	Baseline Agent	SCOPE
Claude 3.7 Sonnet	9.08	25.58	8.80	19.06
Claude 3.5 Sonnet V2	9.00	24.46	8.83	19.54
Claude 3.5 Haiku	8.99	22.26	8.68	14.99
Llama 4 Maverick	0.98	2.30	0.93	1.97
DeepSeek-R1	3.01	11.99	2.91	10.01

Table 3: Impact on latency for extraction with using SCOPE and ARCHITECT

creases latency by an average of 10.16 due to reflection iterations, the corresponding accuracy gain of +64.7% on SWDE demonstrates substantial practical value. Importantly, using ARCHITECT-optimized schemas reduces this latency penalty by an average of 4.05s as fewer reflection rounds are needed (Table 3).

5.6 Qualitative Analysis

ARCHITECT consistently transforms ambiguous field descriptions into precise, context-rich specifications. For example, a generic "price" field becomes "newPriceValue: The specific numerical price value for the product update, excluding currency symbols" with additional validation rules for format and range constraints.

SCOPE's reflection mechanism effectively identifies and corrects common extraction errors. In cases where the baseline agent confused similar entities (e.g., "old price" vs. "new price"), SCOPE's grounding verification caught the error and guided correction through structured reflection.

SWDE Dataset Complexity: The substantial improvements on SWDE (up to 64.7%) stem from

Component	Content
Input HTML	`<title>2010 Subaru Legacy 2.5 i 4dr Sedan</title>` (truncated)
Baseline Schema	`{"model": {"type": "string"}}`
PARSE Schema	`{"model": {"description": "Full model name including year, make and model", "pattern": "^(19[5-9][0-9]|20[0-2][0-9]) [A-Za-z0-9 -+]+$"...}}`
Expected Result	"2010 Subaru Legacy"
Baseline Result	"2010 Subaru Legacy 2.5 i 4dr Sedan"
PARSE Result	"2010 Subaru Legacy"

Table 4: SWDE extraction example demonstrating why PARSE achieves substantial improvements on HTML-structured data. The baseline extracts excessive detail due to insufficient schema guidance, while PARSE's optimized schema with pattern constraints and SCOPE's validation ensure precise extraction.

the dataset's HTML structure requiring precise extraction with specific formatting. Table 4 illustrates a representative case where the baseline struggles with HTML noise while PARSE's optimized schema and guardrails enable accurate extraction. The baseline schema's generic `"model": {"type": "string"}` provides insufficient guidance for parsing complex HTML, while ARCHITECT's optimized schema adds detailed descriptions and pattern constraints that help the LLM focus on relevant content. SCOPE's grounding verification then ensures extracted values match the source text, preventing hallucinations common in HTML extraction tasks.

Refer to Appendix for some qualitative samples on schema optimisations done by ARCHITECT.

Conclusion

We introduced PARSE, a comprehensive framework that addresses structured information extraction challenges through schema optimization and reflection-based guardrails. ARCHITECT automatically refines JSON schemas for LLM consumption while SCOPE ensures reliable extraction through multi-stage validation. Our evaluation across three datasets demonstrates substantial improvements, with up to 64.7% accuracy gains on SWDE and 92% error reduction within first retry, establishing PARSE as an effective solution for reliable LLM agent systems in Software 3.0 applications.

Limitations

Our approach has several important limitations that future work should address. The iterative refinement process in ARCHITECT can be computationally expensive for complex schemas with many attributes. Each refinement iteration requires synthetic data generation, extraction evaluation, and failure analysis, creating potential scalability bottlenecks for large-scale deployments.

ARCHITECT's optimization quality depends heavily on the availability and representativeness of seed datasets. For entirely new domains or rapidly evolving schema requirements, obtaining sufficient high-quality seed data can be challenging. Furthermore, our approach assumes relatively static schema structures that can be optimized offline, which can be challenging where schemas evolve continuously.

Several promising directions emerge from this work. Multi-modal extension represents a natural evolution, where schemas could be optimized for extraction from both textual and visual content. The same principle of treating schemas as optimizable interfaces applies to multi-modal scenarios, but would require extending the validation framework to handle cross-modal grounding.

References

Bhavik Agarwal, Ishan Joshi, and Viktoria Rojkova. 2025. Think inside the json: Reinforcement strategy for strict llm schema adherence.

Shumin Deng, Yubo Ma, Ningyu Zhang, Yixin Cao, and Bryan Hooi. 2022. Information extraction in low-resource scenarios: Survey and perspective.

Kuofeng Gao, Huanqia Cai, Qingyao Shuai, Dihong Gong, and Zhifeng Li. 2024. Embedding self-correction as an inherent ability in large language models for enhanced mathematical reasoning.

Ridong Han, Chaohao Yang, Tao Peng, Prayag Tiwari, Xiang Wan, Lu Liu, and Benyou Wang. 2023. An empirical study on information extraction using large language models.

Qiang Hao. 2011. Structured web data extraction dataset (swde).

Adarsh Kumar, Hwiyoon Kim, Jawahar Sai Nathani, and Neil Roy. 2025. Improving the reliability of llms: Combining cot, rag, self-consistency, and self-verification.

Xiaoya Li, Jingrong Feng, Yuxian Meng, Qinghong Han, Fei Wu, and Jiwei Li. 2020. A unified MRC framework for named entity recognition. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 5849–5859, Online. Association for Computational Linguistics.

Guannan Liang and Qianqian Tong. 2025. Llm-powered ai agent systems and their applications in industry.

Weiwen Liu, Jiarui Qin, Xu Huang, Xingshan Zeng, Yunjia Xi, Jianghao Lin, Chuhan Wu, Yasheng Wang, Lifeng Shang, Ruiming Tang, Defu Lian, Yong Yu, and Weinan Zhang. 2025. The real barrier to llm agent usability is agentic roi.

Yaxi Lu, Haolun Li, Xin Cong, Zhong Zhang, Yesai Wu, Yankai Lin, Zhiyuan Liu, Fangming Liu, and Maosong Sun. 2025a. Learning to generate structured output with schema reinforcement learning.

Yaxi Lu, Haolun Li, Xin Cong, Zhong Zhang, Yesai Wu, Yankai Lin, Zhiyuan Liu, Fangming Liu, and Maosong Sun. 2025b. Learning to generate structured output with schema reinforcement learning.

Junyu Luo, Weizhi Zhang, Ye Yuan, Yusheng Zhao, Junwei Yang, Yiyang Gu, Bohan Wu, Binqi Chen, Ziyue Qiao, Qingqing Long, Rongcheng Tu, Xiao Luo, Wei Ju, Zhiping Xiao, Yifan Wang, Meng Xiao, Chenwu Liu, Jingyang Yuan, Shichang Zhang, Yiqiao Jin, Fan Zhang, Xian Wu, Hanqing Zhao, Dacheng Tao, Philip S. Yu, and Ming Zhang. 2025. Large language model agent: A survey on methodology, applications and challenges.

Abhinav Rastogi, Xiaoxue Zang, Srinivas Sunkara, Raghav Gupta, and Pranav Khaitan. 2020. Towards scalable multi-domain conversational agents: The schema-guided dialogue dataset. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*, pages 8689–8696. AAAI Press.

Anubhav Shrimal, Avi Jain, Kartik Mehta, and Promod Yenigalla. 2022. NER-MQMRC: Formulating named entity recognition as multi question machine reading comprehension. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Industry Track*, pages 230–238, Hybrid: Seattle, Washington + Online. Association for Computational Linguistics.

Anubhav Shrimal, Stanley Kanagaraj, Kriti Biswas, Swarnalatha Raghuraman, Anish Nediyanchath, Yi Zhang, and Promod Yenigalla. 2024. MARCO: Multi-agent real-time chat orchestration. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing: Industry Track*, pages 1381–1392, Miami, Florida, US. Association for Computational Linguistics.

Haoyu Wang, Christopher M. Poskitt, and Jun Sun. 2025. Agentspec: Customizable runtime enforcement for safe and reliable llm agents.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Xin Zhang, Tianjie Ju, Huijia Liang, Ying Fu, and Qin Zhang. 2024. General llms as instructors for domain-specific llms: A sequential fusion method to integrate extraction and editing.

Zikang Zhang, Wangjie You, Tianci Wu, Xinrui Wang, Juntao Li, and Min Zhang. 2025. A survey of generative information extraction. In *Proceedings of the 31st International Conference on Computational Linguistics*, pages 4840–4870, Abu Dhabi, UAE. Association for Computational Linguistics.

A Appendix

A.1 Qualitative Samples of SCOPE predictions

Table 5 gives an overview of how SCOPE performs against Baselines. Notice that SCOPE is able handle longer contexts better while

A.2 Qualitative Samples of ARCHITECT's optimisation

Table 6 shows the qualitative example of how ARCHITECT's optimisation process updates the JSON schema across 5 iterations for various attributes in SWDE dataset.

A.3 RELAY: Ensuring Backward Compatibility

RELAY (Reverse Engineering Layer for Automated Yoking) maintains compatibility between ARCHITECT-optimized schemas and original user schemas through automated code generation. When ARCHITECT transforms a schema, RELAY simultaneously creates Python functions that map outputs between formats.

Example Transformation:

- Original schema: `{"price": {"type": "string"}}`
- Optimized schema: `{"currency_symbol": {"type": "string", "pattern": "[$€¥£]"}, "price_value": {"type": "number", "minimum": 0}}`
- Generated RELAY code:

```python
def reverse_transform(optimized_data):
    return {
        "price": f"{optimized_data['currency_symbol']}"
                 f"{optimized_data['price_value']}"
    }
```

Validation Process: RELAY generates sample data pairs and tests transformation code iteratively. If semantic preservation errors are detected, it refines the transformation logic until accuracy is achieved. Production deployments show consistent transformation accuracy without conversion errors.

Dataset	Input Schema	Context	SCOPE	Baseline

SGD	Restaurant_2	**[USER]**: I want to have a table booked for me at a restaurant, because I want to have a meal.
		[ASSISTANT]: Can you tell me the restaurant you are interested to have a table booked at?
		[USER]: Can you check if you can make a booking of a table at Castle Rock?
		[ASSISTANT]: Can you tell me the time that you want to have the table booked for you?
		[USER]: I prefer to have the booking be made for 12:00.
		[ASSISTANT]: Is there a specific location you are looking for?
		[USER]: I am interested in restaurants which are located in Livermore.
		[ASSISTANT]: Can you please confirm that you want to book a table for 2 people at Castle Rock Restaurant in Livermore, and that you want the table to be booked for you for 12 pm on March 1st?
		[USER]: That is not right. Please make a booking at Mcdonald's.
		. . .
			{
			"ReserveRestaurant": [
			{"location":
			"Livermore"},
			{"restaurant_name":
			"Mcdonald's"},
			{"time": "12 pm"}
			]
			}
				{
				"ReserveRestaurant": [
				{
				"restaurant_name": null,
				"location": "Livermore",
				"time": "12:00",
				"number_of_seats": null,
				"date": null
				}
				]
				}

SGD	Payment_1,
	Trains_1,
	Weather_1
		[USER]: I am looking to make a payment.
		[ASSISTANT]: What is the name of the contact and from what account?
		[USER]: I'd like it from my checking account and the name is Mahmoud.
		[ASSISTANT]: How much will it be?
		[USER]: The amount is $180.
		[ASSISTANT]: Please confirm you're making a transaction for $180 from your debit card with Mahmoud and it will be visible to the public.
		[USER]: That is correct.
		[ASSISTANT]: Your payment has been confirmed and the receiver should see the money soon.
		[USER]: Can you help me find a train for the 12th of March?
		[ASSISTANT]: From which city are you leaving from?
		[USER]: I'm planning to leave from Sacramento, CA.
		[ASSISTANT]: Where are you going?
		[USER]: I'm trying to get to San Fran.
		[ASSISTANT]: I found a train leaving at 7:30 am for $47.
		[USER]: Which station does this train leave from?
		[ASSISTANT]: It is leaving from Sacramento Valley Station.
		[USER]: That works. How will the weather be in San Fran on the 14th?
		. . .
			{
			"GetWeather": [{"city": "San Fran", "date": "the 14th"}],
			"FindTrains": [
			{"date_of_journey":
			"12th of March"},
			{"from":
			"Sacramento, CA"},
			{"journey_start_time":
			"7:30 am"},
			{"to": "San Fran"}
			]
			}
				{
				"GetWeather": [{"city": "San Francisco", "date": "March 14th"}]
				}

Retail-Conv	Sales Data	**[USER]**: Get sales data for "PRODUCT123" in Italy **[ASSISTANT]**: I need a specific timeframe to fetch the sales data. Could you please specify when you want to see the sales for? For example: last week, MTD, or a specific date range?	{NULL}	{"sales": [{"asin": "PRODUCT123", "time-frame": "MTD", "market-place": "IT"}] }
Retail-Conv	Price update	**[USER]**: For PRODUCT234, the old price is 29.99 USD and I want to maybe change it to either 39.99 or 49.99 USD	{NULL}	{"list_of_asin_price_vendor": [{ "asin": "PRODUCT234", "new_list_price_value": 49.99, "new_list_price_currency": "USD"}]}

Table 5: Qualitative comparison between SCOPE and baselines predictions (both with ARCHITECT) on Claude 3.5 Sonnet V2.

<table>
<tr><th>Iteration</th><th>automobile_model_year</th><th>price</th><th>engine_type</th><th>fuel_economy</th></tr>
<tr><td>1</td><td>{
"type": "array",
"description": "List of automobile models with their corresponding years",
"items": {
"type": "object",
"properties": {
"model": {
"type": "string"
}
}}}</td><td>{
"type": "array",
"description": "List of price information",
"items": {
"type": "object",
"properties": {
"price": {
"type": "string",
"description": "Currency symbol + numeric value, eg: $19,995"
}}}}</td><td>{
"type": "array",
"description": "List of engine types",
"items": {
"type": "object",
"properties": {
"engine": {
"type": "string"
}}}}</td><td>{
"type": "array",
"items": {
"type": "object",
"properties": {
"fuel": {
"type": "string"
}}}}</td></tr>
<tr><td>3</td><td>{
"type": "array",
"description":
"List of automobile models with full year, make and model",
"items": {
"type": "object",
"required":
["model"],
"properties": {
"model": {
"type": "string",
"pattern":
"^[0-9]{4}[A-Za-z0-9 -]+",
"minLength": 10
}}}}</td><td>{
"type": "array",
"description": "List of price information",
"items": {
"type": "object",
"required":
["price"],
"properties": {
"price": {
"type": "string",
"pattern":
"^[$€¥][0-9,.]+(?:k|K)?$",
"minLength": 2
}}}}</td><td>{
"type": "array",
"description": "List of engine/powertrain configurations",
"items": {
"type": "object",
"required":
["engine"],
"properties": {
"engine": {
"type": "string",
"pattern":
"^[0-9.]+L(?:\s+[A-Za-z0-9-\s]+)*$",
"minLength": 3
}}}}</td><td>{
"type": "array",
"description": "List of fuel economy ratings",
"items": {
"type": "object",
"required":
["fuel"],
"properties": {
"fuel": {
"type": "string",
"minLength": 5,
"condition": "Must include numeric value with unit (MPG, MPGe)"
}}}}</td></tr>
<tr><td>5</td><td>{
"type": "array",
"description":
"Full model name including year, make, model, hyphens and plus signs",
"items": {
"type": "object",
"required":
["model"],
"properties": {
"model": {
"type": "string",
"pattern":
"^[0-9.]+L(?:\s+[A-Za-z0-9-\s\+]+)*$",
"minLength": 10
}}}}</td><td>{
"type": "array",
"required":
["price"],
"items": {
"price": {
"type": "string",
"pattern":
"^[$€¥£][0-9,.]+(?:k|K)?$",
"minLength": 2
}}}</td><td>{
"type": "array",
"required":
["engine"],
"items": {
"engine": {
"type": "string",
"pattern":
"^[0-9.]+L(?:\s+[A-Za-z0-9-\s]+)*$",
"minLength": 3
}
}
}</td><td>{
"type": "array",
"required":
["fuel"],
"items": {
"fuel": {
"type": "string",
"minLength": 5,
"condition": "Must include numeric value with unit and flexible separators"
}
}
}</td></tr>
</table>

Table 6: Evolution of SWDE Auto schema over iterations. Green highlights new required fields, Blue highlights pattern changes, Yellow highlights description updates.