Anka: A Domain-Specific Language for Reliable LLM Code Generation

Anonymous ACL submission

Abstract

Large Language Models (LLMs) demonstrate remarkable code generation capabilities yet exhibit systematic errors on complex, multi-step programming tasks. We hypothesize these errors stem from the flexibility of general-purpose languages, which permits multiple valid approaches and requires implicit state management. To test this hypothesis, we introduce **Anka**, a domain-specific language (DSL) for data transformation pipelines designed with explicit, constrained syntax that reduces ambiguity in code generation.

Despite zero prior training exposure, Claude 3.5 Haiku achieves 99.9% parse success and 95.8% overall task accuracy across 100 benchmark problems. Critically, Anka demonstrates a **40 percentage point accuracy advantage** over Python on multi-step pipeline tasks (100% vs. 60%), where Python's flexible syntax leads to frequent errors in operation sequencing and variable management. Cross-model validation with GPT-4o-mini confirms this advantage (+26.7 percentage points on multi-step tasks).

Our results demonstrate that: (1) LLMs can learn novel DSLs entirely from in-context prompts; (2) constrained syntax significantly reduces errors on complex tasks; and (3) purpose-built DSLs can outperform general-purpose languages despite extensive LLM training on the latter. We release the complete implementation, benchmark suite, and evaluation framework.[1]

1 Introduction

Large Language Models (LLMs) have transformed software development through their ability to generate code from natural language descriptions (Chen et al., 2021; Nijkamp et al., 2023; Li et al., 2023). Modern code-generation systems power developer tools used by millions (GitHub, 2022). However, despite impressive performance on isolated tasks, LLMs exhibit systematic failures when generating complex, multi-step code (Austin et al., 2021; Hendrycks et al., 2021).

These failures are not random. Prior work has identified consistent error patterns: incorrect variable scoping, off-by-one errors, and state management bugs in sequential operations (Pearce et al., 2023; Jesse et al., 2023). We observe that many of these errors share a common root cause: the *flexibility* of general-purpose programming languages. When multiple syntactically valid approaches exist for expressing the same computation, LLMs must implicitly choose among them, introducing opportunities for inconsistency and error accumulation.

This observation motivates a counterintuitive hypothesis: **constraining** the target language may **improve** LLM code generation accuracy. Rather than allowing the model to choose from Python's many valid patterns, we can design a language where each operation has exactly one canonical form.

To test this hypothesis, we introduce **Anka**, a domain-specific language for data transformation pipelines. Anka enforces explicit syntax through four design principles:

- **One canonical form per operation**: FILTER always uses WHERE...INTO syntax
- **Named intermediate results**: Every operation produces a named output via INTO clauses
- **Explicit step structure**: Sequential operations are organized into named STEP blocks
- **Verbose keywords**: FILTER, MAP, AGGREGATE rather than operators

We evaluate Anka against Python on 100 data transformation tasks spanning eight categories. Our key findings are:

- **Novel DSL acquisition:** Despite zero training exposure, Claude 3.5 Haiku achieves

[1] Anonymous repository (link available upon acceptance).

99.9% parse success, demonstrating that LLMs can learn new programming languages from prompts alone.

- **Multi-step advantage:** Anka achieves 100% accuracy on multi-step pipeline tasks compared to 60% for Python—a 40 percentage point improvement, confirmed across models (GPT-4o-mini: +26.7pp).

- **Overall improvement:** Anka achieves 95.8% overall accuracy compared to 91.2% for Python, despite Python's substantial training data advantage.

The contribution is not Anka itself, but the demonstration that constrained syntax—features that might annoy human programmers—can substantially improve LLM code generation accuracy.

2 Related Work

LLM Code Generation. Codex (Chen et al., 2021) demonstrated that language models could solve programming challenges with human-level competence. Subsequent work scaled these approaches: CodeGen (Nijkamp et al., 2023) introduced multi-turn synthesis, and StarCoder (Li et al., 2023) achieved state-of-the-art performance. Systematic evaluations on HumanEval (Chen et al., 2021), MBPP (Austin et al., 2021), and APPS (Hendrycks et al., 2021) reveal that accuracy degrades substantially as task complexity increases. Our work demonstrates that language design, not just model scale, can address complexity-related failures.

Domain-Specific Languages. DSLs trade generality for expressiveness within narrow domains (Fowler, 2010; Mernik et al., 2005). FlashFill (Gulwani, 2011) uses a DSL for string transformations, and DreamCoder (Ellis et al., 2021) learns DSL primitives during synthesis. Our work differs in designing a DSL specifically for *LLM* generation rather than human use, prioritizing features that reduce LLM errors.

Constrained Generation. Chain-of-thought prompting (Wei et al., 2022) and self-consistency (Wang et al., 2023) improve LLM performance without model modification. Grammar-constrained decoding (Scholak et al., 2021; Poesia et al., 2022) ensures syntactic validity by masking invalid tokens. Our approach is complementary: rather than constraining the *decoding process*, we constrain the *target language* itself.

3 The Anka Language

Anka is a DSL for data transformation pipelines designed to reduce LLM code generation errors. Each design principle addresses specific error patterns observed in LLM-generated Python code.

3.1 Design Principles

Principle 1: One Canonical Form. In Python, filtering can be expressed as `df[df.x > 5]`, `df.query("x > 5")`, or `df.loc[df.x > 5]`. This flexibility forces LLMs to choose among equivalent options. In Anka, filtering has exactly one form:

```
FILTER source WHERE condition
INTO target
```

Principle 2: Named Intermediate Results. Python developers may reuse variable names or chain operations, causing LLM errors when the model loses track of state. Anka requires explicit `INTO` clauses naming each intermediate result.

Principle 3: Explicit Step Structure. Anka organizes operations into named `STEP` blocks, providing "scaffolding" that guides sequential generation.

Principle 4: Verbose Keywords. Keywords like `FILTER`, `MAP`, and `AGGREGATE` leverage LLM language capabilities better than operators.

3.2 Syntax Overview

A complete Anka pipeline consists of a name, typed inputs, steps, and output:

```
PIPELINE transform_sales:
INPUT orders: TABLE[order_id: INT,
customer: STRING, amount: DECIMAL]
STEP filter_large:
FILTER orders WHERE amount > 1000
INTO large_orders
STEP summarize:
AGGREGATE large_orders
GROUP_BY customer
COMPUTE SUM(amount) AS total
INTO summary
OUTPUT summary
```

Anka supports 18 data operations: selection (FILTER, SELECT, DISTINCT), transformation (MAP, RENAME, DROP), aggregation (AGGREGATE with COUNT, SUM, AVG, MIN, MAX), ordering (SORT, LIMIT, SKIP), and combination (JOIN, LEFT_JOIN, UNION).

Feature	Error Prevented	Mechanism
Canonical forms	Inconsistent syntax	Eliminates choices
INTO clauses	Variable shadowing	Explicit naming
STEP structure	Ordering errors	Visual scaffolding
Verbose keywords	Operator confusion	Leverages language

Table 1: Connection between Anka design features and LLM error prevention.

Category	N	Description
filter	10	Single and compound filtering
map	10	Column computation
aggregate	10	Grouping and aggregation
strings	10	String manipulation
multi_step	10	3–5 sequential operations
finance	20	Domain-specific calculations
hard	10	Complex logic with edge cases
adversarial	20	Tasks triggering common errors

Table 2: Benchmark categories and task distribution.

3.3 Implementation

Anka is implemented in Python using Lark for parsing, comprising approximately 6,400 lines including: a formal grammar (98 production rules), 68 AST node types as immutable dataclasses with source location tracking, a tree-walking interpreter, control flow constructs (IF/ELSE, FOR_EACH, WHILE), and 322 unit tests.

4 Methodology

4.1 Benchmark Suite

We constructed 100 data transformation tasks in eight categories (Table 2). Each task specifies a natural language description, typed input schema, and test cases.

The **multi_step** category is critical: these tasks require maintaining state across 3–5 operations, precisely where we expect constrained syntax to help most.

4.2 Evaluation Protocol

For each task, we prompt the LLM to generate code in both Anka and Python:

Prompt Structure. Both prompts follow identical structure: language specification, task description, input schema, and expected output. The Anka prompt includes a concise syntax guide ($\sim$100 lines) teaching the language from scratch; the Python prompt assumes pandas knowledge.

Category	Anka	Python	Δ
multi_step	**100.0%**	60.0%	**+40.0**
finance	**90.0%**	85.0%	+5.0
aggregate	100.0%	100.0%	0.0
filter	96.7%	**100.0%**	–3.3
map	100.0%	100.0%	0.0
strings	100.0%	100.0%	0.0
hard	90.0%	**100.0%**	–10.0
Overall	**95.8%**	91.2%	**+4.6**

Table 3: Task accuracy by category (Claude 3.5 Haiku). Bold indicates better performance.

Sampling. We generate 10 samples per task per language using temperature 0.3.

Models. We evaluate Claude 3.5 Haiku (primary) with GPT-4o-mini for cross-model validation.

4.3 Metrics

We report: **Parse Success** (syntactic validity), **Execution Success** (no runtime errors), **Output Correctness** (matches expected result), and **Task Accuracy** (fraction of tasks where $\geq$50% of samples are correct—our primary metric).

5 Results

5.1 Main Results

Table 3 presents task accuracy by category.

Key Finding 1: Multi-step Advantage. The most striking result is on multi-step tasks: Anka achieves **100% accuracy** vs. Python's 60%—a 40 percentage point improvement. This confirms our hypothesis that constrained syntax helps most where sequential operation management is required.

Key Finding 2: Parse Success. Despite zero training exposure, the model achieves **99.9% parse success**, demonstrating that LLMs can learn novel languages from prompts alone.

Key Finding 3: Overall Improvement. Anka achieves 95.8% overall accuracy vs. 91.2% for Python (+4.6pp), notable given Python's training advantage.

Model	Anka	Python	Δ
Claude 3.5 Haiku	100.0%	60.0%	+40.0
GPT-4o-mini	86.7%	60.0%	+26.7

Table 4: Multi-step task accuracy across model families.

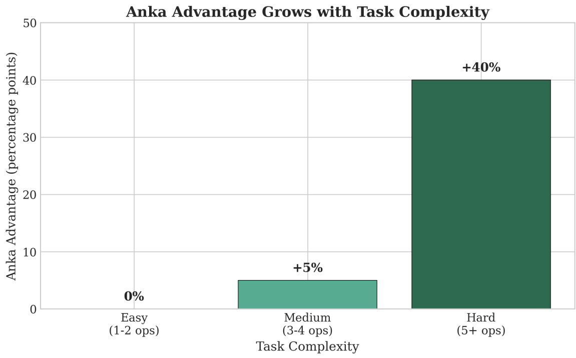

Figure 1: Anka advantage grows with task complexity. Simple tasks (1–2 ops) show no advantage; complex tasks (5+ ops) show +40% advantage.

5.2 Cross-Model Validation

GPT-4o-mini shows a +26.7pp advantage for Anka on multi-step tasks. Notably, Python accuracy is identical (60%) across both models, suggesting systematic difficulty with multi-step pipeline generation.

5.3 Error Analysis

We analyzed failing Python generations:

Variable Shadowing (42% of errors). Python generators frequently reuse variable names like `df` or `result`, losing intermediate state. Anka's `INTO` clause prevents this.

Operation Sequencing (31% of errors). Multi-step tasks require specific ordering. Python's flexibility allows reordering that changes semantics. Anka's `STEP` structure makes ordering explicit.

Chaining Confusion (27% of errors). Method chaining in pandas can obscure intermediate state. Anka's step-by-step structure prevents chaining-related errors.

5.4 Complexity Analysis

Figure 1 shows Anka's advantage as a function of task complexity:

- **Simple (1–2 ops):** 0% advantage
- **Medium (3–4 ops):** +5% advantage
- **Complex (5+ ops):** +40% advantage

6 Discussion

6.1 Why Does Constrained Syntax Help?

Reduced Decision Space. Each syntactic choice point is an opportunity for error. In a 5-step pipeline with 3 choices per step, this represents a reduction from $3^5 = 243$ possible programs to 1.

Explicit State Management. Named intermediate results via `INTO` clauses make state explicit rather than implicit in Python semantics.

Structural Scaffolding. The `STEP` structure provides a template that guides generation sequentially rather than generating a monolithic program.

6.2 When Does Anka Not Help?

Simple Tasks. With only 1–2 operations, insufficient complexity exists for errors to accumulate.

Complex Conditional Logic. "Hard" tasks requiring nested conditionals benefit from Python's flexibility.

Recommendation. Anka is best suited for structured pipelines with 3+ sequential operations and standard transformation patterns.

6.3 Implications for DSL Design

Our results suggest design principles for LLM-targeted DSLs:

1. **Canonicalization:** One way to express each operation
2. **Explicit Naming:** Require names for intermediate results
3. **Structural Templates:** Block structure guides generation
4. **Verbose Keywords:** Prefer English over symbols
5. **Type Documentation:** Include types in prompts

7 Limitations

Benchmark Scope. Our benchmark focuses on data transformation pipelines. Generalization to other programming tasks is not established.

Model Coverage. We evaluate on two models (Claude 3.5 Haiku, GPT-4o-mini). Evaluation on additional model families would strengthen confidence.

No Fine-Tuning Comparison. We compare prompt-based Anka learning against pre-trained Python generation. A comparison against an Anka-fine-tuned model would clarify the ceiling.

No User Study. We have not evaluated human developer experience with Anka.

Single Benchmark Suite. Despite diverse tasks, our benchmark may contain biases favoring Anka.

8 Conclusion

We introduced Anka, a DSL for data transformation designed to improve LLM code generation accuracy through constrained, explicit syntax. Our evaluation demonstrates:

1. **LLMs can learn novel DSLs from prompts alone.** Despite zero training exposure, Claude 3.5 Haiku achieves 99.9% parse success.

2. **Constrained syntax substantially reduces errors on complex tasks.** Anka achieves 100% accuracy on multi-step pipelines vs. 60% for Python—a 40pp improvement.

3. **Purpose-built DSLs can outperform general-purpose languages.** Despite Python's training advantage, Anka achieves higher overall accuracy.

The broader contribution is methodological: **language design** is a viable intervention for improving LLM reliability. Rather than solely improving models through scale, we can design languages that play to LLM strengths.

Future Work. Directions include: evaluation on additional model families; user studies on developer experience; production deployment evaluation; and extension to other domains.

Ethics Statement

This work presents a DSL and benchmark for evaluating LLM code generation. We do not foresee direct negative societal impacts. The benchmark tasks involve synthetic data without personally identifiable information. LLM-generated code should be reviewed before production deployment.

References

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and 1 others. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Kevin Ellis, Catherine Wong, Maxwell Nye, Mathias Sable-Meyer, Lucas Morales, Luke Hewitt, Luc Cary, Armando Solar-Lezama, and Joshua B Tenenbaum. 2021. Dreamcoder: Bootstrapping inductive program synthesis with wake-sleep library learning. *arXiv preprint arXiv:2006.08381*.

Martin Fowler. 2010. *Domain-Specific Languages*. Pearson Education.

GitHub. 2022. Github copilot: Your ai pair programmer. https://github.com/features/copilot. Accessed: 2024-01-15.

Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 317–330.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and 1 others. 2021. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*.

Kevin Jesse, Ahmed Toufique, Sebastian Elbaum, Kathryn T Stolee, and Frank Tip. 2023. Large language models and simple, stupid bugs. *arXiv preprint arXiv:2303.11455*.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, and 1 others. 2023. Starcoder: May the source be with you! *arXiv preprint arXiv:2305.06161*.

Marjan Mernik, Jan Heering, and Anthony M Sloane. 2005. When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4):316–344.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*.

Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2339–2356. IEEE.

Gabriel Poesia, Oleksandr Polozov, Vu Le, Ashish Tiwari, Gustavo Soares, Christopher Meek, and Sumit Gulwani. 2022. Synchromesh: Reliable code generation from pre-trained language models. *arXiv preprint arXiv:2201.11227*.

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 2021. Picard: Parsing incrementally for constrained auto-regressive decoding from language models. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 9895–9901.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2023. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35:24824–24837.

A Complete Grammar Specification

Anka's grammar comprises 98 production rules defined in EBNF notation using the Lark parsing library. The complete grammar is available in the supplementary materials.

Top-level Structure. A program consists of one or more pipelines, each containing input declarations, steps, and an output declaration.

Type System. Supported types include primitive types (`INT`, `STRING`, `DECIMAL`, `BOOL`, `DATE`, `DATETIME`) and composite types (`TABLE[...]`, `LIST[...]`).

Operations. Each operation follows a consistent pattern: `OPERATION source [modifiers] INTO target`. This uniformity simplifies both parsing and LLM generation.

B Extended Examples

Multi-step Pipeline.

```
PIPELINE customer_analysis:
INPUT orders: TABLE[customer: STRING,
amount: DECIMAL, date: DATE]
STEP filter_recent:
FILTER orders
WHERE date >= "2024-01-01"
INTO recent
STEP group_by_customer:
AGGREGATE recent
GROUP_BY customer
COMPUTE SUM(amount) AS total,
COUNT() AS num_orders
INTO by_customer
STEP filter_high_value:
FILTER by_customer
WHERE total > 10000
INTO high_value
STEP sort_output:
SORT high_value BY total DESC
INTO sorted
OUTPUT sorted
```

C Prompt Templates

The Anka prompt includes: (1) language introduction, (2) syntax reference (∼100 lines), (3) examples, and (4) task specification. The Python prompt assumes pandas knowledge and includes equivalent task specification.

D Benchmark Task Examples

Multi-step Task Example. *Description:* "Filter orders above $500, group by customer, calculate total spending and order count, filter to customers with more than 3 orders, sort by total descending."

Input Schema: `TABLE[order_id: INT, customer: STRING, amount: DECIMAL]`

Expected Operations: FILTER → AGGREGATE → FILTER → SORT