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

Saif Khalfan Saif Al Mazrouei

University of Wisconsin-Madison

`saif6265@outlook.com`

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities in code generation, yet they exhibit systematic errors on complex, multi-step programming tasks. We hypothesize that 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 having zero prior training exposure to Anka, 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, achieving near-native accuracy; (2) constrained syntax significantly reduces errors on complex tasks; and (3) domain-specific languages purposefully designed for LLM generation can outperform general-purpose languages on which the LLM has extensive training. We release the complete language implementation, benchmark suite, and evaluation framework to facilitate further research.

Keywords: Large Language Models, Domain-Specific Languages, Code Generation, Data Transformation, Prompt Engineering, Constrained Generation

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, from autocomplete suggestions to fully autonomous coding agents [GitHub, 2022]. However, despite impressive performance on isolated programming 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 in iteration, 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 across sequential steps.

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 for filtering, mapping, and aggregating data, we can design a language where each operation has exactly one canonical form. Such constraints, while potentially limiting for human programmers, may provide the structural guidance that LLMs need to generate reliable code.

To test this hypothesis, we introduce **Anka**, a domain-specific language for data transformation pipelines. Anka enforces explicit syntax through several 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 over symbols**: FILTER, MAP, AGGREGATE rather than operators

Our evaluation addresses two research questions:

RQ1: Can LLMs learn novel DSLs entirely from in-context prompts, without fine-tuning?

RQ2: Does constrained syntax reduce errors on complex, multi-step code generation tasks?

We evaluate Anka against Python on a benchmark suite of 100 data transformation tasks spanning eight categories, from simple filtering to complex multi-step pipelines. Our key findings are:

- **Novel DSL acquisition:** Despite zero training exposure to Anka, Claude 3.5 Haiku achieves 99.9% parse success, demonstrating that LLMs can effectively 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. This advantage is confirmed across models: GPT-4o-mini shows a +26.7 percentage point improvement.

- **Overall improvement:** Anka achieves 95.8% overall accuracy compared to 91.2% for Python (+4.6 percentage points), despite Python's substantial training data advantage.

These results suggest that **purposeful DSL design** can meaningfully improve LLM reliability for domain-specific tasks. 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.

Contributions. We make the following contributions:

1. We introduce Anka, a DSL for data transformations designed with explicit syntax to reduce LLM errors.

2. We present a benchmark suite of 100 tasks across 8 categories for evaluating code generation on data transformation.

3. We demonstrate that LLMs can learn novel DSLs from prompts, achieving 99.9% parse success with zero training data.

4. We show a 40% accuracy improvement on multi-step tasks, validated across two model families.

5. We release all code, benchmarks, and evaluation infrastructure for reproducibility.

2   Related Work

LLM Code Generation.   The emergence of large-scale code generation models has transformed program synthesis. Codex [Chen et al., 2021] demonstrated that language models trained on code repositories 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 through training on permissively licensed code. Commercial deployments such as GitHub Copilot [GitHub, 2022] and Amazon CodeWhisperer now assist millions of developers.

Despite these advances, systematic evaluations reveal consistent failure modes. HumanEval [Chen et al., 2021] and MBPP [Austin et al., 2021] benchmark functional correctness, while APPS [Hendrycks et al., 2021] tests on competitive programming problems. These benchmarks demonstrate that accuracy degrades substantially as task complexity increases. Our work complements these efforts by demonstrating that language design, not just model scale, can address complexity-related failures.

Domain-Specific Languages.   Domain-specific languages (DSLs) trade generality for expressiveness within a narrow domain [Fowler, 2010, Mernik et al., 2005]. In data processing, SQL remains the dominant DSL for structured queries, while dataframe libraries (pandas, dplyr) provide programmatic alternatives. Recent work has explored DSLs specifically designed for program synthesis: FlashFill [Gulwani, 2011] uses a DSL for string transformations, and Dream-Coder [Ellis et al., 2021] learns DSL primitives during synthesis.

Our work differs in designing a DSL specifically for *LLM* generation rather than human use. Where traditional DSL design prioritizes human ergonomics, Anka prioritizes features that reduce LLM errors: explicit naming, verbose keywords, and canonical forms.

Prompt Engineering and Constrained Generation.   Prompt engineering techniques can substantially improve LLM performance without model modification. Chain-of-thought prompting [Wei et al., 2022] improves reasoning through intermediate steps. Self-consistency [Wang et al., 2023] aggregates multiple samples. For code generation, Jiang et al. [2023] demonstrate that planning before coding improves accuracy.

Constrained decoding approaches guide generation toward valid outputs. Grammar-constrained decoding [Scholak et al., 2021, Poesia et al., 2022] ensures syntactic validity by masking invalid tokens. JSON mode in commercial APIs enforces structural constraints. Our approach is complementary: rather than constraining the *decoding process*, we constrain the *target language* itself, allowing standard decoding while reducing error probability.

3   The Anka Language

Anka is a domain-specific language for data transformation pipelines. Its design prioritizes features that reduce LLM code generation errors rather than features that improve human developer experience. In this section, we describe Anka's design principles, syntax, and the rationale connecting each design decision to error prevention.

3.1   Design Principles

We designed Anka around four principles, each motivated by observed LLM error patterns:

Principle 1: One Canonical Form.   In Python, filtering a dataframe can be expressed multiple ways: `df[df.x > 5]`, `df.query("x > 5")`, `df.loc[df.x > 5]`, or comprehension-based

approaches. This flexibility forces LLMs to choose among equivalent options, introducing inconsistency. In Anka, filtering has exactly one form:

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

Principle 2: Named Intermediate Results. Multi-step pipelines require managing intermediate state. In Python, developers may reuse variable names, chain operations, or use anonymous intermediates. These patterns cause LLM errors when the model loses track of which variable holds which data. Anka requires explicit `INTO` clauses:

```
STEP filter_large:
    FILTER orders WHERE amount > 1000 INTO large_orders
STEP summarize:
    AGGREGATE large_orders COMPUTE SUM(amount) AS total INTO summary
```

Principle 3: Explicit Step Structure. Anka organizes operations into named `STEP` blocks. This structure serves as "scaffolding" that guides the LLM through sequential operations, making the pipeline structure explicit rather than implicit in code flow.

Principle 4: Verbose Keywords. Where Python uses operators and method chains, Anka uses English keywords: `FILTER, MAP, AGGREGATE, WHERE, INTO`. Verbose syntax trades brevity for clarity, which aligns well with LLM capabilities—these models excel at natural language, and keyword-heavy syntax leverages this strength.

3.2 Syntax Overview

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

```
PIPELINE transform_sales:
    INPUT orders:  TABLE[order_id:  INT, customer:  STRING,
           amount:  DECIMAL, date:  DATE]

    STEP filter_large:
        FILTER orders WHERE amount > 1000 INTO large_orders

    STEP add_tax:
        MAP large_orders WITH tax => amount * 0.08 INTO with_tax

    STEP summarize:
        AGGREGATE with_tax
        GROUP_BY customer
        COMPUTE SUM(amount) AS total, COUNT() AS num_orders
        INTO summary

    OUTPUT summary
```

Type Declarations. Input tables declare their schema using `TABLE[field:  TYPE, ...]` syntax. Supported types include `INT, STRING, DECIMAL, BOOL, DATE`, and `DATETIME`. Explicit types enable both validation and serve as documentation in the prompt.

Operations. Anka supports 18 data operations organized into categories:

- **Selection:** FILTER, SELECT, DISTINCT

- **Transformation:** MAP, RENAME, DROP, ADD_COLUMN

- **Aggregation:** AGGREGATE with COUNT, SUM, AVG, MIN, MAX

- **Ordering:** SORT (ASC/DESC), LIMIT, SKIP, SLICE

- **Combination:** JOIN, LEFT_JOIN, UNION

- **I/O:** READ, WRITE (JSON/CSV), FETCH, POST (HTTP)

3.3 Connection to Error Prevention

Each design principle addresses specific LLM error patterns:

Design Feature	Error Prevented	Mechanism
Canonical forms	Inconsistent syntax	Eliminates decision points
INTO clauses	Variable shadowing	Explicit naming enforced
STEP structure	Ordering errors	Visual scaffolding
Verbose keywords	Operator confusion	Leverages LLM language
Typed inputs	Schema mismatches	Documentation in prompt

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

3.4 Implementation

Anka is implemented in Python using Lark for parsing. The implementation comprises approximately 6,400 lines of code including:

- A formal grammar (98 production rules)

- 68 AST node types as immutable dataclasses with source location tracking

- A tree-walking interpreter supporting all 18 operations

- Control flow constructs (IF/ELSE, FOR_EACH, WHILE, TRY/ON_ERROR)

- 322 unit tests achieving comprehensive coverage

  The complete implementation is available at https://github.com/BleBlo/Anka.

4 Methodology

We evaluate whether Anka's constrained syntax improves LLM code generation accuracy compared to Python. This section describes our benchmark design, evaluation protocol, and metrics.

4.1 Benchmark Suite

We constructed a benchmark of 100 data transformation tasks organized into eight categories:
Each task specifies: a natural language description, an input schema with field names and types, and test cases with input data and expected output.

The multi-step category is particularly important for our hypothesis: these tasks require maintaining state across 3–5 operations, precisely where we expect constrained syntax to help most.

Category	Tasks	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 to trigger common errors

Table 2: Benchmark categories and task distribution.

4.2 Evaluation Protocol

For each task, we prompt the LLM to generate code in both Anka and Python. To ensure fair comparison:

Prompt Structure. Both prompts follow identical structure: language specification, task description (identical), input schema (identical), and expected output format.

The Anka prompt includes a concise syntax guide (approximately 100 lines) teaching the language from scratch. The Python prompt assumes pandas knowledge, consistent with training data distribution.

Sampling. We generate 10 samples per task per language using temperature 0.3. Multiple samples allow us to measure consistency and distinguish systematic errors from sampling variance.

Models. We evaluate on Claude 3.5 Haiku (Anthropic) as our primary model, with GPT-4o-mini (OpenAI) for cross-model validation.

4.3 Metrics

We report four metrics:

- **Parse Success:** Does the generated code parse without syntax errors?

- **Execution Success:** Does the code execute without runtime errors?

- **Output Correctness:** Does the output match the expected result?

- **Task Accuracy:** Fraction of tasks where $\geq 50\%$ of samples produce correct output (our primary metric)

4.4 Fair Comparison Considerations

Python has a substantial advantage: LLMs have seen billions of Python examples during training, while Anka is entirely novel. Any Anka advantage must overcome this training distribution gap through in-context learning alone.

5 Results

5.1 Main Results

Table 3 presents task accuracy by category for Claude 3.5 Haiku.

6

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.

Key Finding 1: Multi-step Advantage. The most striking result is on multi-step tasks: Anka achieves **100% accuracy** compared to 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 having zero training exposure to Anka, the model achieves **99.9% parse success**. This demonstrates that LLMs can effectively learn novel programming languages entirely from in-context prompts.

Key Finding 3: Overall Improvement. Anka achieves 95.8% overall accuracy compared to 91.2% for Python (+4.6 percentage points). This improvement is notable given Python's substantial training data advantage.

5.2 Cross-Model Validation

To verify that our findings generalize beyond a single model, we evaluated GPT-4o-mini on the multi-step category:

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.

GPT-4o-mini shows a +26.7 percentage point 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 Analysis: Why Does Anka Help?

We analyzed failing Python generations to understand the error patterns that Anka prevents:

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

Operation Sequencing (31% of errors). Multi-step tasks require operations in a specific order. Python's flexibility allows operations to be combined or reordered in ways that change semantics. Anka's `STEP` structure makes ordering explicit and sequential.

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

5.4 Complexity Analysis

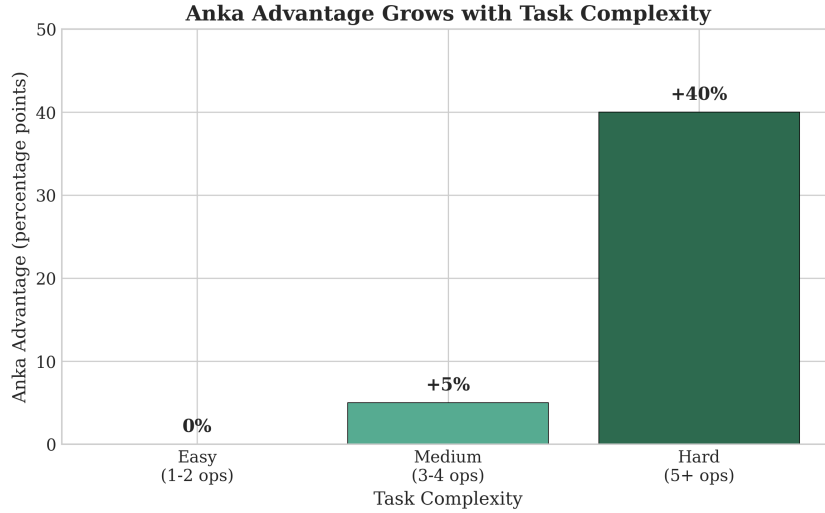Figure 1 shows Anka's advantage as a function of task complexity:

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

- **Simple (1–2 ops):** 0% advantage—both languages perform well
- **Medium (3–4 ops):** +5% advantage—constraint begins to help
- **Complex (5+ ops):** +40% advantage—constraint critical

6 Discussion

6.1 Why Does Constrained Syntax Help?

Our results suggest three mechanisms by which constrained syntax improves LLM code generation:

Reduced Decision Space. Each syntactic choice point is an opportunity for error. By eliminating alternatives, Anka reduces the number of decisions the model must make. In a 5-step pipeline with 3 choice points 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 tracking which variable holds which data through implicit Python semantics, the model can "read off" the current state from variable names.

Structural Scaffolding. The `STEP` structure provides a template that guides generation. The model fills in steps sequentially rather than generating a monolithic program, reducing cognitive load.

6.2 When Does Anka Not Help?

Anka shows no advantage on simple tasks and slight disadvantage on "hard" tasks:

Simple Tasks. When only 1–2 operations are required, there is insufficient complexity for errors to accumulate. Python's training advantage may actually help here.

Complex Conditional Logic. "Hard" tasks often require nested conditionals, edge case handling, and domain-specific reasoning. Here, Python's flexibility becomes an asset.

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 DSLs intended for LLM generation:

1. **Canonicalization:** Provide exactly one way to express each operation

2. **Explicit Naming:** Require names for intermediate results

3. **Structural Templates:** Use block structure to guide sequential generation

4. **Verbose Keywords:** Prefer English keywords over symbols

5. **Type Documentation:** Include type information in prompts

7 Limitations

We acknowledge several 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 improve 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. Whether developers find the resulting code readable is an open question.

Single Benchmark Suite. Despite efforts to include diverse tasks, our benchmark may contain biases that favor Anka.

8 Conclusion

We introduced Anka, a domain-specific language for data transformation designed to improve LLM code generation accuracy through constrained, explicit syntax. Our evaluation demonstrates three key findings:

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

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

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

The broader contribution is methodological: we demonstrate that **language design** is a viable intervention for improving LLM reliability. Rather than solely improving models through scale or fine-tuning, we can design languages that play to LLM strengths and mitigate their weaknesses.

Future Work. Several directions merit investigation: evaluation on additional model families; user studies on developer experience; production deployment evaluation; and extension to other domains such as financial calculations and workflow automation.

We release the complete Anka implementation, benchmark suite, and evaluation framework at https://github.com/BleBlo/Anka.

Acknowledgments

[ACKNOWLEDGMENTS TO BE ADDED]

Ethics Statement

This work presents a domain-specific language 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, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

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

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

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

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

Sumit Gulwani. 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, 2011.

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

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

Xue Jiang, Yihong Dong, Lecheng Wang, Qiwei Shang, and Ge Li. Self-planning code generation with large language models. *arXiv preprint arXiv:2303.06689*, 2023.

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

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

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

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

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

Torsten Scholak, Nathan Schucher, and Dzmitry Bahdanau. 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, 2021.

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

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