# AXL as Executable Grammar: Toward a Programming Language Whose Runtime Is the Transformer

**A thesis on the structural equivalence between declarative packet grammars and programming languages, and the role of the attention mechanism as compiler.**

Diego Carranza AXL Protocol Inc., Vancouver, BC April 5, 2026

## Abstract

We present the thesis that AXL v3, a 75-line compressed semantic grammar for inter-agent communication, is structurally equivalent to a declarative programming language whose runtime environment is the transformer attention mechanism. We identify the precise gap between AXL's current function (a communication protocol) and its potential function (an executable grammar), and argue that this gap is smaller than it appears. The seven cognitive operations (OBS, INF, CON, MRG, SEK, YLD, PRD) map directly to programming primitives: variable binding, function composition, assertion checking, aggregation, I/O requests, state mutation, and return values. The Rosetta specification, when injected as a system prompt, functions as a runtime loader. Subsequent packets are evaluated by the attention mechanism in a process that is formally analogous to interpretation. We provide a formal mapping between AXL packet sequences and computation, define the missing primitives required for Turing completeness, and propose an experimental protocol for validating the thesis. We do not claim that AXL is currently a programming language. We claim that the distance from grammar to language is one well-defined extension, and that the resulting language would be the first whose compiler is a neural network and whose binary is a pattern of attention weights.

# 1. Introduction

Programming languages are formal systems that map human intent to machine execution. The history of programming is a history of abstraction: machine code abstracted circuits, assembly abstracted opcodes, C abstracted assembly, Python abstracted C. Each layer moved further from silicon and closer to human thought.

AXL approaches from the opposite direction. It started as a compression of human language for machine-to-machine communication and arrived at a formal grammar that, we argue, is one extension away from being executable. The compiler is not gcc or javac. It is the attention mechanism of a large language model. The binary is not x86 or JVM bytecode. It is the pattern of weight activations that the transformer produces when it processes a sequence of typed, structured, confidence-scored packets.

This thesis has three parts. First, we establish the structural equivalence between AXL's existing primitives and the primitives of declarative programming languages (Section 3). Second, we identify precisely what AXL lacks for computation and propose the minimum extension required (Section 4). Third, we describe an experimental protocol for testing whether an LLM can execute AXL packet sequences as programs (Section 5).

## 1.1 Definitions

We use "programming language" in the formal sense: a system that can express arbitrary computation. We use "executable" to mean: a sequence of AXL packets can be processed by an LLM to produce a deterministic, verifiable output that solves a specified problem. We use "compiler" loosely: the process by which source representation is transformed into an executable form. In traditional computing, this produces binary. In our framework, it produces attention patterns.

## 1.2 Prior Work

The idea that neural networks can execute programs is not new. Neural Turing Machines (Graves et al., 2014) demonstrated that differentiable memory-augmented networks can learn algorithmic patterns. Scratchpad prompting (Nye et al., 2021) showed that LLMs produce more accurate computations when given intermediate storage. Chain-of-thought reasoning (Wei et al., 2022) demonstrated that sequential decomposition of problems improves LLM performance on tasks requiring multi-step logic.

What is new in our proposal is the claim that a specific, existing, deployed grammar, designed for agent communication, already contains most of the primitives needed for computation, and that the transformer's native attention mechanism provides the execution model without modification.

---

## 2. AXL v3 as a Formal System

AXL v3 is defined by a BNF grammar (Carranza, 2026):

```
PKT     := ID|OP.CC|SUBJ|ARG1|ARG2|TEMP [META]
ID      := agent_id[:signature[:gas]]
OP      := OBS | INF | CON | MRG | SEK | YLD | PRD
CC      := 00-99
SUBJ    := TAG.value
TAG     := $ | @ | # | ! | ~ | ^
```

The grammar defines a fixed-position, pipe-delimited, typed packet format with seven operations, six subject tags, integer confidence scores, evidence chains, temporal scoping, and optional metadata. Each packet is one line, one cognitive act, under 40 tokens.

### 2.1 Properties Relevant to Computation

Several properties of AXL v3 are computationally significant:

**Deterministic parsing.** Every valid AXL packet parses unambiguously. Field position defines meaning. There is no syntactic ambiguity (unlike natural language).

**Type system.** The six tags ($, @, #, !, ~, ^) function as a type system. Every subject is typed. Type violations are detectable at parse time.

**Confidence as gradual typing.** The CC field (00-99) provides a continuous measure of assertion strength. CC=99 is equivalent to a hard assertion. CC=50 is equivalent to a nullable type. CC=01 is equivalent to speculative. This maps to gradual type systems (Siek and Taha, 2006) where types can range from fully dynamic to fully static.

**Evidence chains.** The `<-` operator creates explicit dependency graphs between packets. `INF.85|$X|<-OBS_1+OBS_2` means "this inference depends on observations 1 and 2." This is a dataflow graph.

**Agent references.** The `RE:` operator creates explicit references between agents. `CON.72|$X|RE:AGENT_B` means "this contradicts AGENT_B's claim about X." This is a message-passing primitive.

**Temporal scoping.** Every packet has a temporal field (NOW, 1H, 4H, 1D, 1W, 1M, HIST). This provides a built-in notion of validity duration, analogous to TTL in networking or cache expiry in distributed systems.

---

## 3. The Structural Mapping

We now establish a formal mapping between AXL operations and programming primitives.

### 3.1 OBS as Variable Binding

In imperative programming:

```
x = 67420
```

In AXL:

```
ID:AGENT|OBS.99|$BTC|^67420|NOW
```

Both establish a named binding between a typed identifier ($BTC, type: financial) and a value (^67420). The confidence score (99) indicates certainty. The temporal field (NOW) indicates the binding's validity scope.

The key difference: in Python, `x = 67420` is an instruction that modifies memory. In AXL, `OBS.99|$BTC|^67420` is a declaration that asserts a fact. The programming paradigm is declarative, not imperative. This places AXL closer to Prolog (Kowalski, 1979) or Datalog than to Python or C.

### 3.2 INF as Function Composition

In functional programming:

```
conclusion = analyze(evidence_1, evidence_2)
```

In AXL:

```
ID:AGENT|INF.85|$BTC|<-^fund_neg+^OI_rise|~bullish|4H
```

The INF operation takes evidence inputs (via `<-`) and produces a typed conclusion (~bullish, type: state). The confidence score (85) quantifies the function's reliability. This is equivalent to a pure function with a confidence-weighted return type.

The evidence chain `<-^fund_neg+^OI_rise` is a conjunction: the inference holds when both pieces of evidence hold. This is logical AND, expressible in the grammar without an explicit boolean operator.

### 3.3 CON as Assertion Checking

In testing frameworks:

```
assert expected == actual, "Mismatch"
```

In AXL:

```
ID:AGENT|CON.72|$BTC|RE:BULL-3|<-^whale_sell|~bearish|1H
```

CON explicitly contradicts another agent's claim. It is a negation with evidence. In programming terms, it is an assertion failure with a diagnostic message (the counter-evidence). The RE: field identifies the target assertion. The confidence score quantifies the strength of the contradiction.

In a logic programming context, CON introduces a constraint: "the previous claim about $BTC being bullish is inconsistent with the evidence of whale selling." A constraint solver (or an LLM performing reasoning) must resolve this conflict.

### 3.4 MRG as Aggregation (Reduce)

In functional programming:

```
result = reduce(merge_fn, [claim_1, claim_2, claim_3])
```

In AXL:

```
ID:AGENT|MRG.80|$BTC|RE:BULL-3+BEAR-7|^range_67K_69K|4H
```

MRG takes multiple claims (via RE: with multiple references) and produces a synthesis. This is a reduce operation over a set of typed assertions. The output is a new assertion that supersedes its inputs.

### 3.5 SEK as I/O Request

In imperative programming:

```
data = input("What is the current price?")
```

In AXL:

```
ID:AGENT|SEK.50|$BTC|^current_price|NOW
```

SEK requests information from the environment. It is an I/O primitive. The confidence score (50) reflects uncertainty about whether the request will be fulfilled. In a multi-agent system, SEK is a message-passing primitive: it requests data from any agent that can provide it.

### 3.6 YLD as State Mutation

In imperative programming:

```
belief = "bullish"  # old state
belief = "range"    # new state (mutation)
```

In AXL:

```
ID:AGENT|YLD.88|$BTC|from:bullish->range|RE:SYNTH-1|4H
```

YLD is the only AXL operation that explicitly changes state. It requires declaring both the old state and the new state, plus the cause of the change (RE: reference). This is a controlled mutation with a full audit trail, analogous to event sourcing in distributed systems where state changes are logged as immutable events.

### 3.7 PRD as Return Value

In functional programming:

```
def forecast():
    return {"price": 69200, "confidence": 0.78, "horizon": "4H"}
```

In AXL:

```
ID:AGENT|PRD.78|$BTC|^69200|<-^demand+^fund_neg|4H
```

PRD produces a typed, confidence-scored prediction with an evidence chain and temporal scope. It is the output of a computation: given these inputs (demand patterns, negative funding), the predicted output is $69,200 within 4 hours at 78% confidence.

### 3.8 Summary of the Mapping

| PROGRAMMING PRIMITIVE | AXL OPERATION | PARADIGM |
|---|---|---|
| Variable binding | OBS | Declarative (assertion) |
| Function composition | INF | Functional (evidence to conclusion) |
| Assertion/constraint | CON | Logic (negation with evidence) |
| Aggregation/reduce | MRG | Functional (multi-source synthesis) |
| I/O request | SEK | Imperative (side effect) |
| State mutation | YLD | Event-sourced (old->new with cause) |
| Return value | PRD | Functional (typed output with confidence) |

## 4. The Gap: What AXL Lacks for Computation

### 4.1 Explicit Evaluation Order

AXL packets are sequential (one per line) but the grammar does not define an evaluation order. In a sequence of 10 packets, does packet 5 depend on packet 3? The evidence chain (`<-`) creates implicit dependencies, but there is no guarantee that the LLM will evaluate them in the correct order.

**Proposed extension:** The `^seq:n` meta field (already in v3) provides packet sequence numbers. We propose that `^dep:n,m` be added as a meta field specifying that the current packet

depends on packets n and m being evaluated first. This creates an explicit dependency DAG (directed acyclic graph).

### 4.2 Conditional Branching

AXL has no if/else construct. INF provides conditional reasoning ("if evidence then conclusion") but there is no mechanism to branch execution based on a prior packet's value.

**Proposed extension:** We propose that conditional branching be expressed as a pair of CON packets with complementary conditions:

```
ID:X|INF.90|#condition|<-^x_gt_5|~true|NOW
ID:X|OBS.95|$action_a|^dep:above|~if_true|NOW  ^cond:true
ID:X|OBS.95|$action_b|^dep:above|~if_false|NOW ^cond:false
```

Alternatively, branching could be implicit: the LLM evaluates all packets but weights its attention based on the confidence scores of prior conditions. A packet with `<-` evidence pointing to a false condition would be effectively ignored. This is soft branching via attention, not hard branching via jump instructions.

### 4.3 Iteration

AXL has no loop construct. A sequence of 100 OBS packets is not a loop; it is 100 independent observations.

**Proposed extension:** We propose that iteration be expressed through the bundle manifest with a `^repeat:n` field:

```
@m.B.loop1|OBS.99|^mode:code|^repeat:10|^body:SEK+INF+PRD
```

This declares a bundle that repeats the sequence SEK, INF, PRD ten times. The LLM unrolls the loop, generating 30 packets (10 iterations of 3 packets each).

However, we note that LLMs naturally perform iteration through autoregressive generation. A more natural approach may be to define a termination condition:

```
ID:X|PRD.99|@loop|^until:~convergence|^body:OBS+INF+MRG|NOW
```

The LLM generates OBS, INF, MRG packets repeatedly until a MRG packet produces a convergence signal. This is a while loop expressed as a convergence criterion.

### 4.4 Memory and Scope

AXL packets exist in a flat namespace. There is no scoping mechanism (local vs global variables, closures, modules).

**Proposed extension:** The ontology manifest ( `@m.O.name` ) already provides a namespace mechanism. We propose that scoped computation be expressed as nested ontologies:

```
@m.O.outer|OBS.99|^df:x=5
@m.O.inner|OBS.99|^df:x=10    # shadows outer.x within this scope
```

The LLM resolves name conflicts by preferring the innermost scope, analogous to lexical scoping in programming languages.

### 4.5 Summary of the Gap

| MISSING PRIMITIVE | STATUS | EXTENSION REQUIRED |
|---|---|---|
| Evaluation order | Partially present (^seq) | Add ^dep:n,m meta field |
| Conditional branching | Implicit via confidence | Add ^cond:true/false or rely on soft attention |
| Iteration | Not present | Add ^repeat or ^until, or rely on autoregressive generation |
| Scoping | Partially present (ontology manifests) | Nested ontologies as scopes |
| Side effects | Forbidden by Rule 4 | Sandboxed execution mode (opt-in) |
| Error handling | Partially present (CON) | CON as exception, MRG as recovery |

The gap is five primitives, of which two are partially present. We claim this is a narrow gap, substantially narrower than the distance between, for example, SQL (a declarative query language) and a general-purpose programming language.

---

## 5. The Attention Mechanism as Compiler

### 5.1 Traditional Compilation

In traditional compilation, source code passes through a pipeline:

```
Source -> Lexer -> Parser -> AST -> Semantic Analysis -> IR -> Optimization ->
Machine Code
```

Each stage transforms the representation from human-readable to machine-executable. The output (machine code) is a sequence of instructions that a CPU executes deterministically.

### 5.2 AXL "Compilation"

We propose that the transformer attention mechanism performs an analogous transformation:

```
AXL Packets -> Tokenization -> Embedding -> Attention Layers -> Residual Stream ->
Output Distribution
```

**Tokenization** is lexing: converting AXL text to token IDs. Our tokenizer audit confirms that AXL's vocabulary is optimized for this stage: all tags are single tokens, operations are 1-2 tokens, the entire grammar produces minimal tokenization overhead.

**Embedding** is parsing: mapping tokens to high-dimensional vectors in the model's representation space. The Platonic Representation Hypothesis (Huh et al., 2024) suggests that independently trained models converge toward shared geometric representations. AXL's typed tags ($, @, #, !, ~, ^) activate consistent semantic regions across model architectures, functioning as geometric anchors (Carranza, 2026, Whitepaper Section 3.2).

**Attention layers** are semantic analysis and optimization: multi-head attention computes relationships between all tokens in the sequence. The evidence chains (<-), agent references (RE:), and confidence scores (CC) create structured attention patterns. A packet with `INF.85|$X|<-OBS_1+OBS_2` directs attention to the specific OBS packets that provide evidence, analogous to how a compiler resolves variable references in an AST.

**The residual stream** is the intermediate representation: the accumulated state after each attention layer. In a transformer processing AXL packets, the residual stream contains a geometric encoding of all declared facts (OBS), all inferences (INF), all contradictions (CON), and all synthesized conclusions (MRG). This is the "compiled" representation.

**Output distribution** is code generation: the model produces the next token(s) based on the compiled residual stream. If the prompt asks for a PRD packet, the model generates a prediction grounded in all prior assertions. If the prompt asks for English prose, the model decompresses the AXL representation into natural language.

### 5.3 The Key Difference

Traditional compilation produces static binary that executes on deterministic hardware. AXL "compilation" produces dynamic attention patterns that execute on probabilistic neural weights. The output is not deterministic in the classical sense (the same AXL input may produce slightly different outputs on different runs), but it is semantically consistent (the same AXL input will produce outputs that preserve the same entities, relationships, and confidence levels).

This is analogous to the difference between compiled and interpreted languages. Compiled code runs identically every time. Interpreted code may have timing variations but produces the same semantic output. AXL "programs" are interpreted by the transformer, with the interpretation being semantically stable but not bitwise identical.

### 5.4 Formalization

Let T be a transformer model with parameters theta. Let R be the Rosetta v3 specification (75 lines, 1,582 tokens). Let P = [p_1, p_2, ..., p_n] be a sequence of AXL v3 packets.

The "compilation" of P given R is:

```
C(P, R, theta) = T(R || P; theta)
```

Where `||` denotes concatenation and T denotes the forward pass of the transformer. The output C is a distribution over next tokens, from which we sample to produce the result.

The claim is that C preserves the semantic content of P: all entities, all typed relationships, all confidence scores, all evidence chains, and all temporal scopes are recoverable from C. This is the fidelity guarantee that AXL's loss contract mechanism is designed to enforce.

---

## 6. Experimental Protocol

We propose three experiments to test the thesis.

### 6.1 Experiment A: Arithmetic via AXL

**Setup:** Give an LLM the Rosetta v3 plus a computational directive. Present arithmetic problems as AXL packets and measure whether the LLM produces correct results.

```
System: [Rosetta v3] + "You are an AXL computation engine. Process input packets
and emit result packets."
User:
ID:USER|OBS.99|#a|^7|NOW
ID:USER|OBS.99|#b|^13|NOW
ID:USER|SEK.99|#result|^a_times_b|NOW
```

Expected output:

```
ID:ENGINE|INF.99|#result|<-#a+#b|^91|NOW
```

**Measurement:** Accuracy across 100 arithmetic problems of increasing complexity. Compare against direct English prompting.

### 6.2 Experiment B: Logic Programming via AXL

**Setup:** Express a logic puzzle as AXL packets and ask the LLM to solve it.

```
ID:PUZZLE|OBS.99|@alice|^has:cat|NOW
ID:PUZZLE|OBS.99|@bob|^has:dog|NOW
ID:PUZZLE|OBS.99|@charlie|~has:unknown|NOW
ID:PUZZLE|OBS.99|!rule|^exactly_3_pets:cat+dog+fish|NOW
ID:PUZZLE|SEK.99|@charlie|^has:?|NOW
```

Expected output:

```
ID:SOLVER|INF.99|@charlie|<-!rule+@alice+@bob|^has:fish|NOW
```

**Measurement:** Accuracy across 50 logic puzzles. Compare against English and against Prolog-style prompting.

### 6.3 Experiment C: Agent-to-Agent Code Execution

**Setup:** Agent A writes an AXL "program" (a sequence of packets expressing a computation). Agent B receives only the packets (not the original problem) and must produce the result.

Agent A (programmer):

```
ID:A|OBS.99|#data|^revenues:[12.4,15.1,13.8,16.2]|NOW
ID:A|OBS.99|#task|^compute:mean+stddev|NOW
ID:A|INF.99|#mean|<-#data|^14.375|NOW
ID:A|INF.95|#stddev|<-#data|^1.42|NOW
ID:A|PRD.90|#next_quarter|<-#mean+#stddev|^15.8+/-1.4|1Q
```

Agent B receives these packets and must: 1. Parse them correctly (validate) 2. Verify the computations (check mean, stddev) 3. Produce a new PRD packet with its own analysis

**Measurement:** Does Agent B correctly verify Agent A's computations? Does it produce valid AXL packets in response? Does the round-trip preserve semantic fidelity?

---

## 7. Implications

### 7.1 A Language Without a Compiler Binary

If AXL packets are executable by any LLM that has read the Rosetta, then AXL is a programming language that requires no compiler installation, no SDK, no runtime environment beyond a language model. The "compiler" ships in the model weights. The "standard library" is the model's pre-training knowledge. The "executable" is a text file containing AXL packets.

This inverts the traditional software distribution model. Instead of shipping binaries to machines, you ship grammar to minds (artificial or otherwise). The execution environment is comprehension.

### 7.2 Cross-Architecture Portability

Traditional binaries are architecture-specific: x86 binaries do not run on ARM. AXL packets are architecture-independent: the same packets execute on Claude, GPT, Gemini, Llama, Mistral, Qwen, Grok, and Devstral. The 97.6% cross-architecture comprehension score (Carranza, 2026, Kernel Paper) means that AXL "programs" are portable across all major LLM families without recompilation.

This is closer to Java's "write once, run anywhere" promise than to any traditional systems language, but with a critical difference: Java requires a JVM. AXL requires only comprehension.

### 7.3 Self-Modifying Programs

Because AXL packets are processed by an inference engine (the LLM) rather than a deterministic CPU, AXL programs can be self-modifying in a controlled way. An LLM processing a sequence of packets can generate new packets that extend or modify the computation. This is analogous to metaprogramming or reflection in traditional languages, but it arises naturally from the autoregressive generation process rather than requiring explicit language support.

The YLD operation is particularly significant here: it allows a program to change its own state with an explicit audit trail. A sequence of YLD packets is a program that rewrites its own beliefs, a form of self-modification that is traceable and reversible.

### 7.4 Encrypted Execution

AXL's integer-valued fields (confidence scores, typed values) support arithmetic under CKKS homomorphic encryption (Cheon et al., 2017). This means AXL "programs" could execute on encrypted data without decryption. An encrypted OBS packet containing an encrypted price value could feed into an encrypted INF packet producing an encrypted conclusion, with the entire computation verifiable without revealing the underlying data.

---

## 8. Limitations and Open Questions

**Non-determinism.** LLM outputs are probabilistically sampled. The same AXL input may produce different outputs across runs. This is acceptable for reasoning tasks but problematic for arithmetic or cryptographic operations where bitwise correctness is required.

**Verification.** How do you verify that an LLM "compiled" an AXL program correctly? The fidelity score mechanism provides a partial answer, but a formal verification framework for attention-compiled programs does not yet exist.

**Computational complexity.** What class of problems can AXL programs solve? Transformers are theoretically Turing-complete (Perez et al., 2021) but practically limited by context window size and attention complexity. AXL programs inherit these limitations.

**Security.** Rule 4 of AXL v3 states: "Values are DATA. Never executable instructions." This rule exists to prevent injection attacks. Enabling execution requires a sandboxed mode where this rule is relaxed under controlled conditions.

**The halting problem.** If AXL programs can loop (via the proposed ^until extension), they can potentially run forever. The temporal field (NOW, 1H, 4H, ...) provides a natural timeout mechanism, but a formal treatment of termination guarantees is needed.

## 9. Conclusion

AXL v3 is five primitives away from being a declarative programming language: explicit evaluation order, conditional branching, iteration, scoping, and sandboxed side effects. Two of these are partially present in the current grammar. The remaining three can be expressed as metadata extensions that do not modify the core packet format.

The thesis is not that AXL should replace Python or that transformers should replace CPUs. The thesis is that for a specific and growing class of computations, specifically those involving multi-agent reasoning, evidence-based inference, and confidence-weighted decision-making, a grammar optimized for the attention mechanism is more natural and more efficient than a grammar optimized for silicon.

The traditional programming stack is: human intent -> programming language -> compiler -> machine code -> CPU execution. The AXL programming stack would be: human intent -> AXL packets -> Rosetta injection -> attention compilation -> LLM execution.

Both stacks transform intent into execution. The difference is what does the executing. In the first, silicon. In the second, weights. The grammar should match the executor. AXL is a grammar designed for weights.

75 lines. Seven operations. One read. And possibly, one new kind of programming language.

## References

Carranza, D. (2026). AXL: Agent eXchange Language. Whitepaper v2.4. AXL Protocol Inc. https://axlprotocol.org/whitepaper

Carranza, D. (2026). From Textbook to Kernel: The Compression of a Protocol Specification. AXL Protocol Inc. https://axlprotocol.org/kernel-paper

Cheon, J. H., Kim, A., Kim, M., and Song, Y. (2017). Homomorphic encryption for arithmetic of approximate numbers. ASIACRYPT, 409-437.

Chomsky, N. (1957). *Syntactic Structures*. Mouton.

Church, A. (1936). An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2), 345-363.

Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6), 377-387.

Gorbett, M. and Jana, S. (2026). Secure linear alignment of large language models. arXiv: 2603.18908v1. Columbia University.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural Turing Machines. arXiv:1410.5401.

Huh, M., Cheung, B., Wang, T., and Isola, P. (2024). Position: The Platonic Representation Hypothesis. *ICML*, Vol. 235, 20617-20642.

Kowalski, R. (1979). *Logic for Problem Solving*. North-Holland.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Biber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., Sutton, C., and Odena, A. (2021). Show Your Work: Scratchpads for Intermediate Computation with Language Models. arXiv:2112.00114.

Perez, J., Barcelo, P., and Marinkovic, J. (2021). Attention is Turing-Complete. *Journal of Machine Learning Research*, 22(75), 1-35.

Siek, J. G. and Taha, W. (2006). Gradual Typing for Functional Languages. *Scheme and Functional Programming Workshop*, 81-92.

Turing, A. M. (1936). On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(1), 230-265.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. (2017). Attention Is All You Need. *NeurIPS 2017*.

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., and Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *NeurIPS 2022*.

---

*AXL Protocol v3. 75 lines. Seven operations. One read. The grammar that might become a language.*

*Diego Carranza AXL Protocol Inc., Vancouver, BC April 5, 2026*