Symbolic Reasoning in Large Language Models

A Comprehensive Guide to Induction Heads, Emergent Symbolic Mechanisms, and Practical Leverage Techniques

Synthesized from Research by

Catherine Olsson *et al.* (Anthropic)
Yukang Yang *et al.* (Princeton)
Eric Todd *et al.* (Northeastern/MIT)
Brown Ebouky *et al.* (IBM Research)

January 9, 2026

"These results point toward a resolution of the longstanding debate between symbolic and neural network approaches, suggesting that emergent reasoning in neural networks depends on the emergence of symbolic mechanisms."

— Yang et al., 2025

Preface

This document presents a comprehensive synthesis of cutting-edge research on symbolic reasoning capabilities in large language models. Over the past few years, a remarkable convergence has occurred: multiple independent research groups have discovered that neural networks trained on language develop internal mechanisms that bear striking resemblance to classical symbolic processing.

The implications of this discovery extend far beyond technical curiosity. For decades, artificial intelligence research was divided between two seemingly incompatible paradigms: symbolic AI, which manipulated discrete symbols according to explicit rules, and connectionist approaches, which learned distributed representations from data. The findings synthesized here suggest a resolution to this debate—not by choosing one side, but by demonstrating that symbolic processing can *emerge* from neural computation.

This guide is intended for machine learning practitioners who wish to understand both the theoretical foundations and practical applications of these discoveries. We begin with the mathematical framework that makes precise analysis possible, trace the development of mechanistic understanding from basic attention patterns to sophisticated symbolic architectures, and conclude with concrete techniques for leveraging these mechanisms in practice.

The journey takes us through several primary research threads: the mathematical framework for transformer circuits, the discovery of induction heads and their role in in-context learning, the identification of function vectors as compact task representations, the three-stage symbolic architecture that enables abstract reasoning, the cognitive tools framework that orchestrates these internal mechanisms externally, and activation-level interventions that provide direct control over model behavior.

Throughout, we emphasize the deep connections between these findings. What emerges is not a collection of isolated discoveries, but a unified theory of how symbolic computation arises within neural networks—and how we might harness this under-

standing to build more capable and controllable systems.

The synthesis presented here draws on work by researchers at Anthropic, Princeton, Northeastern, MIT, IBM Research, and others.

Contents

List of Figures

Part I

Theoretical Foundations

Chapter 1

Introduction: The Emergence of Symbolic Reasoning

> *"These results suggest a resolution to the longstanding debate between symbolic and neural network approaches, illustrating how neural networks can learn to perform abstract reasoning via the development of emergent symbol processing mechanisms."*
>
> — Yang et al., 2025

1.1 The Symbolic-Connectionist Debate

The question of how intelligence emerges from computation has divided artificial intelligence research for decades. On one side, the *symbolic* tradition—rooted in logic, rule systems, and explicit representations—argued that intelligence requires the manipulation of discrete symbols according to formal rules. Proponents like Allen Newell and Herbert Simon built systems that reasoned through explicit symbol manipulation, achieving remarkable success in constrained domains but struggling with the messiness of real-world knowledge.

On the other side, the *connectionist* paradigm proposed that intelligent behavior emerges from the distributed, statistical patterns of neural network activations. Rather than explicit rules, these systems learn implicit regularities from data, developing rep-

resentations that are robust, flexible, and often inscrutable.

Large Language Models appeared to offer decisive evidence for the connectionist view: systems trained purely on next-token prediction achieved remarkable capabilities in reasoning, analogy, and abstraction without any explicit symbolic machinery. A model like GPT-4 can solve logic puzzles, draw analogies, and manipulate abstract concepts—all without a single hand-coded rule. Yet this success raised a deeper question: *how* do these systems reason? Are they merely sophisticated pattern matchers, performing a kind of elaborate lookup in their training data? Or do they implement something more structured, something that resembles the systematic reasoning that symbolic AI aspired to?

Recent mechanistic interpretability research has revealed a surprising answer: neural networks trained on language modeling *spontaneously develop symbolic mechanisms*. These are not the brittle, hand-coded rules of classical symbolic AI, but emergent structures that combine the flexibility of neural computation with the systematicity of symbolic reasoning. The transformer architecture, it turns out, provides the raw materials for symbol processing; training on language provides the pressure to develop it.

This discovery carries profound implications. It suggests that the symbolic-connectionist debate rested on a false dichotomy. Neural networks don't need explicit symbolic architecture to perform symbolic computation—they develop their own form of symbol processing through the pressures of learning. Understanding these emergent mechanisms transforms how we interact with language models: we can design prompts that consciously activate symbolic processing, structure contexts that facilitate abstraction, and leverage the model's internal "grammar of thought."

1.2 Why Understanding Mechanisms Matters

One might ask: if language models already work, why do we need to understand *how* they work? The answer lies in the difference between using a tool and mastering it.

Consider an analogy to programming. One can write code through trial and error, trying different approaches until something works. But understanding the underlying abstractions—memory models, type systems, algorithmic complexity—transforms

programming from guesswork into engineering. You can predict what will work, diagnose why something fails, and design solutions that are robust rather than accidentally correct.

The same applies to working with language models. Without mechanistic understanding, prompt engineering remains largely empirical: we try different formulations, observe what works, and build intuitions that may or may not generalize. With mechanistic understanding, we can design prompts that align with computational structure, diagnose failures systematically, push capabilities further through targeted interventions, and anticipate emergent behaviors as models scale.

The shift from heuristic to principled prompting is like the shift from folk medicine to evidence-based medicine. Both can heal, but one does so reliably, with understanding of when and why it works.

1.3 Document Overview

This document synthesizes research from multiple sources to provide both theoretical understanding and practical guidance. The theoretical foundations come from several primary research threads:

- **Induction Heads** (Olsson et al., 2022): The foundational circuit enabling in-context learning, discovered through mechanistic interpretability of transformer models.
- **Emergent Symbolic Architecture** (Yang et al., 2025): The three-stage mechanism (symbol abstraction, symbolic induction, retrieval) that supports abstract reasoning in LLMs.
- **Function Vectors** (Todd et al., 2024): Compact representations of demonstrated procedures that can be extracted and transferred across contexts.
- **Cognitive Tools** (Ebouky et al., 2025): Research on how external structures can enhance the natural reasoning capabilities of language models by orchestrating internal mechanisms.
- **Activation Interventions**: Techniques for directly modifying model behavior by steering internal representations, providing finer control than prompting alone.

The interconnections between these research threads reveal a coherent picture: trans-

formers develop hierarchical symbolic processing that builds from simple pattern matching (induction heads) to abstract variable manipulation (symbolic mechanisms) to transferable procedural knowledge (function vectors). Each layer builds on the previous, and understanding this stack enables increasingly sophisticated interventions—whether through prompt design, external orchestration, or direct activation steering.

Chapter 2

Induction Heads: The Foundation of In-Context Learning

> *"Induction heads might constitute the mechanism for the majority of all 'in-context learning' in large transformer models."*
>
> — Olsson et al., 2022

2.1 The Mystery of In-Context Learning

Before diving into the mechanism, consider what in-context learning actually achieves. A language model, trained on billions of tokens of text, receives a prompt like:

```
apple -> fruit
hammer -> tool
salmon ->
```

Without any weight updates, the model outputs "fish." It has learned, from just two examples in the context, that the task is to produce category labels. This is remarkable. The model's weights were frozen; it learned purely from the structure of the prompt. How?

For years, this remained mysterious. In-context learning seemed almost magical—a

capability that emerged from scale without obvious explanation. The discovery of induction heads provided the first mechanistic account: specific attention circuits that implement a pattern-matching and copying algorithm that underlies in-context learning.

2.2 What Are Induction Heads?

Definition 2.1 (Induction Head)**.** An **induction head** is an attention head that implements a match-and-copy operation over sequences. Given an input context $[\ldots, A, B, \ldots, A]$, the mechanism attends from the second occurrence of A back to the token that followed the first occurrence (B), effectively "completing" the pattern by predicting B as the next token.

The algorithm is deceptively simple: when you see a token you've seen before, look at what followed it last time, and predict that it will follow again. This captures a fundamental regularity in language and structured data: patterns repeat.

But the simplicity of the algorithm belies the sophistication of its implementation. Induction heads don't merely memorize token sequences; they implement a general pattern-matching operation that works across arbitrary tokens. The same circuit that completes "Paris is in France. London is in" with "England" also completes "The function $f(x) = x^2$. The function $g(x) = x^3$. The function $h(x) ='' with“x^4$." The abstraction is not in what tokens are matched, but in the *structure* of the matching operation itself.

> ***Key Insight.*** *The power of induction heads lies not in memorization but in structural pattern matching. They implement the abstract operation "if you've seen A followed by B, and you see A again, predict B"—regardless of what A and B actually are. This is the seed of symbolic reasoning: operations defined over structural roles rather than specific content.*

2.3 The Two-Head Circuit

Induction heads require at least two layers to function, as they depend on *composition*—the output of one attention head feeding into another. This is a crucial architectural point: single-layer transformers cannot implement induction heads, which explains why in-context learning emerges only in multi-layer models.

2.3.1 The Previous Token Head

The first component is a "previous token head" that copies information from each token's predecessor into that token's representation. Mathematically, if we denote the embedding of token t at position i as x_i, the previous token head produces:

$$h_i^{(0)} = \text{Attn}^{(0)}(x_i) \approx x_{i-1} \tag{2.1}$$

This creates what we might call "preceded-by" features. Each token now carries information about what came before it, creating a richer representation that combines identity ("I am token B") with context ("I was preceded by token A").

The elegance of this construction is that it transforms a sequential problem into a parallel one. Rather than needing to search backwards through the sequence, the model has already embedded "what came before" into each position. The previous token head *preprocesses* the sequence for efficient pattern matching.

2.3.2 The Induction Head Proper

The second component uses the "preceded-by" information to identify pattern matches. When processing token A at position j, the induction head forms a query asking "Where in the sequence was something preceded by a token like me?", searches for positions where a token was *preceded by* A, attends strongly to those positions, and copies the token at those positions to predict the next token.

The key insight is that the induction head doesn't attend to tokens *equal to* the current token—that would just find the previous occurrence of A, which isn't useful. Instead, it attends to tokens that were *preceded by* tokens equal to the current token. This finds B, the token that followed A, which is exactly what we want to predict.

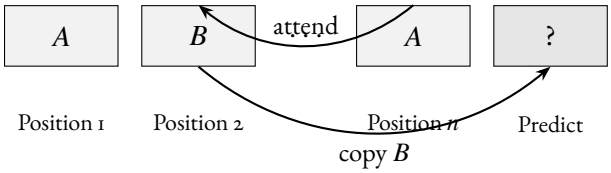

Figure 2.1: The induction head mechanism: attend to tokens preceded by the current token, copy their successors.

2.4 The Residual Stream: A Computational Bus

Before diving deeper into induction head mechanics, we must understand the architectural foundation that makes them possible: the *residual stream*. The transformer is best understood not as stacked layers but as a central information bus that all components read from and write to.

Each layer *adds* to this stream rather than replacing it:

$$\boxed{\mathbf{x}^{(l+1)} = \mathbf{x}^{(l)} + \text{Attn}^{(l)}(\mathbf{x}^{(l)}) + \text{MLP}^{(l)}(\cdot)} \tag{2.2}$$

This additive structure means information deposited by early layers remains accessible to later layers. A head in layer 2 can write information that a head in layer 20 reads. The model is a collaborative workspace, not a pipeline.

Expanding the residual stream reveals it as a superposition of all prior contributions:

$$\mathbf{x}^{(L)} = \mathbf{x}^{(0)} + \sum_{l=1}^{L} \sum_{h=1}^{H} \text{head}_{l,h}(\mathbf{x}^{(l-1)}) + \sum_{l=1}^{L} \text{MLP}_l(\cdot) \tag{2.3}$$

Understanding the model means understanding which terms matter for which outputs. This is why induction heads require composition across layers: the previous token head writes to the stream, and the induction head reads from it.

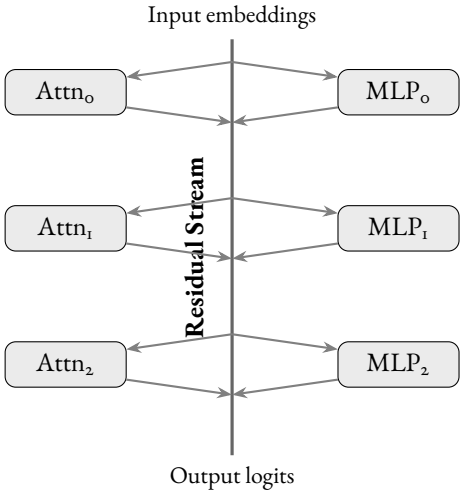

Figure 2.2: The residual stream paradigm: all components read from and write to a shared bus.

2.5 Inside an Attention Head: The QK and OV Circuits

To understand how induction heads work, we must look inside the attention mechanism itself. Each attention head performs two conceptually distinct computations that can be analyzed separately: determining *where* to look, and determining *what* to copy. These two operations are implemented by the **QK circuit** and the **OV circuit**, respectively.

2.5.1 The Two Questions Every Attention Head Answers

Think of an attention head as answering two questions at each position in the sequence:

1. **"Where should I look?"** — The head compares the current position against all other positions to decide which ones are relevant. This comparison produces *attention weights*: numbers between 0 and 1 indicating how much to focus on each position.
2. **"What should I copy?"** — Once the head knows where to look, it must decide what information to extract from those positions and how to transform it before adding it to the output.

The remarkable discovery from mechanistic interpretability research is that these two questions are answered by *largely independent* circuits within each head. This independence means we can understand each circuit separately, then combine our understanding to explain the head's overall behavior.

2.5.2 The QK Circuit: Computing Where to Attend

The first circuit determines the attention pattern—which positions in the sequence the head will focus on. At each position i, the head computes a **query vector** q_i that encodes "what I'm looking for." At every position j, it computes a **key vector** k_j that encodes "what I have to offer." The attention weight from position i to position j is determined by how well the query matches the key:

$$\text{Attention}_{i \to j} \propto \exp\left(\frac{q_i \cdot k_j}{\sqrt{d_k}}\right) \tag{2.4}$$

Both queries and keys are linear projections of the residual stream:

$$q_i = x_i W_Q \qquad\qquad k_j = x_j W_K \tag{2.5}$$

The attention computation can thus be written as a bilinear form:

$$\boxed{A_{ij} = \text{softmax}\left(\frac{x_i W_Q W_K^\top x_j^\top}{\sqrt{d_k}}\right)} \tag{2.6}$$

The matrix $W_{QK} = W_Q W_K^\top$ is the **QK circuit**. It defines a learned similarity function: two positions will attend to each other strongly if their representations, when projected through W_{QK}, have high inner product.

> ***Key Insight*** (What the QK Circuit Learns)***.*** *The eigenstructure of W_{QK} reveals what patterns the head has learned to detect:*
>
> - ***Large positive eigenvalues*** *indicate directions where queries and keys attract each other—the head attends when these features match.*

- ***Near-zero eigenvalues*** *indicate dimensions the head ignores entirely.*

For an induction head, the QK circuit learns to match the current token (in the query) against "preceded-by" information (in the keys), enabling pattern completion.

2.5.3 The OV Circuit: Computing What to Copy

Once the attention weights are computed, the head must extract and transform information from the attended positions. At each attended position j, the head computes a **value vector** $v_j = x_j W_V$ representing the information available there. These values are combined according to the attention weights and projected back into the residual stream:

$$\boxed{\text{head output}_i = \sum_j A_{ij} \cdot x_j W_V W_O} \tag{2.7}$$

The matrix $W_{OV} = W_V W_O$ is the **OV circuit**. It determines what information gets extracted from attended positions and how it's transformed before being written to the output.

The eigenstructure of W_{OV} classifies head behavior:

OV Eigenvalues	Head Behavior
Large positive	**Copying head**: reproduces attended content faithfully
Mixed signs	**Transformation head**: modifies or rotates information
Near-zero	**Suppression head**: blocks information flow

Table 2.1: The eigenvalue signature of the OV circuit reveals what the head does with attended information.

Key Insight (Induction Heads Require Copying)**.** *For induction to work, the head must faithfully reproduce the token that followed the matched pattern. This requires an OV circuit with large positive eigenvalues—a* ***copying*** *circuit. The in-*

> *duction head finds where B appeared (via the QK circuit) and copies B to the output (via the OV circuit).*

2.5.4 The Complete Picture

The full attention head operation combines both circuits:

$$\text{head}(X) = \underbrace{\text{softmax}\left(\frac{XW_QW_K^\top X^\top}{\sqrt{d_k}}\right)}_{\text{QK circuit: where to look}} \cdot \underbrace{XW_VW_O}_{\text{OV circuit: what to copy}} \tag{2.8}$$

This factorization is powerful because it separates *routing* (which positions communicate) from *content* (what gets communicated). The same attention pattern can move different information depending on the OV circuit, and the same information can be routed differently depending on the QK circuit. For induction heads, the QK circuit implements pattern matching while the OV circuit implements copying—two independent skills that combine to produce in-context learning.

2.6 Head Composition: Why Induction Requires Two Layers

We now arrive at the central question: how does the induction head know to attend to the token that *followed* A, rather than to A itself? The answer lies in **composition**—the ability of attention heads in different layers to work together through the shared residual stream. To understand this mechanism, we will trace the flow of information through the model step by step using a concrete example.

2.6.1 A Concrete Example: Harry the Wizard

Consider a real sequence: "`...Harry the wizard...Harry`". When the model reaches the second occurrence of "Harry", it must predict "the". It seems simple: find where "Harry" appeared before and copy what followed. But here is the fundamental problem that makes this task impossible for a single attention head.

2.6.2 The Problem with Single-Layer Induction

Consider the induction task: given the sequence $[A, B, \ldots, A]$, predict that B comes next. A naive approach would be to attend from the second A back to the first A and somehow extract B. But this fails for a fundamental reason:

> *The attention mechanism can only attend to positions, not to "the position after" some other position.*

To see why this matters, consider what each position's key represents in a single-layer model. In our example "Harry the wizard Harry":

- The **key** at position 0 ("Harry") represents "Harry"
- The **key** at position 1 ("the") represents "the"
- The **key** at position 2 ("wizard") represents "wizard"
- The **key** at position 3 (second "Harry") represents "Harry"

The problem becomes clear: to complete the pattern, we need to find the position of "the"—but we're not looking for positions that *contain* "the". We're looking for positions that *were preceded by* "Harry". The keys don't encode this information. A single attention head simply doesn't have access to the necessary information.

The query from position n asks: "Where is there a token like me?" The answer is position 0 (where the first A lives). But attending to position 0 gives us A, not B. We wanted the position *after* where A appeared.

2.6.3 The Solution: Enriching Keys with Predecessor Information

The solution is elegant: before the induction head runs, another head—the **previous token head**—modifies the residual stream so that each position's representation includes information about *what came before it*.

Step 1: The Previous Token Head (Layer 0)

The previous token head operates in Layer 0 with a seemingly trivial task: at each position, it attends entirely to the immediately preceding position and copies that token's

information into the residual stream. Mathematically:

$$\text{Attention}^{(0)}_{i \to j} = \begin{cases} 1 & \text{if } j = i - 1 \\ 0 & \text{otherwise} \end{cases} \tag{2.9}$$

With a copying OV circuit, this head writes the previous token's representation into the current position:

$$x_i^{(\text{after L0})} = x_i^{(\text{embed})} + \underbrace{x_{i-1}^{(\text{embed})} W_V^{(0)} W_O^{(0)}}_{\text{"preceded by } x_{i-1}\text{"}} \tag{2.10}$$

Consider what happens to our sequence "Harry the wizard Harry" after this layer:

Position	**Before Layer 0**	**After Layer 0**
0 (Harry)	only "Harry"	"Harry" + preceding token
1 (the)	only "the"	"the" + **"Harry preceded me"**
2 (wizard)	only "wizard"	"wizard" + "the preceded me"
3 (Harry)	only "Harry"	"Harry" + preceding token

This change is crucial: the residual stream at the position of "the" now contains not only information about "the", but also information about "Harry"—the token that preceded it. This additional information will be readable by the next head.

Step 2: The Induction Head (Layer 1)

The second head can now do something that was impossible before. When it constructs **keys**, it reads from the residual stream that now contains information about the preceding token. When it constructs the **query**, it encodes the current token ("Harry"). Here's what happens:

1. **Key Construction**: The key at position 1 now encodes "the, preceded by Harry", not just "the"
2. **Query Construction**: The query at position 3 (second "Harry") asks "find posi-

tions preceded by Harry"

3. **QK Matching**: The query from position 3 is compared against all keys:
   - Query pos. 3 × Key pos. 1 = **HIGH** (match on "preceded by Harry")
   - Query pos. 3 × Key pos. 2 = low (no match)
4. **OV Copying**: Attention focuses on position 1. The OV circuit copies "the" → Correct prediction!

The key insight is that the **keys** now encode "what token preceded me", not just "what token I am". This transforms the problem: instead of searching for where a token appears, we can search for what *followed* that token.

After Layer 0, each position carries two pieces of information: its own identity *and* its predecessor's identity. Position 1 (containing B) now represents both "I am B" and "I was preceded by A."

2.6.4 K-Composition: How the Pieces Fit Together

When the induction head in Layer 1 computes keys, it reads from this enriched residual stream. The key at position 1 is no longer just "B"—it's "B, preceded by A." This is **K-composition**: the output of the previous token head affects the keys that the induction head uses.

Definition 2.2 (K-Composition)**.** K-composition occurs when an earlier head's output influences a later head's key computation:

$$K^{(L1)} = \left(x^{(\text{embed})} + \text{head}^{(L0)}(x)\right) W_K^{(L1)} \tag{2.11}$$

The keys now encode information written by the earlier head, enabling queries to match against features that weren't present in the original embeddings.

Now the induction head can succeed:

1. The query at position n (second A) encodes: "I am A; where was something preceded by A?"
2. The key at position 1 encodes: "I am B; I was preceded by A."
3. The QK circuit matches these—position 1's key advertises exactly what position n's query seeks.

4. The OV circuit copies *B* from position 1 to the output.

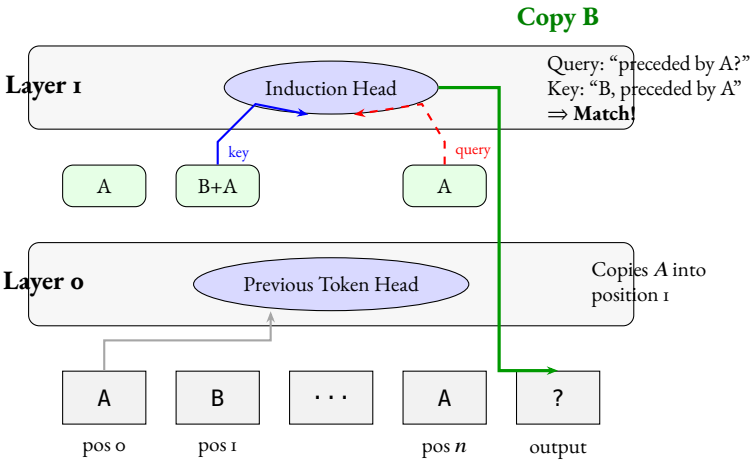

Figure 2.3: The two-layer induction circuit. Layer 0's previous token head enriches each position with predecessor information. Layer 1's induction head uses K-composition to search for positions preceded by the current token, then copies what it finds.

> ***Key Insight*** (The Necessity of Depth)**.** *Induction heads cannot exist in single-layer transformers. The mechanism requires two distinct operations that must happen in sequence:*
>
> 1. ***Write**: The previous token head writes predecessor information into the residual stream.*
> 2. ***Read**: The induction head reads this information through its key computation.*
>
> *Since both operations require attention, and a single layer has only one attention step, two layers are the minimum. This explains why the "phase change" in in-context learning—when models suddenly acquire the ability to learn from examples in context—coincides with the formation of these two-layer circuits during training.*

2.6.5 The Full Mathematical Picture

For completeness, we can write the induction head's attention pattern in Layer 1 as:

$$A_{n,j}^{(L1)} = \text{softmax}\left(\frac{q_n \cdot k_j}{\sqrt{d}}\right) \tag{2.12}$$

where the query and key incorporate the previous token head's contribution:

$$q_n = \left(x_n + \text{head}^{(L0)}(x)_n\right) W_Q^{(L1)} \tag{2.13}$$

$$k_j = \left(x_j + \text{head}^{(L0)}(x)_j\right) W_K^{(L1)} \tag{2.14}$$

When $\text{head}^{(L0)}$ is a previous token head, the key at position j contains information about both token j and token $j-1$. The induction head's QK circuit is trained to produce high attention when the query token matches the key's predecessor component—precisely the condition for successful pattern completion.

2.6.6 The Three Types of Composition

K-composition is just one of three ways attention heads can collaborate across layers. Understanding all three composition types illuminates why deep transformers are so powerful.

K-Composition: Modifying What Gets Searched in Keys We've already seen how K-composition enables induction heads to function. An earlier head writes information to the residual stream, and this information becomes part of the keys that a later head uses to decide where to attend. Intuitively: K-composition allows "tagging" positions with additional information that can then be searched for.

In the induction example, the previous token head "tags" each position with "I was preceded by X", enabling the induction head to search for positions with the right tag.

Q-Composition: Modifying What Is Being Searched For Q-composition is the mirror of K-composition. Instead of modifying the labels being searched, it modifies the search itself. An earlier head can write information that changes what a later head is looking for.

This enables context-dependent queries. For example, in a sentence like "The cat

that the dog chased meowed," the query for determining the subject of "meowed" might be modified by information about syntactic structure processed by earlier heads. The model doesn't simply search for "the nearest noun," but for "the noun that is the main subject."

V-Composition: Modifying What Gets Transferred V-composition influences what actually gets extracted and copied once attention has been allocated. Earlier heads can enrich the representations at source positions, so that when a later head attends to that position, it extracts richer information than was originally present.

Anthropic's research found that V-composition contributes less than Q and K-composition to overall accuracy in the models studied. However, it enables "virtual attention heads"—cases where the combined effect of multiple heads can be equivalent to a single, much more complex head.

> ***Key Insight*** (Why Depth Matters)**.** *Each additional layer multiplies composition possibilities. With two layers, we can have simple K, Q, and V-composition. With three layers, compositions can chain. This explains why deeper models exhibit qualitatively different capabilities: it's not just that they have more parameters, but that they can express fundamentally more complex computational patterns through head composition.*

2.7 The Fundamental Equation of Prompt Design

Now that we understand how the QK circuit determines *where* attention flows and the OV circuit determines *what* gets copied, we can derive a fundamental principle for prompt engineering:

$$\underbrace{\text{Prompt Structure}}_{\text{shapes keys } \boldsymbol{K}} \xrightarrow{\text{QK circuit}} \underbrace{\text{Attention Pattern}}_{\text{what attends to what}} \xrightarrow{\text{OV circuit}} \underbrace{\text{Output}}_{\text{what gets produced}} \tag{2.15}$$

Prompt structure shapes attention, and attention shapes output. When you

structure your prompt in a particular way, you're literally shaping the key representations that the QK circuit will match against. Design prompts that create clear, consistent patterns—this works *with* the model's computation rather than against it.

2.7.1 Why Parallel Structure Matters

Consider the induction head mechanism: it looks for patterns of the form `[A][B]...[A]` and predicts `B`. The QK circuit matches the current position's query against keys at all previous positions. For this to work reliably, the keys need to be consistent—when the same structural role appears multiple times, it should produce similar key representations.

Practical Leverage (Weak vs Strong Pattern Structures)**.** Compare these two prompts asking for capitals:

Weak (structure buried in prose):

```
The capital of France is Paris.  Germany has Berlin
as its capital.  What about Japan?
```

Strong (parallel structure):

```
France ::  Paris
Germany ::  Berlin
Japan ::  ?
```

In the weak version, the relationship "country → capital" appears in different syntactic positions with different surrounding words. The keys are inconsistent.

In the strong version, each example has identical structure: `[Country] :: [Capital]`. The induction head sees: "When `::` follows a country name, a capital follows. I see `Japan ::`—predict what comes next." The parallel structure creates matching key representations, making the pattern trivial to detect.

2.7.2 Practical Prompt Patterns

Practical Leverage (Few-Shot with Consistent Delimiters)**.** Use consistent formatting so that the pattern `[A][B]...[A]` is unambiguous:

```
Input: cat | Output: animal
Input: hammer | Output: tool
Input: salmon | Output:
```

The consistent "`Input: X | Output: Y`" format creates clear pattern boundaries. The induction head can match "what follows `Output:` after `Input: [word] |`" and copy the appropriate token type.

Practical Leverage (Leveraging the Copying OV Circuit)**.** Remember that the OV circuit in induction heads is a *copying* circuit—it reproduces attended content with minimal transformation. Structure prompts to take advantage of this:

```
Q: What is 2+2?
A: 4

Q: What is 3+3?
A: 6

Q: What is 5+5?
A:
```

The model doesn't need to "understand" arithmetic here. The induction head sees the pattern "`Q:` followed by question, then `A:` followed by answer" and copies the structural pattern. The consistent `Q:/A:` format creates clear attention targets for the copying behavior.

Practical Leverage (Code and Function Patterns)**.** For code patterns, structure examples to make the induction pattern explicit:

```
def double(x): return x * 2
def triple(x): return x * 3
def quadruple(x): return x * 4
```

```
def quintuple(x):
```

The induction head matches the pattern "`def [name](x): return x * ` followed by number" and predicts the structural completion. The function name provides context; the multiplier follows the pattern.

Practical Leverage (Explicit Variable Roles for Complex Patterns)**.** For patterns requiring multiple steps, make variable roles explicit:

```
PATTERN: [Subject] [Verb] [Object]. Therefore [Subject]
[Result].

Example 1: Alice studies mathematics. Therefore Alice
knows mathematics.
Example 2: Bob practices guitar. Therefore Bob plays
guitar.

Apply: Carol reads philosophy. Therefore
```

By declaring the pattern explicitly and using consistent variable roles, you engage both the symbol abstraction mechanism (recognizing that "Alice", "Bob", and "Carol" play the same `[Subject]` role) and the induction mechanism (completing the pattern based on prior examples).

Practical Leverage (Document Classification)**.** For classifying documents in professional contexts:

```
Document: lease agreement between John Smith and XYZ
Corp
Category: legal/contractual

Document: invoice #2024-0123 for material supplies
Category: accounting/fiscal

Document: meeting minutes from January 15, 2024 board
```

```
meeting
Category: administrative

Document: lost passport report filed with local police
Category:
```

The structure `Document: [description]` → `Category: [type]` creates a clear pattern. The induction head learns that after `Category:` a classification term always follows, and the symbol abstraction heads recognize the relationship between document content and its category.

Practical Leverage (Structured Entity Extraction)**.** For extracting information from text into a consistent format:

```
Text: "Attorney Mary Johnson represented ABC Inc.
in the trial on March 12, 2024."
Entities: {person: "Mary Johnson", role: "attorney",
organization: "ABC Inc.", date: "March 12, 2024"}

Text: "On February 5th, Dr. James Lee submitted the
proposal to the National Science Foundation."
Entities: {person: "James Lee", role: "doctor",
organization: "National Science Foundation", date:
"February 5th"}

Text: "CEO Sarah Chen of TechStart LLC signed the
partnership agreement on January 20."
Entities:
```

The consistent JSON format for extracted entities leverages both the copying circuit (to reproduce exact names from the text) and pattern matching (to recognize which text elements correspond to which fields).

Practical Leverage (Logical Transformations)**.** For applying consistent transformations:

```
Original:  "The system automatically verifies the data."
Passive:  "The data is automatically verified by the
system."

Original:  "The operator enters the information into
the database."
Passive:  "The information is entered into the database
by the operator."

Original:  "The software generates daily reports."
Passive:
```

Here the pattern involves transformation rather than simple copying, but the consistent structure allows the model to learn the transformation rule from the examples.

2.7.3 The Design Principle

The core insight is simple: **make patterns easy for the QK circuit to match**. This means:

1. **Consistent structure**: Use the same format for all examples
2. **Clear delimiters**: Make boundaries between pattern elements unambiguous
3. **Explicit roles**: When patterns involve variable substitution, make the roles clear
4. **Sufficient examples**: Provide enough examples for the pattern to be unambiguous

When you design prompts this way, you're not fighting the model's architecture—you're leveraging it.

2.8 The Phase Change Phenomenon

One of the most striking findings about induction heads is their relationship to training dynamics. Olsson et al. (2022) discovered that induction heads develop during a specific "phase change" in training—a sudden, dramatic transition rather than gradual improvement.

This phase change manifests as:

- A distinctive "bump" or plateau in the loss curve, where loss temporarily increases before resuming its decline
- Rapid formation of induction head circuits measurable through attention pattern analysis
- Dramatic improvement in in-context learning performance within a narrow window of training
- Changes in how the model processes repeated patterns—from memorization to generalization

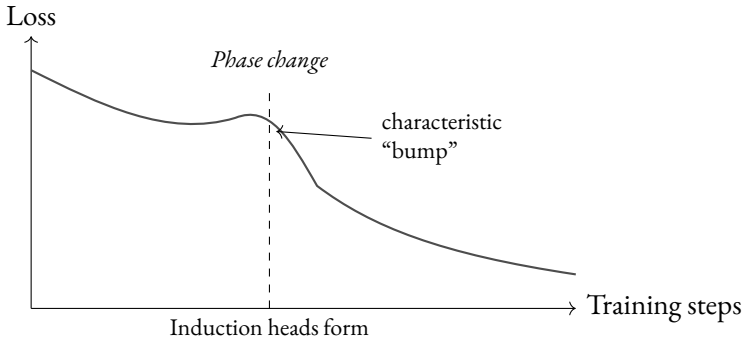

Figure 2.4: Training loss shows a characteristic bump when induction heads form. This phase change marks the sudden emergence of in-context learning capabilities.

The synchronicity of these events provides strong evidence that induction heads are *causally responsible* for in-context learning capabilities. It's not merely that induction heads and ICL happen to emerge together; the formation of induction heads *causes* the improvement in ICL.

Key findings about the phase change:

- In-context learning ability and induction head formation happen **together, suddenly, irreversibly**
- Ablating induction heads **causes** in-context learning to degrade (causal, not just correlational)
- The "smeared keys" experiment showed: when previous-token info is made available architecturally, the phase change happens earlier

> ***Key Insight.*** *The phase change phenomenon suggests that transformer capabilities emerge through circuit formation—the crystallization of computational structures that implement specific algorithms. Capabilities don't gradually get better; they suddenly "click into place" when the right circuit forms. This has profound implications for both capability prediction and safety: dangerous capabilities might emerge just as suddenly as beneficial ones.*

2.9 From Exact to Fuzzy Matching

The basic induction head performs exact token matching: $[A][B] \ldots [A] \rightarrow [B]$. But what about cases where there's no exact match? This is where *fuzzy* or *generalized* induction heads come into play:

Definition 2.3 (Fuzzy Induction)**.** A **fuzzy induction head** performs pattern completion $[A^*][B^*] \ldots [A] \rightarrow [B]$, where $A^* \approx A$ and $B^* \approx B$ are similar in some embedding space, rather than requiring exact matches.

The transition from exact to fuzzy matching is crucial for understanding how models generalize. Consider:

```
The capital of France is Paris.
The capital of Germany is Berlin.
The capital of Japan is
```

Exact matching would fail: "Japan" doesn't appear earlier in the context. But fuzzy matching succeeds: "Japan" is similar to "France" and "Germany" (all are countries), and the structural pattern [country] → [capital] generalizes.

This generalization happens because the model's embeddings place semantically similar items close together in vector space. When the induction head's query looks for "something preceded by a token like Japan," it finds approximate matches in positions preceded by other country names.

> **Practical Leverage.** To leverage fuzzy induction, provide diverse examples that span the semantic space of your target:
>
> ```
>    tiger -> mammal
> eagle -> bird
> salmon -> fish
> cobra -> reptile
> frog -> ?
> ```
>
> Even though "frog" hasn't appeared, fuzzy matching activates because "frog" is semantically similar to the other animals, and the pattern [animal] → [class] generalizes.

Chapter 3

The Three-Stage Symbolic Architecture

> *"In early layers, symbol abstraction heads convert input tokens to abstract variables based on the relations between those tokens. In intermediate layers, symbolic induction heads perform sequence induction over these abstract variables. In later layers, retrieval heads predict the next token by retrieving the value associated with the predicted abstract variable."*
>
> — Yang et al., 2025

3.1 The Architecture of Abstraction

The mechanisms we have explored so far—induction heads that complete patterns, function vectors that encode tasks—are remarkable discoveries. Yet they are pieces of a larger puzzle. Recent research has revealed the complete picture: a three-stage architecture that implements genuine symbolic processing within the neural substrate of language models.

This architecture operates through a division of labor across the network's layers. Early layers house *symbol abstraction heads* that convert concrete tokens into abstract variable representations. Middle layers contain *symbolic induction heads* that perform

pattern completion over these abstractions. Late layers deploy *retrieval heads* that map abstract predictions back to concrete tokens. Together, these three stages implement a computational pipeline that bears striking resemblance to classical symbolic AI—variable binding, rule application, symbol manipulation—yet emerges entirely from the statistics of language.

The discovery of this architecture represents the most complete account we have of how language models implement reasoning. It unifies earlier findings about induction heads and function vectors into a coherent framework. And it provides the mechanistic grounding for the practical techniques we will explore in later chapters.

3.2 Symbol Abstraction Heads

The first stage of the pipeline converts tokens into abstract variable representations. This is the work of symbol abstraction heads, found in the early layers of the transformer.

Consider what happens when a language model processes the sequence "CAT DOG CAT." A human immediately recognizes the pattern: the first and third elements are the same, the second is different. We might call this an "ABA" pattern, using letters to denote abstract positions rather than specific tokens. Now consider a different sequence: "RED BLUE RED." The tokens are entirely different, yet the pattern is identical—also ABA. A reasoning system that can recognize these patterns must operate not on the tokens themselves but on something more abstract.

Symbol abstraction heads create this abstraction. When they process "CAT DOG CAT," they produce an internal representation that captures the relational structure: $[\text{VAR}_1, \text{VAR}_2, \text{VAR}_1]$. Crucially, when they process "RED BLUE RED," they produce the same representation: $[\text{VAR}_1, \text{VAR}_2, \text{VAR}_1]$. The specific tokens have been abstracted away; only the pattern remains.

Definition 3.1 (Symbol Abstraction Heads)**.** Symbol abstraction heads are attention heads in early layers that convert input tokens into abstract variable representations based on relational patterns, rather than token identities.

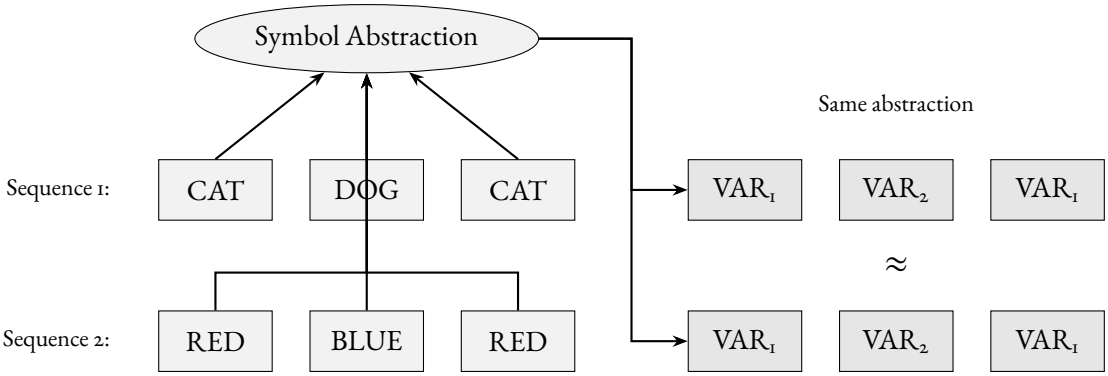

Figure 3.1: Symbol abstraction heads map different token sequences to the same abstract representation when they share relational structure. Both sequences exhibit the ABA pattern and receive identical variable assignments.

The key property of these heads is *invariance*: their output representations remain consistent regardless of which specific tokens fill the variable roles. This invariance is what makes the representations genuinely symbolic. A symbol, in the classical AI sense, is something that stands for something else, that can be manipulated without reference to what it stands for. The variable representations produced by symbol abstraction heads have exactly this property.

$$\text{Abstraction}(\text{“CAT DOG CAT”}) \approx \text{Abstraction}(\text{“RED BLUE RED”}) \tag{3.1}$$

In large models like Llama-3.1 70B, symbol abstraction heads are concentrated in layers 1 through 15. They process not token content but relational structure, determining which tokens play the same role (and should map to the same variable) and which play different roles. The computation resembles what the *abstractor* architecture—a transformer variant explicitly designed for relational reasoning—accomplishes through specialized cross-attention. But here, the capability has emerged without explicit design.

3.3 Symbolic Induction Heads

Once tokens have been abstracted into variables, pattern completion can operate at the abstract level. This is the work of symbolic induction heads, found in the middle layers of the transformer.

Standard induction heads, as we explored in chapter 2, complete patterns over concrete tokens. Given "cat sat ... cat," they predict "sat" because that is what followed "cat" before. Symbolic induction heads do something more general. Given a sequence whose abstract structure is $[\text{VAR}_1][\text{VAR}_2]\ldots[\text{VAR}_1]$, they predict $[\text{VAR}_2]$—not a specific token, but a variable that must be resolved to a token in the final stage.

Definition 3.2 (Symbolic Induction Heads)**.** Symbolic induction heads are attention heads in intermediate layers that perform pattern completion over abstract variables, not concrete tokens.

This distinction is subtle but profound. Standard induction requires exact token matching; if the previous context contained "Cat" (capitalized) but the current token is "cat" (lowercase), the pattern might fail to match. Symbolic induction operates above such surface variation. It recognizes that two positions play the same role in a pattern regardless of the specific tokens that instantiate them. It can complete patterns across variations that would defeat token-level matching.

The empirical evidence for this distinction is striking. Symbolic induction heads show only weak correlation ($r = 0.11$) with standard induction heads, despite performing a superficially similar operation. They are implementing something genuinely different, not merely a variant of the same computation. Yet they show strong correlation ($r = 0.86$) with function vectors, suggesting a deep connection between task representation and symbolic pattern completion.

> ***Key Insight.*** *The correlation between symbolic induction heads and function vectors ($r = 0.86$) illuminates the nature of both mechanisms. Function vectors encode abstract task specifications that can be applied across contexts. Symbolic induction heads apply abstract pattern rules across token instantiations. Both operate at a level above concrete tokens; both exhibit generalization that token-level mechanisms*

> *cannot achieve. The high correlation suggests they are, in some sense, the same phenomenon viewed from different angles—function vectors are what symbolic induction heads* compute.

The key property of symbolic induction heads is *indirection*: they manipulate variables that reference content elsewhere, without being bound to specific values. This is the computational analog of classical symbolic AI's variable binding. A rule like "if the pattern is ABA, predict the token in position B" can be stated and applied without knowing what tokens A and B actually are. The binding is deferred; the rule is abstract.

In large models, symbolic induction heads are concentrated in layers 15 through 45. They receive the abstract variable representations produced by symbol abstraction heads, apply pattern-completion logic, and produce predictions in terms of variables—predictions that must still be grounded in concrete tokens.

3.4 Retrieval Heads

The final stage of the pipeline converts abstract predictions back to concrete tokens. This is the work of retrieval heads, found in the late layers of the transformer.

Symbolic induction has predicted that the next element should be VAR_2. But what token does VAR_2 represent in the current context? The retrieval stage answers this question by looking up the binding: finding the position in the context that was assigned VAR_2 and copying its token to the output.

Definition 3.3 (Retrieval Heads)**.** Retrieval heads are attention heads in later layers that map predicted abstract variables back to concrete tokens by retrieving the values associated with those variables in the current context.

Mathematically, retrieval heads compute:

$$\boxed{\text{output} = \sum_{i:\text{role}(i)=\text{VAR}_k} A_i \cdot \mathbf{x}_i \cdot W_{OV}} \tag{3.2}$$

where A_i is the attention weight to position i, concentrated on positions whose abstract role matches the predicted variable.

This retrieval completes the pipeline. Information flows from concrete tokens through abstract variables back to concrete tokens:

$$\text{Input Tokens} \xrightarrow{\text{Abstraction}} \text{Variables} \xrightarrow{\text{Induction}} \text{Predicted Variable} \xrightarrow{\text{Retrieval}} \text{Output Token} \tag{3.3}$$

In large models, retrieval heads are concentrated in layers 45 and beyond. They attend to positions in the context that correspond to the predicted variable and copy the tokens found there. The mechanism resembles copying heads from earlier in training, but applied to variable-based rather than position-based selection.

3.5 Experimental Evidence

How do we know these mechanisms exist? The evidence comes from *causal mediation analysis*—techniques that go beyond correlation to establish what actually causes model behavior.

The key experimental design uses paired contexts that share an output but differ in abstract structure:

- Context A: "CAT DOG CAT ?" (ABA pattern, expects DOG)
- Context B: "CAT DOG DOG ?" (ABB pattern, also expects DOG)

Both produce the same answer but via different abstract reasoning. The intervention then *patches* activations from Context A into Context B at specific heads. If the head carries abstract pattern information, the model should now "see" ABA despite the tokens spelling ABB.

The experiments confirm this prediction. Patching at symbol abstraction heads causes systematic changes in model output, demonstrating that these heads carry the abstract pattern information that determines reasoning. Similar experiments identify the causal role of symbolic induction heads (which use the abstract representations) and retrieval heads (which convert predictions back to tokens).

Representational similarity analysis provides complementary evidence. If symbol

abstraction heads truly produce invariant representations, then contexts with the same abstract pattern should have similar activations, while contexts with different patterns should have dissimilar activations—regardless of which tokens appear. The data confirm this prediction: within-pattern similarity exceeds across-pattern similarity, establishing that these heads encode abstract structure rather than token content.

3.6 Quantitative Results

The three-stage architecture enables impressive performance on tasks requiring abstract reasoning.

Task	Model	2-Shot Accuracy
ABA/ABB Rule Induction	Llama-3.1 70B	95%
Letter Successor	Llama-3.1 70B	99.2%
Letter Predecessor	Llama-3.1 70B	82.0%
Synonym Analogies	Llama-3.1 70B	77.0%
Antonym Analogies	Llama-3.1 70B	88.4%

Table 3.1: Task performance demonstrating symbolic reasoning capabilities in large language models.

On the ABA/ABB task—determining which abstract pattern a sequence exhibits—models achieve 95% accuracy with just two examples. On letter successor tasks (predicting the next letter in alphabetical sequences), accuracy reaches 99.2%. Performance on semantic tasks like analogies is lower but still substantial, suggesting that the symbolic mechanisms engage even when the "patterns" involve learned semantic relationships rather than simple structural repetition.

Crucially, these mechanisms appear across multiple model families. Llama-3.1, Qwen2.5, and Gemma-2 all exhibit the three-stage architecture, despite being trained by different organizations on different data. This cross-model validation suggests that the architecture is not an artifact of particular training choices but rather a convergent solution that emerges when transformers are trained at scale on language.

3.7 Leveraging Symbolic Architecture in Practice

Understanding the three-stage architecture transforms how we design prompts and structure tasks for LLMs.

3.7.1 Activate Symbol Abstraction with Explicit Variables

Symbol abstraction heads convert tokens to variables based on relational structure. Help them by making that structure explicit:

> **Implicit structure** (model must discover the pattern):
>
> ```
> cat sat mat
> dog ran fan
> bird flew ?
> ```
>
> **Explicit structure** (activates abstraction directly):
>
> ```
> Pattern:  [X] [action] [rhyme-with-X]
>
> cat sat mat
> dog ran fan
> bird flew ?
> ```
>
> Naming the pattern with variables (X, action, rhyme-with-X) primes the model's symbol abstraction stage.

3.7.2 Design for Pattern Generalization

Symbolic induction operates on abstract variables, not tokens. Your examples should highlight the *relationship* that generalizes:

> **Teaching an abstract rule**:
>
> ```
> Rule:  Given pattern [A][B][A], predict [B]
>
> red blue red -> blue
> ```

```
cat dog cat -> dog
1 2 1 -> 2

happy sad happy -> ?
```

The varied instantiations (colors, animals, numbers) force the model to abstract the pattern rather than memorize tokens.

3.7.3 Anchor Retrieval with Clear Bindings

Retrieval heads must map abstract predictions back to concrete tokens. Make the binding unambiguous:

Ambiguous binding (retrieval may fail):

```
John met Mary.  Mary met Susan.  John met ?
```

("Mary" appears twice with different roles)

Clear binding:

```
Person A: John
Person B: Mary
Interaction:  A met B

Person A: Mary
Person B: Susan
Interaction:  A met B

Person A: John
Person B: ?
Interaction:  A met B
```

Explicit variable names (A, B) create unambiguous retrieval targets.

Key Insight (The Three-Stage Prompt Template)***.*** *Design prompts that support all three stages:*

1. ***Abstraction**: Name the variables or pattern explicitly*
2. ***Induction**: Show multiple instantiations of the same abstract rule*
3. ***Retrieval**: Ensure each variable has a unique, unambiguous binding*

3.8 Key Properties of Symbolic Mechanisms

The three-stage symbolic architecture exhibits several fundamental properties that distinguish it from both purely token-level pattern matching and classical symbolic AI. Understanding these properties is essential for leveraging the mechanisms effectively.

Principle 3.1 (Invariance)**.** Symbol abstraction heads produce representations that remain stable regardless of the specific tokens that fill structural roles. Formally:

$$\forall \text{ sequences } S_1, S_2 \text{ with same structure} : \text{Abstraction}(S_1) \approx \text{Abstraction}(S_2) \tag{3.4}$$

This invariance is the defining property of symbolic representation: the ability to represent structure independently of content. It enables transfer across domains—a pattern learned with colors applies equally to animals, numbers, or any other category.

The invariance property explains why diverse examples improve model performance: they force the abstraction mechanism to find what is truly invariant (the structural pattern) rather than what happens to be common across a narrow sample (surface features).

Principle 3.2 (Compositionality)**.** Complex symbolic operations can be built from simpler ones through systematic combination. The three-stage architecture itself exhibits compositionality: abstraction, induction, and retrieval are independent operations that compose to produce symbolic reasoning.

This compositionality extends to the function level: function vectors can be added to produce compound functions, and cognitive tools can be sequenced to produce complex reasoning chains. The underlying principle is that representations preserve enough structure to support meaningful combination.

Compositionality is what makes the symbolic mechanisms genuinely productive: a finite set of basic operations can generate an infinite variety of complex behaviors. The same abstraction heads, induction heads, and retrieval heads that solve simple ABA patterns also contribute to sophisticated analogical reasoning.

Principle 3.3 (Indirection)**.** Symbolic induction operates through indirection: manipulating references to content rather than content itself. Variables are pointers that get resolved in the retrieval stage. This separation of reference from referent is what enables abstract pattern completion.

The indirection mechanism works through what the research calls "binding IDs"—vector representations that track which positions in the context correspond to which abstract roles. When symbolic induction predicts VAR_2, it is predicting a binding ID, not a token. Retrieval heads then resolve this ID to the appropriate token.

The indirection property connects symbolic mechanisms in LLMs to classical notions of variable binding in programming and logic. The key difference is that these bindings emerge from training rather than being explicitly programmed—they are distributed representations that function like symbols without being symbols in the classical sense.

These three properties—invariance, compositionality, and indirection—together characterize what makes the emergent mechanisms genuinely symbolic. They are not merely statistical patterns over tokens; they implement abstract operations over structural roles. This is why understanding these mechanisms transforms how we work with language models: we can design prompts that respect these properties, activating the symbolic machinery rather than working against it.

Chapter 4

Function Vectors: Capturing Procedural Knowledge

> *"Despite being extracted from templated prompts, function vectors transfer effectively to different contexts, including natural text that differs significantly from training examples."*
>
> — Todd et al., 2024

4.1 From Patterns to Procedures

The mechanisms described so far—induction heads and symbolic architecture—enable pattern recognition and completion. But in-context learning encompasses more than patterns: models can learn entire *procedures* from demonstration.

Consider a prompt that demonstrates capitalization:

```
hello -> HELLO
world -> WORLD
python ->
```

The model doesn't just recognize a pattern; it learns a *procedure*—the operation of converting to uppercase. This procedural knowledge can be applied to any input, not

just those similar to the examples.

Function vectors capture this procedural knowledge in a compact, transferable form:

Definition 4.1 (Function Vector)**.** A **function vector** (FV) is a compact representation of a demonstrated function, extracted from language model hidden states during in-context learning. An FV can trigger execution of a specific procedure when injected into model computations.

4.2 Extraction and Properties

Function vectors are extracted through causal analysis:

1. Present the model with ICL prompts demonstrating a function (e.g., antonym, uppercase, category)
2. Identify attention heads that causally contribute to task resolution (via ablation studies)
3. Extract their task-conditioned average activations
4. The resulting vector represents the demonstrated function

The remarkable property of function vectors is their *transferability*: a function vector for "antonym" extracted from a formatted few-shot prompt can be injected into casual conversation and still produce antonyms. The function is context-independent—a genuine representation of the operation itself, not just a statistical regularity of the prompt.

4.3 Compositionality of Function Vectors

Function vectors exhibit semantic algebra properties:

Theorem 4.1 (Function Vector Algebra)**.** Function vectors can be composed to create compound functions. Vector addition often produces meaningful combinations of the component functions.

This is analogous to the famous "king - man + woman = queen" property of word

embeddings, but at the level of *functions* rather than words. For example:

- FV(antonym) + FV(capitalize) may produce behavior that generates capitalized antonyms
- FV(past tense) + FV(negate) may produce negated past tense transformations

The compositionality suggests that function vectors capture something fundamental about how the model represents operations—not as opaque procedures, but as composable elements in a space of transformations.

4.4 Relationship to Symbolic Mechanisms

Function vectors provide a higher-level view of the same phenomena:

- **Symbolic mechanisms** explain *how* abstract reasoning works (the three-stage architecture)
- **Function vectors** provide *tools* for extracting and manipulating the results of that reasoning

The high correlation ($r = 0.86$) between function vectors and symbolic induction heads suggests they capture related aspects of the model's computational structure. Function vectors may be the "output" of symbolic processing, packaged in a form that can be transferred and applied elsewhere.

> ***Key Insight.*** *Think of function vectors as "compiled procedures." When the model learns a task from few-shot examples, it constructs a function vector that encodes the learned operation. This vector can then be applied to new inputs without repeating the learning process. The symbolic architecture is the "compiler"; function vectors are the "compiled code."*

Part II

External Orchestration

Chapter 5

Cognitive Tools

> *"Providing our 'cognitive tools' to GPT-4.1 increases its pass@1 performance on AIME2024 from 32% to 53%, even surpassing the performance of o1-preview."*
>
> — Ebouky et al., 2025

5.1 From Internal Mechanisms to External Orchestration

The previous chapters have explored the symbolic mechanisms that emerge within language models: attention heads that abstract tokens into variables, heads that perform pattern completion over these abstractions, heads that retrieve concrete predictions from abstract representations. These mechanisms exist and operate within the model's forward pass, activated by the statistical patterns of the input.

But knowing that these mechanisms exist raises a question: can we engage them more effectively? Can we design external interventions that leverage the internal machinery to achieve better reasoning performance? The cognitive tools framework answers yes. By providing language models with structured operations for decomposition, verification, abstraction, and other cognitive functions, researchers have achieved substantial improvements on challenging reasoning tasks—improvements that likely arise from more effectively engaging the symbolic mechanisms already present within these models.

This approach represents a different philosophy from recent reasoning-enhanced models like o1 and R1, which achieve deliberate reasoning through extensive reinforcement learning on chain-of-thought traces. Cognitive tools achieve similar gains without additional training, suggesting that the capability for deliberate reasoning was present all along; it merely needed the right scaffolding to emerge.

5.2 The Cognitive Architecture Perspective

The cognitive tools framework draws on decades of research in cognitive psychology, where scientists have long theorized that human cognition relies on a fixed set of mental operations that can be combined to perform any cognitive task.

This view is captured in the concept of a *cognitive architecture*: a theory of the underlying mechanisms that remain constant across different tasks and knowledge domains. Just as computer architecture provides a fixed set of operations (arithmetic, memory access, branching) that can be combined to implement any algorithm, cognitive architecture provides a fixed set of mental operations (attention, retrieval, comparison, transformation) that can be combined to implement any cognitive process.

The relevance to language models is immediate. If human cognition works through modular operations, and if language models have learned to mimic human cognition through training on human-generated text, then language models may benefit from explicit modular structuring of their reasoning processes. The cognitive tools framework tests this hypothesis by providing explicit operations that the model can invoke, each designed to perform a specific cognitive function.

The framework also connects to Daniel Kahneman's influential distinction between System 1 and System 2 thinking. System 1 is fast, automatic, and intuitive—the kind of processing that recognizes faces, reads emotions, and makes snap judgments. System 2 is slow, deliberate, and analytical—the kind of processing that solves algebra problems, plans complex actions, and checks logical arguments.

Standard language model inference resembles System 1. The model receives an input, processes it through a single forward pass, and produces an output. The computation is fast and pattern-based, relying on learned associations rather than explicit

deliberation. Cognitive tools aim to activate something more like System 2 reasoning, enforcing deliberate step-by-step processing where each step is a distinct cognitive operation whose result is explicitly recorded and evaluated.

5.3 The Tool Architecture

A cognitive tool, in this framework, is a self-contained function that encapsulates a specific reasoning operation. When the model needs to decompose a problem into subproblems, it invokes the DECOMPOSE tool. When it needs to verify a proposed solution, it invokes the VERIFY tool. Each invocation runs in a sandboxed context, isolated from the main reasoning state, ensuring that the tool performs its designated function without contamination from other considerations.

Definition 5.1 (Cognitive Tool)**.** A cognitive tool is a self-contained function that encapsulates a specific reasoning operation. It is executed by the LLM itself, takes structured input, produces structured output, and runs in a sandboxed context isolated from the main reasoning loop.

The core tools in the framework correspond to fundamental cognitive operations identified in the psychology literature:

DECOMPOSE breaks a problem into independent subproblems. Complex problems often resist direct solution but yield to divide-and-conquer strategies. The DECOMPOSE tool enforces this strategy explicitly, requiring the model to identify self-contained subproblems that can be solved independently and then combined.

HYPOTHESIZE generates candidate solutions or approaches. Rather than committing immediately to a single approach, this tool encourages exploration of multiple possibilities, increasing the chances of finding a correct solution.

VERIFY checks whether a proposed solution satisfies the problem's constraints. This is perhaps the most important tool, as it provides explicit error-checking that can catch mistakes before they propagate. The model must articulate each constraint, check whether the solution satisfies it, and explain its reasoning.

BACKTRACK abandons a failed approach and tries another. Reasoning often

leads to dead ends; the ability to recognize failure and return to an earlier state is essential for robust problem-solving.

ANALOGIZE finds similar previously-solved problems. Many problems are variations of problems that have been solved before; recognizing the similarity allows transfer of solution strategies.

ABSTRACT extracts general patterns from specific examples. This tool directly engages the abstraction capabilities we explored in chapter 3, asking the model to identify the invariant structure underlying variable instances.

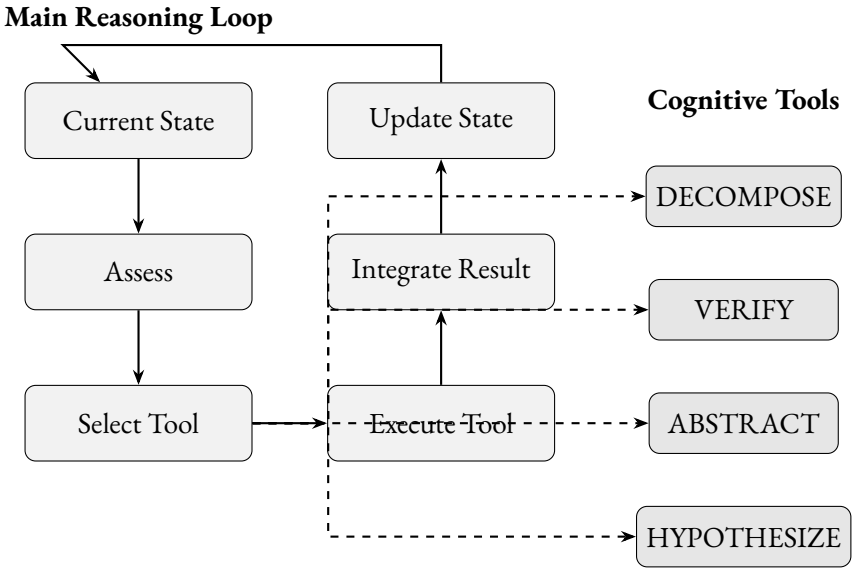

Figure 5.1: The cognitive tools architecture. A main reasoning loop maintains state and orchestrates calls to modular cognitive tools. Each tool performs a specific operation and returns structured results for integration.

5.4 The Orchestration Loop

The power of cognitive tools comes not just from the individual operations but from their orchestration. The main reasoning loop assesses the current state, selects an appropriate tool, executes the tool in sandboxed context, integrates the result into state,

and verifies or backtracks based on outcome.

Tool selection can be guided by various strategies. Simple rule-based approaches work well: if stuck, try BACKTRACK; if given examples, try ABSTRACT; if a candidate solution exists, try VERIFY. More sophisticated approaches let the language model itself choose tools based on its assessment of the current state, or train a policy network to optimize tool selection. The flexibility of the framework allows experimentation with different orchestration strategies.

The termination condition is typically verification: when the VERIFY tool returns "valid" for a proposed solution, the loop terminates with that solution. If the maximum iteration count is reached without a valid solution, the loop fails. Built-in backtracking prevents the system from getting stuck in unproductive directions; when verification fails, the failed approach is recorded, and the system tries another path.

5.5 Connection to Internal Mechanisms

Why do cognitive tools work? One hypothesis is that they externally activate the internal symbolic mechanisms we have explored in previous chapters. When the ABSTRACT tool asks the model to identify patterns, it may engage the symbol abstraction heads that convert tokens to variables. When pattern recognition tasks arise during tool execution, they may engage the symbolic induction heads that complete abstract patterns. When the VERIFY tool checks solutions against constraints, it may engage the retrieval heads that ground abstract predictions in concrete facts.

This hypothesis—that cognitive tools work by orchestrating existing internal mechanisms—has several implications. It explains why the approach works without additional training: the mechanisms already exist, and tools merely channel the model's computation through them more effectively. It explains why tools corresponding to identified internal operations are particularly effective: they align with the model's natural computational structure. And it suggests that understanding internal mechanisms can guide the design of more effective tools.

Several observations support this hypothesis. The performance gains from cognitive tools are largest on tasks requiring abstract reasoning—exactly the tasks where inter-

nal symbolic mechanisms would be most engaged. The tools that correspond to identified internal mechanisms (like ABSTRACT, which aligns with symbol abstraction) tend to be most effective. And the gains are achieved without any additional training, suggesting that the underlying capabilities were present but underutilized.

5.6 Experimental Results

The empirical case for cognitive tools rests on substantial performance improvements across challenging reasoning benchmarks. On AIME 2024, a collection of difficult mathematical problems from the American Invitational Mathematics Examination, GPT-4.1 achieves 32% accuracy with standard prompting. Adding cognitive tools raises this to 53%—a 21 percentage point improvement that surpasses even o1-preview, a model specifically trained for reasoning.

Method	**AIME2024 Pass@1**
GPT-4.1 (baseline)	32%
GPT-4.1 + Cognitive Tools	**53%**
o1-preview	50%

Table 5.1: Cognitive tools enable GPT-4.1 to outperform both its baseline and o1-preview on the challenging AIME2024 benchmark, despite no additional training.

The improvements are robust across model families. Both proprietary models (GPT-4.1, Claude) and open-source models (Llama-3.1, Qwen2.5) show benefits from cognitive tools. This broad applicability suggests that the approach leverages fundamental capabilities present across different architectures and training regimes—capabilities that likely correspond to the symbolic mechanisms we have identified.

5.7 Comparison with Chain-of-Thought

Chain-of-thought prompting encourages step-by-step reasoning through examples or instructions like "let's think step by step." This approach has proven effective across many reasoning tasks, establishing that explicit reasoning steps improve model per-

formance. Cognitive tools can be understood as a more structured form of chain-of-thought.

Aspect	Chain-of-Thought	Cognitive Tools
Structure	Free-form reasoning	Enforced operations
Control	Implicit in prompt	Explicit orchestration
Verifiability	End result only	Each step verified
Backtracking	Requires restart	Built-in support

Table 5.2: Cognitive tools provide more structure and control than standard chain-of-thought prompting, with explicit verification and backtracking support.

Where chain-of-thought leaves the reasoning structure implicit, cognitive tools make it explicit. Each tool call is a "thought" in the chain, but the orchestration loop enforces valid reasoning sequences, preventing the model from skipping steps or making logical leaps. Verification tools catch errors at each step rather than only at the end. Backtracking is built into the framework rather than requiring a complete restart.

The comparison suggests that structure matters. The model has the capability for sophisticated reasoning, but it does not always exercise that capability spontaneously. Explicit structure—whether through chain-of-thought prompting or cognitive tools—helps channel the model's computation through productive paths.

5.8 Leveraging Cognitive Tools in Practice

Even without building a full orchestration system, you can invoke cognitive tool patterns through prompts.

DECOMPOSE pattern:

```
Break this problem into independent subproblems:

Problem:  Calculate the total cost of a road trip
```

```
Subproblem 1:  [Calculate fuel cost based on distance
and MPG]
Subproblem 2:  [Calculate food and lodging per day]
Subproblem 3:  [Calculate toll costs for the route]

Now solve each subproblem, then combine.
```

VERIFY pattern:

```
Verify this solution step by step:

Problem:  Solve x + 5 = 12
Proposed:  x = 7

Check:  Does x + 5 = 7 + 5 = 12?  [YES]
Verdict:  VALID
```

ABSTRACT pattern:

```
Extract the pattern from these examples:

2, 4, 8 -> multiply by 2
3, 9, 27 -> multiply by 3
5, 25, 125 -> ?

Pattern:  Each term is multiplied by the first term.
```

Key Insight (Cognitive Tool Prompts)**.** *Structure your prompts to mimic tool operations: explicit decomposition, verification with articulated checks, abstraction with named patterns. This activates the same reasoning pathways that full cognitive tool systems orchestrate.*

Chapter 6

Activation-Level Interventions

> *"Steering vectors provide a more direct path to behavior modification than prompting—they act on the computational substrate rather than the input."*
>
> — General principle

6.1 Beyond Prompting

Prompt engineering operates at the input level, crafting text that activates desired mechanisms indirectly. But if we understand the internal representations that drive model behavior, we can intervene more directly—adding vectors to the residual stream that steer computation toward desired outcomes. These activation-level interventions provide finer control than prompting alone, enabling behavioral modifications that would be difficult or impossible to achieve through input design.

The techniques in this chapter require access to model internals during inference, making them more technically demanding than prompt engineering. But they offer correspondingly greater power. Function vectors can trigger task execution without any in-context examples. Contrastive steering vectors can modify behavioral tendencies like certainty or verbosity. KV cache modifications can maintain consistent behavior across long generations. These capabilities extend what is possible with language models, opening new applications and control mechanisms.

The connection to the mechanisms we have explored is direct: if we can identify the

attention heads responsible for symbolic abstraction, symbolic induction, or retrieval, we can potentially intervene at those heads to enhance or modify their operation. Activation interventions provide a bridge between mechanistic understanding and practical control.

6.2 Function Vector Steering

The simplest activation intervention adds a pre-extracted function vector to the model's residual stream. As we explored in chapter 4, function vectors encode task specifications in a compact form.

The core operation is simple:

$$\boxed{\mathbf{x}^{(L)} \leftarrow \mathbf{x}^{(L)} + \alpha \cdot \mathbf{FV}_T} \tag{6.1}$$

Run the forward pass normally until reaching layer L, add the scaled function vector to the residual stream, then continue to completion. The added vector influences all subsequent computation, steering the model toward the encoded task.

Three parameters require tuning. The layer L determines where the intervention occurs; middle layers (roughly 40–60% through the network) typically work best, as this is where function vectors are naturally located. The scale α determines the intervention's strength; values around 1.0 are typical starting points, with adjustment based on output quality. The position within the sequence (usually the final token position) determines where the vector is added.

Function vector steering enables several practical applications. A model can be made to perform translation, produce antonyms, classify sentiment, or execute other learned tasks—all without including examples in the prompt. The function vector carries the task specification; the prompt carries only the input to be processed.

> ***Key Insight.*** *Function vector steering completes a circle: the three-stage symbolic architecture produces function vectors during in-context learning; these vectors can then be extracted and reinjected to bypass the learning process entirely. Understand-*

> *ing the mechanism (symbolic architecture) enables a practical technique (function vector steering) that the mechanism naturally supports.*

6.3 Contrastive Steering with ActAdd

While function vectors steer toward specific tasks, contrastive steering modifies broader behavioral tendencies. The ActAdd technique creates steering vectors from prompt contrasts:

$$\boxed{\mathbf{SV} = \text{act}(P^+) - \text{act}(P^-)} \tag{6.2}$$

The resulting steering vector **SV** captures whatever distinguishes the positive behavior from the negative in the model's internal representations. Adding this vector during inference pushes the model toward the positive behavior.

The applications are diverse. A certainty steering vector, extracted from "I am certain that..." versus "I am uncertain whether...", can make models express more or less confidence in their outputs. A reasoning steering vector, from "Let me think through this step by step..." versus "The answer is...", can encourage more deliberate reasoning. A style steering vector, from formal versus casual prose, can modify the register of generated text.

The key insight is that these behavioral tendencies have geometric representations in activation space. The vector from uncertain to certain, from direct to deliberate, from casual to formal—each is a direction in high-dimensional space that can be computed once and applied repeatedly.

6.4 KV Cache Steering

Standard activation steering applies modifications at each token generation step. This works but can cause amplification effects: small steering signals compound over long generations, potentially degrading output quality. KV cache steering offers an alternative that avoids this problem.

The key-value cache stores the keys and values computed during prompt processing, allowing efficient generation without recomputing these quantities. KV cache steering modifies this cache once, after processing the prompt, rather than intervening at each generation step. The modification then influences all subsequent generation through the attention mechanism's use of the cached values.

The procedure extracts steering vectors from contrasting prompts (as in ActAdd) but applies them to the KV cache rather than to per-token activations. This single modification provides stable steering across arbitrarily long generations, without the amplification issues that plague per-token approaches.

KV cache steering is particularly effective for reasoning tasks. By contrasting prompts that encourage step-by-step reasoning with prompts that encourage direct answers, we can extract a "reasoning vector" that, when added to the cache, promotes deliberate reasoning throughout the generation process.

6.5 Dynamic Steering with CREST

The techniques above apply fixed steering throughout generation. But optimal steering may vary: different parts of a reasoning process may benefit from different behavioral modifications. The CREST framework (Cognitive Reasoning Steering at Test-Time) addresses this through dynamic, task-aware steering.

CREST operates in two phases. Offline, it identifies "cognitive heads"—attention heads involved in specific reasoning behaviors like expressing uncertainty, generating examples, validating hypotheses, or backtracking. For each behavior, it computes steering vectors that activate or suppress that behavior.

Online, during inference, CREST applies steering dynamically based on the current task and reasoning state. When the model should express uncertainty, the uncertainty steering vector is applied. When the model should generate examples, the example-generation vector is applied. This dynamic application allows fine-grained control over the reasoning process.

The behaviors that CREST can steer include expressing uncertainty (useful for calibration), generating examples (useful for illustration), hypothesis validation (useful for

checking), and backtracking (useful for recovering from errors). Each behavior corresponds to a steering vector that can be applied or withheld based on task requirements.

6.6 Practical Considerations

Successful activation intervention requires attention to several practical matters.

Layer selection matters significantly. Function vectors are naturally located in middle layers, so function vector steering works best there. Style modifications often work better in later layers, where surface features are determined. Reasoning steering may work best in middle-early layers, before high-level decisions have been made. Experimentation is typically required to find optimal layers for specific applications.

Intervention Type	**Recommended Layers**
Function vectors	Middle (40–60% depth)
Style steering	Later (60–80% depth)
Reasoning steering	Middle-early (30–50% depth)

Table 6.1: General guidelines for layer selection across different intervention types. Optimal layers vary by model and application.

Scale factor tuning is equally important. Too low a scale produces minimal effect; the steering vector is overwhelmed by the model's natural computation. Too high a scale degrades output quality, causing repetition, incoherence, or other artifacts. The optimal range typically falls between 0.5 and 2.0, but varies by model, task, and vector. Starting at 1.0 and adjusting based on output quality is a reasonable approach.

When does steering beat prompting? Steering is preferred when prompt space is limited (the steering vector adds no tokens), when behavior needs to persist reliably across long generations, when fine-grained control is required (steering can be adjusted continuously), or when modifying response style rather than content (style is harder to specify through prompts than through vectors).

6.7 Synthesis: The Control Stack

Activation interventions complete a control stack for language model behavior:

1. **Prompt engineering**: Craft inputs that activate desired mechanisms through natural language
2. **Cognitive tools**: Structure reasoning through explicit operations that leverage internal capabilities
3. **Activation steering**: Directly modify internal representations to steer computation

Each level offers different tradeoffs. Prompting is accessible but indirect; steering is direct but requires model access. Cognitive tools provide structure without requiring weight access; activation interventions require weight access but offer maximum control.

The levels complement each other. A practitioner might use prompt engineering for most tasks, add cognitive tool patterns for complex reasoning, and employ activation steering when precise behavioral control is needed. Understanding all three levels—and their connections to the underlying symbolic mechanisms—enables choosing the right tool for each situation.

> **Practical Leverage.** The control stack can be combined: use carefully designed prompts to set up the context, structure reasoning through cognitive tool patterns, and apply activation steering for fine-grained behavioral control. Each intervention reinforces the others by working with, rather than against, the model's internal symbolic architecture.

Part III

Interconnections and Synthesis

Chapter 7

The Unified Picture

> *"What emerges is not a collection of isolated discoveries, but a unified theory of how symbolic computation arises within neural networks—and how we might harness this understanding to build more capable systems."*
>
> — Synthesis

7.1 Hierarchical Organization of Mechanisms

The various mechanisms discussed form a coherent hierarchy, each building on the previous:

Each level provides the foundation for the next:

1. **Attention Foundation**: The basic query-key-value mechanism enables flexible information routing. This is the raw computational substrate.
2. **Induction Emergence**: Through training, attention heads compose to form induction circuits that match patterns and copy successors. This is the first emergence of learning-from-context.
3. **Symbolic Abstraction**: Further training produces specialized heads that extend induction to abstract variables, enabling domain-independent pattern recognition.
4. **Procedural Encoding**: The results of symbolic processing can be captured as function vectors, enabling knowledge transfer and composition.
5. **External Orchestration**: Cognitive tools leverage internal mechanisms through

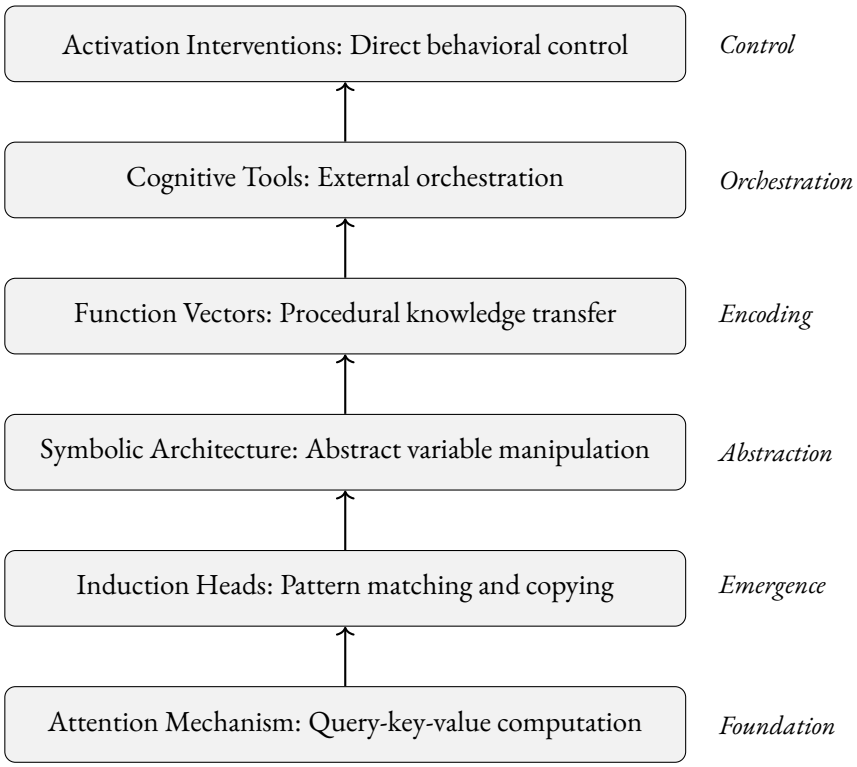

Figure 7.1: Hierarchical organization of symbolic reasoning mechanisms and intervention techniques.

structured external operations, amplifying their effectiveness.

6. **Direct Control**: Activation interventions provide fine-grained control over the computational process, steering behavior at the level of internal representations.

Without attention, no induction heads could form. Without induction heads, symbolic mechanisms would lack the pattern-matching substrate. Without symbolic mechanisms, function vectors would capture only surface-level regularities. Without understanding these mechanisms, cognitive tools and activation interventions would be designed blindly rather than to align with the model's computational structure.

7.2 Cross-Cutting Themes

Three mechanisms appear across all levels:

7.2.1 Composition

All levels rely on composition—combining simpler operations to achieve complex behavior:

- Induction heads compose previous-token heads with pattern-matching heads
- Symbolic architecture composes abstraction, induction, and retrieval
- Function vectors compose multiple task-specific components
- Cognitive tools compose individual operations into reasoning chains
- Activation interventions can compose multiple steering vectors

Composition is the key to the expressiveness of transformer reasoning. Individual attention heads have limited capabilities; their combination creates emergent behavior far exceeding the sum of parts.

7.2.2 Indirection

The ability to reference content stored elsewhere appears at every level:

- Induction heads "point to" previous occurrences via attention
- Symbolic heads use binding IDs to track variable assignments
- Function vectors provide indirect specification of behavior

- Cognitive tools reference abstract operations rather than specific computations
- Activation steering indirectly modifies behavior through representation space

Indirection separates "what to do" from "what to do it with," enabling flexible, context-dependent computation.

7.2.3 Invariance

Representations that remain stable across superficial variations:

- Induction pattern matching across different contexts
- Symbol abstraction regardless of specific tokens
- Function vector transfer across different prompts
- Cognitive tool operations that work across problem domains
- Steering vectors that modify behavior regardless of specific content

Invariance is what makes abstraction useful: it enables learned knowledge to transfer beyond the specific examples encountered during learning.

7.3 Resolution of the Symbolic-Connectionist Debate

These findings suggest that the traditional opposition between symbolic and neural approaches represents a false dichotomy. Neural networks don't need explicit symbolic architecture to perform symbolic computation—they *develop their own form* of symbol processing through training. This organic emergence combines the **flexibility** of neural computation (gradient-based learning, distributed representations, graceful degradation) with the **systematicity** of symbolic processing (compositionality, abstraction, rule following).

The implication is profound: symbolic reasoning isn't something we need to *add* to neural networks; it's something that *emerges* when neural networks are trained on sufficiently rich data. The transformer architecture provides the raw materials (attention, composition, residual streams); language modeling provides the pressure (predicting structure requires recognizing structure); and symbolic mechanisms emerge as the solution.

Earlier Finding	**Role in Unified Picture**
Induction Heads	Pattern-matching foundation
Symbol Abstraction	Token-to-variable mapping
Symbolic Induction	Variable-level pattern completion
Retrieval Heads	Variable-to-token grounding
Function Vectors	Encoded procedural knowledge
Cognitive Tools	External mechanism orchestration
Activation Interventions	Direct behavioral control

Table 7.1: How different discoveries fit into the unified picture of symbolic reasoning in LLMs.

Chapter 8

Leveraging Symbolic Patterns: Strategic Insights

8.1 The Practitioner's Advantage

Understanding symbolic mechanisms transforms prompt engineering from art to science. Rather than relying on intuition and trial-and-error, you can design prompts that consciously activate the computational structures you need.

This chapter distills the theoretical insights into actionable strategies. Each strategy is grounded in mechanism: we explain not just *what* to do, but *why* it works in terms of the underlying computational structures.

8.2 Strategy 1: Activate Symbol Abstraction Through Diverse Instantiation

Strategy 8.1 (Diverse Instantiation)**.** When demonstrating a pattern, provide examples using diverse tokens that share only the structural relationship you want the model to learn. This forces symbol abstraction heads to activate, as they must extract the invariant structure across varying content.

Why it works: Symbol abstraction heads create invariant representations by attending to *relational* features while the value projection captures *positional role* information. When all your examples use similar tokens, the model might rely on token-level

similarity rather than abstraction. Diverse tokens force the model to find what's invariant across the variation—which is precisely the abstract structure.

Application:

```
# WEAK: Similar tokens, abstraction not forced
examples = """
cat -> animal
dog -> animal
bird -> animal
hamster -> ?
"""

# STRONG: Diverse tokens, abstraction required
examples = """
cat -> animal
blue -> color
hammer -> tool
piano -> ?
"""
```

In the strong version, the only commonality across examples is the structural relationship [object] → [category]. The model cannot rely on token similarity; it must abstract.

8.3 Strategy 2: Make Variable Roles Explicit

Strategy 8.2 (Explicit Variable Declaration)**.** Explicitly name and define the abstract variables in your prompt. This provides "binding sites" that retrieval heads can use, and makes the structural roles unambiguous.

Why it works: Retrieval heads solve the variable binding problem by attending to positions where variables were introduced. When variables are implicit, the model must infer binding sites from context. When variables are explicit, binding is unambiguous.

Application:

```
# IMPLICIT: Model must infer variable structure
prompt = """
hot -> cold, up -> down, fast -> ?
"""

# EXPLICIT: Variable roles declared
prompt = """
PATTERN: word X -> antonym of X

Examples:
X = hot, antonym(X) = cold
X = up, antonym(X) = down

Apply:
X = fast, antonym(X) = ?
"""
```

The explicit version provides clear binding: "X" is a variable that can be traced from definition to application. Retrieval heads can attend to "X = fast" to resolve what "X" refers to.

8.4 Strategy 3: Leverage the Three-Stage Architecture

Strategy 8.3 (Stage-Aware Prompting)**.** Structure your prompt to support each stage of the symbolic architecture: provide clear structural patterns for abstraction, consistent relationships for induction, and explicit bindings for retrieval.

Why it works: The three stages—abstraction, induction, retrieval—each have different requirements. A prompt that supports one stage may hinder another. Stage-aware prompting ensures all stages can succeed.

Application:

```
prompt = """
TASK: Complete the pattern

```

```
STAGE 1 - STRUCTURAL PATTERN (for abstraction):
This is an ABA pattern: first element, second element, first
    element again.

STAGE 2 - RELATIONSHIP (for induction):
Pattern: [A] [B] [A]
Each row follows this same structure.

STAGE 3 - CURRENT BINDING (for retrieval):
Current row:
A = "sun"
B = "moon"
A = ?

Complete A:
"""
```

This prompt explicitly supports all three stages: Stage 1 states the abstract pattern type, Stage 2 shows the relationship between variables, and Stage 3 provides explicit current bindings.

8.5 Strategy 4: Use Contrastive Examples

Strategy 8.4 (Contrastive Pairs)**.** Provide examples of what the pattern is NOT, alongside examples of what it IS. This sharpens the boundary of the abstraction and prevents overgeneralization.

Why it works: Symbol abstraction creates representations based on what's *invariant* across examples. Without negative examples, many irrelevant features might appear invariant. Contrastive examples break spurious invariances.

8.6 Strategy 5: Scaffold with Variable Tracking

Strategy 8.5 (Variable-Tracked Reasoning)**.** When using chain-of-thought prompting, explicitly track variable values through each step. This provides retrieval heads with clear binding points and makes the reasoning chain more robust.

Why it works: Chain-of-thought helps by externalizing reasoning steps, but if variable bindings become unclear partway through, retrieval heads may fail to resolve references. Explicit tracking maintains clear bindings throughout.

8.7 Strategy 6: Activate Fuzzy Induction for Semantic Transfer

Strategy 8.6 (Semantic Bridging)**.** When you want the model to transfer a pattern across semantic domains, provide bridging examples that show the pattern works across different content types.

Why it works: Fuzzy induction heads match based on embedding similarity, not exact tokens. By showing the pattern works across semantically diverse examples, you activate fuzzy matching for the novel case.

8.8 Strategy 7: Leverage Function Vector Intuitions

Strategy 8.7 (Function-First Prompting)**.** Describe the function abstractly before showing examples. This activates function-level representations before getting into specifics.

Why it works: Function vectors capture the abstract operation, not just the input-output pairs. By priming with the function description, you bias the model toward encoding a function rather than memorizing examples.

Chapter 9

Practical Leverage: Working With Symbolic Mechanisms

> *"The goal is not to implement symbolic reasoning externally, but to activate the symbolic machinery that already exists within language models through principled prompting and interaction design."*
>
> — Synthesis

9.1 From Theory to Practice

The preceding chapters have established that language models develop genuine symbolic mechanisms: abstraction heads that convert tokens to variables, induction heads that complete patterns over abstractions, and retrieval heads that ground predictions in context. This chapter translates that theoretical understanding into practical guidance for working with these mechanisms through natural language interaction.

The key insight is that symbolic mechanisms are *already present*—they don't need to be implemented, only activated. The challenge is designing interactions that engage these mechanisms effectively. This requires understanding what activates each stage and structuring our prompts and conversations accordingly.

9.2 Activating Symbol Abstraction

Symbol abstraction heads respond to *relational structure*. They activate when the model encounters patterns where the relationship between elements matters more than the elements themselves. To leverage this:

> **Practical Leverage** (The Diversity Principle)**.** Provide examples that vary in surface features but share abstract structure. This forces the model to extract what is invariant—the pattern itself—rather than relying on token-level similarity.
>
> **Weak activation** (similar tokens):
>
> ```
> cat → animal
> dog → animal
> horse → animal
> elephant → ?
> ```
>
> **Strong activation** (diverse tokens, same relationship):
>
> ```
> cat → animal
> hammer → tool
> blue → color
> sadness → ?
> ```
>
> In the strong version, the only commonality is the structural relationship [instance] → [category]. The model cannot rely on semantic similarity between inputs; it must abstract the pattern.

The principle extends beyond simple classification. For any task where you want the model to generalize, ensure your examples span the space of possible instantiations while maintaining consistent structure.

9.3 Engaging Symbolic Induction

Symbolic induction heads complete patterns over abstract variables, not tokens. To engage them effectively:

Practical Leverage (Explicit Pattern Declaration)**.** Name the abstract pattern before showing examples. This primes the induction mechanism by declaring what kind of pattern to look for.

Implicit pattern (model must discover):

```
red blue red → blue
cat dog cat → dog
sun moon sun → ?
```

Explicit pattern (directly activates induction):

```
PATTERN: Given sequence [A][B][A], predict [B]

Examples:
red blue red → blue
cat dog cat → dog

Apply pattern:
sun moon sun → ?
```

Naming the pattern with variables (A, B) engages the symbolic induction stage directly, reducing the abstraction burden.

Practical Leverage (Consistent Structural Framing)**.** Use consistent formatting across examples so that structural roles are unambiguous. The induction mechanism looks for patterns in how elements relate; inconsistent formatting obscures these relationships.

Inconsistent (structural roles unclear):

```
Paris is the capital of France
Germany:  Berlin
Spain's capital is Madrid
Italy → ?
```

Consistent (structural roles clear):

```
France → Paris
Germany → Berlin
Spain → Madrid
Italy → ?
```

The consistent version creates clear "preceded-by" patterns that induction heads can match.

9.4 Supporting Retrieval

Retrieval heads map abstract predictions back to concrete tokens by resolving variable bindings. To support this:

Practical Leverage (Clear Variable Binding)**.** When working with multi-step reasoning or complex patterns, make variable bindings explicit rather than implicit. Each variable should have exactly one unambiguous binding in context.

Ambiguous binding (retrieval may fail):

```
John met Mary at the park.  Later, Mary met Susan at
the cafe.
The next day, John met someone at the library.
Who did John likely meet?
```

Clear binding:

```
BINDINGS:
Person A = John
Person B = Mary
Person C = Susan

PATTERN: A meets B, then B meets C, then A meets ?

Following the pattern, A (John) should meet C (Susan).
```

Explicit variable names create unambiguous retrieval targets.

9.5 The Three-Stage Prompt Design

Combining the principles above, an effective prompt supports all three stages of the symbolic architecture:

Stage-Aligned Prompt Template

```
[STAGE 1:  ABSTRACTION SUPPORT]
PATTERN TYPE: [name the abstract pattern]
VARIABLES: [list the abstract roles]

[STAGE 2:  INDUCTION SUPPORT]
EXAMPLES (diverse instantiations of the pattern):
Example 1:  [show pattern with binding set 1]
Example 2:  [show pattern with binding set 2]
Example 3:  [show pattern with binding set 3]

[STAGE 3:  RETRIEVAL SUPPORT]
CURRENT BINDINGS:
[Variable 1] = [value]
[Variable 2] = [value]

APPLY PATTERN:
```

This template ensures that abstraction heads receive structural cues, induction heads receive consistent pattern examples, and retrieval heads receive clear bindings to resolve.

9.6 Cognitive Tool Patterns in Natural Language

The cognitive tools framework can be invoked through natural language without any external orchestration system. The key is structuring the conversation to trigger the relevant cognitive operations:

Practical Leverage (DECOMPOSE in Natural Language)**.** Request explicit decomposition before attempting solution. This engages problem-structuring mechanisms.

```
Before solving this problem, break it into independent
subproblems.

Problem:  [complex problem statement]

Subproblems:
1.  [identify first self-contained subproblem]
2.  [identify second self-contained subproblem]
3.  [continue as needed]

Now solve each subproblem, then combine the results.
```

Practical Leverage (VERIFY in Natural Language)**.** Request explicit verification against stated constraints. This catches errors before they propagate.

```
Check this proposed solution against the problem constraints.

Problem:  [original problem]
Constraints:  [list explicit constraints]
Proposed solution:  [candidate solution]

Verification:
☐ Constraint 1:  [satisfied/violated because...]
☐ Constraint 2:  [satisfied/violated because...]

Verdict:  [VALID/INVALID]
```

Practical Leverage (ABSTRACT in Natural Language)**.** Request explicit pattern extraction before applying to new cases.

```
Look at these examples and identify the underlying pattern.

Examples:
  example 1
  example 2
  example 3

Pattern description:  describe the abstract rule

Now apply this pattern to:  new case
```

9.7 Chain-of-Thought with Variable Tracking

For complex reasoning, maintain explicit variable bindings throughout the chain of thought. This supports the retrieval mechanism by ensuring clear reference resolution:

Variable-Tracked Reasoning Format

```
PROBLEM: [problem statement]

STEP 1:
Observation:  [what we notice]
Let X = [establish first variable binding]

STEP 2:
Using X = [recall the binding]
Compute:  [operation on X]
Let Y = [result, establish new binding]

STEP 3:
Using X = [value] and Y = [value]
[continue reasoning with explicit references]

CURRENT STATE:
X = [current value]
Y = [current value]
```

```
CONCLUSION: [final answer using established bindings]
```

The explicit binding tracking prevents the model from losing track of values across long reasoning chains, a common failure mode in complex problems.

9.8 Activation Principles for Advanced Tasks

For sophisticated reasoning tasks, combine multiple activation strategies:

Key Insight (Analogical Reasoning)**.** *To activate analogical reasoning:*

1. ***Source domain****: Present a well-understood situation with explicit structure*
2. ***Mapping declaration****: Explicitly name the correspondence between domains*
3. ***Target domain****: Present the new situation with parallel structure*
4. ***Transfer request****: Ask for the corresponding element/relationship*

```
SOURCE DOMAIN:
A physician diagnoses a patient by observing symptoms,
forming hypotheses, and testing with examinations.

MAPPING:
physician ↔ debugger
patient ↔ program
symptoms ↔ error messages
diagnosis ↔ root cause
treatment ↔ fix

TARGET DOMAIN:
A debugger examines a program with errors.
Following the analogy, what should the debugger do
next?
```

Key Insight (Hierarchical Abstraction)**.** *For tasks requiring multiple levels of ab-*

straction:

1. ***Ground level**: Present concrete examples*
2. ***First abstraction**: Extract immediate patterns*
3. ***Higher abstraction**: Identify patterns among patterns*
4. ***Application**: Apply the meta-pattern to new cases*

This mirrors how the symbolic architecture processes information through successive layers of abstraction.

9.9 When Symbolic Mechanisms May Fail

Understanding the limits helps calibrate expectations:

- **Insufficient examples**: Symbol abstraction requires enough diversity to distinguish invariant structure from incidental features. Two examples may be insufficient.
- **Ambiguous bindings**: When the same token plays multiple roles, retrieval heads may fail to resolve the correct binding. Make roles unambiguous.
- **Too-deep nesting**: Extremely deep pattern nesting may exceed the effective depth of symbolic processing. Flatten where possible.
- **Contradictory patterns**: When examples suggest multiple incompatible patterns, induction may fail. Ensure examples are consistent.
- **Novel semantic relationships**: While symbolic mechanisms generalize structure, they rely on learned semantic relationships. Truly novel relationships may require more explicit guidance.

9.10 Additional Practical Prompt Templates

The following templates provide ready-to-use patterns that leverage the symbolic mechanisms described in this document.

Practical Leverage (Data Transformation Pattern)**.** For tasks requiring structured data transformation, make the transformation rule explicit:

```
TRANSFORMATION RULE: [input structure] → [output structure]

Examples:
{name: "John", age: 30} → "John is 30 years old"
{name: "Alice", age: 25} → "Alice is 25 years old"
{name: "Bob", age: 42} → "Bob is 42 years old"

Apply to: {name: "Carol", age: 33}
```

The consistent structure activates pattern matching, while the explicit rule declaration supports abstraction.

Practical Leverage (Multi-Step Reasoning with State)**.** For problems requiring sequential computation, maintain explicit state between steps:

```
PROBLEM: Calculate ((3 + 5) * 2) - 4

TRACE:
state_0: expression = ((3 + 5) * 2) - 4
step_1: 3 + 5 = 8, expression = (8 * 2) - 4
state_1: intermediate = 8
step_2: 8 * 2 = 16, expression = 16 - 4
state_2: intermediate = 16
step_3: 16 - 4 = 12

RESULT: 12

Now apply to: ((7 - 3) * 4) + 2
```

Explicit state tracking helps retrieval heads resolve variable bindings across reasoning steps.

Practical Leverage (Error Correction Pattern)**.** For debugging or correction tasks, structure the error-fix relationship explicitly:

```
FORMAT: [INCORRECT] → [CORRECT] // [REASON]

Examples:
"Their going to the store" → "They're going to the
store" // contraction
"Your amazing" → "You're amazing" // contraction vs
possessive
"Its raining" → "It's raining" // contraction

Correct:  "The dogs wagging it's tail"
```

The explicit reason annotation helps the model generalize the correction pattern rather than memorizing specific fixes.

Practical Leverage (Code Generation from Specification)**.** When generating code, provide specifications in a consistent format that mirrors code structure:

```
SPEC FORMAT:
Function:  [name]
Input:  [type] [description]
Output:  [type] [description]
Behavior:  [what it does]

Example 1:
Function:  double
Input:  int x
Output:  int
Behavior:  returns x multiplied by 2
→ def double(x):  return x * 2

Example 2:
Function:  is_even
Input:  int n
Output:  bool
```

```
Behavior: returns True if n is divisible by 2
→ def is_even(n): return n % 2 == 0

Generate:
Function: square
Input: int x
Output: int
Behavior: returns x raised to the power of 2
```

Practical Leverage (Classification with Reasoning Chain)**.** For classification tasks, elicit both the classification and the reasoning:

```
TASK: Classify sentiment with reasoning

FORMAT: [text] → [sentiment]: [key indicators]

Examples:
"I love this product!" → POSITIVE: exclamation, "love"
"Terrible experience." → NEGATIVE: "terrible"
"It works as expected." → NEUTRAL: factual, no emotion

Classify: "This exceeded all my expectations!"
```

Requiring explicit reasoning activates the abstraction mechanism rather than pure pattern matching.

Practical Leverage (Structured Comparison)**.** For comparison tasks, use a parallel structure that highlights dimensions:

```
COMPARE: [Entity A] vs [Entity B]

Dimensions:
- [Aspect 1]: A = [value], B = [value] → [winner/tie]
- [Aspect 2]: A = [value], B = [value] → [winner/tie]
```

```
- [Aspect 3]:  A = [value], B = [value] → [winner/tie]

SUMMARY: [overall conclusion]

Example:
COMPARE: Python vs Java for web development

Dimensions:
- Learning curve:  Python = easy, Java = moderate →
Python
- Performance:  Python = good, Java = excellent →
Java
- Ecosystem:  Python = mature, Java = mature → Tie

SUMMARY: Python for rapid development, Java for enterprise
scale

Now compare:  PostgreSQL vs MongoDB for user data storage
```

Practical Leverage (Recursive Structure Processing)**.** For tasks involving recursive or nested structures, explicitly show the recursion:

```
TASK: Summarize nested organization

RULE: For each level, format as:
[Level N]: [name] - [role]
[Level N+1]:  → [subordinates]

Example:
Input:  {"CEO":  {"CTO":  ["Engineer1", "Engineer2"],
"CFO":  ["Analyst"]}}

Level 0:  CEO - executive
```

```
Level 1:  → CTO - technical, CFO - financial
Level 2:  → CTO: Engineer1, Engineer2; CFO: Analyst

Apply to:  {"Director":  {"Manager A":  ["Staff 1"],
"Manager B":  ["Staff 2", "Staff 3"]}}
```

Showing the recursive pattern explicitly helps the model apply it to arbitrary nesting depths.

9.11 Summary: The Practitioner's Checklist

When designing prompts to leverage symbolic mechanisms:

1. **For abstraction**: Use semantically diverse examples that share only structural relationship
2. **For induction**: Name patterns explicitly and maintain consistent formatting
3. **For retrieval**: Establish clear, unambiguous variable bindings
4. **For cognitive operations**: Structure the conversation to invoke DECOMPOSE, VERIFY, ABSTRACT, and other tools through natural language
5. **For complex reasoning**: Maintain explicit variable tracking throughout the chain of thought
6. **For robustness**: Provide multiple examples, check for ambiguity, and include verification steps

The goal is not to implement symbolic reasoning but to *activate* the symbolic machinery already present through principled interaction design.

Part IV

Implications and Future Directions

Chapter 10

Deeper Insights and Open Questions

10.1 What These Mechanisms Tell Us About Intelligence

The discovery that neural networks spontaneously develop symbolic mechanisms has implications beyond prompt engineering. It suggests something profound about the nature of intelligence itself.

10.1.1 Symbols as Emergent, Not Innate

Classical symbolic AI assumed that symbolic manipulation was a *primitive*—something that had to be built into the system from the start. Neural networks seemed to refute this by succeeding without explicit symbols. But the picture is more subtle: neural networks develop their own form of symbolic processing through the pressures of learning.

This suggests that symbolic reasoning may be less about having the right *primitives* and more about having the right *pressures*. Given a task that requires abstraction (like language modeling), and an architecture with sufficient expressiveness (like transformers), symbolic mechanisms emerge naturally.

10.1.2 Limits of Emergence

These mechanisms have limits. Models retain token-level information alongside abstractions ("content effects"), meaning they don't achieve pure symbolic processing. The correlation analysis shows symbolic induction heads are distinct from standard induction heads ($r = 0.11$), but they're also distinct from pure symbolic systems.

This suggests a hybrid architecture: neural networks that develop symbolic-*like* processing, but embedded within a fundamentally subsymbolic substrate. The impli-

cations for AGI remain unclear—can this hybrid architecture scale to full human-level reasoning, or are there fundamental limits?

10.2 Safety and Alignment Implications

10.2.1 Phase Changes and Emergent Capabilities

The phase change phenomenon has critical safety implications. Neural network capabilities can abruptly form or change as models train or increase in scale, and are of particular concern for safety as they mean that undesired or dangerous behavior could emerge abruptly.

Induction heads provide a case study of such emergence. Before the phase change, models lack in-context learning; after, they have it. The capability doesn't gradually improve—it suddenly appears when the circuit crystallizes.

If dangerous capabilities (like deceptive reasoning or manipulation) emerge through similar phase changes, we might have little warning. Understanding the mechanistic basis of phase changes could help predict and prevent dangerous emergent behaviors.

10.2.2 Interpretability as a Safety Tool

The symbolic mechanisms described provide interpretable structure within otherwise opaque models. This has practical safety applications: debugging when a model fails at reasoning, steering through interventions at specific stages, and monitoring which mechanisms activate during reasoning.

10.3 Future Research Directions

10.3.1 Higher-Order Symbolic Processing

Current research focuses on single-level abstraction (tokens → variables → tokens). But human reasoning involves multiple levels of abstraction: concepts, metaconcepts, theories about theories. Do larger models develop higher-order symbolic mechanisms—abstraction over abstractions?

10.3.2 Architectural Innovations

Now that we understand how symbolic mechanisms emerge in transformers, can we design architectures that support them more effectively? Candidates include explicit binding mechanisms (rather than implicit binding IDs), multi-scale attention for hierarchical abstraction, and structured memory for maintaining variable bindings.

10.3.3 Training for Symbolism

Can training objectives be designed to encourage stronger symbolic mechanisms? Possibilities include curriculum learning that progresses from concrete to abstract, auxiliary losses that reward invariant representations, and data augmentation that increases structural diversity.

Quick Reference: Prompting Patterns

.1 Variable Declaration Pattern

```
VARIABLES:
- VAR1: [description]
- VAR2: [description]

RELATIONSHIPS:
- VAR1 relates to VAR2 via [relationship]

APPLY TO: [specific instance]
```

.2 Analogy Pattern

```
ANALOGY: A is to B as C is to D

SOURCE:
A = [value], B = [value]
Relationship R(A,B) = [describe]

TARGET:
C = [value], D = ?
Apply R to find D.
```

.3 Induction Pattern

```
EXAMPLES:
1. [input1] -> [output1]
2. [input2] -> [output2]
3. [input3] -> [output3]

PATTERN: [describe abstract rule]

APPLY:
[new_input] -> ?
```

.4 Function Specification Pattern

```
FUNCTION: [name]
INPUT TYPE: [description]
OUTPUT TYPE: [description]
BEHAVIOR: [what it does]

EXAMPLES:
[name]([in1]) = [out1]
[name]([in2]) = [out2]

APPLY:
[name]([new_input]) = ?
```

Glossary

Activation Intervention Technique for modifying model behavior by adding vectors to the residual stream during inference.

Binding ID Vector representing an index that maintains associations between variables and values in context.

Cognitive Tool Self-contained function encapsulating a specific reasoning operation, executed by the LLM in a sandboxed context.

Composition The combination of multiple attention heads to achieve computations that no single head could perform alone.

CREST Cognitive Reasoning Steering at Test-Time; framework for dynamic, task-aware activation steering.

Function Vector Compact representation of a demonstrated procedure, extractable from model hidden states and transferable to new contexts.

Fuzzy Induction Pattern matching where semantically similar (not identical) tokens satisfy the match condition.

In-Context Learning Model's ability to learn new tasks from examples provided in the prompt without weight updates.

Indirection Mechanism where representations refer to content stored elsewhere, enabling abstract manipulation.

Induction Head Attention head circuit that completes patterns like $[A][B] \ldots [A] \rightarrow [B]$.

Invariance Property of representations remaining stable regardless of superficial variations in input.

KV Cache Steering Technique for applying steering vectors to the key-value cache rather than per-token activations.

OV Circuit Output-Value circuit determining what information an attention head retrieves from attended positions.

Phase Change Abrupt transition in training where new capabilities emerge suddenly through circuit formation.

Previous Token Head Attention head that copies information from the preceding token into each position's representation.

QK Circuit Query-Key circuit determining where an attention head attends.

Retrieval Head Attention head that maps abstract variable predictions to concrete tokens by resolving bindings.

Steering Vector Direction in activation space that, when added during inference, modifies model behavior.

Symbol Abstraction Head Attention head converting tokens to abstract variable representations based on structural roles.

Symbolic Induction Head Attention head performing pattern induction over abstract variables rather than tokens.

Key References

.5 Primary Sources

1. Yang, Y., Campbell, D., Huang, K., Wang, M., Cohen, J., & Webb, T. (2025). "Emergent Symbolic Mechanisms Support Abstract Reasoning in Large Language Models." *Proceedings of ICML 2025*. https://arxiv.org/abs/2502.20332
2. Olsson, C., Elhage, N., Nanda, N., et al. (2022). "In-context Learning and Induction Heads." *Transformer Circuits Thread*. https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/
3. Todd, E., Li, M., Sharma, A., et al. (2024). "Function Vectors in Large Language Models." *ICLR 2024*. https://functions.baulab.info/
4. Ebouky, B., Bartezzaghi, A., & Rigotti, M. (2025). "Eliciting Reasoning in Language Models with Cognitive Tools." *arXiv:2506.12115*.
5. Context Engineering Framework. "Symbolic Mechanisms." https://github.com/davidkimai/Context-Engineering

.6 Additional Reading

1. Wei, J., et al. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models." *NeurIPS 2022*.
2. Elhage, N., et al. (2021). "A Mathematical Framework for Transformer Circuits." *Transformer Circuits Thread*.
3. Webb, T., et al. (2023). "Emergent Analogical Reasoning in Large Language Models." *Nature Human Behaviour*.

Conclusion

This document has traced the arc from basic attention mechanisms to emergent symbolic reasoning in large language models, revealing a coherent picture of how neural networks develop the capacity for abstract thought.

The journey began with induction heads—the simplest circuits that enable in-context learning. These attention patterns implement a match-and-copy operation that, despite its simplicity, underlies the model's ability to learn from examples in context. The phase change phenomenon showed us that capabilities don't gradually improve; they suddenly emerge when the right circuits crystallize.

Building on this foundation, we explored the emergent symbolic architecture: symbol abstraction heads that create invariant representations of structural roles, symbolic induction heads that perform pattern matching over these abstract variables, and retrieval heads that resolve variable bindings back to concrete tokens. This three-stage architecture implements genuine symbolic processing—not hand-coded, but spontaneously developed through the pressures of language modeling.

Function vectors provided a higher-level view: the procedures learned during in-context learning can be captured as compact vectors, transferred across contexts, and even composed into complex operations. This suggests that models don't just recognize patterns; they encode *operations*—transferable procedures that generalize beyond their training examples.

Cognitive tools showed us how to leverage these internal mechanisms through external orchestration. By providing structured operations for decomposition, verification, abstraction, and backtracking, we can engage the symbolic machinery more effectively than prompting alone achieves. The substantial performance gains—surpassing even specially-trained reasoning models—demonstrate that the capability was present all along; it merely needed the right scaffolding.

Activation interventions completed the control stack, providing direct access to

the computational substrate. Function vector steering, contrastive steering, KV cache modification, and dynamic steering with CREST offer progressively finer control over model behavior, enabling modifications that would be difficult or impossible through prompting alone.

For practitioners, the implications are transformative. Understanding that models possess these mechanisms changes prompt engineering from trial-and-error to principled design. We can structure prompts to activate symbol abstraction (through diverse instantiation), support symbolic induction (through clear pattern structure), and enable retrieval (through explicit variable bindings). We can orchestrate reasoning through cognitive tool patterns. We can steer behavior through activation interventions. The shift is from "try things until something works" to "design interventions that align with computational structure."

For the field, these findings suggest that the symbolic-connectionist debate rested on a false dichotomy. Neural networks don't need explicit symbolic architecture; they develop their own form of symbol processing when the task demands it. This organic emergence combines neural flexibility with symbolic systematicity, pointing toward a synthesis that transcends both traditions.

The mechanisms described here represent our current best understanding of one of AI's most remarkable capabilities: the emergence of abstract thought from the statistics of language. As models continue to scale and new architectures emerge, we can expect these mechanisms to evolve and elaborate. But the fundamental insight will likely remain: intelligence is not about what primitives you start with, but what structures emerge from learning on rich data.

The journey from tokens to symbols to reasoning continues. Understanding the path already traveled helps us navigate what lies ahead.