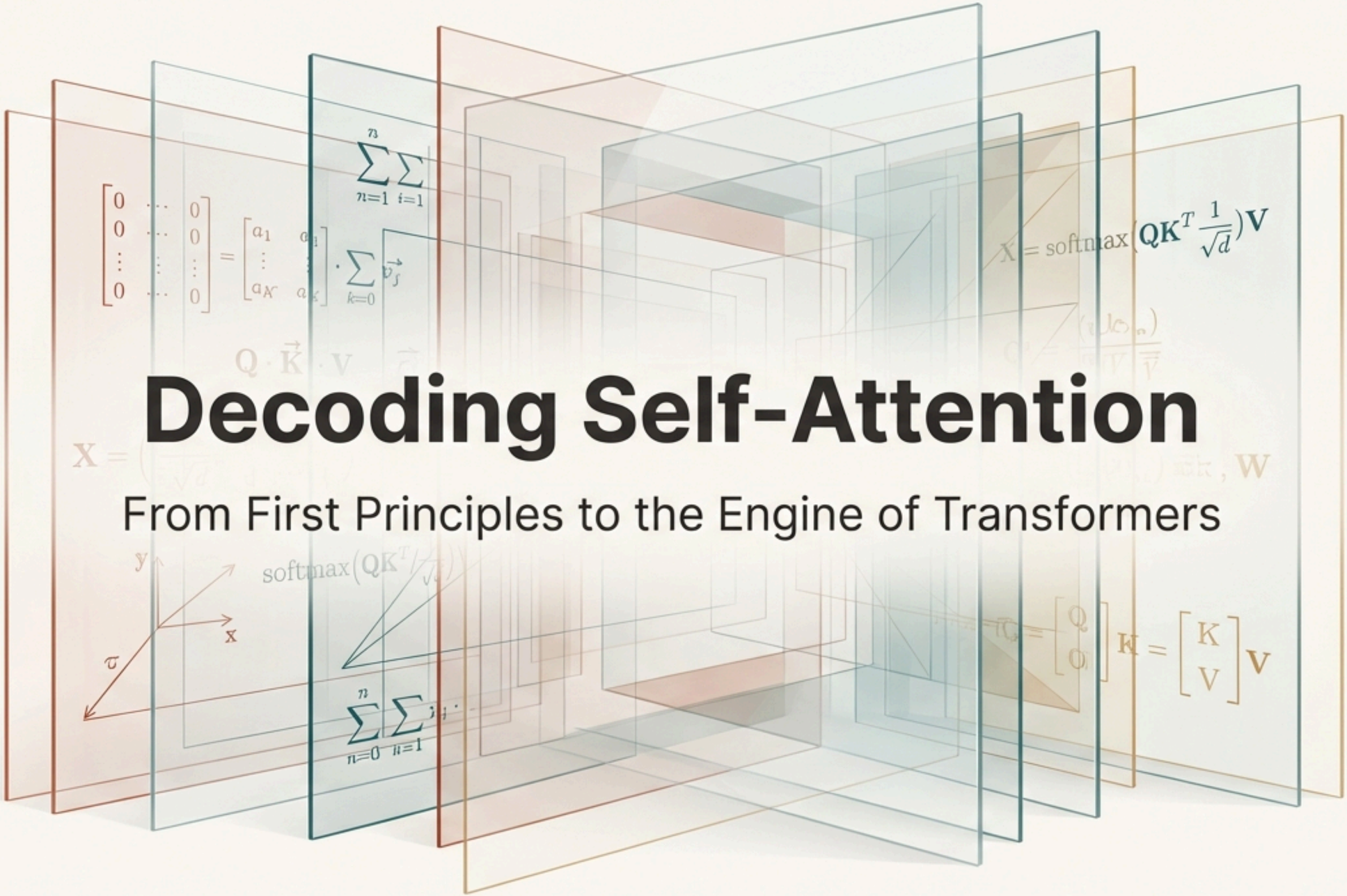


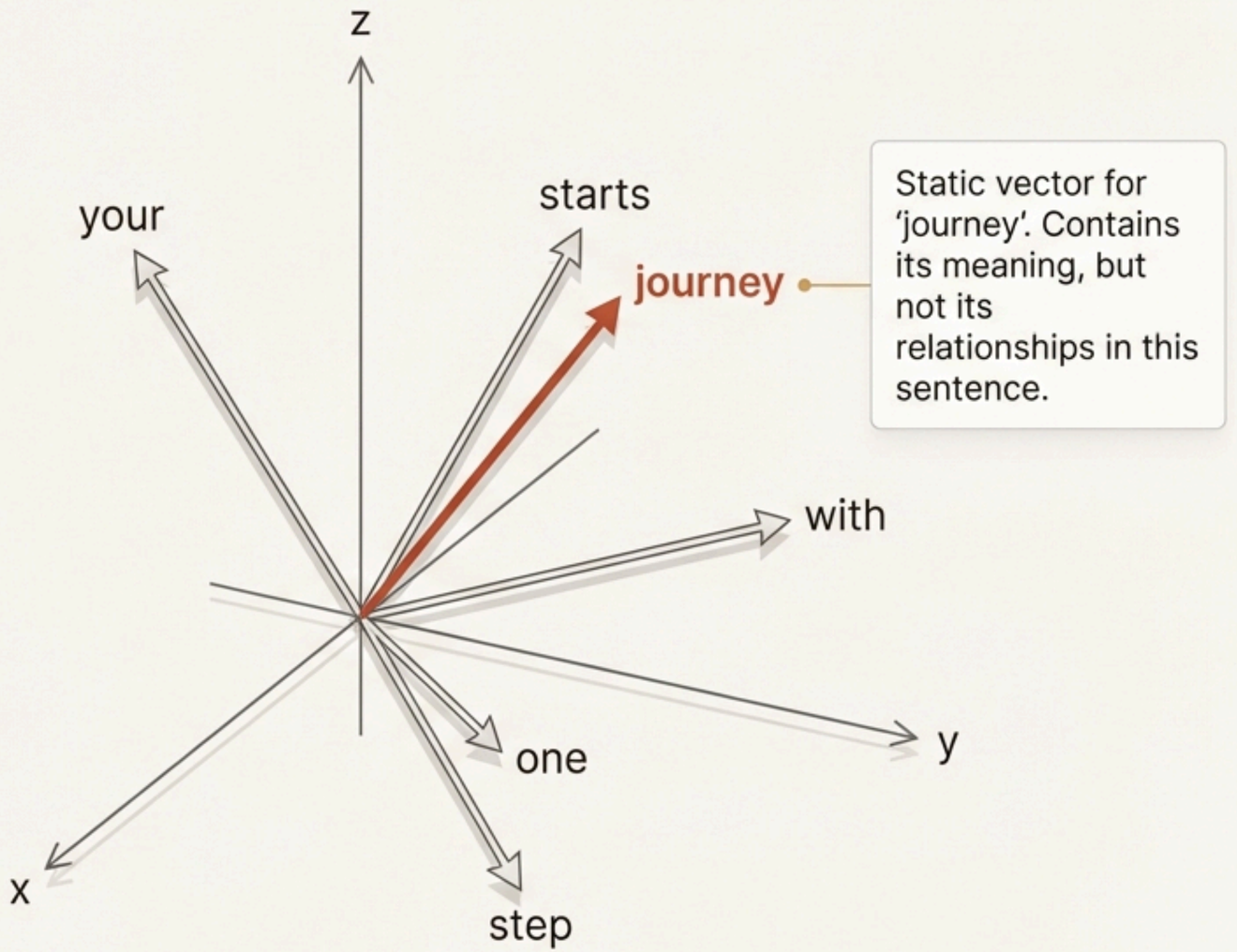
Decoding Self-Attention

From First Principles to the Engine of Transformers

Embeddings Capture Meaning, But Lack Context

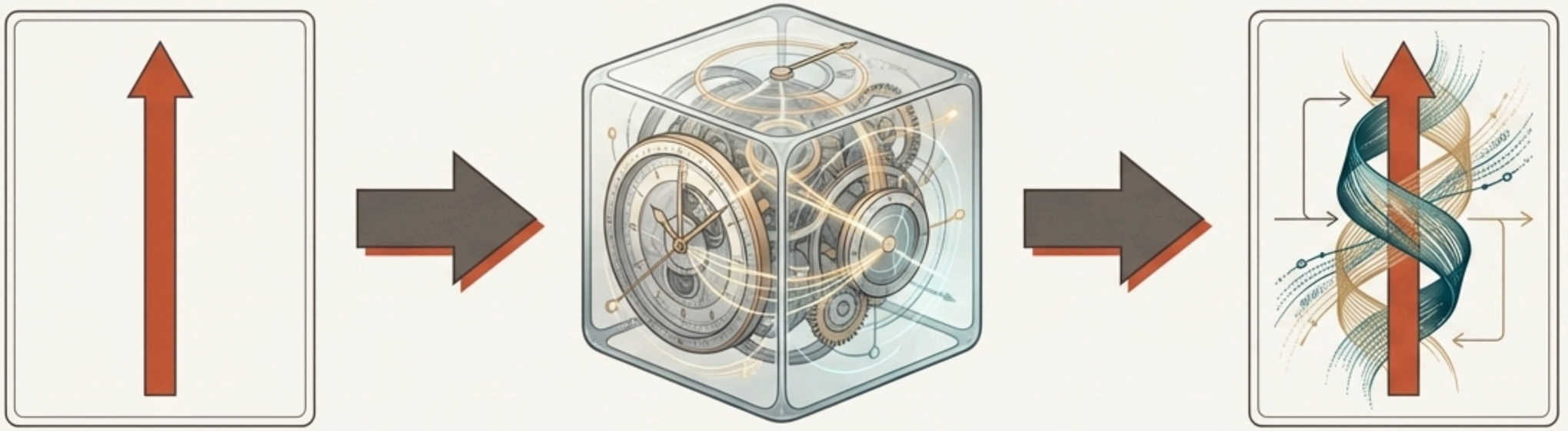
We begin by representing words as vectors, or “embeddings,” in a high-dimensional space. These vectors are powerful because they capture **semantic meaning**—words like “journey” and “starts” are positioned closer to each other.

The Problem: The vector for “journey” is static. It has no information about the other words in the sentence ***“Your journey starts with one step.”*** It is context-blind, unable to understand how “journey” relates to “your” or “starts” within this specific phrase.



Our Goal: To Create a Dynamic, Context-Aware Vector.

The purpose of the attention mechanism is to transform each static input embedding into an enriched "Context Vector." This new vector will be dynamic; it will contain information not just about the word itself, but also its learned relationship with every other word in the sequence.



Input Embedding for "journey"

Attention Mechanism

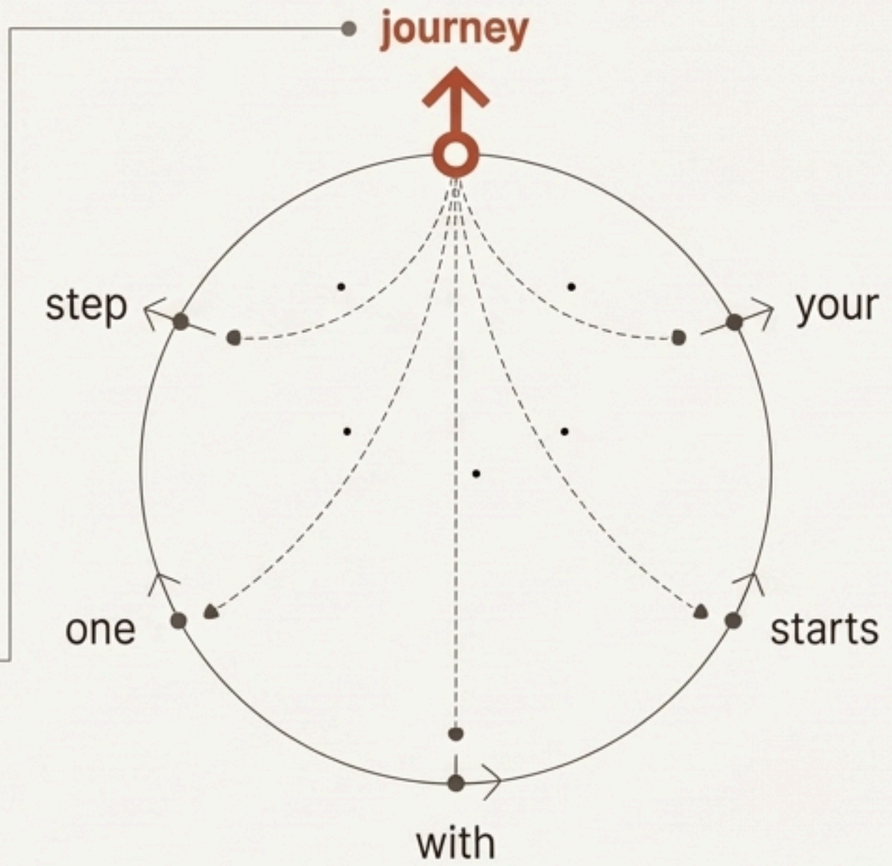
Context Vector

Step 1: Quantifying Relevance with Dot Products.

To build the context for "**journey**" (our **query**), we must first measure its relevance to every other word in the sentence.
An effective way to measure the alignment between two vectors is the **dot product**.

Key Insight: A high dot product signifies high alignment and semantic similarity between two word vectors. A low or zero dot product means the words are unrelated or even orthogonal in meaning.

We can use this to generate raw "Attention Scores."



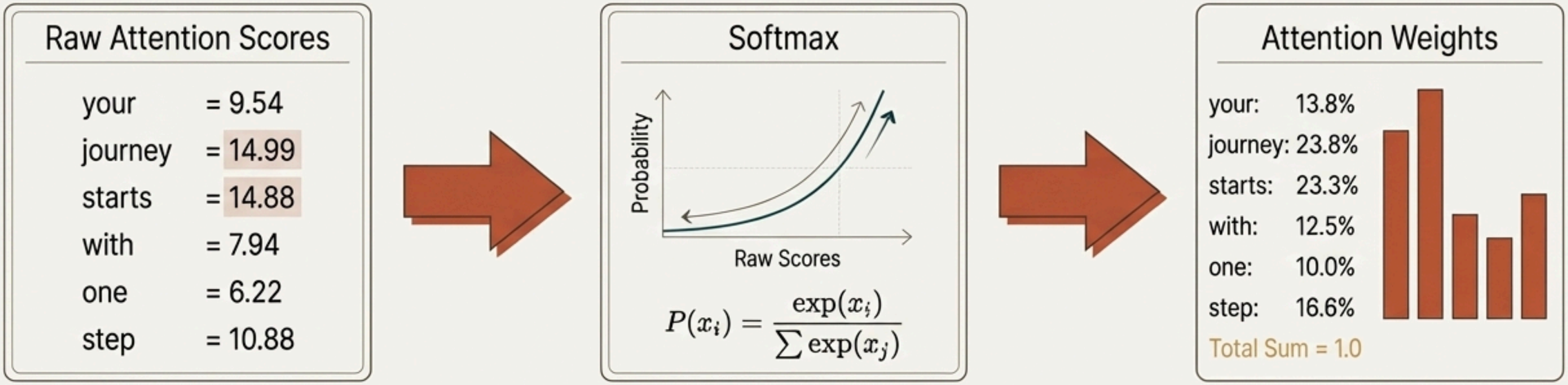
Raw Attention Scores

journey · your	= 9.54
journey · journey	= 14.99
journey · starts	= 14.88
journey · with	= 7.94
journey · one	= 6.22
journey · step	= 10.88

Step 2: Normalizing Scores into Actionable Weights.

The raw attention scores are difficult to interpret. We need to convert them into a probability distribution—a set of weights that sum to 1.0 (or 100%). This tells us exactly what proportion of "attention" our query word should pay to every other word.

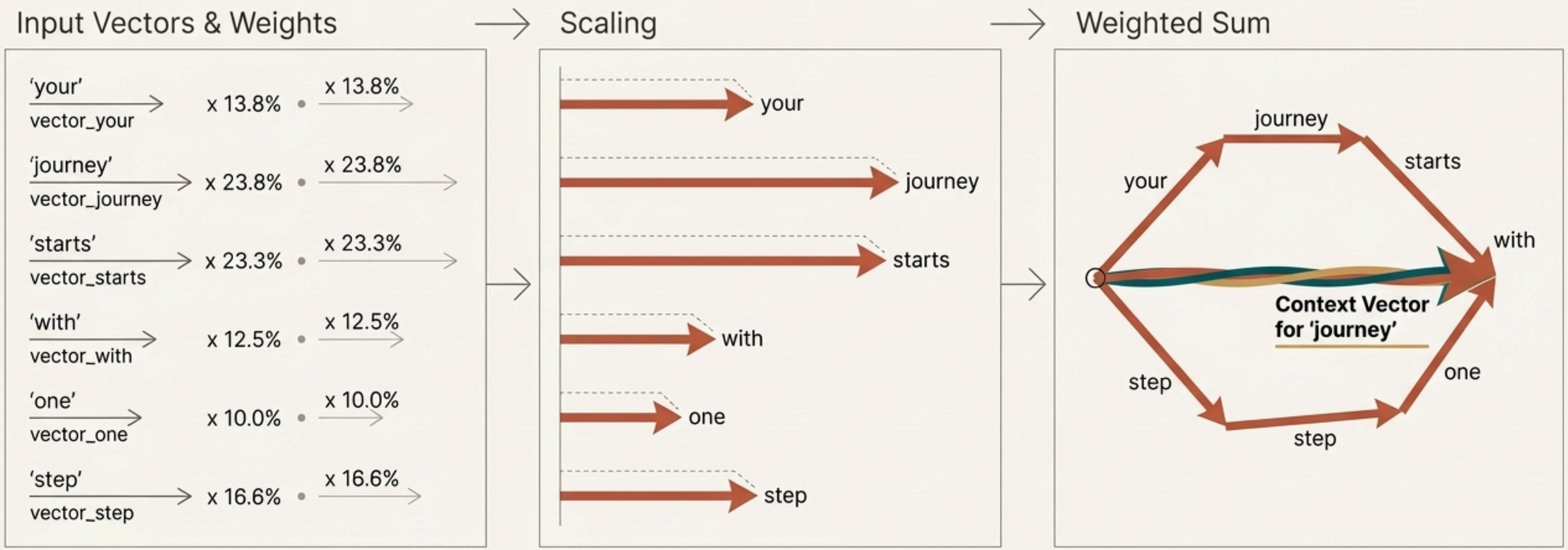
The Tool: The Softmax function is ideal for this. It takes our raw scores, exponentiates them to emphasize higher values, and then normalizes them into "Attention Weights." This is superior to simple division-by-sum because it gracefully handles extreme values, pushing small scores closer to zero and large scores closer to one, which helps prevent optimizers from getting confused during training.



Step 3: Building the Context Vector via Weighted Sum

The final context vector for 'journey' is a weighted sum of all input vectors. We multiply each input vector in the sentence by its corresponding attention weight and then sum the results.

The Result: This creates a new, enriched vector that is pulled in the direction of the words it pays the most attention to. The context vector for 'journey' now contains blended information from 'starts' and other relevant words.



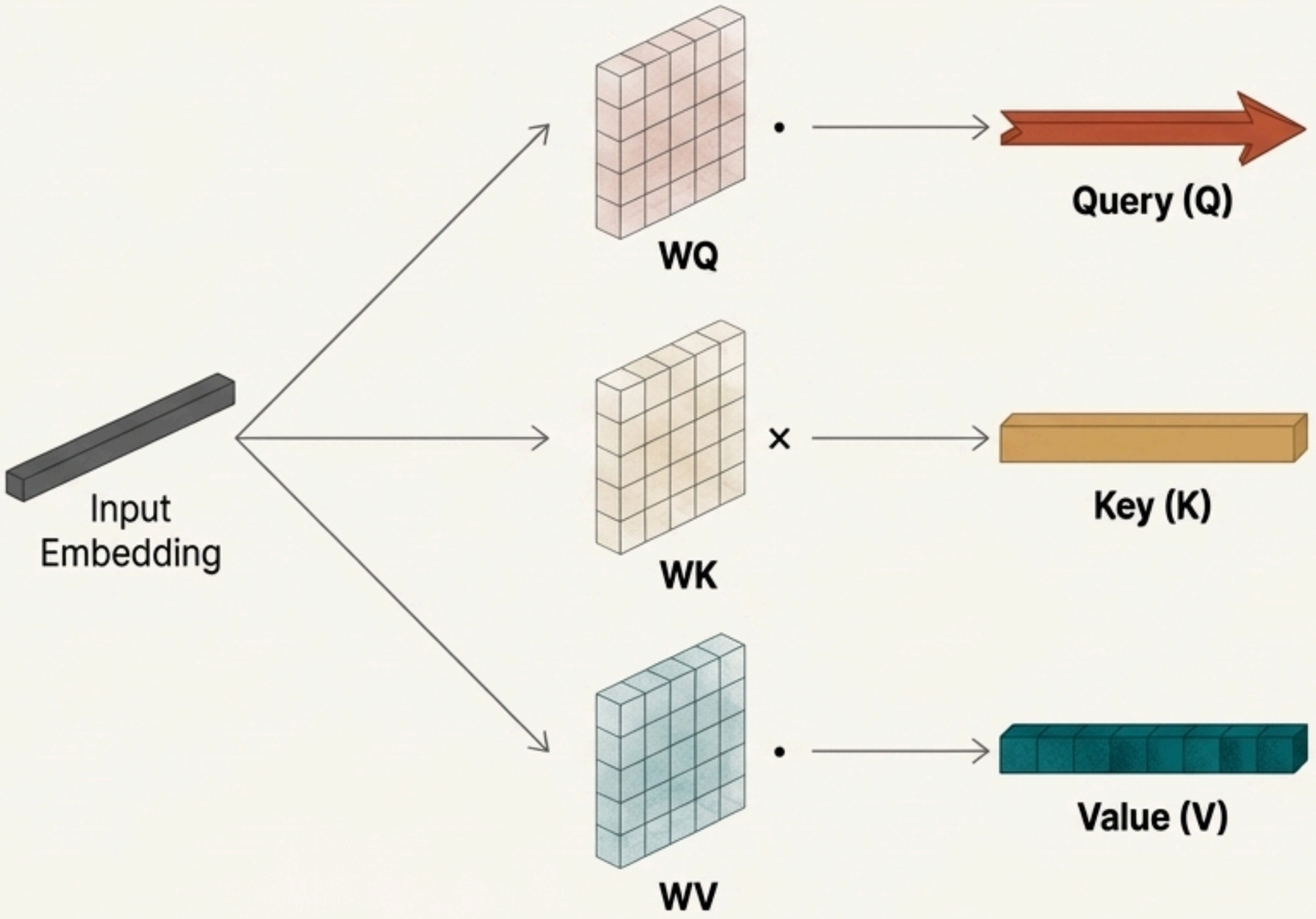
The Simple Model is a Good Start, But It’s Too Rigid.

The Limitation: The simple mechanism bases attention *only* on the pre-existing semantic similarity of the embeddings. It cannot learn new, task-specific relationships.

The Core Problem: For the model to understand this, the word “**it**” needs to attend heavily to “**mat**.” However, the base embeddings for "it" and "mat" are not semantically similar, so their dot product would be low. The simple model fails.

The Need: We need a flexible, *trainable* system that allows the model to learn which relationships matter for a given task, independent of inherent embedding similarity.

The Solution: Projecting Inputs into Three Distinct Roles



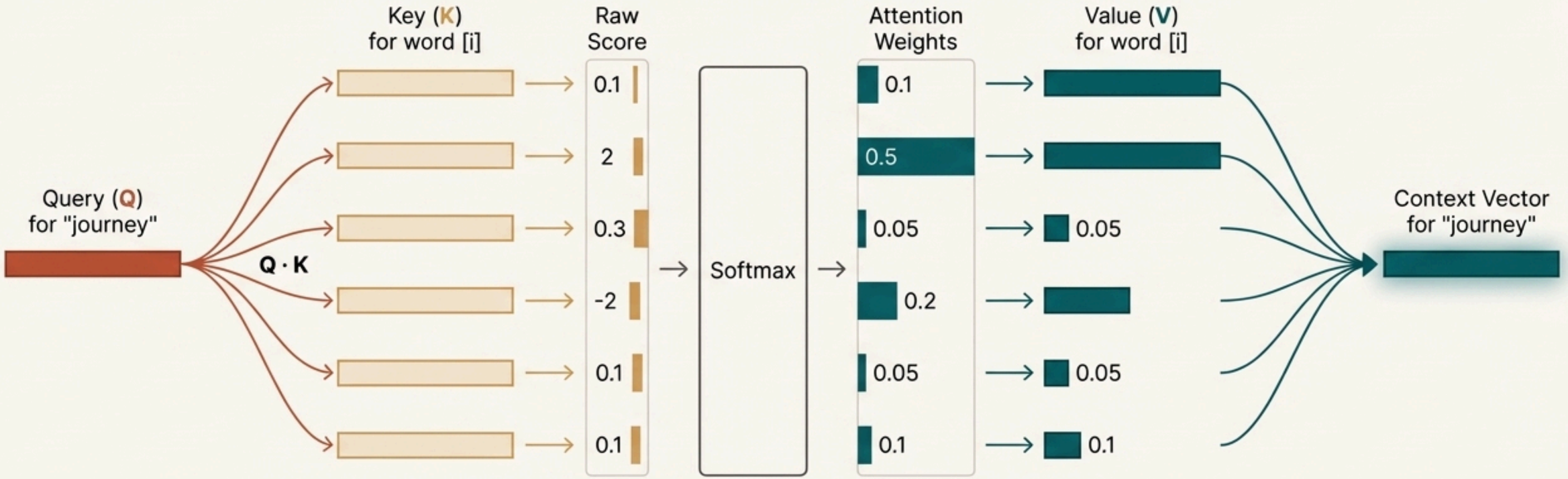
Instead of using the raw input embeddings directly, we introduce three trainable weight matrices: **WQ**, **WK**, and **WV**. We use these to project each input vector into three new, specialized vectors. This gives each word the flexibility to play three different roles:

- **Query (Q):** Represents the current token's focus. It's the probe, asking, *'What am I looking for?'* (colored in #C85A3A)

- **Key (K):** Acts as a label for each token in the sequence. It responds to the query, saying, *'Here's what I have.'* (colored in #D4A05D)

- **Value (V):** Contains the actual content or representation of the token. It's the substance that gets passed along, saying, *'If you find me relevant, here is the information I will provide.'* (colored in #006A70)

The Full Mechanism: Putting Q, K, and V to Work.



1. **Calculate Scores*: Instead of `input · input`, we now calculate the dot product of the current word's **Query (Q)** with the **Key (K)** of every other word in the sequence.
`Scores = Q · K^T`

2. **Normalize*: The scores are passed through a Softmax function to get the attention weights, just as before.

3. **Create Context Vector*: We compute a weighted sum of the **Value (V)** vectors, not the original input vectors.
`Context Vector = Weights · V`

A Critical Detail for Stability: Scaling the Scores

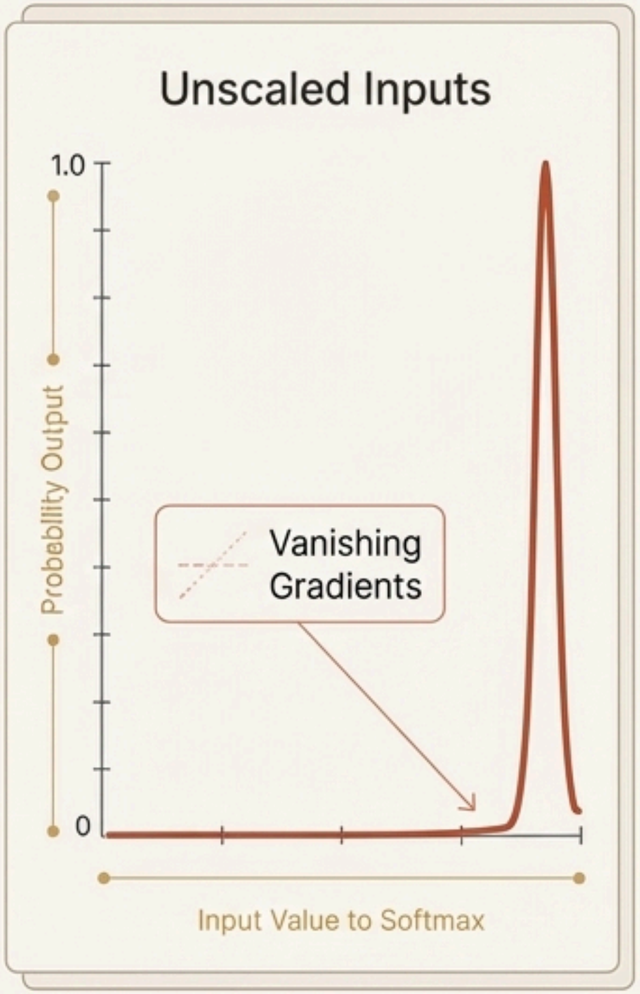
The Problem

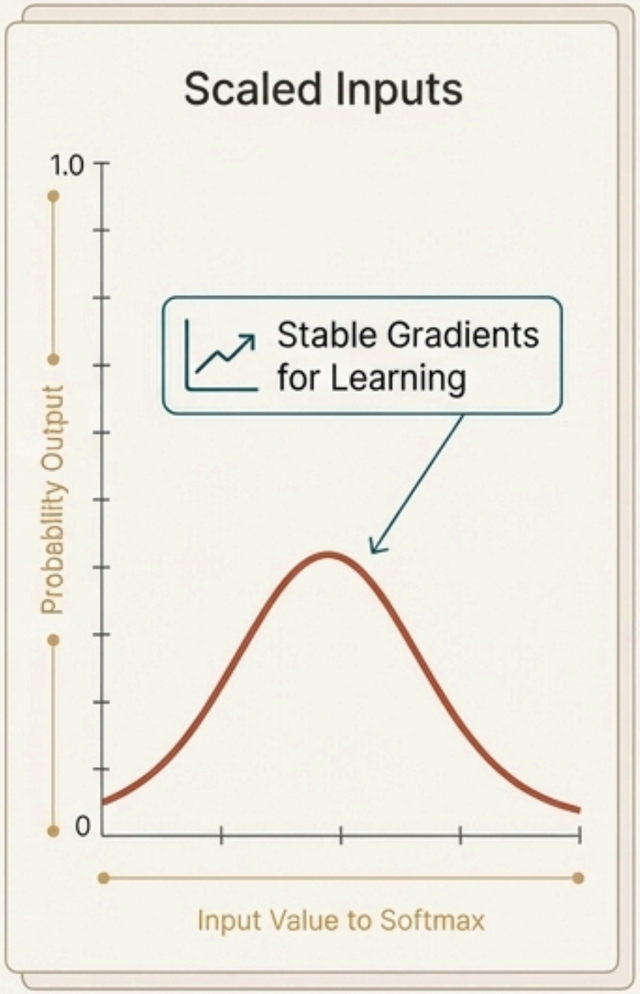
As the dimension of the embeddings (d_k) grows, the magnitude of the dot products can become very large. Large inputs cause the Softmax function to become "spiky"—pushing one value close to 1.0 and all others to 0. This creates extremely ramely small gradients, making training unstable and ineffective.

The Solution

We scale the scores *before* the Softmax function by dividing them by the square root of the dimension of the key vectors ($\sqrt{d_k}$). This simple step keeps the variance of the scores close to 1, regardless of the embedding dimension, ensuring stable gradients and effective learning.

$$\text{Scores} / \sqrt{d_k}$$





The Complete Formula: Scaled Dot-Product Attention.

1. Score:
Calculate the relevance of each Query against all Keys.

3. Normalize:
Convert scores into a probability distribution (attention weights).

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q} \cdot \mathbf{K}^T}{\sqrt{d_k}}\right) \cdot \mathbf{V}$$

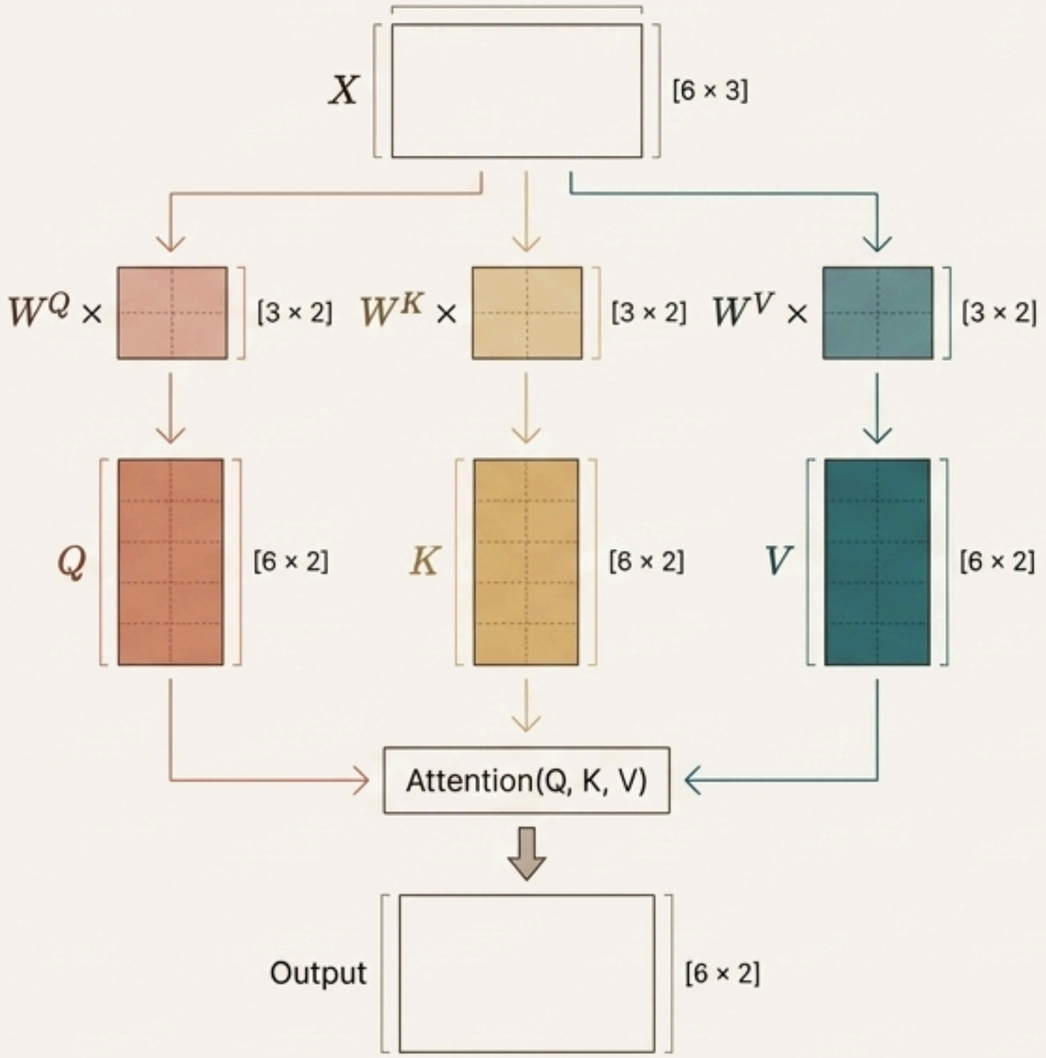
2. Scale:
Divide by the square root of the key dimension to stabilize gradients.

4. Weight & Sum:
Create the final context vector by taking a weighted sum of the Value vectors.

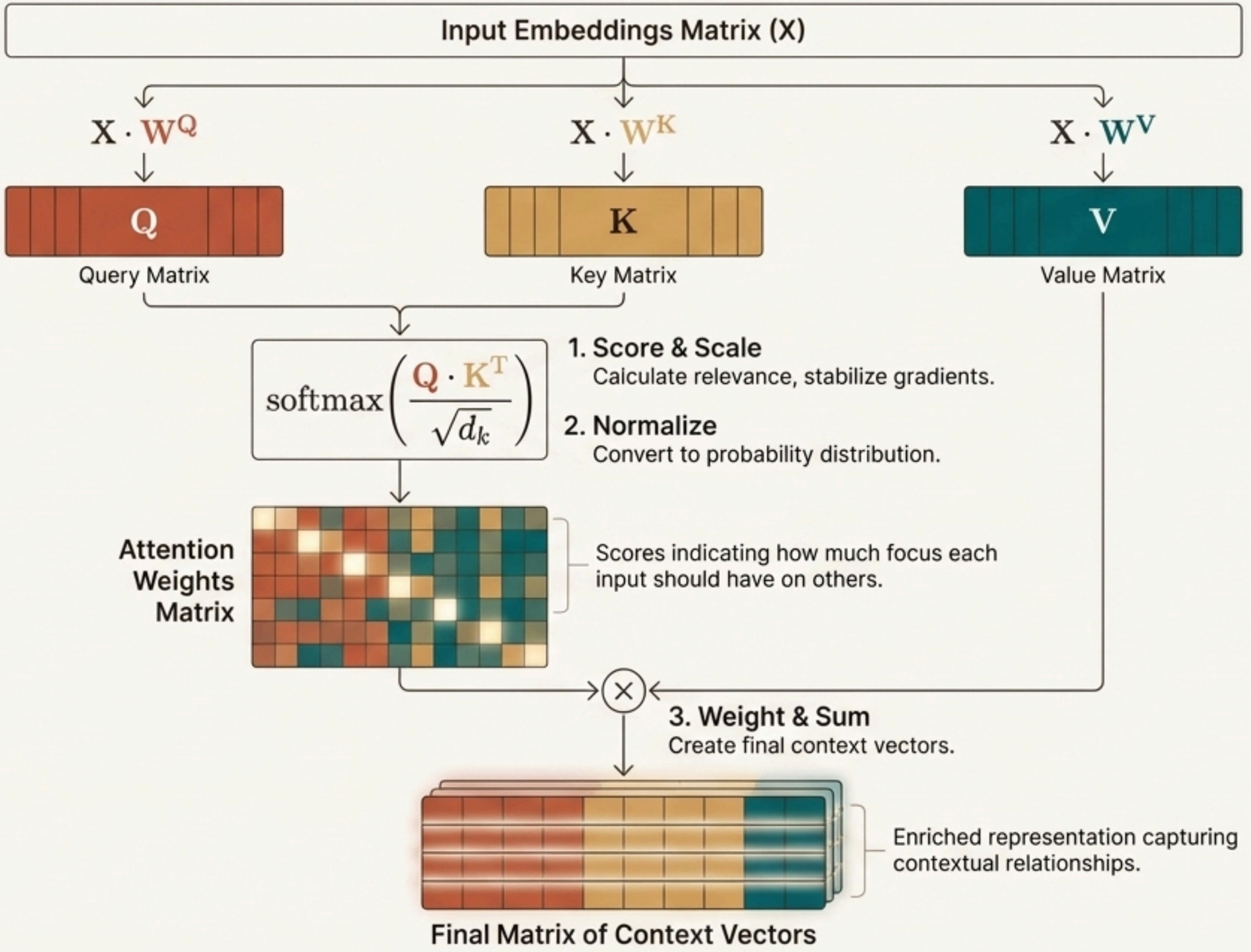
From a Single Vector to a Full Sentence in One Shot.

In practice, we don't calculate attention for one word at a time. We stack all our input embeddings into a single matrix X. Then, the projections to Q, K, and V happen simultaneously through highly efficient matrix multiplication ($Q = XWQ$, $K = XWK$, $V = XWV$).

The Power of Matrices: The entire Scaled Dot-Product Attention formula is then applied to these Q, K, and V matrices at once. This allows the model to compute context vectors for every word in the sequence in a single, parallelized operation.



Self-Attention: The Full Picture

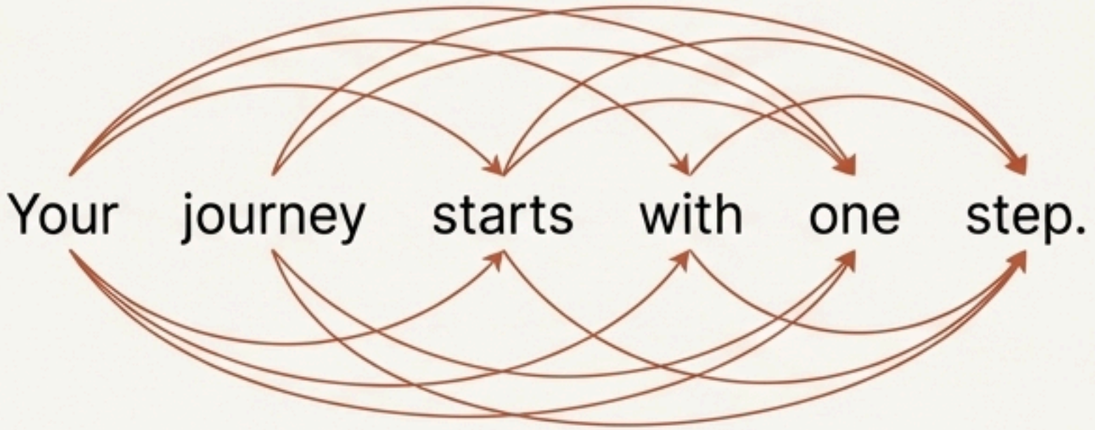


The Engine of the Transformer Architecture

Why “Self”-Attention?

It’s called “self-attention” because the Queries, Keys, and Values all originate from the *same* input sequence. The sequence is looking within itself to determine the context for each of its own words.

This is the mechanism that allows models to understand grammar, resolve ambiguity (like the word "it”), and capture long-range dependencies.



The Impact

Scaled Dot-Product Attention is the core component of the Transformer. Stacking these attention layers (in a structure called “Multi-Head Attention,” which is just running this process multiple times in parallel) allows models like GPT to build incredibly rich, hierarchical understandings of language.