

The LLM Series Notebook version 1.0

Simplifying Language Models



Introduction to the LLM Series

Welcome to the LLM Series, a curated collection of articles designed to simplify the complexities of Large Language Models (LLMs). This notebook brings together my published articles into one cohesive and accessible format, making it easier for readers to understand, explore, and apply the concepts of LLMs.

Each article in this series focuses on a specific aspect of LLMs, presented in a clear and relatable way to make even the most advanced topics easier to grasp. Through real-world examples, analogies, and structured explanations, the series caters to learners at all levels.

The notebook includes the following articles:

- **The Future of LLMs Lies in Domain-Specificity:** Exploring why domain-specific models are the next step in the evolution of LLMs.
- **How an LLM Works: A Simple Way to Explain It:** Breaking down the inner workings of LLMs using relatable examples.
- **Understanding Parameters and Tokens in an LLM: A Simple Breakdown:** Demystifying tokens, parameters, and their relationships in LLMs.
- **Understanding LLMs: How Do They Collect Data?:** Explaining the data collection and training processes behind LLMs.
- **Introduction to the Customization of LLMs:** A guide to creating specialized sub-models with RAG, fine-tuning, and custom GPT.
- **Comparison Between RAG, Fine-Tuning, and Custom GPT:** Highlighting the differences and use cases of these customization techniques.
- **The LLM Inc:** A creative analogy that explains LLM operations as a well-structured company with specialized departments.

This notebook will continue to grow as I publish new articles on LinkedIn. Each update will enhance the series, ensuring it stays relevant and comprehensive.

Whether you're a beginner exploring the basics or a professional looking for advanced insights, the LLM Series offers a structured and engaging journey into the world of language models.

Assem Abdullah Hijazi

[Assem Hijazi | LinkedIn](#)



Contents

1. The future of LLMs lies in domain-specificity.....	5
1. Challenges with Generic LLMs:	5
2. The Need for Specialization:	5
3. The Role of Retrieval-Augmented Generation (RAG)	6
4. Fine-Tuning for Specific Applications	6
5. Data Quality and Quantity:.....	6
2. How an LLM Works: A Simple Way to Explain It.....	7
So, How Does It All Work?.....	7
The Role of Training Data: The LLM’s Rulebook	8
Why the LLM Doesn’t Access Training Data in Real-Time	9
Putting It All Together:.....	9
Understanding Parameters and Tokens in an LLM: A Simple Breakdown	10
Going a Bit Deeper: Vector Representation.....	11
3. Understanding Parameters and Tokens in an LLM: A Simple Breakdown	12
Tokens and Relationships	12
Visualizing the Relationships	12
Changing Contexts and Relationships.....	12
The Role of Training Data in Building These Relationships	13
In Summary	13
Going a Bit Deeper: Vector Representation.....	13
4. Understanding (LLMs): How Do They Collect Data?	14
How Does an LLM Collect Data?	14
Looking Ahead: Retrieval-Augmented Generation (RAG)	15
5.Introduction to the Customization of LLMs	16
Master LLM and Custom Sub LLMs: How They Work Together	17
Comparison Table: RAG vs. Custom GPT.....	19
Comparison Table: RAG vs. Fine-Tuning	20
Comparison Table: Fine-Tuning vs. Custom GPT	21
Master LLM vs. Sub LLMs (or Child LLMs):	22
Important Distinction:	22
How Fine-Tuning and RAG Create Sub LLMs:	22
Why the Master LLM Needs Strong Domain Knowledge:.....	22
Summary:.....	23
6. Comparison Between RAG, Fine-Tuning, and Custom GPT.....	24
RAG vs. Custom GPT	24
RAG vs. Fine-Tuning.....	25

Fine-Tuning vs. Custom GPT	26
7. The LLM Inc	27
The departments details.....	27
1. Tokenization Department.....	27
2. Embedding Department	28
3. Attention Department.....	29
4. Processing Department	30
5. Decision Department	31
The Three Pillars of Data Storage in LLM Inc.....	32
Understanding Q, K, and V Vectors.....	33

1. The future of LLMs lies in domain-specificity

The future of LLMs lies in domain-specificity, RAG integration, and careful fine-tuning. While the generation of text, images, and video might tolerate minor errors, LLMs that contribute to decision-making in critical fields must be robust, safe, and regularly updated to prevent dangerous outcomes. The evolution of AI should be focused on minimizing risks while maximizing efficiency in specialized domains.

The key points are:

1. Challenges with Generic LLMs:

- **Scalability Issues:** As generic LLMs grow in size and complexity, they become more prone to errors and require significant computational resources.
- **Decision-Making Risks:** While errors in text, images, or video generation (e.g., hallucinations or slight inaccuracies) can be relatively minor, errors in decision-making processes (e.g., in IoT systems managing safety or operational workflows) could lead to severe consequences. As domain specific LLMs participate in critical decision-making tasks, the margin for error becomes much smaller.
- **Model Integrity:** As LLMs expand in usage, model integrity becomes a priority. Regular audits, continuous monitoring, and ethical usage policies will be required to prevent harmful or biased outcomes, especially when models are involved in decision-making processes.

2. The Need for Specialization:

- **Generic vs. Domain-Specific Models:** As LLMs grow in size and capability, the current trend of expanding parameters for generic models has proven effective for general knowledge tasks. However, domain-specific LLMs are emerging as more reliable and efficient in specialized fields like law, medicine, and finance. These models are fine-tuned on highly relevant data, improving both accuracy and relevance.
- **Contextual Understanding:** Domain-specific LLMs can be trained on vast amounts of data specific to a particular field, enabling them to understand nuances, jargon, and context much better than generic models.
- **Task-Specific Optimization:** These models can be fine-tuned for specific tasks within a domain, improving their accuracy and efficiency for those applications.

3. The Role of Retrieval-Augmented Generation (RAG)

- **Enhancing Accuracy:** RAG involves incorporating real-time retrieval of external, verified information into the generation process. By combining this with domain-specific LLMs, the future will likely lean towards real-time augmentation of knowledge, ensuring that responses are not only accurate but up-to-date.
- **Reducing Risks:** Instead of relying solely on the model's internal knowledge (which may become outdated or irrelevant), RAG reduces the risk of error by dynamically retrieving external information.

4. Fine-Tuning for Specific Applications

- **Customizing Models:** Fine-tuning allows LLMs to be adapted for specific industries and use cases. As more industries adopt LLM-based solutions, the need for fine-tuning will increase, ensuring the models perform optimally for specific applications
- **Ethical and Safety Considerations:** With more domain-specific applications, models will also require fine-tuning to comply with ethical standards, privacy laws, and operational safety measures relevant to particular sectors.

5. Data Quality and Quantity:

- **Data Quantity:** While the quantity of data is important, quality often outweighs quantity. A smaller, well-curated dataset can yield better results than a larger, noisy one. The quality of the data used to train LLMs is crucial. Domain-specific models can be trained on curated datasets that are more relevant and accurate.

“The digital human system architecture is designed to support each individual and each country independently, using domain-specific language models.”

2. How an LLM Works: A Simple Way to Explain It

I need with this quick article to simplify the understanding of the LLM, when it is still complicated to explain. I tried to simplify it as much as possible and give examples to make it more understandable.

Okay, so imagine that when you ask an LLM (a language model) a question—let's say, something like "What's the population of Paris and how does it affect the economy?"—you're actually giving it a project to work on.

Now, this project isn't handled by one big brain, but instead by a bunch of teams working together.

Each of these teams is made up of layers of the LLM. These layers are like groups of workers, and they all have different roles in breaking down your question and figuring out the best answer. Some teams handle simple tasks, while others are more skilled for tougher projects.

So, How Does It All Work?

1. **The LLM Gets Your Question** (The Project) First off, you give the model a project by asking a question. It could be something simple like *"Is Paris the capital of France?"*, or it could be more complicated like the example above with Paris's population and the economy.
2. **Teams Start Analyzing** (Layers Jump In) Once the question (the project) is in, the different teams (layers) get to work. If it's a simple question, maybe only a few teams are needed. But for more complex questions, more teams get involved. Each team (layer) looks at a different part of the question, passing their results to the next team.
3. **Building Relationships Between Words** (Parameters) While the teams are working, they use parameters—which are basically the connections "relationship" between words they've learned from past projects (training). So, if the model has seen "Paris is the capital of France" hundreds of times in the past, it knows these words have a strong connection. Parameters are what make these links happen.
4. **Spotting Patterns** "keywords", As the teams work through the project, they start noticing patterns in the question. For example, if you ask about the population and the economy of Paris, the LLM recognizes that these two concepts are related—it has seen them linked before. So, it follows these patterns to figure out the best way to answer your question.
5. **No Big Boss Making Decisions** Now, you might think there's some kind of central brain inside the model making decisions, but that's not how it works. There's no boss calling the shots. Instead, the layers (teams) work together, and they all adjust what they're doing based on the relationships they've learned. They cooperate and share information, but there's no single decision-maker.
6. **Back and Forth Adjustments (Feedback)** If the teams start heading in the wrong direction, they adjust their approach. This is where **backpropagation** "feedback between teams" comes in during the training process. It's a fancy way of saying the model learns from its mistakes—if something isn't right, it tweaks how the teams handle things so they get better next time.

7. **More Complicated Questions Need More Teams (Layers)** For something like “*Paris is the capital of France?*”, it’s a simple project, so fewer teams are involved because the model already knows the answer well. But when you ask a more complex question, like “*How does Paris’s population affect its economy?*”, the model brings in more teams (layers) and uses more parameters to connect all the pieces. The harder the question, the more layers work together to figure it out.
8. **Training Data is the Rulebook** All of this—how the layers work together, how the parameters are set up—comes from the training data the LLM has been fed. Think of the training data as a big rulebook. The more accurate the rulebook, the better the model’s understanding. If the model has seen enough reliable data, it knows that “Paris” and “France” are linked and understands more complex things like how population affects the economy.
9. **Delivering the Final Answer** After all the teams have done their work, the LLM puts together an answer. For a simple question, this happens quickly with fewer steps. For complex questions, it takes more steps and more teamwork between the layers to figure out the right connections and deliver a thoughtful response.

This way of thinking about LLMs makes the whole process easier to grasp: each prompt is like a project, and the model’s layers are teams working together, adjusting and learning as they go, without any one boss calling all the shots.

The Role of Training Data: The LLM’s Rulebook

Now, let’s talk about the **training data in more details**, which is the foundation of everything the LLM knows. Think of the training data as the LLM’s **rulebook**—it’s what taught the LLM how words relate to each other and how to handle new projects (prompts).

In short, the **training data** was like the LLM’s teacher. Now that it’s learned everything from that teacher, it can handle new questions based on the **knowledge it has stored** from its training.

Learning from the Rulebook

Before the LLM was deployed, it was trained on massive amounts of text (books, articles, websites). During this training, the model read the data and learned **patterns** and **relationships** between words. This is where it built its **rulebook**.

For example, it learned from this data that “**Paris**” is linked to “**France**” and “**capital**”. The more often it saw these words together, the stronger it understood the connection between them. This process of training is how the LLM filled its **memory** with information that it can use later.

Using the Rulebook Without Going Back to It

Here's an important point: once the LLM is trained, it doesn't need to go back and look at the **training data** every time you ask it a question. It's already **learned** everything it needs from that data—just like how a person doesn't need to re-read a book every time they answer a question about it. Instead, the LLM relies on the **patterns and relationships** it has already memorized.

So, when you ask, *“What's the population of Paris, and how does it affect the economy?”*, the LLM isn't looking up articles or books. Instead, it's using the **knowledge** it gathered from its training data—the **rulebook** it already memorized—to respond to you.

Why the LLM Doesn't Access Training Data in Real-Time

The LLM doesn't need to revisit the training data because:

- **It already learned the material:** The training data taught the LLM how words and concepts fit together. Now, it just uses its memory to apply that knowledge.
- **It's using what's stored in its memory:** Just like how you remember facts after reading a book, the LLM stores the relationships and patterns it learned and applies them in real-time without needing to reference the original data.

Putting It All Together:

- **Prompt = Project:** You give the LLM a project when you ask a question.
- **Teams = Layers:** The layers (teams) work on different parts of the project.
- **Parameters = Relationships:** Parameters are the connections between words, learned from past training.
- **Patterns=keywords:** The teams spot patterns based on what they've seen before.
- **No Central Boss:** No one team is in charge; they all work together without a central decision-maker.
- **Training Data = Rulebook:** The LLM learns from the training data (its rulebook), which helps it provide accurate answers.
- **Complexity:** Simple projects need fewer teams, while complex ones involve more layers and parameters.
- **Training Data = Rulebook:** The LLM learned from the training data, just like how we learn from books, and it doesn't need to keep going back to that data once it's learned the material.

Understanding Parameters and Tokens in an LLM: A Simple Breakdown

We've already established that in an LLM (like GPT), **parameters** are the relationships between words, and **tokens** "they call it tokens" are the words themselves. Now, let's dig a little deeper into how these tokens and parameters work together to help the model understand and respond to prompts.

Tokens and Relationships

First, think of **tokens** as individual **words** or parts of words. Each word has certain **relationships** with other words, and these relationships are built during the model's training. The more training data the model has, the more relationships (or **parameters**) it can establish between different words.

When you give the LLM a **prompt** (we can think of this as a "project"), the model analyzes the relationships between the words in that prompt. It **evaluates** how strongly the words are connected and decides which connections are most relevant. These relationships are measured by **vectors**, which are like numerical scores that show how related two words are. These vectors aren't static—they change dynamically based on the **context** of the prompt.

Visualizing the Relationships

Imagine each word is inside a **circle**, and from that circle, you have multiple **lines** connecting it to other words. Each line represents a potential relationship between the word in the circle and another word. Some of these lines might be **thick and strong** because those words are closely related, while others might be **thin and weak** if the relationship is less relevant.

Note: visit the site for visualize words relationship [Semantically related words for "dubai_NOUN" \(nlpl.eu\)](https://www.nlpl.eu/semantically-related-words-for-dubai-noun)

For example, if you take the word **"bank"** and place it in a circle, you might see lines connecting it to words like **"money," "loan," "river,"** and **"finance."** The strength of these connections depends on the **context**. If you're talking about banking, the connection between **"bank"** and **"money"** will be strong, but if the context is about rivers, the connection between **"bank"** and **"river"** will become stronger instead.

Changing Contexts and Relationships

The beauty of an LLM is that these relationships **adapt** depending on the context of your prompt. Let's say you move from talking about **healthcare** to **finance** and then to **geography**:

- In **healthcare**, the word **"apple"** might be linked closely to words like **"nutrition"** and **"diet."**
- In **finance**, the word **"bank"** might be linked closely to words like **"loan"** and **"interest."**
- In **geography**, **"bank"** might shift its connection to **"river"** and **"water."**

This means that for each new context or domain, the LLM dynamically adjusts the **vectors** (relationships between words) to match the meaning that's most relevant.

The Role of Training Data in Building These Relationships

So, how does the LLM know to adjust these connections? This is where **training data** comes in. During training, the model is exposed to a huge amount of text across different topics and domains. It learns the **patterns** in language and builds initial vectors (or relationships) between words.

At the start, these vectors are **generic** and might even be random. But as the model processes more data, it refines these relationships. For instance, the more the model reads about **banks** and **finance**, the stronger the connection between **"bank"** and **"loan"** becomes.

Once trained, the model doesn't need to go back to the original data every time it gets a new prompt. Instead, it uses what it already learned—just like how you don't need to look up a fact in a book once you've memorized it. The LLM adjusts the relationships between words on the fly, based on the context of the question or prompt.

In Summary

- **Tokens** are words, and the **relationships** between those words are called **parameters**.
- These relationships are represented by **vectors**, which measure how closely two words are related.
- Depending on the **context** of the prompt, the model will adjust the strength of these relationships.
- The **training data** helps the model learn these relationships, so it can respond to different prompts in various domains (like healthcare, finance, or geography) without needing to revisit the original data.

In short, every word is connected to many other words, and the strength of those connections shifts depending on the project or prompt you're working on. It's like having a **web of words**, and the model knows how to highlight the right connections depending on what you ask.

Going a Bit Deeper: Vector Representation

If you're curious about how these relationships are actually represented in the LLM, here's a bit more detail. In models like GPT, every word (or token) is represented by a **vector**—a long list of numbers. These vectors are usually **1,024 dimensions long**, meaning each word is associated with 1,024 different numbers that capture its meaning.

For example:

- **"apple"** might have a vector like **[0.45, -0.32, 0.78, ..., -0.11]**.
- **"fruit"** could have a vector like **[0.42, -0.30, 0.80, ..., -0.10]**. Since **"apple"** and **"fruit"** are related, their vectors will be similar. In this case, the difference between their vectors is small, which shows that the LLM recognizes them as closely related.

These numbers typically range between **-1 and 1** and reflect the word's relationship with other words in the model. Words with similar meanings will have vectors that are closer together in this **high-dimensional space**, while unrelated words will have vectors that are farther apart. The model uses these vectors to understand and adjust the relationships between words based on the context of your prompt.

So, while we visualize these connections as **lines between words in circles**, under the hood, the model is using complex **mathematical relationships** between these **1,024-dimensional vectors** to decide how words are related. This allows the model to adjust dynamically and provide relevant answers, no matter the domain or topic.

3. Understanding Parameters and Tokens in an LLM: A Simple Breakdown

We've already established that in an LLM (like GPT), **parameters** are the relationships between words, and **tokens** "they call it tokens" are the words themselves. Now, let's dig a little deeper into how these tokens and parameters work together to help the model understand and respond to prompts.

Tokens and Relationships

First, think of **tokens** as individual **words** or parts of words. Each word has certain **relationships** with other words, and these relationships are built during the model's training. The more training data the model has, the more relationships (or **parameters**) it can establish between different words.

When you give the LLM a **prompt** (we can think of this as a "project"), the model analyzes the relationships between the words in that prompt. It **evaluates** how strongly the words are connected and decides which connections are most relevant. These relationships are measured by **vectors**, which are like numerical scores that show how related two words are. These vectors aren't static—they change dynamically based on the **context** of the prompt.

Visualizing the Relationships

Imagine each word is inside a **circle**, and from that circle, you have multiple **lines** connecting it to other words. Each line represents a potential relationship between the word in the circle and another word. Some of these lines might be **thick and strong** because those words are closely related, while others might be **thin and weak** if the relationship is less relevant.

Note: visit the site for visualize words relationship [Semantically related words for "dubai_NOUN" \(nlpl.eu\)](https://www.nlpl.eu/semantically-related-words-for-dubai-noun)

For example, if you take the word "**bank**" and place it in a circle, you might see lines connecting it to words like "**money**," "**loan**," "**river**," and "**finance**." The strength of these connections depends on the **context**. If you're talking about banking, the connection between "**bank**" and "**money**" will be strong, but if the context is about rivers, the connection between "**bank**" and "**river**" will become stronger instead.

Changing Contexts and Relationships

The beauty of an LLM is that these relationships **adapt** depending on the context of your prompt. Let's say you move from talking about **healthcare** to **finance** and then to **geography**:

- In **healthcare**, the word "**apple**" might be linked closely to words like "**nutrition**" and "**diet**."
- In **finance**, the word "**bank**" might be linked closely to words like "**loan**" and "**interest**."
- In **geography**, "**bank**" might shift its connection to "**river**" and "**water**."

This means that for each new context or domain, the LLM dynamically adjusts the **vectors** (relationships between words) to match the meaning that's most relevant.

The Role of Training Data in Building These Relationships

So, how does the LLM know to adjust these connections? This is where **training data** comes in. During training, the model is exposed to a huge amount of text across different topics and domains. It learns the **patterns** in language and builds initial vectors (or relationships) between words.

At the start, these vectors are **generic** and might even be random. But as the model processes more data, it refines these relationships. For instance, the more the model reads about **banks** and **finance**, the stronger the connection between **"bank"** and **"loan"** becomes.

Once trained, the model doesn't need to go back to the original data every time it gets a new prompt. Instead, it uses what it already learned—just like how you don't need to look up a fact in a book once you've memorized it. The LLM adjusts the relationships between words on the fly, based on the context of the question or prompt.

In Summary

- **Tokens** are words, and the **relationships** between those words are called **parameters**.
- These relationships are represented by **vectors**, which measure how closely two words are related.
- Depending on the **context** of the prompt, the model will adjust the strength of these relationships.
- The **training data** helps the model learn these relationships, so it can respond to different prompts in various domains (like healthcare, finance, or geography) without needing to revisit the original data.

In short, every word is connected to many other words, and the strength of those connections shifts depending on the project or prompt you're working on. It's like having a **web of words**, and the model knows how to highlight the right connections depending on what you ask.

Going a Bit Deeper: Vector Representation

If you're curious about how these relationships are actually represented in the LLM, here's a bit more detail. In models like GPT, every word (or token) is represented by a **vector**—a long list of numbers. These vectors are usually **1,024 dimensions long**, meaning each word is associated with 1,024 different numbers that capture its meaning.

For example:

- **"apple"** might have a vector like **[0.45, -0.32, 0.78, ..., -0.11]**.
- **"fruit"** could have a vector like **[0.42, -0.30, 0.80, ..., -0.10]**. Since **"apple"** and **"fruit"** are related, their vectors will be similar. In this case, the difference between their vectors is small, which shows that the LLM recognizes them as closely related.

These numbers typically range between **-1 and 1** and reflect the word's relationship with other words in the model. Words with similar meanings will have vectors that are closer together in this **high-dimensional space**, while unrelated words will have vectors that are farther apart. The model uses these vectors to understand and adjust the relationships between words based on the context of your prompt.

So, while we visualize these connections as **lines between words in circles**, under the hood, the model is using complex **mathematical relationships** between these **1,024-dimensional vectors** to decide how words are related.

This allows the model to adjust dynamically and provide relevant answers, no matter the domain or topic

4. Understanding (LLMs): How Do They Collect Data?

I'll continue writing about LLMs (Large Language Models) to help explain how they work in simple terms. With each article, we'll dive a bit deeper to build a solid understanding, particularly of **generic LLMs**. We'll also explore their limitations.

How Does an LLM Collect Data?

LLMs like GPT are trained on massive amounts of data, which helps them understand and generate human-like text. But how exactly do they collect this data, and where does it come from?

1. Data Collection Methods

LLMs don't actively browse the web to gather information in real-time. Instead, during their training phase, they are exposed to large datasets that are compiled from publicly available sources. These sources may include:

- **Web pages:** Publicly accessible websites that are scraped for training purposes.
- **Books and research articles:** Digitized books and academic papers provide structured knowledge.
- **Social media:** Public posts and conversations that reflect everyday language.
- **Documents and public reports:** Official documents, Wikipedia entries, and other publicly available knowledge repositories.

This process, known as **scraping**, involves collecting and organizing huge amounts of data from these sources to create the model's training dataset.

2. Does the LLM Store Source Information?

When an LLM is trained, it processes a lot of data from various sources. However, the important thing to know is that the LLM doesn't actually **store the original sources**. Instead, it learns **patterns and relationships between words** during training and uses those to generate its responses.

3. Data Usage: Datasets, Not Direct Sources

Once the data has been collected, the LLM doesn't store it as an exact copy. Instead, it's processed into **datasets** that allow the model to learn **language patterns**. The LLM doesn't go back and pull specific content from these datasets when answering questions. Rather, it has already absorbed the patterns and generates responses based on what it learned during training.

4. The Problem with Trustworthiness

Not all data is created equal. LLMs are trained on a wide range of topics to build a broad understanding of language, but the **trustworthiness** of the sources they are trained on can be a critical issue. Since LLMs pull from a vast range of data, including **unverified sources**, the accuracy of the information can vary.

- Because the LLM doesn't store or track the original sources, there's no way to **verify** where it got its information from. This means that sometimes, the model might provide an answer that sounds confident but could be based on **unreliable** or uncertified data.

- For instance, the LLM might have learned something from a **reliable academic journal**, but it could also have learned from a **random blog**. Without tracking sources, you can't always be sure of the **accuracy** or **trustworthiness** of the answer.

5. Mathematical Data Structures in LLMs

The LLM is built on a complex **mathematical structure** made up of **tokens**, **vectors**, and **parameters**. Each word (or token) is represented by a **vector**—a series of **1,024 numbers**—which changes dynamically based on the input and the relationships between words. The LLM processes these relationships through **multiple layers**, refining them to generate responses that fit the context of the prompt.

Looking Ahead: Retrieval-Augmented Generation (RAG)

New techniques like **Retrieval-Augmented Generation (RAG)** aim to overcome some of the limitations of LLMs by allowing them to retrieve and incorporate **real-time information** from external sources. This helps keep the data up to date and more relevant, addressing some of the challenges with **outdated** or **unverified data**.

In the next article, we'll dive into **RAG** and **fine-tuning** in simple terms, explaining how they enhance LLMs and make them more effective in specialized contexts.

5.Introduction to the Customization of LLMs

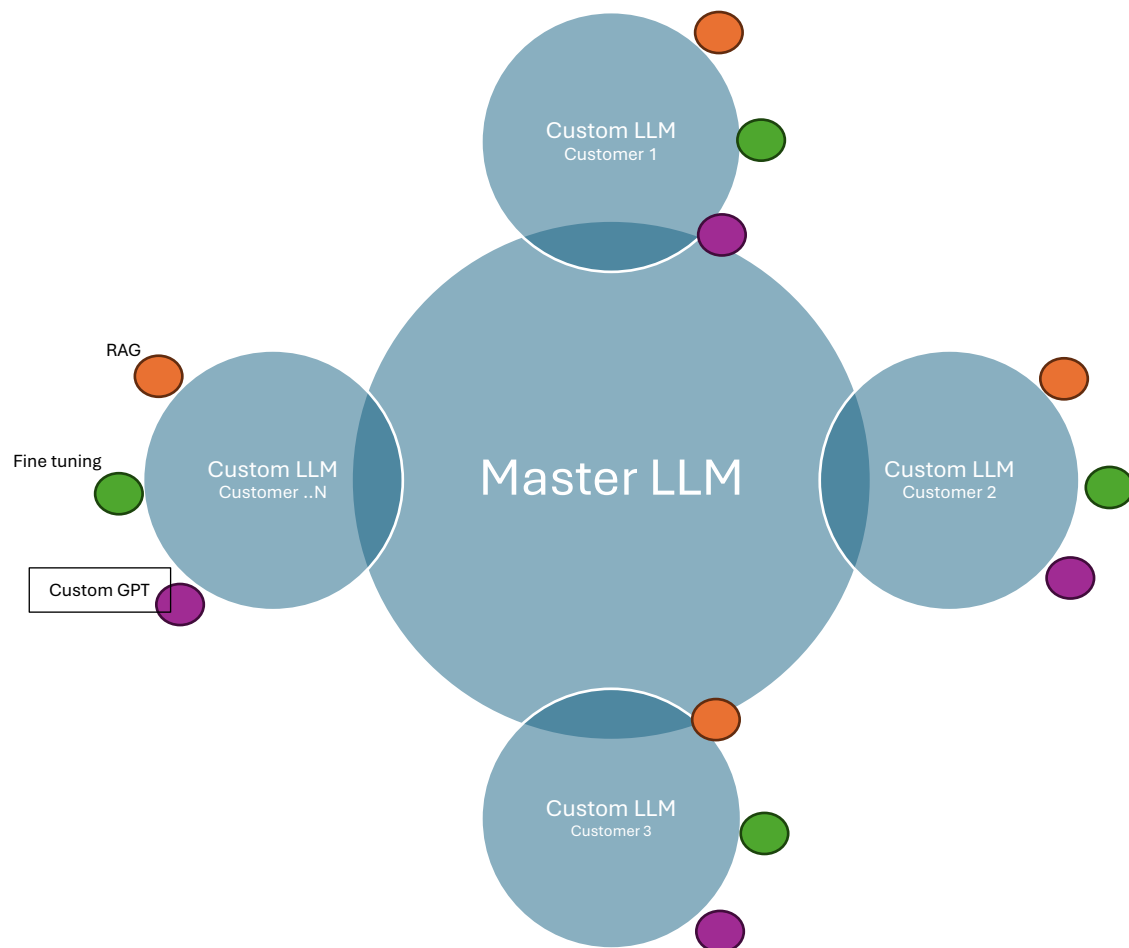
In modern large language models (LLMs) like ChatGPT, customization plays a key role. While the master LLM—the main model—comes pre-trained with a broad set of knowledge, users often need the model to focus on their specific needs, whether it's handling their own data, providing specialized information, or answering domain-specific questions. However, when users request these custom capabilities, they do so without changing the master LLM itself. Instead, the master LLM provides technologies that allow for these custom needs to be met while keeping the core model intact.

When customers or organizations need the LLM to focus on their unique data or knowledge, they can rely on the master LLM for support. The master LLM provides three main customization methods: RAG, fine-tuning, and custom GPT. These methods help users build what we call custom sub LLMs (or child LLMs), tailored to handle their specific requirements.

Think of the master LLM as a main brain with a wealth of solid, pre-trained knowledge that doesn't change “unless done by the main owner of the master LLM”. It is designed using mathematical structures, algorithms, and static training data. But it's also flexible enough to assist users in creating custom LLMs for their own needs, based on external data or specific requests.

It's important to note that when the master LLM helps build custom LLMs, it does so without altering its own core knowledge base or thinking. Instead, it supports these child LLMs by using its existing capabilities to interpret new data, answer specific queries, or integrate user-provided information. It's as if the master brain is lending its thought process to assist in handling specialized tasks, but it doesn't change its own thinking—it's more like an assistant that helps others with custom tasks while keeping its own knowledge intact.

This approach allows users to build small, specialized LLMs for their own purposes, such as answering questions or managing specific types of data. These child LLMs work together with the master LLM to offer precise, personalized responses based on the user's needs.



Master LLM and Custom Sub LLMs: How They Work Together

The Master LLM (represented as a large central circle) is the core model, like ChatGPT's master LLM, which contains general-purpose knowledge trained across a wide variety of topics and domains. This master LLM remains unchanged and serves as the base for all interactions, ensuring that the model can respond across many different areas without being specifically fine-tuned for one.

Around the master LLM are customer-specific sub LLMs (represented by smaller connected circles). Each customer-specific LLM is tailored for a particular user, business, or organization. These custom sub LLMs do not modify the master LLM, but rather, they are created to address specific needs through three key methods RAG (Retrieval-Augmented Generation), Fine-Tuning and Custom GPT.

Each customer-specific sub LLM acts as a child LLM, integrating seamlessly with the master LLM while maintaining its own specialization. This flexible architecture allows customers to create powerful, tailored solutions for their own unique needs, without impacting the broader model's general knowledge and capabilities.

The main definitions

1. Retrieval-Augmented Generation (RAG):

- What it is: RAG retrieves real-time data from external sources, like databases or documents, to enhance the LLM's responses. However, it does not modify the master LLM itself. Instead, it creates

a custom-specific sub LLM or child LLM that pulls external data in real-time to provide accurate, up-to-date answers.

- **How it works:** The master LLM processes the initial query, and then the RAG component retrieves relevant data from external sources. This data is combined with the pre-existing knowledge of the master LLM to generate a response. The real-time data retrieval allows for dynamic responses without permanently altering the core LLM.
- **In simple terms:** RAG is like adding a "live search engine" to the LLM that helps it access and use real-time information for specific queries, while creating a custom sub LLM to handle the specific needs of that task.
- **Key point:** RAG does not update the master LLM, but instead creates a child LLM that integrates real-time data retrieval.
- **Example-Healthcare Application:** A custom sub LLM for a medical institution retrieves real-time patient records and the latest medical research papers to generate responses. The custom sub LLM created with RAG doesn't change the master LLM but accesses relevant external data for more accurate answers

2. Fine-Tuning:

- **What it is:** Fine-tuning involves training a model on specific datasets to make it more specialized for certain tasks. However, it does not modify the master LLM. Instead, it creates a custom-specific sub LLM (or child LLM) that is specialized in the area it was fine-tuned on.
- **How it works:** Developers provide specialized data to fine-tune a model on a specific domain or topic. The resulting custom sub LLM can now handle queries within that specialized area but doesn't alter the core model.
- **In simple terms:** Fine-tuning is like giving the LLM a focused training course that creates a new, specialized child LLM for specific tasks, but without changing the master LLM.
- **Key point:** Fine-tuning creates a custom sub LLM for specific tasks and does not modify the master LLM.
- **Example-Legal Application:** A law firm fine-tunes a model on a dataset of past legal cases to create a custom sub LLM that specializes in legal queries. This fine-tuned model does not update the master LLM, but it becomes an expert in handling specific legal queries.

3. Custom GPT:

- **What it is:** Custom GPT allows users to define specific behaviors, rules, and responses for the LLM without changing its underlying architecture or training. This creates a custom-specific sub LLM that adheres to those user-defined rules and documents.
- **How it works:** Users upload custom documents or define rules that guide the model's responses, creating a custom GPT model that operates based on the user's needs, but again, it doesn't modify the master LLM.
- **In simple terms:** Custom GPT is like setting up a custom rulebook that the LLM follows for specific queries, but it doesn't alter the core model or its original knowledge.

- **Key point: Custom GPT creates a custom sub LLM with user-defined rules, separate from the master LLM.**
- **Example-Retail Company:** A retail company uses custom GPT to create a custom sub LLM that answers customer queries based on company-specific return policies and product information. This model uses static documents but doesn't change the master LLM.

Comparison Table: RAG vs. Custom GPT

Feature	RAG (Retrieval-Augmented Generation)	Custom GPT
Purpose	Retrieves real-time data to enhance the custom sub LLM's responses	Guides the custom sub LLM's responses based on predefined rules or documents
Customization Type	Integrates external data dynamically to generate contextually accurate responses for the custom sub LLM	Customizes the custom sub LLM's behavior using user-uploaded documents or instructions
Impact on the Model	Creates a custom sub LLM that temporarily integrates real-time external data without altering its parameters	Creates a custom sub LLM that follows rules or documents without modifying its parameters
Data Source	Pulls data from real-time external databases or documents	Uses static documents or instructions uploaded by the user
Dynamic/Static Data	Dynamic: Retrieves live data with each request	Static: Operates based on predefined rules and data
Customization Level	Integrates external, real-time data into the custom sub LLM responses	Customizes behavior without altering the custom sub LLM's core structure
Knowledge Update	Doesn't permanently update the custom sub LLM knowledge; dynamically retrieves fresh data	Doesn't update the custom sub LLM knowledge, just applies predefined instructions
Implementation	Requires external data sources (APIs, databases) to pull live data	Requires user-provided documents or instructions to guide responses
Use Case Example	Retrieving live weather or stock market data to enhance the custom sub LLM's response	Using a company's internal documents to guide customer service responses from the custom sub LLM
Scope	Broad and flexible: Can pull any relevant real-time data	Narrower: Follows specific instructions based on user-provided documents
Summary	Adds real-time data to enhance the custom sub LLM's responses, perfect for dynamic, up-to-date information needs.	Relies on predefined documents and rules, ideal for users who want specific behavior but don't need real-time updates

Comparison Table: RAG vs. Fine-Tuning

Feature	RAG (Retrieval-Augmented Generation)	Fine-Tuning
Purpose	Retrieves real-time external data to enhance the custom sub LLM responses without altering its core knowledge	Trains the custom sub LLM on specific datasets to specialize in certain tasks or domains
Customization Type	Dynamically pulls data in real-time to generate relevant answers for the custom sub LLM	Deeply customizes the custom sub LLM by training on specialized datasets
Impact on the Model	Creates a custom sub LLM that uses real-time data but doesn't change its underlying parameters	Creates a custom sub LLM that permanently updates its parameters based on the fine-tuned data
Data Source	Uses external, real-time databases or APIs	Uses static, curated datasets provided for training
Dynamic/Static Data	Dynamic: Pulls live, up-to-date data for every query	Static: Once fine-tuned, the custom sub LLM retains static knowledge based on training data
Customization Level	Adds external real-time data to enhance the custom sub LLM without altering core parameters	Adjusts the custom sub LLM for deeper, more specific tasks or domains by modifying its internal parameters
Knowledge Update	Doesn't update the custom sub LLM's core knowledge; integrates new data dynamically	Permanently updates the custom sub LLM's knowledge based on the fine-tuned dataset
Implementation	Requires external data sources (APIs, databases) to pull live data	Requires training datasets and API access for fine-tuning the custom sub LLM
Use Case Example	Pulling real-time legal updates to supplement the custom sub LLM's legal knowledge	Fine-tuning a custom sub LLM to specialize in medical research for specific diseases
Scope	Broad: Can pull any relevant real-time data	Narrower: Specializes the custom sub LLM in a specific domain or subdomain
Summary	Retrieves external, real-time data to dynamically adjust the custom sub LLM's responses without changing its core structure.	Provides permanent specialization of the custom sub LLM, training it on specific datasets for more accurate, niche task handling.

Comparison Table: Fine-Tuning vs. Custom GPT

Feature	Fine-Tuning	Custom GPT
Purpose	Trains the model (custom sub LLM) on specific datasets to specialize in certain tasks or domains	Guides the model (custom sub LLM) responses based on predefined rules or documents, without changing its core structure
Customization Type	Deep customization: Adjusts the custom sub LLM's parameters based on specialized data	Rule-based customization: Modifies behavior using user-defined instructions, without adjusting custom sub LLM parameters
Impact on the Model	Creates a custom sub LLM (child LLM) specialized in the task or domain, with permanent updates to parameters	Creates a custom sub LLM that follows specific rules or documents, but doesn't change the underlying custom sub LLM's parameters
Data Source	Uses specific, curated datasets provided for training	Works with static documents or instructions uploaded by the user
Knowledge Update	Permanently updates the custom sub LLM's specialized knowledge based on training data	Doesn't update the custom sub LLM's knowledge, just adjusts its response behavior
Scope	Narrower and more focused: Specializes the custom sub LLM in a specific domain or subdomain	Broader scope: Can be used for many purposes depending on the documents or rules defined by the user
Flexibility	Once fine-tuned, the custom sub LLM is locked into that specialized knowledge	More flexible: Users can easily change or update the rules or documents without retraining the custom sub LLM
Data Handling	Requires training data relevant to the domain	Uses user-uploaded documents or specific rules to guide responses
Implementation	Requires API access and training data to fine-tune the custom sub LLM	Can be implemented by uploading rules or documents without the need for retraining
Use Case Example	Training a custom sub LLM to specialize in medical research for specific diseases	Customizing the custom sub LLM to provide customer service responses based on a company's internal policies
Adaptability	Adapts well to complex, specialized tasks over time	Adapts quickly to changing rules or data, but less specialization in specific tasks
Time to Implement	More time-consuming: Requires training and testing the custom sub LLM on a specific dataset	Quicker to implement: Users can upload documents or set rules without needing to train the custom sub LLM
Summary	Fine-tuning creates a highly focused custom sub LLM that specializes in a domain or task by updating its parameters.	Custom GPT allows for quick, flexible customization through rules and documents, creating a custom sub LLM that adapts without changing its core knowledge

Master LLM vs. Sub LLMs (or Child LLMs):

- **RAG, fine-tuning, and custom GPT** do not update the **main LLM** (which we can refer to as the **master LLM**). Instead, they create **sub LLMs** (or **child LLMs**) that work in tandem with the master LLM to provide more tailored or domain-specific functionality.
- These **sub LLMs** are isolated, domain-specific versions of the master LLM that are customized for individual organizations or purposes, but they do not affect or change the core model (master LLM).
- By creating these **sub LLMs**, developers or organizations can leverage the general capabilities of the master LLM while adding their own specialized layers on top for specific use cases, without impacting the broader knowledge of the model.

Important Distinction:

- **Fine-tuning** and **RAG** should not be seen as methods to update the master LLM itself. Instead, they are ways to build specialized, custom versions (sub LLMs) of the master LLM.
- These **sub LLMs** work alongside the master LLM but are tailored to meet specific domain needs, whether through specialized training (fine-tuning) or real-time data access (RAG).
- The master LLM remains **untouched**, ensuring its broad applicability across many domains, while the sub LLMs cater to individual, niche requirements.

How Fine-Tuning and RAG Create Sub LLMs:

- **Fine-Tuning:** When fine-tuning is done, a **sub LLM** is created with specialized knowledge. This sub LLM is connected to the master LLM but only engages for domain-specific questions. It doesn't update the master LLM's parameters.
- **RAG:** When RAG is used, the model retrieves **external data** in real-time, but again, the master LLM's core knowledge is not updated. The retrieval process augments the response without permanently changing the model.

Why the Master LLM Needs Strong Domain Knowledge:

- For these sub LLMs to be effective, the **master LLM** needs to have a **solid foundational knowledge** of the domain or subdomain. If the master LLM lacks understanding in a particular area, the sub LLM might struggle, especially if it encounters entirely new terms or concepts that the master LLM can't relate to.
- The **master LLM** provides the essential framework, and without that foundational knowledge, the sub LLM will face limitations, even if it's fine-tuned or uses RAG for real-time data.

Summary:

- **Master LLM:** The core model that remains unchanged and provides a broad understanding of many domains.
- **Sub LLMs (Child LLMs):** Created through fine-tuning or RAG, they are specialized versions of the master LLM that cater to specific use cases but don't modify the core model.
- **Fine-Tuning and RAG** allow organizations or individuals to have their **own custom sub LLMs** without altering the master LLM.
- It's essential for the master LLM to have foundational domain knowledge, or else the sub LLMs may struggle with new or niche terms.

This approach ensures that organizations can build specific tools and models for their needs without compromising the integrity of the master LLM, while still leveraging its overall capabilities.

6. Comparison Between RAG, Fine-Tuning, and Custom GPT

This article provides a focused comparison between the three core methods for customizing **Large Language Models (LLMs)**: **RAG (Retrieval-Augmented Generation)**, **Fine-Tuning**, and **Custom GPT**. Each method offers distinct advantages depending on the specific needs of an organization, and they serve different purposes in shaping a custom LLM.

RAG vs. Custom GPT

Feature	RAG (Retrieval-Augmented Generation)	Custom GPT
Purpose	Retrieves real-time data to enhance the custom sub LLM's responses	Guides the custom sub LLM's responses based on predefined rules or documents
Customization Type	Integrates external data dynamically to generate contextually accurate responses for the custom sub LLM	Customizes the custom sub LLM's behavior using user-uploaded documents or instructions
Impact on the Model	Creates a custom sub LLM that temporarily integrates real-time external data without altering its parameters	Creates a custom sub LLM that follows rules or documents without modifying its parameters
Data Source	Pulls data from real-time external databases or documents	Uses static documents or instructions uploaded by the user
Dynamic/Static Data	Dynamic: Retrieves live data with each request	Static: Operates based on predefined rules and data
Customization Level	Integrates external, real-time data into the custom sub LLM responses	Customizes behavior without altering the custom sub LLM's core structure
Knowledge Update	Doesn't permanently update the custom sub LLM knowledge; dynamically retrieves fresh data	Doesn't update the custom sub LLM knowledge, just applies predefined instructions
Implementation	Requires external data sources (APIs, databases) to pull live data	Requires user-provided documents or instructions to guide responses
Use Case Example	Retrieving live weather or stock market data to enhance the custom sub LLM's response	Using a company's internal documents to guide customer service responses from the custom sub LLM
Scope	Broad and flexible: Can pull any relevant real-time data	Narrower: Follows specific instructions based on user-provided documents
Summary	Adds real-time data to enhance the custom sub LLM's responses, perfect for dynamic, up-to-date information needs.	Relies on predefined documents and rules, ideal for users who want specific behavior but don't need real-time updates

RAG vs. Fine-Tuning

Feature	RAG (Retrieval-Augmented Generation)	Fine-Tuning
Purpose	Retrieves real-time external data to enhance the custom sub LLM responses without altering its core knowledge	Trains the custom sub LLM on specific datasets to specialize in certain tasks or domains
Customization Type	Dynamically pulls data in real-time to generate relevant answers for the custom sub LLM	Deeply customizes the custom sub LLM by training on specialized datasets
Impact on the Model	Creates a custom sub LLM that uses real-time data but doesn't change its underlying parameters	Creates a custom sub LLM that permanently updates its parameters based on the fine-tuned data
Data Source	Uses external, real-time databases or APIs	Uses static, curated datasets provided for training
Dynamic/Static Data	Dynamic: Pulls live, up-to-date data for every query	Static: Once fine-tuned, the custom sub LLM retains static knowledge based on training data
Customization Level	Adds external real-time data to enhance the custom sub LLM without altering core parameters	Adjusts the custom sub LLM for deeper, more specific tasks or domains by modifying its internal parameters
Knowledge Update	Doesn't update the custom sub LLM's core knowledge; integrates new data dynamically	Permanently updates the custom sub LLM's knowledge based on the fine-tuned dataset
Implementation	Requires external data sources (APIs, databases) to pull live data	Requires training datasets and API access for fine-tuning the custom sub LLM
Use Case Example	Pulling real-time legal updates to supplement the custom sub LLM's legal knowledge	Fine-tuning a custom sub LLM to specialize in medical research for specific diseases
Scope	Broad: Can pull any relevant real-time data	Narrower: Specializes the custom sub LLM in a specific domain or subdomain
Summary	Retrieves external, real-time data to dynamically adjust the custom sub LLM's responses without changing its core structure.	Provides permanent specialization of the custom sub LLM, training it on specific datasets for more accurate, niche task handling.

Fine-Tuning vs. Custom GPT

Feature	Fine-Tuning	Custom GPT
Purpose	Trains the model (custom sub LLM) on specific datasets to specialize in certain tasks or domains	Guides the model (custom sub LLM) responses based on predefined rules or documents, without changing its core structure
Customization Type	Deep customization: Adjusts the custom sub LLM's parameters based on specialized data	Rule-based customization: Modifies behavior using user-defined instructions, without adjusting custom sub LLM parameters
Impact on the Model	Creates a custom sub LLM (child LLM) specialized in the task or domain, with permanent updates to parameters	Creates a custom sub LLM that follows specific rules or documents, but doesn't change the underlying custom sub LLM's parameters
Data Source	Uses specific, curated datasets provided for training	Works with static documents or instructions uploaded by the user
Knowledge Update	Permanently updates the custom sub LLM's specialized knowledge based on training data	Doesn't update the custom sub LLM's knowledge, just adjusts its response behavior
Scope	Narrower and more focused: Specializes the custom sub LLM in a specific domain or subdomain	Broader scope: Can be used for many purposes depending on the documents or rules defined by the user
Flexibility	Once fine-tuned, the custom sub LLM is locked into that specialized knowledge	More flexible: Users can easily change or update the rules or documents without retraining the custom sub LLM
Data Handling	Requires training data relevant to the domain	Uses user-uploaded documents or specific rules to guide responses
Implementation	Requires API access and training data to fine-tune the custom sub LLM	Can be implemented by uploading rules or documents without the need for retraining
Use Case Example	Training a custom sub LLM to specialize in medical research for specific diseases	Customizing the custom sub LLM to provide customer service responses based on a company's internal policies
Adaptability	Adapts well to complex, specialized tasks over time	Adapts quickly to changing rules or data, but less specialization in specific tasks
Time to Implement	More time-consuming: Requires training and testing the custom sub LLM on a specific dataset	Quicker to implement: Users can upload documents or set rules without needing to train the custom sub LLM
Summary	Fine-tuning creates a highly focused custom sub LLM that specializes in a domain or task by updating its parameters.	Custom GPT allows for quick, flexible customization through rules and documents, creating a custom sub LLM that adapts without changing its core knowledge

7. The LLM Inc

Welcome to LLM Inc, a company with one mission: to understand language and respond to any question or statement.

Every time a user inputs a prompt, LLM Inc. springs into action, and each department plays a key role in analyzing, understanding, and responding to that prompt. At LLM Inc., we work as a team of specialized departments, each with specific responsibilities to process language smoothly and efficiently.

The departments details

Each department brings a unique skill set to the table, and by working together, they ensure every user gets an accurate and well-structured response.

Department	Role
1. Tokenization Department	Breaks down the input text into manageable parts, or tokens.
2. Embedding Department	Converts each token into a numerical representation, called a vector, capturing its meaning.
3. Attention Department	Focuses on understanding relationships between words, identifying keywords, and recognizing context.
4. Processing Department	Refines connections and clarifies relationships within the prompt using insights from the Attention Department.
5. Decision Department	Generates a final response that directly answers or addresses the user's input based on all prior insights.

1. Tokenization Department

Department	Role	Algorithms Used
Tokenization Department	Breaks down the input text into manageable parts, or tokens.	Sub-word Tokenization Algorithms (e.g., Byte Pair Encoding (BPE), WordPiece)

Overview of the Tokenization Department's Role and Algorithm:

Role:

- The Tokenization Department's job is to take the user's input sentence and break it into smaller pieces called tokens.
- Tokens can be entire words, parts of words, or even punctuation marks, depending on how the sentence needs to be processed.

Algorithm Used:

- Sub-word Tokenization Algorithms like Byte Pair Encoding (BPE) or WordPiece are commonly used. These algorithms help handle complex language elements, such as rare words and multi-word terms, by breaking them down into more frequently occurring sub-words.
- For instance, in Byte Pair Encoding (BPE), the algorithm starts by tokenizing common words as whole units but breaks less common or complex words into smaller pieces.
- For example, “unbelievable” might be tokenized as “un,” “believ,” and “able,” ensuring the model can still understand it, even if “unbelievable” wasn’t directly encountered in training.

Example with Algorithm in Action:

Using the example sentence, “What’s the weather like today?”:

1. The Tokenization Team applies the BPE algorithm to break the sentence into tokens: “What’s,” “the,” “weather,” “like,” and “today.”

These tokens are now in a form that can be easily processed by the rest of the departments in LLM Inc

2. Embedding Department

Department	Role	Algorithms Used
Embedding Department	Converts each token into a numerical representation, called a vector , capturing its meaning.	Embedding Lookup Table and Matrix Multiplication

Overview of the Embedding Department’s Role and Algorithm:

Role:

- After the Tokenization Department has broken the input sentence into tokens, the Embedding Team’s job is to convert each of these tokens into a numerical vector (a series of numbers).
- This vector, or embedding, captures the basic meaning of the word and makes it readable for the model’s processing layers.

Algorithms Used:

- Embedding Lookup Table: The embedding process starts with a large “dictionary” or lookup table. This table contains a unique vector for each token in the vocabulary, learned from the model’s training data.
- Matrix Multiplication: If the embedding isn’t directly available in the lookup table, the model may use matrix multiplication with learned weights to adjust the vector, refining the meaning based on the token’s use in different contexts.

Example with Algorithm in Action:

Using the example sentence, “What’s the weather like today?”:

- 1. The Embedding Team assigns each token a vector:”1024 numbers mainly” “What’s” becomes [1.2, -0.5, 0.8, ...], “weather” becomes [0.4, 1.1, -0.3, ...], and so on.
- 2. These vectors allow the model to capture the basic meaning of each token, setting up a foundation for the Attention Department to start analyzing relationships.

3. Attention Department

Department	Role	Algorithms Used
Attention Department	Analyzes relationships between tokens to determine which parts of the sentence are most important in context.	Self-Attention Mechanism (also known as Scaled Dot-Product Attention), Multi-Head Attention

Overview of the Attention Department’s Role and Algorithm:

Role:

- The Attention Department focuses on identifying the relationships between words in a sentence to understand context.
- This department uses self-attention to allow each word (or token) to "pay attention" to other words in the sentence. In other words, it calculates which words are most relevant to each other, depending on the overall context.

Algorithms Used:

- **Self-Attention Mechanism:** This algorithm calculates attention scores for each pair of tokens. It does so by creating query (Q), key (K), and value (V) vectors for each token. The model uses these vectors to decide how much focus each word should receive.
- **Multi-Head Attention:** This algorithm runs the self-attention mechanism multiple times (in parallel), each time focusing on different aspects of the sentence. Multi-head attention helps capture various relationships and meanings simultaneously.

Example with Algorithm in Action:

Using the example sentence, “What’s the weather like today?”:

- 1. The Attention Department applies the self-attention mechanism:
 - Team A (Relation Experts): They find that “weather” is a key word in the sentence, with “like” and “today” directly relating to it.
 - Team B (Context Specialists): They understand that “What’s” indicates a question, and that “today” is important for setting the timing (i.e., current weather).
- 2. Each token is now represented with a stronger sense of context and relationship, based on how much attention each word received.

4. Processing Department

Department	Role	Algorithms Used
Processing Department	Refines and adjusts token meanings using the relationships identified by the Attention Department.	Feedforward Neural Network, Activation Functions (e.g., ReLU, GELU)

Overview of the Processing Department’s Role and Algorithm:

Role:

- The Processing Department takes the context-enriched tokens from the Attention Department and refines them.
- It fine-tunes each token’s meaning further, based on its adjusted context, ensuring that the relationships and meanings are as clear as possible.

Algorithms Used:

- **Feedforward Neural Network:** This is a small, simple neural network that processes each token’s vector independently to refine its meaning.
- **Activation Functions:** The Processing Department uses activation functions like ReLU (Rectified Linear Unit) or GELU (Gaussian Error Linear Unit) to add non-linearity, allowing the model to capture complex relationships in the data.

Example of How It Works:

Using the example sentence, “What’s the weather like today?”:

1. **Input from the Attention Department:** The word “weather” already has context (it’s related to “like” and “today”), thanks to the Attention Department.
2. **Feedforward Neural Network:** The Processing Department takes this context-rich vector for “weather” and further adjusts it to reinforce the meaning—clarifying, for example, that “like” refers to describing “weather,” not showing preference.
3. **Refinement for Clarity:** After processing, “weather” has an even sharper meaning in context, setting up the token to contribute clearly to the model’s final response.

5. Decision Department

Department	Role	Algorithms Used
Decision Department	Generates the final response based on all the refined meanings and relationships built up in previous departments.	Softmax Function, Probability Distribution

Overview of the Decision Department's Role and Algorithm:

Role:

- The Decision department is the one responsible for generating the final reply for the user.
- This team takes all the refined information processed by the previous departments (Tokenization, Embedding, Attention, and Processing) and uses it to make the best possible choice for each word in the response.
- This department works to create a coherent, accurate response that best answers the user's input.

Algorithms Used:

- **Softmax Function:** The Softmax function is used to calculate probabilities for each possible next token, based on the information refined by the previous departments.
- **Probability Distribution:** The model uses these probabilities to select the most likely next word, ensuring the response makes sense in context.

Here's how it works:

1. **Input from Other Departments:** The Decision Department gathers all the context, relationships, and refined meanings developed by the earlier departments.
2. **Selecting the Response:** Using probability calculations (via the softmax function), it selects the most likely next word or phrase based on the processed input.
3. **Final Reply:** The Decision Department composes the full response by choosing each word step-by-step, ensuring it is contextually accurate and coherent.

The Decision Department wraps up all the processing into a clear, final response, delivering it back to the user

Example of How It Works:

Using the example sentence, "What's the weather like today?":

1. **Gathering Insights:** The Decision Department receives all the refined information from the Attention and Processing Departments, so each word now has a clear meaning and context.
2. **Applying Softmax:** The model applies the Softmax function to create a list of possible next tokens (words) along with probabilities, based on the prompt and refined token vectors.
3. **Selecting the Response:** The Decision Department selects the highest-probability tokens to generate a response. For example, it might choose the response "The weather today is sunny and 25°C" by selecting tokens with the highest probability in this context.

Decision Department’s Data Sources and Live Data Connection

The Decision Department relies on two main sources of information to generate accurate responses: static training data and live data through Retrieval-Augmented Generation (RAG).

- **Fixed Knowledge vs. Real-Time Data:** The training data is static, meaning the model doesn’t inherently “know” real-time information like the current weather or recent news. To keep responses up-to-date, the Decision Department can access live data sources.
- **Accessing Live Data Sources:** Through RAG or predefined APIs, the Decision Department can access real-time sources like weather, news, or stock prices. When the model identifies a prompt that requires fresh data, it sends a query to these trusted sources.
- **Integrating the Response:** Once live data is retrieved (e.g., “The temperature is 25°C”), the Decision Department seamlessly integrates this information with its internally processed context, creating a coherent, contextually relevant response that includes the real-time update.

The Three Pillars of Data Storage in LLM Inc.

In addition to its specialized departments, LLM Inc. relies on three distinct types of storage to support its operations.

Each storage type serves a specific purpose, from the initial learning process to the real-time generation of responses.

Together, these storage systems enable LLM Inc. to retain knowledge, generate dynamic responses, and continually adapt to new prompts without needing the original training data. Here’s a quick overview of these storage types and their roles.

The three pillars

Storage Type	Description	Purpose
1. Source Training Data Storage	Contains the original data (like websites, books, articles) used to train the model.	Used temporarily during training to teach the model patterns, relationships, and meanings in language.
2. Model’s Learned Weights Storage	Stores the extracted knowledge as learned weights after training is complete.	Acts as the model’s “memory” of what it learned from training; these weights guide all future processing.
3. Layer-Specific Weights Storage	Holds weights specific to each layer (or department) in the model, guiding how tokens are processed.	Allows each department (like Attention or Processing) to dynamically generate Q, K, and V vectors, adapting to context in real-time.

Layer-Specific Weights Storage

Each department within LLM Inc. relies on the Layer-Specific Weights Storage to carry out its functions effectively.

These weights, tailored to each layer, store the knowledge gained during training and allow each department to apply that knowledge dynamically. From Tokenization to Decision-making, every department uses these weights to adjust their processes based on the specific input.

This enables the teams to generate Q, K, and V vectors, refine token meanings, and ultimately produce a contextually accurate response—all without needing the original training data. By tapping into this layer-specific storage, LLM Inc. ensures that each prompt is processed with adaptability and precision

Understanding Q, K, and V Vectors

In LLM Inc., each word (or token) is represented by three essential vectors: Query (Q), Key (K), and Value (V). Each of these vectors plays a distinct role in helping words connect meaningfully in a sentence.

- **Q (Query):** Represents what each word is looking for in other words. It acts as the “question” each token brings to the sentence, helping it identify potential connections with other words.
- **K (Key):** Represents what each word has to offer to others—its unique feature or identity. The K vector allows other words to recognize what this token can contribute to the sentence.
- **V (Value):** Contains the actual information each word provides if it becomes a focus. When a strong connection is made, the V vector supplies relevant details from one token to another.

Together, these vectors enable words to identify and share meaning, setting up the foundation for dynamic connection testing with the prompt.

Example 1

Vector Q of word “A” testing vectors K of multi words

Imagine word A is trying to understand its relationships with nearby words B, C, D, E, and F. Each of these words has its own Q (Query), K (Key), and V (Value) vectors, allowing them to interact meaningfully.

1. **Q Vector of Word A:** Word A begins by using its Q vector, which represents a “question” or focus, to seek out relevant connections. Its Q vector essentially asks, “Which of these words has something valuable to offer?”
2. **K Vectors of Words B, C, D, E, F:** Next, word A compares its Q vector with the K vectors of each nearby word (B, C, D, E, and F) to see if any have something particularly relevant to offer. This step tests what each word has to offer based on its unique identity or characteristics, as captured in their K vectors.
3. **Finding a Match with Word C:** During this comparison, word A discovers that word C’s K vector provides a strong match with what it’s looking for. This indicates that word C has a meaningful or useful connection for word A.
4. **Vectors V of Words A and C:** Once this connection is identified, word A uses its V vector to interact with the V vector of word C. This allows word A to draw specific information from word C, strengthening their relationship and enriching word A’s understanding based on this context.

In this way, word A's Q vector tests potential connections by evaluating the K vectors of nearby words. Once it finds a suitable match with word C, their V vectors connect, enabling the exchange of relevant information. This process is repeated for every word in the sentence, allowing dynamic and context-aware relationships to form based on the prompt and stored training knowledge.

Example 2

Connection Testing and prompt dual processes

1. Connection Testing process

When a prompt enters LLM Inc., each word actively begins to test connections with other words to determine meaningful relationships.

This process, known as connection testing, is guided by each token's Query (Q), Key (K), and Value (V) vectors. As each token's Q vector searches for relevant K vectors in other tokens, it draws on the stored weight data within each layer (or department).

These stored weights, learned from vast training data, act as a foundational knowledge base, helping the model recognize typical language patterns and relationships.

For instance, if the prompt involves a question about "weather," stored weights guide the model to relate "weather" with other timing words like "today" or descriptive words like "like."

2. Prompt process

However, the model doesn't rely solely on these stored patterns. The specific context of the prompt plays a vital role, dynamically shaping how each connection is adjusted.

When a Q-K match is found, the model fine-tunes the connection weight based on both the learned knowledge and the unique prompt context, creating a relationship that fits precisely.

Dual process

This dual process—drawing on stored weights and adapting to the prompt—enables LLM Inc. to interpret language flexibly and accurately, delivering responses that are both relevant and contextually aware.