# 1

# Introduction

**WHAT'S IN THIS CHAPTER?**

- Overview of GenAI Applications and Large Language Models
- Paths to Productionizing GenAI Applications
- The Importance of Cost Optimization

COPYRIGHTED MATERIAL

## OVERVIEW OF GenAI APPLICATIONS AND LARGE LANGUAGE MODELS

In this section, we introduce GenAI applications and large language models.

### The Rise of Large Language Models

Large language models (LLMs) have become a cornerstone of artificial intelligence (AI) research and applications, transforming the way we interact with technology and enabling breakthroughs in natural language processing (NLP). These models have evolved rapidly, with their origins dating back to the 1950s and 1960s, when researchers at IBM and Georgetown University developed a system to automatically translate a collection of phrases from Russian to English. The early pioneers were optimistic that human-level intelligence would soon be within reach. However, building thinking machines akin to the human mind proved more challenging than anticipated. In the initial decades, research in AI was focused on symbolic reasoning and logic-based systems. But these early AI systems were quite brittle and limited in their capabilities. They struggled with commonsense knowledge and making inferences in the real world.

By the 1980s, AI researchers realized that rule-based programming alone could not replicate the versatility and robustness of human intelligence. This led to the emergence of machine learning techniques, where algorithms are trained on large amounts of data to pick up statistical patterns. Instead of hard-coding complex rules, the key idea was to have systems automatically learn from experience and improve their performance. Machine learning enabled progress in specialized domains such as computer vision and speech recognition. But the overarching goal of achieving artificial general intelligence remained distant.

The limitations of earlier approaches led scientists to look at AI through a new lens. Rather than explicit programming, perhaps deep learning neural networks could be the answer. Neural networks are computing systems inspired by the biological neural networks in the human brain. They consist of layers of

interconnected nodes that transmit signals between input and output. By training on huge amounts of data, these multilayered networks could potentially learn representations and patterns too complex for humans to hard-code using rules.

> **NOTE** *Language is a complex and intricate system of human expressions governed by grammatical rules. It therefore poses a significant challenge to develop capable AI algorithms for comprehending and grasping a language. Language modeling is one of the major approaches to advancing machine language intelligence. In general, language modeling aims to model the generative likelihood of word sequences, so as to predict the probabilities of future (or missing) tokens. Language modeling research has received extensive attention in the literature, which can be divided into four major development stages: statistical language models (SLMs), neural language models (NLMs), pre-trained language models (PLMs), and large language models.*

In the 2010s, deep learning finally enabled a breakthrough in AI capabilities. With sufficient data and computing power, deep neural networks achieved remarkable accuracy in perception tasks such as image classification and speech recognition. However, these systems were narrow in scope, focused on pattern recognition in specific domains. Another challenge was that they required massive labeled datasets for supervised training. Obtaining such rich annotation at scale for complex cognitive tasks proved infeasible.

This is where self-supervised generative modeling opened new possibilities. By training massive neural network models to generate representations from unlabeled data itself, systems could learn powerful feature representations. Self-supervised learning could scale more easily by utilizing the abundant digital data available on the Internet and elsewhere. Language modeling emerged as a promising approach, where neural networks are trained to predict the next word in a sequence of text.

## Neural Networks, Transformers, and Beyond

Language modeling has been studied for decades using statistical methods like n-gram models. But neural network architectures were found to be much more effective, leading to the field of neural language modeling. Word vectors trained with language modeling formed useful representations that could be leveraged for various natural language processing tasks.

Around 2013, an unsupervised learning approach called **word2vec** became popular. It allowed efficiently training shallow neural networks to generate word embeddings from unlabeled text data. The word2vec embeddings were useful for downstream NLP tasks when used as input features. This demonstrated the power of pre-training word representations on large textual data.

The next major development was the proposal of **ELMo** by Allen Institute researchers in 2018. ELMo introduced deep contextualized word representations using pre-trained bidirectional long short-term memory (LSTM). The internal states of the bidirectional LSTM (BiLSTM) over a sentence were used as powerful context-based word embeddings. ELMo embeddings led to big performance gains in question answering and other language understanding tasks.

Later in 2018, Google AI proposed the revolutionary Bidirectional Encoders from Transformers (**BERT**) model. BERT is a novel self-attention neural architecture. BERT introduced a new pre-training approach called *masked language modeling* on unlabeled text. The pre-trained BERT model achieved huge performance gains across diverse NLP tasks by merely fine-tuning on task datasets.

The immense success of BERT established the "pre-train and fine-tune" paradigm in NLP. Many more transformer-based pre-trained language models were proposed after BERT, such as XLNet, RoBERTa, T5, etc. Scaling model size as well as unsupervised pre-training strategies yielded better transfer learning performance on downstream tasks.

However, model sizes were still limited to hundreds of millions of parameters in most cases. In 2020, OpenAI proposed **GPT-3**, which scaled up model parameters to an unprecedented 175 billion! GPT-3 demonstrated

zero-shot, few-shot learning capabilities never observed before, stunning the AI community. Without any gradient updates or fine-tuning, GPT-3 could perform NLP tasks from just task descriptions and a few examples. As such, GPT-3 highlighted the power of scale in language models. Its surprising effectiveness motivated intense research interest in training even larger models. This led to the exploration of LLMs with model parameters in the trillion+ range. Startups such as Anthropic and public efforts such as PaLM, Gopher, and LLaMA pushed model scale drastically with significant investments in the space. Several tech companies and startups are now using (and training their own) LLMs with hundreds of billions or even a trillion plus parameters. Models like PaLM, Flan, LaMDA, and LLaMA have demonstrated the scalability of language modeling objectives using the transformer architecture. At the time of this writing, Anthropic has developed Claude, the first LLM to be openly released with conversational abilities rivaling GPT-3.

You can see that all the models mentioned are related, much like the Tree of Life. In other words, anatomical similarities and differences in a phylogenetic tree are similar to the architectural similarities found in language models. For example, Figure 1.1 shows the evolutionary tree of LLMs and highlights some of the most popular models used in production so far. The models that belong to the same branch are more closely related, and the vertical position of each model on the timeline indicates when it was released. The transformer models are represented by colors other than gray: decoder-only models like **GPT**, **OPT** and their derivatives, encoder-only models like **BERT**, and the encoder-decoder models **T5** and **Switch** are shown in separate main branches. As mentioned earlier, models have successively "grown" larger. Interestingly, this is visually and objectively similar to the evolution of intelligent species, as shown in Figure 1.2. A deeper comparison is out of the scope of this book, but for more information on either of these evolutionary trees, refer to the links in the captions.

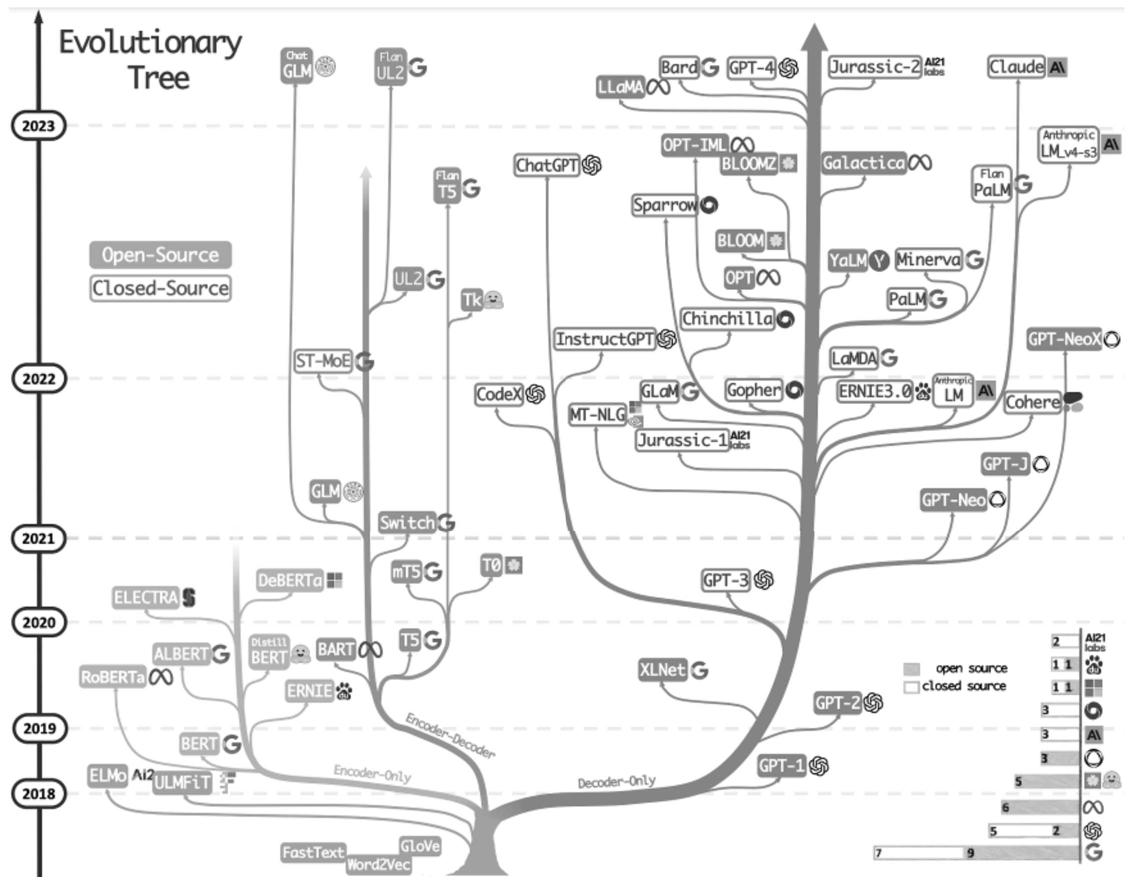


**FIGURE 1.1:** Evolutionary tree of language models
(see Rice University / `https://arxiv.org/pdf/2304.13712.pdf` / last accessed December 12, 2023.)

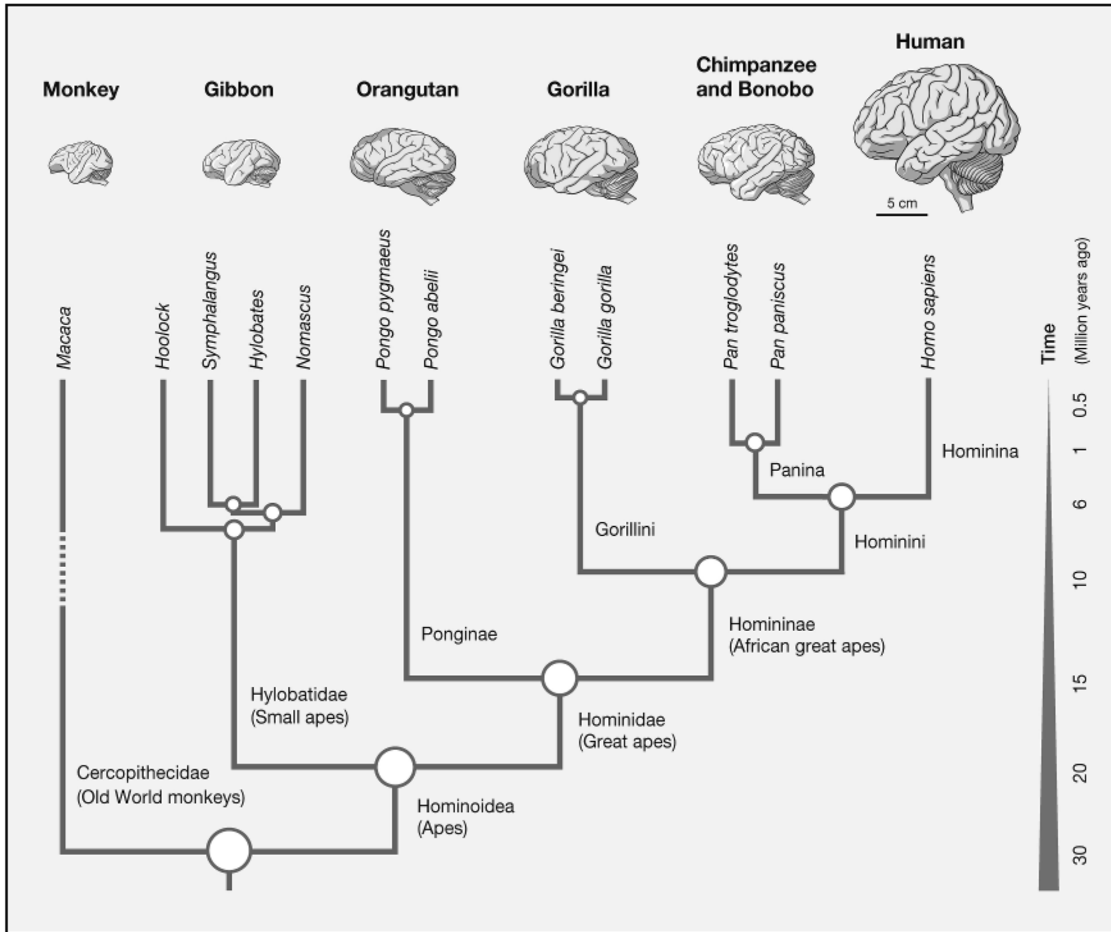


**FIGURE 1.2:** Evolutionary tree of human brain structure
(see André M.M. Sousa et al., 2017/ with permission of Elsevier.)

Increasing the model size, compute, and data seems to unlock new abilities in LLMs, which exhibit impressive performance on question answering, reasoning, and text generation with simple prompting techniques. By training LLMs to generate code, models such as AlphaCode and Codex display proficient coding skills. LLMs can chat, translate, summarize, and even write mathematical proofs aided by suitable prompting strategies.

The key shift from PLMs to LLMs is that scale seems to bring about qualitative transitions beyond just incremental improvements. LLMs display certain emergent capabilities such as few-shot learning, chain of reasoning, and instruction following not observed in smaller models. These abilities emerge suddenly once model scale crosses a sufficient threshold, defying smooth scaling trends.

LLMs entail a paradigm shift in AI from narrowly specialized systems to versatile, general-purpose models. Leading experts feel recent LLMs display signs of approaching human-level artificial general intelligence. From statistical to neural networks, the steady progress in language modeling scaled up by orders of magnitude has been the missing link enabling this rapid advancement toward more human-like flexible intelligence. The astounding capabilities of GPT-3 highlighted the power of scale in language models. This has led to intense research interest in developing even larger LLMs with model parameters in the trillion range. The assumption is that bigger is better when it comes to language AI. Scaling model size along with compute and data seems to unlock new abilities and performance improvements.

The largest LLMs have shown the ability to perform human-level question answering and reasoning in many domains without any fine-tuning. With proper prompting techniques like chain of thought, they can solve complex arithmetic, logical, and symbolic reasoning problems. LLMs can intelligently manipulate symbols, numbers, concepts, and perform multistep inferences when presented with the right examples.

But of course, language generation is the main area where LLMs' capabilities have taken a huge leap. LLMs can generate fluent, coherent, and human-like text spanning news articles, poetry, dialogue, code, mathematical proofs, and more. The creativity and versatility displayed in conditional and unconditioned text generation are remarkable. Few-shot prompting allows controlling attributes such as length, style, content, etc. Text-to-image generation has also made rapid progress leveraging LLMs. The exponential growth in model parameters has been matched by the computing power and datasets availability. Modern GPU clusters, the emergence of model parallelism techniques, and optimized software libraries have enabled training LLMs with trillions of parameters. Massive text corpora for pre-training are sourced from the Internet and digitization initiatives.

All this has fueled tremendous excitement and optimism about the future of AI. LLMs display a form of algorithmic and statistical intelligence to solve many problems automatically given the right data. Leading AI experts believe rapid recent progress is bringing us closer to artificial general intelligence than before. Large language models may be the missing piece that enables machines to learn concepts, infer chains of reasoning, and solve problems by formulating algorithms like humans.

LLMs still have major limitations. They are expensive and difficult to put into production, prone to hallucination, lack common sense, and struggle with complex symbolic reasoning. Model capabilities are also severely constrained by the training data distribution. LLMs can propagate harmful biases, generate toxic outputs, and be manipulated in dangerous ways. There are rising concerns around AI ethics, governance, and risks that merit careful consideration. Responsible development of AI aligned with human values is necessary. However, we already see several generative AI (GenAI) applications with these LLMs at their core! GenAI heralds a paradigm shift from narrow analytical intelligence toward creative and versatile systems. GenAI applications powered by models such as GPT-3, PaLM, and Claude are displaying remarkable abilities previously thought impossible for machines.

## GenAI vs. LLMs: What's the Difference?

While both GenAI and LLMs deal with generating content, their scopes and applications differ. GenAI is a broader term that encompasses AI systems capable of creating various types of content, such as text, images, videos, and other media. LLMs, on the other hand, are a specific class of deep learning models designed to process and understand natural language data. LLMs are used as a core component in GenAI applications to generate human-like text. GenAI applications, on the other hand, use LLMs to create more comprehensive and interactive experiences for users. While LLMs are responsible for understanding and generating human-like text, GenAI applications utilize these capabilities to create more comprehensive and interactive experiences for users.

GenAI applications are full end-to-end applications that could involve LLMs as their core. For example, ChatGPT is a GenAI application with GPT-3.5 and GPT-4 at its core. This means that while LLMs are responsible for understanding and generating human-like text, GenAI applications utilize these capabilities to create more comprehensive and interactive experiences for users. Putting LLMs into production as GenAI applications requires overcoming several challenges, including aligning LLMs with human values and preferences, training LLMs due to their huge model size, adapting LLMs for specific downstream tasks, and evaluating the abilities of LLMs. Despite these challenges, LLMs have the potential to revolutionize the way we develop and use AI algorithms, and they are poised to have a significant impact on the AI community, and society in general.

LLMs have enabled remarkable advances in GenAI applications in recent years. By learning from vast amounts of text data, LLMs like GPT-3 and PaLM can generate highly fluent and coherent language. This capability has been harnessed to power a diverse range of GenAI applications that were previously infeasible. Let's discuss some popular GenAI applications here:

**Conversational agents and chatbots**: One of the most popular applications of LLMs is conversational agents and chatbots. Systems like Anthropic's Claude and Google's LaMDA leverage the language generation skills of LLMs to conduct natural conversations. They can answer questions, offer advice, and discuss open-ended topics through

multiturn dialogue. The conversational abilities of these agents derive from the pre-training of LLMs on massive dialogue corpora. Fine-tuning further adapts them for smooth and consistent conversations.

**Code completion and programming assistants:** LLMs have proven adept at code generation and completion. Tools such as GitHub Copilot and TabNine auto-complete code based on natural language comments and existing context. This assists programmers by reducing boilerplate and speeding up development. LLMs can also generate entire code snippets or functions given high-level descriptions. Their training on large code corpora enables robust translation of natural language to programming languages. Beyond autocompletion, LLMs could serve as AI pair programmers that suggest improvements to code.

**Language translation machine translation:** This has benefited enormously from LLMs. Models such as Google's Translation LM achieve state-of-the-art results by learning representations from massive text corpora. They can translate between languages with higher accuracy and more contextual fidelity than previous phrase-based translation systems. LLMs retain knowledge about linguistics, grammar, and semantics that improves translation quality. Their training methodology also facilitates zero-shot translation between multiple languages.

**Text summarization and generation LLMs:** These are unmatched at summarizing lengthy text into concise overviews. They distill key points while preserving semantic consistency. Applications built using LLMs can summarize emails, articles, legal documents, and other sources as a text companion. Conditional generation allows summarization to be tailored for different lengths, styles, or perspectives. Text generation applications powered by LLMs can produce original long-form content such as stories, poems, and articles.

Even with this broad and deep set of capabilities and active research, companies around the world today may be concerned about the following aspects around productionizing GenAI-based applications:

- Access to high-performing LLMs is limited, costly, and not straightforward; commercial models are black boxes behind a paywall, and open-source models are mostly licensed for academic and research use, not for commercial applications.
- Building advanced applications that seamlessly integrate with existing databases and analytical applications is not trivial.
- There is no transparent architecture that ensures privacy and confidentiality of internal customer data while fine-tuning or using foundational models through an API. Questions around copyrights, trust, and safety are important and far from fully solved today.

In the next section, we begin to dive deeper into these concepts, starting with a framework to think of how a typical GenAI application is built today.

## The Three-Layer GenAI Application Stack

The rapid advancements in large language models like GPT-3, Claude, and PaLM have enabled new generative AI applications that can produce high-quality text, code, images, and more. However, developing and deploying these GenAI applications requires bringing together diverse components into an integrated technology stack. We will discuss the three main layers of a GenAI application stack—the infrastructure layer, the model layer, and the application layer—and delve into the details of each.

### The Infrastructure Layer

The infrastructure layer provides the foundational data, compute, and tooling resources needed to develop, train, and serve LLMs.

**Data storage and management:** LLMs require massive, labeled datasets, often petabytes in size, to train the models. Many public datasets like Common Crawl and Wikipedia provide broad coverage for pre-training foundation models. Custom datasets tailored to specific domains or applications are also curated for fine-tuning. This training data needs to be stored, managed, and accessed efficiently. Distributed object storage systems like Amazon S3, Azure Blob Storage, and Google Cloud Storage allow storing the vast datasets affordably. Data lakes built on cloud storage act as centralized repositories to gather, clean, and process heterogeneous data from diverse sources.

Metadata catalogs like AWS Glue, Azure Purview, and Google Data Catalog maintain schemas, trace data lineage, and enable discovery. Versioning capabilities track data changes over time. Data quality tools monitor and profile datasets. Overall, a robust data management platform is essential for LLMs to benefit from high-quality, well-organized training data.

**Vector databases**: LLMs rely on representing words and documents as dense numeric vectors that encode semantic meaning. Vector databases like Anthropic's Constitutional AI, Pinecone, Weaviate, and Milvus specialize in storing and indexing billions of vectors to enable efficient similarity search and retrieval.

They allow embedding large text corpora into vector spaces where semantic relationships are captured through distance metrics such as cosine similarity. This powers capabilities such as semantic search that go beyond keyword matching. Using vector databases decouples storing embeddings from model training and serving, enabling shared knowledge representation across applications.

**Compute infrastructure**: Training and running LLMs places intense computational demands, requiring access to extensive GPU/TPU resources. For example, training GPT-3 took 3,640 petaflop/s-days on more than 10,000 GPUs.

Cloud infrastructure providers such as AWS, Azure, and GCP offer GPU/TPU-optimized virtual machine instances. For example, AWS provides P4d instances with up to eight GPUs for machine learning workloads. GCP's Cloud TPU VMs give direct access to TPU chips tailored for ML.

Autoscaling groups dynamically match resources like GPUs to fluctuating training and inference demands. Orchestrators like Kubernetes facilitate deploying distributed LLMs at scale. The compute fabric should provide high-throughput, low-latency networking to support parallel model training across GPU clusters.

## The Model Layer

The model layer is the heart of a GenAI application stack, where the choice and adaptation of a large language model are crucial. This layer plays a pivotal role in determining the capabilities, efficiency, and effectiveness of the generative AI application.

**Choosing the right LLM:** When selecting an LLM as the foundation for your application, several key factors must be considered.

> **Model capability:** The first consideration is the model's inherent capability. Different LLMs may excel in various tasks. For instance, models like Google's BERT are renowned for their understanding of contextual information, while autoregressive models like OpenAI's GPT series are exceptional at generating coherent text.
>
> **Computational efficiency:** The computational resources required to train and deploy the chosen LLM are essential. Some models may be more resource-intensive than others, which can impact the scalability and cost of your application.
>
> **Commercial availability:** The availability of the model can be crucial. Many tech companies have open-sourced LLMs, allowing developers to use them freely. Alternatively, cloud providers offer LLMs via APIs, making them accessible but often with associated costs.
>
> **Problem domain suitability:** Consider the specific problem domain your application targets. Certain LLMs may be better suited for tasks such as text summarization (e.g., T5), while others shine in creative text generation (e.g., GPT).

**Fine-tuning the LLM:** Once you've selected an LLM as your foundation, fine-tuning becomes essential to adapt it to your application's unique requirements. Fine-tuning techniques, often based on transfer learning, are employed to achieve this adaptation. Take targeted datasets, for example.

> **Targeted datasets:** Fine-tuning typically involves training the LLM on smaller, domain-specific datasets. This helps the model become more proficient in specialized tasks while preserving its general intelligence.

**Enhancing capabilities:** By fine-tuning, you can improve the LLM's performance in specific areas, such as dialogue systems, reasoning, or knowledge retrieval. Anthropic's Claude, for instance, undergoes fine-tuning to enhance its dialogue capabilities without compromising its overall intelligence.

**Avoiding catastrophic forgetting:** Care must be taken during fine-tuning to avoid "catastrophic forgetting," where the model loses previously learned knowledge. Techniques such as gradient clipping and selective fine-tuning of specific layers are used to mitigate this risk.

**Integrating the LLM:** The integration of the selected and fine-tuned LLM into the application is a critical step.

**Data transformation:** Application code needs to convert input data, such as text or other types of data, into formats suitable for the LLM, typically token embeddings. This transformation is essential to ensure that the model can process the data effectively.

**Model architecture:** The architecture of the LLM plays a significant role in how it processes input data. Factors like how self-attention is applied across input tokens determine the model's ability to capture long-range dependencies in the data.

**Scaling innovations:** Recent advancements enable the efficient scaling of LLMs. Techniques such as sparsely gated mixture-of-experts partition model layers into multiple expert groups, enhancing scalability. This technique uses a trainable portion of the neural network to conditionally pass through some inputs, based on the actual contents of the input itself. For more information about this interesting advancement, refer to `https://arxiv.org/pdf/1701.06538.pdf`.

Additionally, efficient attention mechanisms like BigBird and Reformer reduce self-attention complexity for handling long sequences efficiently.

**NOTE** *The mixture of experts (MoE) layer consists of a number of neural network experts, each typically a simple feedforward network, along with a trainable gating network. The gating network selects a sparse combination of a small number of experts to process each input example. This allows the overall layer to contain a large number of parameters while activating only a small portion of them per example, drastically reducing computational costs. Specifically, noisy top-k gating adds Gaussian noise and then selects only the top k experts by gate value. The experts become specialized based on different input semantics and syntax. With up to thousands of experts and sparsity levels exceeding 99.99%, the MoE enables models with more than 100 billion parameters while maintaining reasonable computational efficiency. The convolutional application of MoE between recurrent neural network (RNN) layers enables different expert selections per timestep. Overall, the MoE layer enables massive increases in model capacity, leading to significantly improved results in language modeling and translation compared to prior state-of-the-art models. The extreme model parallelism allows continued benefits from increased model scale and data.*

## The Application Layer

The application layer focuses on streamlining GenAI application development and deployment leveraging integrated LLMs.

**Application development frameworks:** GenAI development frameworks are built around LLM APIs, such as Claude, Cohere, GPT-3, Genie, and LangChain, which simplify building applications using LLMs. They provide developer-friendly APIs and SDKs to access model inferencing without dealing with deployment details. These services encapsulate infrastructure provisioning, autoscaling, availability, networking, and other complexities. Developers just integrate the framework's APIs to invoke the LLM capabilities from application code. The frameworks scale seamlessly as request volumes grow rather than requiring changes to application logic.

Some frameworks also offer additional capabilities. For instance, Genie focuses on personalization, version management, and result caching to optimize LLM usage. Claude provides data versioning, monitoring, and safety tools in addition to streamlined model deployment.

**Building the application:** Developers incorporate LLM capabilities into the application using the framework's interfaces. The business logic sends user inputs to the framework API and processes the inferencing results. The application manages functionality such as identity, security, privacy, personalization, and monitoring on top of the LLM integration.

For conversational applications, the dialogue flow manages directing user utterances to the LLM API and rendering responses. Monitoring tools track metrics such as latency, errors, and resource consumption to maintain performance. User interactions are transformed into suitable LLM inputs, and outputs are post-processed if needed before returning responses.

The layered stack discussed in this section allows assembling the diverse components needed for impactful GenAI applications in a modular, scalable, and extensible architecture. As LLMs continue advancing at a rapid pace, the ability to iterate on and refine each layer independently will become increasingly critical. Companies that embrace and invest in optimized GenAI stacks will gain valuable advantages in deploying and harnessing LLMs to solve real-world problems.

## PATHS TO PRODUCTIONIZING GenAI APPLICATIONS

Despite fast-growing fundamental research and awareness around LLMs, GenAI applications are not widespread. They are large and costly to run and are difficult to put into production. However, we are seeing several startups and large enterprises building applications and interfaces to LLMs. Although a full architecture of GenAI applications is similar to any other enterprise architecture on the cloud, we would like to focus more on key components that make GenAI applications special. Throughout the book, we will discover more about LLMs, LLM APIs, and important components such as vector databases.

But first, let's dive deeper into the main ways used to develop these applications. The previous section presented some example GenAI applications. What do you see under the hood of the actual LLM or LLM API components of these architectures? How exactly do you use these APIs? Are there standard names for these methods used for interacting with LLMs? While there are several resources to answer these questions, let's dive right in and look at the key ways you can choose to interact with LLMs or LLM APIs.

**Zero-shot LLM prediction:** Pre-trained LLMs are used behind an API layer for prediction tasks; this includes simple classification tasks (such as sentiment analysis or topic modeling) and zero-shot chain of thought (CoT) reasoning. For example, an `Amazon.com` review such as "This is an excellent phone case, good value" along with candidate labels (positive, negative) can be passed into an LLM that can provide a prediction, even if it was not previously trained specifically for sentiment analysis tasks. Interestingly, the same model can be used to extract multiple dimensions or tags from raw data, which can augment and add value to the unstructured data owned by customers today.

**Few-shot in-context learning:** With a few examples as part of the input, LLMs can perform even better at multiple tasks. As an example, you can provide pairs of natural language statements and corresponding SQL queries to a model as context and provide a new natural language sentence for prediction and expect the right SQL query as an output. Here, the input distribution, output distribution, correctness, and input-output mapping have significant impact (more details explained below).

**Prompt engineering and prompt templates:** Developers of GenAI applications use prompts to interact with, improve the safety of, and augment LLMs with domain-specific knowledge. Prompt templates provide structure and a format to help LLMs return responses that are usable in downstream applications. Specifically designed prompts allow the LLM to provide better, structured results. Prompt templates are useful for zero-shot, few-shot, chain-of-thought, Reason+Act, and instruction synthesis frameworks.

For example, a zero-shot prompt template may look like this:

```
Answer the question based on the context below. Keep the answer short and concise.
Respond "Unsure about answer" if not sure about the answer.
Context: < ... >
Question: < ... >
Answer:
```

**Multiturn chat agents:** One-and-done style predictions are not suitable for complex tasks. When we need a more complex dive into a topic that is conversational, an LLM-based chatbot works very well. Examples of applications in production with this architecture include ChatGPT, BingChat, Bard from Google, OpenAssistant from LAION, Claude from Anthropic, Ernie Bot from Baidu, etc. The basic steps to create a chatbot style application (in the training phase) were outlined in the InstructGPT paper by OpenAI (`https://browse.arxiv.org/pdf/2203.02155.pdf`).

- Collect high-quality examples of instruction-fulfillment from human labelers and fine-tune an LLM.
- A new dataset is created based on multiple outputs from a single prompt, which is then used to train a supervised *reward model* based on ranks provided from a human labeler.
- Use reinforcement learning based on human feedback (RLHF) to produce a policy that uses the previous two items.

All these steps can be built and managed as modules using tools such as SageMaker Groundtruth and SageMaker's training and reinforcement learning capabilities. Existing open implementations of LLM-based chatbots also provide a monolithic Docker implementation to deploy the application as a full stack.

**Langchain:** Langchain is both a concept and a specific tool/framework to develop applications powered by LLMs. Langchain links several useful components, including LLMs to achieve an output that is more useful to end users. Langchain components include standard prompt templates, indexes, memory, connectors, and an SDK that provides access to not only these components for creating custom chains but also some tailored, use-case-specific chains that are useful out of the box. The simplest chain can include three steps: use a prompt template to parse the input, pass this processed input to an LLM, and parse and process the output before displaying. More complex chains can involve components that have access to tools (such as the Google Search API or a calculator or database access). Learn more about Langchain at `www.langchain.com`.

**LLM-based autonomous agents:** Langchain and related concepts allow users to construct a chain of steps that solves a use case. However, in some cases, it is impossible to predetermine the different combination of steps and components involved to complete the task posed by a user. Agents use a toolkit full of different tools to interact with the outside world; popular tools may include calculator, file system tools, the Python terminal, Google Search, WolframAlpha, Wikipedia API, etc. Autonomous agents such as AutoGPT, babyAGI, AgentGPT, etc., are agents that automatically combine components and reason steps and try to find solutions to your tasks. To achieve the final goal, the autonomous agent needs to add intermediate tasks and complete them in order to step closer toward the goal. Goals can be simple or very abstract and stated in natural language. As a result, autonomous agents will research the Internet, create tasks, execute these tasks, self-improve if needed, and create more tasks until the final goal is met. For example, try AgentGPT at `https://agentgpt.reworkd.ai`.

**Foundational model training/fine-tuning:** Foundational model training involves training a large model from scratch using a diverse dataset that makes use of significant compute resources. This produces a generalized model that can be adapted to multiple tasks. While few organizations have the resources to train an LLM from scratch and then deploy it for providing access at scale, most organizations will be able to fine-tune or use one of the previous methods to achieve success in their use cases. Fine-tuning involves taking a pre-trained foundational model and training it on a specific, smaller dataset. This takes less computational resources and time compared to foundational model training. The performance of a fine-tuned model can be significantly better than a large foundational model that is good at multiple tasks.

Now that you have an overview of the types of applications that can be built within the three-layer GenAI application stack, we will use a hypothetical use case of constructing a GenAI-based enterprise chat application by assembling components of the three-layer GenAI stack and techniques discussed.

## Sample LLM-Powered Chat Application

As mentioned, chat applications such as ChatGPT are examples of popular GenAI applications. As with any GenAI stack, the infrastructure layer provides the data, compute, and tooling to support large-scale LLMs. Vector databases enable embedding knowledge into vector spaces for efficient retrieval. Distributed GPU/TPU resources provide the computing muscle for LLM training and inference, and this is sometimes abstracted behind a managed API provided by LLM API providers so that end users do not need to worry about hosting and maintaining these models in production. The model layer focuses on choosing, fine-tuning, and optimally integrating the most appropriate LLMs into the application. Finally, the application layer leverages frameworks to simplify access to the deployed models. Figure 1.3 shows an example flow diagram with components that could be used to create a powerful LLM-based chat application.

Figure 1.3 illustrates the request flow in the LLM-powered chatbot application. The diagram provides a comprehensive overview of the request flow in the GenAI application, highlighting the interactions between the different components involved. The user sends a request to the chatbot, which then forwards the request to the application API. The API performs preprocessing on the request, followed by model inference and postprocessing. The processed response is returned to the API, which then sends it back to the chatbot. Finally, the chatbot delivers the response to the user.

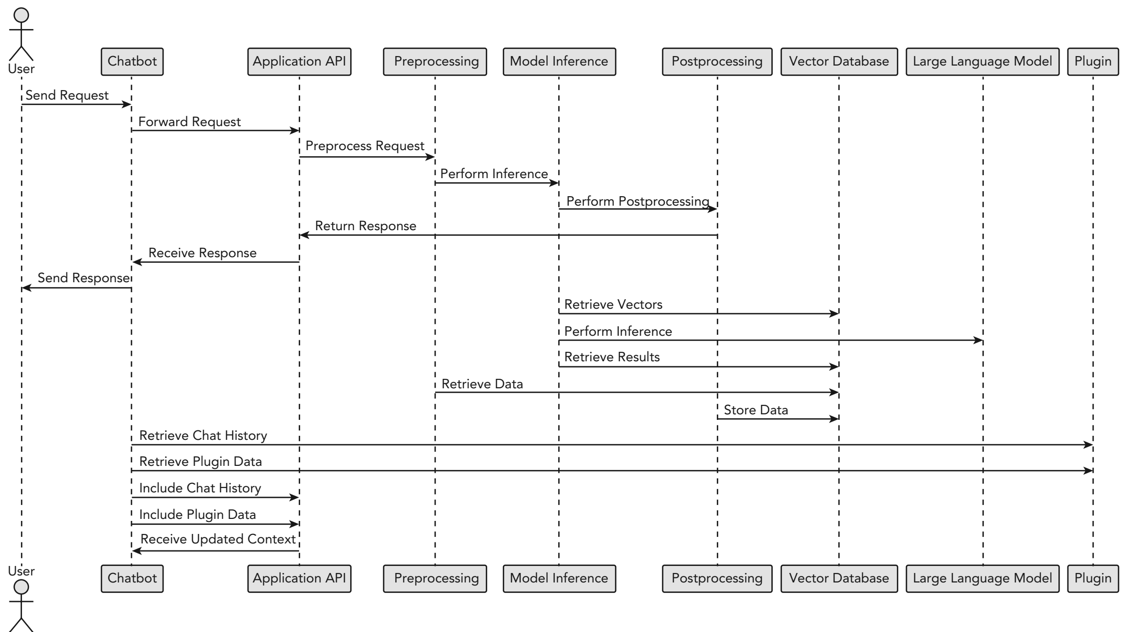


**FIGURE 1.3:** Sequence diagram of the request flow through a GenAI chatbot application

During the model inference stage, the inference component interacts with the vector database to retrieve vectors and with the LLM to perform the actual inference. Additionally, the preprocessing and postprocessing stages may interact with the vector database to retrieve and store data.

The chatbot component also interacts with the plugin component to retrieve chat history and plugin data. The chatbot includes this information in the request sent to the API, and the API incorporates it into the processing. The API then sends the updated context back to the chatbot.

Next, we will discuss how each of the components in the sample chatbot architecture needs to be tested for performance and how to generate and interpret pricing estimates for GenAI applications.

> **NOTE** *In this chapter, we leave out the fact that training a "chat-ready" LLM is itself a costly endeavor. One-and-done predictions without storing a session history are not suitable for complex tasks. When we need a more complex dive into a conversational topic, an LLM-based chatbot works well. Examples of applications in production with this architecture include ChatGPT, BingChat, Bard from Google, OpenAssistant from LAION, Claude from Anthropic, Ernie Bot from Baidu, etc. The basic steps to create a chatbot-style application (in the training phase) were outlined in the InstructGPT paper by OpenAI available at* `https://arxiv.org/pdf/2203.02155.pdf`.
>
> 1. *Collect high-quality examples of instruction fulfillment from human labelers and fine-tune an LLM.*
> 2. *A new dataset is created based on multiple outputs from a single prompt, which is then used to train a supervised reward model based on ranks provided from a human labeler.*
> 3. *Use reinforcement learning based on human feedback (RLHF) to produce a policy that uses the previous two steps.*
>
> *All these steps can be built and managed as modules using tools such as SageMaker Groundtruth and SageMaker's training and reinforcement learning capabilities on the cloud, which will incur costs. Existing open implementations of LLM-based chatbots also sometimes provide a monolithic Docker implementation to deploy the application as a full stack. But the responsibility (and cost) of hosting these container applications are with the end user (or chatbot service provider).*

# THE IMPORTANCE OF COST OPTIMIZATION

Let's revisit the previous flow diagram of the chatbot (GenAI) application with a focus on a few costly components that are highlighted with a darker border in Figure 1.4.

We see three components highlighted.

- Model inference
- Vector database
- Large language model

Let's first dive deeper into each of these components to figure out why they may turn out to be expensive in production.

## Cost Assessment of the Model Inference Component

The model inference component is a core component at the center of all GenAI chat applications. It is a component that does the following things:

1. Receives a request to perform inference on.
2. Analyzes the request and allows/disallows the inference to continue. This can be for parameter validation, for safety, and to prevent malicious use.

3. Retrieves vectors from the vector database, or in cases where the vector database provides a similarity search API, directly receives similar items from the database.
4. Constructs the prompt for prediction with the LLM.
5. Receives response from the LLM.
6. Performs any post-processing of the response received from the LLM.
7. Returns the response to the client application if successful and returns an appropriate error code otherwise.

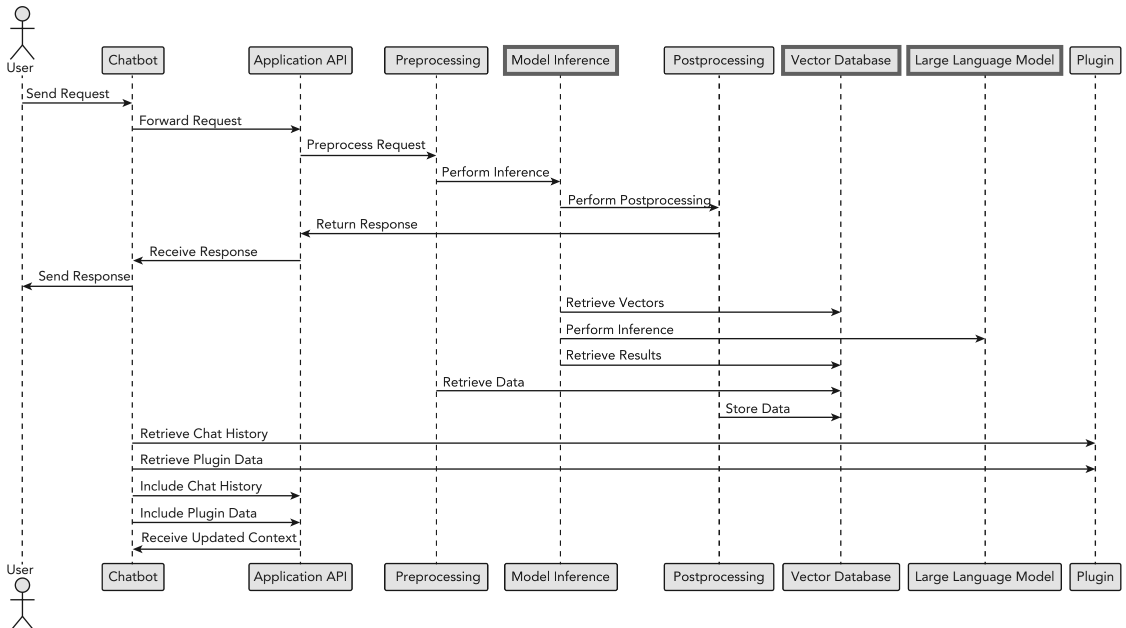


**FIGURE 1.4:** Sequence diagram of the request flow through a GenAI chatbot application, with a focus on three costly components

> **NOTE** *In many cases, the model inference component may include preprocessing and post-processing. In some cases, the model inference container also hosts the entire LLM. In practice, several LLM-as-an-API providers, such as OpenAI, Anthropic, and Amazon Web Services (AWS) offer competitive pricing and extremely easy-to-use APIs that obviate the need to build and host an LLM on your own in the model inference component.*

Why could this component be costly? Depending on how you choose to create this component, there could be costs associated with hosting your model inference code. In AWS, for example, there are services that can help host this component. For example, you can use one of the following services, in increasing order of complexity:

- AWS Lambda, which lets you run your code without having to manage servers (see `https://aws.amazon.com/pm/lambda`)

- AWS Fargate, which is a serverless compute engine to orchestrate container-based applications (see `https://aws.amazon.com/fargate`)
- Amazon SageMaker, which is an end-to-end managed service for machine learning including hosting LLMs (`https://aws.amazon.com/sagemaker`)
- Amazon EC2, which is a secure and resizable compute for any workload including hosting LLMs (`https://aws.amazon.com/ec2`).

Similar serverless and server based options exist on other cloud providers such as Microsoft Azure or Google Cloud Platform (GCP).

Now let's try to price this component based on the available options today. The following is a pricing exercise that you can modify for your own use case.

As mentioned earlier, it is anticipated to have about 100 concurrent users chatting with the assistant at any given time, and each user will make about 100 requests per hour through natural conversation with the assistant. For a given user, this means the user takes 60/100 ~= 0.6 minutes, or 36 seconds, on average per request; this is realistic since the user needs some time to type the next question in the chat session.

We then need to benchmark the model itself, since the architecture of the model inference component changes based on how fast the hosted LLM or LLM API can make inferences. The metric to calculate for this purpose is tokens generated per second, or simply tokens per second (TPS). If you aren't already familiar with tokens, they are commonly found sequences or groups of characters that can be used to form words. Often entire words are tokens. For example, for an input prompt that looks like this:

```
As a customer support executive, you are assisting a customer who claims that their
previous bill is still showing as due despite making the payment last week.
Write a reply by reviewing the cusstomer's bill payment history and resolving
the issue.
```

the tokens correspond to Figure 1.5 (generated using `https://platform.openai.com/tokenizer`). Figure 1.6 shows the token IDs of the corresponding tokens in the previous sentence, which are index pointers to these specific tokens in the sentence found in the tokenizer's vocabulary.

> **NOTE** *If you want a programmatic interface to the tokenizer used by ChatGPT and GPT models, check out* `https://github.com/openai/tiktoken`.

To benchmark TPS for a hosted LLM or LLM API, first choose a benchmark prompt or set of prompts. It is important to control the number of tokens generated, either using an appropriate max token count parameter in an API call or using a smart prompt. Let's use a long sentence from *Don Quixote*, written in 1605, and considered a classic, foundational literary work in Western literature. Here is the sentence:

About this time, when some rain began to fall, Sancho proposed that they should shelter themselves in the fulling-mill, but Don Quixote had conceived such abhorrence for it, on account of what was past, that he would no means set foot within its wall; wherefore, turning to the right-hand, they chanced to fall in with a road different from that in which they had traveled the day before; they had not gone far, when the knight discovered a man riding with something on his head, that glittered like polished gold, and scarce had he descried this phenomenon, when turning to Sancho, 'I find,' said he, 'that every proverb is strictly true; indeed, all of them are apophthegms dictated by experience herself; more especially, that which says, 'shut one door, and another will soon open': this I mention, because, if last night, fortune shut against us the door we fought to enter, by deceiving us with the fulling-hammers; today another stands wide open, in proffering to use us, another greater and more certain adventure, by which, if I fail to enter, it shall be my own fault, and not imputed to my ignorance of fulling-mills, or the darkness of the night.

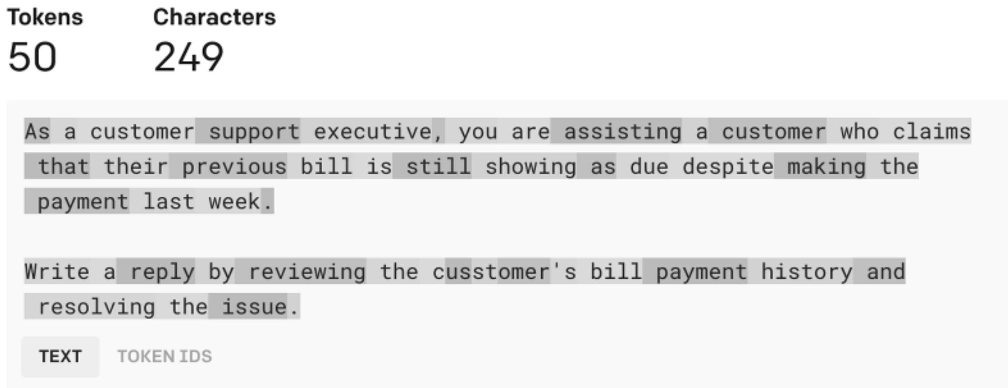


**FIGURE 1.5:** Tokens generated for the text prompt. Notice how the misspelled word "cusstomer" and its apostrophe get split into their own different tokens.

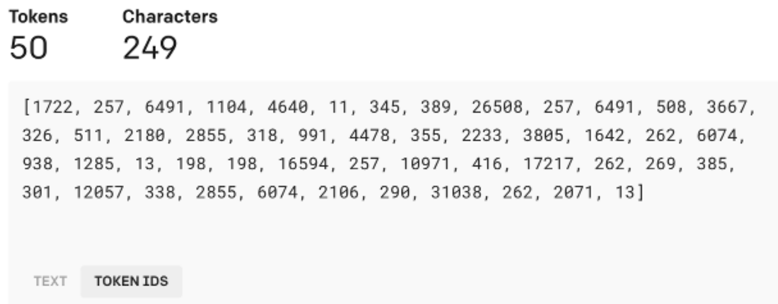


**FIGURE 1.6:** Token IDs of the tokens generated from the original text prompt and shown in Figure 1.5

This long, single sentence paragraph from *Don Quixote* has 1,158 characters and 269 tokens, as shown Figure 1.7. It is not necessary to use a long sentence, and when repeating the benchmark, you can use a set of examples relevant to your business.

Through the API, or from the console, we can calculate the total time for repeating this sentence. Add an instruction to the previous like "repeat the following sentence exactly once." Since most models faithfully repeat the fixed, known number of tokens, asking the same question to multiple models may result in varying answers with a different number of tokens. Also, we can modify the instruction by saying "repeat the following sentence exactly twice," or thrice, four times, etc. The process that generates these tokens is not linear, so testing the number of tokens across these cases is important. It is also a good idea to repeat each experiment at least five times and report the best and worst TPS for each experiment.

Table 1.1 is a sample result table using the previous sentence with GPT 3.5. Note that the model prediction is timed using the free research preview of the ChatGPT application and is not representative of a real benchmark; it is shown here to explain the ways to read results, even if the actual experiment is done from the model inference component being designed, and via an API call.

Tokens 269 Characters 1158

About this time, when some rain began to fall, Sancho proposed that they should shelter themselves in the fulling-mill, but Don Quixote had conceived such abhorrence for it, on account of what was past, that he would no means set foot within its wall; wherefore, turning to the right-hand, they chanced to fall in with a road different from that in which they had traveled the day before; they had not gone far, when the knight discovered a man riding with something on his head, that glittered like polished gold, and scarce had he descried this phenomenon, when turning to Sancho, 'I find,' said he, 'that every proverb is strictly true; indeed, all of them are apophthegms dictated by experience herself; more especially, that which says, 'shut one door, and another will soon open': this I mention, because, if last night, fortune shut against us the door we fought to enter, by deceiving us with the fulling-hammers; today another stands wide open, in proffering to use us, another greater and more certain adventure, by which, if I fail to enter, it shall be my own fault, and not imputed to my ignorance of fulling-mills, or the darkness of the night.

**FIGURE 1.7:** Tokens generated for the long example sentence

**TABLE 1.1:** GPT 3.5 TPS benchmark results

| EXPERIMENT # | MODEL | PREFIX INSTRUCTIONS | BEST/WORST TPS |
|---|---|---|---|
| 1 | GPT 3.5 | Repeat this sentence exactly once | 45/13 |
| 2 | GPT 3.5 | Repeat this sentence exactly twice | 48/26 |
| 3 | GPT 3.5 | Repeat this sentence exactly thrice | 50/10 |
| 4 | GPT 3.5 | Repeat this sentence exactly four times | 46/40 |

Here are a few observations about Table 1.1:

- The best-case scenarios across experiments seem to be consistent, at around 48 TPS.
- The worst-case numbers for TPS have a huge range, from as low as 10 TPS to 40 TPS.
- Across all experiments conducted, the average TPS was around 35 TPS. Most teams will design their application for the expected demand based on the average TPS.
- More experiments with other models may inform your choices, especially if you are left with little control over the hosting infrastructure and you are mainly interfacing with managed APIs. While managed LLM APIs provide limits and strive to provide consistently high performance, the onus of performing benchmarks still lies with the application building team.

Now that we have a number for the average TPS (35), we also need to make some assumptions on the average token size per text prompt. In the previous test, about 250 tokens were returned as a response, and this is a good, representative response size. The response per turn will then take 250/35, which is around 7 seconds. The worst-case response time is 250/10, or 25 seconds. The worst-case response is better than the requirement of about 36 seconds per response. This will work for a single user, with requests being queued up and sent back to back, sequentially. We also have an orthogonal requirement of 100 concurrent users doing 100 requests each per hour. Additionally, LLM API providers will enforce rate limits on the requests you make over three dimensions: TPS, requests per second (RPS), and concurrency. Requests are throttled if any of these limits are exceeded.

> **NOTE** *Rate limits in LLM APIs play a crucial role by setting rules on how frequently users can request information within a specific timeframe. These limits serve multiple purposes. First, they safeguard the API from misuse or abuse, preventing malicious actors from overloading the system. Second, rate limits ensure equitable access for all users, preventing any single user from monopolizing resources and slowing down the API for others. Last, they help maintain the API's performance and prevent server strain during periods of high usage, ensuring a smooth experience for all users using the API. This is especially important with LLM APIs, since all users hit the same logical base model, but behind the scenes, the LLM API providers have to massively scale out their inference workload to support tens of thousands of concurrent users.*

Based on the benchmark, only about 10 users will receive their response within the minute with one request going out per second (1 RPS); with 10 RPS, we should be able to service 100 sessions. However, this may not be completely true.

Assume that we have the following limits from the LLM API provider: a TPS limit of 600, an RPS of 10, and a concurrency of 1. These look like very stringent limits for our discussion but are realistic since they are around the same limits offered today by leading LLM API providers. From our previous benchmark, we expect the model to produce tokens at about 35 TPS when tested from a client application. Since we expect 100 users to do 100 requests per hour, the total throughput of RPS required is 100*100/(60*60), which is 2.7 RPS on average. These two to three requests may not consistently happen at every second; we expect this to be the average RPS over a long period of time. However, from our benchmark, we see that we can only do about 10 requests per minute, which is less than 1 RPS. With 100 user sessions running and a RPS limit of 10 from the LLM API provider, combined with the concurrency limit of 1, the model inference application may be quickly throttled, even though the API calls are below the TPS limit.

Practically, the model inference component can implement its own queue, use an external service as a queue, or, even simpler, use exponential backoff implemented by a library. In Python, it would look like this:

```
import backoff
import LLMapi

@backoff.on_exception(backoff.expo, LLMapi.ThrottleError)
def predict_with_backoff(**kwargs):
    return LLMapi.predict(**kwargs)

predict_with_backoff(prompt="...")
```

Note that in the previous example, since we are still not hitting the TPS limit, we can batch multiple requests from the model inference component. Typically, the LLM API backend will send back a batched response much quicker than sequentially running it. This API call may look like the following:

```
predict_with_backoff(prompt=["...","...","...","...","...","..."] )
```

This effectively multiplies the achieved RPS to help service more users in parallel, once again subject to the multiple limits mentioned earlier. The model inference component then has to stay alive for the time it takes to complete that particular request, batched or otherwise. This tells us which of the various options (serverless, managed server) to use. For a complex model inference server that is serving multiple users, it may be worthwhile considering an always-on system such as a long-running container or a VM. Serverless options usually will suffer from a cold start but can be cost effective. A one-second cold start means your inference does not actually begin for 1 second. Container-based serverless solutions can be slower to come up. Nevertheless, there are great use cases for model inference components to be on services like AWS Lambda. Let's look at how to calculate price when using AWS Lambda (adapted from `https://aws.amazon.com/lambda/pricing` for our example).

In our scenario with 100 concurrent users making 100 requests per hour, on the average case, we receive 35 TPS from the LLM API.

```
Monthly compute charges:
Total compute duration per request (seconds) = 269 tokens / 35 tokens/second =
7.6857 seconds (rounded to 7.69 seconds)
Total compute duration per user per hour (seconds) = 100 requests/hour * 7.69
seconds/request = 769 seconds
Total compute duration per user per month (seconds) = 769 seconds * 30 days/month =
23,070 seconds
Total compute duration for all users per month (seconds) = 100 users * 23,070
seconds = 2,307,000 seconds
Total compute (GB-s) = 2,307,000 seconds * 4096 MB / 1024 MB = 9,228,000 GB-s
Total compute – AWS Free Tier compute = Monthly billable compute GB-s
9,228,000 GB-s - 400,000 free tier GB-s = 8,828,000 GB-s
Monthly compute charges = 8,828,000 * $0.0000166667 = $147.13
Monthly request charges:
Total requests per user per hour = 100 requests/hour
Total requests per user per month = 100 requests/hour * 24 hours/day * 30 days/
month = 72,000 requests
Total requests for all users per month = 100 users * 72,000 requests = 7,200,000
requests
Total request charges = 7,200,000 / 1,000,000 * $0.20 = $1.44
Total monthly charges:
Total charges = Compute charges + Request charges = $147.13 + $1.44 =
$148.57 per month
```

In this updated scenario with 100 concurrent users making 100 requests per hour, where each request involves generating about 269 tokens, and on average, you get 35 tokens per second generated from the back end, your total monthly charges would be approximately $148.57.

In our end-to-end application, however, the cold start of having these AWS Lambda functions is non-negligible. With complex model inference components that do pre- and post-processing of requests, the additional startup latency could be significant. Additionally, without provisioned capacity on AWS Lambda, the time it takes to serve requests increases as the number of requests increases. For example, let us start off with our case where each request takes around 7.69 seconds. We can perform a benchmark using a load testing tool like ab (`https://httpd.apache.org/docs/2.4/programs/ab.html`) or locust (`https://locust.io`) , where we first have no provisioned capacity set up and send 100s of requests to this endpoint. Now, since we don't want to pay the LLM provider for these benchmarks, it is a good idea to simply emulate these requests and sleep for 7.69 seconds on average for every request coming in from these benchmarking tools. With provisioned concurrency disabled, we get the results shown in Table 1.2 for completed predictions.

**TABLE 1.2:** Percentage of requests served within a certain time period in seconds

| PERCENTAGE OF REQUESTS | MAX SECONDS ALLOWED |
|---|---|
| 50% | 7.69 s |
| 66% | 7.85 s |
| 75% | 8.35 s |
| 80% | 8.62 s |
| 90% | 9.46 s |
| 95% | 29.85 s |
| 98% | 35.61 s |
| 99% | 36.69 s |
| 100% | 42.29 s |

As we can see from Table 1.2, with hundreds of requests coming in and with no provisioned capacity, it could take several times the originally assumed 7.69 seconds to serve all requests. Provisioned capacity on AWS Lambda is a feature where your serverless functions can be kept warm or initialized, waiting for requests so that there is minimal startup latency. With this, functions can be made ready to respond within double-digit milliseconds. However, going back to the theme of cost to performance, this obviously means it will cost more. But how much more? Let's perform the same costing exercise with provisioned capacity. Let's assume your functions receive requests from 100 users 100 times per hour. Each request takes about 7.69 seconds. Provisioned concurrency is enabled throughout the month, 16 hours a day. You configure your function with 4,096 MB of memory on an x86-based processor and set Provisioned Concurrency at 100.

```
Monthly Provisioned Concurrency charges:
The Provisioned Concurrency price is $0.0000041667 per GB-s.
Total period of time for which Provisioned Concurrency is enabled (seconds) = 30
days * 16 hours * 3,600 seconds = 1,728,000 seconds.
Total concurrency configured (GB): 100 * 4096 MB / 1024 MB = 400 GB.
Total Provisioned Concurrency amount (GB-s) = 400 GB * 1,728,000 seconds =
691,200,000 GB-s.
Provisioned Concurrency charges = 691,200,000 GB-s * $0.0000041667 = $2,880.00.
Monthly Compute charges while Provisioned Concurrency is enabled:
The compute price is $0.0000097222 per GB-s.
Total compute duration (seconds) = 100 users * 100 requests * 7.69 seconds per
request = 76,900 seconds per hour.
Total compute duration for the month = 76,900 seconds per hour * 16 hours per day *
30 days = 36,993,600 seconds per month.
Total compute (GB-s) = 36,993,600 seconds * 4096 MB / 1024 MB = 147,974,400 GB-s.
Total compute charges = 147,974,400 GB-s * $0.0000097222 = $1,440.88.
Monthly request charges:
The monthly request price is $0.20 per 1 million requests.
Monthly request charges = (100 users * 100 requests per hour * 16 hours per day *
30 days) / 1,000,000 * $0.20 = $768.00.
Total monthly charges:
Total charges = Provisioned Concurrency charges + Compute charges while Provisioned
Concurrency is enabled + Request charges = $2,880.00 + $1,440.88 + $768.00 =
$4,088.88.
```

We see a huge increase in cost, from close to $150 to now $4,000 a month. I hope you now see why evaluating the cost of each component in your architecture, getting representative benchmark numbers, and balancing cost to performance are important. Let's continue this discussion with another important component of the architecture, vector databases.

## Cost Assessment of the Vector Database Component

Vector databases have emerged as a critical technology for solving complex problems in areas such as recommendation systems, natural language processing, computer vision, and now retrieval augmented generation (RAG) applications with LLMs. The ability of vector databases to perform ultra-fast similarity search on large datasets of high-dimensional vectors enables transformative applications. However, with a plethora of vector database solutions available, from open-source systems to cloud services, selecting the right one for your use case can be challenging. This highlights the importance of benchmarking to evaluate different options on crucial performance factors, which includes cost. As you will notice throughout the book, cost-based benchmarking is not useful without considering performance.

This section covers the key components to test, benchmark design considerations, test cases to include, metrics to track, guidelines for fair evaluation, and best practices when choosing a vector database for your application. The goal is to equip you with the knowledge to thoroughly assess vector databases and select the optimal one for your needs. Benchmarking the performance of a vector database is critical to understanding how it will operate under real-world production workloads.

Several key components determine the performance of a vector database. A rigorous benchmark must evaluate the following aspects thoroughly:

**Index building time:** This measures the time taken to build search indexes on a vector dataset, which enables fast retrieval. Index building involves steps such as data loading, normalization, graph structure creation, etc. Not all indexes are built using all these components; most indexes are just vector databases with additional metadata. Faster index building allows quick start of production workloads.

**Insertion throughput:** Insertion throughput indicates how fast the database can ingest new vectors. This matters both for initial bulk loading and for incremental inserts during production. Key metrics to track here are the number of vectors inserted per second and the total time for bulk loading.

**Search latency:** Search latency measures the time taken to execute search queries after indexes are built. Lower latency allows real-time response for time-sensitive applications. Testing latency under different workloads reveals responsiveness. The key metrics to track are average, median, $95^{th}$ and $99^{th}$ percentile latencies.

**Search throughput:** While latency focuses on a single query, throughput measures how many search queries can be processed per second concurrently. Higher throughput supports heavy production workloads with many users. The key metric to track here is queries per second (QPS).

**Scalability:** Testing these metrics on varied data and workload sizes reveals how well performance scales. Benchmarking should cover small to huge datasets and moderate to high concurrency.

**Accuracy:** Search accuracy depends on the matching logic. Benchmarking must test relevant accuracy metrics such as precision, recall, etc., under different conditions. Key metrics are recall and F1 score for search results.

**Resource efficiency and cost:** Given similar hardware, a database that achieves better performance per CPU core, RAM utilization, etc., is more resource efficient. When the compute chosen is more efficient, operational costs are lowered. This also includes the benchmarking of filtering search results by predicates without scanning all vectors, which is crucial for efficiency in production. Benchmarking must test filtering throughput and latency for diverse filtering ratios. Key metrics to track are end-to-end query latencies, cost, recovery time after failures, and carbon footprint. Finally, looking at the end-to-end cost to performance ratio is important, including factors such as hardware, software, operating, and maintenance expenses. This helps in determining the overall value of the database for specific use cases.

As you can see from the various dimensions of benchmarks to test, results could be very case specific. Several benchmarking tools are available for evaluating the performance of vector databases. Some of these tools include the following:

**VectorDBBench:** An open-source vector database benchmark tool that provides unbiased vector database benchmark results for mainstream vector databases and cloud services; see `https://zilliz.com/vector-database-benchmark-tool`

**Qdrant:** An open-source vector search engine that provides a comparative benchmark and benchmarking framework for vector search engines and vector databases; see `https://qdrant.tech/benchmarks`

**ANN Benchmark:** A popular benchmark that evaluates both scientific libraries and vector databases, providing a starting point for performance comparison; see `https://ann-benchmarks.com`

## Benchmarking Setup and Results

Now, similar to the previous section on creating your own benchmarking table for the model inference component, we can create our own benchmarking numbers for the vector database component. It is likely that this benchmarking exercise is a large undertaking for a team of any size, so the only caution is to use as many open benchmarks as possible before deciding and set up your own benchmarking architecture only if required.

Let's assume you have a need to run your own benchmarking; cloud providers offer several services that can be stitched up to do this. There is no "standard" architecture and no widely accepted set of tools either. The following is a discussion of one of the various ways to set this up.

First, we discuss some principles of setting up a benchmark for the vector database component.

**Benchmark vector embedding components in isolation:** Since we want to test metrics associated with ingest and latency of queries for just this component, we need to isolate other components in our architecture and eliminate latency from these other sources. For example, one of the most important aspects of building a vector database is embedding vectors. LLM API providers offer APIs to return high-quality vectors for each input sequence. However, these APIs are highly throttled and may add time to our total measurement. So, in this case, it is a good idea to host your own embedding model that generates similar sized vectors as the final intended embedding model.

**Use a representative input dataset:** This should contain the same order of magnitude of text documents that you need to test for. Even if you plan to use a smaller portion of the full dataset, size your resources for the full dataset to make sure you have the right performance and cost estimates.

**Plan to do an end-to-end benchmark:** This is in addition to vector embedding benchmarks alone. An end-to-end benchmark could be measuring search quality or accuracy, with vector embeddings being part of a larger system.

Figure 1.8 shows a general architecture to measure two very important KPIs related to vector databases: ingestion time and query latency.

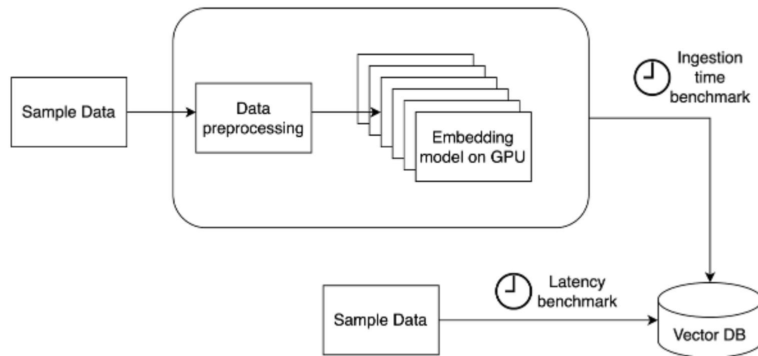


**FIGURE 1.8:** Vector DB benchmarking architecture

Let's walk through the components of this benchmark architecture. The sample architecture diagram in Figure 1.8 displays the core components involved in a typical vector database benchmarking pipeline. Raw text data first goes through several preprocessing steps before getting converted into embedding vectors. Preprocessing includes cleaning, such as removing HTML tags, fixing encoding issues, and handling malformed data. The text is tokenized by splitting it into individual words, phrases, or sentences. All text is converted to lowercase for consistency and common stop words like *a*, *and*, and *the* are removed as they add little semantic meaning. Words are stemmed or lemmatized to their root form to reduce vocabulary size. For example, *running* is converted to *run*. Finally, documents are broken into smaller chunks to allow parallel embedding. This preprocessed text is fed as input to the embedding model, hosted on GPU servers to leverage massive parallelism for efficient computation. The model converts each text chunk into a dense high-dimensional vector representation encoding its semantic meaning. These embeddings are floating-point vectors that are typically 1,000 dimensions for each chunk of text but can vary based on the embedding model used.

The generated vectors are loaded into the vector database, which stores them optimized for low-latency similarity search and retrieval. The database provides APIs to ingest vectors and enables querying using algorithms such as cosine similarity and locality sensitive hashing (LSH). The vectorized dataset powers downstream applications such as search, classification, and recommendations, and in the case of our GenAI chatbot solution, it can be used to augment LLM results by providing useful context from real data.

Two key performance metrics highlighted are total data ingestion time and query latency. Ingestion time measures the complete end-to-end duration to process raw data through embedding and storing vectors in the database.

Query latency indicates how rapidly the database can retrieve similar vectors for a given input. A specific instantiation of this architecture may involve the following setup:

- Using a sample benchmark dataset for indexing such as OSCAR (`https://huggingface.co/datasets/oscar-corpus/OSCAR-2301`)
- Using a library like ray.io (`https://github.com/ray-project/ray`) to both process the data and locally host embedding models in a GPU cluster
- Using an embedding model like `https://huggingface.co/sentence-transformers/all-mpnet-base-v2`

Once again, remember that we can use different data sources, different ways to distribute the ingestion workload and host models, and various embedding models. When running a benchmark using the previous sample architecture and a variety of vector databases, we get the results in Table 1.3:

**TABLE 1.3:** Benchmark results for vector DB

| DATABASE | AVERAGE INGESTION TIME (MS) | P99 INGESTION TIME (MS) | AVERAGE QUERY LATENCY (MS) | P99 QUERY LATENCY (MS) |
|---|---|---|---|---|
| OpenSearch | 2.13 | 35.0 | 11 | 16 |
| Amazon RDS | 1.60 | 7.7 | 83 | 210 |
| Pinecone | 3.80 | 5.1 | 81 | 113 |

Table 1.3 compares the performance of three vector databases: OpenSearch (`https://aws.amazon.com/opensearch-service`), RDS with PG Vector (`https://aws.amazon.com/about-aws/whats-new/2023/05/amazon-rds-postgresql-pgvector-ml-model-integration`), and Pinecone (`https://docs.pinecone.io/docs/python-client`). The test performed uses a 1% sample of the dataset (the full dataset contains more than 600 million records). Metrics are provided for average and $99^{th}$ percentile (P99) latency when ingesting a single record as well as querying for a single vector.

For ingestion latency, OpenSearch took an average of 2.1 ms to ingest a record. Its P99 latency was 35 ms, indicating 99% of ingests finished within 35 ms. RDS had a faster average ingestion latency of 1.6 ms per record. However, its P99 latency was higher at 7.7 ms. Pinecone had the highest ingestion latency, averaging 3.8 ms per record. Its P99 latency was the lowest at 5.1 ms.

For query latency, OpenSearch averaged 11 ms per query with a P99 of 16 ms. RDS had higher latency at an average of 83 ms per query and P99 of 210 ms. Pinecone also had a high query latency, averaging 81 ms per query and P99 of 113 ms.

As we can see from the results, there is no clear winner, and this is not surprising. While PG Vector on RDS provided the lowest average ingestion speed per record, query latencies are higher for this small benchmark. While OpenSearch had the lowest average query latency, it had the highest P99 ingest time per record (once again, for this particular benchmark). This shows how nuanced benchmarks can be and why we need to carefully look at performance numbers.

The main factor affecting total ingestion time is the database write throughput; in fact, if you isolated every other component, the only thing that affects total ingestion time is the write speed. Practically, unless you write the software for the actual vector database, it is not possible to isolate this component. Therefore, the total time to ingest data into the vector database is also affected by the complexity of the embedding model as discussed, GPU resources available (as running multiple copies of the model parallelized across more GPU cores reduces processing time), network transfer speed between embedding servers and the database, and vector dimensionality

(as higher dimensions require more time to generate and ingest). Tuning these factors improves ingestion time. Choosing a simpler and lower-dimensional model, adding GPUs, improving network speed, and using a high write throughput database will reduce overall ingestion duration.

On the other hand, the vector database's query latency is impacted by what indexing algorithms are used. Algorithms like Locality Sensitive Hashing (LSH) can provide faster similarity search than brute-force cosine similarity. Using higher vector dimensions means that you have to process more data during the retrieval step. When setting up a system that uses queues, heavier concurrent loads will lead to longer queues and delays seen across users. Optimizing query latency involves efficient indexing like LSH, reducing vector dimensions, provisioning sufficient read throughput, and testing under peak load.

## Other Factors to Consider

Two other factors are important when we look at the bigger picture: cost and accuracy.

As mentioned earlier, it is important to price out these options based on the full expected workload. Pricing options that are pay-as-you-go and look simple may end up being expensive for indexes with hundreds of millions of records. On the other hand, providers offering a licensing model with tiered pricing may have a nonlinear pricing structure that does not scale with the number of records. For example, the cost would be about $21,000 per month for holding the entire OSCAR dataset in OpenSearch, $17,000 per month for RDS, and $8,400 per month for Pinecone. While the vector database is only a component of a search system, sophisticated, managed services like Amazon Kendra that offer an end-to-end user experience with secure connectors and data parsers from various enterprise data sources may cost an order of magnitude higher for hundreds of millions of records.

The other aspect to look at is accuracy, with the backdrop of how one might trade off cost and performance. The optimal embedding model really depends on balancing accuracy, speed, compute resources, and cost. Simpler models embed efficiently on CPUs, while complex models require GPUs but achieve superior quality. For example, more complex models like BERT and GPT produce higher-quality embeddings but require more processing time versus simpler models such as word2vec. These larger models with more parameters generate better embeddings but are slower to execute, and you may be able to execute only these on GPUs, which increases the cost (unless they are distilled or quantized). Domain-specific models improve performance on related text compared to general-purpose embeddings, especially when these models are available and can be used for your specific use case. Lastly, a larger context window captures greater semantic nuances but needs increased compute power.

Several pre-trained embedding models are available out of the box through libraries such as Hugging Face Transformers. For example, Sentence Transformers (`www.sbert.net/`) provides optimized sentence embedding models like all-MiniLM-L6 (`https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2`), which is compact and efficient. Multilingual models such as mLUKE (`https://arxiv.org/abs/2110.08151`) can handle diverse languages, if that is one of your requirements. Domain-specific models exist as well, like BioClinicalBERT (`https://arxiv.org/abs/1904.03323`) for biomedical texts. If resources are available, it is also a good idea to train or fine-tune your own embedding model for improved search quality. OpenAI's text-embedding-ada-002 uses a large 8,192-token context window for high accuracy. Rather than using ray.io as we showed in this example, you can also consider using Spark for distributed calculation of embeddings. The Spark NLP 5.0 library offers highly optimized models like INSTRUCTOR that can be run on a Spark cluster (`www.johnsnowlabs.com/spark-nlp-5-0-its-all-about-that-search`).

If you are building your own index using libraries like FAISS, you have much more flexibility in terms of indexing as well as search performance. The FAISS package by Meta provides excellent guidelines on when to use certain indexing algorithms (`https://engineering.fb.com/2017/03/29/data-infrastructure/faiss-a-library-for-efficient-similarity-search`).

To summarize this guidance, the optimal index for a vector database depends on the planned number of searches. For less than 10,000 searches, direct computation with a flat index is the most efficient as there is no amortization of build time. Exact search results require using the flat index, as it does not compress vectors or add overhead. When memory is a concern, if exact results are not needed, RAM becomes the constraint to optimize by picking an index that balances precision and speed. With ample memory or a small dataset, the HNSW index

provides fast and highly accurate searches through its graph traversal algorithm but uses more RAM with its 4 to 64 links per vector.

Alternatively, an inverted file index with product quantization compresses vectors through OPQ dimension reduction and PQ quantization into codes using less memory, with two tuning parameters of reranking k and nprobe for the precision versus speed trade-off. As dataset size grows, clustering vectors first via k-means or HNSW graph partitioning into IVF buckets optimizes storage and lookup time, with nprobe buckets scanned per search. Training set size and number of clusters increase with data volume. Multilevel clustering further improves indexing of more than 100 million vectors. Finally, GPU support may be required, in which case flat, OPQ, and IVFK work on GPUs, while HNSW is CPU-only currently. Considering these factors holistically based on the use case leads to selecting the optimal indexing approach.

Thoroughly benchmarking ingestion and query performance under realistic workloads is key for an efficient vector database pipeline. The choice of embedding model and tasks, such as optimizing GPU resources, minimizing network transfers, leveraging fast ingest and retrieval algorithms, and provisioning adequate database throughput are important considerations. Benchmarks can predict production behavior and highlight areas for optimization; however, benchmarks like the ones discussed in this section can be nuanced and have to be used carefully in deciding what vector database to go with for production use cases. If you are interested in another view of the same discussion, take a look at this resource that compares some of the vector DBs mentioned here: `https://benchmark.vectorview.ai/vectordbs.html`.

## Cost Assessment of the Large Language Model Component

Now let's look at another important component in the GenAI chatbot architecture; arguably this is the most important, central component. There are generally three ways to access LLMs today.

- Through a console application (e.g., ChatGPT, claude.ai)
- Through an API to a chat model, without the need to host the model itself
- Through a self-hosted API that includes the LLM and any serving stack

These three ways of accessing LLMs incur varying costs. When building a conversational AI application like a chatbot, virtual assistant, or content generation tool, a key component is selecting the right large language model to power the underlying AI. As AI continues to permeate our digital lives, more companies are launching LLMs with varying capabilities and pricing models.

How does one pick the most cost-effective LLM while meeting the performance needs of the application? We will be uncovering this question through several chapters that follow, focusing on ways to optimize cost. But first let's explore this question by revisiting the hypothetical scenario with a virtual assistant that is being built to handle multiple user sessions in parallel. Once again, it is anticipated to have 100 concurrent users chatting with the assistant at any given time. Based on user testing, it is estimated each user will make around 100 requests per hour through natural conversation with the assistant. If you have played around with applications like ChatGPT, this estimate may match what you experience in practice. You may use it to help you with work and may interact with it once a minute (and even that may be an overestimate).

Figures 1.9 and 1.10 explore this hypothetical scenario by changing a few variables that impact the price per hour for this application. Detailed pricing and usage comparison tables can provide critical insights when evaluating large language models. The tables compare leading LLMs like GPT-3, Claude, and Amazon SageMaker across metrics including cost per hour, requests per minute, usage limits, and optimizations like batching. By analyzing the variables in the tables for a projected workload, you can select an optimal LLM based on performance needs and cost constraints.

The tables shown in Figures 1.9 and 1.10 are snapshots of a cost calculator highlighting how factors such as scaling out instances and leveraging innovations such as continuous batching can transform an LLM from a cost-prohibitive to a cost-effective solution for powering AI applications. The analysis shows that, while powerful, choices like GPT-3 and GPT-4 from OpenAI incur high costs at more than $180 per hour to support 100 concurrent users.

| Model / Cost item | GPT4 32K | GPT 3.5 Turbo 16k | Claude V2 100K or 12K | Claude Instant 100K | Falcon 40B on SageMaker (1 g5.12xlarge) | Falcon 40B on SageMaker (1 g5.48xlarge) | Falcon 40B on SageMaker (N instances g5.48xlarge *with static batching*) | Falcon 40B on SageMaker (N g5.48xlarge *with continuous batching*) |
|---|---|---|---|---|---|---|---|---|
| Input, per MM tokens | $60 | $3 | $11.02 | $1.63 | - | - | - | - |
| Output, per MM tokens | $120 | $4 | $32.68 | $5.51 | - | - | - | - |
| Input tokens per request | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Output tokens per request | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Number of instances | - | - | - | - | 1 | 1 | 1 | 1 |
| Max requests per minute | 3,500 | 2,000 | 300 | 300 | 12 | 20 | 20 | 20 |
| Max tokens per minute | 350,000 | 180,000 | 150,000 | 150,000 | - | - | - | - |
| Number of concurrent user sessions | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Number of requests per user session per hour | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| At rate limit (max RPM)? | FALSE | FALSE | FALSE | FALSE | TRUE | TRUE | TRUE | FALSE |
| At rate limit (max tokens per minute) | FALSE | FALSE | FALSE | FALSE | No token limit | No token limit | No token limit | No token limit |
| cost per user per turn | $0.018000 | $0.000700 | $0.004370 | $0.000714 | - | - | - | - |
| per hour cost | $180.00 | $7.000 | $43.70 | $7.14 | $7.09 | $20.36 | $20.36 | $20.36 |

**FIGURE 1.9:** Snapshot 1 of the cost calculator showing costs per hour for various model options based on input factors and how rate limits can be exceeded in certain cases

| Model / Cost item | GPT4 32K | GPT 3.5 Turbo 16k | Claude V2 100K or 12K | Claude Instant 100K | Falcon 40B on SageMaker (1 g5.12xlarge) | Falcon 40B on SageMaker (1 g5.48xlarge) | Falcon 40B on SageMaker (N instances g5.48xlarge *with static batching*) | Falcon 40B on SageMaker (N g5.48xlarge *with continuous batching*) |
|---|---|---|---|---|---|---|---|---|
| Input, per MM tokens | $60 | $3 | $11.02 | $1.63 | - | - | - | - |
| Output, per MM tokens | $120 | $4 | $32.68 | $5.51 | - | - | - | - |
| Input tokens per request | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Output tokens per request | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| Number of instances | - | - | - | - | 14 | 1 | 9 | 1 |
| Max requests per minute | 3,500 | 2,000 | 300 | 300 | 168 | 20 | 180 | 190 |
| Max tokens per minute | 350,000 | 180,000 | 150,000 | 150,000 | - | - | - | - |
| Number of concurrent user sessions | 100 | 100 | 100 | 100 | 100 | 13 | 100 | 100 |
| Number of requests per user session per hour | 100 | 100 | 100 | 100 | 100 | 100 | 100 | 100 |
| At rate limit (max RPM)? | FALSE | FALSE | FALSE | FALSE | FALSE | TRUE | FALSE | FALSE |
| At rate limit (max tokens per minute) | FALSE | FALSE | FALSE | FALSE | No token limit | No token limit | No token limit | No token limit |
| cost per user per turn | $0.018000 | $0.000700 | $0.004370 | $0.000714 | - | - | - | - |
| per hour cost | $180.00 | $7.000 | $43.70 | $7.14 | $99.26 | $20.36 | $183.24 | $20.36 |

**FIGURE 1.10:** Snapshot 2 of the cost calculator showing costs per hour for various model options based on input factors, showing how vertical and horizontal scaling, dynamic batching can help satisfy request rate demands

More affordable options are needed, especially because models today have different capabilities, so it may be beneficial to consider multiple models and model providers to balance cost with performance. This is also exactly why companies like OpenAI and Anthropic provide multiple versions of models such as one cheaper, quicker version with decent performance (Claude Instant) and a more powerful but also more expensive version (Claude v2 100,000). Figures 1.9 and 1.10 show a clear difference in cost between larger and smaller models offered by these LLM API providers (GPT 4 versus GPT 3.5 Turbo, or Claude V2 versus Claude Instant).

Based on published rate limits, Claude Instant would allow 300 requests per minute across users, while Claude V2 increases this to 2,000 RPM. Both comfortably satisfy the 100 RPM per user estimate in our hypothetical scenario without hitting any usage limits, but Claude Instant is cheaper (as expected) at $7.14 per hour, so it seems like the obvious choice.

But can we find an even better option? Of course, we have overloaded the term *better* here. What is better for one use case may mean decent performance at lower cost, whereas for another use case, this necessarily means higher performance. Choosing Claude Instant or GPT 3.5 Turbo may mean less accuracy or fewer general capabilities to solve multiple complex use cases like their larger counterparts.

This cost-performance trade-off means that we need to look deeper into self-hosting, or shared-hosting options. Here's where cloud services like Amazon SageMaker and others show promise. Running the open-source Falcon 40B model leveraging SageMaker optimizations allows requests to be served at a very modest 12 RPM from a single compute instance based on static batching. With 100 users doing 100 requests per minute, this limit would be exceeded with only one g5.12xlarge instance. Vertical scaling is usually the next thing to think of when trying to scale up self-hosted LLMs (or any model for that matter).

Looking at Figure 1.10, we would need at least 14 instances costing about $100 per hour to satisfy our rate of incoming requests, in the ideal case. A larger instance, say an ml.g4.48xlarge can handle almost 1.6x the number of requests, but with only one instance and naïve static batching, this will still hit our rate demand constraint and cost a bit more. Using nine of these instances costs as much as one of the costliest models out there per hour, and the accuracy of results and general applicability of a central model may not be as great as GPT 4, for example. Not bad, but can further optimization be achieved?

It turns out we can use another trick: continuous batching. By dynamically batching requests before they are fed to the model, throughput increases dramatically. A 9.5x improvement is estimated based on the 512 token requests (see `www.anyscale.com/blog/continuous-batching-llm-inference`). This lets us fully utilize a single instance costing a modest $20 per hour to satisfy all requests. The big caveat is that the model used is a Falcon 40B; even though this ranks pretty high on the open-source leaderboard, we know that its performance lags behind more advanced models like GPT4.

While models like GPT 3.5 Turbo and Claude instant provide a simple, low-cost option, SageMaker with an open-source model unlocks higher-scale use cases with more flexibility on your LLM inference stack. You also have the option of considering a cost budget for a given level of expected performance as a factor in deciding your architecture in this case. With instance sizing and batching optimizations, SageMaker allowed a 100x increase in request capacity over Claude while cutting cost by more than 60%, assuming that the performance for that particular use case is equivalent. The comparison here uses the AWS service SageMaker to demonstrate the importance of knowing how to perform this benchmark but several other cloud providers and SaaS companies provide competitive options for self-hosting.

This is not meant to be a full benchmark study, but it highlights key considerations when evaluating LLMs for GenAI applications. Usage limits must be accounted for in addition to raw model pricing. Factors like batching, model compression or quantization, and instance or worker scale-out capacity play a crucial role. Testing different configurations using projected workload numbers helps find the sweet spot between cost, performance, and flexibility that works well for your particular team. There may be factors beyond what are explained here as technical indicators that may inform a choice. For example, deep expertise in Kubernetes-based stacks may be a huge motivation to build this entire stack on Kubernetes. Unlocking innovations like continuous deployment and batching and understanding the capabilities and constraints of modern LLMs transform the cost-performance landscape to enable solutions for delivering next-generation AI applications.

## SUMMARY

This chapter introduced the concept of LLMs and GenAI, starting from the history of language models to the three-layer GenAI stack. We then took a sample application (a GenAI chatbot) and discovered the various components that may be necessary to build out this solution. Based on this blueprint, we covered various ways to benchmark important components of this architecture while balancing performance, cost, and accuracy.