eBook

GenAI Architectures on Databricks

Including prompt templates, RAG, fine-tuning and more

databricks

Contents

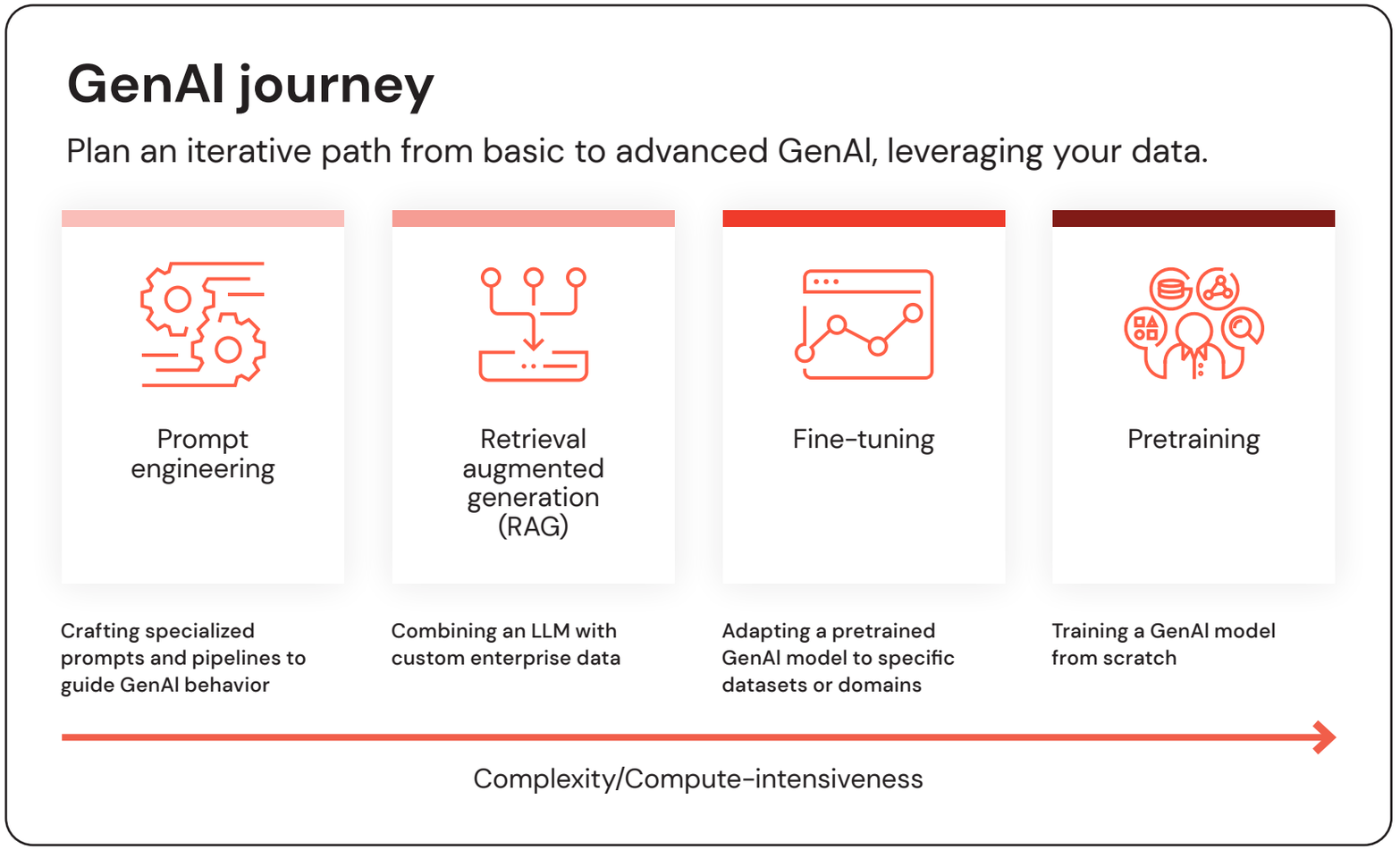
Introduction

In today's rapidly changing AI landscape, generative AI has become a top priority for many organizations and is at the forefront of innovation. When surveyed for an MIT study, 91% of organizations reported they are experimenting with or investing in generative AI. To drive this point even further, an IBM study shows that 75% of CEOs say companies with advanced GenAI will have a competitive advantage. Today there are enormous gains to be found by embracing generative AI — and those who are able to leverage it most effectively will be the ones leading the future of business.

Generative AI, a subset of artificial intelligence, focuses on creating new content — text, images or even code — from existing data. It's not just about analyzing data but about generating new, previously unseen outputs that can think, imagine and innovate. This capability opens up a new realm of possibilities for businesses looking to stay ahead of the curve.

There are many challenges along the path to fully embracing generative AI. One of these challenges is understanding both what it means to actually implement generative AI in your organization and what current architecture patterns currently exist. This book provides a foundational explanation of the different concepts and architectures that organizations looking to implement GenAI are likely to encounter.

We've organized this book as a journey that ascends the path of complexity throughout the generative AI experience. Starting with the easiest point of entry, you'll progress through architectures and concepts that get more complex and involved. The following diagram summarizes this journey.



Before diving into the more detailed architecture of these stages, we've provided a brief overview of each one. Note that these stages are not mutually exclusive and often build upon each other (i.e., you will likely still be using prompt engineering with a fine-tuned or pretrained model).

Prompt engineering: At the heart of the generative AI interaction model lies prompt engineering, a technique that involves crafting queries or instructions to guide AI in generating desired outputs. This simple yet powerful approach allows even nontechnical practitioners to harness AI's creative power, enabling personalized customer experiences, content creation and innovative product designs.

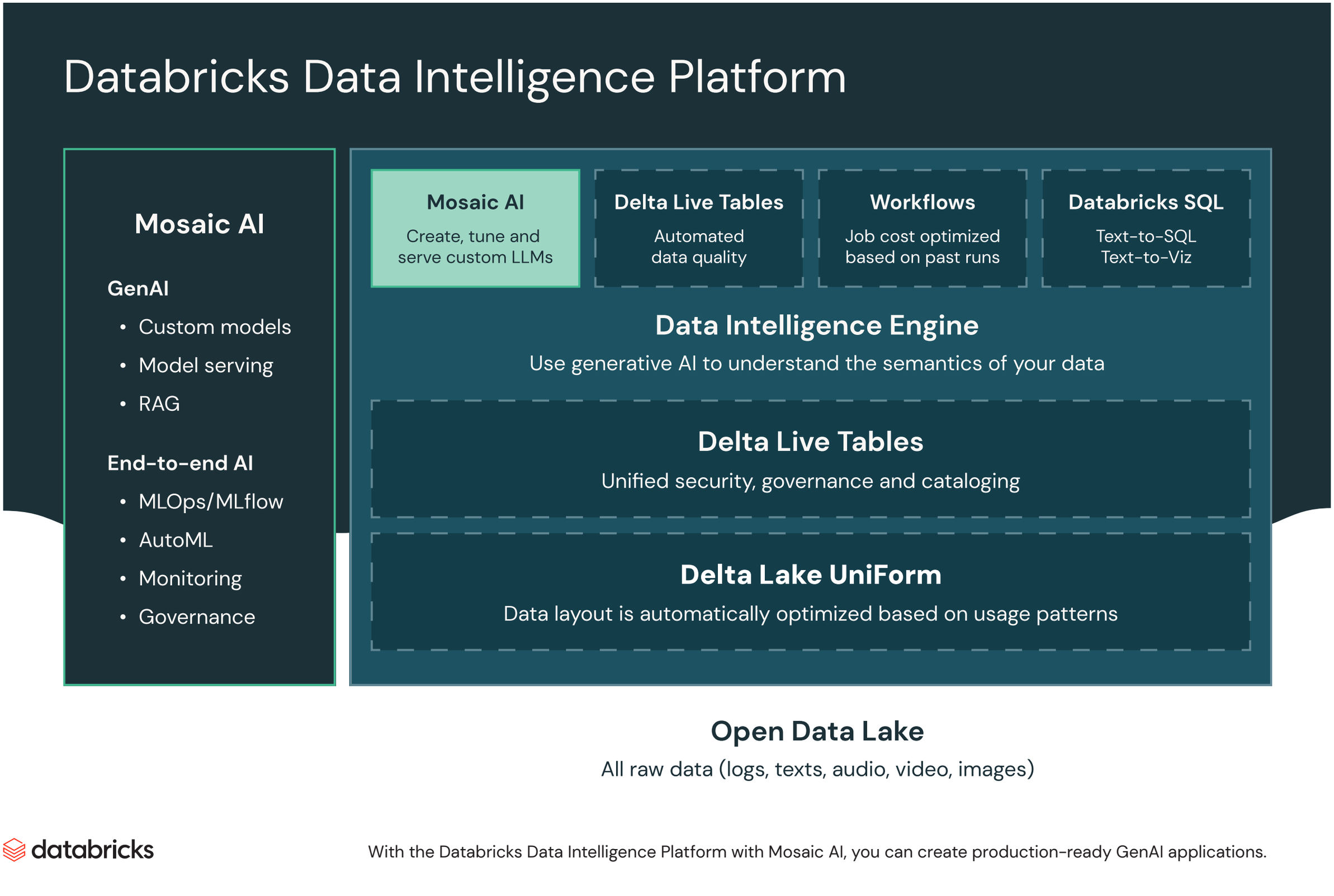
Retrieval augmented generation (RAG): RAG architecture combines the best of both worlds — retrieving relevant information from a database and generating coherent, contextually appropriate responses. This technology can revolutionize customer service, market research and any area where rapid, informed responses are valuable.

Fine-tuning: Fine-tuning is all about customizing AI models to fit specific organizational needs and contexts, enhancing their accuracy and relevance. It allows businesses to adapt pretrained models with their data, optimizing AI performance for unique challenges and opportunities, and it lets organizations apply more control to their models and responses.

Pretraining: For those looking for maximum control over their model, pretraining refers to the concept of creating an AI model from scratch. While it does involve the highest level of compute power and large training datasets, it gives organizations full control over their AI models to maximize both performance and cost.

LLM Operations/evaluation: Large language model (LLM) operations and evaluation encompass the processes of deploying, monitoring and assessing the performance of these models in real-world scenarios. This stage involves continuous testing and tweaking to ensure models respond accurately and ethically to user queries across diverse contexts. Operational excellence in LLMs is achieved through rigorous performance metrics, error analysis and feedback loops that refine the models over time. Additionally, evaluating LLMs also includes ensuring compliance with privacy standards and ethical guidelines to foster trust and reliability in generative AI systems. This phase is crucial for maintaining the integrity and utility of AI deployments in any enterprise setting.

It's also important to note that none of these concepts exist in a vacuum. Typically, you also need a platform that can support these GenAI architectures and provide the governance and lineage to enable users to create production-ready applications that will bring value to their organizations. The Databricks Data Intelligence Platform with Mosaic AI provides both the foundation and the tools to create, govern and deploy the different GenAI architectures that will be explored further in this book.



databricks

With the Databricks Data Intelligence Platform with Mosaic AI, you can create production-ready GenAI applications.

The field of generative AI and LLMOps is quickly evolving, and the architectures reviewed in this eBook have emerged as key components to consider while building out GenAI-based applications. It's also important to note that some, but not necessarily all, of the following components make up a single GenAI- or LLM-based application. As we navigate through each chapter, we'll explore how these architectures not only advance technological capabilities but also offer a blueprint to jump-start implementation. This book will provide the knowledge and insights you need to embark on this journey so you can harness the power of generative AI to reshape the future of your organization.

In this eBook, you'll discover architectures that cover:

- Prompt engineering and adjusting model behavior using prompt templates
- RAG systems, what they entail and how to effectively tie them together
- Fine-tuning models to better align their core behavior to your specific needs
- When to pretrain your own model and how to do it in a cost-effective manner

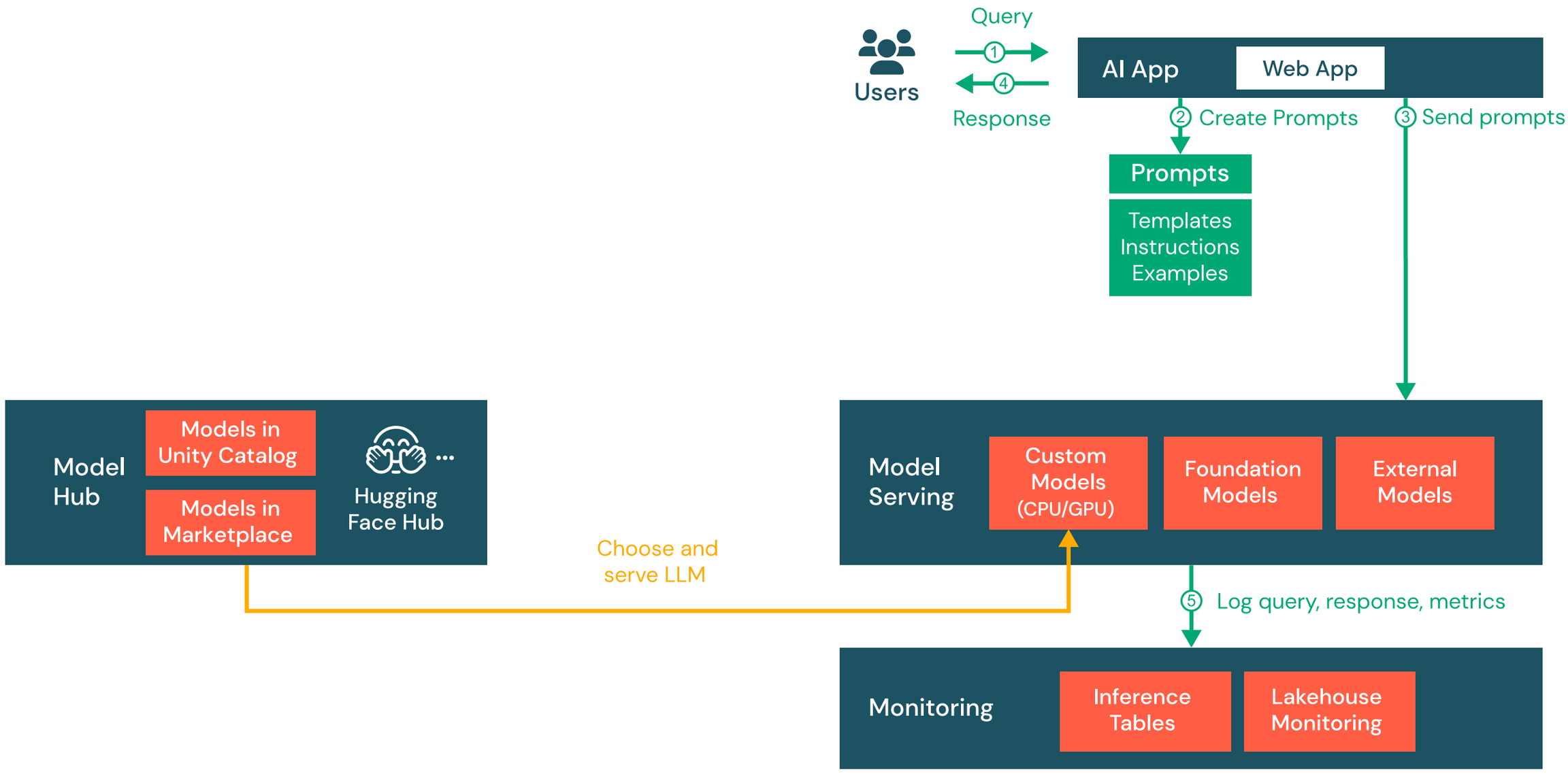
Prompt Engineering

Prompt engineering is the practice of adjusting the text prompts given to an LLM to elicit more accurate or relevant responses. While it's a nascent field, several best practices are emerging. We'll discuss tips and best practices and link to useful resources.

1. Prompts and prompt engineering are model-specific. A prompt given to two different models will generally not produce the same results. Similarly, prompt engineering tips do not apply to all models. In extreme cases, many LLMs have been fine-tuned for specific natural language processing (NLP) tasks and do not require prompts. On the other hand, very general LLMs benefit greatly from carefully crafted prompts.
2. When approaching prompt engineering, go from simple to complex: track, templatize and automate.
    a. Start by tracking queries and responses so that you can compare them and iterate to improve prompts. Existing tools such as MLflow provide tracking capabilities; see MLflow LLM Tracking for more details. Checking structured LLM pipeline code into version control also helps with prompt development, and Git diffs allow you to review prompt changes over time. Also, see the section in our eBook on packaging models and pipelines for more information about tracking prompt versions.
    b. Then consider using tools for building prompt templates, especially if your prompts become complex. Newer LLM-specific tools such as LangChain and LlamaIndex provide such templates and more.
    c. Finally, consider automating prompt engineering by replacing manual engineering with automated tuning. Prompt tuning turns prompt development into a data-driven process akin to hyperparameter tuning for traditional ML. The DSPy framework is a good example of a tool for both defining and automatically optimizing LLM pipelines.

3. Most prompt engineering tips currently published online are for ChatGPT, due to its immense popularity. Some of these generalize to other models as well. We'll provide a few tips here:
    a. Use clear, concise prompts, which may include an instruction, context (if needed), a user query or input, and a description of the desired output type or format.
    b. Provide examples in your prompt ("few-shot learning") to help the LLM understand what you want.
    c. Tell the model how to behave, such as admitting if it cannot answer a question.
    d. Tell the model to think step-by-step or explain its reasoning.
    e. If your prompt includes user input, use techniques to prevent prompt hacking, such as making it very clear which parts of the prompt correspond to your instruction vs. user input.

The following diagram depicts application architecture for prompt engineering of an LLM.

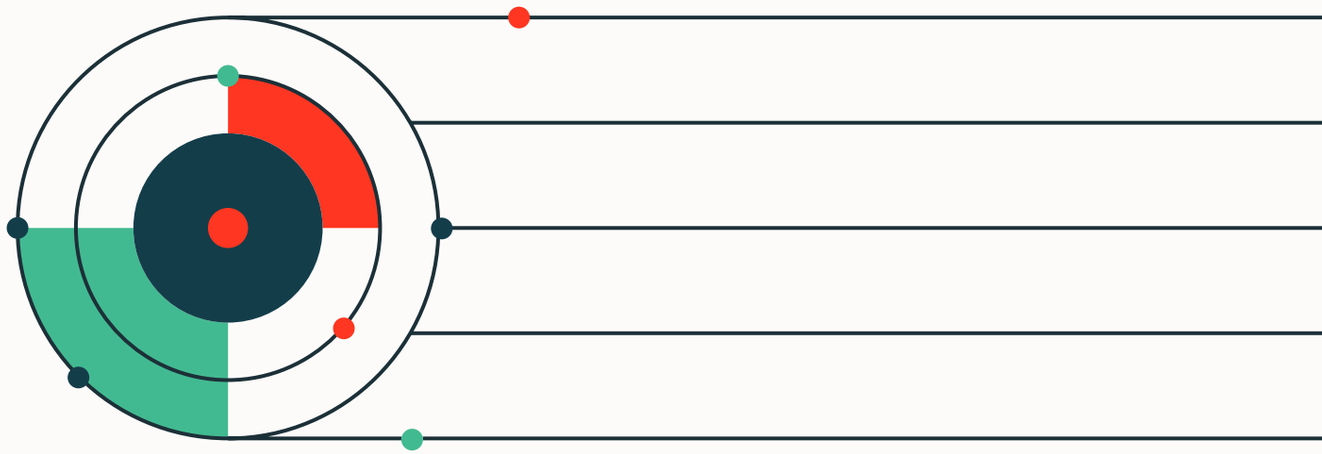


When considering which approach to take, like any ML application, it's crucial to assess your specific needs, business objectives and constraints. A good rule of thumb is to start with the simplest approach possible, such as prompt engineering with a third-party LLM API, to establish a baseline. Once this baseline is in place, you can incrementally integrate more sophisticated strategies like RAG or fine-tuning to refine and optimize performance. Using standard MLOps tools such as MLflow is equally crucial in LLM applications to track performance over different approach iterations.

A range of factors will inform the choice of technique, including (but not limited to):

- The volume and quality of your data
- Required application response time
- Computational resources available
- Budgetary constraints
- The specific domain or application at hand

Regardless of the technique selected, building a solution in a well-structured, modularized manner ensures that you'll be prepared to iterate and adapt as you uncover new insights and challenges. In the following sections, we'll look at each of these techniques to leverage your data, outlining the considerations and best practices associated with each.



Retrieval Augmented Generation (RAG)

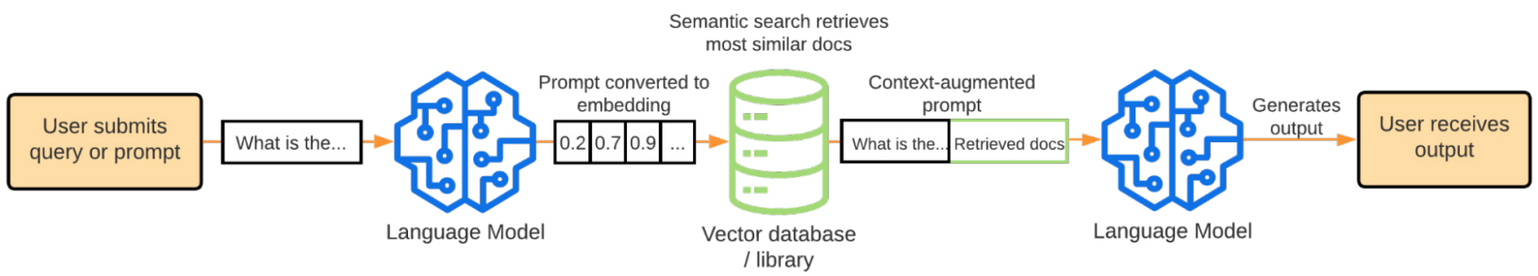
Retrieval augmented generation (RAG) offers a dynamic solution to a fundamental limitation of LLMs: their inability to access information beyond their training data cutoff point. RAG connects static LLMs with real-time data retrieval. Instead of relying solely on pretrained knowledge, a RAG workflow pulls relevant information from external sources and injects it into the context provided to a model. RAG workflows are often used in document question-answering use cases where the answer might evolve over time or come from up-to-date, domain-specific documents that were not part of the model's original training data.

RAG provides a number of key benefits:

- **LLMs as reasoning engines:** RAG ensures that model responses are not just based on static training data. Rather, the model is used as a reasoning engine augmented with external data sources to provide responses that are up-to-date, accurate and relevant.
- **Reduce hallucinations:** By grounding the model's input on external knowledge, RAG attempts to mitigate the risk of producing hallucinations — instances where the model might generate inaccurate or fabricated information.
- **Domain-specific contextualization:** A RAG workflow can be tailored to interface with proprietary or domain-specific data. This ensures that the LLM outputs are not only accurate but also contextually relevant, catering to specialized queries or domain-specific needs.
- **Efficiency and cost-effectiveness:** In use cases where the aim is to create a solution adapted to domain-specific knowledge, RAG offers an alternative to fine-tuning LLMs (more detail in our Big Book of MLOps) by enabling in-context learning without the overhead associated with traditional fine-tuning. This can be particularly beneficial in scenarios where models need to be frequently updated with new data. Note that whereas RAG reduces development time and cost, fine-tuning reduces inference time and cost.

Typical RAG workflow

There are many ways to implement a RAG system, depending on specific needs and data nuances. Below we outline one commonly adopted workflow to provide a foundational understanding of the process.



Source: Databricks — Large Language Models: Application Through Production

1. **User prompt:** The process begins with a user query or prompt. This input serves as the foundation for what the RAG system aims to retrieve.
2. **Embedding conversion:** The user's prompt is transformed into a high-dimensional vector (or embedding). This representation captures the semantic essence of the prompt and is used to search for relevant information in the database.
3. **Information retrieval:** Using the prompt's vector representation, RAG queries the external data sources or databases. A vector database (see below) can be particularly effective here, allowing for efficient and accurate data fetching based on similarity measures.
4. **Context augmentation:** The most relevant pieces of information are retrieved and concatenated to the initial prompt. This enriched context provides the LLM with supplemental information that can be used to produce a more informed response.
5. **Response generation:** With the augmented context, the language model processes the combined data (original prompt + retrieved information) and generates a contextually relevant response.
6. **Feedback loop:** Some RAG implementations might encompass a multi-hop feedback mechanism (see this paper for an example). In cases where the response is deemed unsatisfactory, the system can revisit its search criteria, tweak the context or even refine its retrieval strategy, subsequently generating a new answer.

Vector databases

RAG hinges on the efficient retrieval of relevant data. At the heart of this retrieval process are embeddings — numerical representations of text data. To understand how RAG utilizes these vectors, let's first distinguish between a number of terms.

Vector index: A specialized data structure optimized to facilitate similarity search within a collection of vector embeddings

Vector library: A tool to manage vector embeddings and conduct similarity searches. They predominantly:

- Operate on in-memory indexes
- Focus solely on vector embeddings, often requiring a secondary storage mechanism for the actual data objects
- Are typically immutable; postindex creation changes necessitate a complete rebuild of the index

Examples of vector libraries include FAISS, Annoy and ScaNN.

Vector database: Distinguished from vector libraries, vector databases:

- Store both the vector embeddings and the actual data objects, permitting combined vector searches with advanced filtering
- Offer full Create, Read, Update, Delete (CRUD) operations, allowing dynamic adjustments without rebuilding the entire index
- Are generally better suited for production-grade deployments due to their robustness and flexibility

In the RAG workflow described in the previous section, we assume that the retrieved external data has been converted into embeddings. These embeddings are stored in a vector index, managed either through a vector library or more holistically with a vector database. The choice between the two often hinges on the application's specific requirements, the volume of data and the need for dynamic updates.

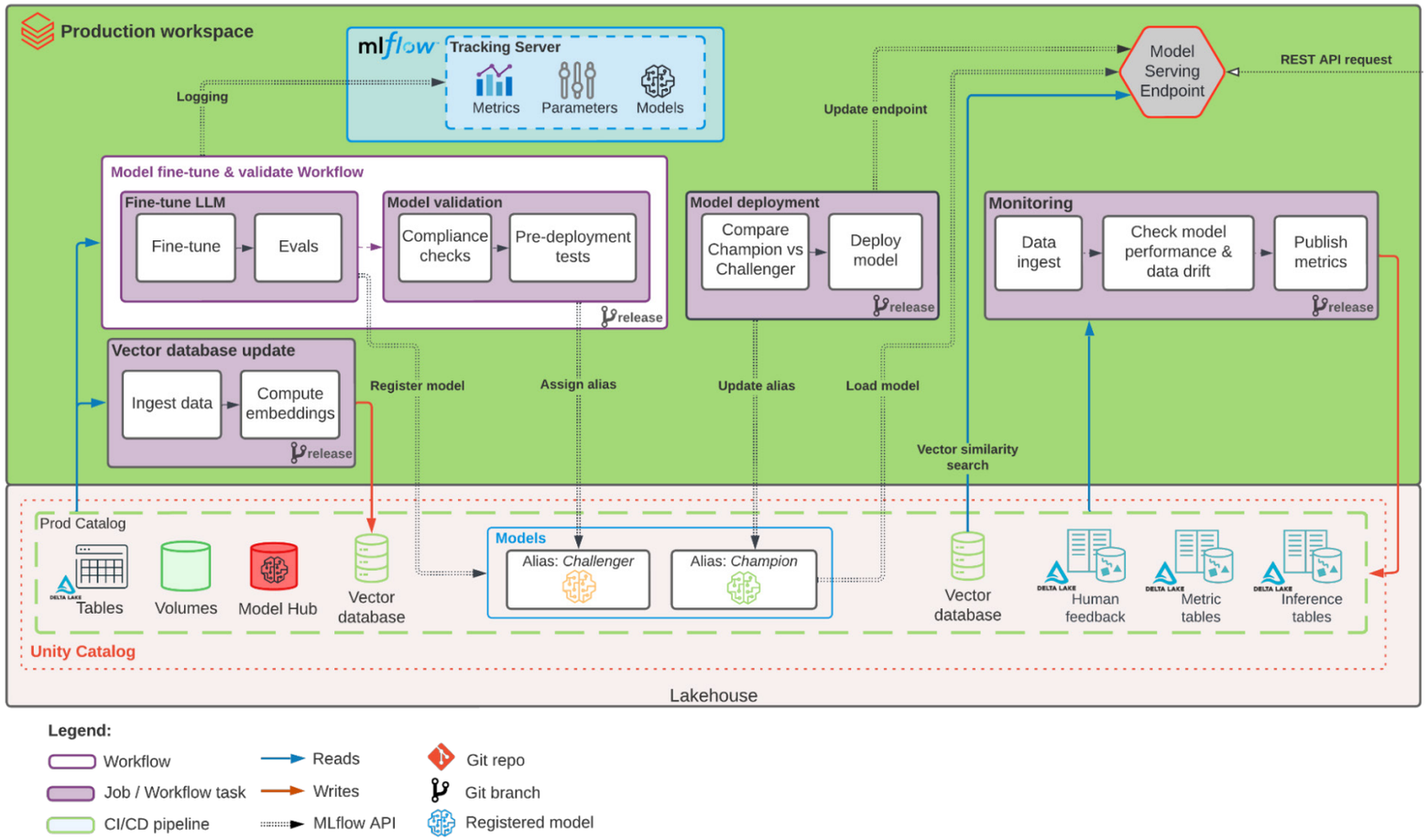
BENEFITS OF VECTOR DATABASES IN A RAG WORKFLOW

Following are a number of reasons why a vector database may be preferable over a vector library when implementing a RAG workflow:

- **Holistic data management:** Storing both vector embeddings and original data objects allows a RAG system to retrieve relevant context without needing to integrate with multiple systems
- **Advanced filtering:** Beyond just similarity search, vector databases allow for application of filters on the stored data objects. This ensures more precise retrieval, enabling RAG to fetch contextually relevant and specific information based on both semantic similarity and metadata criteria.
- **Dynamic updates:** In fast-evolving domains, the ability to update the database without a complete rebuild of the vector index ensures that the language model accesses the up-to-date information
- **Scalability:** Vector databases are designed to handle vast amounts of data, ensuring that as data grows, the RAG system remains efficient and responsive

Detailed architecture examples

RAG WITH A FINE-TUNED OSS MODEL



This diagram details what an example production RAG architecture looks like when deployed on the Databricks Data Intelligence Platform.

Resources

- **Databricks Blog:** Using MLflow AI Gateway and Llama 2 to Build Generative AI Apps
- **Databricks Tutorial:** Deploy Your LLM Chatbot With Retrieval Augmented Generation (RAG), Foundation Models and Vector Search
- LangChain question-answer example

Fine-Tuning

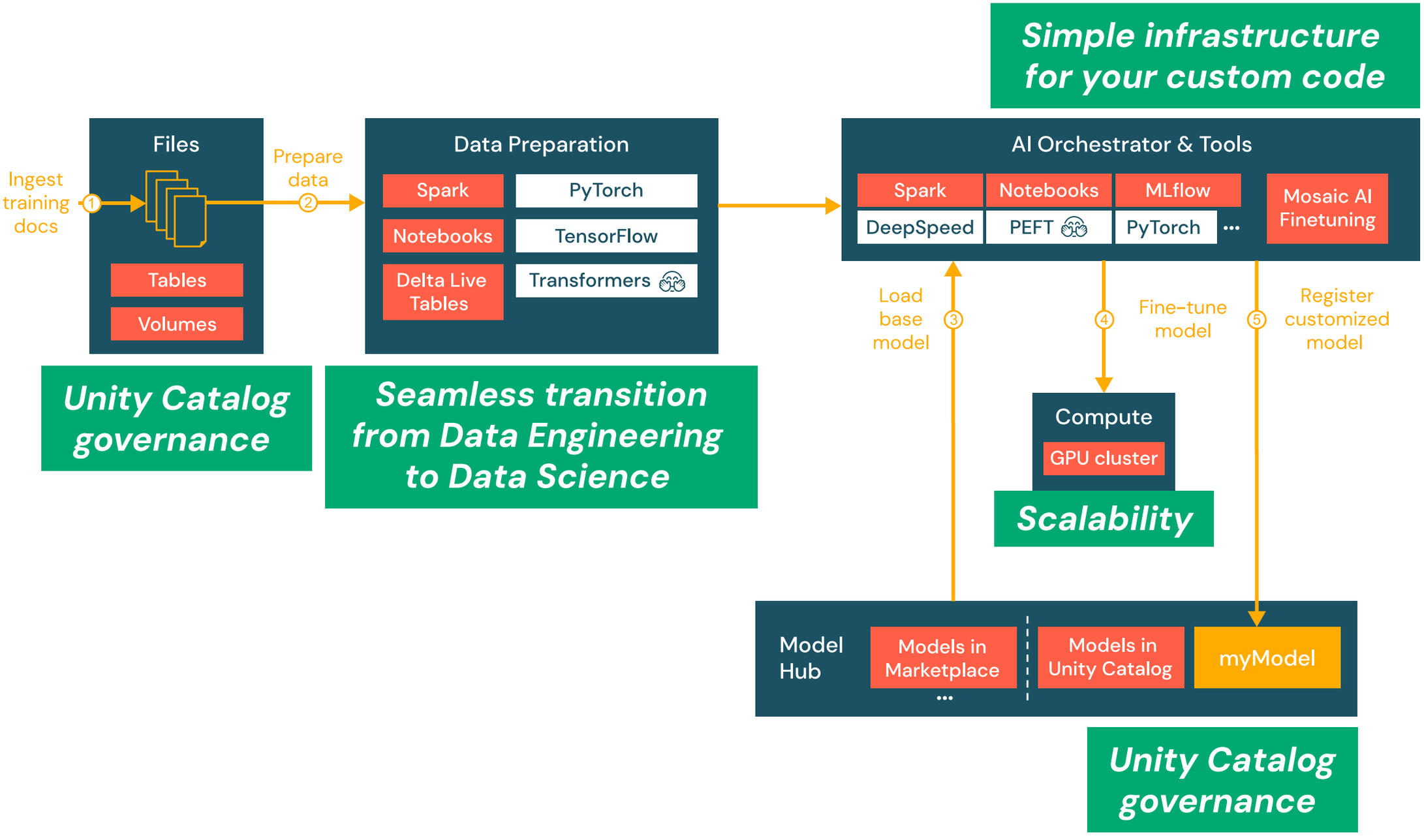
While prompt engineering and retrieval augmented generation (RAG) offer robust methods to guide a model's behavior, there are instances where they might not be adequate, especially for entirely novel or domain-specific tasks. In such cases, fine-tuning an LLM can be advantageous.

Fine-tuning is the process of adapting a pretrained LLM on a comparatively smaller dataset that is specific to an individual domain or task. During the fine-tuning process, only a small number of weights are updated, allowing the LLM to learn new behaviors and specialize in certain tasks.

The term "fine-tuning" can refer to several concepts, with the two most common forms being:

- **Supervised instruction fine-tuning:** This approach involves continuing training of a pretrained LLM on a dataset of input-output training examples — typically conducted with thousands of training examples. Instruction fine-tuning is effective for question-answering applications, enabling the model to learn new specialized tasks such as information retrieval or text generation. The same approach is often used to tune a model for a single specific task (e.g., summarizing medical research articles), where the desired task is represented as an instruction in the training examples.
- **Continued pretraining:** This fine-tuning method does not rely on input and output examples but instead uses domain-specific unstructured text to continue the same pretraining process (e.g., next token prediction, masked language modeling). This approach is effective when the model needs to learn new vocabulary or a language it has not encountered before.

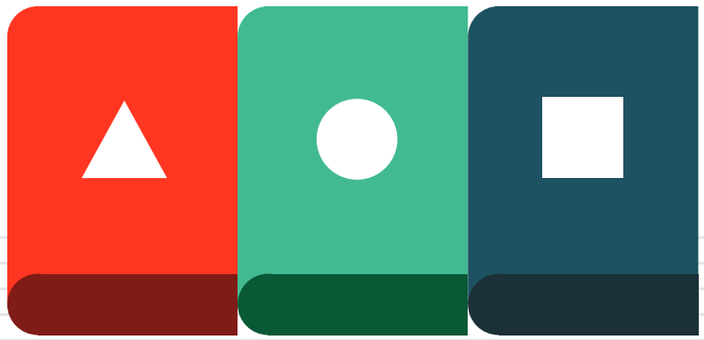
The following diagram illustrates an application workflow for fine-tuning a model.



When to use fine-tuning

Choosing to fine-tune an open source LLM offers several advantages that tailor the model's behavior to better fit specific organizational needs. The following are motivations for choosing to fine-tune:

- **Customization and specialization:** LLMs trained on large, generic datasets — such as those provided by third-party APIs — often have broad knowledge but lack depth in niche areas. Fine-tuning allows organizations to specialize models for their specific domains or applications.
- **Full control over model behavior:** Fine-tuning provides granular control over the model's outputs. It allows organizations to address specific biases, enforce correctness and refine a model's behavior based on feedback.



Fine-tuning in practice

Fine-tuning, while advantageous, comes with practical considerations. Often, the optimal approach is not solely fine-tuning, but a blend of fine-tuning and retrieval methods like RAG. For example, you might fine-tune a model to generate specific outputs but also use RAG to inject data relevant to user queries.

Notably, fine-tuning large models with billions of parameters comes with its own set of challenges, particularly in terms of computational resources. To accommodate the training of such models, modern deep learning libraries like PyTorch FSDP and DeepSpeed employ techniques like Zero Redundancy Optimizer (ZeRO), Tensor Parallelism and Pipeline Parallelism. These methods optimize the distribution of the model training process across multiple GPUs, ensuring efficient use of resources.

A resource-efficient alternative to fine-tuning all parameters of an LLM is a class of methods referred to as parameter-efficient fine-tuning (PEFT). PEFT methods such as LoRA and IA3 fine-tune LLMs by adjusting a limited subset of model parameters or a small number of extra-model parameters. These approaches not only conserve GPU memory, they often match the performance of full model fine-tuning. Open source libraries such as the Hugging Face PEFT library have been developed to easily implement this family of fine-tuning techniques for a subset of models.

It's worth noting the significance of human feedback in a fine-tuning context, especially for applications like question-answering systems or chat interfaces. Such feedback helps refine the model, ensuring its outputs align more closely with user needs and expectations. To facilitate this iterative process, you can use libraries like Hugging Face trl or CarperAI trlX to apply methods like proximal policy optimization (PPO) which incorporate human feedback into the fine-tuning process.

Pretraining

Pretraining a model from scratch refers to the process of training a language model on a large corpus of data (e.g., text, code) without using any prior knowledge or weights from an existing model. This is in contrast to fine-tuning, where an already pretrained model is further adapted to a specific task or dataset. The output of full pretraining is a base model that can be directly used or further fine-tuned for downstream tasks.

When to use pretraining

Choosing to pretrain an LLM from scratch is a significant commitment, in terms of both data and computational resources. Here are some scenarios where it makes sense:

1. **Unique data sources:** If you possess a unique and extensive corpus of data that is distinct from what available pretrained LLMs have seen, it might be worth pretraining a model to capture this uniqueness.
2. **Domain specificity:** Organizations might want a base model tailored to their specific domain (e.g., medical, legal, code) to ensure even the foundational knowledge of the model is domain-specific.
3. **Full control over training data:** Pretraining from scratch offers transparency and control over the data the model is trained on. This may be essential for ensuring data security, privacy and custom tailoring of the model's foundational knowledge.
4. **Avoiding third-party biases:** Pretraining ensures that your LLM application does not inherit biases or limitations from third-party pretrained models.

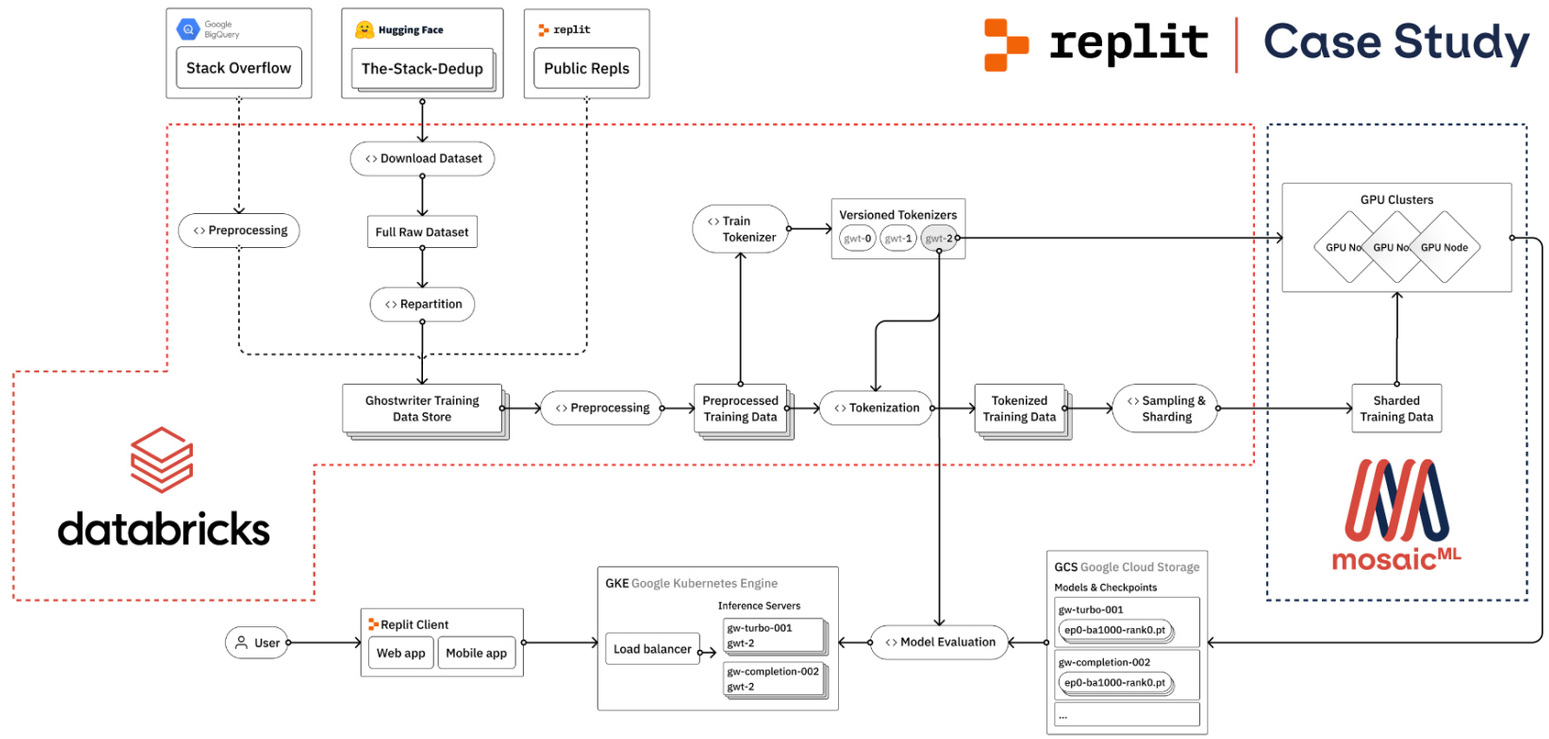
Pretraining in practice

Given the resource-intensive nature of pretraining, careful planning and sophisticated tooling are required. Libraries like PyTorch FSDP and DeepSpeed, mentioned previously in the fine-tuning section, are similarly required for their distributed training capabilities when pretraining an LLM from scratch. The following only scratches the surface of some of the considerations to take into account when pretraining an LLM:

- **Large-scale data preprocessing:** A pretrained model is only as good as the data it is trained on. Thus, it becomes vitally important to ensure robust data preprocessing is conducted prior to model training. Given the scale of the training data involved, this preprocessing typically requires distributed frameworks like Apache Spark™. Consideration must be given to factors such as dataset mix and de-duplication techniques to ensure the model is exposed to a wide variety of unique data points.
- **Hyperparameter selection and tuning:** Before executing full-scale training of an LLM, determining the set of optimal hyperparameters is crucial. Given the high computational cost associated with LLM training, extensive hyperparameter sweeps are not always feasible. Instead, make informed decisions based on smaller-scale searches or prior research. Once a promising set is identified, use these hyperparameters for the full training run. Tooling like MLflow is essential to manage and track these experiments.
- **Maximizing resource utilization:** Given the high costs associated with long-running distributed GPU training jobs, it is hugely important to maximize resource utilization. MosaicML's composer is an example of a library that uses PyTorch FSDP with additional optimizations to maximize Model FLOPs Utilization (MFU) and Hardware FLOPs Utilization (HFU) during training.
- **Handling GPU failures:** Training large models can run for days or even weeks. During large-scale training for this length of time, hardware failures, especially GPU failures, can (and typically do) occur. It is essential to have mechanisms in place to handle such failures gracefully.
- **Monitoring and evaluation:** Close monitoring of the training process is essential. Saving model checkpoints regularly and evaluating on validation sets not only act as safeguards but also provide insights into model performance and convergence trends.

In cases where pretraining an LLM from scratch is required, MosaicML Training provides a platform to conduct training of multibillion parameter models in a highly optimized and automated manner. Automatically handling GPU failures and resuming training without human intervention and MosaicML Streaming for efficient streaming of data into the training process are just some of the capabilities provided out of the box.

The following diagram shows an example of what a production pretraining architecture looks like on Databricks — the full story can be found here.



Model Evaluation

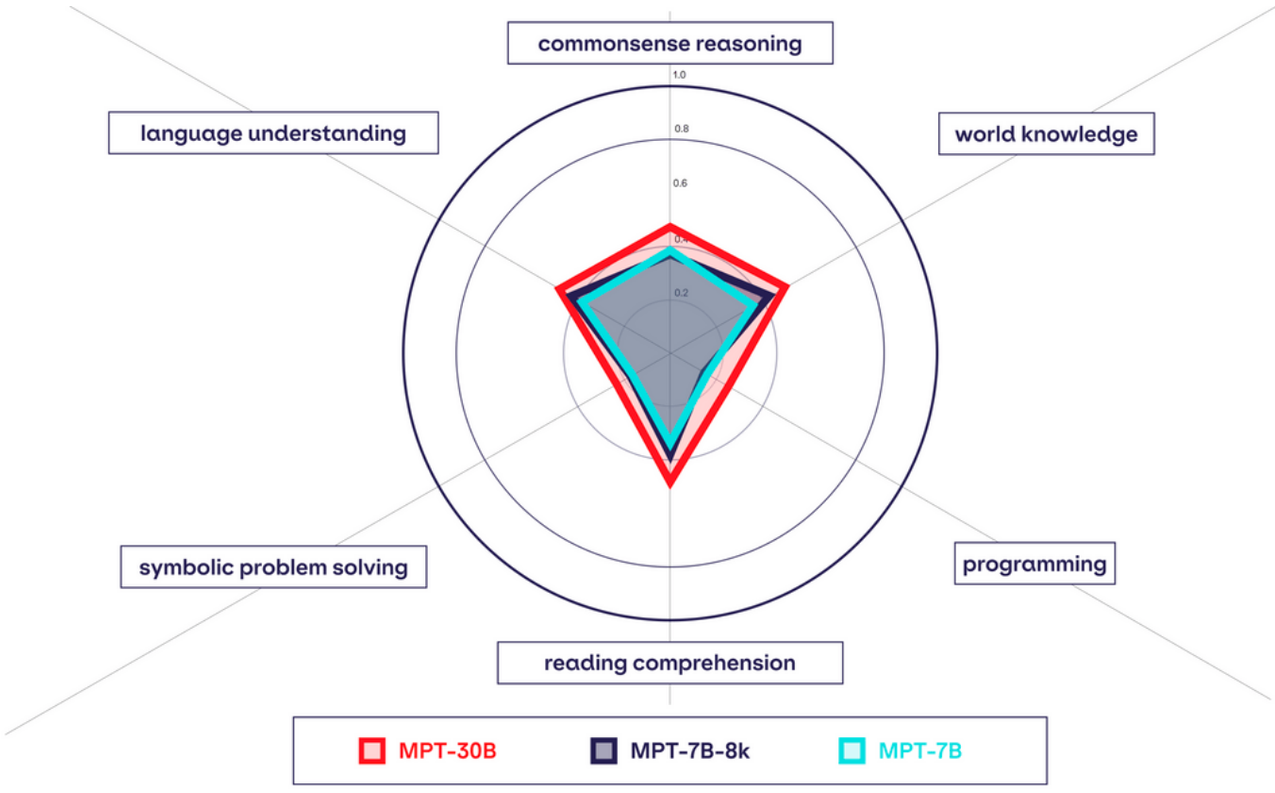
Evaluating LLMs is a challenging and evolving domain, primarily because LLMs often demonstrate uneven capabilities across different tasks. An LLM might excel in one benchmark, but slight variations in the prompt or problem can drastically affect its performance. The dynamic nature of LLMs and their vast potential applications only amplify the challenge of establishing comprehensive evaluation standards.

The following are a number of challenges involved with evaluating LLM-powered applications:

- **Variable performance:** LLMs can be sensitive to prompt variations, demonstrating high proficiency in one task but faltering with slight deviations in prompts.
- **Lack of ground truth:** Since most LLMs output natural language, it is very difficult to evaluate the outputs via traditional NLP metrics (BLEU, ROUGE, etc.). For example, suppose an LLM is used to summarize a news article. Two equally good summaries might have almost completely different words and word orders, so even defining a "ground-truth" label becomes difficult or impossible.
- **Domain-specific evaluation:** For domain-specific fine-tuned LLMs, popular generic benchmarks may not capture their nuanced capabilities. Such models are tailored for specialized tasks, making traditional metrics less relevant. This divergence often necessitates the development of domain-specific benchmarks and evaluation criteria. See the example of Replit's code generation LLM.
- **Reliance on human judgment:** LLM performance is often evaluated in domains where text is scarce or there is a reliance on subject matter expert knowledge. In such scenarios, evaluating LLM output can be costly and time-consuming.

Some prominent benchmarks used to evaluate LLM performance include:

- BIG-bench (Beyond the Imitation Game benchmark)
  - A dynamic benchmarking framework, currently hosting over 200 tasks, with a focus on adapting to future LLM capabilities
- EleutherAI LM Evaluation Harness
  - A holistic framework that assesses models on over 200 tasks, merging evaluations like BIG-bench and MMLU, promoting reproducibility and comparability
- Mosaic Model Gauntlet
  - An aggregated evaluation approach, categorizing model competency into six broad domains (shown below) rather than distilling to a single monolithic metric



Source: Mosaic Model Gauntlet

LLMs as evaluators

A number of alternative approaches have been proposed to use LLMs to assist with model evaluation. These range from using LLMs to generate evaluations to entrusting LLMs as judges to evaluate the capabilities or outputs of other models. Ideally, when evaluating an LLM, a larger or more capable LLM should be employed as the evaluator because a "smarter" model can plausibly produce a more accurate evaluation. The benefits of using LLMs as evaluators include:

- Speed, as they are faster than human evaluators
- Cost-effectiveness
- In certain cases, they offer comparable accuracy to human evaluators

See this Databricks blog post for a more detailed exploration of using LLMs as evaluators.

Human feedback in evaluation

While human feedback is important in many traditional ML applications, it becomes much more important for LLMs. Humans — ideally your end-users — become essential for validating LLM output. While you can pay human labelers to compare or rate model outputs (or have an LLM evaluate your application as mentioned above), the best practice for user-facing applications is to build human feedback into the applications from the outset. For example, a tech support chatbot may have a "click here to chat with a human" option, which provides implicit feedback indicating whether the chatbot's responses were helpful. Explicit feedback can also be captured in this case by presenting users with the ability to click thumbs up/down buttons.

Summary

In an era defined by data-driven decision-making and intelligent automation, the importance of MLOps can't be overstated. MLOps provides the essential scaffolding for developing, deploying and maintaining AI models at scale, ensuring they remain accurate and continue to deliver business value. The emergence of LLMOps highlights the rapid advancement and specialized needs of the field of generative AI. However, at its heart, LLMOps is still rooted in the foundational principles of MLOps.

Whether you're implementing traditional machine learning solutions or LLM-driven applications, the four core tenets remain constant:

- **Business goal:** Always keep your business goals in mind
- **Data-centric:** Prioritize a data-centric approach
- **Modular:** Implement solutions in a modular manner
- **Automated:** Aim for processes to guide automation

Databricks stands uniquely positioned as a unified, data-centric platform for both MLOps and LLMOps. Serving as the foundation for our Data Intelligence Platform, Unity Catalog provides a single governance solution for all data and AI assets. This is complemented by MLflow for experiment tracking, Model Serving for real-time deployment, Lakehouse Monitoring to ensure long-term efficiency and performance stability and Databricks Workflows to seamlessly orchestrate data pipelines.

GenAI training

- Generative AI engineer learning pathway: Take self-paced, on-demand and instructor-led courses on generative AI
- Free LLM course (edX): An in-depth course to learn GenAI and LLMs inside and out
- GenAI webinar: Learn how to take control of your GenAI app performance, privacy and cost, and drive value with generative AI

Additional resources

- The Big Book of MLOps: A deep dive into the architectures and technologies behind MLOps — including LLMs and GenAI
- Mosaic AI: Product page covering the features of Mosaic AI within Databricks

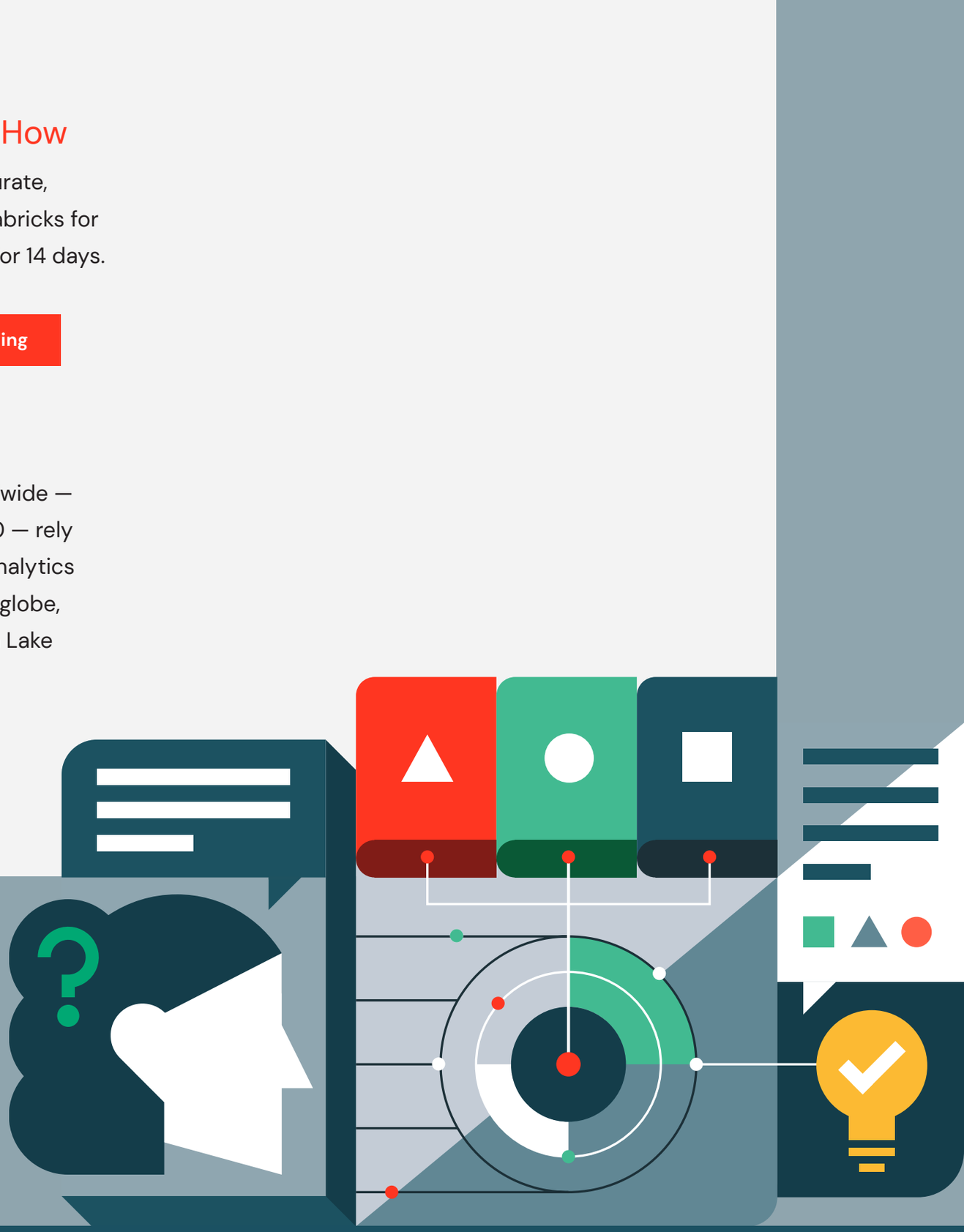
Build Production-Quality GenAI Applications — See How

Create high-quality generative AI applications and ensure your output is accurate, governed and safe. See why over 10,000 organizations worldwide rely on Databricks for all their workloads from BI to AI — test-drive the full Databricks Platform free for 14 days.

Try Databricks free **Take Generative AI Fundamentals On-Demand Training**

About Databricks

Databricks is the data and AI company. More than 10,000 organizations worldwide — including Comcast, Condé Nast, Grammarly and over 50% of the Fortune 500 — rely on the Databricks Data Intelligence Platform to unify and democratize data, analytics and AI. Databricks is headquartered in San Francisco, with offices around the globe, and was founded by the original creators of Lakehouse, Apache Spark™, Delta Lake and MLflow. To learn more, follow Databricks on LinkedIn, X and Facebook.



databricks