MODULE 5 · STUDY NOTES

Working with Foundation Models

Choosing the right FM, inference parameters, RAG, customization techniques, and the Amazon Bedrock and SageMaker AI toolkits

In a traditional application, the backend is a database you configure but did not build; in a generative AI application, the backend is a foundation model — and the craft is choosing and shaping one, not building it. This module works through that craft end to end: the three approaches (reuse, adapt with data, build from scratch), the seven factors that decide which pre-trained FM fits a use case, how to evaluate the choice and keep monitoring it in production, the inference parameters that steer creativity versus determinism, RAG for grounding responses in your own current data, the customization techniques that genuinely retrain the model, and where Amazon Bedrock and Amazon SageMaker AI fit into all of it.

LEARNING OBJECTIVES

- Discuss the factors to consider when selecting and evaluating a foundation model (FM)
- List common inference parameters, and describe their effect on model outputs
- Describe Retrieval-Augmented Generation (RAG)
- Distinguish between FM customization techniques, including fine-tuning and continuous pre-training
- Recognize key features of Amazon Bedrock and Amazon SageMaker AI for working with FMs

5.1 Considerations for choosing and adapting a foundation model

When you build a traditional software application, the backend is typically a data store such as a relational database — and you choose an existing database engine rather than writing your own. In a generative AI application the main backend is an **FM**, and the same logic applies: choosing a **pre-trained** FM lets you implement generative AI quickly without investing in developing your own model, so the work shifts to tailoring its outputs to your business problem. The options group into three approaches: **reuse** an existing model (use prompts), **adapt** an FM with additional data (reference a knowledge base or customize it), or **build from scratch** (build your own).

OPTIMIZING A PRE-TRAINED FM – THE MAIN TASKS

- **Choose a suitable model** — consider its capabilities, assess training data quality and diversity, and evaluate the cost / speed / size / complexity trade-offs.
- **Adjust inference parameters** — experiment with randomness and diversity; configure maximum length or stop sequences to constrain the size of results.
- **Optimize the prompt** — iterate and refine; for some use cases these first three tasks (the *reuse* approach) are all that is needed.
- **Incorporate custom data** — integrate external data to refine the input (RAG), or perform additional training using a relevant dataset (customization).
- **Evaluate performance across every step** — and after deployment, continue to monitor and assess the application in production.

USE A PRE-TRAINED FM	BUILD YOUR OWN FM
the default for most use cases ▪ Fastest path to generative AI capability ▪ No investment in developing your own model ▪ Work centers on tailoring outputs: model choice, inference parameters, prompts, custom data ▪ Evaluate, deploy, then keep monitoring	*the exception — and a serious commitment* ▪ Need for a high degree of customization or highly specialized domain knowledge ▪ Opportunity for competitive advantage ▪ Requirement for complete control; data privacy and security concerns ▪ Requires deep technical skills and resources for the full ML lifecycle

5.2 Choosing the right foundation model for a use case

The goal is the model that best suits the particular use case — and "best" depends on which factors matter most. A healthcare company launching a patient question-and-answer application might prioritize higher complexity and accuracy and the lowest possible risk of ethical concerns; a start-up launching an internal chat application might prioritize keeping costs low and ease of implementation. Seven consideration categories frame the decision.

TABLE 5.1 Considerations for choosing a pre-trained FM

FACTOR	THE QUESTION	WHAT SITS BEHIND IT
Capabilities	What does the model need to do?	Task specialization, multimodal support, language support, context window, customization flexibility
Size & complexity	How complex are the tasks, and how critical is accuracy?	Parameter count drives capability — and cost
Inference speed	How quickly must the model respond?	Latency plus throughput; smaller models respond faster
Training data	What can you find out about it?	Quality, volume, diversity, relevance, recency, potential bias
Ethical practices	Is there a risk of bias or other ethical concerns?	Intended use and harm, privacy, transparency, regulation, values
Costs	What does it cost to use the model?	Licensing, pricing model, compute, customization, scaling, operations
Implementation	What is the effort to integrate, deploy, and run?	Ease of integration, infrastructure compatibility, scalability

Capabilities are the natural starting point: choose a model designed for your task type — an image-generation model for high-quality images, an LLM for text-based tasks, a multimodal model to process text and images together. For LLMs, dig deeper: does it support the languages you need? How large a **context window** — the number of tokens the model can process, counting both the input prompt and the output it produces — does the application require? A chat application can live with a smaller window; an application that produces reports from documents supplied in the prompt needs a much larger one. If you expect to incorporate your own data, also evaluate the effort the model's **customization** options demand. And if no pre-trained model meets your needs, that is the signal that building your own might be necessary.

Size and complexity. *Parameters* are the internal variables a model learns and adjusts during training to capture patterns and relationships in the data; the parameter count is the standard measure of model size and complexity, and typical FMs carry **billions or even trillions** of parameters. More parameters generally means capacity for more complex understanding and a wider range of tasks — but also higher computational requirements, longer training times, and greater memory and storage demands, which equates to higher cost. Providers ship model families at different sizes for exactly this reason: Anthropic's Claude Sonnet models are larger and more complex than Claude Haiku — both powerful LLMs suited to different use cases. Choose a size that handles the complexity you need without paying for resources you don't.

LARGER MODELS SUIT	SMALLER MODELS SUIT
capability first ▪ Diverse, multimodal tasks (text, image, and audio) ▪ Generating long-form, coherent content ▪ Complex reasoning and problem-solving tasks	*speed and footprint first* ▪ Specific, narrow-domain tasks with limited scope ▪ Deployment on mobile devices ▪ Real-time applications that require quick responses

Inference speed. *Latency* is the time between the request to and the response from an FM; *inference speed* encompasses latency plus **throughput** — how many inferences can be processed in a given time frame. Faster is generally better, and smaller models are generally faster; if you need a larger model's capabilities, expect to trade away some speed. Judge against the real requirement: a real-time chat application that thousands of customers use in parallel needs far lower latency and better throughput than an application that generates periodic reports for employees. Hardware also has significant impact — GPU-based chips and accelerators are common, and AWS designs purpose-built chips for ML training (**AWS Trainium**) and inference (**AWS Inferentia**), offered through EC2 instance families such as **Inf1 and Inf2** that are built for high-performing AI inference.

> **WHEN RESPONSES ARE TOO SLOW**
>
> - **Batch processing** — perform inference offline in batches of requests, for applications that don't require real-time responses.
> - **Caching** — store previous results so a repeated input returns the cached answer instead of recomputing from scratch.
> - **Service features** — for example, Amazon Bedrock **Provisioned Throughput** for applications that need greater throughput, and SageMaker AI's inference-optimization techniques.

Training data. With a pre-trained FM you have **no control over the training data**, and details of the datasets used are typically not available — so use the model documentation to learn what you can (Amazon's models, for example, publish **AI service cards**). Assess six dimensions: **quality** (a model trained on high-quality curated data generates more accurate results — and usually costs more than one trained on raw web crawls); **volume** (a massive training dataset means more parameters and more coherent, contextually appropriate output — at the price of size and speed); **diversity** (a model trained on academic papers, books, websites, and social media handles a wider range of tasks than one trained only on news articles); **relevance** (a model pre-trained on biomedical literature and clinical data produces better medical answers without customization); **recency** (a fast-moving domain such as generative AI punishes years-old training data, while a historical-information application tolerates it); and **bias** (prefer models whose builders applied rigorous bias-mitigation processes during training).

Ethical practices. Responsible AI starts with model choice. Weigh the intended use and potential for harm — a model trained on a broad, less-curated corpus is versatile but likelier to generate biased or inappropriate content, while a

curated dataset is less prone to harmful outputs. Consider fairness and bias; privacy and data rights (a model trained on proprietary data can carry stricter privacy obligations than one trained on public data); transparency and accountability (clear documentation and oversight beat an opaque-box model, especially for high-stakes applications); regulatory compliance (a GDPR-bound business needs a GDPR-compliant model); and alignment with organizational values — down to preferring a model trained with sustainable, energy-efficient methods over one with a high carbon footprint.

TABLE 5.2 Cost factors for a production FM

COST FACTOR	WHAT TO WEIGH
Licensing fees	Open source models carry no license fee and lower upfront cost; licensed models may offer more features and support
Pricing model	Some charge a flat fee per API call; others charge on **input and output tokens** — managing prompt and output sizes is cost management
Size & compute	A lightweight model can run on a standard laptop; a complex model may demand a powerful GPU
Customization expenses	Understand the pricing structure for customizing before committing to it
Scalability & infrastructure	Cloud pay-as-you-go favors experimentation and rapid scaling; on-premises may be required in a regulated environment
Operations & maintenance	Deploy on your own hardware, on cloud resources via SageMaker AI, or hand resource management to Bedrock / Amazon Q; some models need frequent updates, others follow stable schedules

Implementation. Finally, ask how hard the model is to put into production: **ease of integration** with your existing systems and tools (plug-and-play integration minimizes infrastructure change, and moving to a newer model version should cause minimal disruption); **infrastructure compatibility** (can you obtain the computational power to run inference at the pace production requires?); and **scalability** (if your chat application must stay fast through a post-launch usage surge, can the infrastructure supporting the selected model grow to handle that throughput?).

> COMMON PITFALL
>
> "Pick the biggest model" is wrong on the exam and in production. More parameters buy complexity handling, but they cost more and respond more slowly — smaller models are the right call for narrow scope, mobile deployment, and real-time latency budgets. The factors interlock: size drives complexity, speed, and cost together.

5.3 Evaluation and monitoring

Choosing the most suitable FM calls for a structured evaluation process with clearly documented outcomes — and evaluation can occur across every phase of selection and refinement, not just at the start.

> THE EVALUATION PROCESS
>
> - **Define metrics** — clear evaluation metrics aligned with the use-case requirements; also assess deployment requirements such as infrastructure, latency, and scalability needs.
> - **Prepare test datasets** — representative data that reflects real-world scenarios.
> - **Test and compare** — comparative testing of multiple models, using consistent methodologies.
> - **Iterate and refine** — act on results, and consider further options such as incorporating relevant data.

Evaluation plans differ by use case, but **human evaluation is almost always part of the process**, complemented by automated evaluation jobs that capture quantitative metrics. Both Amazon Bedrock and Amazon SageMaker AI provide features supporting automated *and* human-based evaluation of LLMs. You don't need to memorize an exhaustive metric list — the useful skill is recognizing common language-oriented metrics and the primary scenario each one suits. One worth knowing by name: **BERTScore**, which uses BERT (*bidirectional encoder representations from transformers*) to generate contextual embeddings and measure the **semantic similarity** between two pieces of text.

AFTER DEPLOYMENT – ONGOING MONITORING AND ASSESSMENT

- Monitor model performance in production to identify areas for improvement.
- Collect and analyze user feedback to understand real-world effectiveness.
- Stay informed about new model releases that might offer improved capabilities.
- Periodically reassess the selected model as the use case and requirements evolve.
- Consider **multi-model approaches** that use different models' strengths for different subtasks.

COMMON PITFALL

Selecting a foundation model is **not a one-time decision**. New models arrive rapidly and requirements drift — treat selection as an ongoing process of monitoring, feedback analysis, and periodic reassessment so the application keeps delivering optimal performance and value.

5.4 Configuring inference parameters

An FM's output is always a **prediction** — an LLM writes its response by continually predicting the next most probable word or sequence of words. Inference parameters are the knobs that set how broadly or narrowly the model considers candidates for that next token. Think of a cat that will *probably* jump onto the couch but *could* jump onto a bowling ball: you are tuning whether the model thinks deterministically (couch) or creatively (bowling ball). Different models don't all expose the same parameters, but three are the most common — and in every case, **lower values yield more deterministic output and higher values yield more creative output**.

TABLE 5.3 Randomness and diversity parameters

PARAMETER	WHAT IT DOES	RANGE	TYPICAL SETTINGS
Temperature	Controls the randomness of the output — higher ⇒ more diverse and creative; lower ⇒ more deterministic and focused	0–1	0.2 for precise, coherent responses (technical documentation, factual queries) · 0.8 for brainstorming or creative writing
Top P	Nucleus sampling — controls the cumulative probability of tokens included as top candidates for the next token; P is a percentage of tokens, and higher values open the pool to lower-probability candidates	0–1	0.1 for highly relevant, focused answers (specific Q&A) · 0.9 for varied output (story or product ideas)
Top K	Restricts sampling to the K most probable next tokens; K is a specific number of tokens	1–K	10 for precise, focused responses · 50 for creative content or brainstorming

Two further parameters cap the output rather than shape its character. **Max length** is the maximum number of tokens the model can generate in a single response: set it around 50 for a chat application that should answer “What's the weather today?” with a brief forecast, or around 200 to allow an extensive-but-concise summary of a long article. **Stop**

sequences are specific tokens or strings that signal the model to stop generating further tokens; you can define one or several, and they can be words, phrases, punctuation marks, or special characters. They are the tool of choice when output must stay within a specific boundary or format — a single sentence, one paragraph, or exactly one answer to a question. The optimal values for all of these depend on the task; experiment and observe.

> **COMMON PITFALL**
>
> Top P versus top K: both widen or narrow the candidate pool, but top P is a cumulative **probability threshold** (range 0–1) while top K is an **actual number of tokens** to consider (starting at 1). Exam distractors swap the two — check whether the value named is a fraction or a count.

The module's hands-on activity uses **PartyRock**, an Amazon Bedrock playground, to simulate temperature and top P: adjust the sliders, try different prompts, and compare how the character of the output shifts.

> **SAMPLE EXAM QUESTION, WORKED**
>
> *A data scientist is configuring an LLM for a creative writing task. Which inference parameter would be MOST effective in generating diverse and unexpected outputs?* Key phrases: *creative writing task*, *inference parameter*, *diverse and unexpected outputs*. Decreasing **top P** would limit diversity by shrinking the pool of tokens considered — the wrong direction. Reducing **top K** likewise narrows the candidates and decreases diversity. Expanding the **context window** gives the model more information to work with but doesn't directly increase creativity. **Increasing temperature** introduces more randomness into token selection, producing more diverse and creative output — the answer.

5.5 Retrieval-Augmented Generation (RAG)

RAG is the process of referencing an **authoritative knowledge base outside the LLM** before generating a response for a prompt — combining the precision of retrieval systems (think of a search engine) with the power of an LLM. The name is the recipe: **retrieve** information relevant to the user's input, **augment** the prompt with it as additional context, then **generate**. It is like giving the LLM a research assistant: the assistant takes the question, finds the relevant documentation, and the model answers using those resources plus its own knowledge. Your own data flows into the results without manually pasting context into every prompt and **without customizing the model itself** — and because the knowledge base can hold newer data than the model was trained on, the application stays up to date without retraining. RAG is less expensive than retraining, with one trade-off: the augmentation step means each request takes a bit longer — a **latency** cost.

> **INSIDE A RAG SOLUTION**
>
> - **Vector database** — stores *embeddings* created from the data source you want to reference.
> - **Orchestrator** — the software component managing the flow of information between the initial input, the vector database, and the LLM.
> - Request path: the user's input is *not* sent straight to the LLM — the orchestrator searches the vector store for relevant data, augments the prompt with the retrieved context, and sends the augmented prompt to the LLM.
> - Both components require specialized technologies and skills to build yourself.

BENEFITS OF RAG

- **Relevancy and accuracy** — grounding responses in external, domain-relevant factual information provides more relevant, up-to-date answers and reduces the tendency to hallucinate.
- **Adaptability** — the model accesses the most current information without requiring constant retraining.
- **Transparency** — the vector database is a controlled source, so organizations gain control over generated output and users gain insight into how the LLM produced the response.

TABLE 5.4 Example applications of RAG

SCENARIO	HOW RAG HELPS
Customer support	A chat application references the latest product manuals — customers get quick access to the most relevant, up-to-date troubleshooting information
Knowledge management	A large, widely distributed company exposes IT policies and resources through an internal Q&A application — employees quickly find the policies relevant to their workplace
Decision support	A healthcare company references the latest research on evidence-based care — providers use a Q&A application to inform patient-care decisions and offer personalized treatment suggestions

AMAZON BEDROCK KNOWLEDGE BASES	AMAZON Q BUSINESS
managed RAG — you configure, AWS integrates ▪ Fully managed RAG workflow; no custom integrations to build ▪ Retrieves from structured or unstructured sources; stores unstructured data as vector embeddings ▪ Your choice of parsing options, embedding models, and vector stores ▪ Provides citations for responses ▪ "Chat with your document": test responses against a sample data source with no vector store setup	*interactive chat on your data — no coding* ▪ Fully managed generative AI assistant, powered by Amazon Bedrock and inheriting its safety, security, and responsible-AI controls ▪ Answers from your enterprise data alone, or your data combined with LLM knowledge; supports a variety of LLM tasks ▪ Integrated access control: responses draw only on content the end user has permission to access, managed via familiar AWS identity services ▪ Cites source documents with every answer; administrative controls and guardrails protect sensitive information and enforce organizational policy

COMMON PITFALL

RAG never changes the model — it changes the **prompt**. When a question hinges on improving responses with current company data *without retraining or customizing the model*, RAG is the answer; when the model itself must learn, that is customization.

5.6 Foundation model customization techniques

A RAG solution improves performance by feeding the model better context, but it is not always the optimal way to adapt an FM. In some cases you get better results by actually **training the FM a bit more** on a dataset meaningful to

your use case. Three techniques cover the ground, differing in their goals, the type of data they use, and their relative cost and effort.

TABLE 5.5 Customization techniques – goal and data

TECHNIQUE	GOAL	DATA IT USES
Fine-tuning	Perform better on the target task while retaining the general knowledge from the original pre-training	A relatively small **labeled** dataset relevant to the task or domain
Instruction-based fine-tuning	Improve the model's ability to respond the way you want — more controllable, predictable outputs	Labeled examples of sample **instructions and expected outputs** — like a more extensive version of few-shot prompting
Continued pre-training	Expand the model's knowledge base or adapt it to a new domain **without changing its fundamental capabilities**	A large amount of new, high-quality data — the **same types of data and techniques** as the original training

TABLE 5.6 Customization techniques – pros and cons

TECHNIQUE	PROS	CONS
Fine-tuning	Improves task performance with relatively less data and cost than training from scratch	A quality dataset can be difficult to obtain; might not perform as well on tasks outside the target
Instruction-based fine-tuning	Enhances the model's ability to follow instructions; more predictable output	Requires careful curation of instruction–output pairs; could limit the model's creativity on open-ended tasks
Continued pre-training	Develops new expertise without changing capabilities; keeps the model up to date	Requires a large amount of new, high-quality data; computationally expensive and time-consuming

> MATCHING THE TECHNIQUE TO THE SCENARIO
>
> **Sentiment analysis on movie reviews** → fine-tune an LLM with a dataset of labeled reviews, each annotated with a sentiment score (positive, negative, or neutral). **Product summaries in a house style** → instruction-based fine-tuning, where each dataset row holds an instruction, the text to summarize, and an output representing the ideal answer. **A pharmaceutical company deepening a model's understanding of recent medical research and drug development** → continued pre-training on a large body of recent medical literature and research papers.

A related term worth knowing: **transfer learning** describes fine-tuning a model for a new *related* task. A machine learning model that identifies images of dogs can be trained to identify cats using a smaller image set that highlights the feature differences between dogs and cats.

THE STUDENT ANALOGY

Think of the model as the brain of a student facing a test. **RAG** lets the student write a quick-reference guide and bring it into the exam — the least effort and cost, achievable without deep in-house AI/ML expertise, though finding each answer takes a moment. **Fine-tuning** is a crash course on the topic: more effort, time, cost, and in-house expertise than RAG, but the student now answers off the top of their head — faster — while struggling if questions move outside the topic studied. **Continued pre-training** is an advanced degree in a related field: the greatest effort, expertise, and cost of all, rewarded with the deepest ability to answer in-depth questions across the broader field.

COMMON PITFALL

Keep the two "more training" options straight: **fine-tuning** teaches the model to do a *task* better from a small labeled dataset; **continued pre-training** grows the model's *knowledge* from a large dataset using the original training techniques, leaving its fundamental capabilities unchanged.

5.7 Working with foundation models on AWS

Every use case sits somewhere on a spectrum. **Prompt engineering** is generally the least complex and least costly approach to getting tailored outputs; **building your own FM** anchors the opposite end of the complexity and cost scale. At each step, weigh the potential business value of bringing more data into the mix against the added cost and effort.

TABLE 5.7 Approach → AWS service support

APPROACH	SUPPORTED BY
Prompt engineering	Amazon Q, Amazon Bedrock, Amazon SageMaker AI
RAG	Amazon Q, Amazon Bedrock, Amazon SageMaker AI
Fine-tuning / continued pre-training	Amazon Bedrock, Amazon SageMaker AI
Build your own FM	Amazon SageMaker AI

AMAZON BEDROCK

the on-ramp for generative AI development

- Access to many pre-trained FMs; quickly experiment and compare results across models and inference parameters
- Run evaluation jobs and review results in the console
- Knowledge Bases implement RAG without integration work; create customized versions of models in your own account
- **Unified API** — straightforward integration and easy model swapping; integrates with many familiar AWS services, lowering the learning curve

AMAZON SAGEMAKER AI

the full ML lifecycle

- FM access through **SageMaker JumpStart** — deploy JumpStart models, or build and deploy your own FMs
- **SageMaker Clarify** supports automated and human-based evaluation jobs; **Ground Truth** supports human-in-the-loop review of model responses
- Build your own RAG solution from example notebooks; tools for fine-tuning and additional training of FMs
- Integrated tools and an IDE spanning the entire ML lifecycle, on AWS Cloud infrastructure provisioned on your behalf

Both services provide fully managed infrastructure and integrated tools for building generative AI applications. The module closes with a recorded demo of the Amazon Bedrock console — browsing and comparing models and working with the very parameters and features this module covered.

Module summary

- Three approaches to FMs: reuse an existing model (prompts), adapt it with data (RAG or customization), or build from scratch — complexity and cost rise along that spectrum.
- Choose a pre-trained FM on seven factors — capabilities, size & complexity, inference speed, training data, ethical practices, costs, implementation — weighted by the use case.
- Parameters are the learned internal variables that measure model size: more parameters means more capability but higher cost and slower inference; match size to the job.
- Evaluate with defined metrics and representative test datasets, comparing models consistently; human evaluation almost always joins automated jobs — and selection is never final: monitor, gather feedback, reassess.
- Temperature, top P, and top K tune randomness — lower = deterministic, higher = creative; max length and stop sequences keep output from running long.
- RAG retrieves from a vector database of embeddings via an orchestrator and augments the prompt: more relevant, current, transparent answers with less hallucination, no model change — at a small latency cost.
- Customization is additional training: fine-tuning (small labeled dataset, target task), instruction-based fine-tuning (instruction–output pairs, desired responses), continued pre-training (large dataset, original techniques, deeper knowledge).
- Amazon Bedrock (many FMs, unified API, Knowledge Bases, evaluation jobs, customization) and Amazon SageMaker AI (JumpStart, Clarify, Ground Truth, full ML lifecycle, build-your-own) are the AWS homes for FM work; Amazon Q Business delivers RAG-style chat with no coding.

Review questions

1. Why do most teams choose a pre-trained FM over building their own — and what tips the balance the other way?

Answer: A pre-trained FM delivers generative AI quickly with no investment in model development; the work becomes tailoring outputs. Building your own wins when you need a high degree of customization, highly specialized domain knowledge, competitive advantage, complete control, or strong data privacy/security — and you have the deep technical skills and resources for the full ML lifecycle.

2. A chat app that gives short answers and a report generator that digests long documents differ most on which capability consideration?

Answer: The context window — the number of tokens the model can process across input and output. The report application needs a much larger window to take whole documents in the prompt; the chat app can use a smaller one.

3. Your real-time chat application is responding too slowly at scale. What are your options?

Answer: Caching previous results; service features such as Amazon Bedrock Provisioned Throughput or SageMaker AI inference optimizations; inference-optimized hardware (AWS Inferentia-based Inf1/Inf2 instances); or a smaller, faster model. Batch processing helps only when real-time response isn't required.

4. You can't see a pre-trained model's training data. What do you do, and what six dimensions do you judge?

Answer: Study the model documentation (for example, Amazon's AI service cards) to learn what you can, then weigh quality, volume, diversity, relevance, recency, and potential bias.

5. **You need precise, factual, coherent answers. How do you set temperature and top P, and why?**

   Answer: Low — for example temperature 0.2 and top P 0.1. Low values restrict token selection to the most probable candidates, making output more deterministic and focused; high values open the pool and add creativity.

6. **Walk a user's question through a RAG solution.**

   Answer: The input is not sent straight to the LLM. The orchestrator searches the vector database (embeddings built from your data source) for relevant data, augments the prompt with the retrieved context, and sends the augmented prompt to the LLM — which generates a more relevant, grounded response.

7. **A pharmaceutical company wants its model to master recent medical research without changing what the model can do. Which technique, and why not the others?**

   Answer: Continued pre-training on a large body of recent medical literature — it expands the knowledge base without changing fundamental capabilities. Fine-tuning targets a specific task with a small labeled dataset; RAG would reference the literature at query time but not deepen the model itself.

8. **Which evaluation features do Amazon Bedrock and Amazon SageMaker AI offer for comparing LLMs?**

   Answer: Both support automated and human-based evaluation jobs. Bedrock runs evaluation jobs with results reviewable in its console; SageMaker Clarify supports automated and human-based evaluation, and SageMaker Ground Truth supports human-in-the-loop review of responses.