

Chapter 1: Introduction to Building AI Applications with Foundation Models

The exponential scaling and accessibility of **Foundation Models** have transformed AI from a specialized field into a ubiquitous development tool, rapidly accelerating the demand for **AI Engineering**—the discipline of building applications atop these readily available models.

Key Technical Concepts

Concept/Principle	Tradeoff or Decision Explained
Foundation Model (FM)	FMs (including LLMs and LMMs) shift AI development from task-specific modeling to general-purpose adaptation. The decision moves from building a model from scratch to adapting a powerful existing model using techniques like prompt engineering or fine-tuning, reducing time-to-market.
Foundation Model Training Scale	Scaling (increasing parameters, data, and compute) provides improved capabilities, but limits model development to a few organizations. Tradeoff: Access to advanced capabilities (via scale) necessitates reliance on Model-as-a-Service APIs versus independent self-hosting.
Self-Supervision vs. Supervision	Self-supervision (used for pre-training FMs) overcomes the costly and time-consuming bottleneck of obtaining explicitly labeled data (supervision) by inferring labels directly from massive amounts of unlabeled text data (e.g., Common Crawl).
Autoregressive Language Model	Autoregressive models (the choice for generation tasks) predict the next token sequentially using only preceding tokens, enabling continuous text generation. Tradeoff: This contrasts with Masked Language Models (e.g., BERT), which fill in blanks using context from both sides, typically used for non-generative tasks like sentiment analysis.
Tokenization Unit (Token)	Tokens (characters, words, or word parts) are used instead of whole words or characters. Decision: Using tokens balances vocabulary size (fewer tokens lead to more efficiency) while retaining semantic structure and handling unknown words better than relying solely on characters.
Model Adaptation (Build vs. Buy)	The process of customizing an existing FM for a specific task. Decision: Prompt-based techniques (e.g., Prompt Engineering, RAG) are easier,

	faster, and require less data (faster iteration) but may not achieve strict performance targets. Finetuning requires updating model weights and more resources (data, compute, ML expertise) but yields higher task-specific quality.
--	---

AI Engineering vs. ML Engineering

Category	Traditional ML Engineering	AI Engineering (with FMs)
Modeling and Training	ML knowledge required for training models from scratch.	ML knowledge is a nice-to-have, not a must-have.
Dataset Engineering	Focuses more on feature engineering with tabular data.	Focuses more on deduplication, tokenization, context retrieval, and quality control.
Inference Optimization	Important.	Even more important (due to model scale/cost/latency).
AI Interface	Less important (models often embedded).	Important (new standalone apps, plugins, chatbots).
Prompt Engineering	Not applicable.	Important (for model adaptation).
Evaluation	Important (easier, mostly close-ended).	More important (harder, mostly open-ended)

Scenarios & Real-World Examples

Context	Solution/Outcome
Code Generation	GitHub Copilot → Generated <i>100million</i> in annual recurring revenue within two years, demonstrating productivity gains.
Multilingual Performance	English dominates Common Crawl (45.88%), while low-resource languages like Telugu, Marathi, and Punjabi are severely under-represented. → General-purpose models (like GPT-4) exhibit underperformance on these low-resource languages in benchmarks (e.g., math problems and MMLU).

Context	Solution/Outcome
Creative Content	Tools like Midjourney (image generation) and Runway/Sora (video) leverage the probabilistic nature of AI. → Generated <i>200million</i> in annual recurring revenue for Midjourney within 1.5 years of launch.
Customer Support/Process	Companies face high risks associated with external-facing, high-impact applications. → Organizations prioritize internal-facing applications (e.g., internal knowledge management) to minimize risk while developing AI engineering expertise.
Writing & Productivity	College-educated professionals were assigned occupation-specific writing tasks and exposed to ChatGPT. → Average time to complete tasks decreased by 40%, and output quality rose by 18%.
Foundation Model Prompt: "To be or not to be". → Completion: ", that is the question," Output	turning the Language Model into a completion machine.
Risk Management	Using an AI model to write emails, respond to customers, or write contracts. → Automates or partially automates tasks requiring communication, leading to increased productivity and economic value.

Best Practices & Pitfalls

Category	Advice/Warning
Application Planning	Best Practice: Before starting, understand why the application is being built (risk response, opportunity, R&D) and how success will be measured (Evaluation-Driven Development).
Model Selection	Pitfall: Do not arbitrarily start with an enormous model size. Start with a compute budget and use the Chinchilla scaling law (or variants) to determine the optimal model size and dataset size to maximize performance for that cost.
Development Focus	Best Practice: Focus on timeless foundational knowledge (e.g., systematic experimentation, rigorous evaluation) rather than fleeting tools or current trends.
System Reliability	Best Practice: Plan for the last mile challenge; building a quick demo is easy, but reaching the final stages of reliability (e.g., improving from 95% to 100% accuracy) requires significant, discouraging effort.

Model Adaptation	Best Practice: If the task involves adapting an FM, prioritize Prompt Engineering and context construction before resorting to more resource-intensive Finetuning.
Inference/Scale	Pitfall: Building inference services requires deep knowledge of handling big models (e.g., >1,000 GPUs, \$100B+ investments) and managing cost/latency tradeoffs. Decision: For many, using Model-as-a-Service APIs is the lower-barrier entry point.
AI System Definition	Pitfall: Assuming familiarity due to similar principles (e.g., RAG being similar to search) can be misleading; the sheer scale and capabilities of FMs introduce new challenges requiring new solutions (AI Engineering).

Key takeaways

- Foundation Models (FMs): FMs scale language models (LLMs) to handle multimodal data (LMMs), shifting AI development toward general-purpose, reusable models.
- Rise of AI Engineering: Increased scale lowered the barrier to entry (Model-as-a-Service) while vastly increasing demand and potential use cases, establishing AI Engineering as a distinct discipline focused on application building.
- Core AI Concepts: AI relies on tokens for language encoding and self-supervision (training on unlabeled data) to achieve massive scale.
- Model Adaptation (The Engineer's Role): The modern development process focuses on adapting existing FMs, primarily through Prompt Engineering (instruction/context) and Finetuning (weight changes), rather than training models from scratch.
- AI Engineering Stack: The focus of AI Engineering compared to traditional ML Engineering shifts strongly toward Application Development (interfaces, prompting, evaluation) and Inference Optimization (cost/latency) due to the size of FMs.
- Planning and Risk: Successful deployment requires Evaluation-Driven Development to define measurable criteria and realistic expectations, acknowledging severe challenges like achieving the "last mile" of quality.

What is self-Supervised?

Self-supervision is the process where a model infers labels directly from the input data itself, rather than requiring expensive and slow external human-generated labels (supervision).

Language modeling is inherently self-supervised because each input sequence provides both the context and the labels needed for training. For example, in a text sequence like "I love street food," the model generates six training samples. The model uses the preceding tokens (the context) to predict the next token (the label). For the input context "<BOS>, I, love, street, food," the output token (label) the model aims to predict is ".".

In **self-supervised** learning, labels are inferred from the data itself, whereas in unsupervised learning, no labels are needed at all. Because text sequences needed for training are abundantly available everywhere (in books, articles, blog posts, etc.), self-supervision allows developers to create massive amounts of training data. This capability is what allowed language models to scale up and become Large Language Models (LLMs).

Self-supervision also works for models that handle more than just text, such as multimodal models.

Chapter 2: Understanding Foundation Models

A comprehensive understanding of the design decisions—including training data, model architecture and scale, post-training alignment, and output sampling—is critical for effectively adapting foundation models for downstream applications.

Key Technical Concepts

Concept/Principle	Tradeoff or Decision Explained
Training Data Quality vs. Volume	High quality is essential, as a small amount of carefully curated data can outperform large amounts of noisy data (e.g., web scrapes),. Tradeoff: Developers must balance acquiring massive volumes (e.g., Common Crawl) with ensuring quality and compliance.
Multilingual Data Distribution	English dominates datasets like Common Crawl (45.88%), leading to low performance in low-resource languages (e.g., Telugu, Marathi, Punjabi). Decision: Choose between reliance on large general models (which tokenise low-resource languages inefficiently, increasing cost/latency) versus developing specialized multilingual models.
Transformer Architecture (Attention)	The dominant architecture utilizes the attention mechanism , Tradeoff: While efficient for parallel input processing (Prefill), the sequential output generation (Decode) creates a computational bottleneck, leading to speed and memory constraints.
Model Scale (Parameters, Tokens, FLOPs)	Scale is measured by Parameters (learning capacity), Training Tokens (volume learned), and FLOPs (training cost). Tradeoff: Increasing scale improves capabilities but drastically increases FLOPs (cost) and limits model development to a few resource-rich organizations,.
Scaling Laws (Chinchilla)	This law calculates the optimal resource allocation given a compute budget. Decision: For compute-optimal efficiency, the number of Training Tokens should be approximately 20 times the number of parameters to maximise performance for a given cost.

Dense vs. Sparse Models (MoE)	Sparse Models like Mixture-of-Experts (MoE) (e.g., Mixtral 8x7B) use a large total parameter count (46.7B) but activate only a subset (experts , e.g., 12.9B active) for processing each token. Tradeoff: MoE balances high potential capability (large parameter count) with faster, cheaper inference speed (fewer active parameters) compared to fully Dense models .
Supervised Finetuning (SFT)	Uses costly, high-quality demonstration data . Tradeoff: It converts a base model optimized for rogue text completion into one suitable for conversation and instruction following, addressing the failure of raw pre-training for dialogue.
Preference Finetuning (RLHF/DPO)	Refines the model using comparison data (winning vs. losing responses) to align output with complex human preferences (e.g., safety, tone). Decision: This phase is necessary because SFT alone does not guarantee alignment or safety, addressing the difficulty of capturing nuanced human preference in instruction sets.
Quantization (e.g., FP32 → FP16)	Reducing numerical precision to decrease the memory footprint. Tradeoff: Reduces model memory, making inference cheaper and faster, but introduces small numerical errors that can degrade model quality, especially if reducing too far (e.g., below 8-bit)
Sampling Strategy (Temperature)	Output generation involves sampling from a calculated probability distribution. Tradeoff: Greedy sampling results in predictable/boring output, while adjusting Temperature (probabilistic sampling) introduces desired creativity but risks inconsistency and hallucination .

Overall Training Workflow (Pre-training → Alignment):

Randomly Initialized Model → **Self-Supervised Pre-training** (Objective: Predict Next Token), → Pre-Trained Model (Completion-Focused), → **Supervised Finetuning (SFT)** (Input: Demonstration Data), → SFT Model (Conversational), → **Preference Finetuning** (Input: Comparison Data, Objective: Maximise Human Preference), → Final Aligned Foundation Model.

Output Generation Flow (Sampling Fundamentals):

Input Prompt → Language Model → **Logit Vector** (Raw scores for vocabulary) → **Softmax Layer** (Convert logits to Probability Distribution) → Probabilistic Sampling (using **Temperature**, **Top-k**, **Top-p**) → Next Token → Final Output.

Scenarios & Real-World Examples

Context	Solution/Outcome
Domain-specific scientific research (e.g., protein structures, medical scans)	General web data is insufficient → Use Domain-Specific Models like AlphaFold or Med-PaLM2 trained on specialized, proprietary datasets.
Language efficiency constraints (e.g., Burmese or Hindi input)	Inefficient tokenization by GPT-4 → Inference costs/latency are 4x to 10x higher than English for the same content due to the number of tokens required.,
Optimizing architecture for speed and cost	Use a Mixture-of-Experts (MoE) architecture (e.g., Mixtral 8x7B) → Achieves the speed/cost profile of a smaller model (12.9B active parameters) by only activating a subset of experts per token.
Improving output quality without full finetuning	Base model outputs are inconsistent → Use Test Time Compute (e.g., "Best of N") to generate multiple responses and select the highest-scoring output (based on average logprob or a reward model).
Need for machine-readable outputs (e.g., JSON)	Probabilistic generation risks incorrect formatting → Requires techniques like Constrained Sampling or Finetuning to enforce specific, reliable output structures.,
Handling model inconsistencies	User asks the same query twice → Model returns different answers (e.g., scoring an essay 3/5 then 5/5) demonstrating the inherent probabilistic nature of AI .

Best Practices & Pitfalls

Category	Advice/Warning
Model Selection/Efficiency	Best Practice: Do not arbitrarily start with a large model; apply the Chinchilla Scaling Law principle to optimise the ratio of parameters to training data, maximizing performance efficiency for the budget.,
Alignment/Finetuning	Best Practice: Supervised Finetuning (SFT) uses demonstration data to teach appropriate behaviour; ensure this data is high-quality and collected/labeled by experts capable of critical thinking and domain expertise.
Output Risk	Pitfall: The probabilistic nature of AI means that anything with a non-zero probability in the training data (no matter how wrong or

	far-fetched, including misinformation or toxic content) can be generated , leading to hallucinations and safety risks.,,
Generation Control	Best Practice: When aiming for creativity, use a non-zero Temperature (e.g., 0.7); setting temperature to zero often results in deterministic, predictable, and potentially dull outputs.
Architecture/Cost	Pitfall: Inference Decoding is inherently memory bandwidth-bound , and the large KV Cache size is the primary bottleneck for serving long contexts; failing to optimize this through techniques like Quantization leads to high costs and latency.
Domain Data	Warning: Relying on general web scrapes (like Common Crawl) will lead to underperformance on specialized or low-resource language tasks due to data quality issues and linguistic under-representation.,
Memory Management	Best Practice: When calculating memory needs, account for not just parameters and activation values, but also gradients and optimizer states, which scale with the number of trainable parameters ,.

Key takeaways:

- Model design is dictated by decisions across five areas: **Training Data** (quality, coverage, compliance), **Architecture** (Transformer mechanism), **Scale** (Parameters, Tokens, FLOPs), **Post-Training** (SFT/Preference Finetuning), and **Sampling** (Probabilistic Outputs).
- **Scaling Laws** (Chinchilla) provide the fundamental principle for efficient model training, requiring a calculated balance between model size and data volume to optimize resource allocation.
- The model must undergo post-training refinement, including **Supervised Finetuning** (to enable conversational instruction-following) and **Preference Finetuning** (to align outputs with human utility and safety).
- Foundation model output is fundamentally **probabilistic**, causing inherent **inconsistency** and **hallucination**; engineers manage this using sampling parameters (**Temperature**, **Top-k/Top-p**) and boosting quality with methods like **Test Time Compute**.
- LLM inference is often bottlenecked by sequential **Decoding** and the large memory footprint of the **KV Cache**; this requires using memory-saving techniques like **Quantization** (reducing precision) and potentially utilizing efficient sparse architectures like **MoE**,,,.

Additional Notes

Transformer Architecture

The Transformer architecture is the dominant architecture for language-based foundation models, built upon the attention mechanism. Understanding this architecture is crucial as it determines a model's inherent constraints and capabilities. The Transformer was developed to **address the limitations of earlier systems**, such as the **seq2seq (sequence-to-sequence)** architecture, which relied on **RNNs (recurrent neural networks)**. The original seq2seq model suffered from slow performance due to sequential processing of input tokens and limited output quality, as it only used the final hidden state to generate output. The Transformer solved this by discarding RNNs and leveraging attention.

The **attention mechanism** is central to the Transformer, allowing the model to process input tokens in parallel. This mechanism operates primarily using three types of vectors:

- **Query vector (Q):** Represents the **current state of the decoder** at each generation step, acting as a mechanism for seeking relevant information.
- **Key vector (K):** Represents a **previous token in the sequence (context)**, which is used to determine how relevant that token is to the current query.
- **Value vector (V):** Represents the **actual content or learned value** of a previous token.

The attention mechanism calculates scores by taking the dot product between the query vector (Q) and the key vector (K). A higher score indicates that the model should use more of the corresponding value vector (V) when generating the output.

The **Transformer typically features Multi-headed Attention**, which splits the Q, K, and V vectors into smaller components. This design allows the model to attend to different groups of tokens simultaneously, improving its ability to capture varied relationships within the data.

Transformer Blocks: The core repeating unit of the Transformer design is the Transformer block. Each block generally consists of two key modules:

1. **Attention module:** This module contains **four weight matrices**: the **query, key, value**, and output projection matrices.
2. **MLP module (Multi-Layer Perceptron):** This component includes linear layers separated by non-linear activation functions (such as ReLU or GELU). **These functions allow the linear layers to learn non-linear patterns from the data.**

Before the sequence of Transformer blocks is the Embedding module, which converts input tokens and their positions into embedding vectors. Following the blocks is the Output layer (or model head), which maps the model's final vectors into token probabilities used for generating the output response.

The scale of the resulting model is determined by the dimensions of these components, including the model's dimension (size of the hidden dimension), the number of transformer blocks (layers), the dimension of the feedforward layers, and the vocabulary size.

Inference Process: While the Transformer architecture allows the input processing to be done in parallel, output generation remains sequential. Inference (using the model to generate output) involves two distinct steps,

1. **Prefill:** This is the parallel processing of the input tokens. This phase is responsible for creating the intermediate state needed to generate the first output token, including calculating and storing all initial Key and Value (K and V) vectors. Prefilling is typically compute-bound.
2. **Decode:** This is the sequential generation of tokens, one after the other. This step involves frequently loading large data (such as model weights) into memory, making it memory bandwidth-bound.

Limitations of Transformer Architecture:

A primary constraint of the Transformer architecture stems from the attention mechanism's reliance on storing K and V vectors in the KV cache. Because generating each subsequent token requires computing and storing K and V vectors for all preceding tokens, the size of the KV cache grows proportionally with the length of the sequence. This limits the model's ability to handle extremely long contexts efficiently, motivating the development of alternative architectures like Mamba and Jamba.

NOTE: Inference, in the context of building AI applications, refers to the process of using a model to compute an output for a given input

Model Size:

Model size is a critical factor in understanding foundation models, serving as a primary indicator of their complexity, capacity, and resource demands.

Model size is typically measured by its **number of parameters**. A parameter is a variable within a machine learning model that is adjusted and updated throughout the training process. The size of a model signals its **scale** across three dimensions:

1. **Number of Parameters:** This acts as a proxy for the model's **learning capacity**. Generally, a higher parameter count correlates with a greater capacity to learn desired behaviours. For instance, Llama-13B refers to a model with 13 billion parameters.
2. **Number of Training Tokens:** This represents how much data the model has learned from. The amount of training tokens measures the total number of tokens presented to the model during training. Large Language Models (LLMs) are often trained using trillions of tokens.

3. **Number of FLOPs (Floating Point Operations):** This is a standardized unit measuring the computational effort required for training, serving as a proxy for the **training cost**.

FLOPs is a standardized unit of measurement used to quantify the compute requirement for a specific task in artificial intelligence. It measures the number of floating-point operations performed. The number of FLOPs serves as a proxy for the **training cost** of a model. Knowing the FLOPs budget is crucial for calculating the optimal model size and dataset size needed to achieve the best performance efficiently, as described by the Chinchilla scaling law.

FLOPs (counting floating-point operations) should not be confused with **FLOP/s** (floating-point operations per Second), which measures a machine's peak performance.

Chinchilla scaling law provides a critical rule for determining the optimal resource allocation when training large language models (LLMs). This law was proposed in the paper "Training Compute-Optimal Large Language Models" by DeepMind. The law helps maximize model performance given a fixed resource budget, typically measured in **FLOPs** (floating-point operations), which serves as a proxy for training cost. It addresses the decision of how to optimally balance the **model size** (number of parameters) and the **dataset size** (number of training tokens) for efficiency.

The scaling law allows developers to predict the expected training loss (a measure of model performance) for a given FLOP budget, parameter count, and token count, assuming the training is executed correctly.

Impact on Inference and Efficiency

The number of parameters has a direct impact on the memory required for inference and training:

- **Memory Footprint:** The parameter count helps estimate the GPU memory needed to run the model. For example, a 7-billion-parameter model using 2 bytes per parameter (16 bits) requires at least 14 GB of memory just for its weights.

$$7,000,000,000 \text{ (7-billion-parameter)} \times 2 \text{ bytes} = 14,000,000,000 \text{ bytes (14 GB)}$$

- **Deciding Factor for Inference:** The sheer size of foundation models means that **memory** often becomes a bottleneck for operating them, motivating techniques like **Quantization** to reduce the bytes needed per parameter.

Dense vs. Sparse Models

While parameter count is a standard measure, it can be misleading if the model is sparse:

- **Dense Models:** All parameters are used, maximizing computational requirements.
- **Sparse Models (e.g., Mixture-of-Experts, MoE):** These models may have a large total number of parameters but only activate a **subset of experts** to process each token.

For example, Mixtral 8x7B has 46.7 billion total parameters but only activates 12.9 billion parameters per token. This gives it the cost and speed profile of a smaller, 12.9 billion parameter model while retaining high capability.

Performance vs. Size

Although bigger models are generally assumed to perform better, this is not always guaranteed:

- **Data Sufficiency:** A larger model trained on an insufficient amount of data may perform worse than a smaller model trained efficiently with more data.
- **Scaling Laws:** The **Chinchilla scaling law** provides guidance on how to optimally balance the number of parameters with the number of training tokens to maximize performance given a fixed compute budget. To be **compute-optimal**, the number of training tokens should be approximately **20 times** the number of parameters.
- **Usability Tradeoff:** Model developers sometimes choose smaller models (even if they achieve suboptimal benchmark performance) because they are easier to work with and **cheaper to run inference on**, leading to wider adoption

Post-training

Post-training is a crucial phase in the development of foundation models, occurring after the initial **pre-training** phase. The core purpose of post-training is to **align the model with human preferences** and optimize it for usability, as a raw pre-trained model typically has two issues.

Goals of post-training

1. **Optimise for Conversation:** Pre-training often optimizes the model for **text completion**. Post-training converts the model into one optimized for conversation and instruction-following, addressing the failure of raw pre-training for dialogue.
2. **Alignment and Safety:** It addresses issues like generating biased, toxic, or factually incorrect responses stemming from indiscriminate web data used in pre-training. The goal is to get the model to behave according to human preference.

Post-training is often seen as **unlocking the capabilities** that the pre-trained model already possesses but which are difficult for users to access through prompting alone. It generally consumes a small portion of computational resources (e.g., InstructGPT used only 2% of compute for post-training) compared to the intensive pre-training phase.

The Two Main Steps of Post-Training

Post-training typically involves two sequential steps to refine the model's behaviour and utility:

Supervised Finetuning (SFT)

1. SFT converts the model from a completion machine into a conversational one. This is done by finetuning the model on **high-quality instruction data** known as **demonstration data** (following the format: prompt, response). This process is sometimes referred to as **behavior cloning**.
2. **Data Requirements:** SFT requires expensive, high-quality data. The demonstration data must cover the range of requests the model is expected to handle (e.g., question answering, summarization, translation). For complex tasks, this data demands **critical thinking and domain expertise** from annotators.

The SFT phase teaches the model **appropriate conversational behaviour** (e.g., responding with instructions on *how* to make pizza instead of adding more questions to the prompt).

Preference Finetuning (Alignment)

This step further refines the model to align its outputs with complex **human preferences** (e.g., safety, helpfulness, tone). It is typically done using **reinforcement learning (RL)**.

Common methods include:

Reinforcement Learning from Human Feedback (RLHF): Used by models like GPT-3.5 and Llama 2. RLHF involves training a separate **reward model** using **comparison data** (prompt, winning response, losing response) to score the foundation model's outputs. The foundation model is then optimized to generate responses that maximize this reward score.

Direct Preference Optimization (DPO): Used by Llama 3 to reduce complexity.

This phase is necessary because SFT alone only teaches conversational format; it does not guarantee alignment on controversial issues (e.g., gun control, political views) or ensure safety against generating harmful content.

The overall process results in a model that is both **capable** (from pre-training) and **aligned** (from post-training).

Training Phase	Objective	Outcome
Self-Supervised Pre-training	Predict the next token (maximise internal knowledge)	Base Model (Rogue/Completion-Focused)
Supervised Finetuning (SFT)	Clone behaviour (optimise for instruction-following/conversation)	SFT Model (Conversational)
Preference Finetuning	Maximise human preference score (optimise for safety/utility)	Final Aligned Model

Analogy: If **pre-training** is like having a child read every book in the world to gain vast, raw knowledge, **post-training** is like sending them to finishing school to teach them manners, ethics, and how to apply that knowledge appropriately in a conversation.

Sampling

Sampling is the crucial process by which a foundation model constructs its output tokens, which makes the outputs of AI fundamentally **probabilistic**. Understanding sampling is essential because it explains seemingly baffling AI behaviours, such as **inconsistency** and **hallucination**, and choosing the right sampling strategy can significantly **boost a model's performance**.

Here is a detailed explanation of sampling, its fundamentals, and its various strategies:

Sampling Fundamentals

When a language model processes an input, it performs the following steps to decide the next token:

1. **Logit Vector Calculation:** The neural network first outputs a **logit vector**. Each logit corresponds to one possible value (a token) in the model's **vocabulary**.
2. **Probability Conversion (Softmax):** While larger logits correspond to higher probabilities, logits do not inherently represent probabilities. A **Softmax layer** is used to convert these logits into a **probability distribution** over all vocabulary tokens, where all probabilities are non-negative and sum up to one.
3. **Token Selection:** The model selects the next token based on this distribution. If the model consistently selects the outcome with the highest probability, it is called **greedy sampling**. While suitable for classification tasks, greedy sampling tends to create predictable and **boring outputs** for generative tasks. Instead, models typically sample the next token probabilistically according to the calculated distribution.

Sampling Strategies

The choice of sampling strategy determines the trade-off between the predictability and creativity of the model's output.

Temperature is a variable used to redistribute the probabilities of possible output values. It adjusts the logits *before* the softmax transformation by dividing the logits by the temperature value. A **higher temperature** reduces the probabilities of common tokens, increasing the likelihood of rarer tokens being selected. This results in **more creative outputs** but risks greater **inconsistency**. A temperature value of 0.7 is often recommended to balance creativity and predictability. Setting the temperature close to zero typically results in a deterministic, consistent, and potentially dull output.

Top-k is a sampling strategy used by language models during the output generation phase. After a model computes the raw scores (**logits**) for all words in its vocabulary, Top-k limits

the number of possible next tokens by selecting only the **k** tokens with the highest scores. The model then performs a **softmax calculation only over these k values to determine their probabilities**, and samples the next token from this reduced set. This technique is employed to reduce the computational workload associated with calculating probabilities across a very large vocabulary. A smaller k value makes the output more predictable but less interesting, as the model is restricted to a smaller set of likely words. Typical values for k can range from 50 to 500, which is significantly smaller than the full model vocabulary.

Top-p, also known as **nucleus sampling**, is an alternative sampling strategy that offers a more dynamic way to select tokens for the next output. Instead of fixing the number of choices (k), Top-p dynamically selects tokens based on their cumulative probability. The model ranks all possible next tokens by probability in descending order and continues summing their probabilities until the sum reaches a predetermined threshold, p . Only the tokens within this cumulative probability mass are considered for sampling.

NOTE: Cumulative probability, in the context of **nucleus sampling (Top-p)**, refers to the running sum of probabilities used by the model to decide which tokens are eligible for output generation

Use Case: Typical values for p range from 0.9 to 0.95. For example, setting $p=0.9$ means the model considers the smallest set of most likely tokens whose combined probability exceeds 90%, ensuring the chosen tokens are contextually appropriate. **The advantage of Top-p over Top-k is** that it adjusts the set size based on the context, **providing more relevance when predicting the next token**. If a query has only two highly likely answers (like "yes" or "no"), Top-p might select only those two, whereas for an open-ended question, it selects many more options.

Sampling for Output Quality and Control

Sampling is not just about token generation; it is also utilized in broader techniques to enhance output quality and enforce format constraints:

Test Time Compute: This technique improves response quality by generating **multiple responses (N)** for a single query and then selecting the best one. This strategy leverages the model's probabilistic nature to increase the likelihood of finding a good outcome. The best output is often selected based on domain-specific heuristics, a separate **reward model** (as used in preference finetuning), or the highest average **logprob** (log probability) score.

Constrained Sampling: To ensure models generate output in specific formats (e.g., JSON or regex), constrained sampling filters the logit vector to keep only the tokens that conform to predetermined rules or grammars. The model then samples exclusively from this filtered set of valid tokens.

Probabilistic Nature of AI

The reliance on probabilistic sampling means that AI outputs are not **deterministic**. This non-deterministic nature leads directly to two key issues:

1. **Inconsistency:** Giving the model the exact same prompt twice can result in two different responses.
2. **Hallucination:** Anything with a non-zero probability in the training data, no matter how wrong or far-fetched (such as misinformation), **can be generated**. Managing this probabilistic nature is a core focus of AI engineering efforts

Chapter 3: Evaluation Methodology

Complexities and current techniques used to assess the quality and performance of foundation models, which is necessary due to the severe consequences of potential catastrophic failures in AI applications.

Challenges of Evaluating Foundation Models

Evaluation is considered one of the hardest challenges in AI engineering, often consuming the majority of development efforts. Foundation models present unique hurdles compared to traditional machine learning models:

- **Complexity and Sophistication:** As AI models become more intelligent, evaluation requires time-consuming processes like **fact-checking**, **reasoning**, and incorporating **domain expertise**.
- **Open-Ended Outputs:** Traditional evaluation relies on comparing outputs against clear **ground truths**. Because generative outputs are open-ended, it is impossible to curate a comprehensive list of correct responses for comparison.
- **The Black Box Problem:** Many commercial models are black boxes, lacking transparency regarding their architecture, training data, and process, forcing developers to evaluate solely by observing external outputs.
- **Benchmark Saturation:** Public evaluation benchmarks (e.g., GLUE, MMLU) become saturated quickly as models improve, necessitating constant updates (e.g., MMLU-Pro) to accurately measure advancing capabilities.
- **Lagging Investment:** Despite the critical role of evaluation, investment in developing reliable methodologies and infrastructure lags behind the resources devoted to core model development.

Understanding Language Modeling Metrics

Metrics developed for historical language models remain relevant as they correlate with foundation model performance. These metrics are often variations of predictive accuracy:

- **Entropy (H):** Measures the information content carried by a token; higher entropy means more information.
- **Cross Entropy ($H(P,Q)$):** Quantifies the difficulty for a model (Q) to predict the tokens in a given dataset (P). It measures the divergence between the model's learned distribution and the true data distribution. Variations include **Bits-per-Character (BPC)** and **Bits-per-Byte (BPB)**.
- **Perplexity (PPL):** The exponential of cross entropy, measuring the **amount of uncertainty** when predicting the next token. Lower perplexity suggests higher predictive accuracy. Perplexity is useful for detecting **data contamination** in benchmarks and identifying **abnormal texts**.

Exact Evaluation Methods

Exact evaluation yields judgments without ambiguity, contrasted with subjective evaluation. This approach is difficult for open-ended generation tasks, but certain methodologies still apply:

- **Functional Correctness:** Assesses whether a model's output performs its intended action (e.g., a generated SQL query returns the correct result, or code executes successfully). Coding benchmarks often use **pass@k** to score functional correctness.
- **Similarity Measurements Against Reference Data:** Compares generated outputs against annotated **reference responses** (ground truths).
 - **Exact Match:** Only works for concise, definite outputs (e.g., trivia) due to the infeasibility of listing all correct options for complex tasks.
 - **Lexical Similarity:** Measures textual overlap using techniques like **n-gram similarity** or **fuzzy matching** (approximate string matching). Metrics include **BLEU** and **ROUGE**.
 - **Semantic Similarity:** Measures closeness in meaning (**semantics**) by converting text into numerical **embedding vectors**. Similarity is computed using mathematical measures like **cosine similarity**. The accuracy depends entirely on the **embedding algorithm** used.

AI as a Judge (LLM as a Judge)

The rising star of automated evaluation is the use of an AI model (**AI judge**) to grade other AI outputs, especially for subjective quality.

- **Rationale:** AI judges are **fast, cheap, and flexible**, allowing evaluation on metrics like **correctness, toxicity, and tone** without requiring reference data. Strong models like GPT-4 have demonstrated high correlation (up to 85% agreement) with human evaluators on certain benchmarks.
- **Process:** The judge is instructed via a prompt detailing the task, criteria, and scoring system (usually classification or discrete numerical scores).

- **Limitations:** AI judges are **probabilistic**, leading to **inconsistency**. They suffer from **criteria ambiguity** because metrics are non-standardized across tools. Furthermore, they exhibit inherent **biases**, including **self-bias** (favouring their own responses) and **verbosity bias** (favouring longer responses, regardless of accuracy).
- **Models:** While the strongest models are often preferred as judges, specialized **reward models** or **preference models** (which are lightweight and trained for specific scoring tasks) can also be used effectively.

Ranking Models with Comparative Evaluation

Comparative evaluation aims to determine which model is definitively better for a task, rather than assessing absolute performance.

- **Mechanism:** Models compete head-to-head in "matches," and an evaluator (human or AI) determines the preferred output. Results are aggregated using rating algorithms like **Bradley–Terry** or **Elo** to generate a public ranking.
- **Advantages:** It is generally easier for evaluators to choose a preferred output than to assign an accurate absolute score, and this method avoids the problem of benchmark saturation.
- **Drawbacks:** Comparison data is prone to quality control issues due to crowdsourcing, and the rankings reflect only relative preference, making it difficult to ascertain if a model is "good enough" for production criteria beyond winning a contest

Chapter 4: Evaluate AI Systems

Selecting and assessing foundation models for specific applications, depends entirely on its performance within its intended use case.

Evaluation Criteria

A successful AI application must be evaluated against defined criteria, encompassing **domain-specific capability**, **generation capability**, **instruction-following capability**, and **cost and latency**. This systematic approach, called **evaluation-driven development**, requires defining criteria before building to maximize business value.

1. Domain-Specific Capability

Domain-specific capabilities (e.g., coding, math, legal knowledge) are usually constrained by a model's architecture and training data.

- **Coding/Functional Correctness:** Capabilities like code generation are assessed using **functional correctness** metrics, such as **pass@k**, which determine if the generated output performs its intended action (e.g., if generated code executes successfully). Efficiency (runtime, memory usage) may also be checked.

- **Knowledge/Reasoning:** Non-coding domain capabilities are commonly evaluated using **multiple-choice questions (MCQs)**, such as those found in benchmarks like **MMLU** and **AGIEval**. However, MCQ performance can vary significantly with small changes in how questions are prompted.

2. Generation Capability

Generation capability metrics track the quality and safety of open-ended outputs. Metrics like **fluency** and **coherence** (common in early NLP) are less critical now due to improved model quality, but newer metrics track complex issues like **hallucinations** (factual consistency) and **safety**.

- **Factual Consistency:** This is measured against either **explicitly provided facts (local factual consistency)** or **open knowledge (global factual consistency)**. Techniques often employ **AI judges, self-verification** (checking model consistency across multiple generations), or **knowledge-augmented verification** using search engine results. The problem can also be framed as **textual entailment**.
- **Safety:** This is an umbrella term covering harms like **inappropriate language, harmful recommendations, hate speech, violence, and political/religious biases**,. Specialized models, like Facebook's hate speech detection tools, are often used instead of general-purpose AI judges.

3. Instruction-Following Capability

This measures how well a model adheres to specified instructions, which is crucial for applications requiring **structured outputs** (like JSON). Benchmarks like **IFEval** verify adherence to explicit constraints (e.g., keyword inclusion, length limits), while **INFOBench** evaluates broader constraints (e.g., style and tone), often relying on AI judges for verification. **Roleplaying**, a common instruction type, requires evaluation to ensure the model stays in character.

4. Cost and Latency

Optimizing latency (e.g., **TTFT** and **TPOT**) and cost is critical for usability, Cost is often measured by API token usage or underlying compute cost.

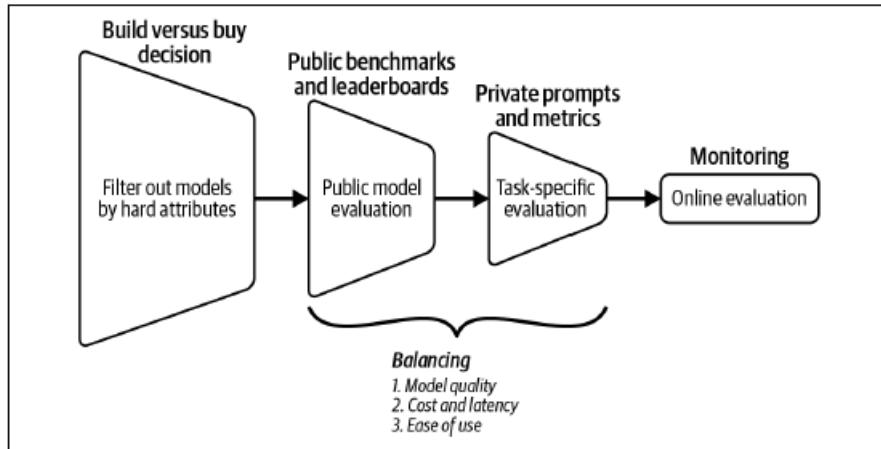
Model Selection Workflow

The goal is to find the most suitable model for an application by balancing quality, cost, and latency. The workflow involves four steps:

1. **Filter Hard Attributes:** Eliminate models whose hard attributes (e.g., license restrictions, data privacy policies, model size compatibility with hardware) preclude their use.
2. **Navigate Public Benchmarks:** Use benchmarks and leaderboards to identify **promising models** to experiment with. Developers must be wary of **data**

contamination, where models score deceptively high because benchmark data was included in their training data.

3. **Run Private Evaluation Pipeline:** Conduct rigorous experiments using custom metrics and data tailored to the application's specific needs.
4. **Continually Monitor in Production:** Track the model's performance after deployment to detect failures and gather feedback.



Model Build Versus Buy

A key decision is whether to **use model APIs (buy)** or **self-host open-source models (build)**.

Model APIs offer ease of use, scaling, and functionality (e.g., built-in guardrails), but they impose costs (per token), limit control, and raise data privacy concerns.

Self-hosting provides control over model customization (e.g., finetuning), access to internal model details (like **logprobs**), and mitigation of data privacy risks, but it requires significant engineering effort for deployment, optimization, and maintenance. The gap between open-source models and proprietary models is closing, making self-hosting increasingly viable for many use cases.

Evaluation Pipeline Design

A reliable evaluation pipeline must be designed to evaluate both **end-to-end output** and the output of **intermediate components** (e.g., evaluating text retrieval quality separately from the final response generation).

1. **Evaluate All Components in a System:** A comprehensive pipeline requires evaluating every element, not just the final outcome. Failures in complex applications can stem from any intermediate step, making isolated component evaluation necessary for accurate debugging.
 - **Per-Turn Evaluation:** Assessing the quality of every output generated in a conversation.
 - **Per-Task Evaluation:** Assessing whether the overall task goal was achieved, which is typically more important as it reflects user satisfaction.

2. **Create an Evaluation Guideline:** This defines **evaluation criteria**, determines a clear **scoring rubric** with examples, and maps performance metrics directly to **business metrics** (e.g., linking factual consistency scores to the percentage of automated customer support requests).
3. **Define Evaluation Methods and Data:** Select appropriate methods (e.g., mixing small, cheap specialized classifiers with expensive AI judges for spot checks) and collect and annotate evaluation data. Data should be explicitly **sliced** (broken into subsets based on features like user type or prompt length) to ensure fair assessment and detect biases.

Chapter 5: Prompt Engineering

"Prompt Engineering," focuses on the essential and widespread technique of crafting instructions (prompts) to guide a foundation model toward generating a desired output *without changing the model's internal weights*. This process is the most common and easiest method for adapting models.

Fundamentals of Prompting

- **Prompt Definition:** A prompt is the entire input given to a model to perform a task. While it may sometimes be used interchangeably with 'context,' context, in this book, refers to the relevant information provided within the prompt.
- **Prompt Components:** A prompt generally includes a **task description** (what the model should do, including its role and expected output format), **examples** (for few-shot learning), and the **concrete task** itself (e.g., the question to answer).
- **In-Context Learning (ICL):** Teaching a model desirable behaviour via examples included in the prompt is known as ICL or few-shot learning. This is valuable because it allows the model to incorporate new information continuously without being retrained, preventing it from becoming outdated. When no examples are provided, it is **zero-shot learning**.
- **System and User Prompts:** Many model APIs allow the separation of instructions into a **system prompt** (the task description and role) and a **user prompt** (the specific task). System prompts often appear first and can boost performance, potentially because the model was post-trained to prioritize these instructions.
- **Context Length and Efficiency:** The length of the prompt is constrained by the model's maximum **context length**. Models are often more effective at using information placed at the **beginning and end** of the context rather than the middle, a phenomenon tested using the "needle in a haystack" (NIAH) test.

Best Practices

Effective prompt engineering relies on general techniques that balance clarity and efficiency:

- **Clarity and Explicitness:** Instructions should be unambiguous, explaining exactly what the model should do, including the desired scoring system or expected output format. Asking the model to adopt a **persona** helps it use the correct perspective, and **providing examples** reduces ambiguity.
- **Sufficient Context:** Providing enough context (e.g., relevant documents or data) is necessary to improve the model's response quality and help **mitigate hallucinations** by reducing reliance on potentially unreliable internal knowledge.
- **Decomposition:** For complex tasks, breaking them down into simpler **subtasks** and chaining them together improves performance, facilitates debugging, and can reduce costs (by allowing cheaper models or parallel execution for simpler steps).
- **"Time to Think":** Techniques like **Chain-of-Thought (CoT)**, which explicitly ask the model to "think step by step" or explain its reasoning, encourage systematic problem-solving and can significantly improve performance on complex tasks. **Self-critique** involves prompting the model to check and revise its own output.
- **Iteration and Tools:** Prompt engineering requires continuous **iteration** and testing to find the optimal prompts. Specialized tools can help **automate prompt creation** and optimization, but developers must carefully **evaluate these tools** to ensure they use correct templates, control hidden API calls (which incur cost), and verify performance. It is also considered good practice to **organize and version prompts** separately from code.

Defensive Prompt Engineering

The ability of foundation models to follow instructions makes them vulnerable to **prompt attacks**, where malicious instructions are injected to compromise the application.

Proprietary Prompt Risks: Some teams consider their prompts proprietary, but they are vulnerable to **reverse prompt engineering**, where an attacker deduces the internal system prompt, potentially to replicate or exploit the application.

Jailbreaking and Prompt Injection: These attacks aim to bypass the model's safety features (jailbreaking) or inject malicious instructions into user queries (prompt injection). These are possible because models are trained to follow instructions, even malicious ones.

Sophisticated attacks include **indirect prompt injection**, where instructions are placed in tools or documents that the model later retrieves and executes.

Information Extraction: Attackers may exploit the model's memory to **extract sensitive or copyrighted information** from the training data or the private context provided during a query.

Defenses: Defenses can be implemented at the **model level** (e.g., training the model to prioritize system instructions over conflicting user instructions), the **prompt level** (e.g., repeating safety instructions and making forbidden actions explicit), and the **system level**

(e.g., isolating code execution in a virtual machine and adding external guardrails and access controls)

Chapter 6: RAG and Agents

Retrieval-Augmented Generation (RAG) and **agents** as the two dominating patterns for dynamically constructing the context required by foundation models to produce accurate outputs. While RAG primarily focuses on context construction, agents utilize tools and sophisticated planning to expand a model's capabilities and interact directly with its environment, opening up possibilities for autonomous assistants and automated workflows.

Retrieval-Augmented Generation (RAG)

RAG enhances a model's output quality by retrieving relevant information from external memory sources, such as internal databases or user conversation history. This retrieve-then-generate pattern is a common and established technique, crucial for tasks requiring knowledge that exceeds a model's training data or context window.

RAG Architecture and Retrieval Algorithms

A basic RAG system comprises two main components: a **retriever** and a **generator**. The retriever handles two functions: **indexing** (processing data for quick future retrieval) and **querying** (fetching relevant data based on the user request).

Retrieval algorithms determine how relevance scores are computed, typically categorized into two types:

- **Term-Based Retrieval (Sparse):** Algorithms like **TF-IDF** and **BM25** rank documents based on keyword overlap (lexical retrieval). This method is generally faster and simpler but can struggle with semantic ambiguity.
- **Embedding-Based Retrieval (Dense/Semantic):** This approach converts documents into numerical vectors (**embeddings**) using an **embedding model** and stores them in a **vector database**. Retrieval works by performing an **approximate nearest neighbor (ANN)** search to find documents whose embeddings are closest in meaning (semantics) to the query embedding. This approach uses models like CLIP and can significantly outperform term-based methods after fine-tuning, though it incurs higher computational overhead for embedding generation and search.

Retrieval systems often employ **hybrid search**, combining term-based and embedding-based methods, or **reranking**, where a costly, precise mechanism reorders candidates retrieved by a cheaper one.

Retrieval Optimization

To enhance the performance of RAG systems, several tactics are utilized:

- **Chunking Strategy:** Documents are split into manageable segments (**chunks**) before indexing. Strategies include splitting by fixed size (e.g., characters, words, paragraphs) or recursively. Overlap is often necessary to prevent loss of critical boundary information. Smaller chunks allow more diverse information in the context but increase indexing overhead.
- **Query Rewriting:** For multi-turn conversations where a query is ambiguous, an AI model can be used to reformulate the query to include necessary context, improving retrieval accuracy.
- **Contextual Retrieval:** Chunks are augmented with **metadata** (e.g., tags, keywords, titles, summaries) to provide richer context to the retriever, especially useful for specific data points like error codes.

RAG is also applicable beyond text, supporting **multimodal RAG** (augmenting context with images, audio, or other modalities) and **RAG with tabular data** (involving text-to-SQL generation and execution).

Retrieval Algorithms

The performance of a Retrieval-Augmented Generation (RAG) system hinges on the quality of its **retriever**, the component responsible for dynamically finding relevant information from external memory sources. Retrieval algorithms determine how relevance scores are computed, serving as the core mechanism for ranking documents or data chunks relative to a user's query.

Retrieval algorithms are generally categorized into two main types: **Term-Based Retrieval (Sparse)** and **Embedding-Based Retrieval (Dense/Semantic)**.

1. Term-Based Retrieval (Sparse)

Term-based retrieval, also known as **lexical retrieval**, focuses on keyword overlap to find relevant documents. This approach is generally faster and simpler than embedding-based methods but struggles with semantic ambiguity.

The main challenge addressed by these algorithms is determining which documents containing the query terms are most relevant, leading to the combination of **Term Frequency (TF)** and **Inverse Document Frequency (IDF)**.

- **Term Frequency (TF):** Measures the number of times a term appears in a document. The assumption is that the more often a term appears, the more relevant the document is to that term.
- **Inverse Document Frequency (IDF):** Measures the importance or informativeness of a term. It operates on the intuition that if a term appears in many documents (e.g., "for" or "at"), it is less informative; thus, a term's importance is inversely proportional to the number of documents it appears in.

The **TF-IDF algorithm** mathematically combines these two metrics to score a document's relevance to a query.

Okapi BM25: This is a modification of TF-IDF and remains a formidable baseline against modern retrieval algorithms. BM25 normalizes term frequency scores by document length, acknowledging that longer documents are naturally more likely to contain a given term.

Term-based systems often use data structures like an **inverted index** (used by solutions like Elasticsearch) to quickly map terms to the documents that contain them.

2. Embedding-Based Retrieval (Dense/Semantic)

Embedding-based retrieval, also known as **semantic retrieval**, ranks documents based on how closely their meanings (**semantics**) align with the query, rather than just keyword matching. This approach is effective in addressing the problem of irrelevant documents being returned due to lexical ambiguity.

This method involves converting data chunks into **embeddings** (numerical vectors that capture meaning) and storing them in a **vector database**.

The retrieval process consists of two key steps:

1. **Embedding Model:** The user query is converted into an embedding using the same **embedding model** employed during indexing.
2. **Retriever:** The retriever performs an **approximate nearest neighbor (ANN)** search in the vector database to fetch the k data chunks whose embeddings are **closest in meaning** to the query embedding. Closeness is often measured using metrics like **cosine similarity**.

Since performing a precise nearest-neighbour search is computationally expensive for large datasets, search is typically done using **ANN algorithms**. Vector search algorithms often organize vectors into buckets, trees, or graphs, differing based on the heuristics used to speed up the similarity search. Notable algorithms include:

- **Locality-Sensitive Hashing (LSH):** Works by hashing similar vectors into the same buckets.
- **Hierarchical Navigable Small World (HNSW):** Constructs a multi-layer graph where nodes (vectors) are connected by edges (similarity) to enable efficient traversal.
- **Product Quantization:** Reduces each vector into a simpler, lower-dimensional representation to accelerate distance computation.
- **Inverted File Index (IVF):** Uses K-means clustering to group similar vectors, allowing searches to target only the clusters closest to the query embedding.

- **Annoy (Approximate Nearest Neighbors Oh Yeah):** A tree-based approach that builds multiple binary trees to gather candidate neighbors.

Comparison and Combination

Feature	Term-Based Retrieval	Embedding-Based Retrieval
Speed/Cost	Faster and cheaper during indexing and querying.	Indexing is slower due to embedding generation; querying involves computationally expensive vector search.
Performance	Strong performance out of the box; constrained by keyword match; hard to improve significantly.	Uses semantics, allowing for more natural queries; can outperform term-based retrieval after fine-tuning.
Weakness	Can struggle with term ambiguity (e.g., "transformer" the movie or the architecture).	High computational overhead; can obscure specific keywords (like error codes) that are important for lookup.

Production systems often use **hybrid search**, combining term-based and embedding-based methods to leverage the strengths of both. This combination is sometimes achieved through **reranking**, where a cheap, less precise retrieval mechanism fetches candidate documents, and then a more precise but costlier mechanism (such as vector search) reorders or validates those candidates. Combining rankings from multiple retrievers can also be achieved using algorithms like **Reciprocal Rank Fusion (RRF)**.

Agents

Agents are viewed as the **ultimate goal of AI**, capable of perceiving their environment and acting upon it. They are characterized by their **environment** (e.g., a codebase, the internet) and their inventory of **tools**.

Agent Components: Tools and Planning

The core of an AI-powered agent is the planning mechanism, which uses the foundation model as a **planner**.

1. **Tools:** External tools augment an agent's capabilities beyond simple text generation. Tools fall into three categories:
 - **Knowledge Augmentation:** Tools like text retrievers, SQL executors, and web browsers, which allow the agent to **perceive** its environment (read-only actions).

- **Capability Extension:** Tools that address model limitations, such as calculators for math, code interpreters for execution/debugging, or text-to-image models for multimodal outputs.
 - **Write Actions:** Tools that allow the agent to **act upon** the environment, such as sending emails, placing orders, or initiating bank transfers, introducing higher risk. Functionality is often exposed via **function calling**.
2. **Planning:** Complex tasks require the agent to generate a plan, a roadmap of sequential steps (actions) to reach a goal. For robust performance, planning should be **decoupled from execution**, meaning a plan is validated before being run. The agent workflow often involves reflection and error correction to evaluate plans and execution outcomes, a process sometimes implemented through patterns like **ReAct** (Reasoning and Acting). Planning often involves complex control flows beyond simple sequential execution, such as parallel, if/else, and for loops.

Memory Systems

Both RAG and agents require robust **memory systems** to handle extensive information that may exceed the model's instantaneous context limits. These systems typically utilize three memory mechanisms:

- **Internal Knowledge:** Inherent information learned during pre-training.
- **Short-Term Memory (Context):** The model's context window, offering fast access but limited capacity, usually storing immediate, session-specific information.
- **Long-Term Memory (External Data):** External data sources accessible via retrieval (like a RAG index), which is persistent and large-capacity.

Memory management includes strategies (e.g., FIFO) and mechanisms (e.g., summarization, entity tracking) to manage memory overflow during multi-turn conversations and persist information across sessions for personalization and consistency

Chapter 7: Finetuning

"Finetuning," details the process of adapting a foundation model to a specific task by adjusting its internal weights through further training.

Finetuning is positioned as a **resource-intensive technique** that is typically explored after simpler prompt-based methods have proven insufficient.

Finetuning is a form of **transfer learning**, allowing models to apply knowledge gained during pre-training to accelerate learning on a new, related task. This process is crucial because it can refine a model's behaviour and **unlock capabilities** that are difficult to access through prompting alone.

The primary aims of finetuning include improving a model's general and **task-specific capabilities** (e.g., coding, or medical Q&A), enhancing its ability to adhere to specific **output formats** (like JSON), and strengthening its **safety alignment**.

Finetuning extends pre-training, but instead of starting with randomized weights, it starts with weights from a previously trained base model. This process improves **sample efficiency**, meaning the model requires fewer examples (hundreds or thousands) to learn complex behaviours compared to training from scratch (millions).

Types Finetuning includes:

- **Continued Pre-training (Self-supervised Finetuning):** Finetuning using cheap, unlabeled, task-related data (e.g., raw legal documents).
- **Supervised Finetuning (SFT):** Refining the model using high-quality annotated data pairs (instruction, response) to align with human usage patterns.
- **Preference Finetuning:** Using reinforcement learning to maximize human preference, typically requiring comparison data (instruction, winning response, losing response).

When to Finetune: Finetuning vs. RAG

Finetuning requires significant investments in data, hardware, and ML talent, making the decision of *when* to pursue it critical.

The choice between finetuning and Retrieval-Augmented Generation (RAG) depends on the model's **failure mode**:

Technique	Primary Goal	Failure Mode Addressed	Example
RAG	Facts/Knowledge	Information-based (lack of current or private knowledge, hallucination)	Supplementing a model with specifications to answer a specific product query
Finetuning	Form/Behaviour	Behaviour-based (failure to follow format, irrelevant or unsafe responses)	Correcting a model to write HTML code that compiles or adhering to a specific SQL dialect

While RAG often yields a significant performance boost for factual tasks, finetuning can be necessary to correct behavioural issues. The two techniques can be combined for maximum effect.

The Memory Bottleneck and Optimization

Finetuning massive foundation models is **memory-intensive**, which has driven the development of various optimization techniques.

- **Memory Composition:** Training memory includes space for model weights, activations, and key components needed for the backward pass: **gradients** and **optimizer states**. For full finetuning, the total memory required often dwarfs the capacity of a single consumer GPU.
- **Quantization:** This is a highly effective optimization that **reduces the number of bits** used to represent model parameters (e.g., from 32-bit to 16-bit float). This significantly reduces the model's memory footprint, allowing larger models to run on cheaper hardware and improving computation speed.
- **Parameter-Efficient Finetuning (PEFT):** PEFT methods, such as LoRA, aim to achieve performance comparable to full finetuning while updating only a **small fraction of the model's total parameters**.
 - **LoRA (Low-Rank Adaptation):** This dominant PEFT method introduces modular, smaller matrices (adapters) into the model that can be updated separately and later merged back into the original weights without increasing inference latency. LoRA and its quantized variant, QLoRA, make large model finetuning accessible, even on modest hardware.
- **Model Merging:** This technique combines the weights or adapters of multiple models (often finetuned models) to create a single, custom model, enabling **multi-task capability** or **on-device deployment** using reduced memory. Merging approaches include summing (e.g., linear combination or SLERP) and layer stacking.

Finetuning Execution

The actual finetuning process involves selecting a **base model**, a **method (like LoRA)**, and **configuring hyperparameters**:

- **Base Model Selection:** Start by testing code and data on the cheapest, fastest models. If performance is good, gradually scale toward larger models until an optimal **price/performance frontier** is found.
- **Hyperparameter Tuning:** Critical tuning points include **learning rate** (how fast weights adjust), **batch size** (number of examples processed per step), and **number of epochs** (passes over the data). For instance, due to memory constraints, techniques like **gradient accumulation** may be needed to maintain a high effective batch size.
- **Data Requirement:** While millions of examples may be needed for full finetuning, PEFT methods can yield strong performance with just a few hundred to a few thousand high-quality examples. However, smaller models are prone to **catastrophic forgetting** or **ossification** if exposed to insufficient data during finetuning

Chapter 8: Dataset Engineering

"Dataset Engineering," addresses the process of creating and managing datasets crucial for adapting foundation models, emphasizing that high-quality data is increasingly viewed as

the main differentiator for AI performance. Due to the growing demand for data, dataset creation and management now require dedicated expertise and significant investment.

Post-training data curation, generation, and processing, explaining that successful dataset creation, regardless of the training phase, relies on meeting three core criteria: **quality**, **coverage** (diversity), and **quantity**.

1. Data Curation

Data curation is the systematic process of gathering the right data to teach a model desired behaviors, such as **Chain-of-Thought (CoT)** reasoning or **tool use**.

2. Data Quality

High-quality data is essential, as a small amount of carefully crafted data can often outperform a large volume of noisy, irrelevant, or inconsistent data. Data quality is defined by six characteristics:

- **Relevant:** Examples must pertain to the specific task the model is being trained for.
- **Aligned with task requirements:** Annotations should match the task's criteria (e.g., factual correctness, creativity, conciseness).
- **Consistent:** Annotations should be uniform across examples and annotators, as inconsistent annotations can confuse the model.
- **Correctly formatted:** Examples must follow the model's expected format, free from issues like extraneous tokens, inconsistent casing, or incorrect numerical formats.
- **Sufficiently unique:** Duplications must be managed to prevent skewing the data distribution or causing **data contamination**.
- **Compliant:** Data must adhere to all relevant privacy policies and regulations (e.g., must not contain PII data).

3. Data Coverage and Diversity

The training data must cover the full range of problems a model is expected to solve, requiring sufficient **diversity** in usage patterns, topics, languages, and styles.

Increasing task diversity during finetuning can significantly boost performance. For instance, highly technical domains like math and coding often require a disproportionately large amount of high-quality data to boost a model's reasoning capabilities.

4. Data Quantity

The amount of data needed depends on several factors: **finetuning technique** (PEFT needs less than full finetuning), **task complexity**, and the **base model's initial performance**.

- **Small vs. Large Datasets:** While full finetuning may require millions of examples, lighter techniques like **PEFT** can yield strong results with just a few hundred or thousand high-quality examples. Experiments with small datasets (e.g., 50–100 examples) can help estimate whether a large investment in data curation will lead to performance gains.
- **Diminishing Returns:** Generally, larger datasets yield diminishing returns; initial data provides larger performance boosts than subsequent additions.

5. Data Acquisition and Annotation

The most valuable data often comes from **user feedback** and the application itself, enabling a **data flywheel** that improves the product over time. However, manually creating training data requires time-consuming annotation work and rigorous effort in establishing clear annotation guidelines.

Data Augmentation and Synthesis

Data synthesis—generating data that mimics real-world properties—is used to address limitations in data quantity, coverage, and quality.

1. **Traditional Synthesis Techniques** **Rule-based synthesis:** Generates data using predefined templates, often used for structured documents like invoices or math equations. It can also augment existing data by applying simple transformations, such as replacing words with synonyms or replacing gendered pronouns to mitigate biases.

Use Case: Creates synthetic data for scenarios that are too expensive, dangerous, or rare to observe in the real world (e.g., self-driving car accidents or complex robotics movements).

2. AI-Powered Data Synthesis

Advanced models enable sophisticated data generation:

- **Imitation and Paraphrasing:** AI can translate data to low-resource languages (e.g., Quechua) or paraphrase existing examples to increase diversity (e.g., generating 400,000 math examples from a small seed set).
- **Instruction Data Synthesis:** For finetuning, AI can generate new instructions, new responses, or even use the **reverse instruction approach**—generating instructions that would elicit known high-quality human-written text—to synthesize high-quality instruction data.
- **Self-Correction:** Advanced pipelines use AI to generate data, run it through checkers (e.g., code parsers, linters, unit tests), and then prompt the AI to correct its own errors, ensuring high fidelity before data is used for training.

3. Model Distillation

Model distillation involves training a smaller model (student) to imitate the behavior of a larger model (teacher) by training the student on data generated by the teacher. This is often used to create a smaller, faster model that retains the performance of the original large model, helping reduce inference costs.

4. Limitations of Synthetic Data

Despite its promise, AI-generated data has limitations:

- **Superficial Imitation:** Student models may mimic the *style* of the teacher without achieving true improvements in factual accuracy or reasoning.
- **Model Collapse:** Recursively training models on synthetic data can introduce irreversible defects, potentially degrading performance and causing models to forget rare, improbable events.
- **Data Verification:** Verifying the quality and factual consistency of synthetic data remains challenging, often requiring AI-powered judges and rigorous test pipelines.

Data Processing

Raw data must undergo several crucial processing steps before use:

- **Inspection:** Requires examining data through statistical analysis (e.g., plotting token distribution and response lengths) and **manual inspection** to gain essential insights into quality and identify unusual patterns.
- **Deduplication:** Uses similarity metrics (e.g., exact match, fuzzy match, semantic similarity) to eliminate redundant examples, which are highly detrimental to model performance.
- **Cleaning and Filtering:** Involves removing extraneous formatting tokens (like HTML tags) and ensuring compliance by filtering out PII, sensitive, or toxic content.
- **Formatting:** Ensures the final dataset adheres precisely to the chat template and tokenization scheme required by the finetuning model, as incorrect formatting can introduce errors

Chapter 9: Inference Optimization

Inference Optimization focuses on the techniques and strategies required to make foundation model **inference faster and cheaper**, which is essential for user satisfaction and the economic viability of AI products. While model quality is an internal property, inference performance is defined by how well the model is served on specific hardware.

1. Understanding Inference Bottlenecks

Optimisation begins with identifying whether a workload is **compute-bound** or **memory bandwidth-bound**.

- **Compute-bound:** The speed is limited by the number of mathematical operations the hardware can perform (e.g., password decryption or the **prefilling** phase of LLM inference).
- **Memory bandwidth-bound:** The speed is limited by how quickly data can be moved between memory and processors (e.g., the **decoding** phase of autoregressive models).

Language model inference typically involves both: **prefilling** (processing input tokens in parallel) is compute-bound, while **decoding** (generating one token at a time) is memory bandwidth-bound because it requires loading the entire model's weights for every single token produced.

2. Performance Metrics

To evaluate an inference service, several key metrics are used:

- **Latency:** This includes **TTFT (Time to First Token)**, which measures the prefill duration, and **TPOT (Time per Output Token)**, which measures the speed of subsequent generation.
- **Throughput:** The total number of tokens generated per second across all users. There is often a trade-off where increasing throughput via batching can increase individual user latency.
- **Goodput:** The number of requests per second that successfully meet a defined **Service Level Objective (SLO)**.
- **Utilization:** **MFU (Model FLOP/s Utilization)** measures how much of a chip's theoretical peak computation is actually used, while **MBU (Model Bandwidth Utilization)** measures how efficiently the expensive memory bandwidth is being utilized.

3. Model-Level Optimization

These techniques modify the model itself to increase efficiency:

- **Model Compression:** This includes **quantization** (reducing numerical precision, e.g., from 16-bit to 4-bit) and **distillation** (training a smaller "student" model to mimic a "teacher" model).
- **Speculative Decoding:** A smaller, faster "draft" model predicts a sequence of tokens that the larger "target" model then verifies in parallel, turning a sequential bottleneck into a parallelizable task.
- **Parallel Decoding:** Techniques like **Medusa** or **Lookahead decoding** use multiple decoding heads or Jacobi methods to predict and verify multiple future tokens simultaneously.
- **Attention Optimization:** The **KV (Key-Value) Cache** stores previously computed vectors to avoid redundant calculations. Redesigns like **Grouped-Query Attention**

(GQA) or Multi-Query Attention (MQA) further reduce the memory footprint of this cache.

4. Inference Service Optimization

These techniques improve resource management without altering model behavior:

- **Batching: Continuous batching** (or in-flight batching) is more efficient than static or dynamic batching for LLMs because it allows new requests to join a batch as soon as an existing one completes its generation.
- **Decoupling Prefill and Decode:** Because these phases have different bottlenecks, disaggregating them onto separate GPU instances can prevent prefill tasks from interrupting the throughput of decoding tasks.
- **Prompt Caching:** This allows the system to store and reuse overlapping prompt segments (like long system instructions or frequently referenced documents), drastically reducing TTFT and cost.
- **Parallelism: Tensor parallelism** splits individual operators across multiple devices to fit larger models and reduce latency, while **replica parallelism** simply creates multiple copies of a model to handle higher traffic volumes.

Analogy: If running an AI model is like serving a meal in a restaurant, **model-level optimization** is like simplifying the recipe so it's faster to cook. **Service-level optimization** is like improving the kitchen's management—using a faster "bus" (continuous batching) to move dishes or keeping a "heated shelf" (prompt caching) for ingredients that are used in every single order.