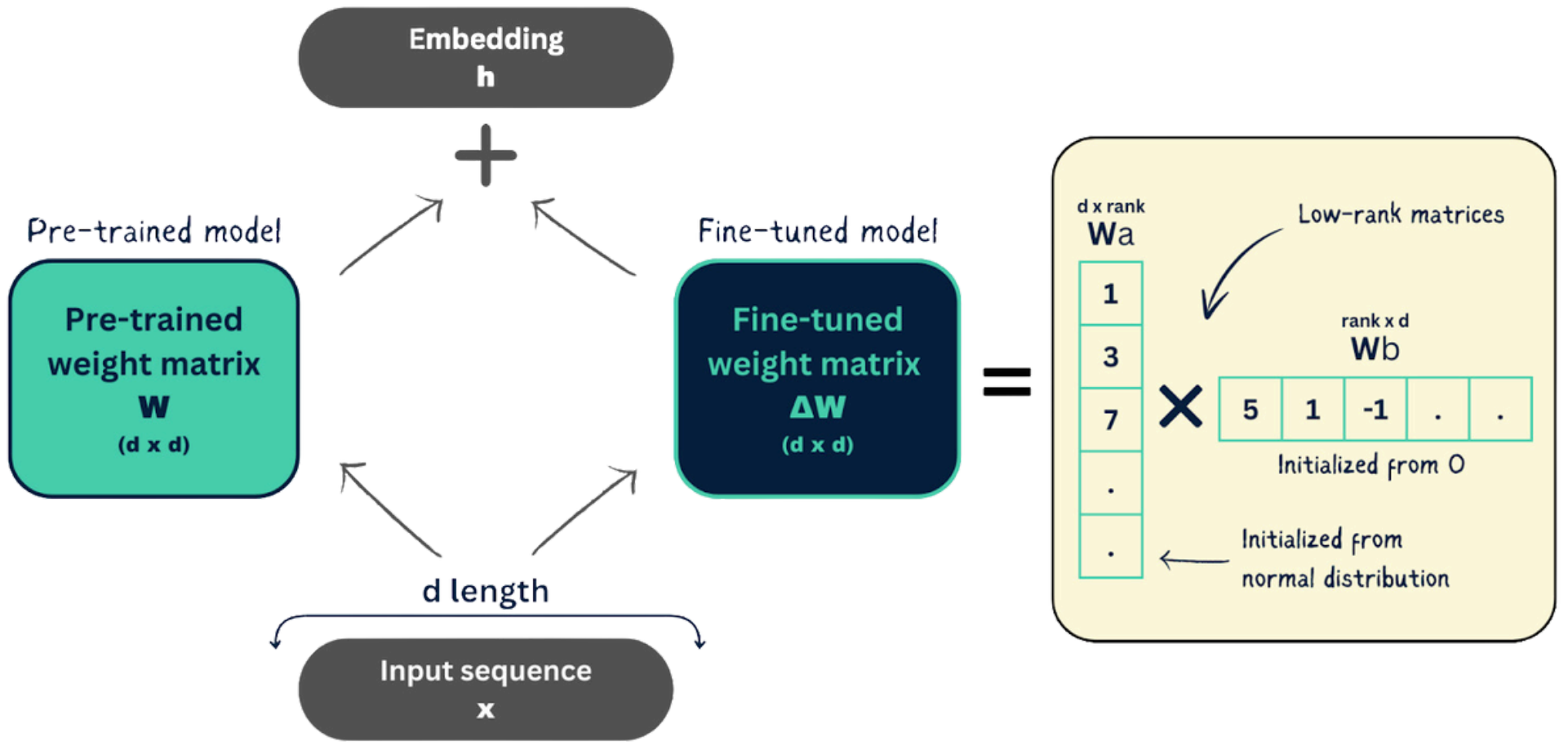
15 STRATEGIES TO REDUCE LLM COSTS

Bhavishya Pandit

1. LOW-RANK ADAPTATION (LORA)

LoRA adapts small, low-rank layers in a neural network for task-specific fine-tuning. The rest of the model remains untouched, which minimizes memory and computational demands.



Source

How it's done*:

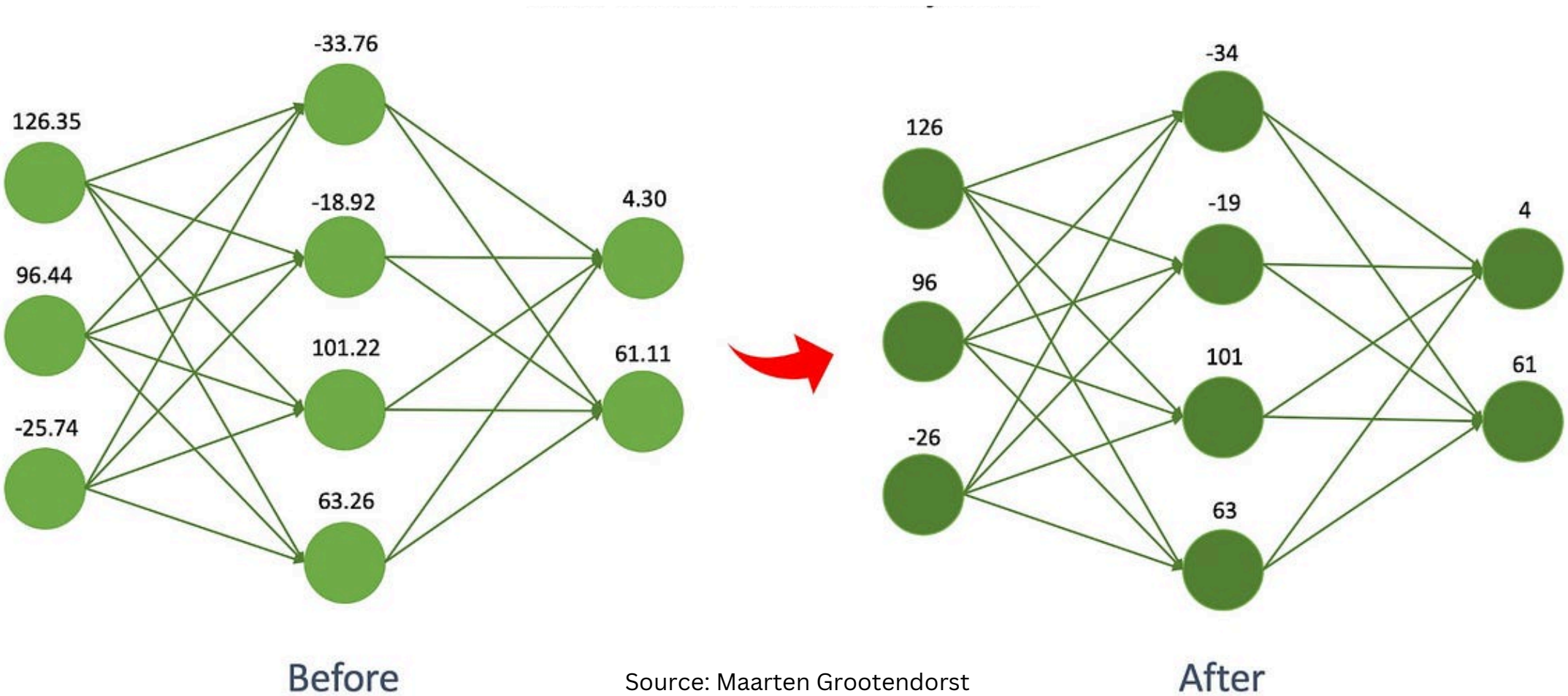
Introduce low-rank decomposition layers to the existing model. These layers adapt the pretrained model without altering its primary structure. Libraries like LoRA in PyTorch simplify implementation.

It's highly effective for customizing models across tasks without inflating resource usage, making it a cost-effective fine-tuning solution.

*This is meant to give just overview

2. MODEL QUANTIZATION

Quantization reduces the precision of the model weights, typically from 32-bit floating point to 8-bit integers, resulting in reduced cost, lower memory usage and faster computations.



Source: Maarten Grootendorst

How it's done*:

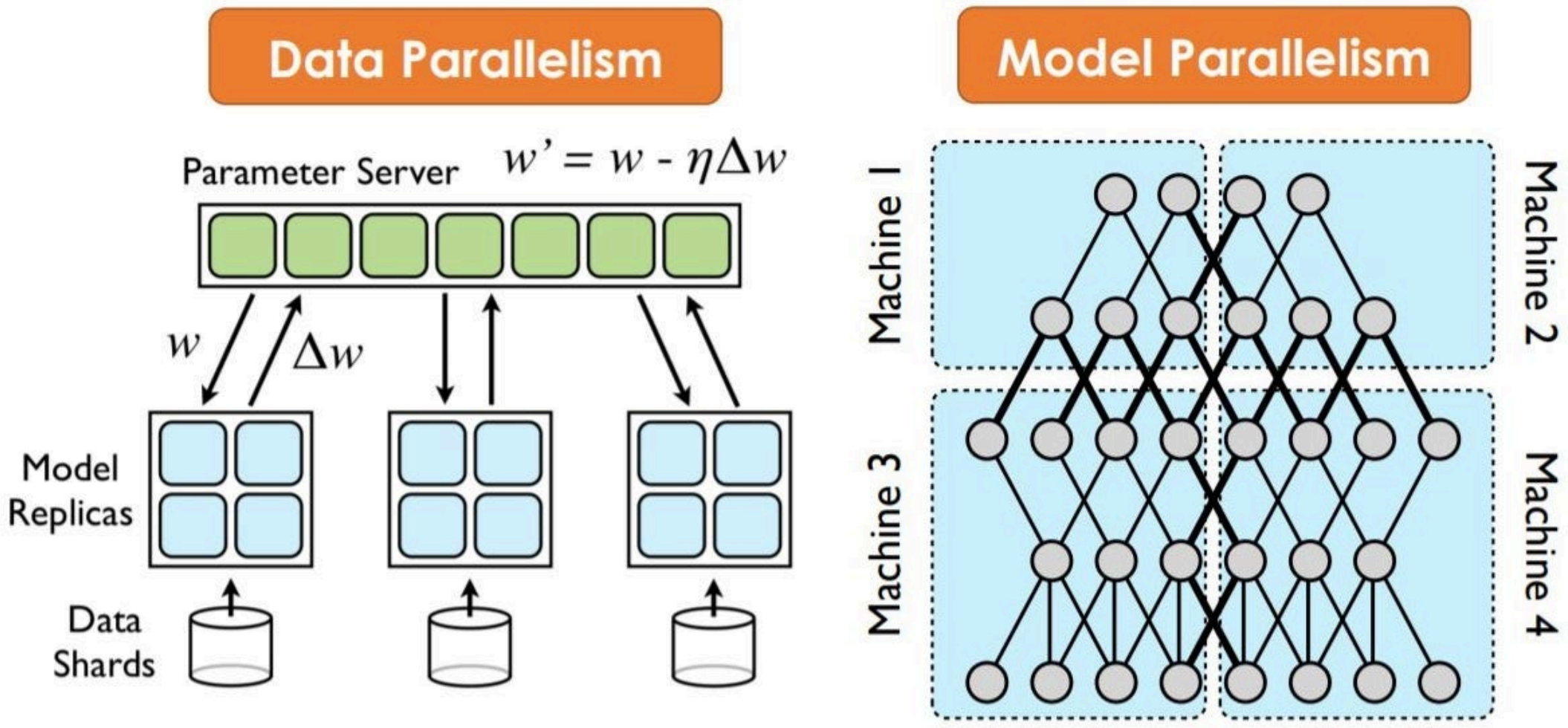
Use quantization-aware training or post-training quantization techniques. Tools like TensorFlow Lite, PyTorch's quantization toolkit, or ONNX runtime can help. These convert models to use lower-precision data types like INT8.

Lower precision means fewer resources needed for storage and computation, reducing costs while maintaining a good level of model performance.

*This is meant to give just overview

3. PIPELINE PARALLELISM

Pipeline parallelism splits a model's computation into smaller segments distributed across multiple devices. Each device processes a specific part of the model in parallel.



Source

How it's done*:

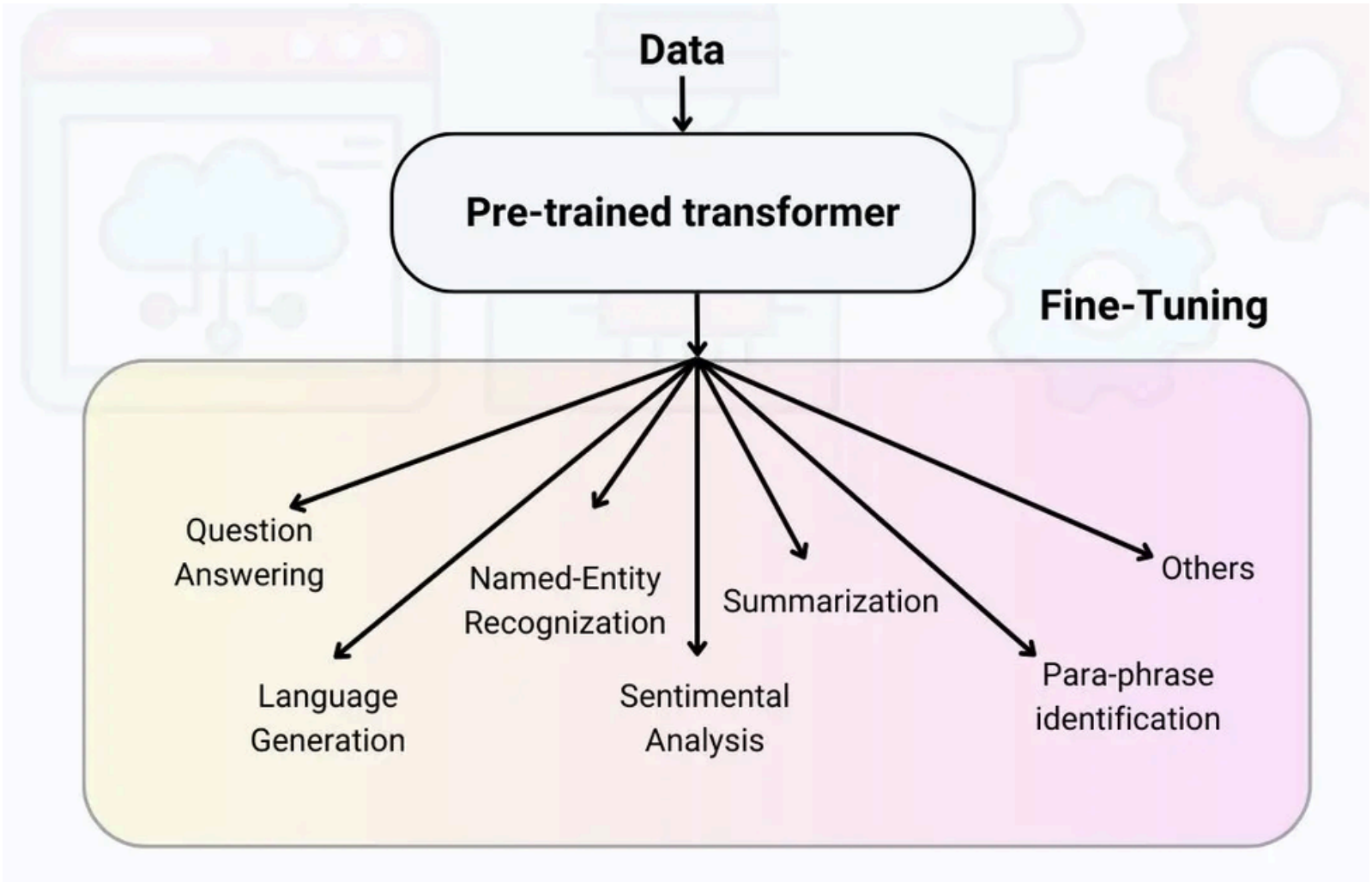
Divide the model into segments, each processed on a different GPU or device in a pipeline-like fashion. Frameworks like PyTorch's pipeline parallelism and DeepSpeed help manage these distributed computations.

It optimizes the use of hardware resources, accelerates training, and makes it feasible to handle very large models.

*This is meant to give just overview

4. FINE-TUNING

Fine-tuning involves adapting a pretrained model to a specific task by training it further on domain-specific or task-specific data. This customization adjusts the model's parameters for enhanced performance on new objectives.



Source

How it's done*:

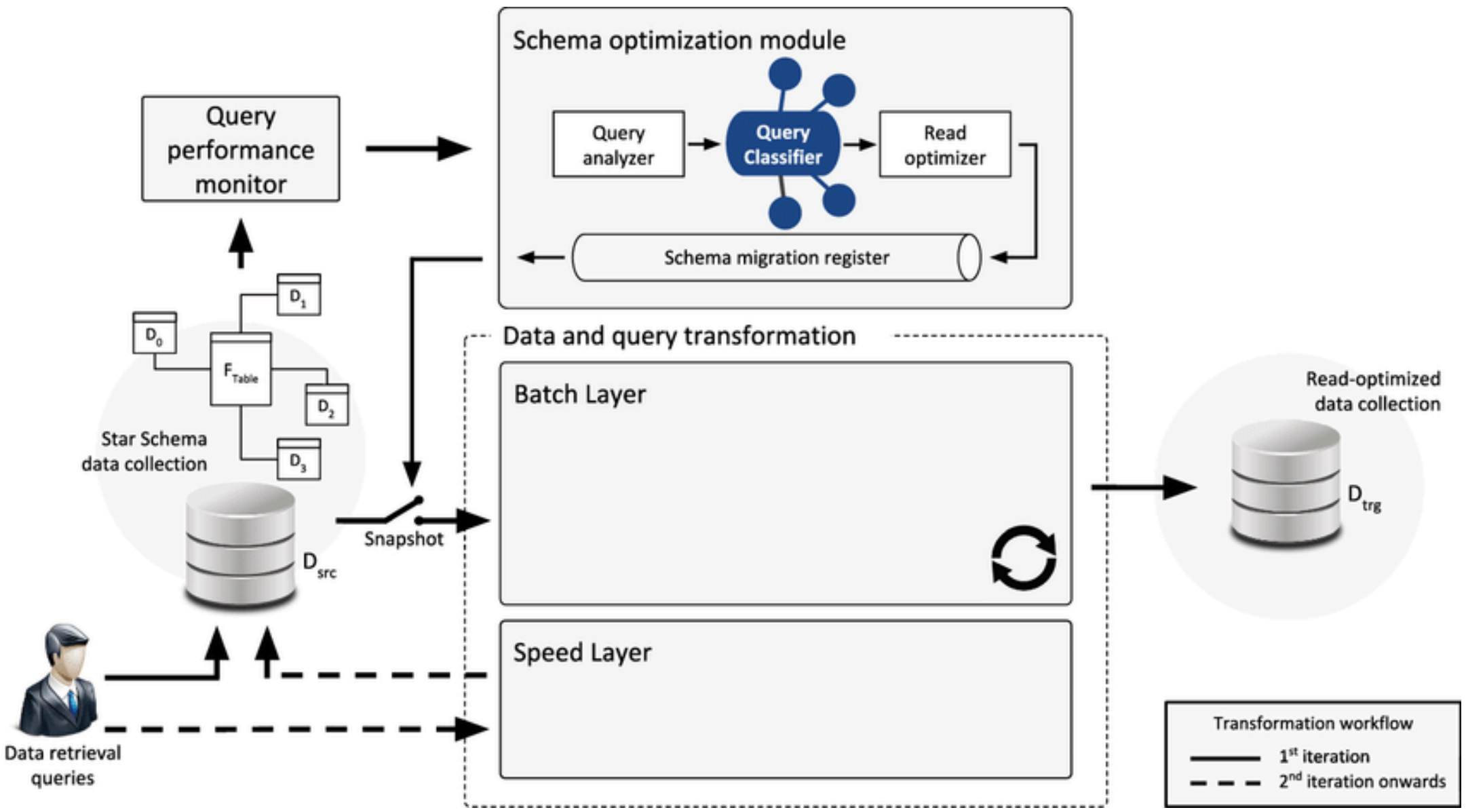
Begin with a pretrained model and fine-tune it using domain-specific datasets. Use frameworks like Hugging Face Transformers, PyTorch, or TensorFlow. Adjust only the task-relevant layers or employ techniques like transfer learning to minimize compute usage.

It avoids the cost and time of training a model from scratch while allowing precision tuning for better results in specialized use cases.

*This is meant to give just overview

5. DATA OPTIMIZATION

Data optimization focuses on cleaning, augmenting, and improving training datasets. This ensures only the most relevant, high-quality data is used for training, reducing redundancy.



Source

How it's done*:

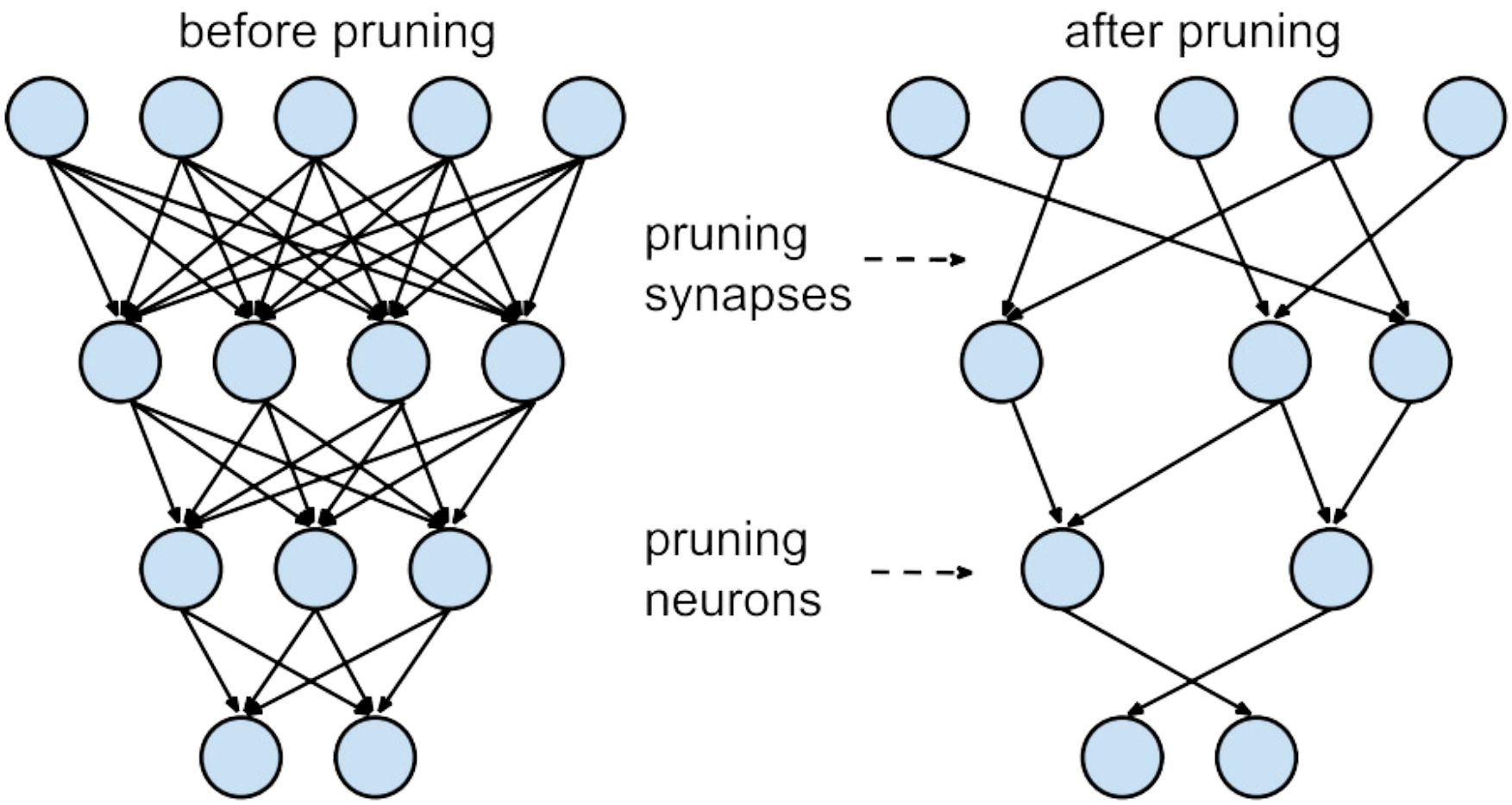
Curate high-quality datasets by cleaning, augmenting, or deduplicating the data. Tools like Label Studio, Snorkel, and dataset-specific preprocessing pipelines ensure efficient and accurate training data.

High-quality datasets enhance model performance with fewer training iterations, cutting down computational costs and training time.

*This is meant to give just overview

6. PRUNING

Pruning removes unnecessary neurons or weights from a trained model to streamline its architecture. This involves identifying and eliminating less impactful parameters while maintaining accuracy.



Source

How it's done*:

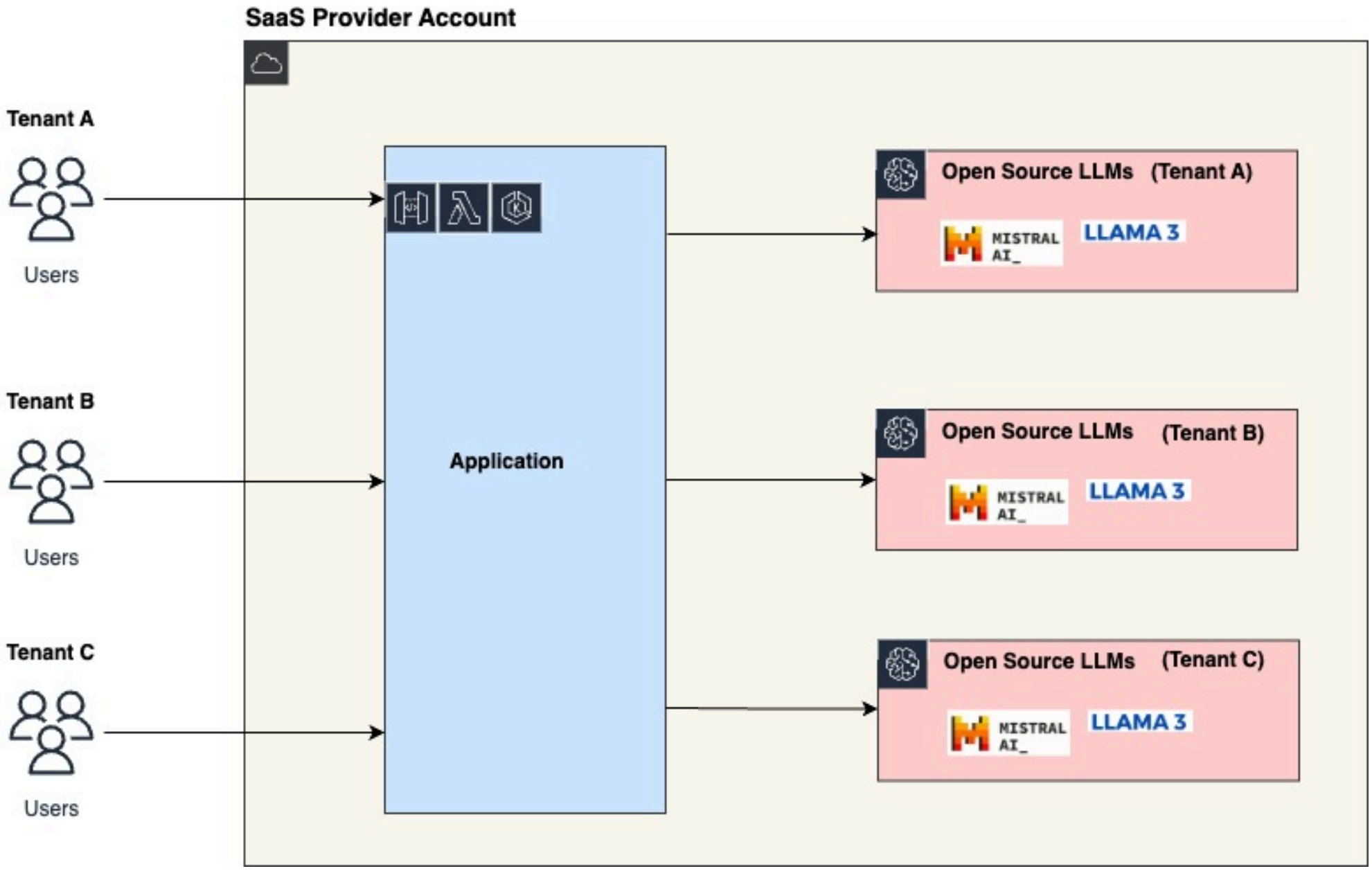
Analyze the model's parameters to identify those with low weights or minimal contribution. Methods like magnitude pruning or structured pruning are applied, followed by retraining to recover lost accuracy.

It significantly reduces the model's size and inference time, leading to lower operational costs for real-time and large-scale applications.

*This is meant to give just overview

7. MULTI-TENANCY

Multi-tenancy allows a single model instance to serve multiple users or applications simultaneously by efficiently managing workloads.



Source

How it's done*:

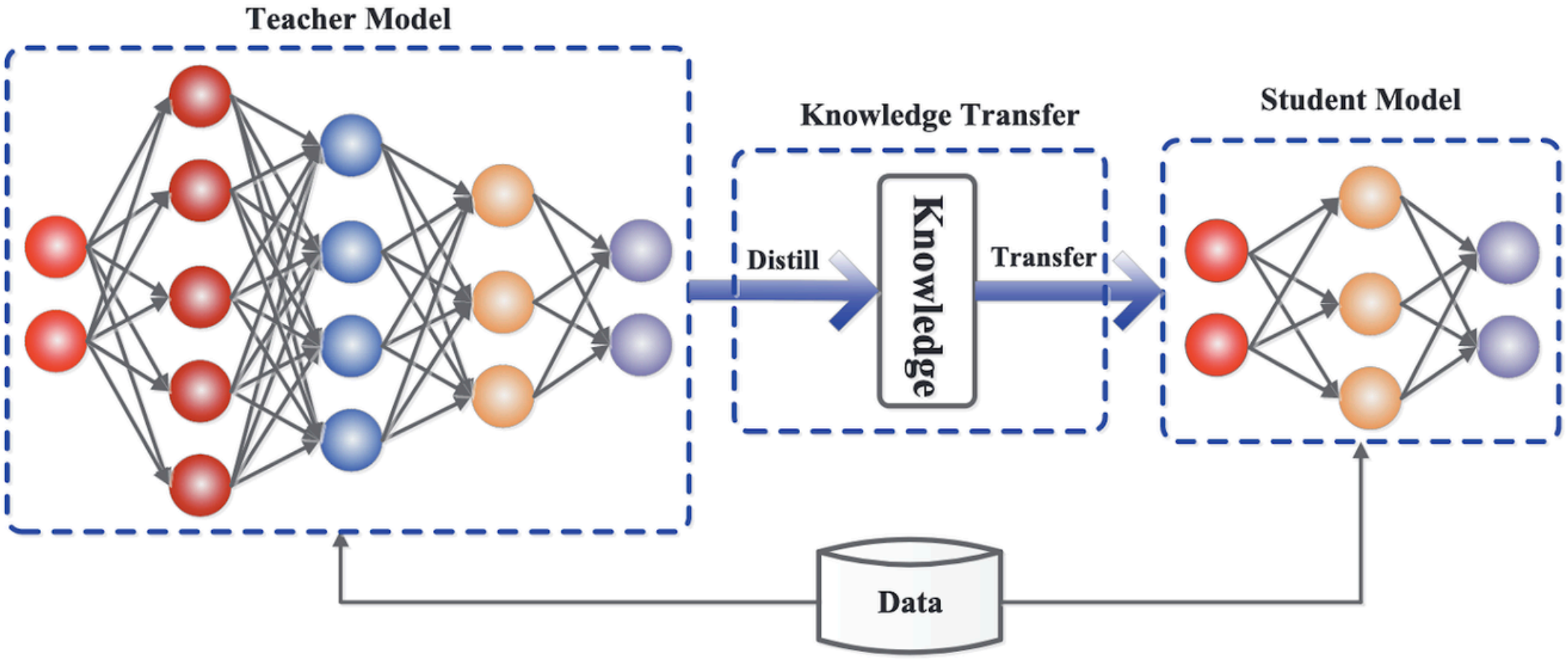
Deploy the model as a shared service with workload management. Use orchestration tools like Kubernetes or frameworks like TensorFlow Serving to handle concurrent requests from multiple applications efficiently.

It ensures maximum utilization of resources, reduces idle time, and lowers operational costs for applications requiring shared infrastructure.

*This is meant to give just overview

8. DISTILLATION

Model distillation transfers the knowledge from a large, complex model (teacher) to a smaller, simpler model (student). The student learns to replicate the teacher's predictions while being significantly smaller in size.



Source

How it's done*:

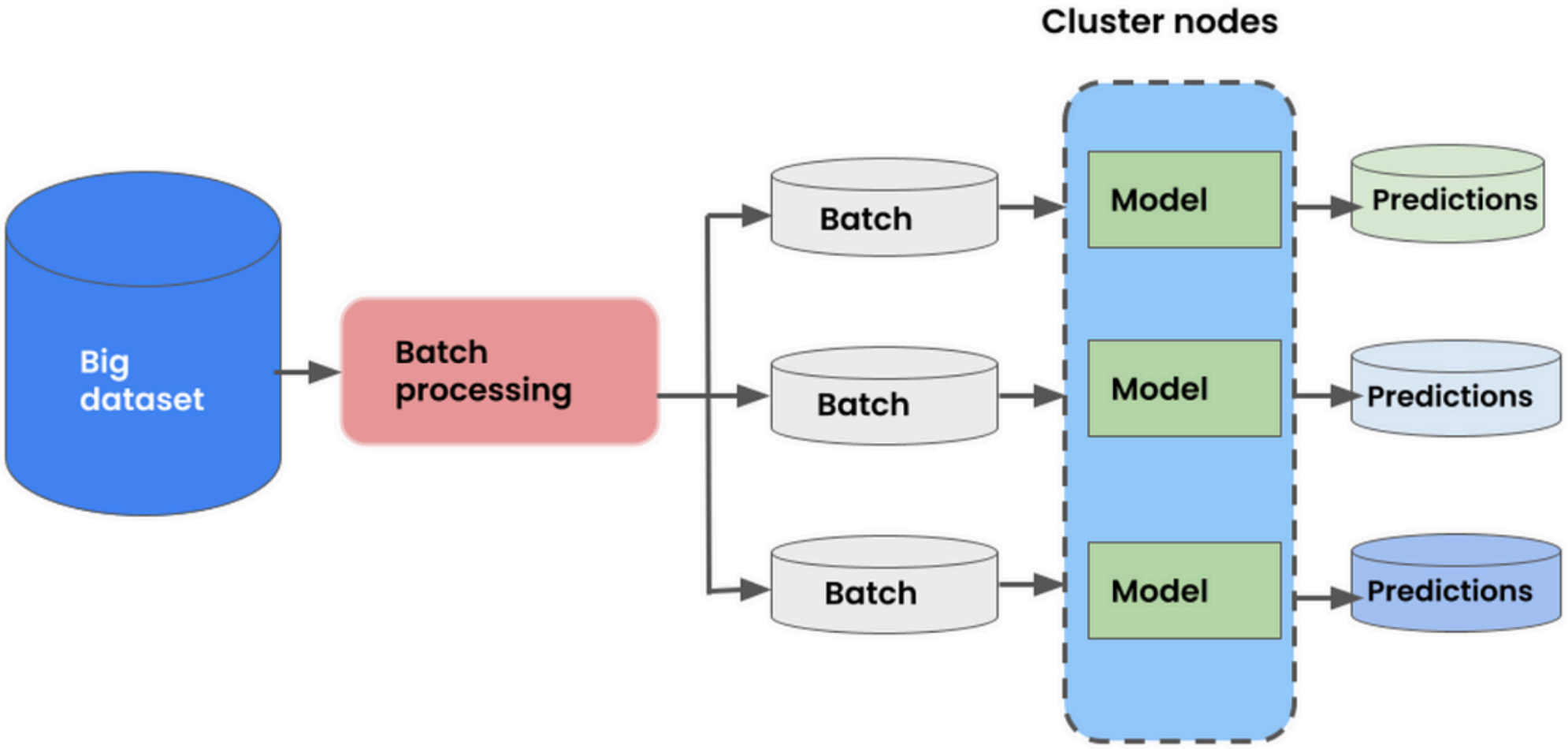
Train a smaller "student" model using the soft outputs (logits) from a larger "teacher" model as labels. This mimics the teacher's behavior while optimizing for smaller architecture. Tools like Hugging Face or custom scripts facilitate this.

Distillation provides faster inference and lower deployment costs, enabling lightweight models to perform at near-teacher levels.

*This is meant to give just overview

9. BATCHING

Batching combines multiple requests or data points into a single operation during inference or training. Instead of processing requests one by one, they are grouped to maximize hardware utilization.



Source

How it's done*:

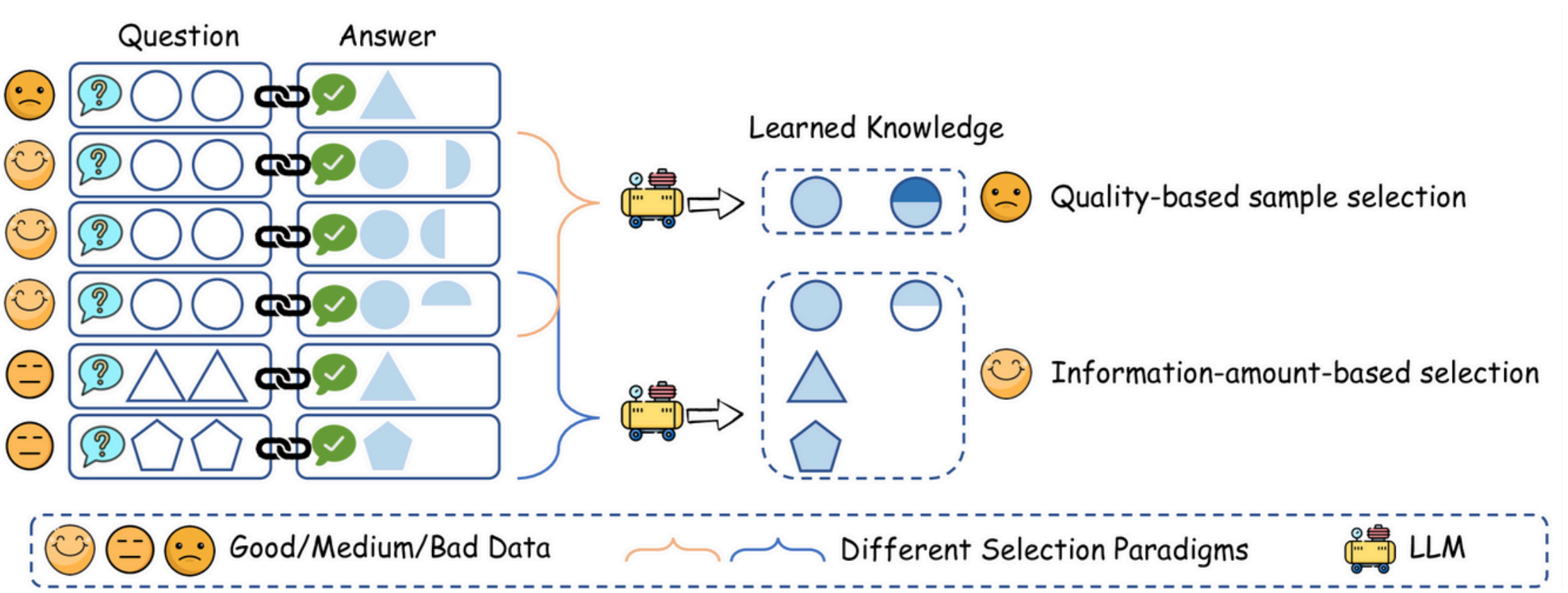
Group requests with similar computational needs into batches and process them together using frameworks like TensorFlow, PyTorch, or custom batching logic in APIs. Batch sizes are tuned to balance latency and throughput.

Batching increases computational efficiency, reduces idle time for GPUs/CPUs, and significantly lowers operational costs, especially for high-volume applications.

*This is meant to give just overview

10. MODEL COMPRESSION

Model compression reduces the size of large language models by eliminating redundancy or reducing complexity. Techniques include pruning, quantization, or knowledge distillation.



Source

How it's done*:

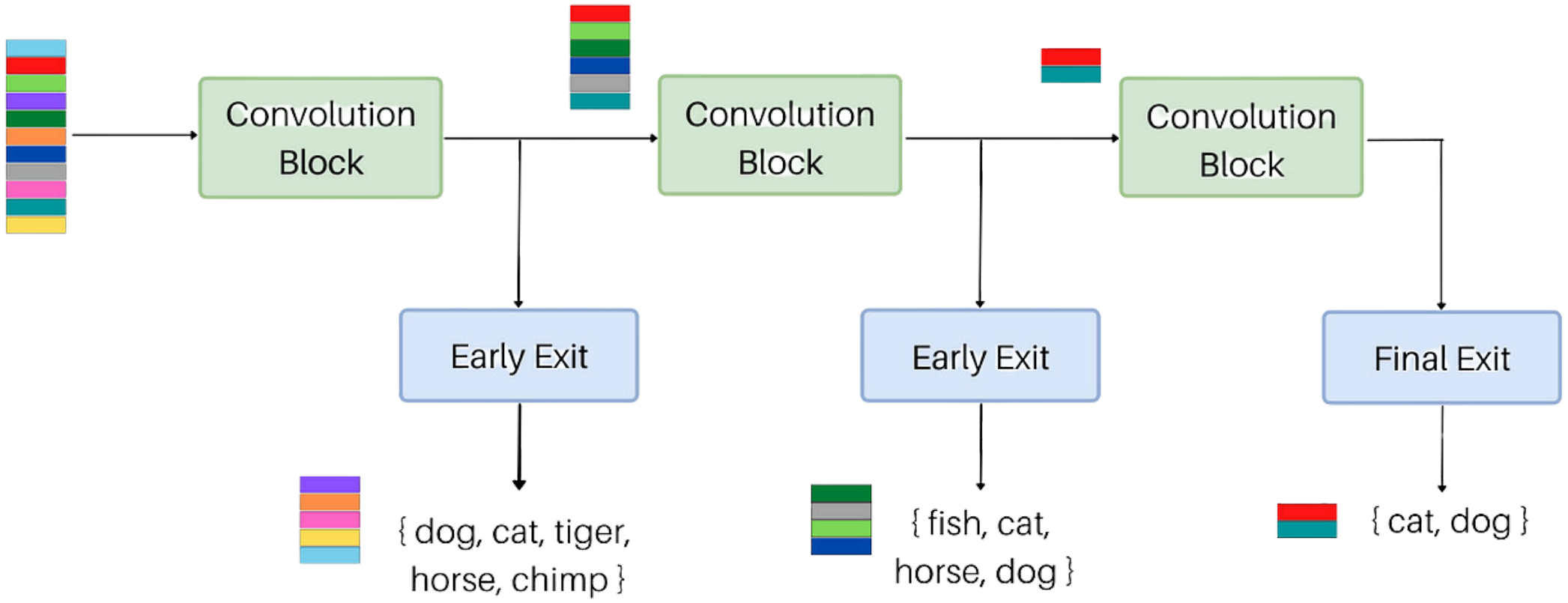
Apply pruning to remove unnecessary parameters, quantization to reduce data precision, or distillation to create smaller, student models that replicate the performance of larger models. Use tools like TensorFlow Lite or ONNX for implementation.

It lowers memory requirements, accelerates inference, and reduces costs for storage and compute while maintaining performance in resource-constrained environments.

*This is meant to give just overview

11. EARLY EXITING

Early exiting allows a model to terminate computations once sufficient confidence in the result is achieved, skipping further layers or operations.



Source

How it's done*:

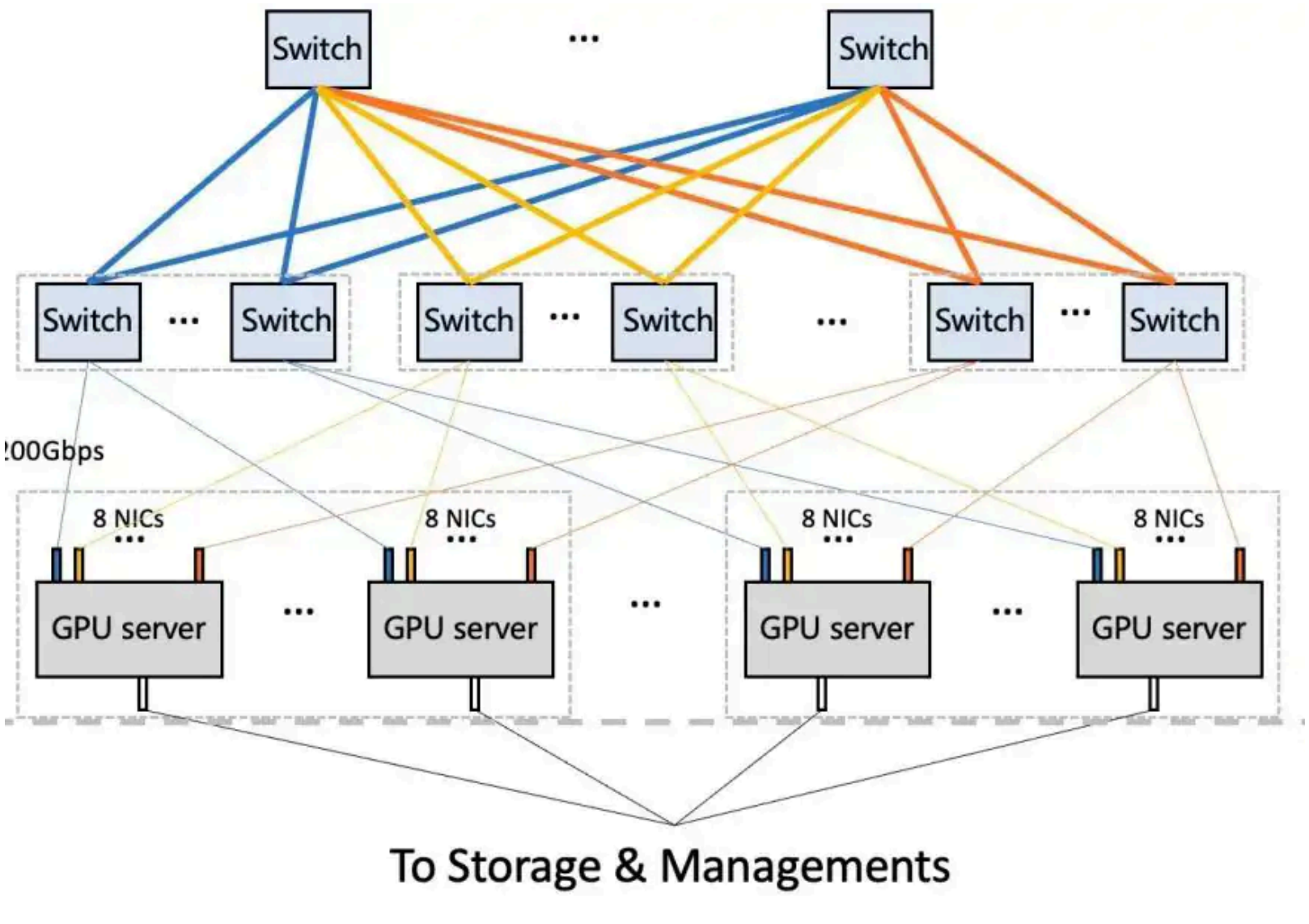
Introduce intermediate classifiers at various points in the model. If confidence surpasses a predefined threshold, the computation halts early, saving resources.

It avoids unnecessary computations for simpler inputs, reducing latency and compute costs while maintaining acceptable accuracy.

*This is meant to give just overview

12. OPTIMIZED HARDWARE

Optimized hardware refers to specialized devices, such as TPUs, GPUs, or custom ASICs, designed to handle AI workloads more efficiently than general-purpose processors.



Source: NADDOD

How it's done*:

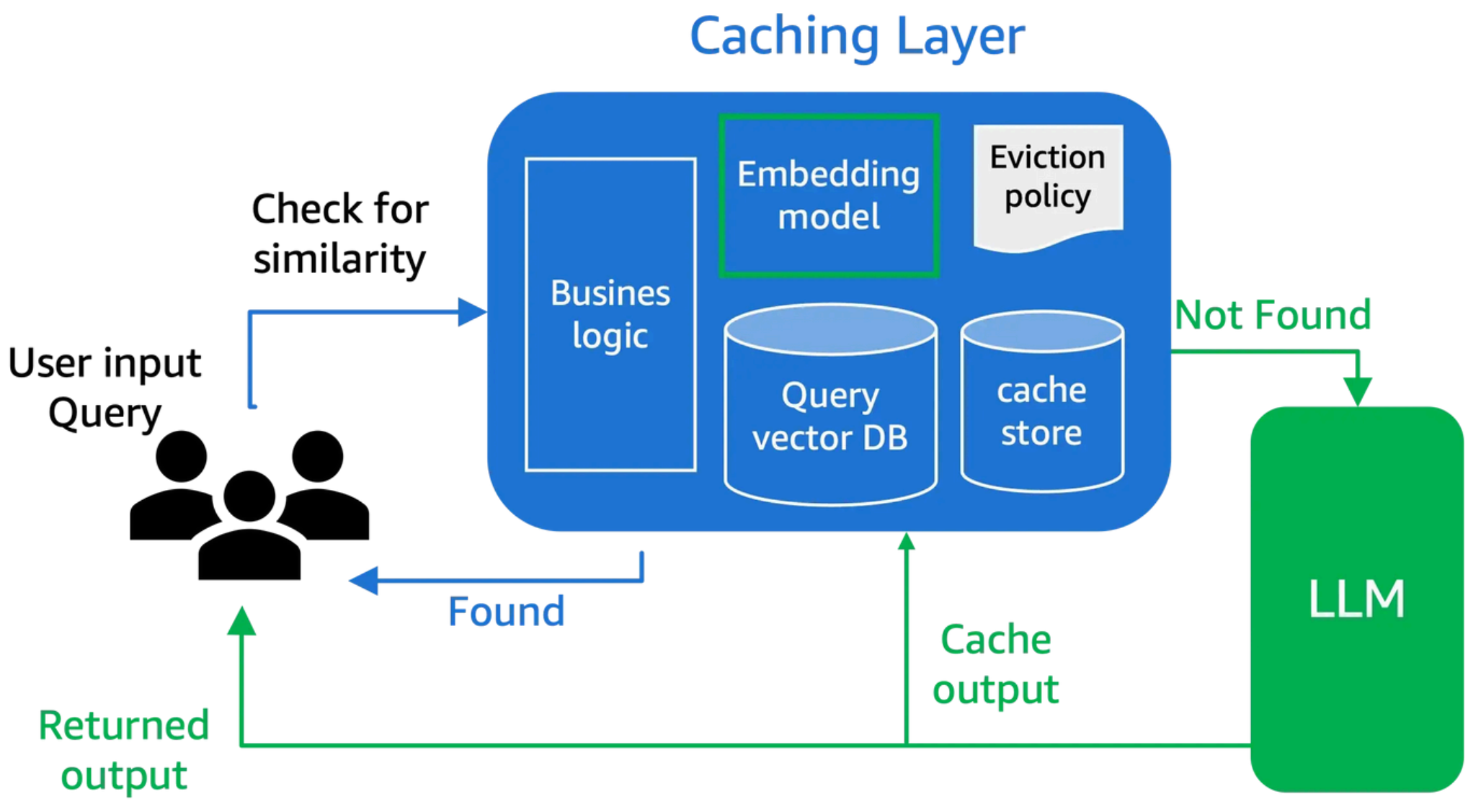
Select AI-dedicated hardware, such as NVIDIA A100 GPUs or Google TPUs, based on workload requirements. Leverage optimized libraries like CUDA or TensorFlow's XLA compiler for hardware-specific acceleration.

It provides faster computation, reduces energy consumption, and lowers long-term costs for training and inference in large-scale AI applications.

*This is meant to give just overview

13. CACHING

Caching stores the results of frequently used computations, such as model outputs, to avoid recalculating them repeatedly. This is especially beneficial for repetitive tasks or queries.



Source

How it's done*:

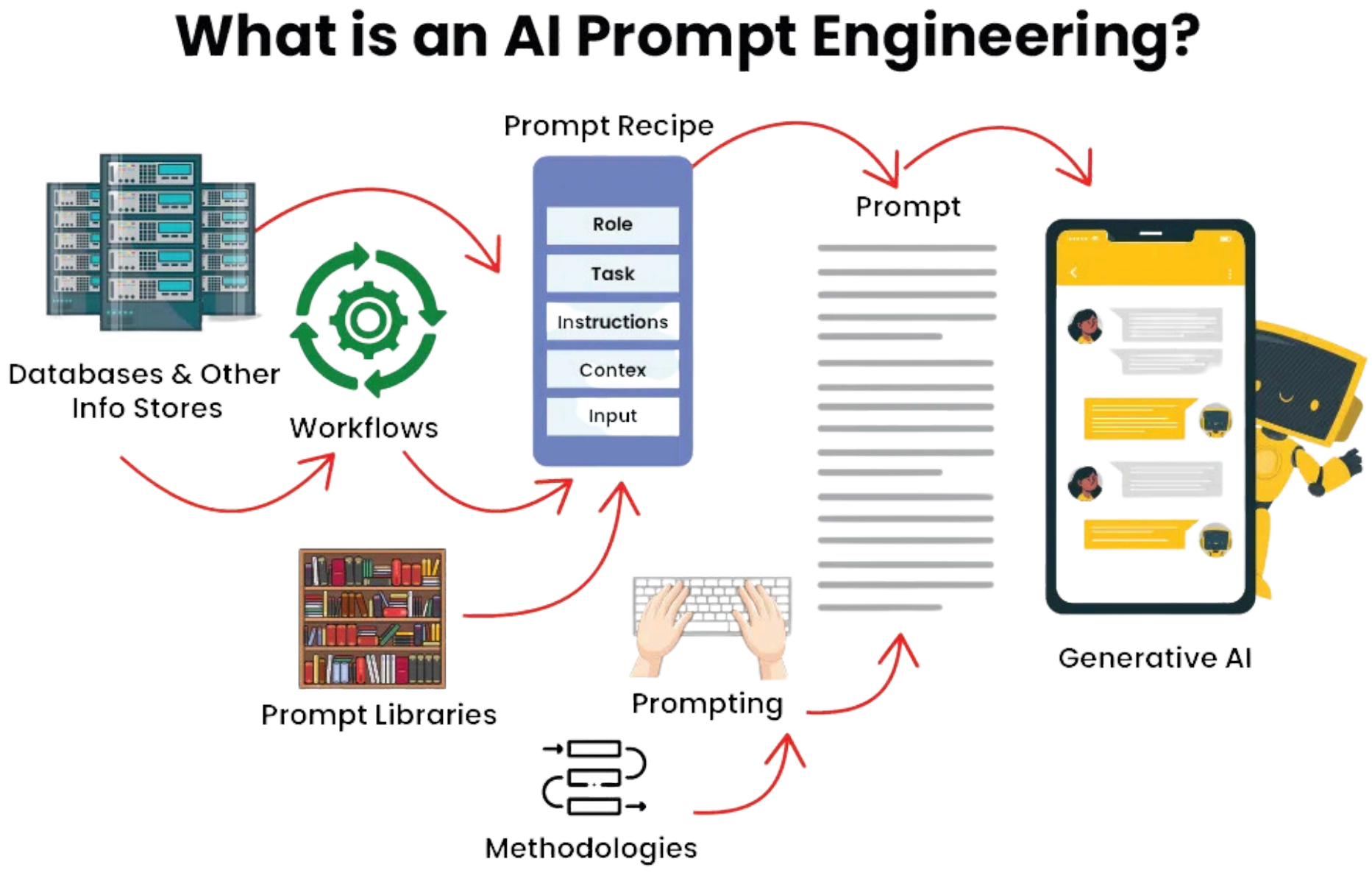
Implement caching mechanisms using tools like Redis or Memcached. Cache outputs at strategic points in the pipeline and retrieve them dynamically based on query patterns.

This saves computational resources, reduces latency, and cuts costs for applications with repetitive or similar queries.

*This is meant to give just overview

14. PROMPT ENGINEERING

Prompt engineering involves crafting input prompts in a way that guides the model to generate more accurate and efficient outputs with minimal computational effort.



Source

How it's done*:

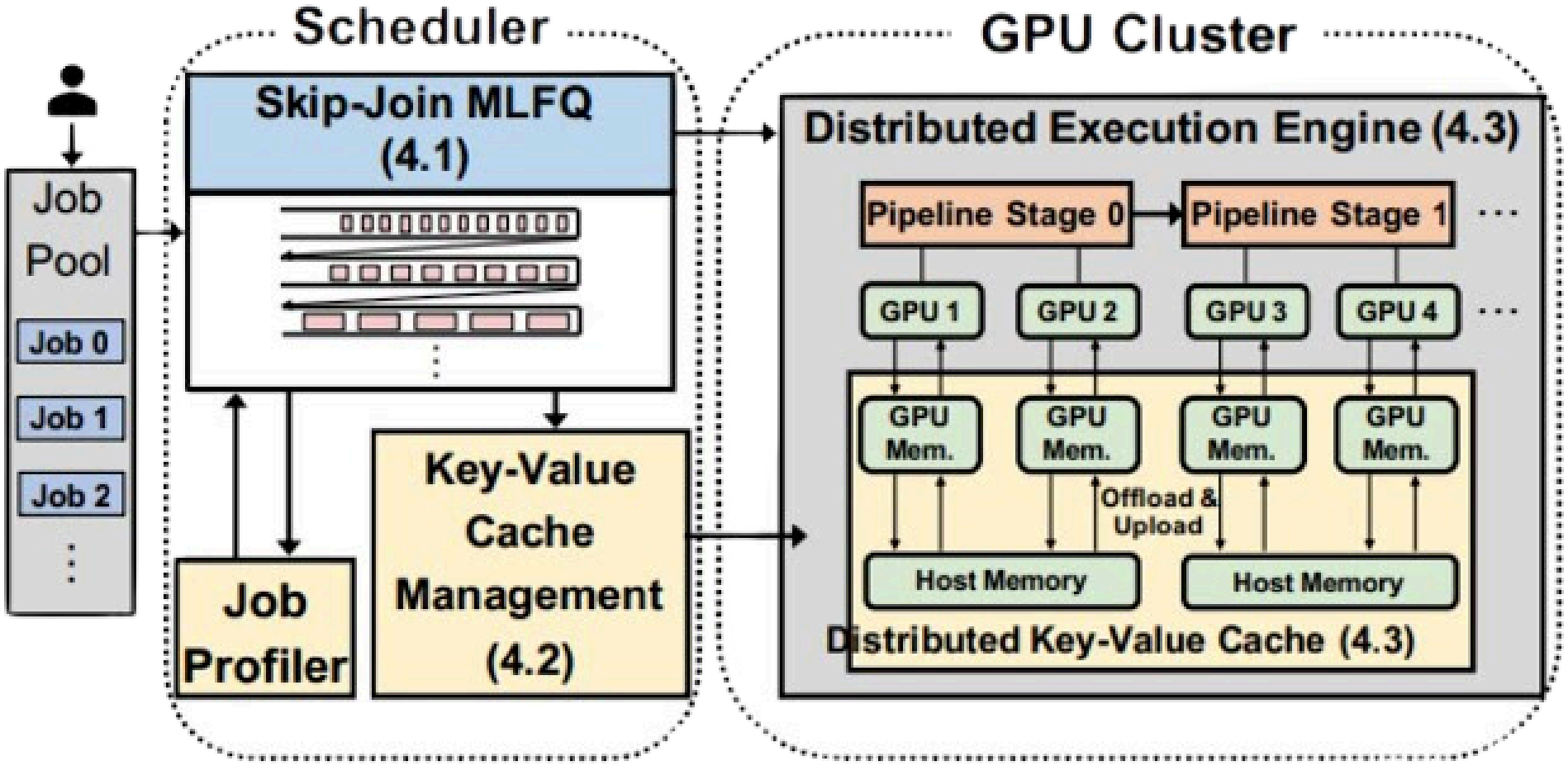
Design concise and specific prompts, use contextual keywords, and structure questions to align with the model's training. Test and iterate to optimize output quality and token usage.

Prompt engineering reduces token usage, optimizes API costs, and ensures high-quality results by minimizing the computational burden on the model.

*This is meant to give just overview

15. DISTRIBUTED INFERENCE

Distributed inference splits the workload of running a model across multiple machines or devices, enabling parallel computation and efficient scaling.



Source

How it's done*:

Divide the model into segments or replicate it across nodes. Use frameworks like Horovod or PyTorch Distributed to manage communication and workload distribution.

This enables low-latency inference for large models, reduces bottlenecks, and scales operations efficiently to handle high-volume workloads.

*This is meant to give just overview

Follow to stay updated on Generative AI

LIKE

COMMENT

REPOST