Lecture 9.3 - Scaling Inference and Deployment

Generative AI Teaching Kit

This lecture

- Inference of LLMs vs. Trainings LLMs
- Inference Optimization
- Production-Scale Inference
- End-to-End Deployment Optimizations

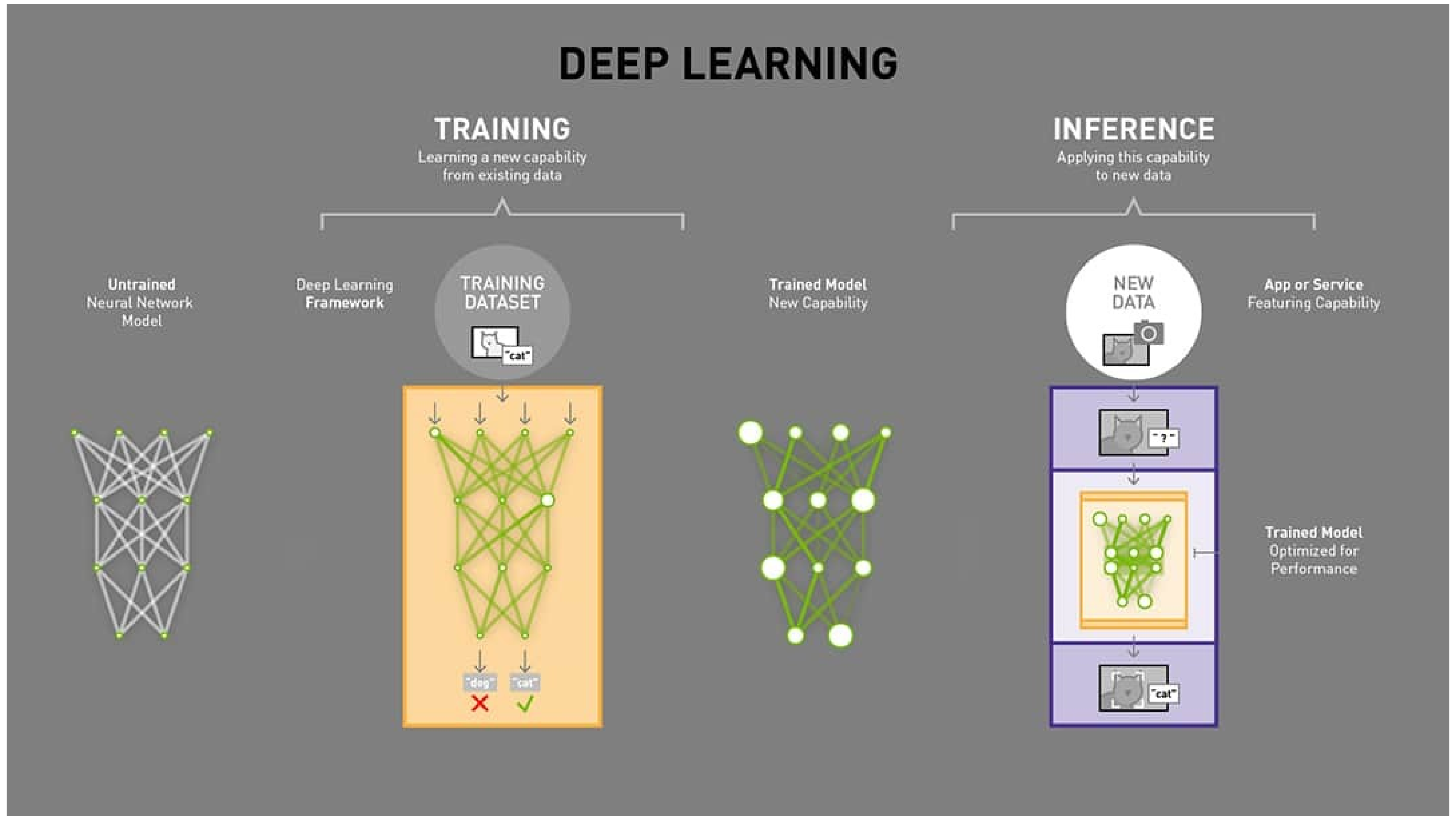
Inference vs. Training LLMs

Showing what you’ve learned

Training vs. Inference

Unlike training, **inference** is the process of using a trained model to make predictions or generate outputs for unseen data.

This is the “deployment” stage where models provide value, powering applications like chatbots, recommendation systems, and image generation.


DEEP LEARNING
TRAINING
Learning a new capability from existing data
INFERENCE
Applying this capability to new data
Untrained Neural Network Model
Deep Learning Framework
TRAINING DATASET
Trained Model New Capability
NEW DATA
App or Service Featuring Capability
"cat"
"dog"
"cat"
"?"
Trained Model Optimized for Performance
"cat"

Characteristics of Inference

- **Latency Sensitivity**:

Critical for applications like chatbots or live translation.

Performance is measured in milliseconds per query.

- **Resource Efficiency**:

Must run on diverse hardware, from high-end GPUs to edge devices.

Optimization often trades accuracy for speed.

- **Predictability**:

Unlike training, inference workloads are more predictable and involve fixed computations

Challenges in GenAI

Inference in GenAI is typically constrained by specific features related to how these models work and are used.

Token-by-Token Generation:

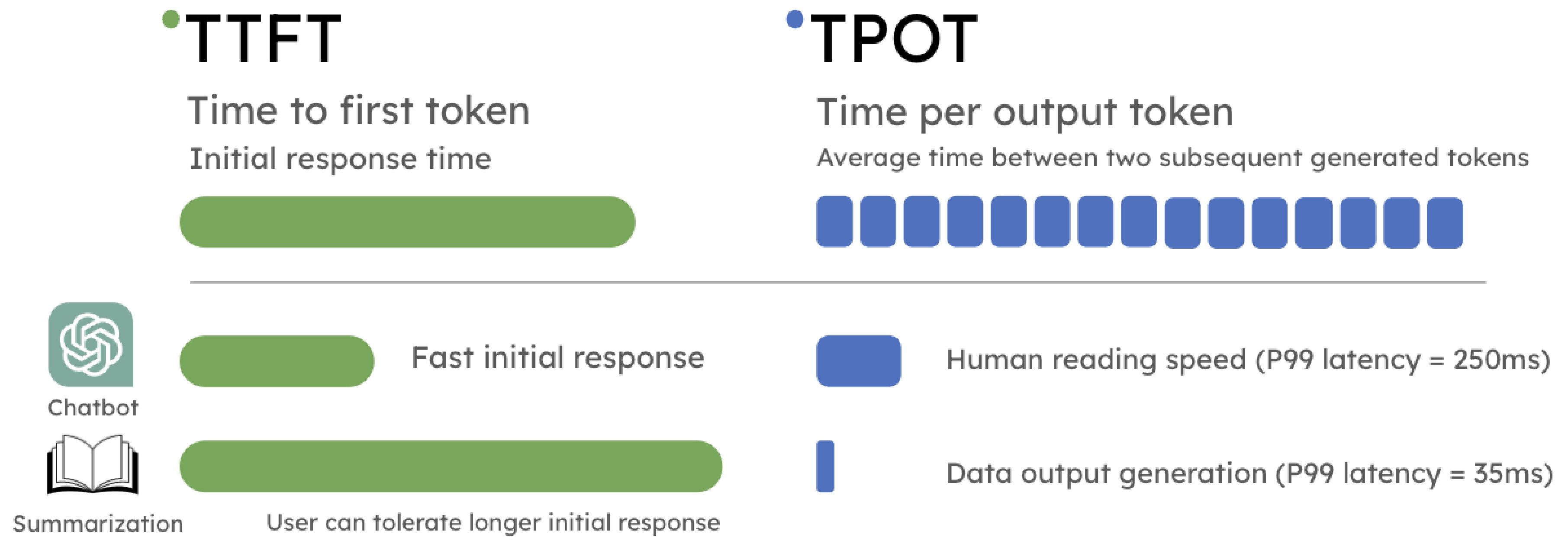
- Generative models (e.g., GPT) output text sequentially, requiring multiple forward passes.
- Latency scales with sequence length, making optimizations essential.

Memory Bottlenecks:

- Large models may exceed memory limits of inference hardware.
- Techniques like model sharding and offloading are used.

Concurrency:

- Handling simultaneous user requests efficiently.
- Scaling through distributed systems is crucial for high-demand applications.


TTFT
Time to first token
Initial response time
TPOT
Time per output token
Average time between two subsequent generated tokens
Chatbot
Fast initial response
Human reading speed (P99 latency = 250ms)
Summarization
User can tolerate longer initial response
Data output generation (P99 latency = 35ms)

Inference Optimization

Getting the most of our model

Why do we need to “optimize”?

Once training has finished and the model is performing its best at predicting an output, we can often benefit by optimizing the model for deployment.

Pushing the model to be as large as possible to achieve the training objectives allows us more room to compress the model without losing too much accuracy whilst improving:

Latency

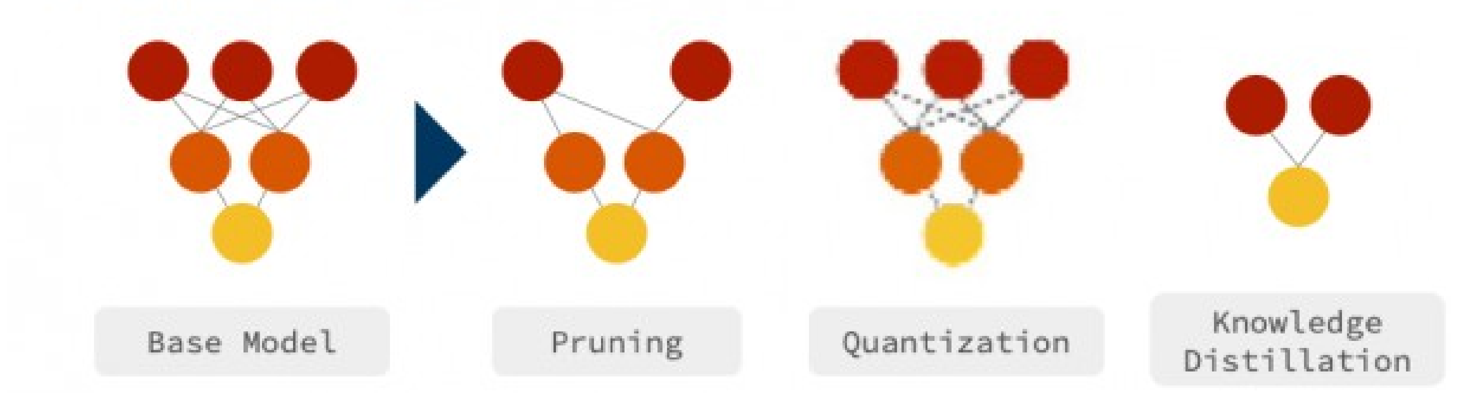
- The more neurons and layers in a model has increases the time it takes to calculate an output

Size

- The smaller a model can be, the more batches can be run and the cheaper a deployment can be on hardware

Throughput

- The combination of latency and size informs the total throughput of the model when deployed as smaller, faster models can process more data faster



Libraries for NN Optimization

To automate and standardize different techniques to optimize models for deployment, a number of popular libraries have been developed and adopted by the industry. These libraries focus on improving performance, scalability, and efficiency for inference, especially in distributed and real-time settings.

ONNX (Open Neural Network Exchange)

A standard format for representing machine learning models across different frameworks (e.g., PyTorch, TensorFlow).

TensorRT / LLM

NVIDIA's library for high-performance inference on NVIDIA GPUs, tailored for both general and large language models (LLMs).

vLLM

A specialized library designed for efficient and scalable inference of large language models.

Triton Inference Server

A scalable and production-ready inference server designed for hosting multiple models and managing inference pipelines.

ONNX

TensorRT

vLLM

NVIDIA
TRITON INFERENCE SERVER

NVIDIA TensorRT

TensorRT is NVIDIA's high-performance library designed for optimizing and deploying deep learning models for inference on NVIDIA GPUs.
The core idea of TensorRT is to take a trained model (e.g., from PyTorch or TensorFlow) and convert it into a highly optimized version for inference. This optimization process involves several steps:

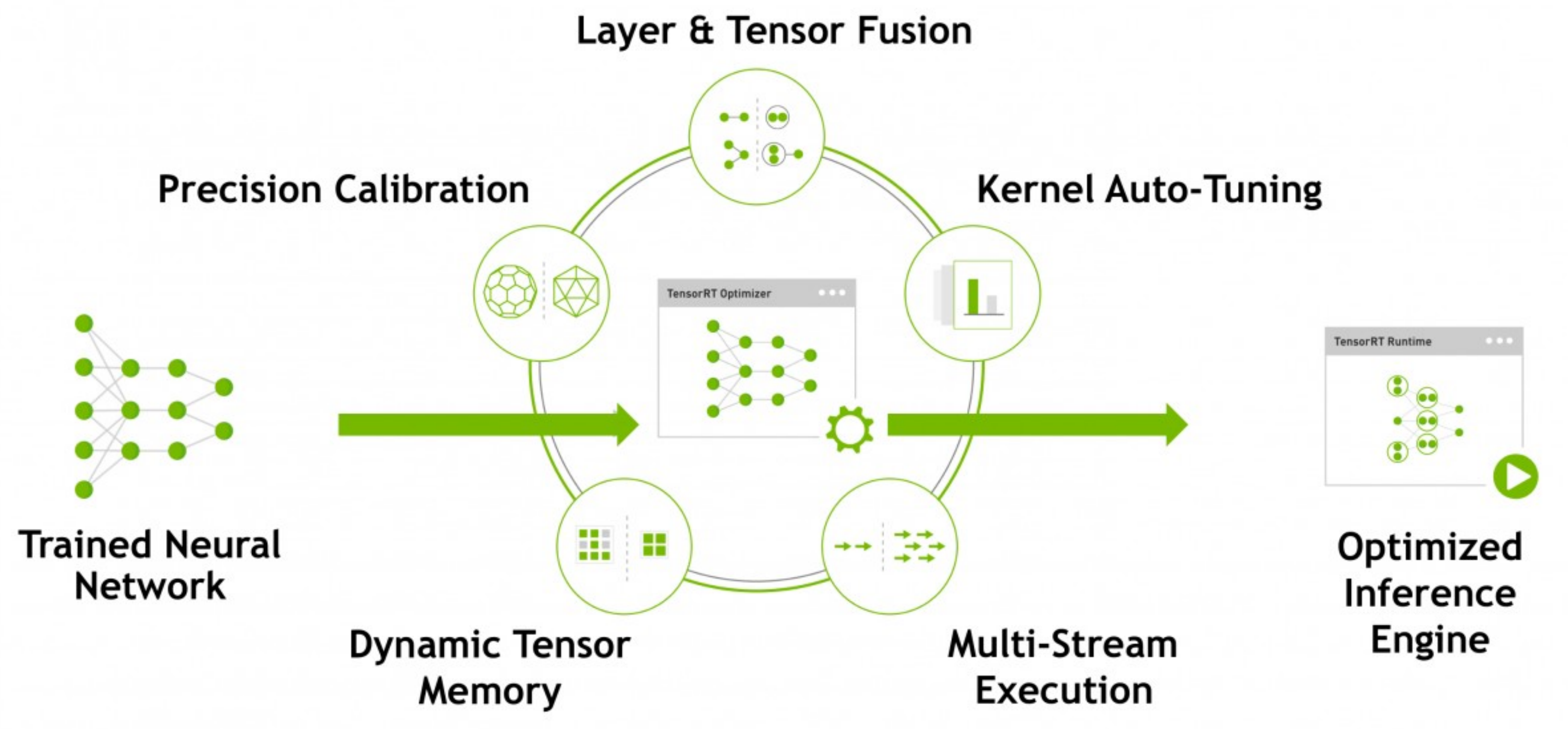
1. **Parsing**: TensorRT reads and parses the model definition, often in a format like ONNX or directly exported from a framework like PyTorch. This step ensures the model structure is understood.

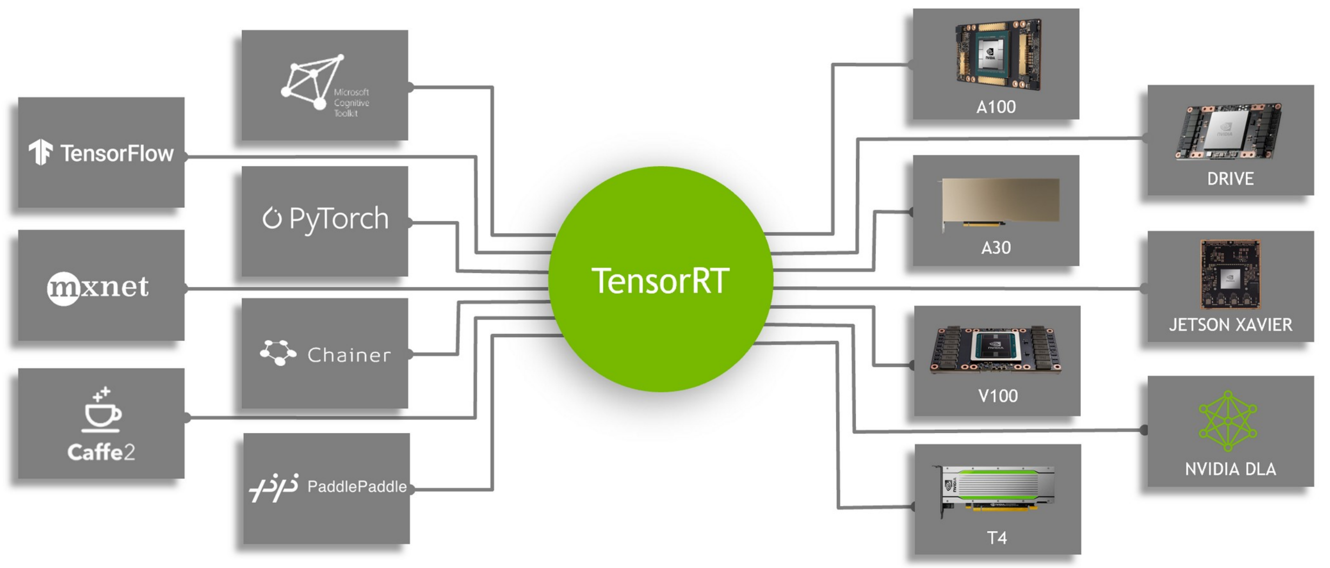
2. **Optimization**: TensorRT applies a series of performance enhancements. These include:
   - Layer fusion: Combining multiple operations into a single GPU kernel to reduce memory overhead.
   - Precision calibration: Reducing precision from FP32 to FP16 or INT8 to save memory and boost computation speed, with minimal impact on model accuracy.
   - Kernel auto-tuning: Selecting the most efficient GPU kernels for each operation.

3. **Serialization**: Once optimized, the model is serialized into an engine file, which is essentially a binary that can be directly executed by TensorRT. This serialized engine is specific to the GPU hardware it was optimized for.

4. **Inference Execution**: TensorRT provides APIs to load and run the optimized engine. In Python, the TensorRT Python API allows for easy integration into applications.

A typical workflow with TensorRT in Python might look like this:
- Convert your model (e.g., export to ONNX if using PyTorch).
- Use TensorRT's Python API to parse and optimize the model.
- Save the optimized engine.
- Load the engine for inference and integrate it into your application.





TensorRT-LLM

TensorRT-LLM is a specialized extension of TensorRT, optimized specifically for deploying large language models. It builds on TensorRT's general capabilities while introducing features and optimizations tailored to the unique demands of LLM inference.

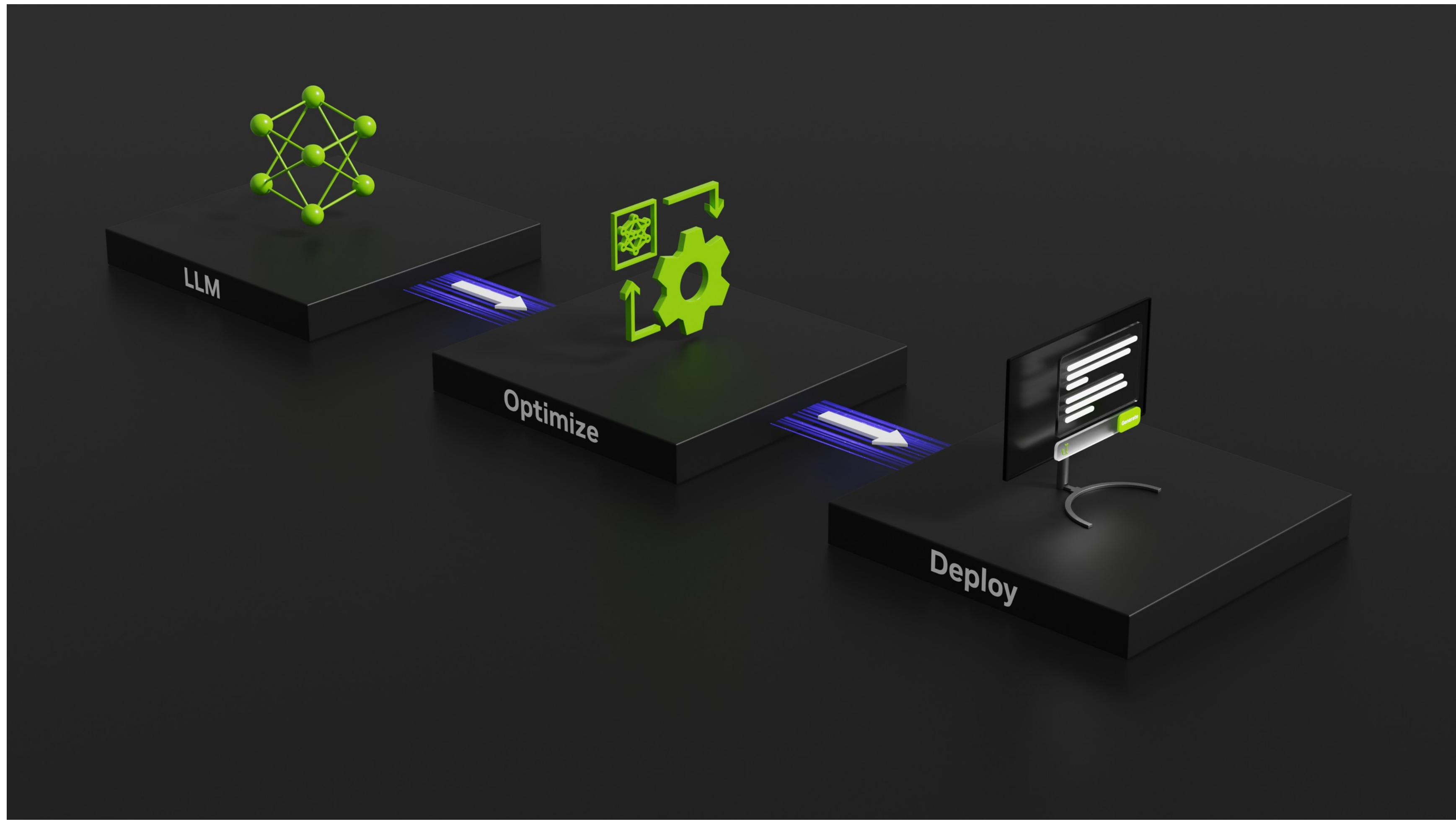
The primary focus of TensorRT-LLM is on maximizing the efficiency of token generation.

1. TensorParallel and Multi-GPU Support:
TensorRT-LLM enables seamless distribution of the model across multiple GPUs. It uses techniques like tensor parallelism.

2. Optimized Kernels for LLMs:
TensorRT-LLM provides custom kernels optimized for transformer-based architectures. These include improvements in self-attention computation, faster layer normalization, and fused operations that reduce memory access overhead.

3. Dynamic Sequence Length Management:
In LLM inference, sequences can vary in length, and managing these dynamically is important to minimize wasted computation. TensorRT-LLM supports efficient handling of dynamic batch sizes and sequence lengths, which is crucial for real-time generation tasks.

4. High Throughput and Low Latency:
With support for mixed precision (FP16 and INT8), TensorRT-LLM achieves significant reductions in inference latency while maintaining model accuracy. This makes it ideal for applications like chatbots or content generation, where response time is critical.

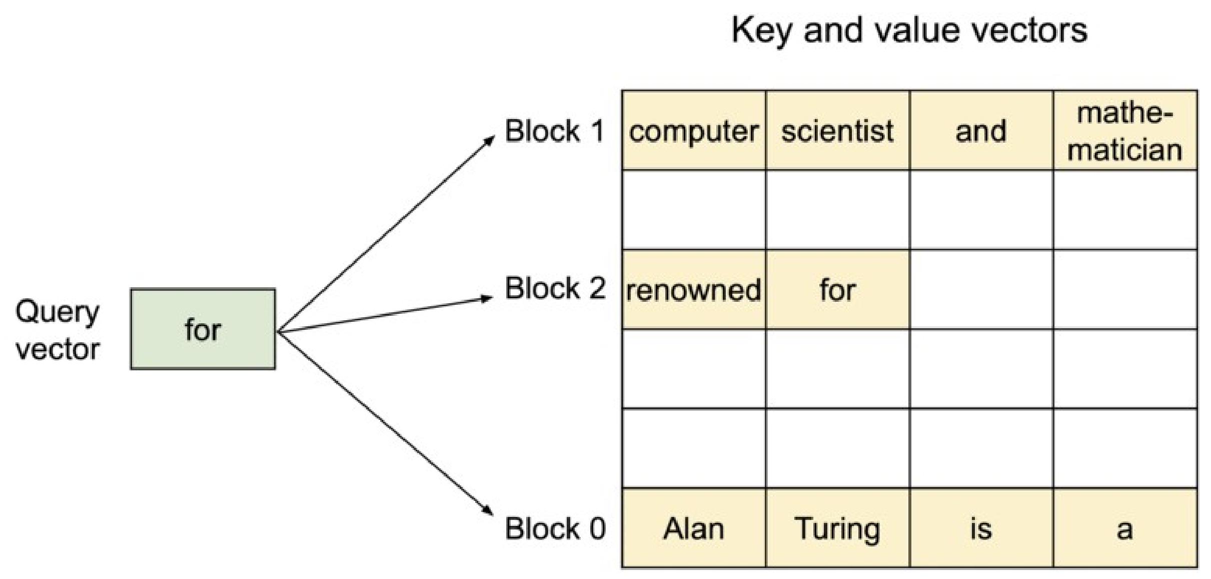


vLLM

vLLM is a library designed specifically for efficient and scalable inference of large language models (LLMs). It focuses on optimizing token generation and maximizing GPU utilization to handle real-world, high-demand use cases.

One of vLLM's standout features is its state management system. Unlike traditional LLM inference, where past key-value pairs (used for attention mechanisms) are recomputed or inefficiently managed, vLLM introduces a high-performance caching mechanism.

vLLM is particularly effective in handling concurrent requests. It uses dynamic batching to combine multiple inference requests into a single GPU operation, maximizing throughput while maintaining low latency.



PagedAttention

Production-Scale Inference

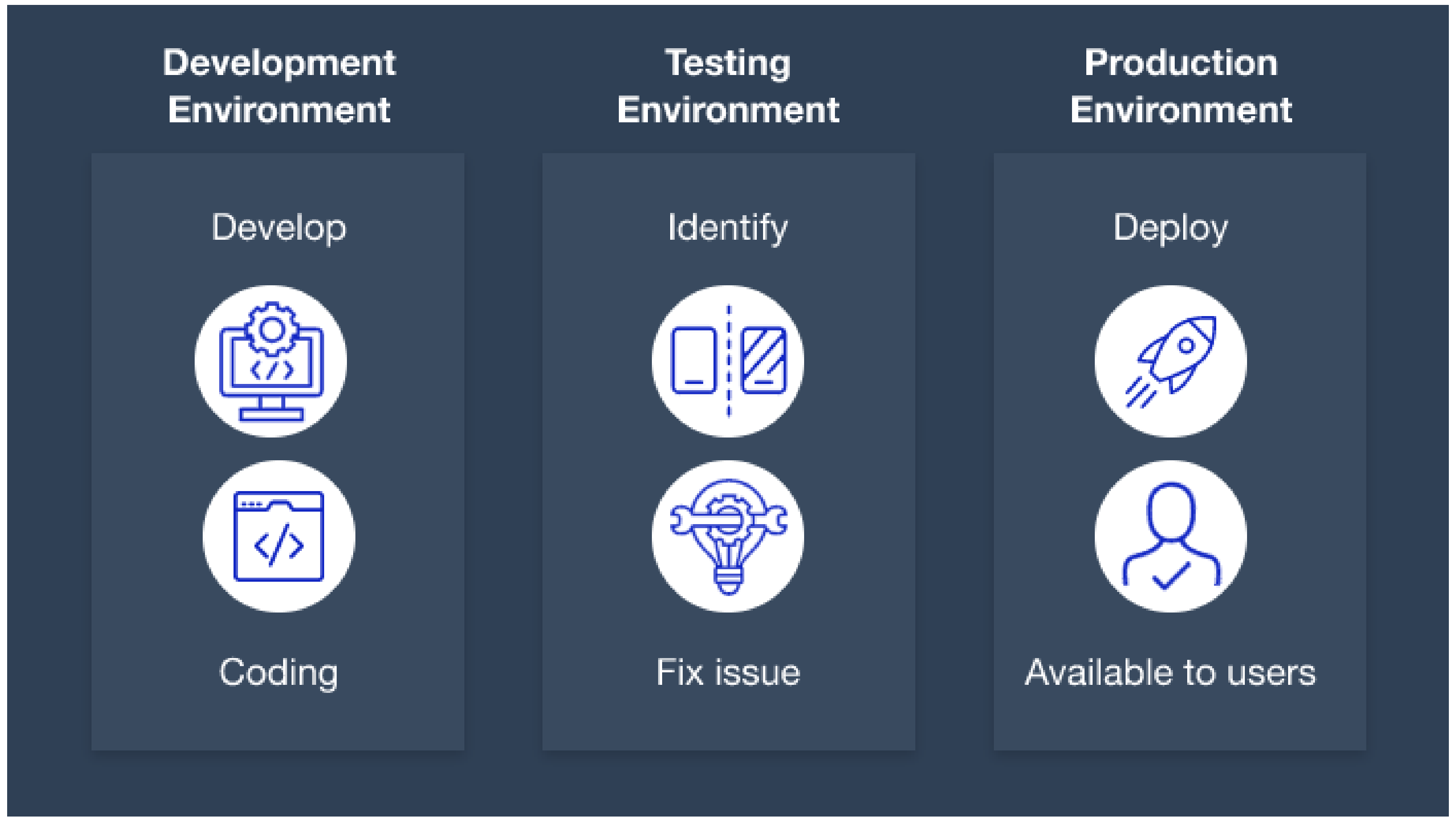
Connecting text to data

Development vs. Production-scale Inference Challenges

Development and Production environments present different challenges

Development Scale:
- Single-user or small-team testing.
- Focused on correctness, model debugging, and small data batches.
- Less emphasis on latency and resource utilization.

Production Scale:
- Multi-user, high-demand scenarios with concurrent requests.
- Prioritizes low latency, high throughput, and cost efficiency.
- Involves dynamic scaling and robust fault tolerance.



Load Balancing

What is Load Balancing?

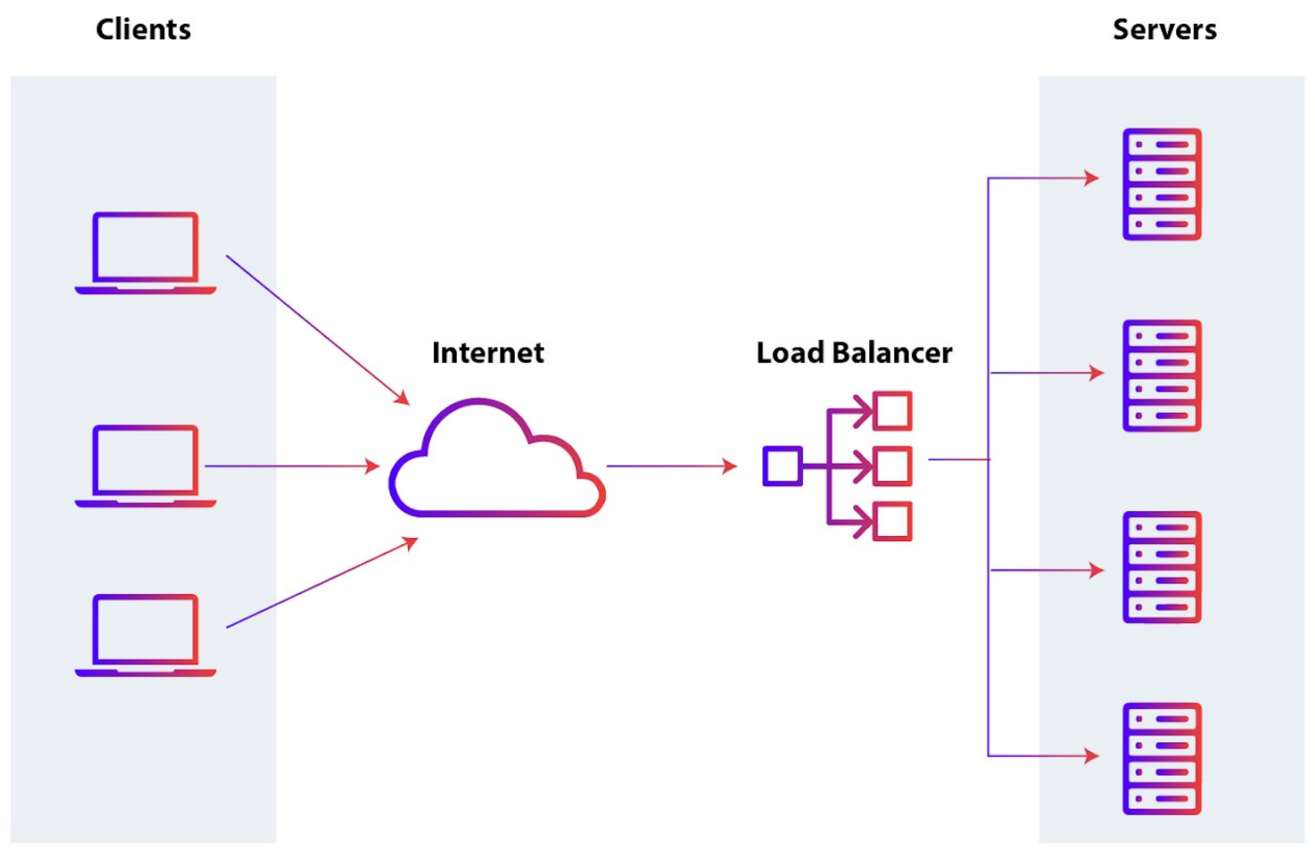
- Even distribution of inference requests across servers or GPUs.
- Prevents bottlenecks and ensures high availability.

Strategies:

1. Round Robin: Requests are assigned sequentially across nodes.
2. Weighted Distribution: Prioritizes more powerful nodes for heavier loads.
3. Dynamic Balancing: Adapts to workload changes in real time.

Tools:

- Kubernetes for horizontal scaling.
- Triton Inference Server with dynamic batching support.



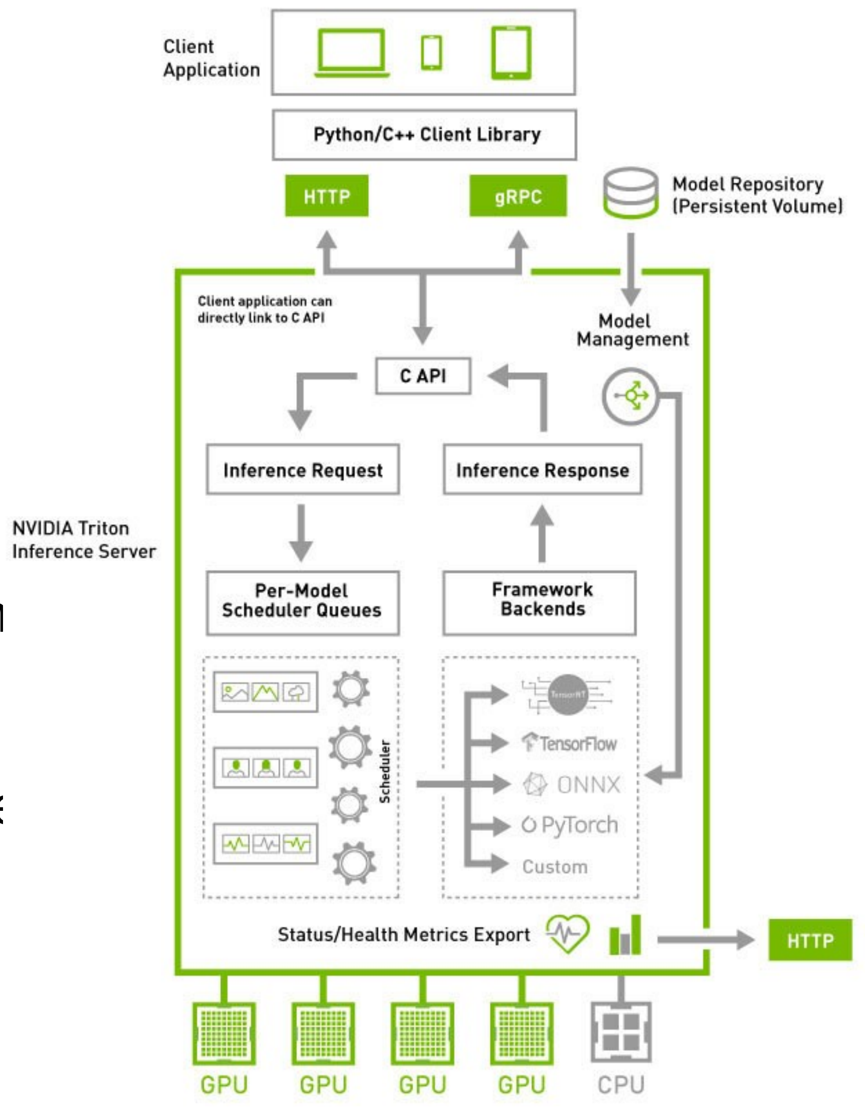
Triton Inference Server

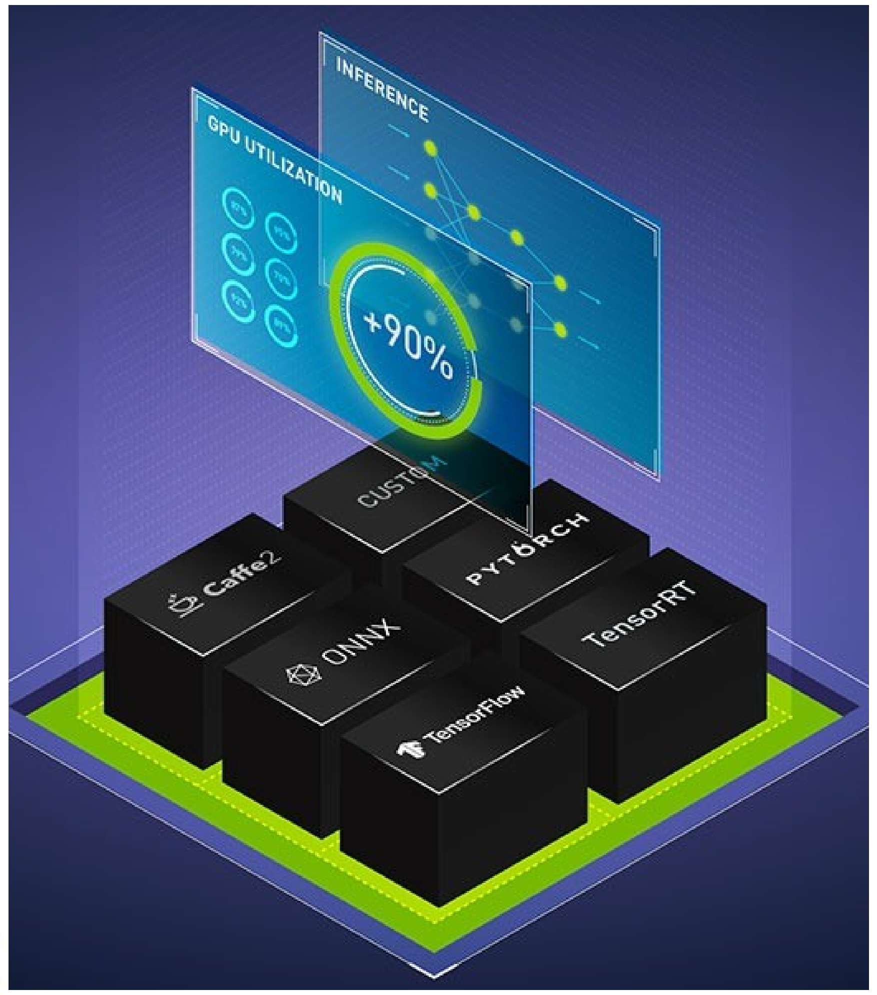
Triton Inference Server is a production-ready platform for deploying and serving
deep learning models at scale. It is designed to handle the complexities of inference in distributed environments, offering flexibility, scalability, and high performance.

A key feature of Triton is **dynamic batching.** Unlike static batching, where requests must arrive together to be processed in a batch, dynamic batching aggregates requests in real time. This ensures high GPU utilization without compromising latency for smaller requests.

In distributed setups, Triton works well with container orchestration tools like Kubernetes. It enables horizontal scaling by deploying multiple instances, with load balancing to manage traffic efficiently.

Triton Inference Server is particularly well-suited for GenAI applications, wher token-by-token generation and high concurrency are common. Its dynamic batching, multi-model support, and optimized GPU usage make it a go-to solution for serving deep learning models in production.


Client Application
Python/C++ Client Library
HTTP
gRPC
Model Repository (Persistent Volume)
Client application can directly link to C API
Model Management
C API
Inference Request
Inference Response
NVIDIA Triton Inference Server
Per-Model Scheduler Queues
Framework Backends
Scheduler
TensorFlow
ONNX
PyTorch
Custom
Status/Health Metrics Export
HTTP
GPU
GPU
GPU
GPU
CPU

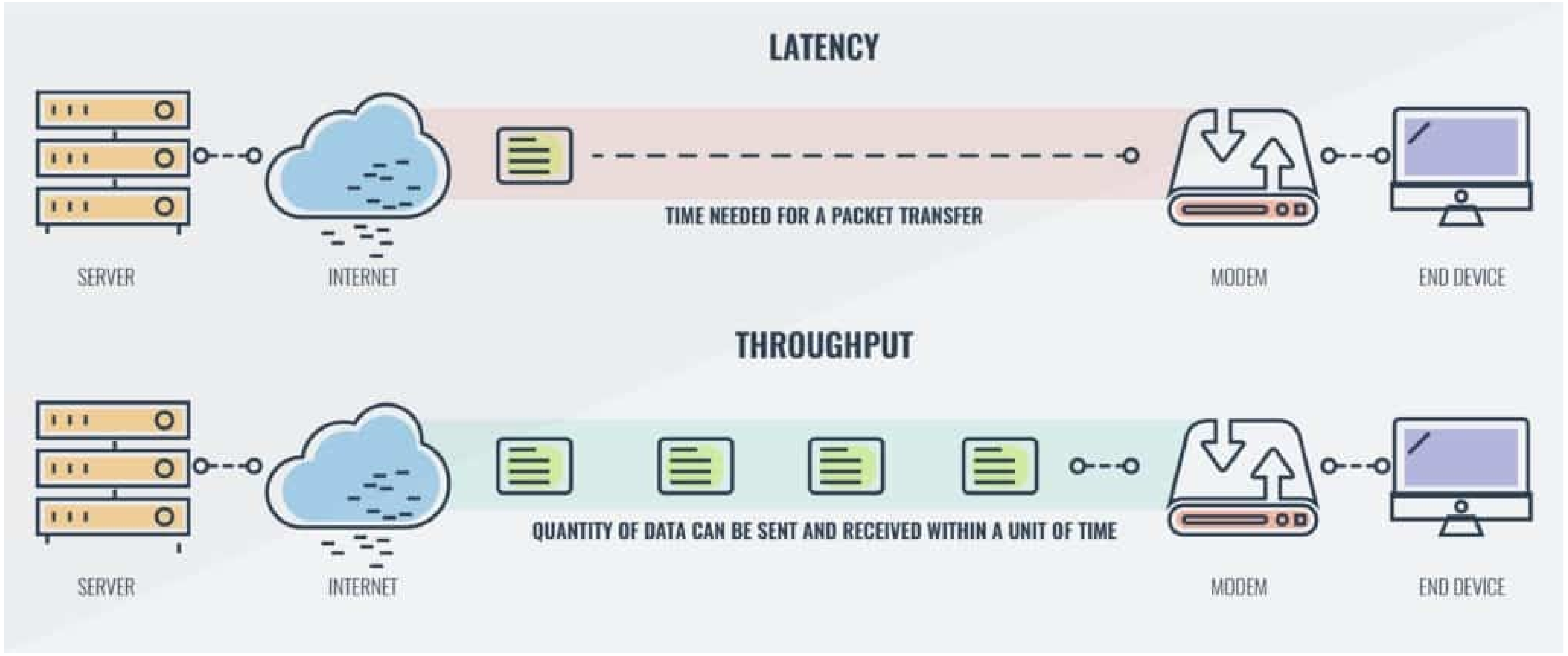


Latency vs. Throughput

Latency:
Definition: Time taken for a single request to be processed.
Key metric for real-time applications (e.g., chatbots, live translation).
Prioritized in low-concurrency, interactive scenarios.

Throughput:
Definition: Total number of requests processed per second.
Key metric for batch processing (e.g., recommendation systems).
Prioritized in high-concurrency or backend pipelines.

Trade-offs:
Increasing throughput often increases latency (e.g., batching delays).
Balancing depends on the application (e.g., GenAI text generation often prioritizes latency).


LATENCY
TIME NEEDED FOR A PACKET TRANSFER
SERVER
INTERNET
MODEM
END DEVICE
THROUGHPUT
QUANTITY OF DATA CAN BE SENT AND RECEIVED WITHIN A UNIT OF TIME
SERVER
INTERNET
MODEM
END DEVICE

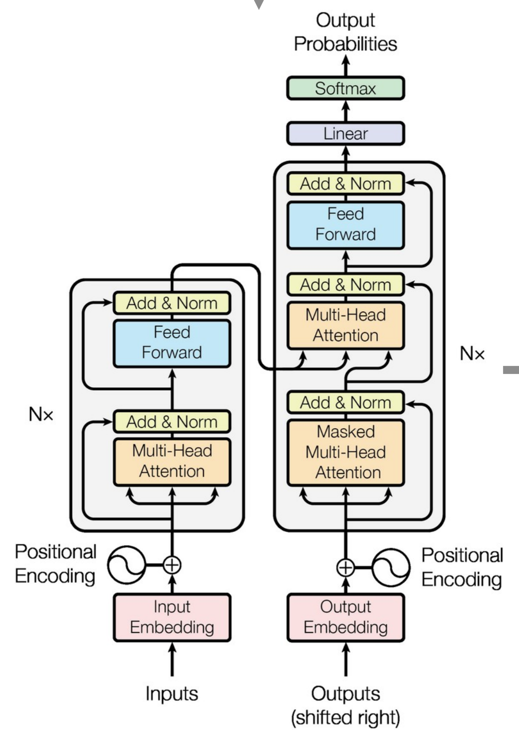
End-to-End Deployment with TensorRT-LLM

Putting everything together

Life Cycle of an LLM

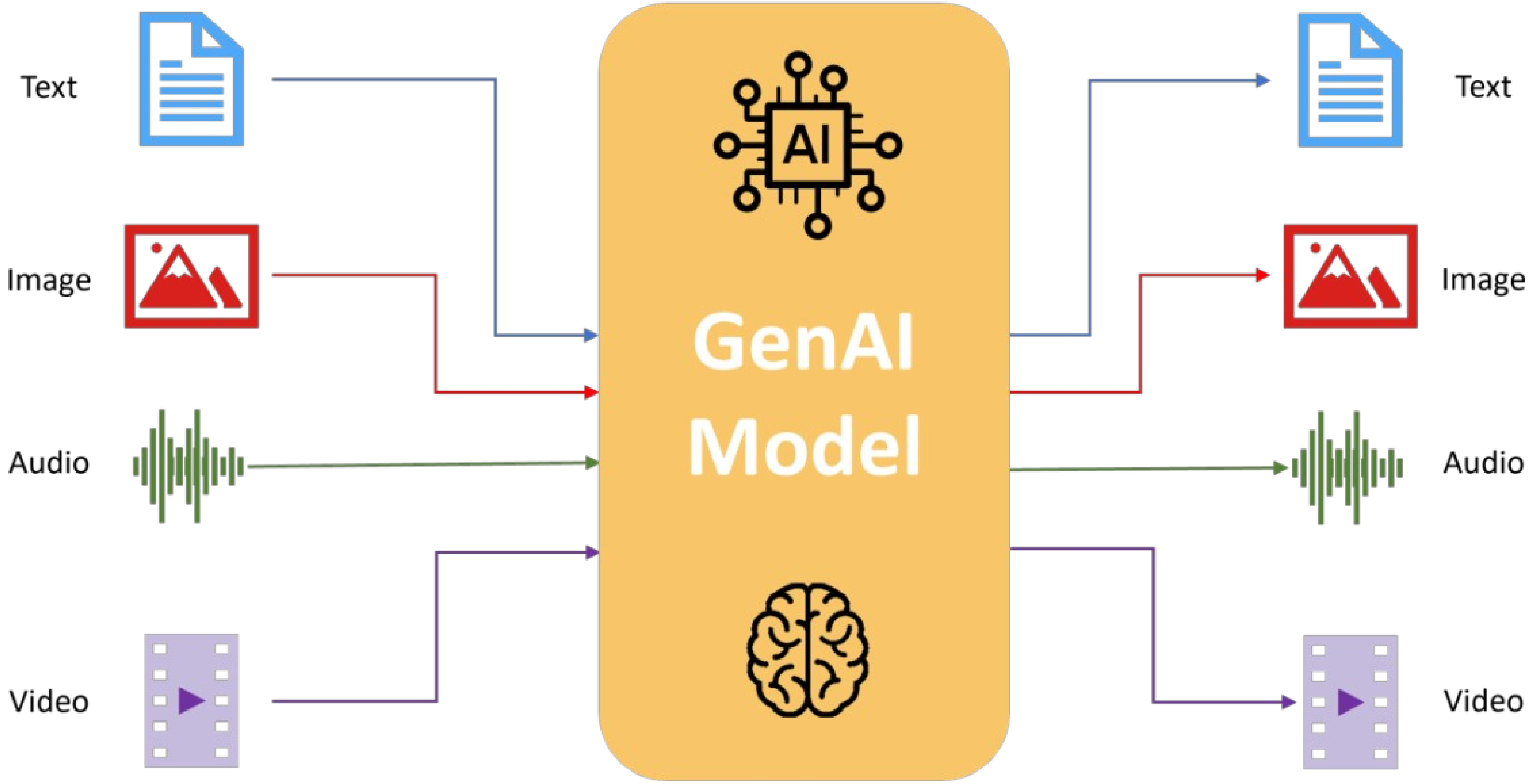
At this point in the course we have seen all of the necessary components to take a model from ideation to deployment. Let’s go through the steps below and build a chatbot and deploy it:

1. Finding the right task to solve
2. Gathering data
3. Building the right model architecture
4. Developing a smaller model
5. Training a large model with distributed computing
6. Preparing the trained model for inference



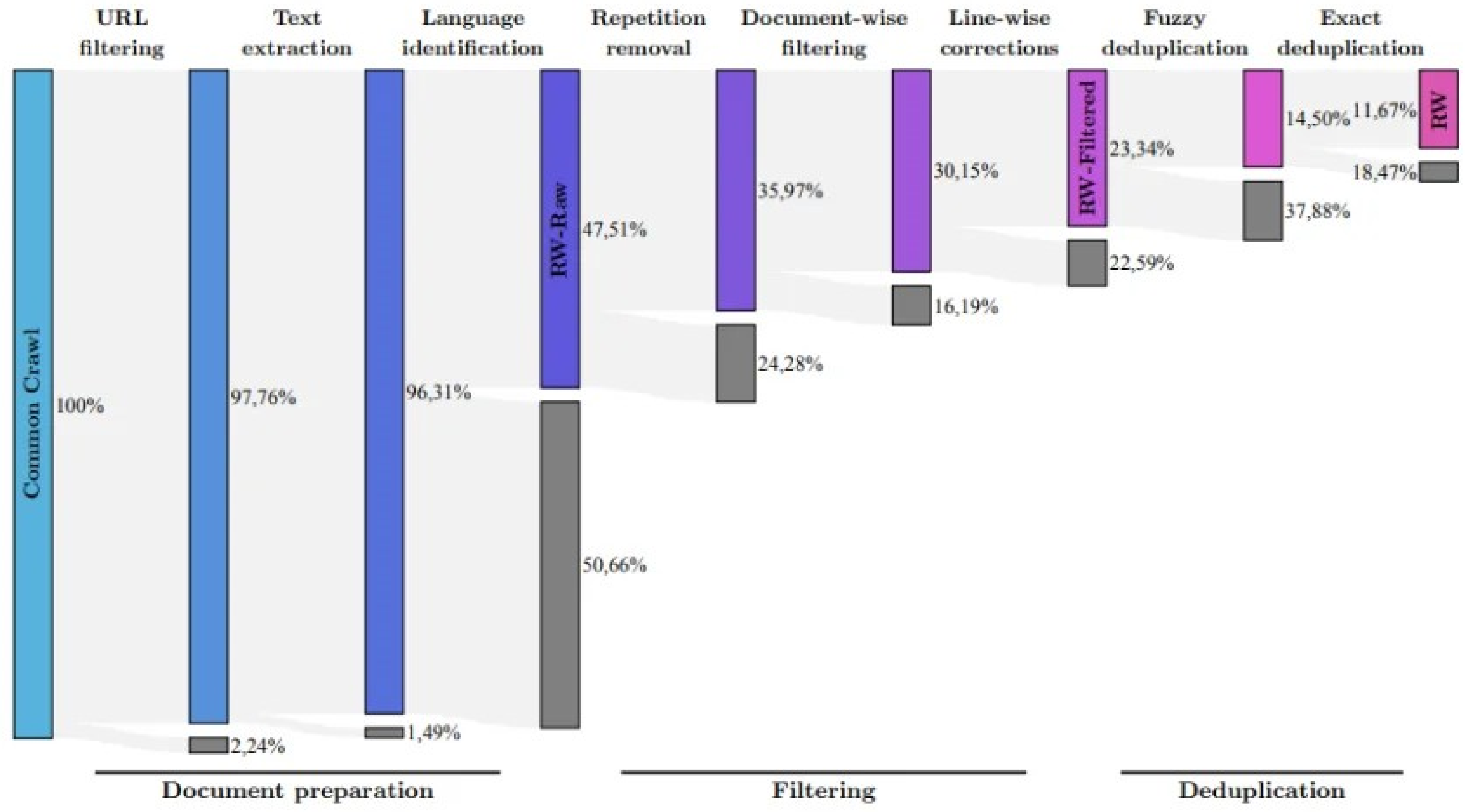
Life Cycle of an LLM

Step 1: Finding the right task to solve
Before begging, we need to decide what our task is to solve:



Life Cycle of an LLM

Step 2: Gathering data
With the task chosen, we then need to collect and collate the right data to build our model. For example, if we want to build a chatbot, we need a large amount of high-quality data from the web. This involves several stages of preparation.

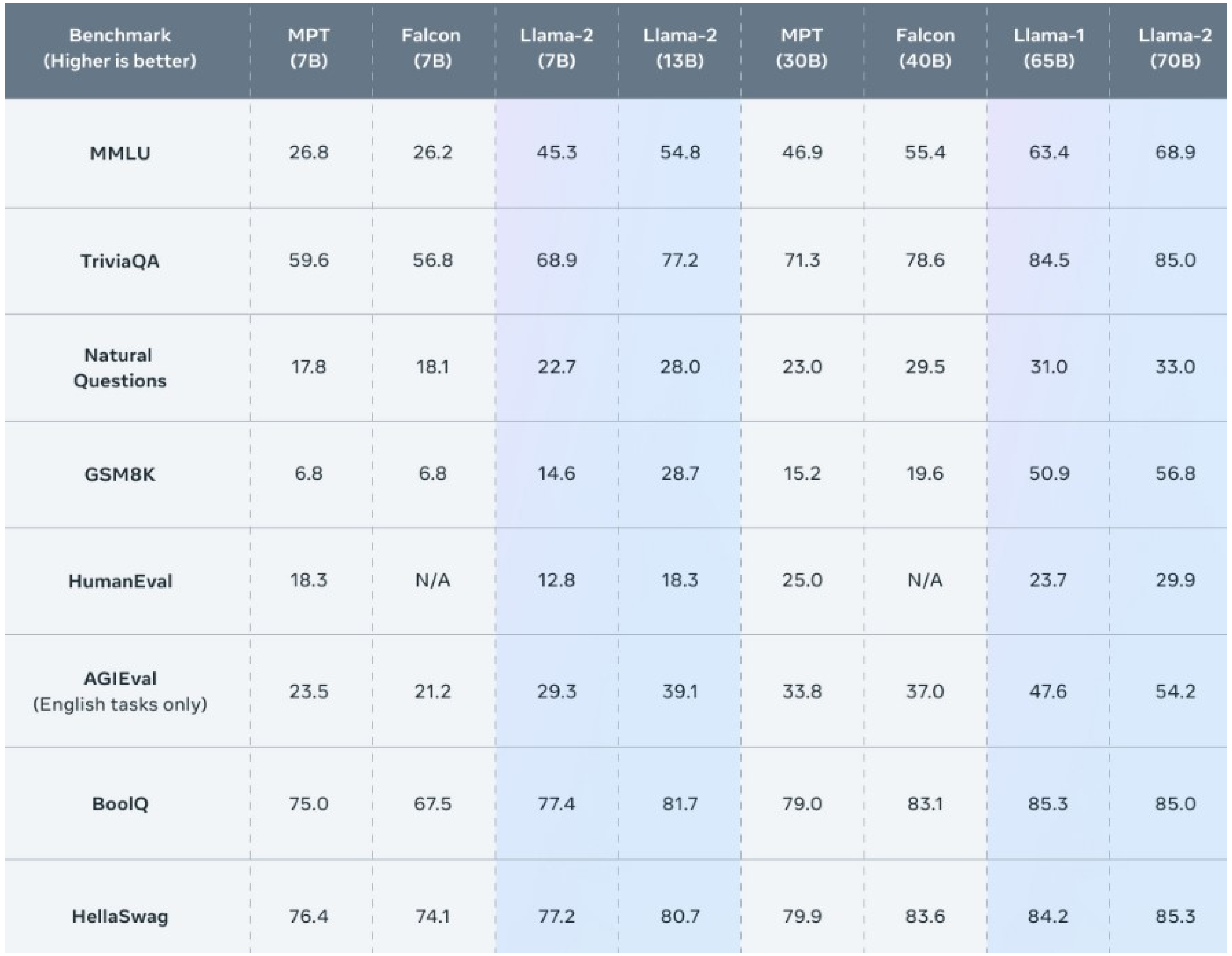


Life Cycle of an LLM

Step 3: Building the right model architecture

Based on the task and data we have chosen we now need to settle on the architecture given our performance expectations and compute budget.

This could be the decoder style, or different layer types, or the number of parameters.



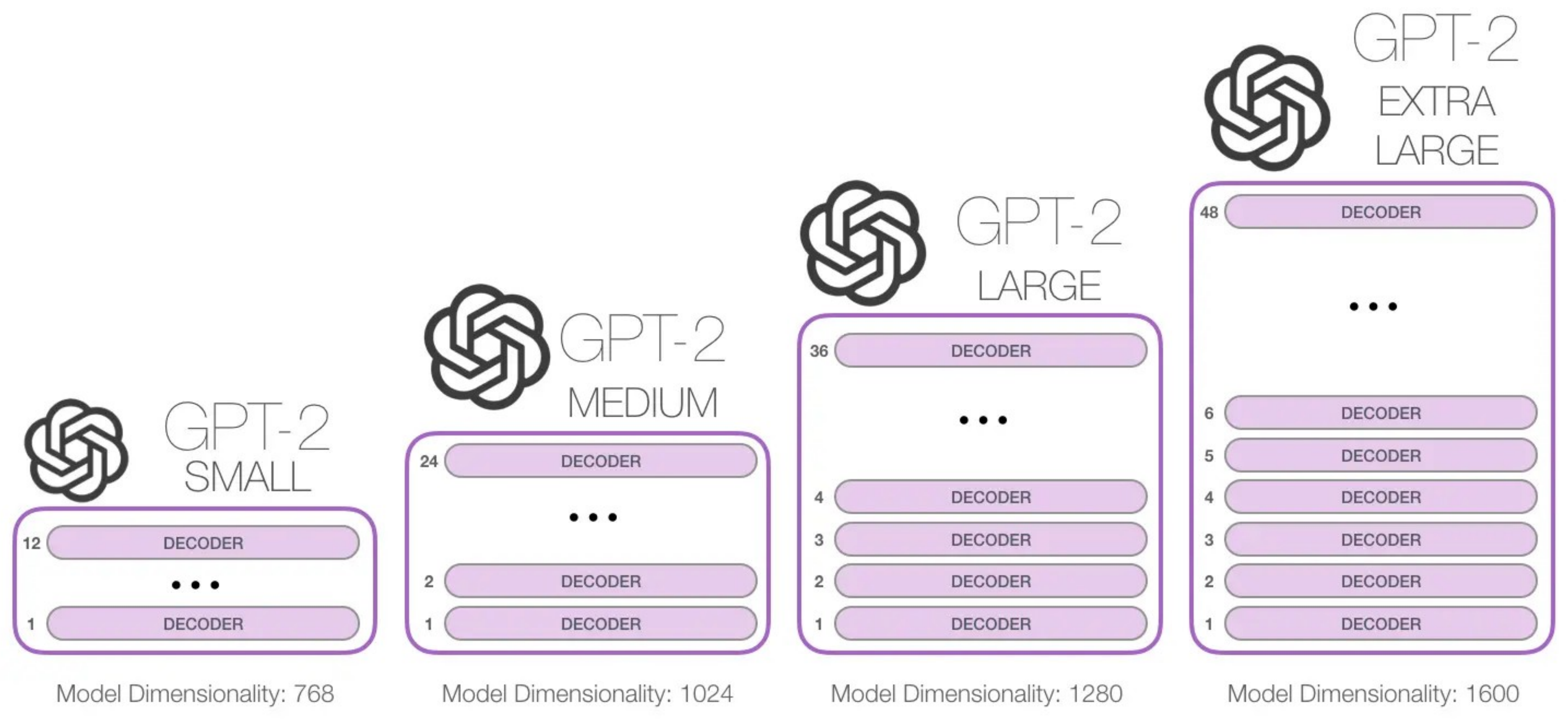
Benchmark (Higher is better)	MPT (7B)	Falcon (7B)	Llama-2 (7B)	Llama-2 (13B)	MPT (30B)	Falcon (40B)	Llama-1 (65B)	Llama-2 (70B)
MMLU	26.8	26.2	45.3	54.8	46.9	55.4	63.4	68.9
TriviaQA	59.6	56.8	68.9	77.2	71.3	78.6	84.5	85.0
Natural Questions	17.8	18.1	22.7	28.0	23.0	29.5	31.0	33.0
GSM8K	6.8	6.8	14.6	28.7	15.2	19.6	50.9	56.8
HumanEval	18.3	N/A	12.8	18.3	25.0	N/A	23.7	29.9
AGIEval (English tasks only)	23.5	21.2	29.3	39.1	33.8	37.0	47.6	54.2
BoolQ	75.0	67.5	77.4	81.7	79.0	83.1	85.3	85.0
HellaSwag	76.4	74.1	77.2	80.7	79.9	83.6	84.2	85.3

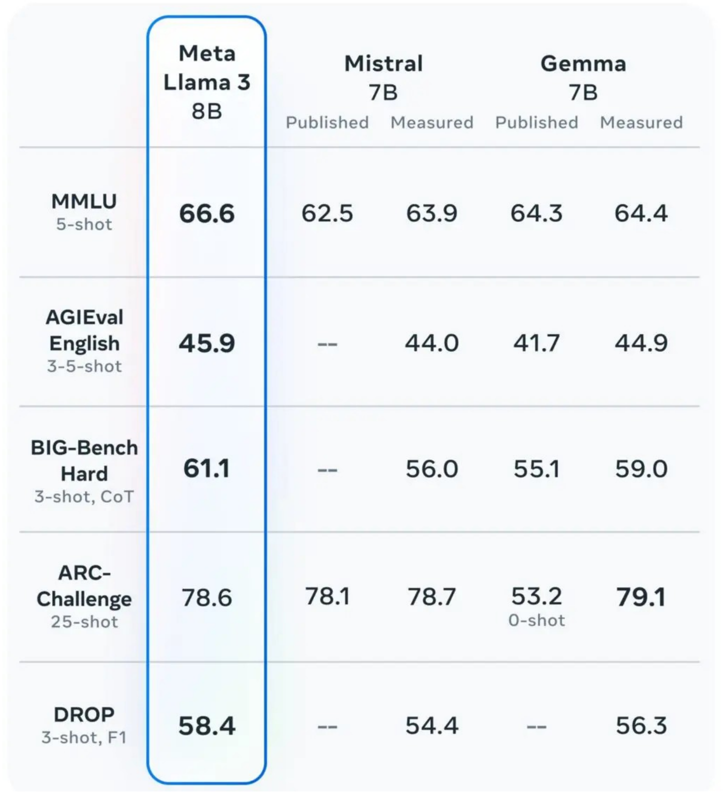
Life Cycle of an LLM

Step 4: Developing a smaller model

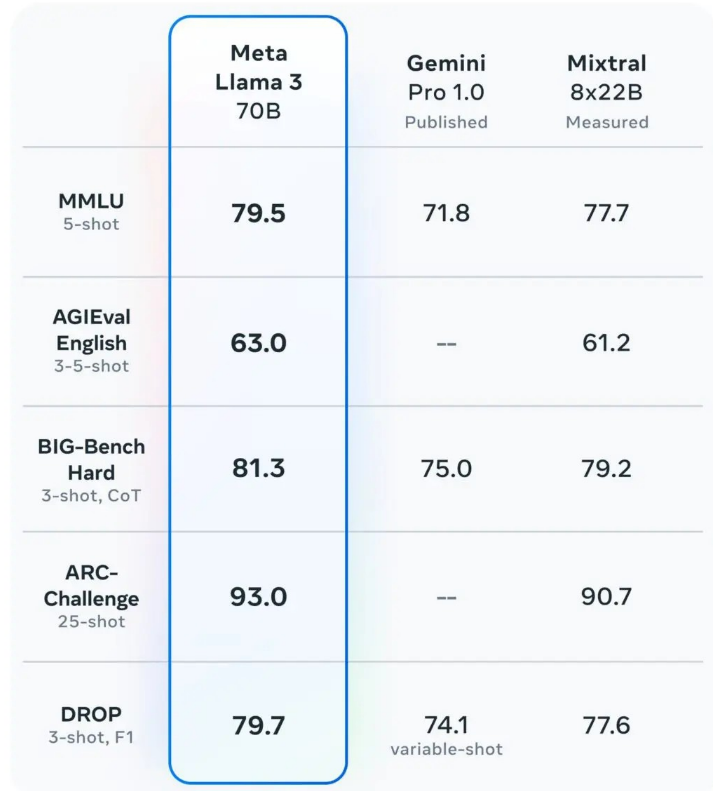
Our first step in building the model is starting with a smaller model that will let us:

1. **Pipeline Validation**:
   - Allows us to ensure the end-to-end workflow (data preprocessing, model training, evaluation, and inference) is functioning correctly without investing significant computational resources.
   - Helps identify potential bottlenecks or compatibility issues early in the development process.
2. **Faster Iterations**:
   - Smaller models train and infer more quickly, enabling rapid experimentation with different configurations or features.
   - This is especially useful when testing new techniques or architectures.
3. **Baseline Establishment**:
   - A smaller model serves as a baseline for performance metrics like accuracy, latency, and memory usage.
   - Provides a reference point to measure improvements when scaling up the model.





	Meta Llama 3 8B	Mistral 7B		Gemma 7B	
		Published	Measured	Published	Measured
MMLU 5-shot	66.6	62.5	63.9	64.3	64.4
AGIEval English 3-5-shot	45.9	--	44.0	41.7	44.9
BIG-Bench Hard 3-shot, CoT	61.1	--	56.0	55.1	59.0
ARC-Challenge 25-shot	78.6	78.1	78.7	53.2 0-shot	79.1
DROP 3-shot, F1	58.4	--	54.4	--	56.3



	Meta Llama 3 70B	Gemini Pro 1.0	Mixtral 8x22B
		Published	Measured
MMLU 5-shot	79.5	71.8	77.7
AGIEval English 3-5-shot	63.0	--	61.2
BIG-Bench Hard 3-shot, CoT	81.3	75.0	79.2
ARC-Challenge 25-shot	93.0	--	90.7
DROP 3-shot, F1	79.7	74.1 variable-shot	77.6

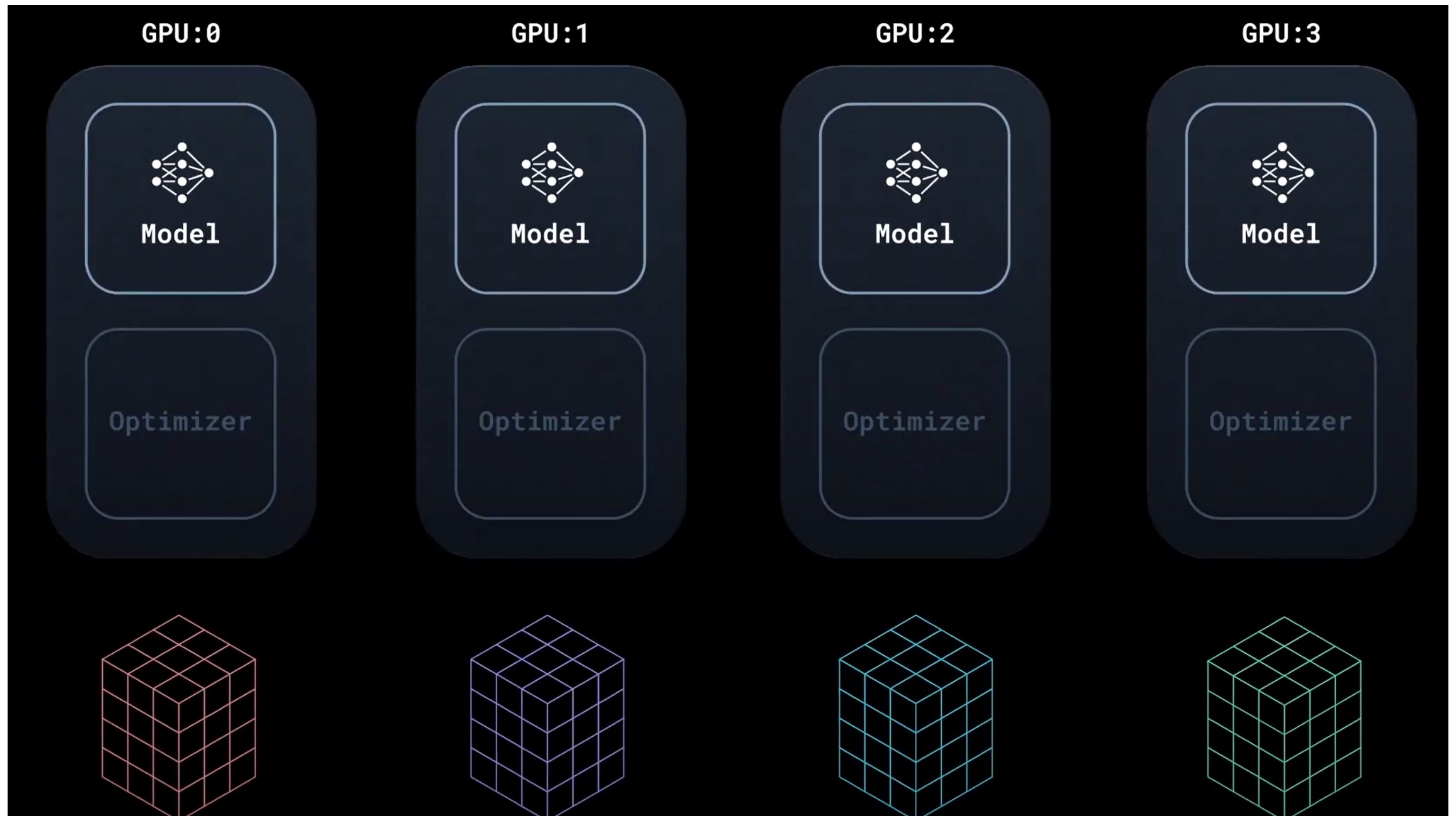
Life Cycle of an LLM

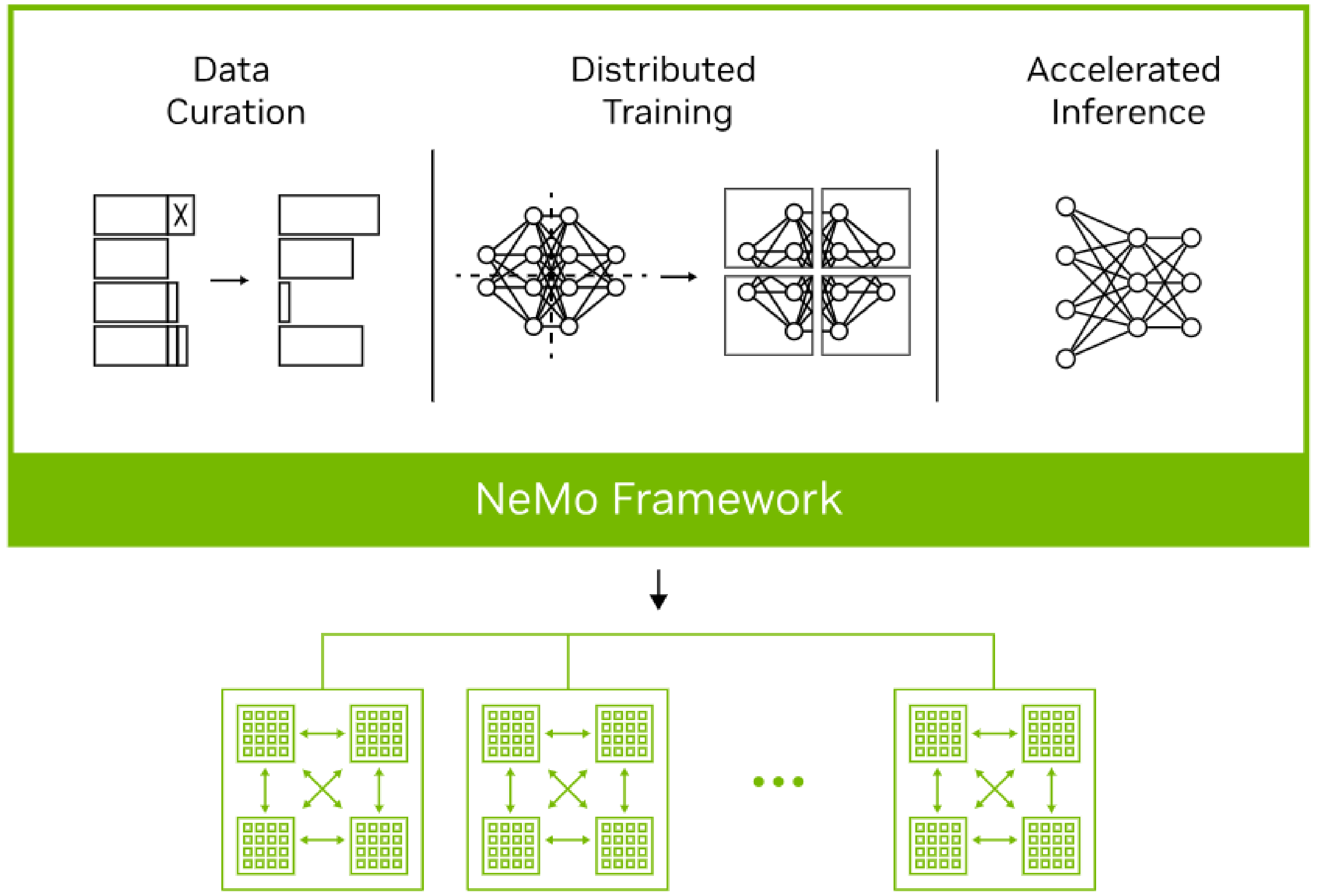
Step 5: Training a large model with distributed computing

With a smaller model and pipeline now validated and showing the right performance characteristics for scale, we now implement the distributed training procedures.

Data and Model Parallelism
Depending on the model scaling, data scaling and resource availability data parallelism, such as FSDP could be utilized but if the model becomes too large, model parallelism may also be required to split the model across the compute resources.

This stage will be the most expensive and time consuming so good monitoring and robustness in the approach is critical.
`





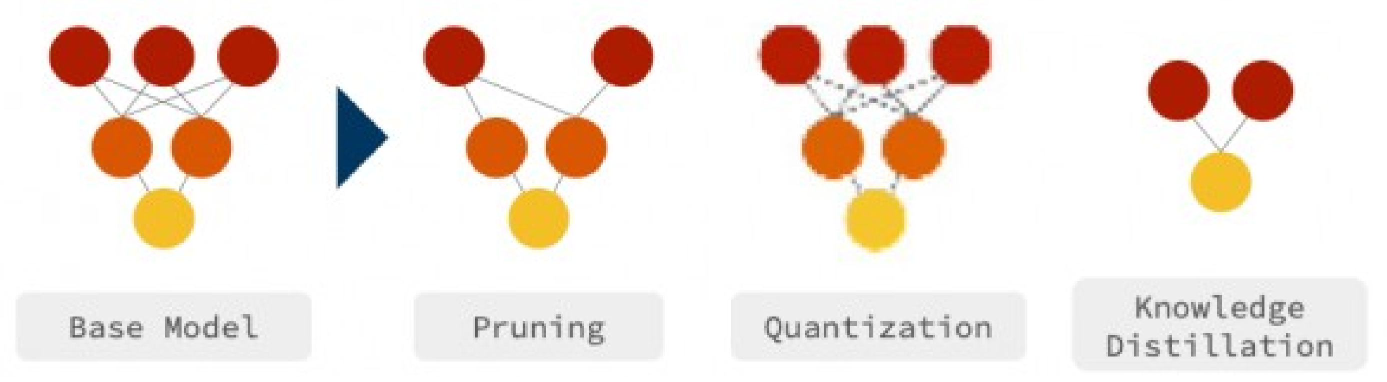
Life Cycle of an LLM

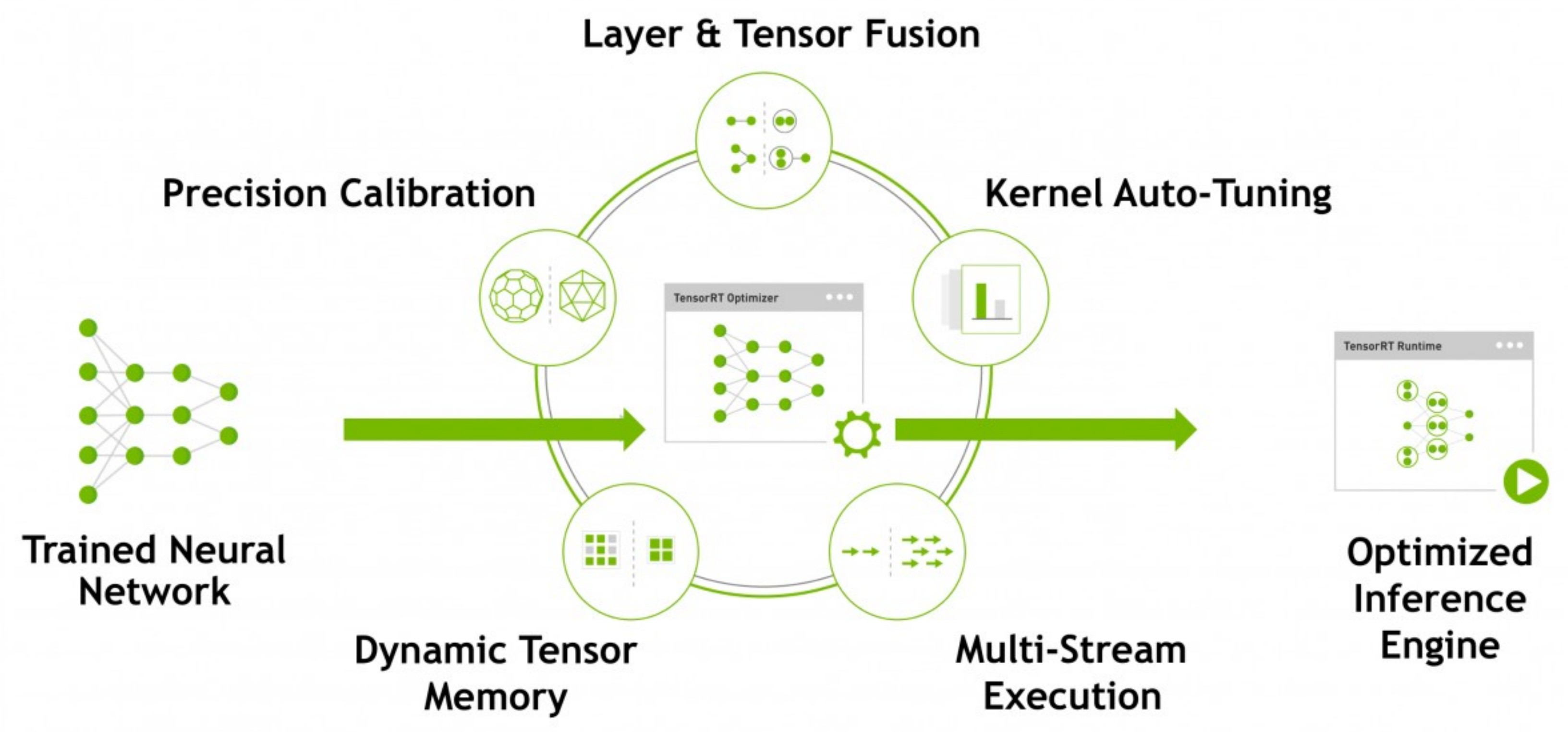
Step 6: Preparing the trained model for inference

With the model trained, it is now ready for deployment so it can be used for inference.

Optimizing the model involves pruning and quantizing the model to sufficiently balance performance.

This will involve libraries like TensorRT/LLM to utilize specially designed CUDA kernels and optimizations to convert the trained model into a package ready for inference deployment.





Life Cycle of an LLM

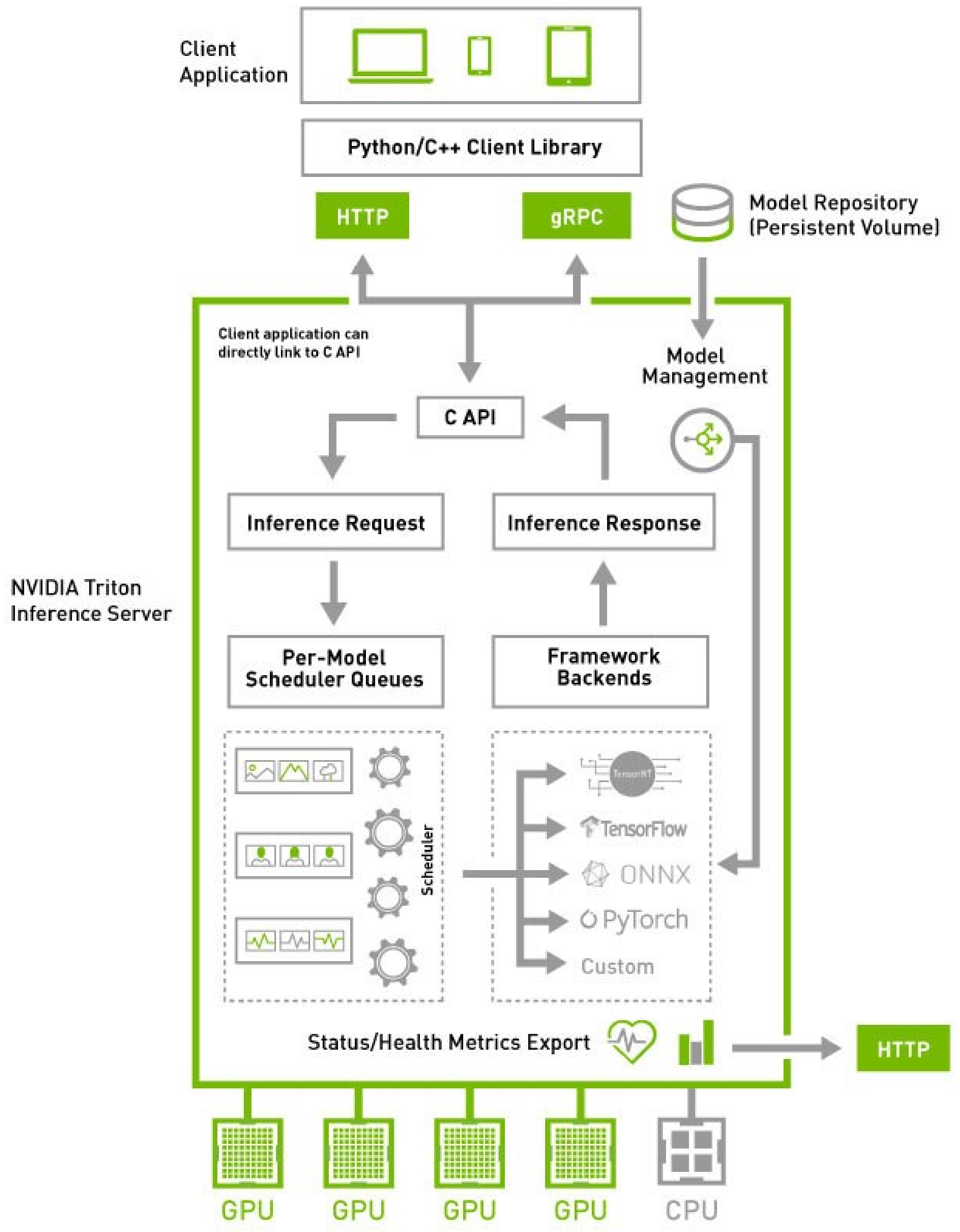
Step 7: Deploying the trained model on an inference server

The final stage is deploying the LLM for production inference. In this stage the model has been optimized but may be processed further, including processes like model parallelism to ensure that throughput and latency can be maximized for performance.

Before deployment, careful consideration should be given to :

- **Load balancing**
- **Dynamic batching**
- **Resource allocation**

Monitoring tools are also set up to track inference metrics, such as latency, GPU utilization, and error rates, ensuring the system remains robust under production workloads.

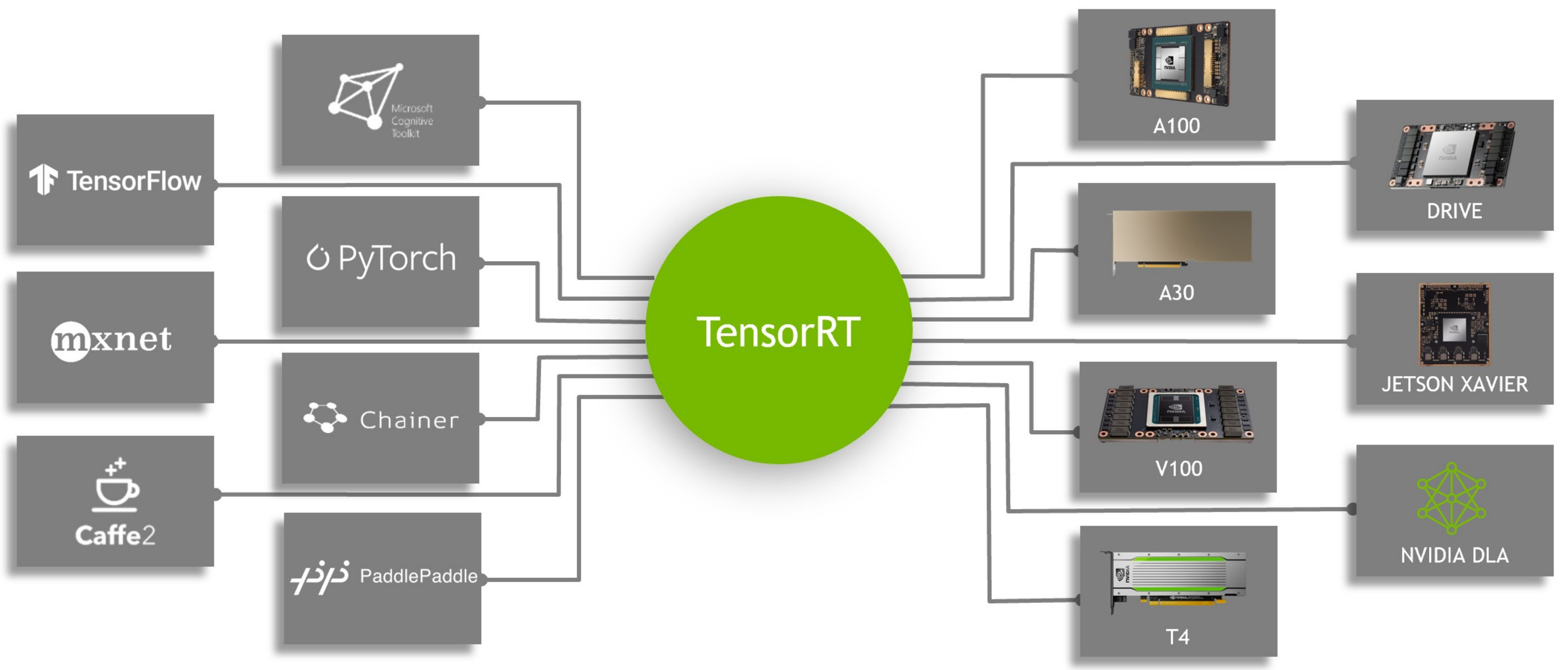


Client Application
Python/C++ Client Library
HTTP
gRPC
Model Repository (Persistent Volume)
Client application can directly link to C API
Model Management
C API
Inference Request
Inference Response
NVIDIA Triton Inference Server
Per-Model Scheduler Queues
Framework Backends
Scheduler
TensorRT
TensorFlow
ONNX
PyTorch
Custom
Status/Health Metrics Export
HTTP
GPU
GPU
GPU
GPU
CPU

Wrap Up

Scaling Inference and Deployment

- Today we concluded this module on distributed computing and deployment
- We saw the different priorities that inference/deployment of models.
- Libraries like TensorRT, Triton, and vLLM were introduced as tools to convert our trained model to be deployment ready.



NVIDIA

DARTMOUTH ENGINEERING

1867

Thank you!