session 4

how to serve model predictions (in a clever way)

ml.school

1 model versioning

2 model serving tradeoffs

3 model serving strategies

4 human-in-the-loop workflows

5 model compression

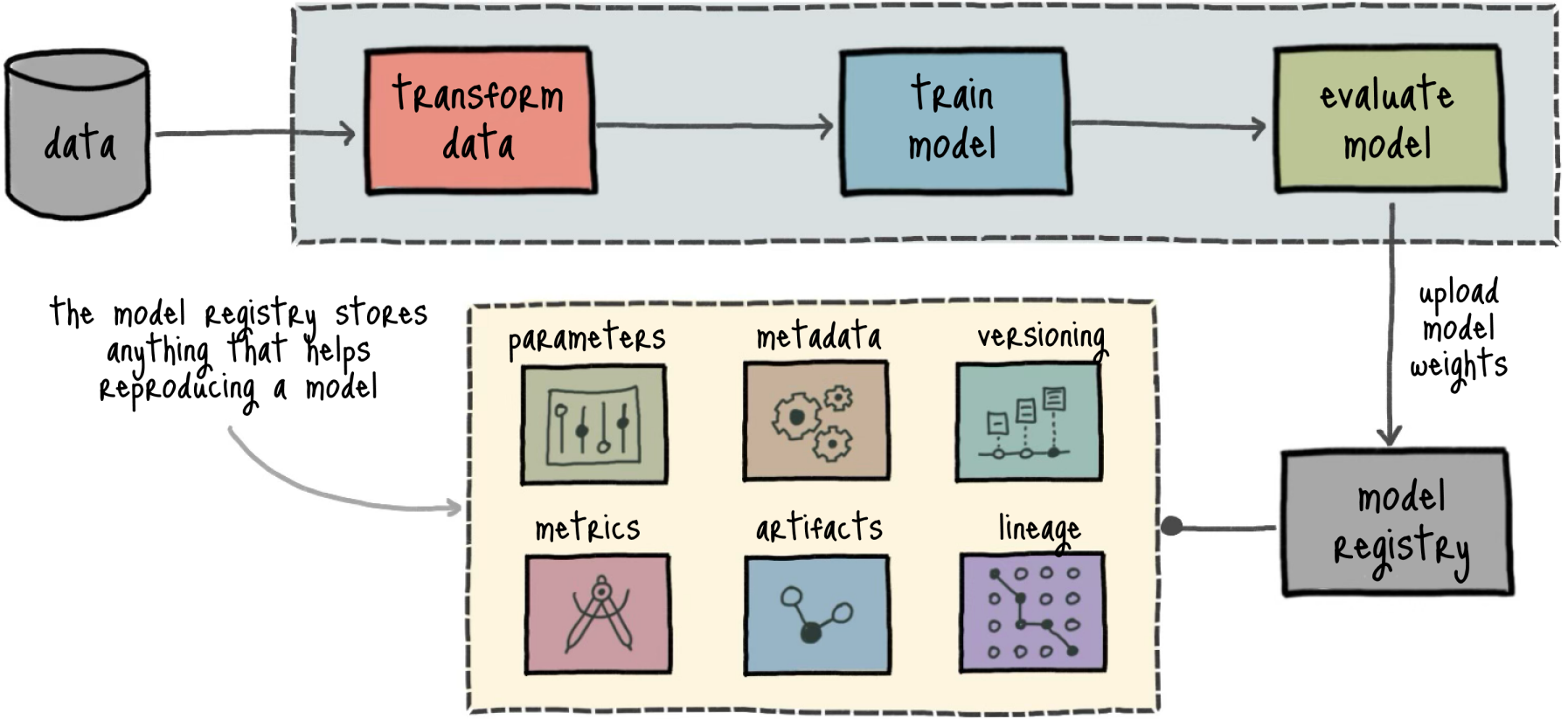
Deploying machine learning models to production is what separates projects that generate value from simple academic exercises.

High performance alone doesn’t make good models. We need a process to build, store, version, share, maintain, and serve model predictions at scale.

Model versioning is the process of systematically tracking and managing different versions of a model to ensure **reproducibility** and **traceability** over time.

Model versioning involves tracking any **data** used to build the model, **source code**, **parameters**, **metrics**, and any other **metadata** to ensure full reproducibility.

model versioning

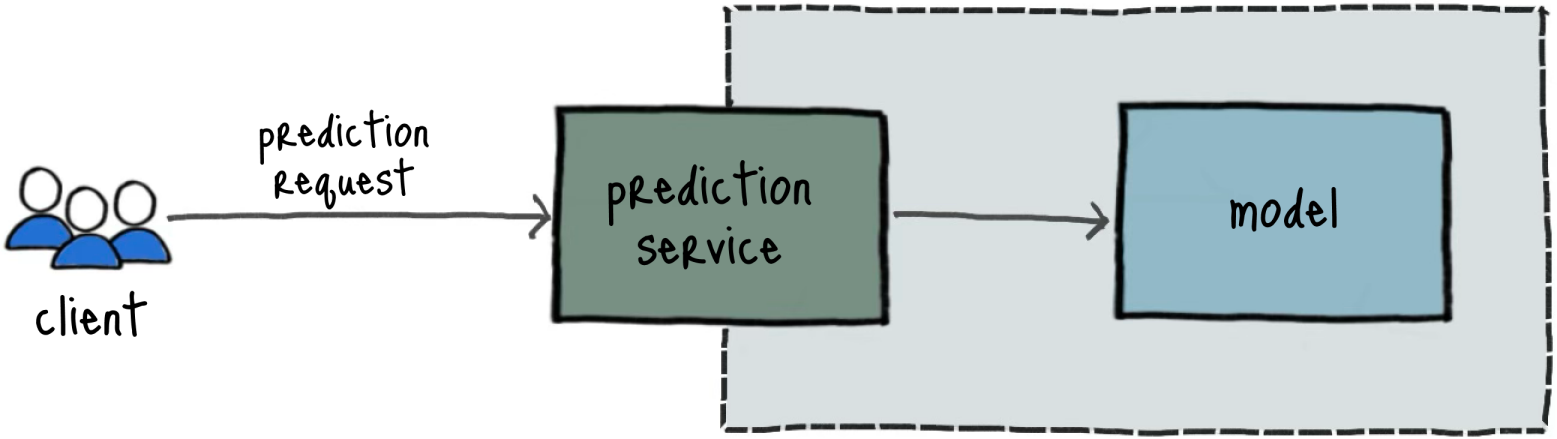


Without model versioning, it becomes **impossible** to trace a model's evolution, reproduce results reliably, or safely roll back to previous versions if any issue arises.

Model serving is the process of **deploying** a trained model so that it can be accessed by other applications, allowing it to make **predictions** on **new data**.

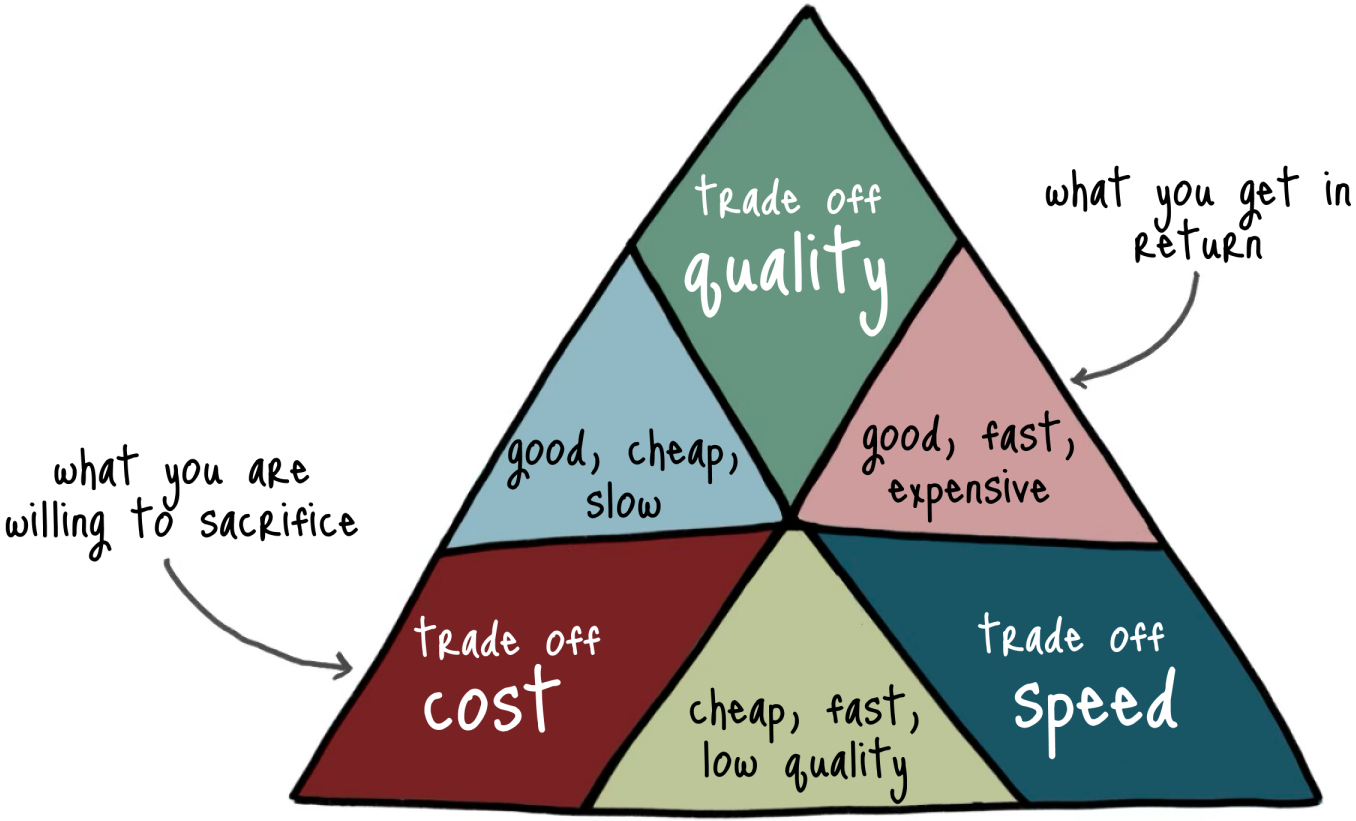
Model as a Service is the most common method used to deploy models. This approach wraps the model with an **API**, making it easy to integrate it into any system.

model serving



Deploying a model is simple, except when prediction **quality**, **costs**, and **speed** matter. These are key tradeoffs to consider when serving model predictions.

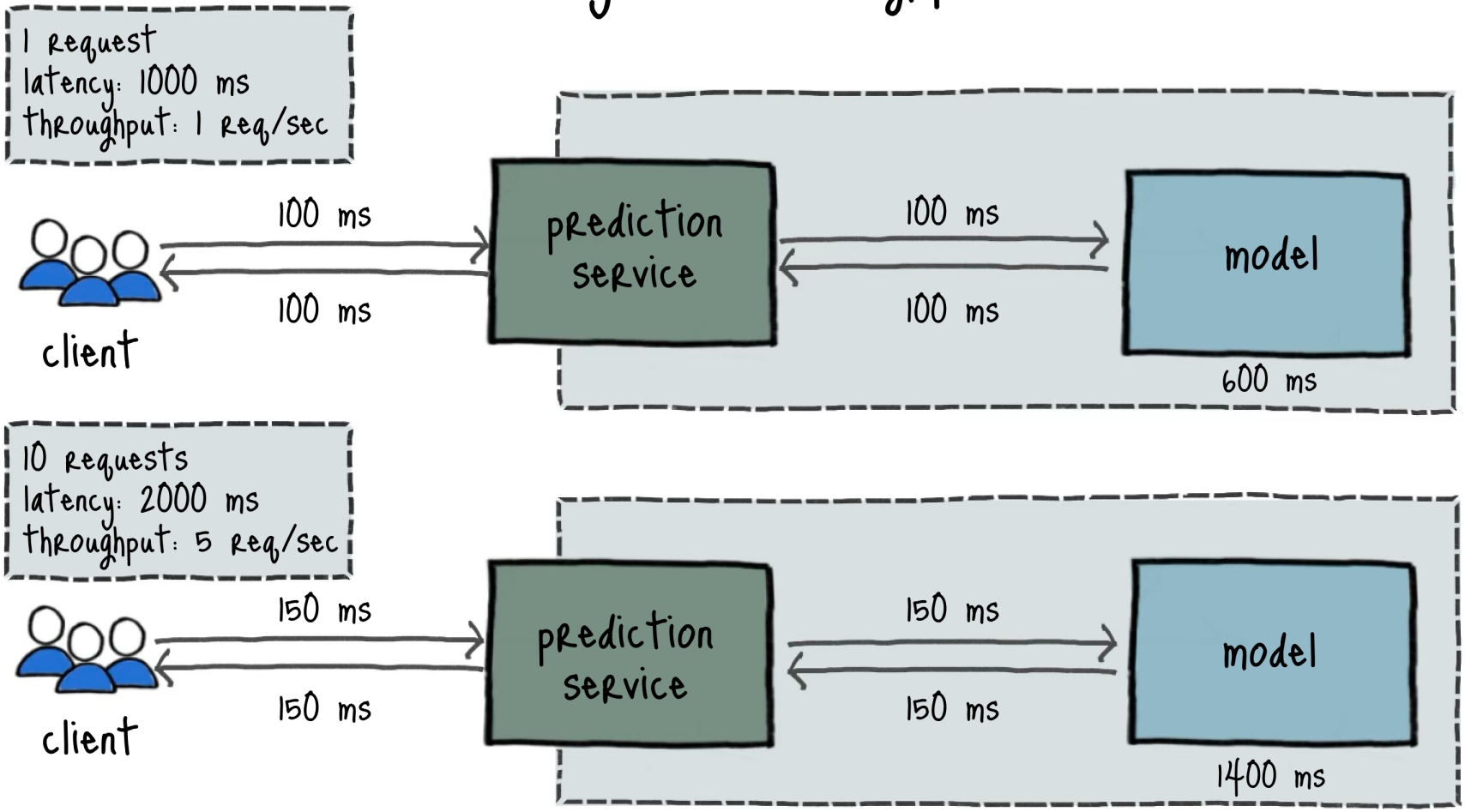
the unattainable priority triangle



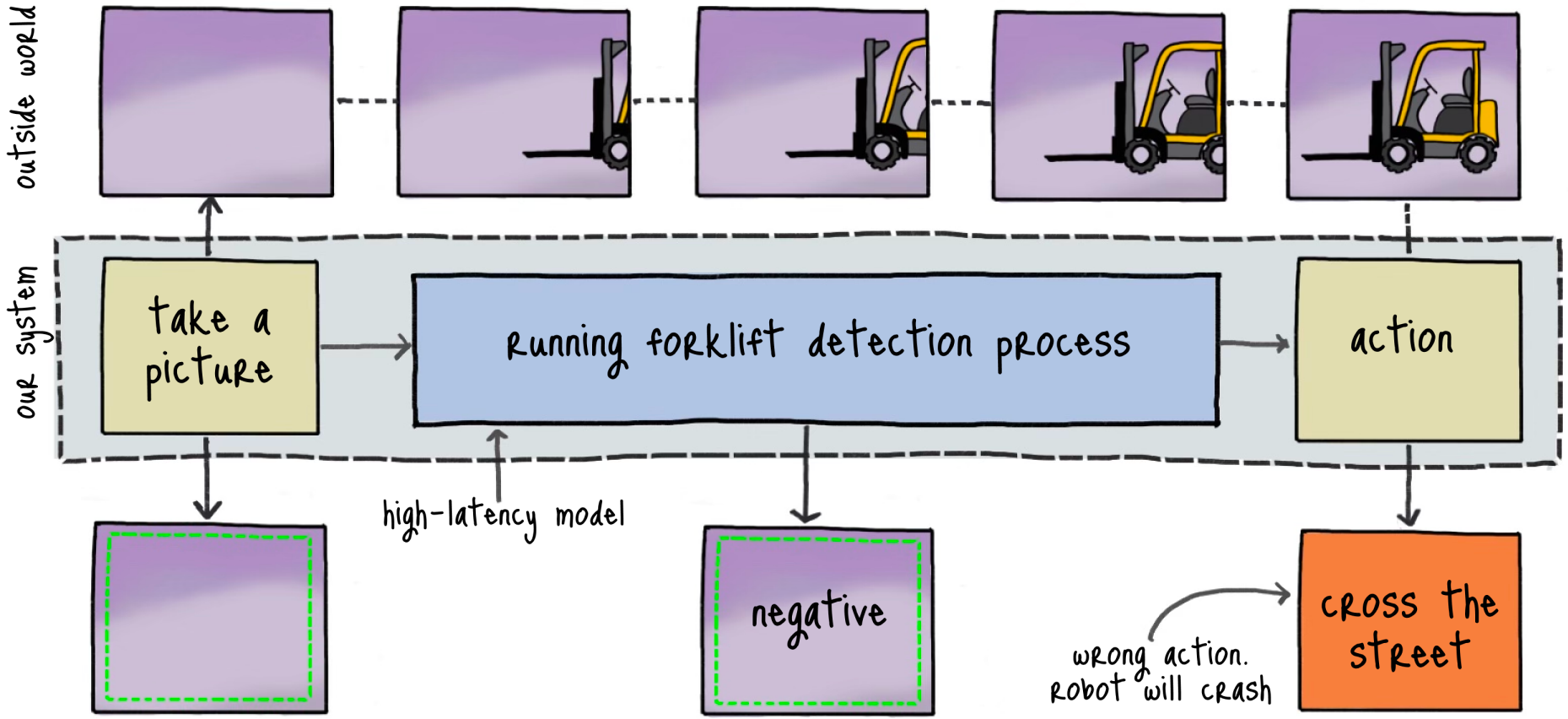
Tradeoffs in quality, cost, and speed impact the model's **latency** and **throughput**. These two metrics are crucial to understanding the model's **real-world** performance.

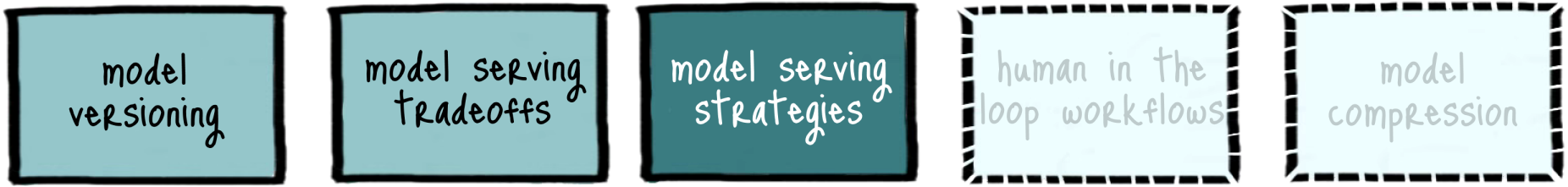
Latency is a measure of delay experienced in a system, while **throughput** refers to the amount of data processed by a system in a given amount of time.

latency and throughput



detecting forklifts in a warehouse





An **inference pipeline** is a sequence of steps and models for transforming inputs into predictions, ensuring **consistency** between training and inference.

inference pipeline

input image

prediction service

object detection

identifies every component on the input image

for each...

multi-class classification

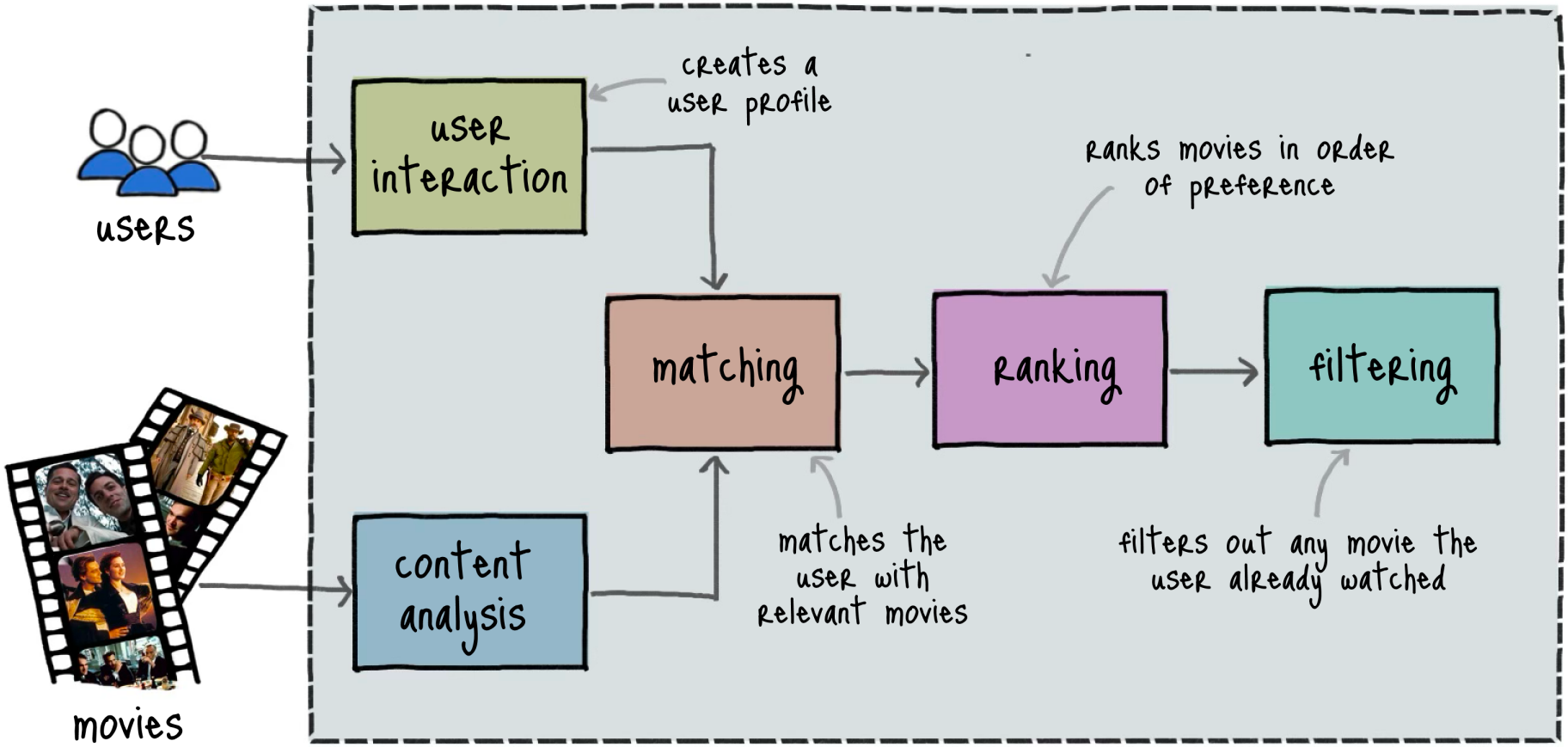
classifies every component from the previous step

insulator qc model

arrester qc model

transformer qc model

an inference pipeline for a movie recommendation system



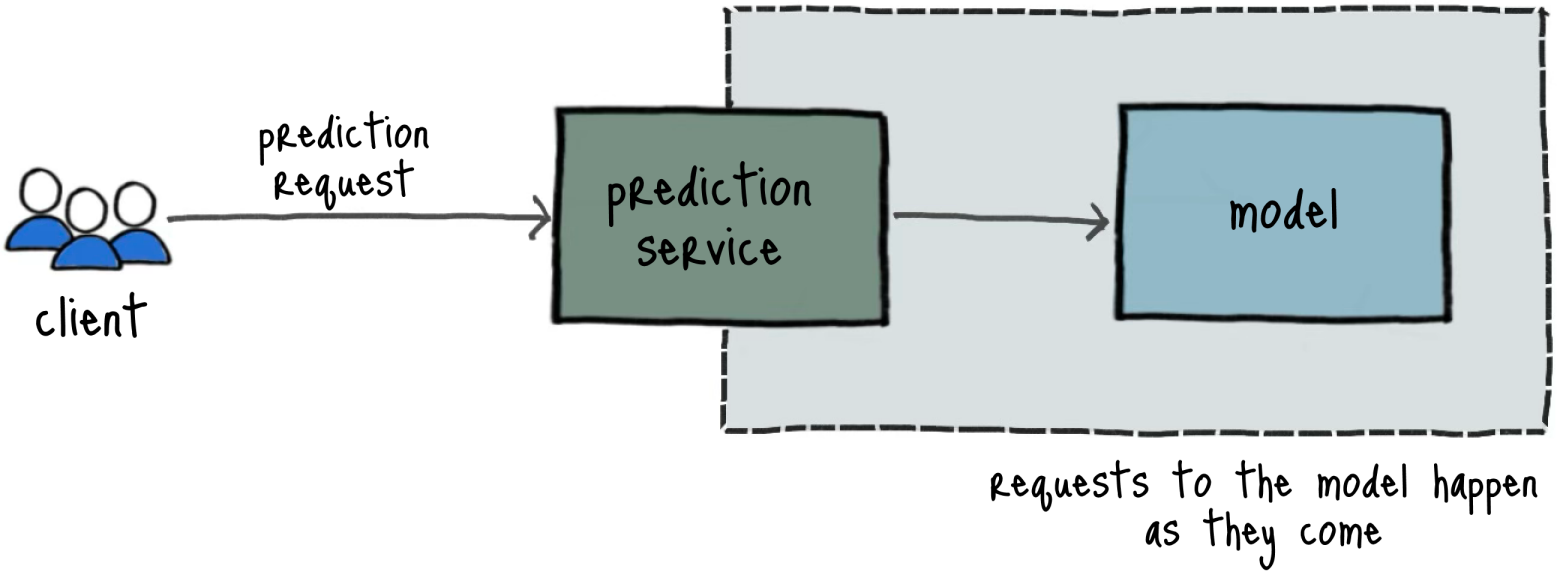
Deploying an inference pipeline in production can be very different from deploying traditional software. The correct strategy depends on the specific use case.

Dynamic serving computes predictions in real time.

It has **lower storage costs** and **variable latency**.

It works best for **high cardinality** problems.

dynamic serving



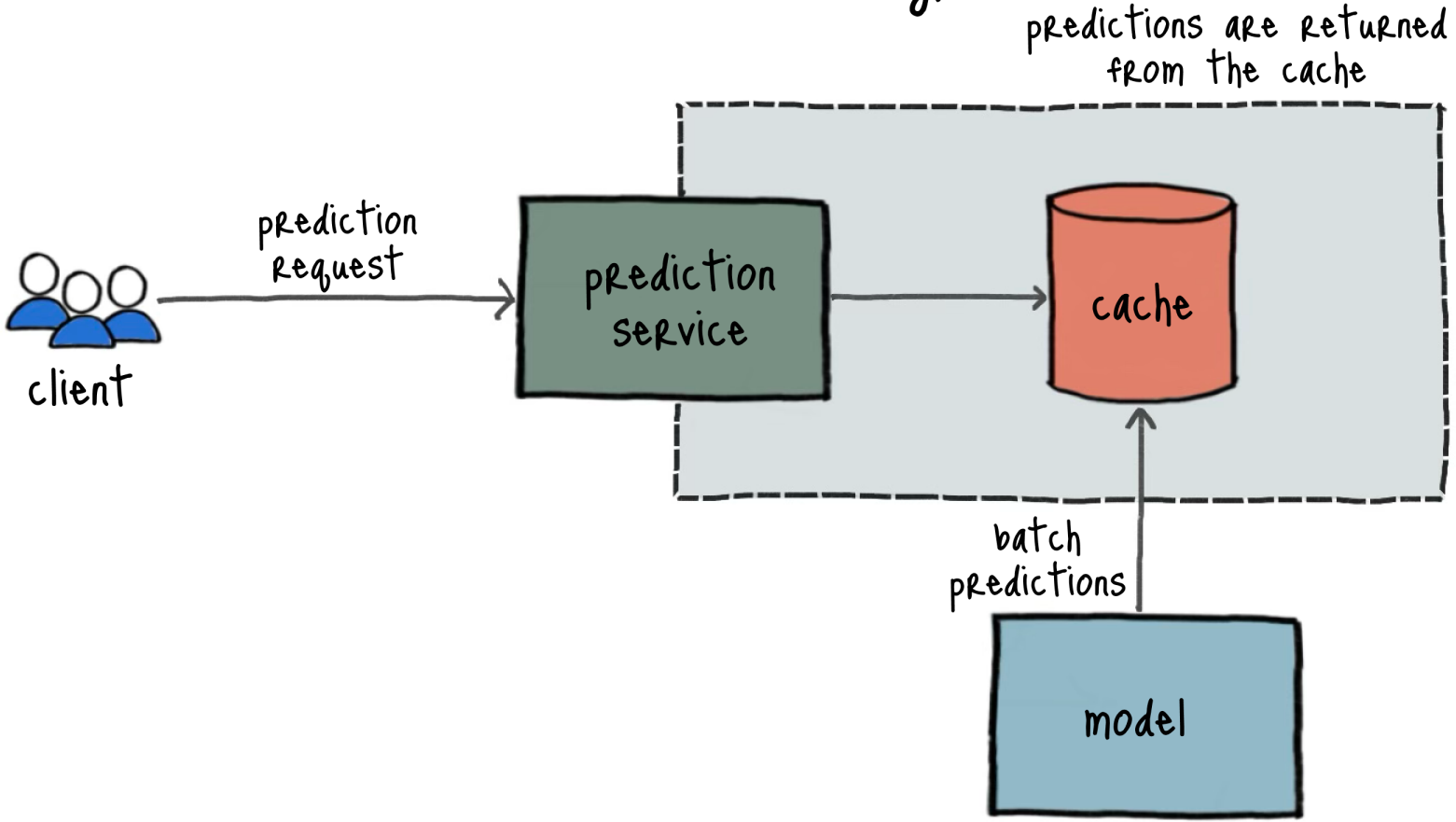
Dynamic serving has high computational **costs** and requires **complex infrastructure** to ensure scalability and availability when serving multiple users.

Static serving pre-computes predictions ahead of time.

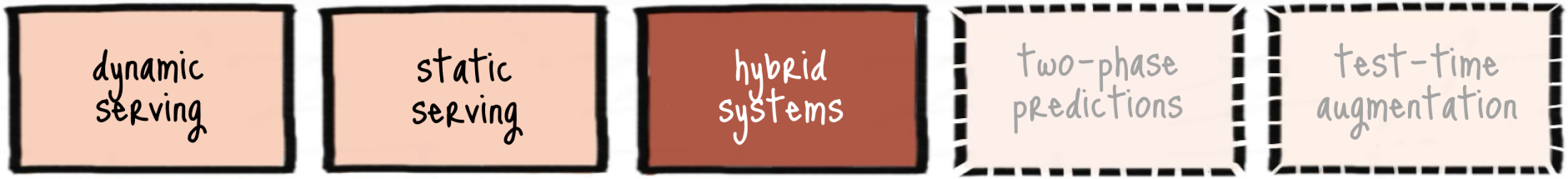
It has higher **storage costs** and **low latency**.

It works best for **low cardinality** problems.

static serving

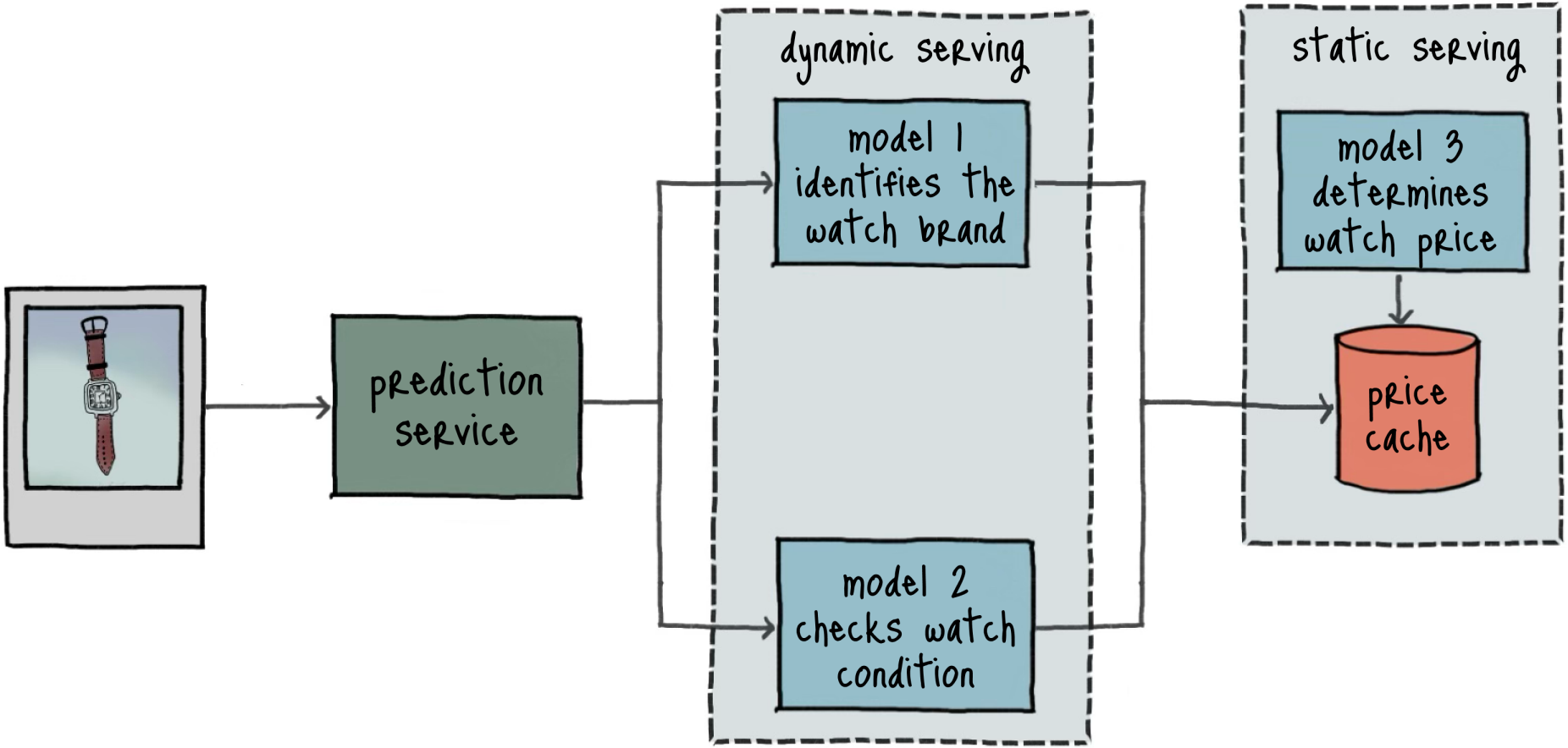


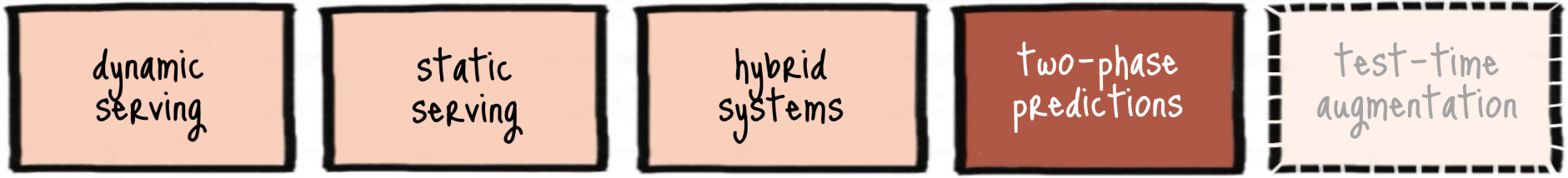
Static serving requires an **upfront** computational load. It relies on pre-generated predictions, making the system **less responsive** to new data or user behavior.



A **hybrid** system combines **dynamic** and **static** serving. It pre-computes and serves low-cardinality predictions from a cache while serving the rest in real time.

a hybrid system using dynamic and static serving

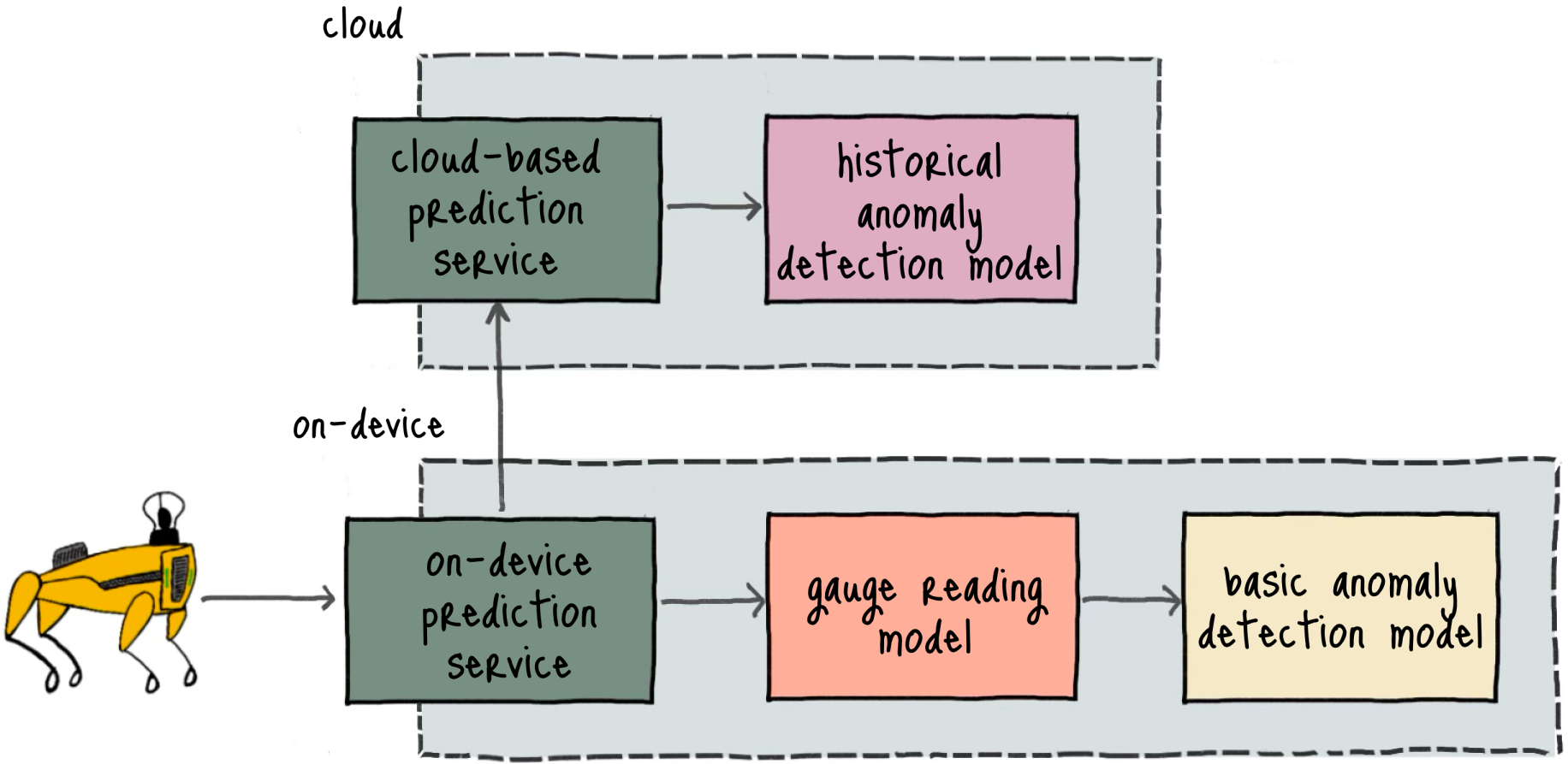


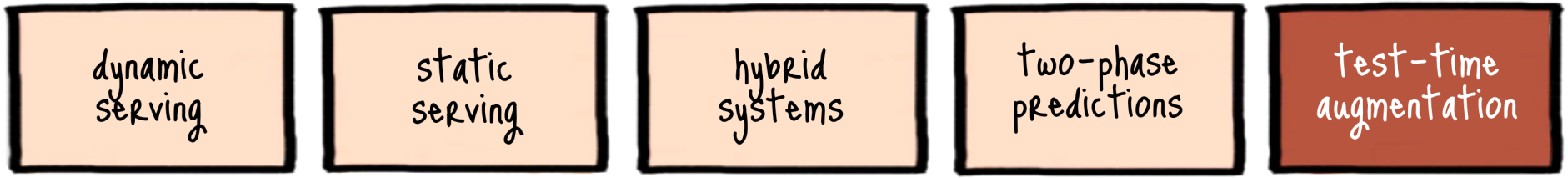


Two-phase predictions is a great approach to run large models efficiently by reducing computational time without compromising prediction quality.

Two-phase predictions uses an initial, **simpler** model to filter or narrow down inputs, followed by a more **complex** model for detailed predictions.

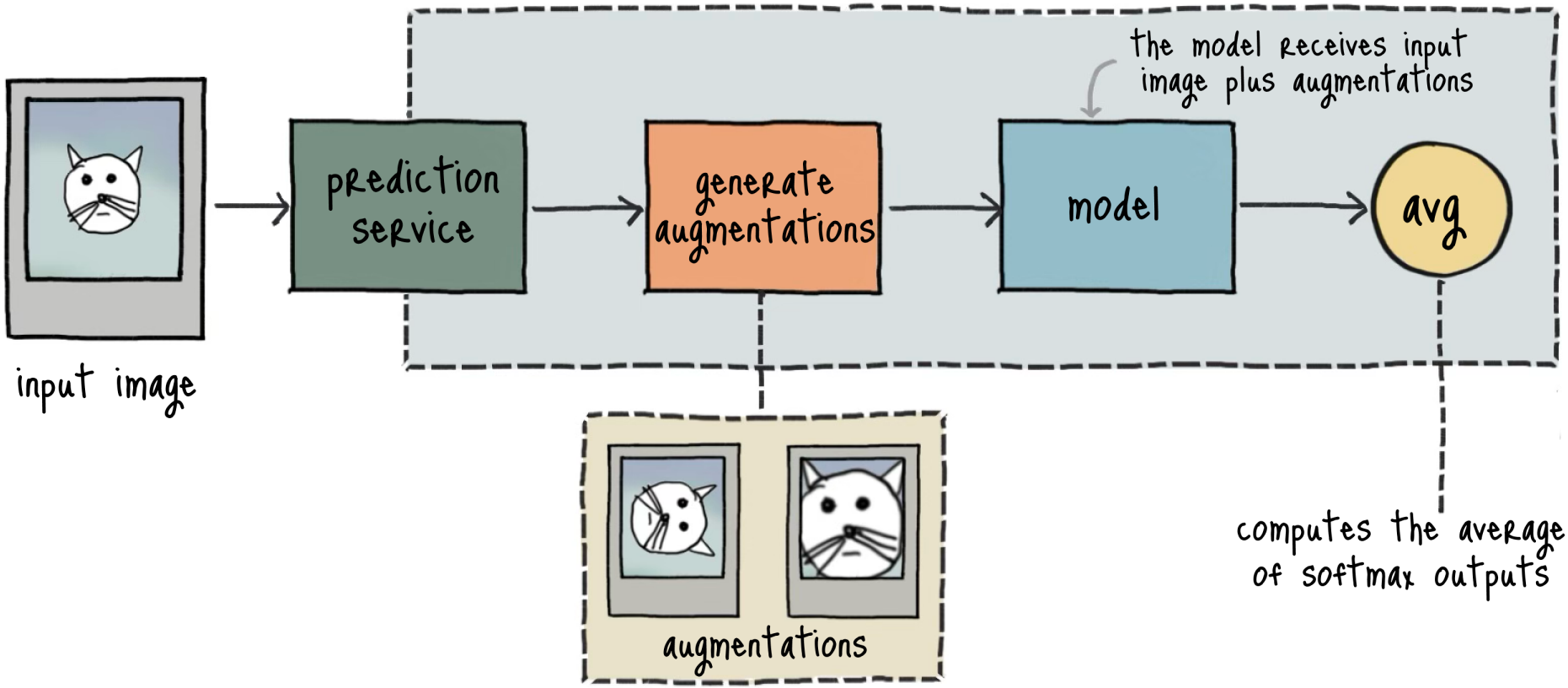
two-phase predictions

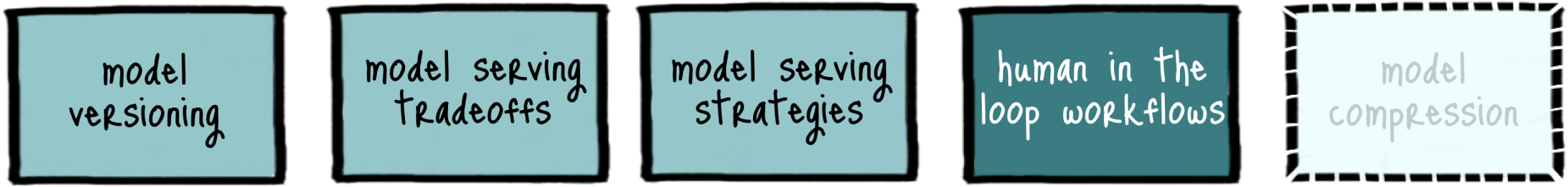




Test-time augmentation is a technique for boosting prediction accuracy and robustness by simulating a broader range of data conditions during inference.

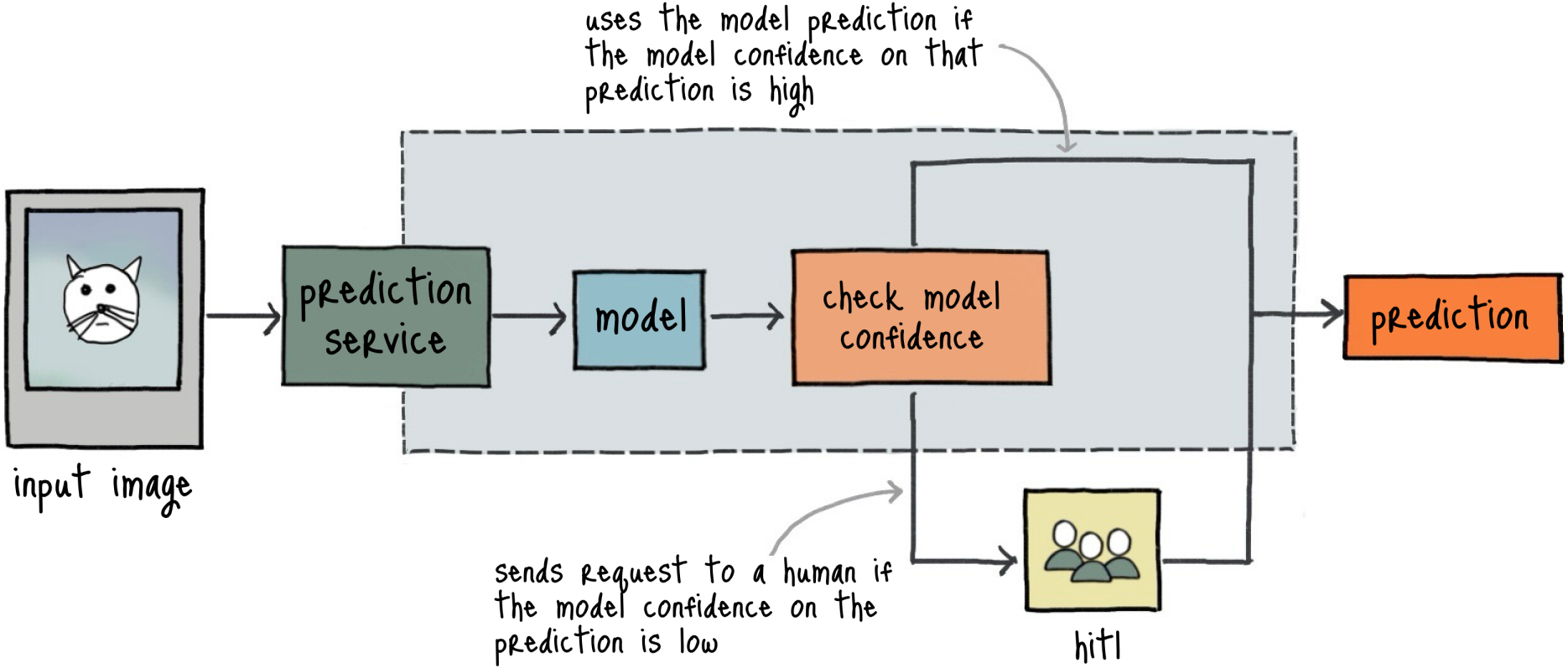
test-time augmentation





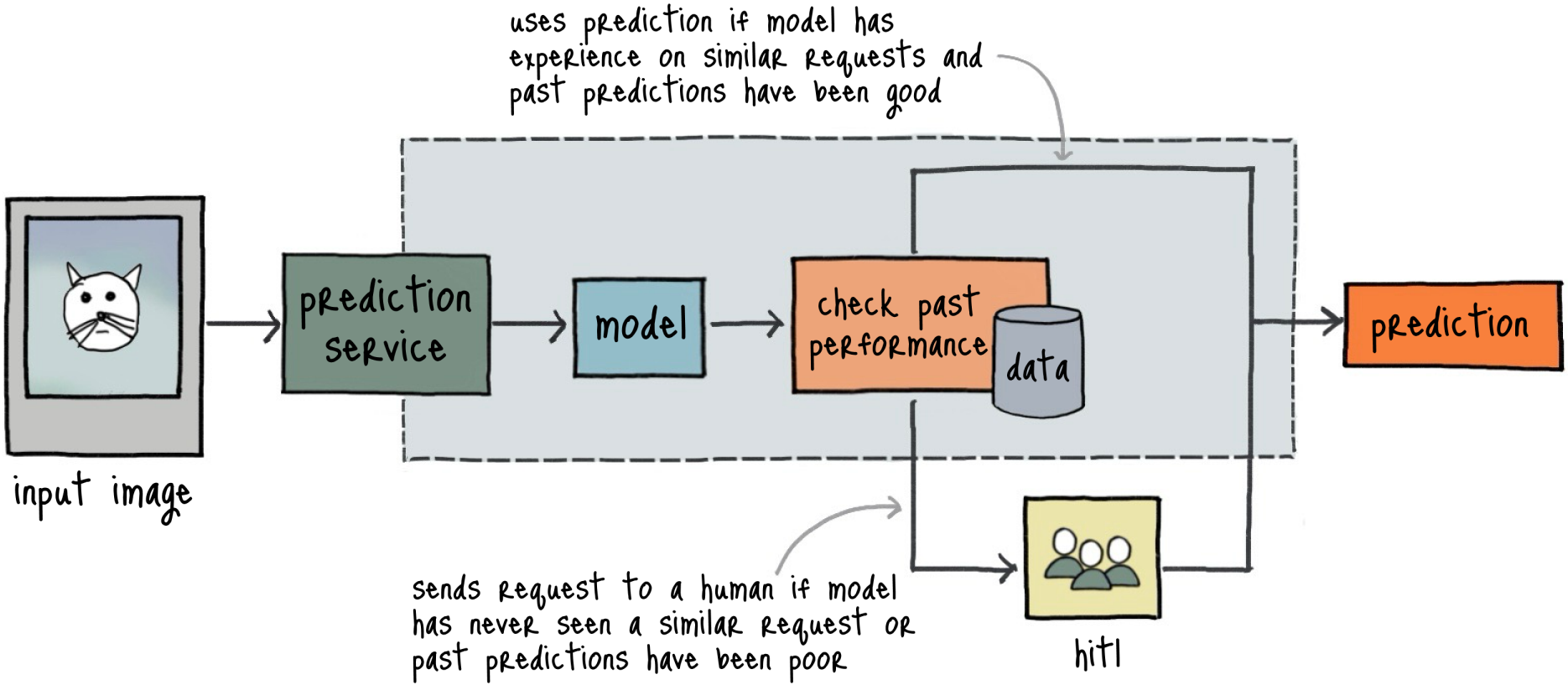
Human-in-the-loop systems combine model predictions with human expertise to enhance decision-making, especially in high-stakes applications.

human-in-the-loop



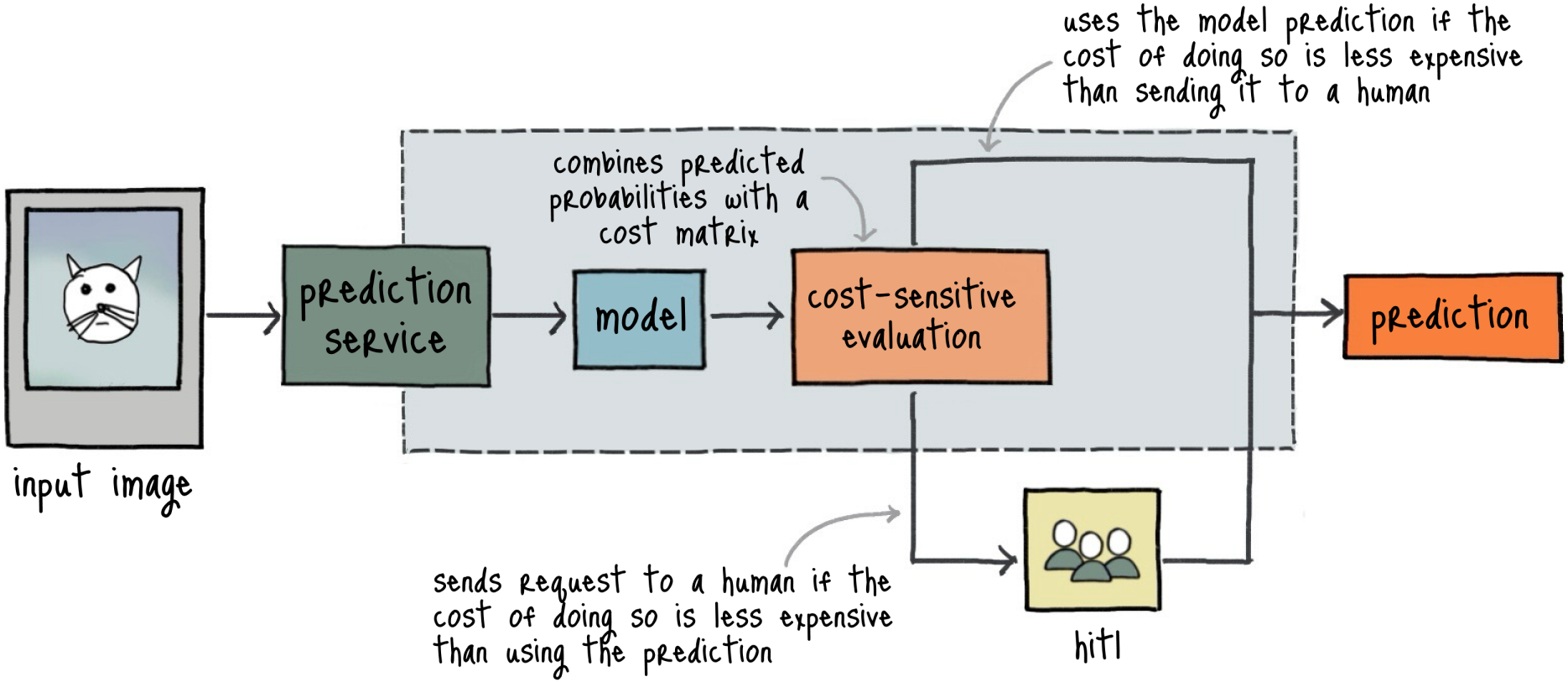
Human-in-the-loop systems use triggers such as model **confidence**, **uncertainty**, or **specific scenarios** to decide when a human review is required.

human-in-the-loop as a fallback strategy

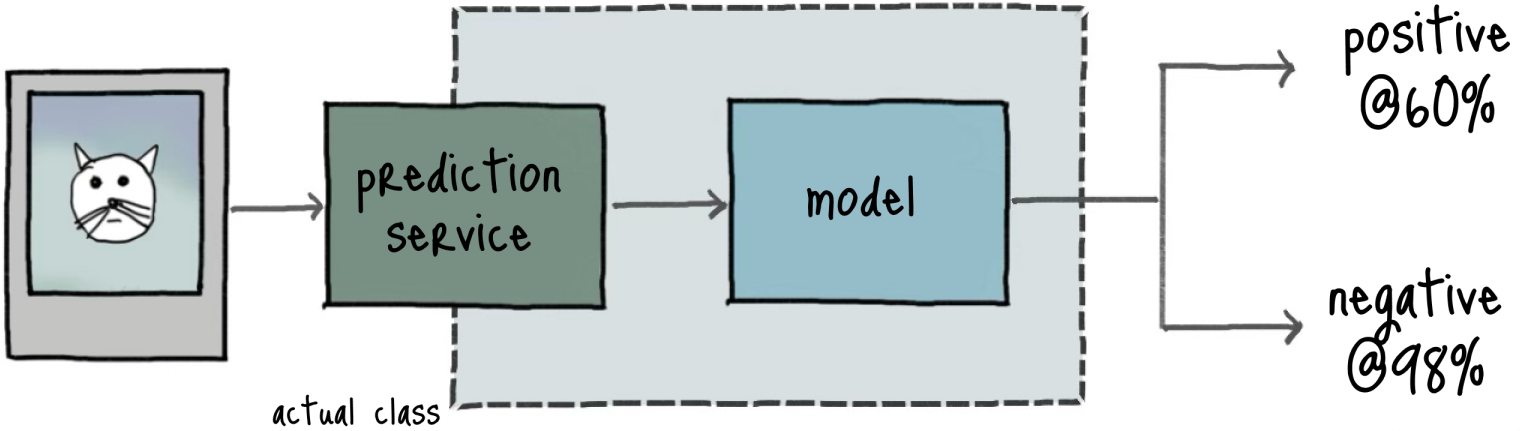


Cost-sensitive learning can be used to compute the potential cost of making a prediction and compare it with the cost of requesting a human review.

cost-sensitive human-in-the-loop

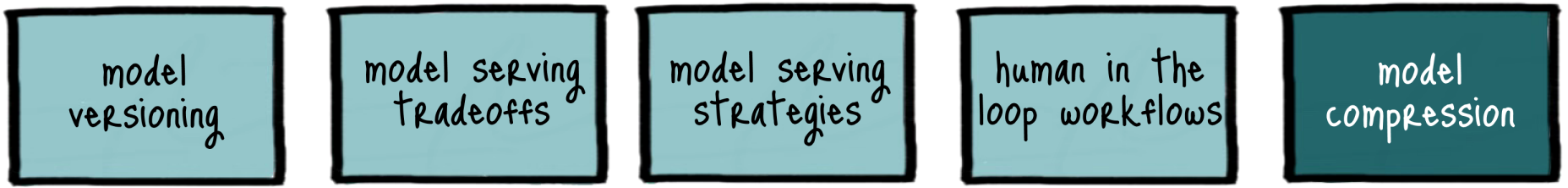


cost-sensitive human-in-the-loop

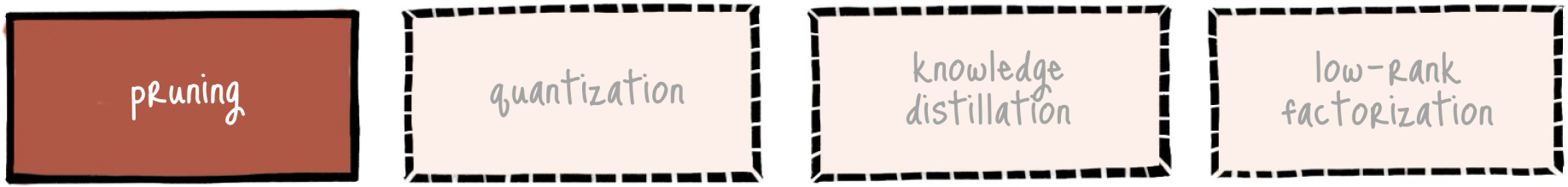


predicted class	positive	negative
positive	tp -	fp $1000
negative	fn $4000	tn -
human cost	$200	

scenario	cost-sensitive computation	decision
positive @60%	cost = (1 - 0.6) * 1000 = $400 $400 > human cost of $200	send to a human
negative @98%	cost = (1 - 0.98) * 4000 = $80 $80 < human cost of $200	make direct prediction

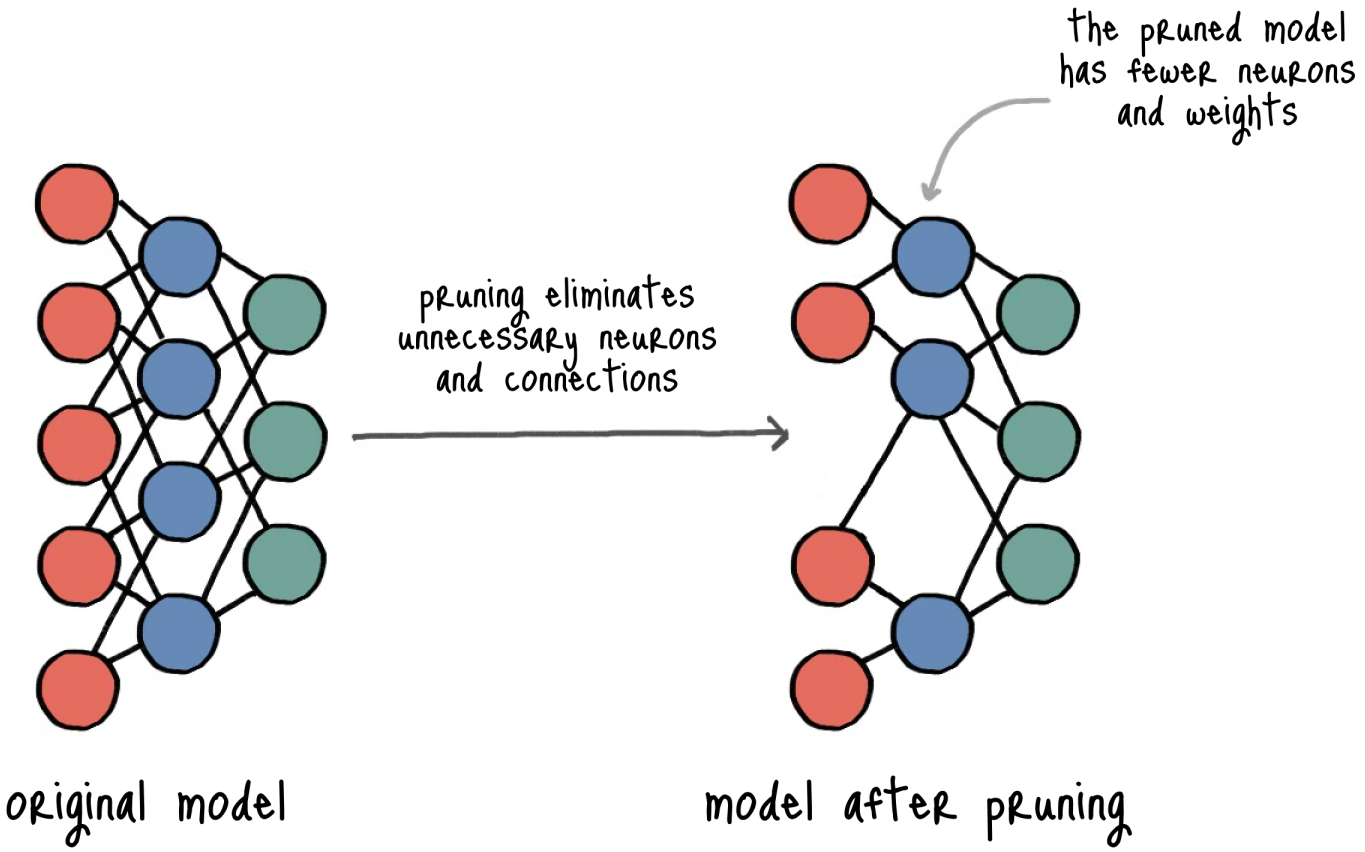


Model compression reduces a model's size and computational requirements, making it faster and more efficient for deployment without sacrificing accuracy.

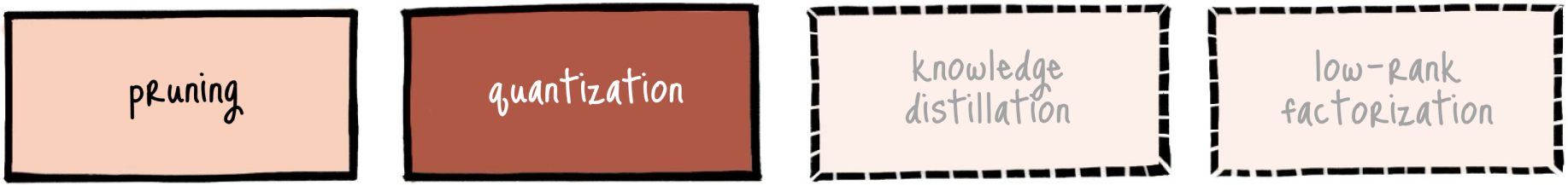


Pruning removes any unnecessary weights or neurons from a model, reducing its size and complexity while maintaining its accuracy

pruning

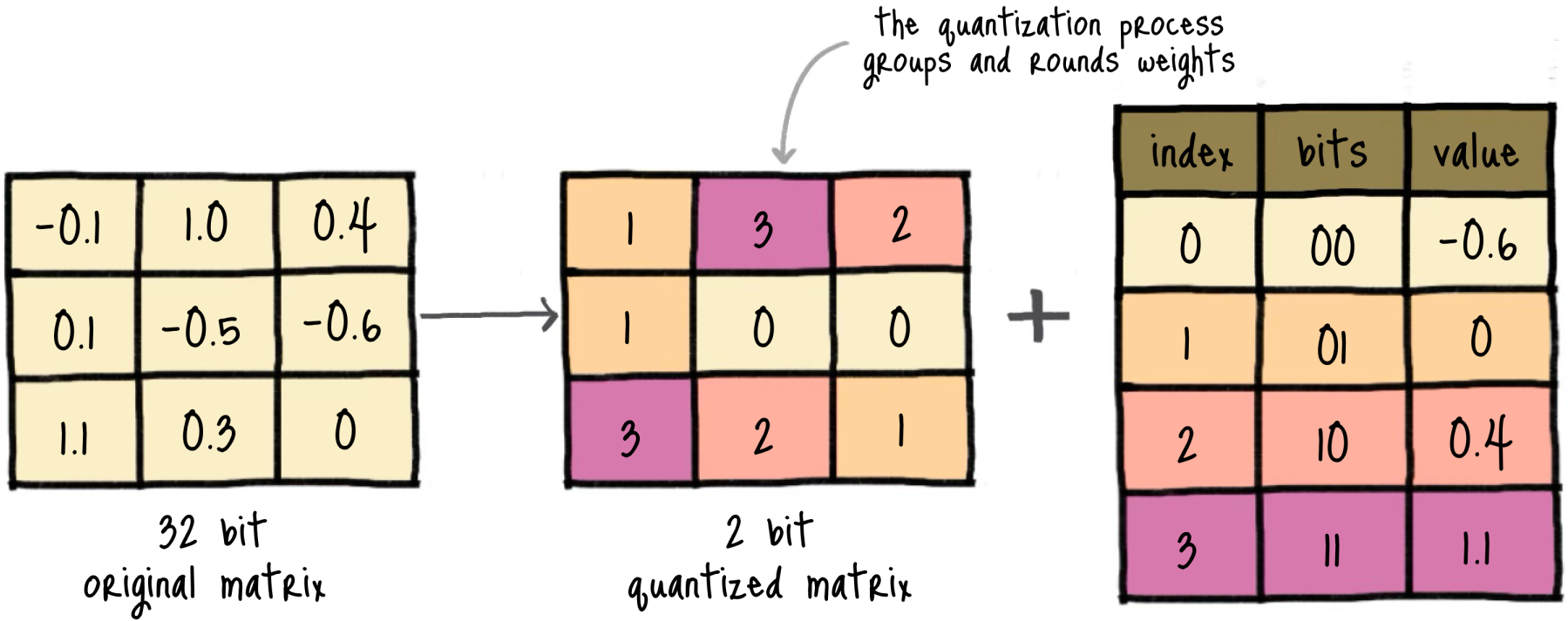

the pruned model has fewer neurons and weights
pruning eliminates unnecessary neurons and connections
original model
model after pruning

Pruning increases the sparsity of the model making it faster, but it also increases the risk of overfitting. Finding which parameters to prune is expensive.



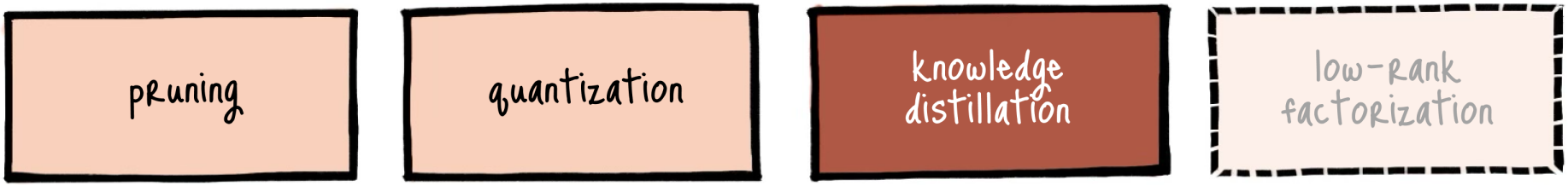
Quantization reduces the precision of the model parameters by using fewer bits to represent them. This reduces the model size and increases its speed.

quantization

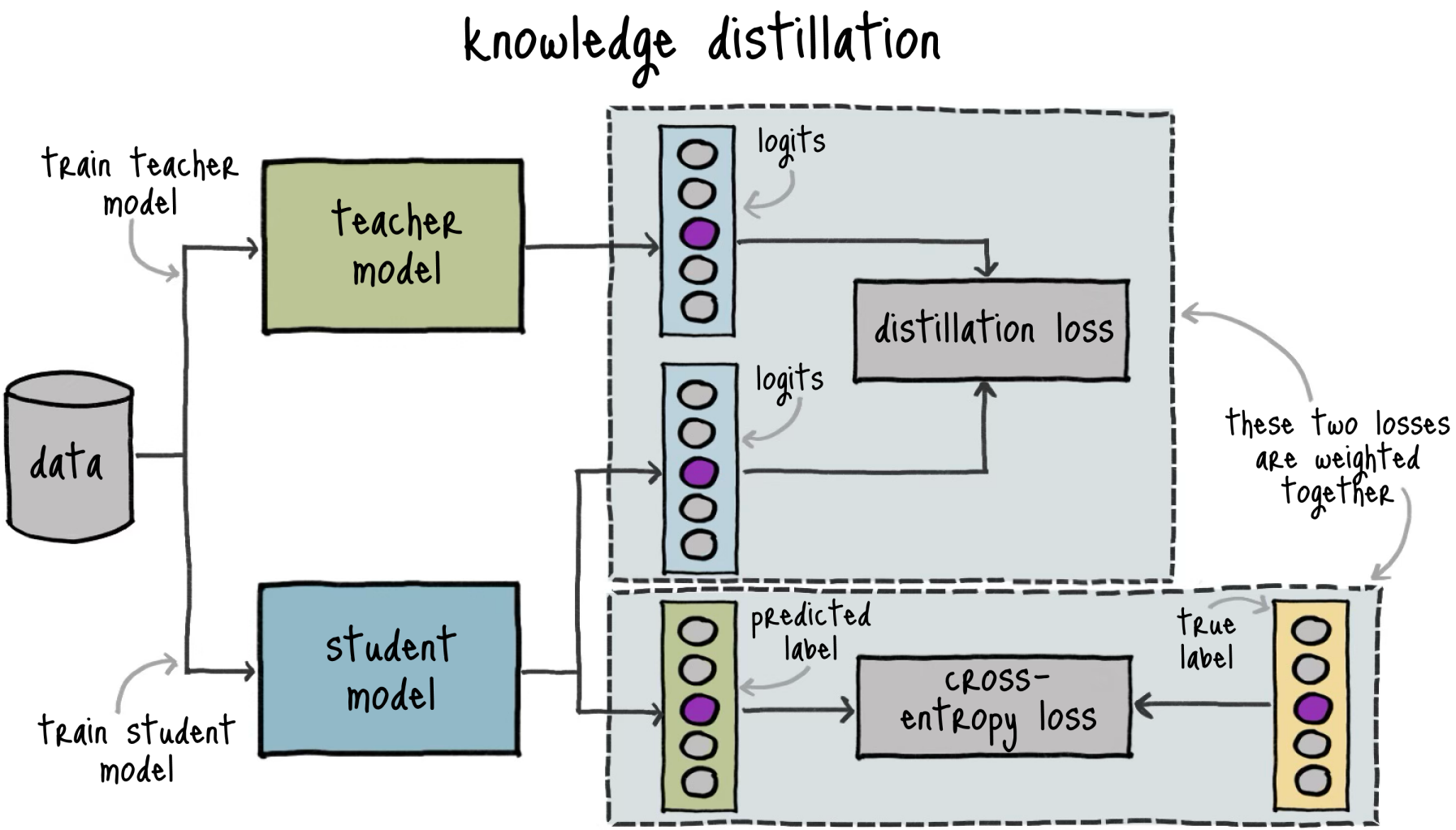


Quantized models take **less space** and run **faster**, but they can represent a **smaller** range of values, leading to more rounding errors and **worse** performance.

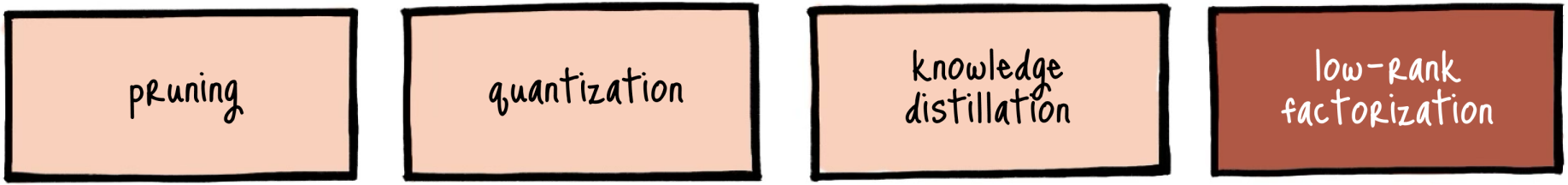
Post-training quantization is the most popular way to compress models and it's easier to use. **Quantization-aware** training is better for accuracy preservation.



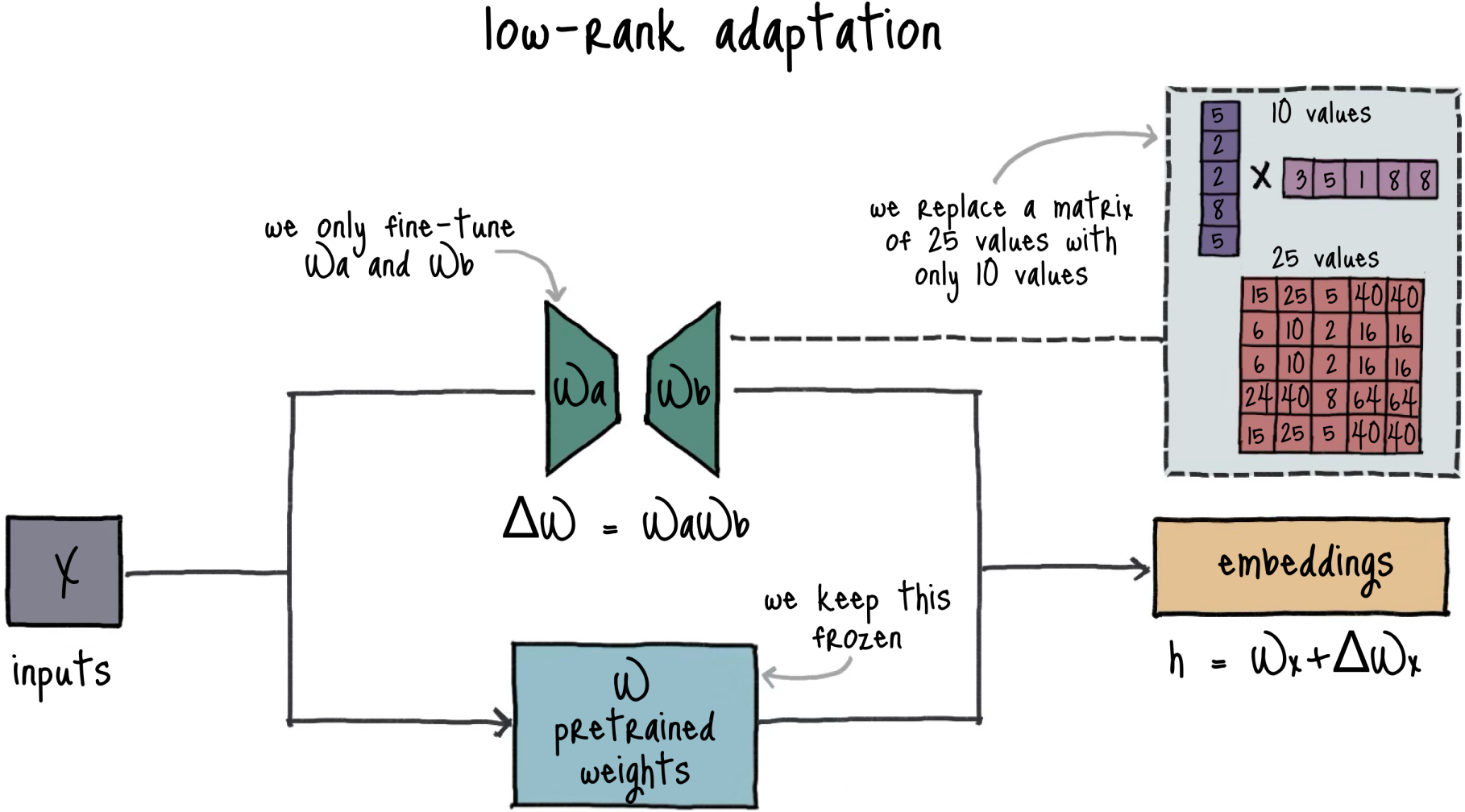
Knowledge distillation transfers the knowledge of a large model—the **teacher**—into a smaller model—the **student** while retaining its original accuracy.


knowledge distillation
train teacher model
teacher model
logits
distillation loss
logits
data
these two losses are weighted together
student model
predicted label
true label
cross-entropy loss
train student model

Knowledge distillation often requires **extensive** training time to match the teacher's performance and can lead to **reduced accuracy** in the student model.



Low-rank factorization reduces model size by decomposing large weight matrices into smaller, lower-rank matrices, making computations faster.


low-rank adaptation
we only fine-tune Wa and Wb
we replace a matrix of 25 values with only 10 values
10 values
5
2
2
8
5
X
3 5 1 8 8
25 values
15 25 5 40 40
6 10 2 16 16
6 10 2 16 16
24 40 8 64 64
15 25 5 40 40
Wa
Wb
ΔW = WaWb
X
inputs
we keep this frozen
W pretrained weights
embeddings
h = Wx+ΔWx

Low-rank factorization **saves space** and **speeds up** inference, but it can lead to **accuracy loss**, especially when working with complex models.

the end