Language Models in Cloud-Native AI

Lesson 9

LLMs, SLMs and Operational Workflows

Introduction

Introduction to API-Driven AI in Cloud-Native Systems

Modern cloud-native applications increasingly rely on AI models exposed through ready-to-use APIs. Instead of building models from scratch, organisations integrate language-understanding and language-generation capabilities by calling external or managed services. This shift matters because API-driven AI removes the burden of training, hosting, and scaling the models, allowing teams to focus on business logic while still benefiting from advanced cognitive abilities.

Role of Language Models

The lecture introduces language models as the fundamental engines behind today's cognitive services. These models interpret text, generate responses, extract information, and reason about the context contained within enterprise documents. A cloud-native system that uses summarisation, search, contract analysis, or conversational interfaces is ultimately powered by a language model sitting behind an API endpoint. This lecture prepares you to understand what those models do, why they differ in size and capability, and how they influence architectural decisions.

Large Models and Small Models

A central learning point is the difference between **Large Language Models (LLM)** and **Small Language Models (SLM)**.

- Large models offer broad general intelligence because they are trained on enormous datasets and contain a very high number of parameters.
- Small models are designed for efficiency and targeted performance. They are useful in scenarios with hardware limits, strict cost controls or a need for domain specialised behaviour.

API Based Use versus Custom Development

This is a common architectural decision. Teams must choose between using a pre trained model through an API or developing a customised model that runs in their own environment.

- API based access provides convenience, reliability and speed of adoption.
- Custom development provides control, privacy and domain specification. Many exam questions ask students to reason through this trade off.

Operational Management of Language Models

This refers to the practices required to monitor, evaluate, update and fine tune models once they are in production. Continuous measurement is essential because model behaviour changes with new data and with evolving business requirements.

Language Models

Understanding Language Models

Large Language Models are defined as deep learning systems trained on extensive text datasets so that they can recognise, extract, summarise, predict and generate natural language. Their training material typically includes books, articles and web pages, which exposes them to a wide range of vocabulary, sentence structures and semantic patterns. Through this exposure, they develop the ability to interpret context, follow reasoning, and produce text that remains grammatically correct and semantically appropriate.

How Language is Learned

Training is performed on large text corpora, which may include books, articles and web pages. Exposure to such diverse material allows the model to learn the relationships between words, the dependencies across phrases and the contextual cues that affect meaning. This process enables the model to infer what words are likely to appear next, how sentences are structured and how ideas progress. As a result, the model can answer questions, draft text, summarise content or carry out translation without being explicitly programmed for each narrow task.

Large Language Models (LLMs)

Nature of Large Language Models

Large Language Models are presented as deep learning systems trained on very extensive text datasets. Their purpose is to recognise patterns, understand context and generate text that is coherent and meaningful. Their training typically includes material from books, articles and web pages. Through this exposure, they develop the ability to perform tasks such as summarisation, translation, question answering and text completion. The defining characteristic of an LLM is the scale of its parameters, which allows it to capture subtle relationships within language.

Scale and the Meaning of Large

The designation of a model as large arises from its parameter count. Parameters are the trainable components within the neural network that adjust during training. High parameter counts allow the model to represent complex patterns in language. Typical ranges that qualify as large, with the example of GPT-3 containing **one hundred and seventy five billion** parameters. This quantity enables very fine grained modelling of syntax, semantics and context.

Role of Parameters in Language Modelling

Parameters encode the relationships between linguistic elements. When text is processed, these parameters influence how meaning is interpreted and how responses are generated. During training, the model predicts the next token in a sequence and then adjusts its parameters to reduce the prediction error. Repeated exposure to large volumes of text gradually shapes the internal structure of the model, allowing it to generate coherent answers to diverse prompts. A simple illustration may be found by comparing this process to a decision tree. In a decision tree, parameters determine how data is split at each node and what output appears at each leaf. In a language model, the parameters serve a similar role of shaping decisions, although the operations are far more complex and involve continuous numeric adjustments rather than discrete branches.

A Journey Through a Decision Tree Classifier

A simple decision tree classifier is used to illustrate how model parameters are learned during training. These parameters control the internal structure of the tree and determine how predictions are produced.

Decision node splits refer to the points within each internal node where the data is divided according to specific feature values. These split positions are learned from the training data. The goal of the learning process is to choose split points that maximise the separation between classes. Each split represents a decision that influences the subsequent flow of samples down the tree.

In the next slide we will look at some definitions regarding decision tree algorithm and then look at an example to understand how it works.

Definitions

Decision Node Splits

A decision node split is the moment in a decision tree where the data is divided into two or more groups based on a feature. It acts like a question the tree asks, for example: *“Is the price greater than 1,000?”* The tree learns these questions during training so that each split separates the data in a way that makes the classes clearer and easier to distinguish.

Leaf Nodes

A leaf node is the end point of a branch in the decision tree. Once the tree has asked all its questions along a path, the leaf node gives the final answer. For classification, this answer is usually the predicted class. Everything that flows into that leaf gets the same prediction.

Tree Depth

Tree depth is the number of steps from the root (the first question) down to the deepest leaf (the final decision). A deeper tree can capture more detailed patterns because it asks more questions, but if it becomes too deep, it starts memorising the data instead of learning general rules, which leads to overfitting.

Example Classification

Let us walk through how a decision tree classifier actually works using a simple classification problem. Suppose we have a dataset of students, each described by features such as **hours studied**, **attendance percentage**, and **number of assignments** completed. The target we want to predict is whether the student will pass an upcoming exam.

The journey begins at the root of the tree. The algorithm evaluates all available features and determines which one gives the most effective split between students who passed and those who failed. After analysing the training data, it might discover that the most informative feature is the number of **hours studied**. If students who study more than four hours per day mostly pass, while those below that threshold mostly fail, the first split becomes "*Hours studied > 4*".

Now the data is separated into two branches. On the branch where hours studied is greater than four, the tree examines the remaining features to decide whether further splitting will improve class purity. It may find that **attendance** still varies significantly, so it introduces a second decision: "*Attendance > 80%*". Students meeting both conditions (high study hours and high attendance) may form a nearly pure group where most students pass, causing this path to terminate at a leaf node labelled "Pass".

On the other side of the tree, where students study fewer than four hours, the classifier again evaluates remaining features. It may discover that **assignment completion** plays a significant role here. If students with **low study hours but more than eight completed assignments** still tend to pass, the tree adds the condition "*Assignments completed > 8*" and creates a leaf based on the dominant outcome. Those who fall below both thresholds may end up in a leaf predicting "Fail".

- By the time the model finishes building, every path from the root to a leaf represents a chain of conditions that captures different patterns present in the data.
- The tree does not rely on formulas or assumptions; it relies solely on how the training data behaves.
- During prediction, a new student simply travels down the tree. Their feature values are checked against each condition until they reach the appropriate leaf, which gives the final result.

This journey illustrates how a decision tree classifier takes raw data, learns the most discriminative feature-based splits, and converts them into a structured decision-making process. The result is a model that is easy to follow, logically coherent, and able to classify new instances according to patterns extracted directly from the data.

Difference Between Parameters and Hyperparameters

A distinction is maintained between parameters and hyperparameters.

Parameters are learned automatically during training.

Hyperparameters are chosen before training begins and control aspects of the learning process. In a decision tree, examples include maximum depth, minimum samples required to split a node and minimum samples required in each leaf. In the context of LLMs, hyperparameters influence how the model learns but do not store learned linguistic knowledge.

Parameters define the model's capacity, while hyperparameters define the conditions under which learning occurs.

Hyperparameters in Decision Trees

max_depth
This controls how *tall* the tree is allowed to grow. A deeper tree can learn very detailed patterns, but if it grows too deep, it may start memorising the training data instead of learning from it. Limiting the depth keeps the model focused on the most important patterns.

Example: If max_depth = 3, the tree can make at most three decisions in a path. Even if the data allows more splits, the tree stops growing after three levels to avoid overfitting.

min_samples_split
This sets the minimum number of data points needed before the tree is allowed to split a node into two branches. If you allow splits with only a few samples, the tree can make unreliable decisions. By requiring more samples, the tree makes splits only when they are backed by enough evidence.

Example: If min_samples_split = 10, a node must contain at least 10 samples before the tree is allowed to create a new split. A node with 8 samples will not split, even if a split looks mathematically possible.

Hyperparameters in Decision Trees

min_samples_leaf
This ensures that every leaf node (final decision point) has a minimum number of samples. If a leaf contains only one or two samples, its prediction is unstable. Setting this value prevents the tree from creating leaves that are too small, leading to more reliable predictions.

Example: If min_samples_leaf = 5, each leaf must contain at least five samples. If a potential split produces leaves with only two samples each, the tree will reject that split.

criterion
This tells the tree how to measure the quality of each possible split. Options like “gini” or “entropy” evaluate how pure or mixed the classes are after a split. The tree uses this measure to choose the most informative question at each step.

Example: If criterion = "gini", the tree will look for the split that reduces class impurity the most. If a dataset has a mix of “Pass” and “Fail”, the tree will pick the feature that creates groups with clearer separation between the two.

overfitting

Overfitting happens when a model learns the training data *too well*, including noise, random fluctuations, and patterns that do not actually generalise to new data. Instead of learning the underlying trend, the model memorises specific details that are unique to the training set. As a result, the model performs extremely well on the data it was trained on, but performs poorly when shown new, unseen data.

Example:
Suppose we train a decision tree to predict whether a student passes an exam based on hours studied and attendance. If the tree is allowed to grow very deep, it may create extremely specific rules like:

- "If attendance is 83% and hours studied is 4.1 → Pass"
- "If attendance is 82% and hours studied is 4.0 → Fail"
- "If attendance is 82% but hours studied is 4.05 → Pass"

These splits may perfectly explain the training data, but they are too specific and based on tiny differences that do not matter in real life. When the model sees a new student with slightly different values, it may get confused and make incorrect predictions.

In short, **overfitting is when the model memorises the answers instead of learning the concepts**. Controlling hyperparameters like max_depth, min_samples_split, and min_samples_leaf helps prevent this by stopping the tree from becoming overly detailed.

Training Data Sources

A key dataset is **Common Crawl**, which consists of hundreds of billions of web pages collected over several years. Such datasets expose the model to diverse topics, writing styles and domains. This variety allows the model to generalise across many tasks and produce responses that remain contextually appropriate in different situations.

When a model is trained on Common Crawl, it is exposed to:

Wide Topical Coverage

The dataset contains content from domains such as e-commerce, computers and electronics, politics, medicine, religion, automobiles, and sports. Each domain contributes its own vocabulary, expressions, tone, and structure. This breadth allows the model to understand questions or prompts from many subject areas, even if the topics differ significantly from one another.

Multiple Writing Styles and Formats

Web text includes product reviews, news articles, blogs, FAQs, discussion posts, guides, and corporate pages. Exposure to these formats teaches the model how language is used informally and formally, casually and professionally, briefly and in long paragraphs. This supports generalisation across tasks such as summarisation, answering questions, explaining concepts, or rewriting text.

Multilingual Content

The dataset contains both English and non-English pages. This multilingual presence enables models to develop some degree of cross-lingual understanding. Even when fine-tuned for English tasks, the model benefits from having encountered multiple languages because it learns general principles of sentence structure, word patterns, and semantic relationships.

Large Quantities of Text

The scale of the dataset matters because the model encounters countless examples of how words are used together in different contexts. This repeated exposure teaches the model to identify subtleties such as tone, implied meaning, and sentiment. It also helps the model handle unusual or rare expressions, since large corpora increase the chance of encountering them.

Real-World Diversity

The web naturally contains inconsistent phrasing, errors, abbreviations, slang, and incomplete sentences. Although imperfect, this diversity strengthens a model’s ability to deal with real inputs from users, which often contain informal language, spelling variations, or mixed expressions.

Tokenisation and Processing Units

Language models operate on tokens, which are the smallest text units they process. Tokens may represent whole words or subdivided word fragments.

A word like *unbelievable* is broken into parts such as *un*, *believ*, and *able* because the model cannot store every possible word in its vocabulary. Instead, it keeps a smaller set of common pieces. These pieces appear in many words, so the model learns them well. When a long or uncommon word appears, it is rebuilt from these familiar parts, allowing the model to understand it without needing the full word stored separately.

Token count affects both the cost of processing and the limits of the model's context window. A prompt with a response consumes a defined number of tokens, and this total determines how much input the model can process in a single interaction.

Examples of Prominent LLMs

- GPT-4 (OpenAI)
- LLaMA 3 (Meta)
- Claude 3 (Anthropic)
- Mistral 7B
- PaLM 2 [Google - primarily text]
- Gemini [Google - Multimodal]
- Grok (xAI) [Elon Musk]
- ERNIE 4.0 (Baidu) [Chinese]
- Falcon (Technology Innovation Institute)

Small Language Models (SLMs)

Large Language Models are extremely big. They contain billions or even trillions of parameters, which means they need powerful hardware, a lot of memory, and significant computing time to run.

Small Language Models work at a much smaller scale. Instead of billions, they may have only thousands to a few million parameters. Because they are lighter, they can run on ordinary machines, laptops, or small servers.

This smaller size makes them more practical for organisations or researchers who cannot afford the heavy computational demands of very large models. SLMs are easier to deploy, cheaper to operate, and more suitable for targeted or domain-specific applications where a giant model is not necessary.

SLM Advantages

Small Language Models give more **control and tailoring**, meaning the model can be shaped closely to the organisation's specific needs. Since the model is smaller, fine tuning becomes easier and faster, allowing it to specialise in tasks such as legal text processing, customer support, medical summaries, or policy analysis.

They provide **enhanced security**, because the entire model can often be hosted inside the organisation's own infrastructure. Sensitive data does not need to be sent outside, which is important for compliance and privacy.

SLMs also offer **effective performance** for focused tasks. A smaller model that is fine tuned can outperform a large general model in domains where precision and relevance matter.

They support **scalability**, because more users or applications can run the model without requiring heavy hardware. The reduced size makes deployment and expansion straightforward.

Rapid prototyping becomes possible. Teams can test ideas, build internal tools, or create proof of concepts quickly without waiting for long training cycles or high cost compute resources.

Finally, SLMs deliver **cost efficiency**. They require less memory, cheaper hardware, and lower operational expenses.

Differences Between LLMs and SLMs

Large Language Models contain billions or even trillions of parameters. This scale allows them to understand broad contexts, handle complex reasoning, and generate detailed responses across many domains. However, they require powerful hardware, large memory, and significant computational resources. Running or fine tuning such models is expensive, and they are usually deployed in large cloud environments.

Small Language Models contain far fewer parameters, often in the range of thousands to a few million. Because of this, they need much less computing power and can run on ordinary hardware. They are easier to fine tune for domain specific tasks and can be deployed within an organisation’s own infrastructure. Their smaller size provides cost efficiency, faster experimentation, and better control over data and security.

In summary, LLMs offer broad capability but demand substantial resources, while SLMs offer practical, efficient solutions for focused tasks with lower computational requirements.

Popular SLM Models

DistilBERT: A smaller version of Google’s BERT. Used for: text classification and question answering.

TinyBERT: Even smaller than DistilBERT. Used for: sentiment analysis, question answering, and text summarisation.

MiniLM (Microsoft): Designed for lightweight information retrieval tasks.

GPT-NeoX-20B Lite: A lighter variant of the GPT-NeoX family.

TinyGPT: Extremely small, designed for simple language tasks.

Phi-3.5-mini-instruct (Microsoft): A compact instruction-tuned model. Used for: summarisation, question answering, dialog and chatbots, text generation, text classification, and code-related queries.

LLMOps

LLMOps refers to the set of practices used to manage Large Language Models throughout their lifecycle in production. It focuses on how an organisation selects a model, tests it, improves it, deploys it, and monitors it, just like DevOps does for software systems. The goal is to ensure that LLM-based applications run reliably, stay accurate, and can be improved over time.

A real life example is an enterprise customer support chatbot built on a large language model. The organisation first selects a suitable foundational model. The team designs prompts to guide the chatbot’s behaviour and checks whether the answers match company guidelines. If gaps appear, the model may be fine tuned using past support conversations. Once the chatbot meets the quality expectations, it is deployed gradually using methods such as A/B testing or canary rollout, so only a small group of employees interacts with the updated version at first. After deployment, its responses are monitored to ensure accuracy and detect drift. If issues arise, the prompts or the fine tuned model are updated, and the cycle continues.

This closed loop of selection, refinement, controlled deployment, and monitoring illustrates how LLMOps supports the stable use of language models in real applications.

Development to Production Workflow

The workflow begins by selecting a foundational model. Once the model is chosen, prompt engineering is used to shape how the model behaves. The results are evaluated to check if the responses meet the required quality.

If there is room for improvement, the next step is to decide whether a team and labelled data are available. If both are available, the model can be fine tuned to match the organisation's domain needs. If not, the process loops back to adjust prompts and evaluate again.

Once the output quality is acceptable, the model is deployed to production. At this stage, deployment strategies such as A/B testing, canary releases, and blue-green deployments are used to safely roll out updates. These methods help test changes on a smaller group of users before making them available system-wide.

This workflow ensures that LLM-based systems deliver reliable value while allowing continuous improvement.