SAMPLE

Building the Natural Language Processing of Tomorrow

A Practical Guide to the Technology Teaching Machines to Understand Language

Aiman JoeHanz

Free sample: Introduction & Chapter 1

putrabooks.github.io

Buy the full book:

Carousell carousell.sg/p/building-the-natural-language-processing-of-tomorrow-ebook-pdf-1465169103

Gumroad

Ko-fi

ElevenReader

Payhip

Contents

Introduction

Chapter 1: The Current State of Natural Language Processing

Chapter 2: Where Today's NLP Falls Short

Chapter 3: Common Mistakes in NLP Projects

Chapter 4: Foundational Principles of NLP Design

Chapter 5: Building NLP Applications, One Step at a Time

Chapter 6: Implementation Strategies for Production NLP Systems

Chapter 7: Technology Deep Dive: Transformers, Attention, and the LLM Stack

Chapter 8: Case Studies: Real-World NLP Deployments

Chapter 9: Evaluating and Testing NLP Systems

Chapter 10: The Future of NLP

Chapter 11: Action Plan: Building Your NLP Practice

Chapter 12: Advanced Topics: Safety, Alignment, and Responsible NLP

Chapter 13: Building an NLP-Ready Organization

Frequently Asked Questions

Glossary

Bibliography

Appendix A: NLP Architecture Patterns in Detail

Appendix B: Evaluation Techniques in Depth

Appendix C: Cost Optimization Techniques

Appendix D: NLP Project Anti-Patterns

Appendix E: The RAG Quality Optimization Guide

Appendix F: The Model Selection Decision Guide

Appendix G: Production Deployment Patterns

Appendix H: The NLP Team Operations Guide

This free sample contains the Introduction and Chapter 1. The complete book contains

the full contents above.

Introduction

Natural language processing has undergone a revolution. In 2018, the introduction of the Transformer architecture changed the trajectory of the field. Models that once required task-specific architectures and months of feature engineering could now be pre-trained on massive text corpora and fine-tuned for any language task. By 2022, large language models could generate fluent, coherent text, answer questions, write code, and reason about complex problems—capabilities that seemed impossible just five years earlier.

This book is for people who need to understand NLP as an engineering discipline, not as a research curiosity. Software engineers building NLP features. Product managers evaluating language AI for their products. Technical leaders deciding whether and how to adopt LLM technology. Data scientists transitioning from traditional NLP to the LLM era. Anyone who wants to move beyond the hype and understand what NLP actually does, how it works, and where it is going.

The field has changed so rapidly that much of the conventional wisdom is outdated. Fine-tuning a BERT model on task-specific data was the state of the art in 2020; today, a single API call to a frontier LLM often outperforms a fine-tuned BERT model. But the new capabilities come with new challenges—hallucination, cost, latency, safety, and the difficulty of evaluating systems that generate open-ended text. This book covers both the capabilities and the challenges, with practical guidance for building production NLP systems.

This book covers NLP as an engineering discipline. It treats NLP as a technology stack—models, embeddings, vector databases, retrieval, fine-tuning, evaluation, and monitoring—with specific capabilities, limitations, and trade-offs. It covers the current state of NLP technology, including the models, tools, and patterns in use today. It covers the limitations and failure modes that production systems encounter. It covers the engineering practices that distinguish successful NLP deployments from demos.

The book follows a progression from understanding to action. Chapters 1-3 cover the current state, limitations, and common mistakes. Chapters 4-6 cover design principles and implementation strategies. Chapter 7 provides a technical deep dive into the Transformer architecture and the LLM stack. Chapters 8-9 cover case studies and evaluation. Chapter 10 covers the future of the field. Chapters 11-13 cover action plans, safety, and organizational readiness.

This book assumes programming experience and basic understanding of machine learning concepts. No NLP background is required—the book explains

NLP concepts from first principles. However, readers with NLP background will find the coverage of LLM-era patterns, evaluation techniques, and production challenges valuable.

Familiarity with Python, basic linear algebra, and general software engineering practices. Experience with APIs, databases, and production systems is helpful but not required.

Chapter 1: The Current State of Natural Language Processing

NLP is the field of computer science that deals with understanding and generating human language. In practice, NLP systems perform a set of core tasks: classification (assigning text to categories), extraction (pulling specific information from text), generation (producing new text), translation (converting text between languages), summarization (condensing text), question answering (providing answers to questions based on text), and search (finding relevant documents).

Before the LLM era, each task required a different model, different training data, and different evaluation metrics. A sentiment analysis system could not translate text. A summarization system could not answer questions. The LLM era changed this: a single model can perform all of these tasks, often with better quality than task-specific models. This is both the promise and the complexity of modern NLP—you get more capability with less engineering, but you also get less control over how the model behaves.

That trade — capability for control — is the defining engineering tension of the era, and it deserves to be treated as a design constraint rather than a complaint. A task-specific model is legible: its behavior is bounded by its training distribution, and when it errs, the error is usually in-distribution and diagnosable. An LLM is a general system whose failure surface is not enumerable in advance; the same generality that lets it handle an unanticipated phrasing lets it handle an unanticipated domain — including ones it was never meant to touch. Production design under this constraint means compensating elsewhere: constraining inputs, validating outputs, and instrumenting behavior, because the model itself can no longer be the boundary. The teams that succeed are the ones that stop trying to make the model predictable and start making the system robust to its unpredictability.

The shift from task-specific models to general-purpose LLMs has profound implications for how NLP systems are built. Instead of training a model from scratch for each task, you prompt a pre-trained model. Instead of collecting labeled data for training, you collect a golden set for evaluation. Instead of optimizing model architecture, you optimize prompts and retrieval. The engineering effort shifts from model training to system design—choosing the right model, building the right pipeline, and evaluating the right metrics.

The shift has a consequence for team composition that many organizations still haven't absorbed: the scarce skill is no longer the researcher who can design a loss function but the systems engineer who can architect, measure,

and iterate a pipeline with a stochastic component in the middle. Prompt and retrieval work look unglamorous beside model training, but they carry the same weight production carried in every previous generation — the difference is that the artifact being tuned is no longer weights but the surrounding scaffolding: what context the model sees, in what order, with what instructions, and against what checks. "System design" is not a demotion of engineering; it is engineering returning to its actual subject, which was never the model — it was always the system that makes the model useful.

The Transformer architecture, introduced in the 2017 paper "Attention Is All You Need" by Vaswani et al., is the foundation of modern NLP. The key innovation was the self-attention mechanism, which allows the model to weigh the importance of different words in a sentence when producing a representation. Unlike recurrent neural networks (RNNs) that process text sequentially, Transformers process all tokens in parallel, enabling both better performance and more efficient training.

The Transformer's parallel processing capability was crucial for scaling. Training a model on billions of tokens requires massive parallelism, which RNNs could not efficiently exploit. Transformers could be trained on GPU clusters with thousands of processors, enabling the scale of data and computation that produced models like GPT, BERT, and their successors. Without the Transformer architecture, the LLM era would not have been possible.

The historical lesson inside that sentence generalizes to a portfolio principle: progress in AI has repeatedly hinged not on cleverer learning rules but on architectures that expose more parallelism to available hardware. RNNs were not wrong about language — they were wrong about hardware, serializing a computation the GPU era was built to parallelize. The pattern is worth internalizing by anyone planning a career or a roadmap around these systems: when a new architecture arrives, the first question is not "is it more accurate?" but "does it map better onto the hardware curve?" The architectures that win are almost always the ones that convert silicon improvements directly into model improvements.

The Transformer architecture has several variants. Encoder-only models (BERT)

are optimized for understanding tasks—classification, extraction, similarity. Decoder-only models (GPT) are optimized for generation tasks—text completion, question answering, code generation. Encoder-decoder models (T5, BART) are optimized for sequence-to-sequence tasks—translation, summarization. The dominant architecture today is decoder-only, as scaling decoder-only models has proven to produce the most general-purpose

capabilities.

The decoder's victory has a root cause worth knowing, because it explains much of the field's behavior: next-token prediction is a learning signal available at unlimited scale. An encoder trained on classification needs labels — a bounded, expensive resource — while a decoder trained on next-token prediction can learn from every token of every document ever digitized, because the label is simply the next token. Scale proved to be the decisive variable, and the architecture whose supervision was free-scale-compatible won the era. Understanding this explains why capabilities keep surprising: nobody curated what the model knows; the curriculum was the raw internet, and the model's breadth is a reflection of that corpus rather than any designer's intent.

The LLM era began with GPT-2 in 2019, which demonstrated that a large decoder-only Transformer trained on web text could generate coherent, contextually appropriate text. GPT-3 in 2020 showed that scaling to 175 billion parameters enabled few-shot learning—the model could perform new tasks with just a few examples in the prompt, without any fine-tuning. ChatGPT in 2022 made LLMs accessible to the general public and demonstrated that RLHF (Reinforcement Learning from Human Feedback) could make LLMs follow instructions and produce helpful, safe responses.

The LLM era is characterized by several shifts from traditional NLP. Pre-training replaces task-specific training: models are trained on massive text corpora to predict the next token, learning general language understanding. Prompting replaces feature engineering: instead of designing features and training models, you write instructions and examples. Fine-tuning replaces architecture design: instead of designing model architectures, you adapt pre-trained models to specific domains. Evaluation replaces training as the primary challenge: the hardest part of production NLP is no longer training the model but evaluating whether the output is correct.

The inversion deserves to be stated in its full difficulty, because it is what makes modern NLP engineering unlike most software engineering: the developer no longer knows what correct looks like at generation time, and correctness varies per input, per run, and per evaluator. Traditional software tests assert exact outputs against fixed inputs — an approach that dies instantly when the component in the middle is a probability distribution over plausible text. The field's response — golden sets, rubric-based grading, pairwise comparison, statistical sampling of production traffic — is still young, and teams consistently underestimate how much of their schedule evaluation will consume. A useful planning number: teams that treat evaluation as a side

task spend roughly a third of their post-launch engineering hours fighting quality regressions they lack the instrumentation to even detect.

The economic implications are significant. Pre-training a frontier LLM costs millions of dollars in compute, creating a high barrier to entry. But using a frontier LLM via API costs cents per query, making NLP capabilities accessible to any application. The economics favor a small number of organizations training frontier models and a large number of organizations building applications on top of them.

This division of labor has a strategic implication for every organization below the frontier: the model layer is a commodity in slow motion — capabilities improve quarterly and prices fall continuously, and competitive advantage migrates to the layers above it, where data, distribution, workflow integration, and evaluation are defensible. A company spending its strategic budget fine-tuning models that a provider will obsolete in six months is renting a moat the landlord drains annually. The durable assets for an application builder are proprietary data, the golden sets and evaluation harness that encode quality standards, and the workflows the product is embedded in — none of which arrive free with an API key.

Before LLMs, the NLP technology stack consisted of several layers. Tokenization split text into words or subwords. Part-of-speech tagging identified the grammatical role of each word. Named entity recognition identified people, organizations, and locations. Dependency parsing identified the grammatical structure of sentences. Word embeddings (Word2Vec, GloVe) represented words as vectors capturing semantic relationships. Task-specific models (often BERT or BiLSTM) were trained for each application.

This stack was powerful but limited. Each component was a separate model, requiring separate training data and evaluation. Components had to be chained together, with errors propagating through the pipeline.

The old stack's failure mode is worth remembering precisely because it names the risk the new stack reintroduces in new clothing. Error propagation through a chain of specialized models meant the POS-tagger's mistake corrupted the parser's input, and no downstream component could recover — the system's accuracy was bounded by its weakest early link. Modern pipelines replace specialized links with retrieval, ranking, generation, and validation, and the pathology returns: a bad chunking decision poisons retrieval, an irrelevant document misleads generation, and the confident final answer launders the upstream error. The lesson survives the architecture: in any pipeline with a learned component mid-chain, quality work concentrates at the seams, because errors never stay where they were made. The system could only

handle the specific tasks it was trained for—adding a new capability meant training a new model. The system required deep NLP expertise to build and maintain.

The LLM stack is simpler and more powerful. A single LLM replaces the entire pre-LLM stack—tokenization, POS tagging, NER, parsing, embeddings, and task-specific models are all subsumed by the LLM. The LLM stack adds new components: embedding models for RAG, vector databases for storage, retrieval pipelines for knowledge access, and evaluation infrastructure for quality measurement. The complexity moves from model training to system design.

However, the pre-LLM stack still has a role. For simple, well-defined tasks (keyword matching, regex extraction, basic classification), traditional NLP is cheaper, faster, and more reliable than an LLM. A regex-based email parser will always correctly extract email addresses; an LLM might hallucinate. The best systems combine traditional NLP for deterministic tasks with LLMs for tasks requiring language understanding.

The model landscape in 2026 includes several tiers. Frontier models (GPT-5, Claude 4, Gemini 2) represent the state of the art in capability, with the largest parameter counts, longest context windows, and best performance on benchmarks. They are available via API at $5-15 per million input tokens and $15-75 per million output tokens.

Open-source models (LLaMA 3, Mistral Large, Qwen 2.5) approach frontier model quality for many tasks and can be self-hosted. LLaMA 3 70B approaches GPT-4 quality on many benchmarks while being self-hostable on a single A100 GPU. Mistral 7B and LLaMA 3 8B are small enough to run on consumer hardware while providing strong quality for many tasks.

Specialized models serve specific niches. Code models (Code LLaMA, StarCoder) are optimized for programming languages. Embedding models (BGE, E5, OpenAI embeddings) are optimized for producing vector representations of text. Vision-language models (GPT-4V, LLaVA) process images and text together. Speech models (Whisper, Bark) handle speech-to-text and text-to-speech.

The model landscape is rapidly evolving. New models are released monthly, and model quality improves continuously. The strategy for production systems should be model-agnostic: design the system so that models can be swapped without changing the application code. This is the separation of concerns principle applied to model selection.

NLP applications in production span several categories. Customer service

chatbots handle user queries, reducing support costs and providing 24/7 availability. Document analysis systems extract information from contracts, invoices, and reports. Content generation systems produce marketing copy, documentation, and creative writing. Code generation assistants help developers write, debug, and understand code. Search and retrieval systems find relevant information in large document collections. Translation systems convert text between languages. Summarization systems condense long documents into brief summaries. Classification systems categorize text into predefined categories.

Each application has different requirements for accuracy, latency, cost, and safety. A customer service chatbot needs low latency (under 3 seconds) and high safety. A document analysis system needs high accuracy and can tolerate batch processing. A code generation assistant needs high correctness and can tolerate higher latency. Understanding these requirements is the first step in choosing the right NLP approach.

The application landscape is expanding as LLMs become more capable. Agentic systems—LLMs that use tools, browse the web, and complete multi-step tasks—are an emerging category. These systems combine LLMs with external tools (APIs, databases, code execution) to perform tasks that require both language understanding and action. The agent pattern is powerful but introduces new challenges around safety, reliability, and cost.

Retrieval-Augmented Generation (RAG) is the dominant pattern for building production NLP systems. The RAG pattern works by retrieving relevant documents from a knowledge base, adding them to the LLM prompt as context, and generating a response grounded in the retrieved documents. RAG addresses several critical limitations of LLMs: it provides up-to-date information (the knowledge base can be updated without retraining), domain-specific information (the knowledge base contains the organization's data), and verifiable responses (the system can cite the source of each claim).

The RAG pipeline consists of several stages. Documents are collected from internal sources (wikis, FAQs, documentation, databases). Documents are chunked into smaller pieces (typically 500-1000 tokens) with overlap. Chunks are embedded using an embedding model and stored in a vector database. At query time, the user's question is embedded, the vector database is searched for the most similar chunks, and the top-k chunks are added to the LLM prompt. The LLM generates a response using the retrieved context.

RAG is not without challenges. The retrieval quality determines the response quality—if the retrieval misses the relevant documents, the LLM cannot use them. Chunking strategy affects retrieval—chunks that are too small lose

context, chunks that are too large dilute relevance. The knowledge base requires maintenance—new documents must be added, outdated ones removed, and embeddings regenerated when the embedding model changes.

Despite these challenges, RAG is the most practical way to build production NLP systems that require factual accuracy and domain knowledge.

Fine-tuning adapts a pre-trained LLM to a specific task or domain. Full fine-tuning updates all model parameters on task-specific data, providing maximum quality but requiring significant compute (multiple GPUs for days or weeks). Parameter-efficient fine-tuning (LoRA, QLoRA) updates a small number of additional parameters while keeping the pre-trained weights frozen, reducing compute requirements by 90%+ while retaining most of the quality.

Fine-tuning is appropriate when the task requires a specific style, format, or domain reasoning that prompting alone cannot achieve. For example, fine-tuning can teach a model to respond in a specific brand voice, format outputs as JSON, or reason about domain-specific concepts. Fine-tuning is not appropriate for adding knowledge—that is better handled by RAG, which is cheaper, faster, and easier to maintain.

The fine-tuning workflow includes several steps. Collecting training data (typically 500-10,000 input-output pairs). Formatting the data as instruction-response pairs in JSONL format. Choosing a base model and fine-tuning method. Training on GPU infrastructure (or using API-based fine-tuning). Evaluating on the golden set. Deploying the fine-tuned model. The entire workflow typically takes 1-2 weeks for a first iteration.

The agent pattern extends LLMs beyond text generation to taking actions. An

LLM agent receives a task, plans the steps needed to complete it, uses tools (APIs, databases, code execution) to execute the steps, observes the results, and adjusts its plan based on the observations. The agent pattern is powerful for tasks that require both language understanding and action—booking flights, managing calendars, executing trades, processing refunds.

The agent stack includes several components. The LLM provides reasoning and planning. Tools are functions the agent can call (search, calculator, database query, API call). Memory stores context across multiple steps. Planning breaks the task into subtasks. Execution calls tools and collects results. Reflection evaluates progress and adjusts the plan.

Agent systems introduce significant challenges. Safety is critical—an agent that can take actions can cause real harm. Cost is a concern—agents make multiple LLM calls per task, increasing token usage. Reliability is challenging—

agents can get stuck in loops, fail to recover from errors, or take unexpected actions. For production agent systems, bounded actions (limiting what the agent can do), human approval for high-stakes actions, and comprehensive logging are essential.

NLP has historically been English-centric. Pre-training data is predominantly English, benchmarks are in English, and model quality is highest for English. The multilingual challenge is making NLP work equally well for all languages, particularly low-resource languages that have less training data available.

Modern LLMs handle multiple languages better than earlier models, but quality varies. A model that performs at 95% accuracy on English tasks may perform at 70-80% on other languages. For production systems in non-English languages, evaluation on the specific language is critical. Techniques for improving multilingual performance include: fine-tuning on language-specific data, using multilingual embedding models for RAG, and translating queries to English before processing.

The cultural dimension adds complexity. NLP systems must understand not just the language but the cultural context—politeness conventions, idioms, date formats, measurement units. A chatbot that works well in English may fail in Japanese not because of language quality but because it does not understand Japanese business etiquette.

Evaluating NLP systems is fundamentally harder than evaluating traditional software. Traditional software produces deterministic output—given the same input, the output is always the same. NLP systems produce probabilistic output—the same input may produce different outputs on different runs, and the quality of the output is subjective.

The evaluation challenge has several dimensions. There is no single metric that captures NLP quality—different tasks need different metrics (BLEU for translation, ROUGE for summarization, F1 for classification). Human evaluation is the gold standard but is expensive and slow. Automated metrics are fast but may not capture quality dimensions that humans care about. LLM-as-judge provides a middle ground but introduces its own biases.

Production evaluation requires a multi-layered approach: a golden set of known-good outputs for regression testing, LLM-as-judge for rapid iteration, human evaluation for final validation, user feedback for production monitoring, and slice-based evaluation to identify performance gaps across different input types.

NLP technology is creating significant economic value. Customer service automation reduces support costs by 30-70%. Document analysis systems

reduce manual processing time by 50-90%. Content generation systems increase content production throughput by 5-10x. Code generation assistants improve developer productivity by 20-40%. These productivity gains translate to billions of dollars in cost savings and revenue generation.

The economic impact is not evenly distributed. Organizations that invest in NLP capabilities—data infrastructure, engineering expertise, evaluation systems—capture more value. Organizations that treat NLP as a black box (calling an API without building evaluation or monitoring) see initial gains that plateau or degrade over time. The difference between successful and unsuccessful NLP deployments is not the model but the engineering practice around the model.

The NLP ecosystem is divided between proprietary models (GPT, Claude, Gemini) and open-source models

(LLaMA, Mistral, Qwen). Proprietary models offer the best quality, ease of use (API access), and no infrastructure requirements. Open-source models offer data privacy (no data leaves your infrastructure), no per-query costs, customization (full fine-tuning), and no vendor lock-in.

The choice depends on the use case. For prototyping and low-volume production, proprietary APIs are faster and cheaper. For high-volume production (millions of queries per day), self-hosted open-source models are cheaper. For sensitive data (healthcare, finance), self-hosted models may be required by regulation. For maximum quality, proprietary frontier models are still ahead.

The gap is narrowing. Open-source models like LLaMA 3 70B and Mistral Large approach GPT-4 quality on many benchmarks. For many production use cases, the quality difference is not significant enough to justify the cost and privacy trade-offs of proprietary APIs. The trend is toward open-source models becoming viable for production, while proprietary models maintain an edge for the most demanding applications.

NLP and AI are increasingly regulated. The EU AI Act, enacted in 2024, classifies AI systems by risk level and imposes requirements for high-risk systems. The US has sector-specific guidance (FTC on AI claims, FDA on medical AI, SEC on financial AI). China has AI regulations covering algorithmic recommendations and generative AI. Other countries are developing their own frameworks.

Key regulatory themes include transparency (users must know when they interact with AI), accountability (organizations must be responsible for AI outputs), fairness (AI must not discriminate), privacy (AI must protect personal

data), and safety (AI must not cause harm). Production NLP systems must be designed with these requirements in mind, with audit trails, bias testing, safety monitoring, and user disclosure.

The pace of NLP innovation is unprecedented. New models are released monthly. New techniques (better quantization, new attention mechanisms, improved training methods) are published weekly. New tools and frameworks are released continuously. Keeping up with the field requires significant effort, and the half-life of technical knowledge is short.

For production systems, the key is to build systems that are model-agnostic and technique-agnostic. Do not hard-code model names, prompt formats, or API endpoints. Build abstractions that allow models, prompts, and retrieval pipelines to be swapped without changing application code. This separation of concerns allows the system to benefit from new innovations without requiring rewrites.

Data labeling has become a significant industry in the NLP ecosystem. Human annotators create training data for fine-tuning, golden sets for evaluation, and preference data for RLHF. The quality of labeled data directly determines the quality of the NLP system—garbage in, garbage out.

Data labeling platforms (Scale AI, Labelbox, Snorkel) provide tools for managing annotation projects, with features for inter-annotator agreement, quality control, and active learning. The cost of labeling varies widely: simple classification tasks cost $0.05-0.50 per example, complex generation tasks cost $1-10 per example, and expert domain labeling (medical, legal) can cost $50+ per example.

For production NLP systems, data labeling is an ongoing investment. The golden set needs to be expanded over time, production errors need to be labeled for improvement, and new use cases require new labeled data. Organizations that invest in data labeling infrastructure see compounding returns—the labeled data improves the model, which improves the product, which generates more data.

The competitive landscape for NLP models and platforms is intense. OpenAI, Anthropic, Google, Meta, and Mistral compete on model quality, context length, and price. Cloud providers (AWS, Azure, GCP) compete on infrastructure, managed services, and enterprise features. Vector database providers (Pinecone, Qdrant, Weaviate) compete on performance, scalability, and developer experience. Framework providers (LangChain, LlamaIndex) compete on ease of use and ecosystem.

For practitioners, the competition is beneficial—prices are falling, quality is

improving, and capabilities are expanding. But it also means the landscape changes rapidly, and systems must be designed to adapt. Vendor lock-in is a risk: if you build your system around a specific model or platform, switching costs can be high. The mitigation is abstraction: build interfaces between your application and the model, database, and framework layers.

Serving LLMs in production requires specialized infrastructure. The inference stack includes the model server (vLLM, TGI, TensorRT-LLM), which handles batching, caching, and GPU management. The routing layer (Load balancer, API gateway) distributes requests across GPU instances. The caching layer (Redis, semantic cache)

stores recent responses to avoid redundant computation. The monitoring layer (Datadog, Prometheus, custom dashboards) tracks latency, throughput, and error rates.

Inference optimization is critical for cost management. Techniques include: quantization (reducing model precision from FP16 to INT8 or INT4, reducing memory and increasing throughput), batching (processing multiple requests simultaneously to maximize GPU utilization), KV cache (caching attention keys and values to avoid recomputing for repeated prefixes), and speculative decoding (using a smaller model to generate candidate tokens that the larger model verifies).

Tokens are the unit of processing for LLMs. A token is roughly 4 characters of English text, or about 0.75 words. Tokenization affects cost, latency, and quality. A prompt with 1000 tokens costs 10x more than a prompt with 100 tokens. A model with a 128K token context window can process 10x more text than a model with a 12K window.

Tokenization is language-dependent. English text tokenizes efficiently (about 0.75 tokens per word). Non-English text, especially languages with different scripts (Chinese, Arabic, Hindi), may tokenize less efficiently (2-3 tokens per word), increasing cost and reducing the effective context window. Code tokenizes differently from natural language, with tokenization optimized for common programming patterns.

The data used to pre-train LLMs is a growing concern. The largest models are trained on trillions of tokens of web text, which includes copyrighted content, low-quality content, biased content, and personal data. The legal and ethical implications are still being worked out.

Key concerns include: copyright (training on copyrighted text without permission), privacy (personal data in training data), quality (low-quality text degrades model quality), bias (training data reflects societal biases), and

contamination (test data appearing in training data, making benchmarks unreliable). These concerns are leading to more careful data curation, licensing agreements for training data, and the development of synthetic data as an alternative.

The field faces an evaluation crisis. Benchmark scores (MMLU, HumanEval, GSM8K) are used to compare models, but models are increasingly trained on benchmark data, making scores unreliable. The benchmarks may not reflect real-world performance—a model that scores well on a benchmark may perform poorly on production tasks. And the rapid pace of model development means benchmarks become outdated quickly.

The solution is custom evaluation: build a golden set that reflects your specific use case, evaluate models on that set, and use the results to make decisions. Generic benchmarks are useful for initial model selection but should never be the sole basis for production decisions. The golden set is the ground truth that matters.

End of free sample. The full book is available at putrabooks.github.io

ABOUT THE AUTHOR

Aiman JoeHanz writes about the technologies that are quietly reshaping the world. His Building of Tomorrow series examines the infrastructure behind artificial intelligence -- from blockchain and cloud computing to agentic systems and token optimization -- and asks how these technologies will evolve over the coming decade.

His books are built on technical research and engineering documentation, written for readers who want to understand systems rather than trends. He covers what is being built today and what it will take to build what comes next.

He believes that understanding infrastructure is the difference between watching the future happen and helping to create it.