# Deconstructing Gemini and Building a Code-Generation Language Model from First Principles

## Introduction

This report addresses the dual objectives of deconstructing a state-of-the-art large language model and providing a practical blueprint for creating a new one. We will begin with an in-depth analysis of Google's Gemini family, a series of models that have pushed the boundaries of multimodality, context length, and reasoning. We will explore the key architectural decisions, such as the adoption of a Mixture-of-Experts framework, that enable their scale and efficiency. Following this analysis, we will transition to a complete, hands-on guide for building a custom, GPT-style transformer model from scratch using PyTorch. This guide is tailored to the specific task of code generation, addressing the user's unique data challenge by providing a full data pipeline to scrape, parse, and prepare a programming dataset from web URLs. The report culminates in a discussion of best practices for evaluation, ensuring the developed model can be rigorously and meaningfully assessed.

## Section 1: Deconstructing the Gemini Architecture: A Technical Deep Dive

This section provides a granular analysis of the Gemini models, synthesizing details from available technical reports and research papers to build a coherent picture of their design philosophy and technical underpinnings.

### 1.1 The Architectural Blueprint: From Dense Transformers to Sparse Mixture-of-Experts (MoE)

The foundation of modern large language models (LLMs) is the Transformer architecture, first introduced in the seminal 2017 paper "Attention is All You Need".[1] This architecture, with its self-attention mechanism, revolutionized the processing of sequential data by dispensing with the recurrent connections of models like LSTMs, thereby enabling massive parallelization and scalability.[2] Early successful LLMs, including the initial GPT series, were based on a "dense" implementation of this architecture. In a dense model, every parameter is activated for every input token during a forward pass. While effective, this approach presents a significant scaling challenge: as the number of parameters increases to enhance model capability, the computational cost of both training and inference grows proportionally.

A key architectural evolution observed in the Gemini 1.5 Pro and Gemini 2.5 models is the strategic shift to a **Sparse Mixture-of-Experts (MoE) Transformer architecture**.[4] This design is a direct and sophisticated response to the scaling limitations inherent in dense models. Instead of a single, monolithic feed-forward network (FFN) layer within each transformer block, an MoE model incorporates a collection of smaller, parallel FFNs, referred to as "experts." For each token processed, a lightweight "gating network" or "router" dynamically learns to select a small subset of these experts (often just two) to handle the computation.[5] The outputs of the selected experts are then combined, typically through a weighted sum determined by the gating network's output.

The significance of this approach cannot be overstated. It fundamentally decouples the total number of model parameters from the computational cost per token. A model can thus be scaled to trillions of parameters, vastly increasing its capacity to store knowledge, without a corresponding explosion in the floating-point operations (FLOPs) required for training or inference.[5] This allows for the creation of models that are both immensely powerful and computationally efficient to serve.[4]

This architectural pivot reveals a crucial trend in the development of frontier AI models. The competitive landscape is no longer defined solely by the brute-force scaling of parameter counts. The immense financial and hardware costs associated with training dense, multi-trillion-parameter models—requiring vast clusters of high-performance GPUs or TPUs, massive power consumption, and sophisticated cooling infrastructure—have become a prohibitive bottleneck.[8] The logical progression from this economic and computational pressure is the adoption of more efficient architectures. The move to sparse MoE frameworks, as seen in Gemini and reportedly in models like GPT-4 [10], demonstrates that the leading edge of AI research is now characterized by a pursuit of capital efficiency. The "arms race" is not just about building the biggest model, but about building the most capable model for a given computational budget.

## 1.2 Natively Multimodal by Design

Another defining characteristic of the Gemini family is that the models are **natively multimodal**.[11] This represents a significant architectural departure from earlier approaches to multimodality. Preceding systems often employed a "stitching" or "late fusion" methodology, where separate, pre-trained models were used as encoders for different data types—for example, a Vision Transformer (ViT) for images and a text-based transformer for language. The outputs of these separate encoders would then be combined and fed into a final

processing layer. While functional, this approach can create representational bottlenecks and limit the depth of cross-modal understanding.
Gemini's native multimodality implies a unified architecture designed from the ground up to process interleaved sequences of diverse data types—text, images, audio, and video—within a single, coherent computational graph.[12] The technical reports indicate that the visual encoding is inspired by prior work on models like Flamingo, CoCa, and PaLI.[12] In this paradigm, a video is not treated as a distinct object but is decomposed into a sequence of frames, which are then interleaved with text or audio tokens within the model's vast context window. Similarly, audio is ingested not as transcribed text but directly as features from a Universal Speech Model (USM), which preserves crucial prosodic and paralinguistic information (such as tone and emotion) that is lost in a text-only representation.[12] The model can output responses that also interleave different modalities, such as text and images.[15]
This unified design enables a far more sophisticated and deeply integrated form of cross-modal reasoning. Because all data types are processed within the same representational space, the model can identify and reason about complex relationships that span modalities. A compelling example highlighted in the technical reports is the model's ability to interpret a student's handwritten physics problem, which involves understanding the visual diagram, parsing the handwritten mathematical equations, identifying the logical flaw in the student's reasoning steps, and generating a correctly typeset explanation.[13] This task, which requires the simultaneous processing of image, text, and symbolic logic, would be exceptionally challenging for a model with siloed modality encoders.
The shift towards native multimodality is more than an incremental improvement; it is a fundamental paradigm shift towards a more generalized form of artificial intelligence. Human cognition does not operate in isolated modalities; we seamlessly and continuously integrate sensory inputs from sight, sound, and language to form a holistic understanding of the world. The limitations of siloed AI models in performing complex, real-world tasks that require this integrated reasoning created a clear impetus for change. This led to the need for a unified representational space for all data types, which in turn drove the development of natively multimodal architectures like Gemini. The broader implication is that the future of AI lies in these holistic models, which can reason across any combination of data inputs, unlocking previously intractable applications in areas like scientific discovery, where insights may be hidden in the correlations between experimental images, sensor data logs, and research literature.

## 1.3 The Long Context Revolution: Reasoning Over Millions of Tokens

Perhaps the most striking capability of the Gemini 1.5 and 2.5 models is their exceptionally large context window. While previous state-of-the-art models were typically limited to context lengths of 32,768 (32k) or 128,000 (128k) tokens [6], Gemini 1.5 Pro expanded this by an order of magnitude, demonstrating the ability to process inputs of up to
**1 million tokens**, and has been tested in research settings up to **10 million tokens**.[4]

The technical reports attribute this breakthrough not to a single innovation but to "a host of improvements made across nearly the entire model stack," encompassing architecture, data, optimization, and systems-level engineering.[7] This suggests that achieving such a massive context window required a concerted effort to optimize every component of the model, likely including more memory-efficient attention mechanisms, novel positional encoding schemes that can generalize to extreme lengths, and a highly optimized training and inference infrastructure.

To rigorously validate this long-context capability, researchers employed synthetic **"needle-in-a-haystack" (NIAH) evaluations**.[6] In a typical NIAH test, a small, specific piece of information—the "needle"—is programmatically inserted at a random location within a vast corpus of distractor text, video, or audio—the "haystack." The model is then prompted to retrieve this specific piece of information. The results are remarkable: Gemini 1.5 Pro achieves near-perfect recall (over 99%) on these tasks across all modalities, even when the context window is extended to millions of tokens.[6] This demonstrates an unprecedented ability to maintain high-fidelity information retrieval over vast input sequences, without performance degradation.

The practical implications of this capability are transformative. An LLM with a multi-million token context window is no longer just a tool for processing short prompts or documents. It becomes a powerful analysis engine capable of ingesting and reasoning over entire codebases (e.g., 30,000+ lines of code), full-length novels (e.g., the entirety of "War and Peace"), or hours of video and audio recordings in a single pass.[7] This allows for new applications such as performing a comprehensive analysis of a large software repository, summarizing a full movie including visual and auditory cues, or learning to translate a low-resource language from a grammar manual and dictionary provided entirely within the context window.[6]

## 1.4 Advanced Reasoning and "Thinking" Mechanisms

The Gemini 2.5 family introduces an explicit mechanism for enhanced reasoning, marketed as **"Deep Think"** or simply "thinking".[11] This is not merely a branding exercise but refers to a specific post-training technique designed to improve performance on complex, multi-step problems.

The underlying mechanism involves training the model with Reinforcement Learning to utilize additional computational resources at inference time. When faced with a difficult prompt, instead of generating a response directly in a single forward pass, the "thinking" model can perform an internal search process, exploring multiple potential reasoning paths or hypotheses.[5] This can involve tens of thousands of internal forward passes, allowing the model to test different strategies, evaluate intermediate results, and ultimately converge on a more accurate and well-reasoned final answer.[14] This deliberate, step-by-step approach has proven particularly effective for challenging domains like competitive programming, advanced mathematics, and scientific discovery, where the optimal solution is not immediately obvious

and requires careful planning and consideration of trade-offs.[11]
A crucial feature of this system is its controllability. Developers are given fine-grained control over the "thinking budget," allowing them to manage the trade-off between response quality, latency, and computational cost.[11] For applications where speed is paramount, the thinking budget can be minimized or disabled. For high-stakes tasks that demand the highest possible accuracy, the budget can be increased, allowing the model to expend more effort to arrive at a superior solution. The model can also operate adaptively, assessing the complexity of a task and calibrating its own thinking budget when one is not explicitly set by the user.[11]

## 1.5 Training at Scale: Data, Hardware, and Process

The performance of any foundational model is inextricably linked to the quality and scale of its training regimen. The Gemini models are the product of a sophisticated, multi-stage process powered by a massive infrastructure.
**Training Data:** The pre-training corpus is a vast, multimodal, and multilingual dataset. It is sourced from a diverse collection of publicly available web documents, books, and a large corpus of source code in various programming languages.[4] Crucially, this textual data is augmented with extensive image, audio, and video data to support the model's native multimodal capabilities. A significant engineering effort is dedicated to data quality, involving rigorous filtering to remove low-quality or harmful content, extensive deduplication to improve training efficiency, and safety filtering to align with responsible AI principles.[5]
**Training Process:** The development of a Gemini model follows a well-established, multi-stage pipeline:

1. **Pre-training:** In this initial and most computationally intensive phase, the model learns general-purpose knowledge and patterns from the massive, largely unlabeled dataset. The objective is typically next-token prediction, where the model learns to predict the next token in a sequence given the preceding ones.[4]
2. **Instruction Fine-Tuning (IFT):** After pre-training, the model has a broad understanding of language and concepts but does not inherently know how to follow user instructions. In the IFT stage, the model is further trained on a smaller, high-quality, curated dataset of paired instructions and desired responses. This teaches the model to act as a helpful assistant.[4]
3. **Alignment with Human Preferences:** The final stage involves aligning the model's behavior with human values, focusing on principles of helpfulness, honesty, and harmlessness. This is typically achieved using Reinforcement Learning from Human Feedback (RLHF), where human labelers rank different model responses to the same prompt, and this preference data is used to train a reward model. The LLM is then fine-tuned using reinforcement learning to maximize the scores produced by this reward model, effectively steering its behavior towards outputs that humans prefer.[4]

**Hardware Infrastructure:** Training models of Gemini's scale requires a purpose-built supercomputing infrastructure. Google leverages its custom-designed **Tensor Processing**

**Units (TPUs)**, specifically large-scale pods of TPUv4 and, for the latest models, TPUv5p accelerators.[4] TPUs are Application-Specific Integrated Circuits (ASICs) optimized for the dense matrix multiplication and tensor operations that dominate transformer workloads. For large-scale distributed training, TPUs can offer significant performance-per-dollar and performance-per-watt advantages over general-purpose GPUs, making them a cornerstone of Google's ability to train frontier models efficiently.[19]

| Feature | Gemini 1.0 (Ultra/Pro) | Gemini 1.5 Pro | Gemini 2.5 Pro |
|---|---|---|---|
| **Core Architecture** | Dense Transformer | Sparse Mixture-of-Experts (MoE) Transformer | Sparse Mixture-of-Experts (MoE) Transformer |
| **Max Context Window** | 32,000 tokens | 1 Million (Production), 10 Million (Research) | 1 Million (Production), 2 Million+ (Coming Soon) |
| **Multimodality** | Integrated (Text, Image, Audio, Video) | Natively Multimodal, Long-Context Multimodality | Natively Multimodal, Enhanced Video Processing (up to 3 hrs) |
| **Key Capability** | Advanced General Reasoning | Long-Context Retrieval & In-Context Learning | Enhanced Reasoning via "Deep Think" Mechanism |

## Section 2: Building a GPT-like Model for Code Generation from Scratch

This section provides a complete, practical guide for building a custom GPT-style transformer model, with a specific focus on code generation. It addresses the entire lifecycle, from data acquisition using web scraping to the implementation of the model architecture and the training loop, all using Python and the PyTorch deep learning framework.

### 2.1 Foundational Components of a Transformer Model

Before constructing the model, it is essential to establish a clear understanding of its core architectural components. A GPT-style model is a decoder-only transformer, meaning it is designed for auto-regressive text generation. Its primary components work in concert to process a sequence of input tokens and predict the most likely subsequent token.

- **Tokenization and Embeddings:** An LLM does not operate on raw text. The first step is **tokenization**, the process of converting a string of characters into a sequence of integers.[21] For code, as with natural language, subword tokenization algorithms like

Byte-Pair Encoding (BPE) are highly effective. They break down text into frequently occurring subword units, allowing the model to handle a virtually unlimited vocabulary of identifiers and keywords while keeping the tokenizer's vocabulary size manageable.[22] Once tokenized, each integer ID is mapped to a high-dimensional vector through an **embedding layer**. This vector is a learned representation that captures the semantic meaning of the token.[24]

- **Positional Encoding:** The self-attention mechanism, by its nature, is permutation-invariant; it treats the input as an unordered set of tokens. To provide the model with information about the sequence order, we must inject **positional encodings**. The original transformer paper proposed using a set of fixed sinusoidal functions of different frequencies, which are added to the token embeddings to give each position in the sequence a unique signature.[1]
- **Multi-Head Self-Attention:** This is the central mechanism of the transformer. For each token in the sequence, the model learns three vectors: a **Query (Q)**, a **Key (K)**, and a **Value (V)**. The Query vector represents the current token's request for information. The Key vectors of all other tokens in the sequence represent their "advertised" content. The attention score is calculated by taking the dot product of the current token's Query with every other token's Key. This score, which represents the relevance of each token to the current one, is scaled and passed through a softmax function to create attention weights.[27] The final output for the current token is a weighted sum of all the Value vectors in the sequence, where the weights are the attention scores.
  **Multi-Head Attention** enhances this process by performing it multiple times in parallel with different, learned linear projections for Q, K, and V. This allows the model to jointly attend to information from different representational subspaces at different positions, capturing a richer set of dependencies.[27]
- **Feed-Forward Networks and Residual Connections:** Following the multi-head attention sub-layer, each transformer block contains a simple, position-wise **feed-forward network (FFN)**, which consists of two linear layers with a non-linear activation function in between.[2] To enable the training of very deep networks (i.e., stacking many transformer blocks), two crucial techniques are employed: **residual connections** and **layer normalization**. A residual connection adds the input of a sub-layer to its output (a form of skip connection), which helps mitigate the vanishing gradient problem. Layer normalization is applied before or after each sub-layer to stabilize the activations and accelerate training.

## 2.2 Data Acquisition and Preprocessing Pipeline: From URLs to a Trainable Dataset

This section provides a complete, production-ready Python pipeline to address the specific challenge of creating a code generation dataset from a list of URLs, where each URL contains a problem description and a corresponding code solution. The quality of this pipeline is paramount, as the performance of the final model is fundamentally constrained by the quality of its training data. A robust, repeatable, and clean data preparation process is non-negotiable for building a high-quality model.

### 2.2.1 Step 1: Web Scraping with requests and BeautifulSoup

To acquire the raw data, we must first fetch the HTML content from each target URL. For this task, the requests library is an excellent choice due to its simplicity, robustness, and widespread adoption for making HTTP requests in Python.[30] Once the HTML is fetched, it must be parsed. Real-world web pages often contain malformed or non-standard HTML. The BeautifulSoup library is specifically designed to handle this complexity, parsing messy markup into a navigable tree structure that is easy to query and manipulate.[30]

The initial part of our script will focus on reading a list of URLs from a file and systematically fetching the HTML content for each one, incorporating essential error handling and polite scraping practices.

### 2.2.2 Step 2: Parsing HTML to Extract Problem-Solution Pairs

This is the most critical and often most challenging step in the data pipeline. The goal is to reliably extract the problem description (natural language) and the code solution from the parsed HTML. Since the structure of web pages can vary widely, this requires identifying consistent patterns in the HTML markup. The standard workflow involves using a web browser's developer tools to "Inspect Element" on a sample page. By examining the HTML structure, one can identify the specific tags, classes, or IDs that uniquely contain the desired content.[30] For example, a problem description might be consistently located within a <div class="problem-description"> tag, while the code solution is typically found within <pre><code>...</code></pre> tags.

Our script will leverage BeautifulSoup's powerful CSS selector capabilities (via the .select() method) or its tag-finding methods (like .find_all()) to locate these specific elements and extract their textual content. The logic must be resilient, gracefully handling pages where the expected structure is not found.

### 2.2.3 Step 3: Structuring and Saving Data in JSONL Format

For large-scale machine learning tasks, storing data in a single, massive JSON file is inefficient and can lead to memory issues. A far superior format is **JSON Lines (.jsonl)**.[35] In a .jsonl file, each line is an independent, valid JSON object. This format allows for data to be streamed and processed line-by-line, which is highly memory-efficient and ideal for use with PyTorch's Dataset and DataLoader classes.

The final step of our pipeline will be to structure the extracted problem and solution strings into a Python dictionary (e.g., {"problem": "...", "solution": "..."}) and append this dictionary, serialized as a JSON string, as a new line to our output file.

## Complete Annotated Code for Data Pipeline

The following Python script provides a complete, end-to-end implementation of the data acquisition and preprocessing pipeline. It requires the requests and beautifulsoup4 libraries to be installed (pip install requests beautifulsoup4).

Python

```python
import requests
from bs4 import BeautifulSoup
import json
import time
import logging
from typing import List, Dict, Optional, Tuple

# Configure logging to provide progress and error information
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -
%(message)s')

def read_urls_from_file(file_path: str) -> List[str]:
    """
    Reads a list of URLs from a text file, with one URL per line.

    Args:
        file_path (str): The path to the text file containing URLs.

    Returns:
        List[str]: A list of URLs.
    """
    try:
        with open(file_path, 'r') as f:
            urls = [line.strip() for line in f if line.strip()]
        logging.info(f"Successfully read {len(urls)} URLs from {file_path}")
        return urls
    except FileNotFoundError:
        logging.error(f"URL file not found at {file_path}")
        return
```

```
def fetch_html_content(url: str, timeout: int = 10) -> Optional[str]:
    """
    Fetches the HTML content of a given URL.

    Args:
        url (str): The URL to scrape.
        timeout (int): The request timeout in seconds.

    Returns:
        Optional[str]: The HTML content as a string, or None if the request fails.
    """
    try:
        # Using a generic user-agent to mimic a browser
        headers = {
            'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/91.0.4472.124 Safari/537.36'
        }
        response = requests.get(url, headers=headers, timeout=timeout)
        response.raise_for_status()  # Raise an exception for bad status codes (4xx or 5xx)
        return response.text
    except requests.exceptions.RequestException as e:
        logging.error(f"Failed to fetch URL {url}: {e}")
        return None

def parse_problem_solution(
    html_content: str,
    problem_selector: str,
    solution_selector: str
) -> Optional]:
    """
    Parses HTML content to extract a problem description and a code solution
    using CSS selectors.

    Args:
        html_content (str): The raw HTML of the page.
        problem_selector (str): The CSS selector for the problem description element.
        solution_selector (str): The CSS selector for the code solution element.

    Returns:
        Optional]: A dictionary with 'problem' and 'solution' keys,
                          or None if either element is not found.
    """
    soup = BeautifulSoup(html_content, 'html.parser')

    problem_element = soup.select_one(problem_selector)
    solution_element = soup.select_one(solution_selector)

    if problem_element and solution_element:
        problem_text = problem_element.get_text(separator='\n', strip=True)
        solution_text = solution_element.get_text(strip=True)

        return {
            "problem": problem_text,
            "solution": solution_text
        }
    else:
        if not problem_element:
            logging.warning("Problem element not found with selector.")
        if not solution_element:
            logging.warning("Solution element not found with selector.")
        return None

def save_to_jsonl(data: Dict[str, str], output_file: str) -> None:
    """
    Appends a dictionary to a JSON Lines (.jsonl) file.

    Args:
        data (Dict[str, str]): The dictionary to save.
        output_file (str): The path to the output.jsonl file.
    """
    try:
        with open(output_file, 'a', encoding='utf-8') as f:
            f.write(json.dumps(data) + '\n')
    except IOError as e:
        logging.error(f"Failed to write to {output_file}: {e}")

def main_scraper(
    url_file: str,
    output_file: str,
    problem_selector: str,
    solution_selector: str,
    rate_limit_seconds: int = 1
) -> None:
    """
    Main function to orchestrate the web scraping and data saving process.
    Args:
        url_file (str): Path to the file with URLs.
        output_file (str): Path to the output.jsonl file.
        problem_selector (str): CSS selector for the problem description.
        solution_selector (str): CSS selector for the code solution.
        rate_limit_seconds (int): Delay between requests to be polite to servers.
    """
    urls = read_urls_from_file(url_file)
    if not urls:
        return

    logging.info(f"Starting scraping process. Output will be saved to {output_file}")

    success_count = 0
    for i, url in enumerate(urls):
        logging.info(f"Processing URL {i+1}/{len(urls)}: {url}")

        html = fetch_html_content(url)
        if html:
            parsed_data = parse_problem_solution(html, problem_selector, solution_selector)
            if parsed_data:
                save_to_jsonl(parsed_data, output_file)
                success_count += 1
                logging.info(f"Successfully extracted and saved data from {url}")
            else:
                logging.warning(f"Could not parse required data from {url}")

        # Polite rate limiting
        time.sleep(rate_limit_seconds)

    logging.info(f"Scraping complete. Successfully processed {success_count}/{len(urls)}
URLs.")


if __name__ == '__main__':
    # --- Configuration ---
    # This is where the user customizes the script for their target website.

    # 1. Create a file named 'urls.txt' and add the target URLs, one per line.
    URL_LIST_FILE = 'urls.txt'

    # 2. Specify the output file name.
```

```
OUTPUT_JSONL_FILE = 'coding_dataset.jsonl'

# 3. Inspect the target website's HTML and find the CSS selectors.
#    Example for a hypothetical competitive programming site:
#    - The problem description is in a div with class 'problem-statement'
#    - The code solution is in the first 'pre' tag inside a div with class 'solution-code'
PROBLEM_CSS_SELECTOR = 'div.problem-statement'
SOLUTION_CSS_SELECTOR = 'div.solution-code pre'

# 4. Set a delay between requests (in seconds).
#    Be respectful to the website's servers. 1-2 seconds is a good starting point.
RATE_LIMIT = 1

# --- Execution ---
main_scraper(
    url_file=URL_LIST_FILE,
    output_file=OUTPUT_JSONL_FILE,
    problem_selector=PROBLEM_CSS_SELECTOR,
    solution_selector=SOLUTION_CSS_SELECTOR,
    rate_limit_seconds=RATE_LIMIT
)
```

## 2.3 Implementing the Model from Scratch in PyTorch

To gain the deepest possible understanding of the model's inner workings, the implementation will be done from first principles using PyTorch, without relying on high-level abstractions from libraries like Hugging Face's transformers. This approach, championed by educational resources like Sebastian Raschka's "Build a Large Language Model (From Scratch)," ensures that every computational step is explicit and understood.[36]
The code will be structured into modular torch.nn.Module classes, each representing a distinct architectural component. This promotes code clarity, reusability, and makes the overall architecture easier to reason about.

Python

```
import torch
import torch.nn as nn
from torch.nn import functional as F
import math
```

```
class MultiHeadAttention(nn.Module):
    """
    Implementation of Multi-Head Self-Attention.
    """
    def __init__(self, n_heads, d_model, d_head, dropout):
        super().__init__()
        self.n_heads = n_heads
        self.d_model = d_model
        self.d_head = d_head

        self.qkv_proj = nn.Linear(d_model, 3 * n_heads * d_head)
        self.out_proj = nn.Linear(n_heads * d_head, d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        batch_size, seq_len, _ = x.shape

        # Project to Q, K, V and reshape
        qkv = self.qkv_proj(x)
        q, k, v = qkv.chunk(3, dim=-1)

        q = q.view(batch_size, seq_len, self.n_heads, self.d_head).transpose(1, 2)
        k = k.view(batch_size, seq_len, self.n_heads, self.d_head).transpose(1, 2)
        v = v.view(batch_size, seq_len, self.n_heads, self.d_head).transpose(1, 2)

        # Scaled dot-product attention
        scores = (q @ k.transpose(-2, -1)) / math.sqrt(self.d_head)

        if mask is not None:
            scores = scores.masked_fill(mask == 0, float('-inf'))

        attention_weights = F.softmax(scores, dim=-1)
        attention_weights = self.dropout(attention_weights)

        attention_output = attention_weights @ v

        # Reshape and project output
        attention_output = attention_output.transpose(1, 2).contiguous().view(batch_size,
seq_len, self.n_heads * self.d_head)
        output = self.out_proj(attention_output)

        return output
```

```
class FeedForward(nn.Module):
    """
    Position-wise Feed-Forward Network.
    """
    def __init__(self, d_model, d_ff, dropout):
        super().__init__()
        self.linear_1 = nn.Linear(d_model, d_ff)
        self.linear_2 = nn.Linear(d_ff, d_model)
        self.dropout = nn.Dropout(dropout)
        self.activation = nn.ReLU()

    def forward(self, x):
        x = self.linear_1(x)
        x = self.activation(x)
        x = self.dropout(x)
        x = self.linear_2(x)
        return x

class TransformerBlock(nn.Module):
    """
    A single Transformer block combining multi-head attention and a feed-forward network.
    """
    def __init__(self, n_heads, d_model, d_head, d_ff, dropout):
        super().__init__()
        self.attention = MultiHeadAttention(n_heads, d_model, d_head, dropout)
        self.ffn = FeedForward(d_model, d_ff, dropout)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Pre-LayerNorm variant
        x = x + self.dropout(self.attention(self.norm1(x), mask))
        x = x + self.dropout(self.ffn(self.norm2(x)))
        return x

class GPTModel(nn.Module):
    """
    The complete GPT-style decoder-only Transformer model.
    """
    def __init__(self, vocab_size, max_seq_len, n_layers, n_heads, d_model, d_ff, dropout):
        super().__init__()
        self.d_model = d_model
        self.token_embedding = nn.Embedding(vocab_size, d_model)
        self.positional_embedding = nn.Embedding(max_seq_len, d_model)

        d_head = d_model // n_heads
        self.transformer_blocks = nn.ModuleList()

        self.final_norm = nn.LayerNorm(d_model)
        self.lm_head = nn.Linear(d_model, vocab_size, bias=False)

        # Weight tying
        self.token_embedding.weight = self.lm_head.weight

    def forward(self, idx, targets=None):
        batch_size, seq_len = idx.shape

        # Get embeddings
        token_embeds = self.token_embedding(idx)

        positions = torch.arange(0, seq_len, dtype=torch.long, device=idx.device).unsqueeze(0)
        pos_embeds = self.positional_embedding(positions)

        x = token_embeds + pos_embeds

        # Causal mask for decoder self-attention
        mask = torch.tril(torch.ones(seq_len, seq_len, device=idx.device)).view(1, 1, seq_len,
seq_len)

        # Pass through transformer blocks
        for block in self.transformer_blocks:
            x = block(x, mask)

        x = self.final_norm(x)
        logits = self.lm_head(x)

        loss = None
        if targets is not None:
            # Reshape for cross-entropy loss calculation
            logits_flat = logits.view(-1, logits.size(-1))
            targets_flat = targets.view(-1)
            loss = F.cross_entropy(logits_flat, targets_flat, ignore_index=-1) # Assuming -1 is
padding
```

```
    return logits, loss
```

## 2.4 The End-to-End Training Loop

With the model architecture and data pipeline defined, the final piece is the training script to orchestrate the learning process. This involves several key steps: training a custom tokenizer, setting up a data loader, and implementing the core training loop.

1. **Custom Tokenizer Training:** While pre-trained tokenizers are convenient, training a tokenizer specifically on the target domain corpus (in this case, our scraped code data) is a critical best practice. It ensures that the tokenizer's vocabulary is optimized for the specific syntax, keywords, and common identifiers present in the data, leading to more efficient tokenization (fewer tokens per program) and potentially better model performance.[38] The
   tokenizers library from Hugging Face provides an efficient implementation of algorithms like BPE for this purpose.
2. **PyTorch Dataset and DataLoader:** To feed data to the model efficiently, we will create a custom PyTorch Dataset class. This class will be responsible for loading the .jsonl file, reading it line by line, applying the trained tokenizer to convert the problem-solution pairs into token IDs, and formatting them into tensors. The DataLoader will then wrap this Dataset, handling tasks like creating mini-batches, shuffling the data each epoch, and enabling multi-process data loading to prevent I/O bottlenecks.
3. **Training Loop:** The core training loop iterates over the data provided by the DataLoader. In each iteration, it performs the following steps:
   - Moves a batch of data to the appropriate device (GPU).
   - Performs a **forward pass** by feeding the input tokens to the model to get the output logits and the loss.
   - Clears any old gradients from the optimizer.
   - Performs a **backward pass** (loss.backward()) to compute the gradients of the loss with respect to the model's parameters.
   - Applies **gradient clipping**, a technique to prevent exploding gradients by capping the norm of the gradients, which is crucial for stable training of large transformers.
   - Updates the model's weights by taking a step with the optimizer (optimizer.step()).
   - Updates the learning rate according to a pre-defined schedule.
   - Logs metrics like training loss for monitoring.
4. **Utilities:** A production-grade training script also requires several utilities, including a learning rate scheduler (e.g., cosine annealing with warmup) to dynamically adjust the learning rate during training, and a checkpointing mechanism to periodically save the model's state. Checkpointing is essential for long training runs, as it allows training to be resumed from the last saved state in case of an interruption.

## 2.5 Hyperparameter Tuning and Hardware Considerations

The performance and stability of the training process are highly sensitive to the choice of hyperparameters. While optimal values depend on the specific model size and dataset, there are well-established starting points and principles.
**Hardware Considerations:** Training a large language model from scratch is a computationally intensive task that requires specialized hardware.

- **GPU:** The most critical component is the Graphics Processing Unit (GPU). Training requires GPUs with large amounts of high-bandwidth memory (VRAM) to store the model parameters, gradients, and optimizer states. For models with billions of parameters, enterprise-grade GPUs like the NVIDIA A100 (40-80GB VRAM) or H100 are the standard.[8] A consumer-grade GPU, even a high-end one like an RTX 4090 (24GB VRAM), would only be sufficient for training much smaller models (hundreds of millions of parameters).
- **Multi-GPU Training:** To train billion-parameter models, a multi-GPU setup is necessary. This requires high-speed interconnects between the GPUs, such as NVIDIA's NVLink, to minimize communication bottlenecks during distributed training.[8]
- **CPU and RAM:** A powerful multi-core CPU and a large amount of system RAM (e.g., 128 GB or more) are needed to handle data preprocessing and loading without bottlenecking the GPUs.[41]
- **Storage:** Fast storage, such as NVMe SSDs, is required to load the large dataset and save model checkpoints efficiently.[8]

| Hyperparameter | Description | Typical Range / Starting Value | Impact |
|---|---|---|---|
| **learning_rate** | The step size for optimizer updates. | 1e−5 to 6e−4 (with warmup and decay) | Too high leads to instability/divergence; too low leads to slow convergence. Crucial for successful training.[42] |
| **batch_size** | Number of sequences processed in one forward/backward pass. | 16-1024 (constrained by GPU memory) | Larger batches provide more stable gradient estimates but require more memory. Affects training speed.[42] |
| **n_layers** | The number of stacked Transformer blocks (depth of the model). | 12-96 | Increases model capacity and ability to learn complex patterns, but also increases |

| | | | |
|---|---|---|---|
| | | | computational cost and risk of overfitting.[43] |
| **n_heads** | The number of parallel attention heads. | 8-32 | More heads allow the model to attend to different representation subspaces. Must be a divisor of d_model.[44] |
| **d_model** | The dimensionality of the token embeddings and hidden states. | 768-12288 | The "width" of the model. Larger values increase model capacity but also memory usage and computation.[2] |
| **dropout_rate** | The probability of randomly zeroing out activations. | 0.0 - 0.2 | A regularization technique to prevent overfitting. A rate of 0.1 is a common starting point.[42] |
| **weight_decay** | L2 regularization term added to the loss to penalize large weights. | 0.01 - 0.1 | Helps prevent overfitting by discouraging complex models.[45] |

## Section 3: Evaluation, Refinement, and Future Directions

Developing a model is only half the battle; rigorously evaluating its performance is essential to measure progress, identify weaknesses, and guide further improvements. For code generation, the choice of evaluation metric is particularly critical.

### 3.1 Evaluating Functional Correctness: The pass@k Metric

For many natural language tasks, metrics like BLEU (Bilingual Evaluation Understudy) are used to measure the similarity between the generated text and a reference text based on n-gram overlap.[46] However, for code generation, such metrics are fundamentally flawed and can be highly misleading.[49] A generated code snippet can be perfectly correct from a functional standpoint—compiling without errors and producing the correct output—while being textually very different from a reference solution. It might use different variable names, a different algorithm, or a different control flow structure. In such a case, it would receive a very low BLEU score despite being a perfect solution.

This fundamental disconnect between textual similarity and functional correctness led to the development and adoption of **pass@k** as the industry-standard metric for evaluating code generation models.[50] The

pass@k metric measures the probability that at least one of k code samples generated for a given problem will pass a set of predefined unit tests. It directly assesses what matters: does the code work correctly?

The evaluation process is as follows:

1. For a single programming problem from a held-out test set, the model is prompted to generate k independent solutions (e.g., by using temperature-based sampling).
2. Each of the k generated solutions is executed against a suite of unit tests associated with the problem.
3. If at least one of the k solutions passes all the unit tests, the problem is considered "solved."
4. The pass@k score is the fraction of all problems in the test set that were solved.

The choice of evaluation metric actively shapes the entire development process. A research team optimizing for a BLEU-like score will inadvertently prioritize models that learn to mimic the syntactic style of the reference solutions in their training data. In contrast, a team optimizing for pass@k will be driven to develop models that have a deeper, more robust understanding of programming logic and problem-solving, as the only thing that matters is generating functionally correct code. This shift from measuring stylistic similarity to measuring functional correctness was a critical step in the maturation of the code generation field, leading to the development of far more practically useful models. Adopting pass@k is therefore not just a best practice; it is a prerequisite for correctly measuring and driving meaningful progress on the task of code generation.

| Metric | What It Measures | Pros | Cons | Best For |
|---|---|---|---|---|
| **pass@k** | Functional correctness via unit tests. | Directly measures if the code works; robust to stylistic variations; reflects real-world developer workflow.[50] | Requires a test suite for each problem; can be computationally expensive to run tests for many samples.[52] | The primary, gold-standard evaluation for code generation tasks. |
| **BLEU/CodeBLEU** | N-gram overlap with a reference solution. | Fast to compute; does not require code execution. | Poor correlation with functional correctness; penalizes valid alternative solutions; | Quick, low-fidelity checks or tasks where syntactic similarity to a template is desired (e.g., code |

| | | | | |
|---|---|---|---|---|
| | | | sensitive to variable names and comments.[49] | translation). |
| **Static Analysis** | Code quality attributes (e.g., complexity, style). | Provides insights into code maintainability, readability, and potential bugs without execution.[53] | Says nothing about whether the code is functionally correct or solves the intended problem. | Assessing non-functional aspects of generated code, such as adherence to coding standards. |

### 3.2 Next Steps: Fine-Tuning, Scaling, and Alignment

Once a base model has been pre-trained, several avenues exist for further improvement and specialization.

- **Instruction Fine-Tuning:** The pre-trained model is a powerful next-token predictor, but it is not an "assistant." The next logical step is to perform instruction fine-tuning. This involves curating a dataset of instructions (e.g., "Write a Python function to compute the Fibonacci sequence") and corresponding high-quality code solutions. By fine-tuning the model on this dataset, it learns to follow commands and respond helpfully to user prompts, transforming it from a raw language model into a useful tool.[36]
- **Scaling Laws:** Research in LLMs has revealed the existence of "scaling laws"—predictable, power-law relationships between model performance, model size, and the amount of training data. This implies that one of the most reliable ways to improve the model's capabilities is to scale it up: increase the number of layers and attention heads, and train it on a larger, more diverse dataset.
- **Advanced Architectures:** The field of transformer architectures is constantly evolving. For code generation, future research could explore modifications specifically tailored to the structure of programming languages. This could include developing specialized tokenizers that are more aware of code syntax (e.g., treating indentation or brackets as special tokens), or designing attention mechanisms that can better capture the long-range dependencies and hierarchical structures inherent in software, such as function calls and class inheritance.[39]

# Conclusion

This report has provided a comprehensive journey from the architectural heights of Google's Gemini to the practical foundations of building a custom code-generation model. The analysis of Gemini reveals that the frontier of artificial intelligence is being advanced through a multi-faceted strategy focused on computational efficiency via sparse architectures like

Mixture-of-Experts, deeper reasoning through natively multimodal designs, and enhanced analytical power with massive context windows and explicit "thinking" mechanisms. These innovations are not isolated improvements but are part of a cohesive effort to build more general and capable AI systems.
Simultaneously, the practical guide has demonstrated that the core principles behind these frontier models can be applied to build powerful, domain-specific systems from scratch. With a principled approach to data acquisition, a clear, first-principles understanding of the transformer architecture, and a commitment to rigorous, function-first evaluation using metrics like pass@k, it is entirely feasible to engineer a capable AI coding assistant. The provided methodologies and complete code implementations serve as a robust toolkit for any researcher or engineer looking to transform a specialized collection of web resources into a functional and valuable large language model. The journey from deconstruction to construction underscores a key truth in the field: while frontier models define the horizon of what is possible, a deep understanding of the fundamentals empowers us to build the specialized tools of the future.

## Works cited

1. What is a Transformer Model? - IBM, accessed August 18, 2025, https://www.ibm.com/think/topics/transformer-model
2. Transformer (deep learning architecture) - Wikipedia, accessed August 18, 2025, https://en.wikipedia.org/wiki/Transformer_(deep_learning_architecture)
3. What is an attention mechanism? | IBM, accessed August 18, 2025, https://www.ibm.com/think/topics/attention-mechanism
4. Gemini 1.5 Technical Report: Key Reveals and Insights - Gradient Flow, accessed August 18, 2025, https://gradientflow.com/gemini-1-5-technical-report/
5. Gemini 2.5 Deep Think - Model Card - Googleapis.com, accessed August 18, 2025, https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-2-5-Deep-Think-Model-Card.pdf
6. Gemini 1.5: Unlocking multimodal understanding across ... - arXiv, accessed August 18, 2025, https://arxiv.org/pdf/2403.05530
7. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context - Kapler o AI, accessed August 18, 2025, https://www.kapler.cz/wp-content/uploads/gemini_v1_5_report.pdf
8. What hardware is required to train an LLM? - Milvus, accessed August 18, 2025, https://milvus.io/ai-quick-reference/what-hardware-is-required-to-train-an-llm
9. Hardware Guide for Large Language Models and Deep Learning | by Youssef Fenjiro, accessed August 18, 2025, https://medium.com/@fenjiro/hardware-guide-for-large-language-models-and-deep-learning-b619af574cca
10. OpenAI GPT-4: Architecture, Interfaces, Pricing, Alternative - Acorn Labs, accessed August 18, 2025, https://www.acorn.io/resources/learning-center/openai/

11. Gemini 2.5 Pro - Google DeepMind, accessed August 18, 2025, https://deepmind.google/models/gemini/pro/
12. Brief Review — Gemini: A Family of Highly Capable Multimodal Models | by Sik-Ho Tsang, accessed August 18, 2025, https://sh-tsang.medium.com/review-gemini-a-family-of-highly-capable-multimodal-models-615c2a100592
13. Gemini: A Family of Highly Capable Multimodal Models - arXiv, accessed August 18, 2025, https://arxiv.org/pdf/2312.11805
14. Gemini - Google DeepMind, accessed August 18, 2025, https://deepmind.google/models/gemini/
15. Gemini: A Family of Highly Capable Multimodal Models - Googleapis.com, accessed August 18, 2025, https://storage.googleapis.com/deepmind-media/gemini/gemini_1_report.pdf
16. Gemini 1.5 Pro vs GPT-4 Turbo Benchmarks - Bito AI, accessed August 18, 2025, https://bito.ai/blog/gemini-1-5-pro-vs-gpt-4-turbo-benchmarks/
17. Gemini 1.5: Google's Generative AI Model with Mixture of Experts Architecture - Encord, accessed August 18, 2025, https://encord.com/blog/google-gemini-1-5-generative-ai-model-with-mixture-of-experts/
18. Gemini 2.5: Our most intelligent AI model - Google Blog, accessed August 18, 2025, https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/
19. GPU and TPU Comparative Analysis Report | by ByteBridge - Medium, accessed August 18, 2025, https://bytebridge.medium.com/gpu-and-tpu-comparative-analysis-report-a5268e4f0d2a
20. bytebridge.medium.com, accessed August 18, 2025, https://bytebridge.medium.com/gpu-and-tpu-comparative-analysis-report-a5268e4f0d2a#:~:text=Cost%20differences%20significantly%20influence%20AI,compared%20to%20NVIDIA%20A100%20GPUs.
21. Introduction to LLM Tokenization - Airbyte, accessed August 18, 2025, https://airbyte.com/data-engineering-resources/llm-tokenization
22. Understanding tokens - .NET | Microsoft Learn, accessed August 18, 2025, https://learn.microsoft.com/en-us/dotnet/ai/conceptual/understanding-tokens
23. The Technical User's Introduction to LLM Tokenization - Christopher Samiullah, accessed August 18, 2025, https://christophergs.com/blog/understanding-llm-tokenization
24. What are Transformers? - Transformers in Artificial Intelligence Explained - AWS, accessed August 18, 2025, https://aws.amazon.com/what-is/transformers-in-artificial-intelligence/
25. 5 Types of Word Embeddings and Example NLP Applications - Swimm, accessed August 18, 2025, https://swimm.io/learn/large-language-models/5-types-of-word-embeddings-and-example-nlp-applications

26. Word embedding - Wikipedia, accessed August 18, 2025, https://en.wikipedia.org/wiki/Word_embedding
27. Transformer Attention Mechanism in NLP - GeeksforGeeks, accessed August 18, 2025, https://www.geeksforgeeks.org/nlp/transformer-attention-mechanism-in-nlp/
28. The Transformer Attention Mechanism - MachineLearningMastery.com, accessed August 18, 2025, https://machinelearningmastery.com/the-transformer-attention-mechanism/
29. Introduction to Transformers and Attention Mechanisms | by Rakshit Kalra - Medium, accessed August 18, 2025, https://medium.com/@kalra.rakshit/introduction-to-transformers-and-attention-mechanisms-c29d252ea2c5
30. BeautifulSoup Web Scraping: Step-By-Step Tutorial - Bright Data, accessed August 18, 2025, https://brightdata.com/blog/how-tos/beautiful-soup-web-scraping
31. Implementing Web Scraping in Python with BeautifulSoup - GeeksforGeeks, accessed August 18, 2025, https://www.geeksforgeeks.org/python/implementing-web-scraping-python-beautiful-soup/
32. Top 7 Python Web Scraping Libraries - Bright Data, accessed August 18, 2025, https://brightdata.com/blog/web-data/python-web-scraping-libraries
33. research.aimultiple.com, accessed August 18, 2025, https://research.aimultiple.com/python-web-scraping-libraries/#:~:text=Beautiful Soup%20is%20a%20Python%20library,pieces%20of%20information%20incredibly%20simple.
34. Best Python Web Scraping Libraries: Selenium vs Beautiful Soup - Research AIMultiple, accessed August 18, 2025, https://research.aimultiple.com/python-web-scraping-libraries/
35. Scrapy Database Guide - Saving Data To JSON Files - ScrapeOps, accessed August 18, 2025, https://scrapeops.io/python-scrapy-playbook/scrapy-save-json-files/
36. rasbt/LLMs-from-scratch: Implement a ChatGPT-like LLM in ... - GitHub, accessed August 18, 2025, https://github.com/rasbt/LLMs-from-scratch
37. Build a Large Language Model (From Scratch) - Sebastian Raschka - Manning Publications, accessed August 18, 2025, https://www.manning.com/books/build-a-large-language-model-from-scratch
38. Tokenization in Large Language Models (LLMs) | by Shashank Agarwal | Medium, accessed August 18, 2025, https://medium.com/@shashankag14/tokenization-in-large-language-models-llms-0ba0aea6b1d6
39. Research on Compressed Input Sequences Based on Compiler Tokenization - MDPI, accessed August 18, 2025, https://www.mdpi.com/2078-2489/16/2/73
40. MegatronLM: Training Billion+ Parameter Language Models Using GPU Model Parallelism, accessed August 18, 2025, https://nv-adlr.github.io/MegatronLM
41. Recommended Hardware for Running LLMs Locally - GeeksforGeeks, accessed

August 18, 2025, https://www.geeksforgeeks.org/deep-learning/recommended-hardware-for-running-llms-locally/

42. Mastering LLM Hyperparameter Tuning for Optimal Performance - DEV Community, accessed August 18, 2025, https://dev.to/ankush_mahore/mastering-llm-hyperparameter-tuning-for-optimal-performance-1gc1
43. A Guide to LLM Hyperparameters | Symbl.ai, accessed August 18, 2025, https://symbl.ai/developers/blog/a-guide-to-llm-hyperparameters/
44. Large Language Model Training in 2025 - Research AIMultiple, accessed August 18, 2025, https://research.aimultiple.com/large-language-model-training/
45. Training Large Language Models (LLMs): Techniques and Best Practices - Nitor Infotech, accessed August 18, 2025, https://www.nitorinfotech.com/blog/training-large-language-models-llms-techniques-and-best-practices/
46. AI scorers: Evaluating AI-generated text with BLEU - Wandb, accessed August 18, 2025, https://wandb.ai/byyoung3/Generative-AI/reports/AI-scorers-Evaluating-AI-generated-text-with-BLEU--VmlldzoxMDc3MDE3OA
47. Demystifying the BLEU Metric: A Comprehensive Guide to Machine Translation Evaluation, accessed August 18, 2025, https://www.traceloop.com/blog/demystifying-the-bleu-metric
48. NLP - BLEU Score for Evaluating Neural Machine Translation - Python - GeeksforGeeks, accessed August 18, 2025, https://www.geeksforgeeks.org/nlp/nlp-bleu-score-for-evaluating-neural-machine-translation-python/
49. Out of the BLEU: how should we evaluate code generation models? - YouTube, accessed August 18, 2025, https://www.youtube.com/watch?v=KabVGPGSAmQ
50. Pass@k: A Practical Metric for Evaluating AI-Generated Code | by Ipshita - Medium, accessed August 18, 2025, https://medium.com/@ipshita/pass-k-a-practical-metric-for-evaluating-ai-generated-code-18462308afbd
51. Why does the pass@k metric not "behave like" probability? - AI Stack Exchange, accessed August 18, 2025, https://ai.stackexchange.com/questions/39460/why-does-the-passk-metric-not-behave-like-probability
52. Top Pass: Improve Code Generation by Pass@k-Maximized Code Ranking - arXiv, accessed August 18, 2025, https://arxiv.org/html/2408.05715v1
53. Measuring the Performance of AI Code Generation: A Practical Guide - Walturn, accessed August 18, 2025, https://www.walturn.com/insights/measuring-the-performance-of-ai-code-generation-a-practical-guide
54. mlabonne/llm-datasets: Curated list of datasets and tools for post-training. - GitHub, accessed August 18, 2025, https://github.com/mlabonne/llm-datasets
55. How to Rewrite a Transformer Model: Step-by-Step Customization of LLM

Architectures, accessed August 18, 2025, https://medium.com/@oluwamusiwaolamide/how-to-rewrite-a-transformer-model-step-by-step-customization-of-llm-architectures-82ef5cfa1caf
56. A Comparative Study on Code Generation with Transformers - arXiv, accessed August 18, 2025, https://arxiv.org/html/2412.05749v1