Case Study: Autoregressive Contract Clause Generation for Legal Document Drafting

From Blank Page to First Draft -- Training a GPT-Style Model to Generate Legal Contract Clauses

---

Section 1: Industry Context and Business Problem

Industry: Legal Technology -- Contract Lifecycle Management

The global legal services market generates over 900 billion dollars annually, and a substantial fraction of that cost is tied to drafting, reviewing, and negotiating contracts. Large law firms typically maintain thousands of template clauses across practice areas -- mergers and acquisitions, commercial leases, employment agreements, software licensing, and more. Associates spend 30 to 60 percent of their billable hours on what is fundamentally pattern-matching work: finding a relevant precedent clause, adapting it to the current transaction, and ensuring consistency with the rest of the document.

The core inefficiency is that contract language is highly structured and repetitive, yet the drafting process remains largely manual. A typical commercial software license agreement contains 40 to 80 clauses, of which 70 to 85 percent are drawn from standard templates with minor modifications. The remaining 15 to 30 percent require negotiation-specific customization -- indemnification caps, liability limitations, governing law selections, and termination triggers that vary by deal.

Company Profile: Lexis Draft AI

Lexis Draft AI is a Series A legal technology company headquartered in New York, NY, founded in 2022 by a team of three former BigLaw associates and two NLP researchers from Columbia University. The company has raised 28 million dollars across two rounds (Seed: 8M, Series A: 20M) and employs approximately 65 people -- 25 in engineering, 12 in legal product, and the rest in sales, customer success, and operations.

Lexis Draft's core product is a contract drafting assistant deployed in 18 mid-size law firms (50 to 300 attorneys each) across the United States. The platform integrates with document management systems (iManage, NetDocuments) and word processors (Microsoft Word via add-in, Google Docs). Attorneys select a contract type and a set of deal parameters, and the system generates a first-draft contract with appropriate clauses.

The current system uses a **retrieval-based approach**: it searches a clause library of 45,000 manually curated clauses using TF-IDF similarity, retrieves the top-5 matching clauses for each section, and presents them to the attorney for selection and editing. There is no generative capability -- the system can only retrieve existing clauses, not create new ones.

Business Challenge

Lexis Draft's retrieval-based system achieves a **clause relevance score of 62 percent** (measured as the percentage of retrieved clauses that attorneys accept with minor edits rather than rewriting from scratch) and a **first-draft acceptance rate of 38 percent** (percentage of generated contracts where the attorney makes fewer than 20 edits to the clause language).

The core failure modes are:

1. **Template rigidity.** The clause library was curated from a fixed set of precedent documents. When attorneys need clauses for novel deal structures -- such as a hybrid SaaS-plus-services agreement with milestone-based pricing -- the retrieval system returns only loosely related clauses that require extensive rewriting. Approximately 35 percent of all queries receive no clause with a relevance score above 0.5.

2. **Inconsistency across sections.** The system retrieves each clause independently. This means the indemnification clause might use "Licensee" while the limitation of liability clause uses "Customer" for the same party. Defined terms, cross-references, and stylistic conventions are frequently inconsistent across the generated document.

3. **Inability to adapt to firm style.** Each law firm has its own drafting conventions -- preferred sentence structures, capitalization rules for defined terms, and even specific phrases that partners insist on. The retrieval system cannot adapt to these firm-specific preferences because the clause library is shared across all clients.

4. **Missing context-dependent language.** Clauses that depend on previously specified terms -- for example, an indemnification clause that should reference the specific representations and warranties listed earlier in the agreement -- are retrieved without that context. The attorney must manually thread these references through the document.

Why It Matters

The financial and operational impact is substantial:

- Lexis Draft's 18 law firm clients collectively draft approximately **24,000 contracts per year** across commercial, corporate, and technology practice groups.
- At the current 38 percent first-draft acceptance rate, attorneys spend an average of **3.2 hours per contract** editing the generated draft, compared to a target of 1.5 hours. This represents approximately **40,800 hours of additional attorney time per year** across the client base.

- At an average associate billing rate of 450 dollars per hour, this excess editing time represents approximately **18.4 million dollars per year** in billable hours that could be redirected to higher-value work.
- Lexis Draft's annual contract renewal rate has dropped to 72 percent -- below the 80 percent threshold required by Series A investors. Four firms have cited "insufficient quality improvement over manual drafting" as the primary reason for non-renewal.
- Competing products from CaseText (now acquired by Thomson Reuters) and Harvey AI have begun offering generative clause drafting, and three of Lexis Draft's existing clients have initiated evaluation of these alternatives.

The VP of Engineering has been tasked with building a **generative clause drafting engine** that can produce novel, contextually appropriate contract clauses conditioned on deal parameters, previously drafted sections, and firm-specific style. The target: **increase the clause relevance score from 62 percent to 85 percent and the first-draft acceptance rate from 38 percent to 65 percent** within 6 months.

Constraints

The engineering team must work within the following constraints:

- **Compute budget:** 20,000 dollars per month for cloud GPU instances (AWS). Training must be feasible on 8x A100 80GB GPUs within 72 hours.
- **Latency:** A complete clause (typically 100 to 300 tokens) must be generated within **5 seconds**. Attorneys will not tolerate longer waits in an interactive drafting workflow.
- **Confidentiality:** All client contract data is governed by attorney-client privilege and firm confidentiality agreements. No client data may leave the firm's designated cloud environment. The model must be deployable within each firm's AWS VPC or Azure tenant. No data may be sent to external APIs (this rules out hosted LLM APIs for production use).
- **Data availability:** Lexis Draft has access to approximately **320,000 contract clauses** drawn from its 18 client firms (with appropriate data use agreements), plus 85,000 publicly available contract clauses from SEC EDGAR filings. The clauses span 12 contract types and 8 practice areas.
- **Model size:** The production model must fit within **24 GB of GPU memory** at inference time (firms use g5.xlarge instances with A10G GPUs). This constrains the model to approximately 350 million parameters in FP16.
- **Team expertise:** The ML team consists of 8 engineers. Three have NLP experience (two from the founding team), three are strong generalists with PyTorch experience, and two are recent hires with transformer fine-tuning backgrounds. The team has no prior experience training language models from scratch.

Section 2: Technical Problem Formulation

Problem Type: Conditional Autoregressive Text Generation

The core task is **conditional language generation** -- given a structured prompt containing the contract type, deal parameters, previously drafted sections, and firm-specific style tokens, generate the next clause token by token using a decoder-only transformer architecture.

Why autoregressive generation and not retrieval? The fundamental limitation of retrieval is that it can only return clauses that already exist in the library. For novel deal structures -- which account for 35 percent of queries -- no sufficiently relevant clause exists. An autoregressive model can compose novel language by combining patterns learned from hundreds of thousands of training examples. It generates one token at a time, conditioned on everything that came before, allowing it to produce coherent clauses that have never appeared verbatim in the training data.

We also considered a sequence-to-sequence (encoder-decoder) architecture where the encoder reads the deal parameters and the decoder generates the clause. However, the decoder-only approach is more natural for our use case because: (1) the "input" is not a fixed-length structured specification but rather a variable-length sequence of previously drafted contract sections that grows as the attorney works through the document, and (2) decoder-only models handle this growing context naturally through causal attention over the entire sequence so far.

Input Specification

Input: A structured prompt concatenated with the contract context, represented as a sequence of tokens.

- **Prompt structure:** `[CONTRACT_TYPE] Commercial Software License [GOVERNING_LAW] Delaware [FIRM_STYLE] formal_conservative [SECTION] Limitation of Liability [CONTEXT] ... previously drafted clauses ... [GENERATE]`
- **Tokenization:** We use a custom BPE tokenizer trained on legal text. The vocabulary includes 32,000 base tokens plus 200 special tokens for contract types, section headers, firm style markers, and control tokens. Legal terminology receives dedicated tokens to avoid subword fragmentation -- "indemnification" is a single token rather than "in-dem-ni-fi-cation."
- **Maximum sequence length:** 2,048 tokens. This accommodates the prompt structure (approximately 50 tokens), up to 1,500 tokens of previously drafted context, and 500 tokens for the generated clause. Contracts longer than the context window are handled by a sliding window with summary compression of earlier sections.
- **Input tensors:** `input_ids` (token indices in the BPE vocabulary), `attention_mask` (1 for real tokens, 0 for padding), `position_ids` (0 to T-1 for standard positional encoding).

Output Specification

Output: A sequence of token IDs generated autoregressively until a `[END_CLAUSE]` token is produced or the maximum clause length (500 tokens) is reached.

The generated token sequence is decoded back to text using the BPE tokenizer, producing a contract clause in natural legal language. The output is expected to:

1. Reference defined terms consistently with the rest of the document.
2. Follow the syntactic patterns of the specified firm style.
3. Contain appropriate legal substance for the specified contract type and section.
4. Include cross-references to previously drafted sections where appropriate.

Mathematical Foundation

Autoregressive Factorization. The probability of a generated clause $c = (c_1, c_2, \ldots, c_T)$ conditioned on the prompt p is factored autoregressively:

$$P(c \mid p) = \prod_{t=1}^{T} P(c_t \mid c_1, c_2, \ldots, c_{t-1}, p)$$

Each factor $P(c_t \mid c_{<t}, p)$ is computed by the GPT model's forward pass: the input sequence $(p, c_1, \ldots, c_{t-1})$ is processed through the transformer stack, and the output at the final position is projected to a distribution over the vocabulary via a linear layer and softmax.

Causal Self-Attention. The decoder uses causal (masked) self-attention to ensure that each token position can only attend to positions at or before it:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

where M is the causal mask with $M_{ij} = 0$ if $i \geq j$ and $M_{ij} = -\infty$ if $i < j$. This ensures that generating token t depends only on tokens 1 through $t-1$, preserving the autoregressive property.

Training Loss. The model is trained to minimize the cross-entropy loss over the next-token prediction task:

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log P_\theta(t_i \mid t_1, t_2, \ldots, t_{i-1})$$

where N is the total number of tokens in the training batch, θ represents the model parameters, and the sum runs over all token positions (both prompt and clause tokens during training, though at inference time the prompt tokens are given and only clause tokens are generated).

During training, we apply the loss only to the clause tokens (not the prompt tokens). This is implemented by setting the label to -100 for prompt positions, which PyTorch's `cross_entropy`

function ignores. This focuses the model's learning on clause generation rather than prompt reconstruction.

Loss Function Justification

1. **Cross-entropy over alternatives.** We use cross-entropy rather than mean squared error because the output is a categorical distribution over vocabulary tokens. MSE would impose an artificial ordering on token IDs (treating token 500 as "closer" to token 501 than to token 100), which has no linguistic meaning.

2. **Token-level averaging.** We average the loss over the number of clause tokens to make the loss invariant to clause length. Without averaging, longer clauses would dominate the gradient, biasing the model toward generating shorter outputs.

3. **Label smoothing (epsilon = 0.1).** We apply label smoothing to prevent the model from becoming overconfident. Instead of training toward a one-hot target, we train toward a distribution that places $1 - \epsilon$ probability on the correct token and distributes ϵ uniformly across all other tokens. This improves generation quality by preventing the model from memorizing exact training sequences.

Evaluation Metrics

Primary Metric: Clause Relevance Score

A panel of 5 senior attorneys scores each generated clause on a 1-to-5 scale across three dimensions: legal accuracy, stylistic appropriateness, and contextual consistency. The clause relevance score is the percentage of generated clauses scoring 4 or above on all three dimensions.

Target: Clause relevance score >= 85 percent (up from 62 percent with the retrieval system).

Secondary Metrics:

Metric	Target	Rationale
First-draft acceptance rate	>= 65%	Attorneys make fewer than 20 edits to clause language
Perplexity (held-out clause set)	< 12.0	Lower perplexity indicates better language modeling
Defined term consistency	>= 95%	Generated clauses use the same defined terms as the contract context
Generation latency (p95)	< 5 seconds	Interactive drafting workflow requirement
Model size (FP16)	< 24 GB	Must fit on A10G GPU

Baseline

Retrieval Baseline (Current System):

The existing system uses TF-IDF similarity over the 45,000-clause library with the following pipeline:

1. **Query construction:** Concatenate the contract type, section header, and key deal parameters into a query string.
2. **TF-IDF retrieval:** Compute cosine similarity between the query TF-IDF vector and all clause TF-IDF vectors. Return the top 5 matches.
3. **Re-ranking:** A lightweight BERT cross-encoder re-ranks the top 5 clauses based on semantic similarity to the full contract context.

This system achieves: - Clause relevance score: **62%** - First-draft acceptance rate: **38%** - Defined term consistency: **45%** (because each clause is retrieved independently) - Generation latency: **~800ms** (fast retrieval, but no generation)

The retrieval system is insufficient because: - It cannot generate novel clause language for unprecedented deal structures. - It retrieves each clause independently, causing inconsistency across the document. - It cannot adapt to firm-specific drafting conventions. - The clause library requires continuous manual curation, which is expensive and slow.

Why a GPT-Style Architecture is the Right Approach

A decoder-only autoregressive transformer matches Lexis Draft's requirements precisely:

1. **Autoregressive generation handles open-ended composition.** Unlike retrieval, which is limited to existing clauses, an autoregressive model can compose novel legal language token by token. Each generated token is conditioned on the full history -- including the prompt, the contract context, and all previously generated tokens in the current clause. This allows the model to produce clauses that have never appeared verbatim in the training data while maintaining coherence and legal accuracy.

2. **Causal attention preserves document coherence.** Because each token position attends to all previous positions through causal masking, the model naturally maintains consistency with previously drafted sections. When generating an indemnification clause, the model can attend to the defined terms established in the preamble, the representations and warranties section, and any other previously drafted content. This is fundamentally different from retrieval, where each clause is selected without awareness of the rest of the document.

3. **Conditional generation supports firm-specific style.** By including firm style tokens in the prompt, the model can learn to generate different syntactic patterns for different firms. Firm A might prefer "Notwithstanding anything to the contrary herein" while Firm B prefers "Subject to the limitations set forth in Section X." The autoregressive model learns these patterns from training data labeled with firm identifiers.

4. **The model size fits deployment constraints.** A 350-million-parameter GPT model requires approximately 700 MB in FP32 or 350 MB in FP16 -- well within the 24 GB A10G GPU budget with ample room for KV-cache during generation. This is substantially smaller

than general-purpose models like GPT-3 (175B) because our domain is narrow: legal contract clauses follow highly structured patterns that can be captured by a smaller model.

5. **The training data is sufficient.** With 405,000 total clauses (320K proprietary + 85K public) averaging approximately 200 tokens each, we have approximately 81 million tokens of training data. While this is small compared to general-purpose LLM pretraining corpora, it is adequate for a domain-specific model because: (a) legal language is much more structured and repetitive than general text, reducing the effective complexity of the distribution, and (b) we can augment with a curriculum that first trains on public clauses and then fine-tunes on firm-specific data.

Technical Constraints Summary

Constraint	Value	Justification
Max model parameters	~350M	24 GB A10G GPU with room for KV-cache
Max inference latency	5 seconds per clause	Interactive drafting workflow
Max sequence length	2,048 tokens	Prompt + context + clause within window
Training compute	8x A100 80GB for <= 72 hours	20,000 dollars/month GPU budget
Training data	405,000 clauses (~81M tokens)	Proprietary + public contract clauses
Min clause relevance score	85%	Business requirement for renewals

---

Section 3: Implementation Notebook Structure

This section outlines the Google Colab notebook that students will implement. Each subsection contains a mix of provided code (for setup and context) and TODO exercises (for the student to complete). The notebook uses a **publicly available legal contract dataset** -- the CUAD (Contract Understanding Attainment Dataset) from the Atticus Project, which contains 510 commercial contracts with clause-level annotations across 41 categories. We supplement this with synthetically generated contract prompts to simulate the conditional generation setup.

3.1 Data Acquisition and Preprocessing

Context for Students:

In our Lexis Draft AI scenario, we would be working with proprietary contract clauses from law firm clients. For this notebook, we use a combination of CUAD contract excerpts and a synthetic clause dataset. The techniques you learn here transfer directly to production legal text generation.

Provided Code:

```
# Install dependencies
!pip install transformers datasets tiktoken matplotlib -q
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter

# For reproducibility
torch.manual_seed(42)
np.random.seed(42)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Using device: {device}")
```

TODO 1: Build a Legal Text Tokenizer and Prepare Training Data

```
def build_legal_tokenizer_and_data(texts, vocab_size=4096, max_seq_len=256):
    """
    Build a character-level tokenizer for legal text and prepare
    training sequences.

    For simplicity we use character-level tokenization (as in the
    article's minimal GPT). In production, you would train a BPE
    tokenizer on the legal corpus.

    Args:
        texts: list of contract clause strings
        vocab_size: maximum vocabulary size
        max_seq_len: maximum sequence length for training

    Returns:
        - encode: function that maps string -> list[int]
        - decode: function that maps list[int] -> string
        - train_data: torch.Tensor of shape (num_sequences, max_seq_len)
        - actual_vocab_size: int, the true vocabulary size

    Steps:
        1. Collect all unique characters across all texts.
        2. Sort them and assign integer IDs starting from 0.
        3. Create encode and decode functions.
        4. Concatenate all texts with newline separators.
        5. Encode the full corpus into a 1D tensor of token IDs.
        6. Split into non-overlapping chunks of max_seq_len.
        7. Return the components.

    Hints:
        - sorted(set(full_text)) gives you the unique characters
        - Use a dict for char->id and id->char mappings
        - torch.tensor(encoded_ids) creates the 1D tensor
        - tensor.view(-1, max_seq_len) reshapes into chunks
          (discard the last incomplete chunk)
    """
    # YOUR CODE HERE
    pass
```

3.2 Model Architecture: Building a GPT from Scratch

Context for Students:

Now we build the complete GPT model following the architecture described in the article. We implement three classes: `CausalSelfAttention`, `TransformerBlock`, and `GPT`. Every line maps directly to a concept from the article.

TODO 2: Implement Causal Self-Attention

```
class CausalSelfAttention(nn.Module):
    """
    Multi-head causal (masked) self-attention.

    This is the core attention mechanism of GPT. Each token can
    only attend to tokens at the same position or earlier -- never
    to future tokens.

    Architecture:
        1. Single linear projection for Q, K, V (3 * d_model)
        2. Reshape into (batch, n_heads, seq_len, d_k)
        3. Compute scaled dot-product attention with causal mask
        4. Concatenate heads and project output

    Args:
        d_model: model dimension
        n_heads: number of attention heads
        max_seq_len: maximum sequence length (for mask buffer)

    Hints:
        - d_k = d_model // n_heads
        - QK^T has shape (batch, n_heads, seq_len, seq_len)
        - Causal mask: use torch.triu with diagonal=1
        - Scale by 1/sqrt(d_k) before masking
        - masked_fill with float('-inf') before softmax
    """
    def __init__(self, d_model, n_heads, max_seq_len=256):
        super().__init__()
        # YOUR CODE HERE
        pass

    def forward(self, x):
        # YOUR CODE HERE
        pass
```

TODO 3: Implement the Transformer Block and Full GPT Model

```
class TransformerBlock(nn.Module):
    """
    A single GPT Transformer block with Pre-LN architecture.

    Architecture (Pre-LN, as used in GPT-2):
        x = x + Attention(LayerNorm(x))
        x = x + FFN(LayerNorm(x))

    The FFN expands to 4 * d_model and compresses back.
    Uses GELU activation.

    Args:
        d_model: model dimension
        n_heads: number of attention heads
        max_seq_len: maximum sequence length

    Hints:
        - LayerNorm is applied BEFORE attention and FFN (Pre-LN)
        - The residual connection adds the ORIGINAL input (before LN)
        - FFN: Linear(d_model, 4*d_model) -> GELU -> Linear(4*d_model, d_model)
    """
    def __init__(self, d_model, n_heads, max_seq_len=256):
        super().__init__()
        # YOUR CODE HERE
        pass

    def forward(self, x):
        # YOUR CODE HERE
        pass


class GPT(nn.Module):
    """
    Complete GPT model for autoregressive language modeling.

    Architecture:
```

```
        1. Token embedding (vocab_size -> d_model)
        2. Positional embedding (max_seq_len -> d_model)
        3. N Transformer blocks
        4. Final LayerNorm
        5. Linear projection to vocabulary (d_model -> vocab_size)

    Args:
        vocab_size: number of tokens in the vocabulary
        d_model: model dimension
        n_heads: number of attention heads
        n_layers: number of Transformer blocks
        max_seq_len: maximum sequence length

    Forward:
        Input: token_ids of shape (batch_size, seq_len)
        Output: logits of shape (batch_size, seq_len, vocab_size)

    Hints:
        - Add token embeddings + positional embeddings
        - Pass through all Transformer blocks sequentially
        - Apply final LayerNorm
        - Project to vocab_size with a Linear layer (no bias)
    """
    def __init__(self, vocab_size, d_model, n_heads, n_layers, max_seq_len=256):
        super().__init__()
        # YOUR CODE HERE
        pass

    def forward(self, idx):
        # YOUR CODE HERE
        pass
```

3.3 Training: Loss Computation and Backpropagation

Context for Students:

With the model built, we now implement the training loop. The training objective is next-token prediction using cross-entropy loss, exactly as described in the article. We use the AdamW optimizer with a cosine learning rate schedule.

Provided Code: Training Configuration

```
# Model hyperparameters (scaled for Colab)
config = {
    'vocab_size': None,  # Set after tokenizer creation
    'd_model': 128,
    'n_heads': 4,
    'n_layers': 4,
    'max_seq_len': 256,
    'batch_size': 32,
    'learning_rate': 3e-4,
    'num_steps': 2000,
    'eval_interval': 200,
    'eval_steps': 20,
}
```

TODO 4: Implement the Training Loop

```
def train_gpt(model, train_data, config, device):
    """
    Train the GPT model using next-token prediction.

    The training loop:
        1. Sample a random batch of sequences from train_data
        2. Create input x (all tokens except the last) and
           target y (all tokens except the first)
        3. Forward pass: compute logits
```

```
        4. Compute cross-entropy loss
        5. Backward pass: compute gradients
        6. Update weights with optimizer.step()
        7. Log loss every eval_interval steps

    Args:
        model: GPT model instance
        train_data: tensor of shape (num_sequences, seq_len)
        config: dict with training hyperparameters
        device: torch device

    Returns:
        losses: list of (step, loss) tuples

    Hints:
        - Sample batch indices with torch.randint(0, len(train_data), (batch_size,))
        - x = batch[:, :-1], y = batch[:, 1:]
        - Reshape logits to (B*T, vocab_size) and y to (B*T,) for cross_entropy
        - Remember optimizer.zero_grad() before loss.backward()
    """
    # YOUR CODE HERE
    pass
```

3.4 Generation: Sampling from the Trained Model

TODO 5: Implement Autoregressive Text Generation

```
@torch.no_grad()
def generate(model, prompt_ids, max_new_tokens, temperature=0.8, top_k=40):
    """
    Generate text autoregressively from the trained model.

    At each step:
        1. Feed the current sequence through the model
        2. Take the logits at the last position
        3. Divide logits by temperature
        4. Optionally apply top-k filtering
        5. Sample from the resulting distribution
        6. Append the sampled token to the sequence
        7. Repeat until max_new_tokens or end token

    Args:
        model: trained GPT model
        prompt_ids: 1D tensor of prompt token IDs
        max_new_tokens: maximum tokens to generate
        temperature: controls randomness (lower = more deterministic)
        top_k: if > 0, only sample from top-k most likely tokens

    Returns:
        generated_ids: 1D tensor of all token IDs (prompt + generated)

    Hints:
        - Start with idx = prompt_ids.unsqueeze(0) for batch dim
        - Crop to max_seq_len if the sequence gets too long
        - For top-k: set logits below the k-th largest to -inf
        - Use torch.multinomial(probs, num_samples=1) to sample
    """
    # YOUR CODE HERE
    pass
```

3.5 Evaluation and Analysis

TODO 6: Analyze the Trained Model

```
def analyze_model(model, encode, decode, device):
    """
    Analyze the trained model's behavior.

    Produce the following analyses:
```

```
        1. Generate 5 clause completions from different prompts
        2. Plot the attention weights from the last layer for a
           sample generation (which previous tokens does each
           generated token attend to most?)
        3. Compute and plot perplexity across a held-out validation set
        4. Measure generation latency (tokens per second)

    This function should print results and display matplotlib plots.

    Args:
        model: trained GPT model
        encode: tokenizer encode function
        decode: tokenizer decode function
        device: torch device

    Hints:
        - Perplexity = exp(average cross-entropy loss)
        - For attention visualization, modify the model to return
          attention weights, or use hooks
        - Time generation with torch.cuda.Event for accurate GPU timing
    """
    # YOUR CODE HERE
    pass
```

Section 4: Production and System Design Extension

4.1 From Notebook to Production: Architecture Overview

Deploying the GPT clause generation model in production at Lexis Draft AI requires addressing several challenges that do not arise in a notebook environment. The production system must handle concurrent requests from hundreds of attorneys across 18 law firms, maintain strict data isolation between firms, and provide consistent low-latency responses.

Production Architecture:

```
Attorney's Word Add-in
        |
        v
API Gateway (per-firm VPC)
        |
        v
Prompt Construction Service
   - Retrieves contract context from document store
   - Constructs structured prompt with deal parameters
   - Applies firm-specific style tokens
   - Manages sliding window for long documents
        |
        v
Model Serving (vLLM on A10G GPU)
   - KV-cache for efficient autoregressive generation
   - Continuous batching for throughput
   - FP16 inference
   - Model weights stored in S3, loaded at startup
        |
        v
Post-processing Service
   - Defined term consistency check
   - Cross-reference validation
   - Formatting and numbering
   - Confidence scoring
        |
        v
Response to Attorney
```

4.2 Scaling Considerations

Multi-Firm Model vs. Per-Firm Models. The most critical architectural decision is whether to train a single model on pooled data from all 18 firms or train separate models per firm.

The hybrid approach works best: train a **base model** on all 405,000 clauses (pooled across firms and public data), then create **per-firm adapters** using LoRA (Low-Rank Adaptation). Each LoRA adapter adds only 2 to 4 million parameters (less than 1 percent of the base model) and captures firm-specific drafting conventions. This approach: - Maximizes the benefit of the full training corpus for general legal language modeling. - Allows each firm's style preferences to be captured without the cost of separate full models. - Reduces storage from 18 x 700 MB = 12.6 GB (separate models) to 700 MB + 18 x 8 MB = 844 MB. - Enables new firm onboarding with as few as 5,000 clauses for the LoRA adapter.

KV-Cache for Efficient Generation. During autoregressive generation, the model recomputes attention over all previous tokens at each step. The KV-cache stores the Key and Value tensors from all previous positions, so each new token only requires computing attention for one new position rather than the full sequence. For a 2,048-token context, this reduces generation time from O(T^2) to O(T) per token.

The KV-cache memory requirement for our model: - Per layer: 2 (K and V) x batch_size x n_heads x seq_len x d_k x 2 bytes (FP16) - For 16 layers, 8 heads, d_k = 64, seq_len = 2048, batch_size = 1: approximately 64 MB - This fits comfortably within the A10G's 24 GB alongside the model weights.

4.3 Data Pipeline and Continuous Learning

Data Collection. Every time an attorney accepts, edits, or rejects a generated clause, this feedback is logged (with appropriate consent and within the firm's data boundary). Over time, this creates a high-quality preference dataset: - **Accepted clauses**: positive examples weighted 1.0. - **Lightly edited clauses** (fewer than 5 edits): positive examples with the edited version weighted 1.0 and the original weighted 0.7. - **Heavily edited or rejected clauses**: negative examples for future RLHF or DPO fine-tuning.

Periodic Retraining. The base model is retrained quarterly on the expanded clause corpus. LoRA adapters are updated monthly using firm-specific feedback data. Each update is validated against a held-out clause set before deployment, ensuring that the model's quality does not regress.

4.4 Monitoring and Guardrails

Output Validation. Every generated clause passes through a validation pipeline before being presented to the attorney:

1. **Defined term check:** Verify that all capitalized terms used in the generated clause are defined in the contract context. Flag undefined terms.

2. **Cross-reference check:** Verify that all section references (e.g., "Section 4.2") correspond to actual sections in the contract.
3. **Prohibited language check:** A blocklist of phrases that should never appear in contracts (e.g., outdated legal terms, terms from the wrong jurisdiction).
4. **Confidence threshold:** If the model's average token-level probability falls below 0.15, the clause is flagged as low-confidence and the retrieval fallback is triggered instead.

Monitoring Dashboard. Real-time metrics tracked in production: - Clause relevance score (rolling 7-day average, per firm). - First-draft acceptance rate (per contract type, per firm). - Generation latency (p50, p95, p99). - Model confidence distribution. - Attorney feedback volume and sentiment.

4.5 Ethical and Professional Considerations

Attorney-Client Privilege. All contract data is privileged. The system architecture enforces strict data isolation: - Each firm's data resides in a separate AWS VPC or Azure tenant. - Model weights trained on pooled data never contain verbatim client text (the model learns patterns, not memorized clauses). - Per-firm LoRA adapters are stored within the firm's own cloud environment and are never shared. - Regular audits verify that the model does not reproduce verbatim passages from training data (using n-gram overlap detection).

Unauthorized Practice of Law. The system is explicitly positioned as a drafting assistant, not a legal advisor. Generated clauses are always presented as suggestions that require attorney review. The interface includes a persistent disclaimer and requires the attorney to affirmatively accept each clause before it is incorporated into the document.

Bias and Fairness. The training data is drawn predominantly from US commercial contracts. The model may underperform on international contracts, contracts in specialized domains (maritime, aviation), or contracts involving underrepresented jurisdictions. Lexis Draft maintains a coverage matrix documenting which contract types and jurisdictions have sufficient training data and flags queries that fall outside the well-represented areas.