Case Study: Clinical NLP for Automated Medical Coding

From Unstructured Physician Notes to Accurate ICD Billing Codes Using BERT

Section 1: Industry Context and Business Problem

Industry: Healthcare Revenue Cycle Management

The US healthcare system processes over 3.6 billion insurance claims annually, generating more than 4 trillion dollars in total expenditure. At the center of this system is **medical coding** -- the process of translating clinical documentation (physician notes, discharge summaries, operative reports) into standardized billing codes (ICD-10-CM for diagnoses, CPT for procedures).

Medical coding is the financial backbone of every hospital. If a diagnosis is coded incorrectly, the insurance claim is denied. If it is coded incompletely, the hospital is underpaid. If it is coded too aggressively, the hospital faces compliance audits and potential fraud penalties.

Today, this work is done primarily by human coders -- certified professionals who read through clinical notes and assign the appropriate codes. The US Bureau of Labor Statistics estimates there are approximately 350,000 medical coders in the US, and the profession faces a persistent shortage. The average coder processes 20-30 charts per hour, and the work is mentally demanding, error-prone, and increasingly unsustainable as clinical documentation grows in volume and complexity.

Company Profile: MedScribe AI

MedScribe AI is a Series B healthtech startup headquartered in Boston, MA, founded in 2021 by a team of former Epic Systems engineers and an NLP researcher from MIT. The company has raised 42 million dollars across two rounds (Seed: 6M, Series A: 14M, Series B: 22M) and employs approximately 85 people -- 35 in engineering, 15 in clinical operations, and the rest in sales, compliance, and administration.

MedScribe's core product is a clinical documentation intelligence platform deployed in 12 mid-size hospital systems across the US (50-500 bed facilities). The platform ingests unstructured clinical notes from the hospital's EHR (Electronic Health Record) system, extracts clinical entities (diagnoses, procedures, medications), and suggests ICD-10-CM billing codes to the coding team.

The current system uses a **rule-based NLP pipeline** built on regular expressions, medical dictionaries (SNOMED CT, RxNorm), and a legacy keyword-matching engine. This system was adequate for MedScribe's initial product launch, but it has reached its performance ceiling.

Business Challenge

MedScribe's rule-based system achieves an entity extraction accuracy of **68%** and an ICD code suggestion accuracy of **61%**. For context, human coders achieve approximately 85-92% accuracy depending on specialty.

The core failure modes are:

1. **Negation blindness.** The system cannot distinguish "patient denies chest pain" from "patient reports chest pain." Both trigger a chest pain diagnosis code. This accounts for approximately 22% of all coding errors.

2. **Context-dependent terminology.** The word "discharge" means one thing in "discharge summary" (a document type) and another in "wound discharge" (a clinical finding). The rule-based system treats them identically. Similarly, "positive" in "COVID-19 positive" versus "positive outlook on recovery" are conflated.

3. **Implicit references.** Physicians frequently use pronouns and implicit references: "The mass was biopsied. It was benign." The rule-based system cannot resolve "it" back to "the mass" and misses the benign finding.

4. **Abbreviation ambiguity.** Medical abbreviations are notoriously ambiguous -- "MS" can mean multiple sclerosis, mitral stenosis, morphine sulfate, or mental status, depending on context.

Why It Matters

The financial impact is substantial:

- MedScribe's 12 hospital clients collectively process approximately **1.2 million clinical encounters per year**.
- At the current 61% code suggestion accuracy, the coding team must manually review and correct nearly 40% of all suggestions, negating much of the automation benefit.
- Hospital clients report an average **claim denial rate of 8.2%** partly attributable to coding errors, compared to the industry benchmark of 5-6%.
- Each denied claim costs an average of 118 dollars in rework (staff time for appeal, resubmission, follow-up), totaling approximately **11.6 million dollars per year** across MedScribe's client base.
- MedScribe's contract renewal rate has dropped to 74% -- below the 85% threshold their Series B investors require. Three hospitals have cited "insufficient accuracy improvement over manual coding" as the reason for non-renewal.

The company's VP of Engineering has been tasked with building a **next-generation NLP engine** that can replace the rule-based pipeline with a deep learning system. The target: **increase entity extraction accuracy from 68% to 88%+ and ICD code suggestion accuracy from 61% to 82%+** within 6 months.

Constraints

The engineering team must work within the following constraints:

- **Compute budget:** 15,000 dollars/month for cloud GPU instances (AWS). No on-premise GPU infrastructure. Training must be feasible on 4x A100 GPUs or equivalent within 48 hours.
- **Latency:** Code suggestions must be generated within **3 seconds** of a note being finalized in the EHR. Physicians and coders will not tolerate longer waits.
- **Privacy and compliance:** All clinical data is protected under **HIPAA**. No patient data may leave the hospital's Virtual Private Cloud (VPC). The model must be deployable within each hospital's AWS GovCloud or on-premise environment. No data may be sent to external APIs.
- **Data availability:** MedScribe has access to approximately **180,000 annotated clinical notes** across its 12 clients (entity spans and ICD codes labeled by certified coders). Additionally, the team can use publicly available clinical NLP datasets for development and benchmarking.
- **Model size:** The production model must fit within **4 GB of GPU memory** at inference time (hospitals use g4dn.xlarge instances with T4 GPUs).
- **Team expertise:** The ML team consists of 5 engineers with strong Python and PyTorch experience. Two have prior NLP experience; three are transitioning from computer vision. The team is familiar with HuggingFace Transformers.

Section 2: Technical Problem Formulation

Problem Type: Sequence Labeling (Token Classification)

The core task is **Named Entity Recognition (NER)** applied to clinical text -- given an unstructured clinical note, identify and classify every mention of a clinical entity (diagnosis, procedure, medication, anatomy, lab test) with its precise span in the text.

Why NER and not text classification? Consider the input: "Patient presents with acute myocardial infarction. Started on aspirin 81mg and metoprolol 25mg. Scheduled for coronary angiography tomorrow." A text classifier could tell us this note is "cardiac-related," but that is insufficient for medical coding. We need to extract the exact entities:

- "acute myocardial infarction" -> Diagnosis
- "aspirin 81mg" -> Medication
- "metoprolol 25mg" -> Medication
- "coronary angiography" -> Procedure

We also considered framing this as a sequence-to-sequence generation problem (input: clinical note, output: list of ICD codes). However, this approach has two drawbacks: (1) it does not provide explainability -- coders need to see exactly which text spans map to which codes, and (2) it requires significantly more training data to learn the mapping from free text to structured codes end-to-end. The NER approach provides interpretable entity spans that can then be mapped to ICD codes using a deterministic lookup.

Input Specification

Input: A clinical note represented as a sequence of WordPiece tokens.

- **Raw input:** A string of clinical text, typically 200-2000 words. Example: a discharge summary, progress note, or operative report.
- **Tokenized input:** The text is tokenized using BERT's WordPiece tokenizer into subword units. A 500-word note typically produces 600-800 tokens.
- **Maximum sequence length:** 512 tokens (BERT's positional embedding limit). Notes longer than 512 tokens are processed using a sliding window approach with 128 tokens of overlap.
- **Input tensors:** `input_ids` (token indices), `attention_mask` (1 for real tokens, 0 for padding), `token_type_ids` (segment IDs, all 0 for single-sentence NER).

Each token carries lexical information (what word or subword it represents), positional information (where it appears in the sequence), and through BERT's self-attention layers, contextual information (what surrounding words reveal about its meaning). This contextual information is precisely what the rule-based system lacks and what BERT provides.

Output Specification

Output: A sequence of BIO (Beginning-Inside-Outside) labels, one per token.

The BIO tagging scheme works as follows: - **B-{entity}**: The first token of an entity span (e.g., B-Diagnosis) - **I-{entity}**: A continuation token within an entity span (e.g., I-Diagnosis) - **O**: A token that is not part of any entity

For our clinical NER task, the label set is:

Label	Description	Example
B-DIAG	Beginning of a diagnosis mention	"acute" in "acute myocardial infarction"
I-DIAG	Inside a diagnosis mention	"myocardial", "infarction"
B-PROC	Beginning of a procedure mention	"coronary" in "coronary angiography"
I-PROC	Inside a procedure mention	"angiography"
B-MED	Beginning of a medication mention	"aspirin" in "aspirin 81mg"
I-MED	Inside a medication mention	"81mg"
B-ANAT	Beginning of an anatomy mention	"left" in "left ventricle"
I-ANAT	Inside an anatomy mention	"ventricle"

Label	Description	Example
B-TEST	Beginning of a test/lab mention	"troponin" in "troponin level"
I-TEST	Inside a test/lab mention	"level"
O	Outside any entity	"the", "patient", "was"

This gives us 11 labels total (5 entity types x 2 BIO tags + O).

Why BIO over simpler tagging schemes? A flat per-token classification (entity type only, no B/I distinction) cannot handle adjacent entities of the same type. For example, in "prescribed aspirin metoprolol," we need B-MED for "aspirin" and B-MED for "metoprolol" to distinguish them as separate entities. Without the B/I distinction, we would merge them into one span.

Mathematical Foundation

Before defining the loss function, let us build up the mathematical foundations that make BERT effective for this task.

Contextual Representations via Self-Attention. The fundamental reason BERT works for clinical NER is self-attention. For a sequence of n tokens with hidden dimension d, the self-attention mechanism computes:

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

where $Q = XW^Q$, $K = XW^K$, $V = XW^V$ are linear projections of the input $X \in \mathbb{R}^{n \times d}$.

The key insight for clinical NER is in what the attention matrix $A = \text{softmax}(QK^T/\sqrt{d_k})$ represents. Each entry A_{ij} is the attention weight from token i to token j -- how much token i should incorporate information from token j. In the sentence "patient denies chest pain," the representation of "chest pain" will attend heavily to "denies," allowing the model to learn that this is a **negated** finding.

This is precisely the bidirectional context that the rule-based system cannot capture. A left-to-right model processing "patient denies chest" has not yet seen "pain" and does not know the complete entity. A right-to-left model processing "pain chest denies patient" loses the natural syntactic structure. BERT processes all tokens simultaneously, allowing "chest pain" to attend to "denies" regardless of position.

Why Cross-Entropy for Token Classification. For each token position i, the model produces a probability distribution over the 11 labels via a softmax layer:

$$P(y_i = c \mid \mathbf{h}_i) = \frac{\exp(\mathbf{w}_c^T \mathbf{h}_i + b_c)}{\sum_{c'=1}^{C} \exp(\mathbf{w}_{c'}^T \mathbf{h}_i + b_{c'})}$$

where $\mathbf{h}_i \in \mathbb{R}^{768}$ is the hidden state of token i from BERT's final layer, $\mathbf{w}_c \in \mathbb{R}^{768}$ and b_c are the classification head parameters for class c, and $C = 11$ is the number of labels.

The cross-entropy loss for a single token measures how surprised the model is by the true label. If the model assigns probability 0.95 to the correct label, the loss is $-\log(0.95) = 0.05$ (small). If it assigns probability 0.01, the loss is $-\log(0.01) = 4.6$ (large). This is a direct consequence of maximum likelihood estimation -- minimizing cross-entropy is equivalent to maximizing the likelihood of the observed labels under the model's predicted distribution.

The Pre-training Advantage. BERT's pre-training on 3.3 billion words of text using Masked Language Modeling (MLM) gives it a strong prior understanding of English syntax and semantics. When we fine-tune on 180,000 clinical notes, we are not learning language from scratch -- we are adapting an already-competent language model to the clinical domain. This is especially important given the constraint that our training dataset, while large by clinical standards, is modest by general NLP standards.

The mathematical justification comes from transfer learning theory. The pre-trained weights θ_{pre} provide a strong initialization point in parameter space that is already near a good solution for language understanding tasks. Fine-tuning with a small learning rate (2×10^{-5}) means we make small adjustments from this initialization, staying close to the pre-trained solution while adapting to clinical text patterns.

Loss Function

The training objective is the token-level cross-entropy loss, summed over all non-padding tokens:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c})$$

where: - N is the total number of non-padding tokens across the batch - $C = 11$ is the number of entity labels - $y_{i,c} \in \{0, 1\}$ is the one-hot encoded true label for token i - $\hat{y}_{i,c} = P(y_i = c \mid \mathbf{h}_i)$ is the predicted probability for class c

Justification of each component:

1. **Token-level averaging ($1/N$).** We average over tokens rather than summing to make the loss invariant to sequence length. Without this, longer notes would dominate the gradient, causing the model to overfit on long discharge summaries at the expense of shorter progress notes.

2. **Cross-entropy over alternatives.** We use cross-entropy rather than mean squared error because the output is a categorical distribution. MSE would treat label 3 as "closer" to label 4 than to label 1, but our labels are categorical -- B-DIAG is not "closer" to I-DIAG than to B-MED in any meaningful sense.

3. **Class weighting (optional but recommended).** In clinical text, approximately 85% of tokens are O (outside any entity). This severe class imbalance means the model can achieve 85% accuracy by simply predicting O for everything. To address this, we apply inverse-frequency class weights:

$$w_c = \frac{N}{C \times n_c}$$

where n_c is the number of tokens with label c. This upweights rare entity labels, forcing the model to pay more attention to them. If we remove this weighting, the model converges to predict O for nearly all tokens -- an ablation that is worth running to demonstrate its importance.

1. **Ignoring padding tokens.** We mask out padding tokens (attention_mask = 0) from the loss computation. Including padding would add noise -- the model would receive gradient signal for predicting labels on tokens that carry no information.

2. **Ignoring special tokens.** We also exclude the [CLS] and [SEP] tokens from the loss, as they do not correspond to real text that needs labeling.

Evaluation Metrics

Primary Metric: Entity-Level F1 Score

We use the **strict entity-level F1 score** as the primary metric. An entity prediction is counted as correct only if both the entity type AND the exact span boundaries match the ground truth.

$$\text{Precision} = \frac{\text{Correctly predicted entities}}{\text{Total predicted entities}}, \quad \text{Recall} = \frac{\text{Correctly predicted entities}}{\text{Total true entities}}$$

$$F1 = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}}$$

Target: Entity-level F1 >= 0.88 (up from 0.68 with the rule-based system).

Why entity-level F1 and not token-level accuracy? Consider the entity "acute myocardial infarction" (3 tokens). If the model predicts "myocardial infarction" (missing "acute"), it gets 2/3 tokens correct (67% token accuracy) but the entity-level prediction is wrong -- the span boundaries do not match. For medical coding, a partial entity extraction is as useless as no extraction, so entity-level F1 is the appropriate metric.

Secondary Metrics:

Metric	Target	Rationale
Per-entity-type F1	>= 0.85 each	Ensures no entity type is neglected
Negation detection accuracy	>= 0.92	Critical for avoiding false-positive diagnoses
Inference latency (p95)	< 500ms per note	Must fit within the 3-second end-to-end budget
Model size	< 4 GB GPU memory	Must fit on T4 GPU

Baseline

Rule-Based Baseline (Current System):

MedScribe's existing system uses a three-stage pipeline:

1. **Dictionary lookup:** Match text spans against SNOMED CT (a medical terminology database) using exact and fuzzy string matching.
2. **Regular expression patterns:** Hand-crafted regex patterns for common clinical phrases (e.g., `(diagnos(is|ed)|dx):?\s+(.+?)(?:\.|,|;)` to capture diagnosis mentions).
3. **Negation detection:** The NegEx algorithm (Chapman et al., 2001), which uses a list of ~180 negation trigger phrases ("denies", "no evidence of", "ruled out") and a simple window-based approach to mark entities within 5 tokens of a trigger as negated.

This system achieves: - Entity extraction F1: **0.68** - Negation detection accuracy: **0.74** - Inference latency: **~200ms per note** (fast, as it is rule-based)

The rule-based system is insufficient because: - It fails on paraphrases and novel expressions not in the dictionary - Its negation detection is too simplistic (window-based, not contextual) - It cannot handle implicit references or complex sentence structures - Adding new rules is labor-intensive and creates a brittle, unmaintainable codebase

Why BERT is the Right Technical Approach

BERT's fundamental properties match MedScribe's requirements precisely:

1. **Deep bidirectional context resolves negation.** The core failure of the rule-based system -- negation blindness -- is precisely what BERT's bidirectional self-attention addresses. When processing "patient denies chest pain," every layer of BERT allows "chest pain" to attend to "denies." By layer 12, the representation of "chest pain" is deeply informed by the negation context. This is not shallow concatenation (as in ELMo) -- it is learned jointly across all layers.

2. **Pre-training provides domain-general language understanding.** BERT-Base was pre-trained on 3.3 billion words of general English text. This gives it robust knowledge of syntax, semantics, and common-sense reasoning that transfers to clinical text. Clinical language, while specialized, still follows English grammar and sentence structure.

3. **Fine-tuning is data-efficient.** With 180,000 annotated notes, MedScribe has a substantial but not enormous training set. BERT's pre-training means the model already understands language; fine-tuning only needs to teach it the specifics of clinical entity types. This data efficiency is critical given the high cost of clinical annotation (15-25 dollars per note for expert annotation).

4. **Model size fits the deployment constraint.** BERT-Base has 110M parameters, requiring approximately 440 MB in FP32 or 220 MB in FP16. Even with the token classification head, this fits comfortably within the 4 GB T4 GPU memory constraint. For even tighter budgets, DistilBERT (66M parameters) is a viable alternative.

5. **Clinical BERT variants exist.** Models like BioBERT (pre-trained on PubMed), ClinicalBERT (pre-trained on MIMIC clinical notes), and PubMedBERT provide domain-

adapted starting points. These models have already learned clinical vocabulary, abbreviations, and writing patterns during pre-training, giving them a significant advantage over general-domain BERT for clinical NER.

Technical Constraints

Constraint	Value	Justification
Max model size	~440 MB (FP32)	4 GB T4 GPU with room for batch processing
Max inference latency	500 ms per note	3-second end-to-end budget, with 2.5 seconds for pre/post-processing
Max sequence length	512 tokens	BERT positional embedding limit
Training compute	4x A100 for <= 48 hours	15,000 dollars/month GPU budget
Training data	180,000 annotated notes	Available from MedScribe's 12 hospital clients
Min entity-level F1	0.88	Business requirement for contract renewal

---

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 the **BC5CDR dataset** (BioCreative V Chemical Disease Relation) -- a publicly available biomedical NER dataset with 1,500 annotated PubMed abstracts containing chemical and disease entity annotations. This dataset is freely available through HuggingFace Datasets and closely mirrors the structure of clinical NER while being accessible without credentialed access.

3.1 Data Acquisition and Preprocessing

Context for Students:

In our MedScribe AI scenario, we would be working with clinical notes from hospital EHR systems. For this notebook, we use the BC5CDR dataset -- a widely-used biomedical NER benchmark that contains PubMed abstracts annotated with chemical (medication) and disease (diagnosis) entities. The techniques you learn here transfer directly to clinical NER.

Provided Code:

```
# Install dependencies
!pip install transformers datasets seqeval matplotlib scikit-learn -q

import torch
import numpy as np
from datasets import load_dataset
from transformers import BertTokenizerFast

# Load the BC5CDR dataset (biomedical NER)
dataset = load_dataset("tner/bc5cdr")

# Examine the dataset structure
print("Dataset splits:", dataset)
print("\nSample entry:")
sample = dataset['train'][0]
print("Tokens:", sample['tokens'])
print("Tags:", sample['tags'])
print("\nLabel mapping:")
label_names = dataset['train'].features['tags'].feature.names
for i, name in enumerate(label_names):
    print(f"  {i}: {name}")

# Initialize BERT tokenizer
tokenizer = BertTokenizerFast.from_pretrained('bert-base-uncased')
```

TODO 1: Data Exploration

```
def explore_dataset(dataset):
    """
    Analyze the BC5CDR dataset to understand its characteristics.

    Compute and print the following statistics:
    1. Number of examples in train/validation/test splits
    2. Average number of tokens per example
    3. Distribution of entity types (count of B-Chemical, I-Chemical,
       B-Disease, I-Disease, O labels across the training set)
    4. Percentage of tokens that are entity tokens vs. O tokens
    5. Average number of entities per example
    6. Distribution of entity lengths (in tokens)

    Args:
        dataset: HuggingFace dataset with 'tokens' and 'tags' fields

    Returns:
        dict with keys: 'split_sizes', 'avg_tokens', 'label_distribution',
              'entity_percentage', 'avg_entities', 'entity_lengths'

    Hints:
    - Use dataset['train'].features['tags'].feature.names to get label names
    - An entity starts at any B- tag and continues through consecutive I- tags
    - To count entities, iterate through tags and count B- tag occurrences
    """
    # YOUR CODE HERE
    pass
```

TODO 2: Tokenization Alignment

```
def tokenize_and_align_labels(examples, tokenizer, max_length=128):
    """
    Tokenize text and align NER labels with WordPiece subword tokens.

    BERT's WordPiece tokenizer may split a single word into multiple subword
    tokens. For example, "myocardial" might become ["my", "##ocard", "##ial"].
    When this happens, we need to decide what label to assign to the subword
    tokens.

    Strategy: Assign the original word's label to the FIRST subword token.
    Assign -100 to all subsequent subword tokens (this tells PyTorch to
    ignore them in the loss computation).

    Args:
        examples: dict with 'tokens' (list of list of strings) and
                  'tags' (list of list of ints)
        tokenizer: BertTokenizerFast instance
        max_length: maximum sequence length (default 128)

    Returns:
        dict with 'input_ids', 'attention_mask', 'labels' fields
        ready for model training

    Hints:
    - Use tokenizer(examples['tokens'], is_split_into_words=True,
          truncation=True, max_length=max_length, padding='max_length')
    - Use tokenized_inputs.word_ids(batch_index=i) to get the mapping
      from subword tokens to original words
    - word_ids() returns None for special tokens ([CLS], [SEP], [PAD])
      -- assign -100 to these positions
    - For the first subword of each word, copy the original label
    - For subsequent subwords of the same word, assign -100

    Step-by-step:
    1. Tokenize all examples with is_split_into_words=True
    2. For each example in the batch:
       a. Get word_ids for the tokenized sequence
       b. Initialize labels list
       c. Track previous_word_id (starts as None)
       d. For each token position:
          - If word_id is None: append -100
          - If word_id != previous_word_id: append the original label
          - If word_id == previous_word_id: append -100
       e. Update previous_word_id
    3. Add 'labels' to the tokenized output dict
    """
    # YOUR CODE HERE
    pass
```

3.2 Exploratory Data Analysis

TODO 3: Visualize Entity Distributions

```
def plot_entity_analysis(dataset, label_names):
    """
    Create a comprehensive EDA visualization with 4 subplots:

    1. Bar chart: Count of each entity type (B-Chemical, I-Chemical,
       B-Disease, I-Disease, O) in the training set
    2. Histogram: Distribution of sequence lengths (number of tokens
       per example)
    3. Histogram: Distribution of entity lengths (number of tokens per
       entity) -- separate colors for Chemical vs Disease
    4. Bar chart: Top 20 most frequent entity surface forms (the actual
       text of extracted entities)

    Args:
        dataset: HuggingFace dataset with 'tokens' and 'tags'
        label_names: list of label name strings

    Hints:
    - Use matplotlib with plt.subplots(2, 2, figsize=(14, 10))
    - To extract entity surface forms, iterate through tokens/tags
      and concatenate tokens that form each entity
    - Use Counter from collections for frequency counting
    - Add clear titles, axis labels, and a tight_layout()

    Thought questions (answer in a markdown cell after this code):
    - Is the dataset balanced between Chemical and Disease entities?
    - What is the most common entity length? Does this inform your
      choice of max_length?
    - Are there any surprising entries in the top 20 entity list?
    """
    # YOUR CODE HERE
    pass
```

3.3 Baseline Approach

Context for Students:

Before training BERT, we establish a baseline using a simple dictionary-based approach -- similar to MedScribe's original rule-based system. This gives us a lower bound on performance and helps us appreciate the value of contextual representations.

TODO 4: Dictionary-Based NER Baseline

```
def build_entity_dictionary(dataset, label_names):
    """
    Build a simple entity dictionary from the training set.

    Extract all entity surface forms (e.g., "aspirin", "myocardial infarction")
    and their entity types from the training data. Store them in a dictionary
    mapping surface form -> most common entity type.

    Args:
        dataset: training split with 'tokens' and 'tags'
        label_names: list of label name strings

    Returns:
        dict mapping entity string (lowercased) -> entity type string

    Hints:
    - Iterate through each example's tokens and tags
    - When you encounter a B- tag, start collecting tokens for a new entity
    - Continue collecting while you see I- tags of the same type
    - When the entity ends, join the tokens and add to the dictionary
    - If an entity form appears with multiple types, keep the most common
    """
    # YOUR CODE HERE
    pass


def dictionary_ner(tokens, entity_dict):
    """
    Perform NER using simple dictionary lookup with longest-match.

    For each position in the token sequence, check if any subsequence
    starting at that position matches an entry in the entity dictionary.
    Use longest-match-first strategy.

    Args:
        tokens: list of token strings
        entity_dict: dict mapping entity string -> entity type

    Returns:
        list of BIO tag strings, same length as tokens

    Hints:
    - For each start position, try matching subsequences of decreasing
      length (longest match first)
    - When a match is found, assign B-{type} to the first token and
      I-{type} to subsequent tokens in the match
    - Skip positions that are already tagged (inside a previously
      matched entity)
    - Assign 'O' to all unmatched tokens
    """
    # YOUR CODE HERE
    pass
```

Provided Code: Evaluate Baseline

```
from seqeval.metrics import classification_report, f1_score

def evaluate_baseline(dataset_split, entity_dict, label_names):
    """Evaluate the dictionary baseline on a dataset split."""
    true_labels = []
    pred_labels = []

    for example in dataset_split:
        tokens = example['tokens']
        true_tags = [label_names[t] for t in example['tags']]
        pred_tags = dictionary_ner(tokens, entity_dict)
        true_labels.append(true_tags)
        pred_labels.append(pred_tags)

    print("Dictionary Baseline Results:")
    print(classification_report(true_labels, pred_labels))
    return f1_score(true_labels, pred_labels)

# Build dictionary and evaluate
entity_dict = build_entity_dictionary(dataset['train'], label_names)
baseline_f1 = evaluate_baseline(dataset['test'], entity_dict, label_names)
print(f"\nBaseline Entity-Level F1: {baseline_f1:.4f}")
```

3.4 Model Design

Context for Students:

Now we build the BERT-based NER model. The architecture is straightforward: BERT-Base produces a 768-dimensional hidden state for each token, and we add a single linear classification layer that maps each hidden state to one of our 11 labels (5 entity types x 2 BIO tags + O). The power comes entirely from BERT's pre-trained representations and the fine-tuning process.

This is a direct application of the token classification approach described in the BERT paper (Section: Fine-tuning for NER). Each token's contextualized representation from the final Transformer layer is independently classified.

TODO 5: Build the NER Model

```
import torch.nn as nn
from transformers import BertModel

class BertForClinicalNER(nn.Module):
    """
    BERT-based model for clinical Named Entity Recognition.

    Architecture:
    1. BERT-Base encoder (768-dim hidden states, 12 layers, 12 heads)
    2. Dropout layer for regularization
    3. Linear classification head: 768 -> num_labels

    The classification head is applied independently to each token position.
    During training, tokens with label -100 are ignored in the loss computation.

    Args:
        num_labels: number of NER label classes (default: 11)
        dropout_rate: dropout probability (default: 0.1)
        model_name: pre-trained BERT model name (default: 'bert-base-uncased')

    Forward pass:
        Input: input_ids, attention_mask, labels (optional)
        Output: dict with 'logits' (always) and 'loss' (if labels provided)

    Hints:
    - Use BertModel.from_pretrained(model_name) to load BERT (NOT
      BertForTokenClassification -- build it yourself to understand
      the architecture)
    - The BERT output has shape (batch_size, seq_len, 768)
    - Apply dropout to the BERT output before the linear layer
    - Use nn.CrossEntropyLoss(ignore_index=-100) to handle masked tokens
    - The loss should be computed over ALL non-masked token positions

    Step-by-step for __init__:
    1. Load pre-trained BERT model
    2. Create dropout layer (nn.Dropout)
    3. Create linear classifier (nn.Linear: 768 -> num_labels)
    Step-by-step for forward:
    1. Pass input_ids and attention_mask through BERT
    2. Get the last hidden state from BERT output
    3. Apply dropout
    4. Pass through the linear classifier to get logits
    5. If labels are provided, compute cross-entropy loss
    6. Return dict with logits (and loss if applicable)
    """
    def __init__(self, num_labels=5, dropout_rate=0.1,
                 model_name='bert-base-uncased'):
        super().__init__()
        # YOUR CODE HERE
        pass

    def forward(self, input_ids, attention_mask, labels=None):
        # YOUR CODE HERE
        pass
```

Verification Cell:

```
# Test your model
model = BertForClinicalNER(num_labels=len(label_names))
print(f"Total parameters: {sum(p.numel() for p in model.parameters()):,}")
print(f"Trainable parameters: {sum(p.numel() for p in model.parameters() if p.requires_grad):,}")

# Test forward pass with dummy input
dummy_input = tokenizer("The patient was diagnosed with diabetes mellitus",
                        return_tensors="pt", padding="max_length",
                        max_length=32, truncation=True)
dummy_labels = torch.zeros(1, 32, dtype=torch.long)
output = model(dummy_input['input_ids'], dummy_input['attention_mask'],
               labels=dummy_labels)

assert 'logits' in output, "Output must contain 'logits'"
assert 'loss' in output, "Output must contain 'loss' when labels are provided"
assert output['logits'].shape == (1, 32, len(label_names)), \
    f"Expected logits shape (1, 32, {len(label_names)}), got {output['logits'].shape}"
print("All assertions passed. Model architecture is correct.")
```

3.5 Training Strategy

Context for Students:

Fine-tuning BERT requires careful hyperparameter choices. The pre-trained weights encode valuable language knowledge, and aggressive training can destroy this knowledge (catastrophic forgetting). We use a small learning rate and warm up gradually.

Provided Code: Training Configuration

```
from torch.optim import AdamW
from transformers import get_linear_schedule_with_warmup
from torch.utils.data import DataLoader

# Training hyperparameters
BATCH_SIZE = 16
LEARNING_RATE = 3e-5       # Small LR to preserve pre-trained knowledge
NUM_EPOCHS = 5
WARMUP_RATIO = 0.1         # Warm up over first 10% of training steps
WEIGHT_DECAY = 0.01        # L2 regularization (excludes bias and LayerNorm)
MAX_GRAD_NORM = 1.0        # Gradient clipping for stability

# Prepare tokenized datasets
tokenized_train = dataset['train'].map(
    lambda x: tokenize_and_align_labels(x, tokenizer, max_length=128),
    batched=True, remove_columns=dataset['train'].column_names
)
```

```
tokenized_val = dataset['validation'].map(
    lambda x: tokenize_and_align_labels(x, tokenizer, max_length=128),
    batched=True, remove_columns=dataset['validation'].column_names
)

tokenized_train.set_format('torch')
tokenized_val.set_format('torch')

train_loader = DataLoader(tokenized_train, batch_size=BATCH_SIZE, shuffle=True)
val_loader = DataLoader(tokenized_val, batch_size=BATCH_SIZE)

# Setup optimizer with weight decay (exclude bias and LayerNorm)
no_decay = ['bias', 'LayerNorm.weight']
optimizer_grouped_parameters = [
    {'params': [p for n, p in model.named_parameters()
                if not any(nd in n for nd in no_decay)],
     'weight_decay': WEIGHT_DECAY},
    {'params': [p for n, p in model.named_parameters()
                if any(nd in n for nd in no_decay)],
     'weight_decay': 0.0}
]
optimizer = AdamW(optimizer_grouped_parameters, lr=LEARNING_RATE)

total_steps = len(train_loader) * NUM_EPOCHS
warmup_steps = int(total_steps * WARMUP_RATIO)
scheduler = get_linear_schedule_with_warmup(
    optimizer, num_warmup_steps=warmup_steps,
    num_training_steps=total_steps
)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model.to(device)
print(f"Training on: {device}")
print(f"Total training steps: {total_steps}")
print(f"Warmup steps: {warmup_steps}")
```

TODO 6: Training Loop

```
def train_epoch(model, train_loader, optimizer, scheduler, device,
                max_grad_norm=1.0):
    """
    Train the model for one epoch.

    Args:
        model: BertForClinicalNER instance
        train_loader: DataLoader for training data
        optimizer: AdamW optimizer
        scheduler: learning rate scheduler
        device: torch device (cuda or cpu)
        max_grad_norm: maximum gradient norm for clipping

    Returns:
        float: average training loss for the epoch

    Hints:
    - Set model to training mode with model.train()
    - For each batch:
      1. Move input_ids, attention_mask, labels to device
      2. Zero the gradients
      3. Forward pass through the model
      4. Get the loss from the output
      5. Backward pass (loss.backward())
      6. Clip gradients with torch.nn.utils.clip_grad_norm_
      7. Update weights (optimizer.step())
      8. Update learning rate (scheduler.step())
      9. Accumulate the loss for logging
    - Return the average loss across all batches
    - Print progress every 50 batches (batch loss and learning rate)
    """
    # YOUR CODE HERE
    pass


def evaluate(model, val_loader, device, label_names):
    """
    Evaluate the model on a validation/test set.

    Args:
        model: BertForClinicalNER instance
        val_loader: DataLoader for validation/test data
        device: torch device
        label_names: list of label name strings

    Returns:
        tuple: (average_loss, entity_f1, classification_report_string)

    Hints:
    - Set model to evaluation mode with model.eval()
    - Use torch.no_grad() context manager
    - For each batch:
      1. Forward pass to get logits and loss
      2. Get predictions via argmax over the label dimension
      3. Convert predictions and true labels to label name strings
      4. IMPORTANT: Skip tokens where the true label is -100
         (subword tokens, special tokens, padding)
    - Use seqeval.metrics.f1_score and classification_report
    - Collect predictions at the SEQUENCE level (list of list of
      label strings) for seqeval
    """
    # YOUR CODE HERE
    pass
```

Provided Code: Training Execution

```
import time

best_f1 = 0.0
train_losses = []
val_f1_scores = []

for epoch in range(NUM_EPOCHS):
    start_time = time.time()

    # Train
    train_loss = train_epoch(model, train_loader, optimizer, scheduler,
                             device, MAX_GRAD_NORM)
    train_losses.append(train_loss)

    # Evaluate
    val_loss, val_f1, report = evaluate(model, val_loader, device, label_names)
    val_f1_scores.append(val_f1)

    epoch_time = time.time() - start_time

    print(f"\nEpoch {epoch+1}/{NUM_EPOCHS} ({epoch_time:.1f}s)")
    print(f"  Train Loss: {train_loss:.4f}")
    print(f"  Val Loss:   {val_loss:.4f}")
    print(f"  Val F1:     {val_f1:.4f}")

    if val_f1 > best_f1:
        best_f1 = val_f1
        torch.save(model.state_dict(), 'best_model.pt')
        print(f"  New best model saved (F1: {best_f1:.4f})")

    print("\n" + report)

print(f"\nTraining complete. Best validation F1: {best_f1:.4f}")
```

3.6 Evaluation

TODO 7: Comprehensive Evaluation

```
def comprehensive_evaluation(model, test_loader, device, label_names,
                             baseline_f1):
    """
    Perform a thorough evaluation comparing BERT to the baseline.

    Generate the following:
    1. Entity-level classification report (precision, recall, F1 per type)
    2. Confusion matrix for token-level predictions (use sklearn)
    3. A comparison bar chart: Baseline F1 vs BERT F1 for each entity type
    4. Training curves plot: train loss and validation F1 vs epoch

    Args:
        model: trained BertForClinicalNER (load best_model.pt weights)
        test_loader: DataLoader for test data
        device: torch device
        label_names: list of label name strings
        baseline_f1: float, the dictionary baseline F1 score

    Returns:
        dict with 'overall_f1', 'per_type_f1', 'improvement_over_baseline'

    Hints:
    - Load best model weights: model.load_state_dict(torch.load('best_model.pt'))
    - Use seqeval for entity-level metrics
    - Use sklearn.metrics.confusion_matrix for token-level confusion
    - Use matplotlib for all plots
    - For the comparison chart, you will need to compute per-entity-type
      F1 for the dictionary baseline as well

    Thought questions (answer in a markdown cell after this code):
    - Which entity type benefits most from BERT? Why?
    - Is there a significant gap between precision and recall for any
      entity type? What does this tell you about the model?
    - How does the improvement scale with the baseline performance?
    """
    # YOUR CODE HERE
    pass
```

3.7 Error Analysis

TODO 8: Systematic Error Analysis

```
def error_analysis(model, dataset_split, tokenizer, device, label_names,
                   num_examples=100):
    """
    Perform systematic error analysis on model predictions.

    Analyze the first num_examples from the dataset and categorize errors
    into the following types:
    1. Boundary errors: entity detected but span boundaries are wrong
    2. Type errors: entity span is correct but type is wrong
    3. Missing entities: true entity completely missed (false negative)
    4. Hallucinated entities: entity predicted where none exists (false positive)
    5. Negation errors: entity in a negated context incorrectly tagged
       (check for negation trigger words within 5 tokens)

    For each error category, collect and print:
    - Count and percentage of total errors
    - 3 representative examples showing the token sequence, true labels,
      and predicted labels (highlight the error region)

    Args:
        model: trained BertForClinicalNER
        dataset_split: dataset split to analyze
        tokenizer: BertTokenizerFast instance
        device: torch device
        label_names: list of label name strings
        num_examples: number of examples to analyze

    Returns:
        dict mapping error_type -> list of error instances

    Hints:
    - For each example, compare predicted entity spans to true entity spans
    - An entity span is defined by its start index, end index, and type
    - Boundary error: overlap between predicted and true span, but not exact
    - Use negation triggers: ['no', 'not', 'without', 'denies', 'denied',
      'negative', 'absent', 'ruled out', 'unlikely']
    - Format examples clearly with aligned columns for readability
    """
    # YOUR CODE HERE
    pass
```

3.8 Inference Optimization

TODO 9: Latency Profiling and Optimization

```
def profile_inference(model, tokenizer, device, sample_texts=None):
    """
    Profile model inference latency and explore optimization strategies.

    Steps:
    1. Measure baseline inference latency on sample clinical texts
       (average over 100 runs, report mean, p50, p95, p99)
    2. Compare latency for different sequence lengths (32, 64, 128, 256, 512)
    3. Measure the effect of batch size on throughput (notes/second)
    4. Implement and measure dynamic padding (pad to longest in batch,
       not max_length) -- report the speedup

    Args:
        model: trained BertForClinicalNER
        tokenizer: BertTokenizerFast
        device: torch device
        sample_texts: optional list of sample clinical texts. If None,
            use the following defaults:
            [
                "Patient presents with acute chest pain radiating to left arm.",
                "Prescribed metformin 500mg twice daily for type 2 diabetes.",
                "CT scan of abdomen shows no evidence of appendicitis.",
                "History of hypertension, currently on lisinopril 10mg.",
            ]

    Returns:
        dict with latency statistics and throughput measurements

    Hints:
    - Use torch.cuda.synchronize() before timing on GPU
    - Use time.perf_counter() for high-resolution timing
    - For dynamic padding, tokenize the batch without max_length,
      then pad to the longest sequence in the batch
    - Calculate throughput as: batch_size / batch_latency
    - Create a line plot: sequence length vs latency
    - Create a bar chart: batch size vs throughput

    MedScribe's requirement: p95 latency < 500ms per note.
    Does your model meet this requirement?
    """
    # YOUR CODE HERE
    pass
```

3.9 Ethical and Regulatory Analysis

TODO 10: Clinical AI Ethics Assessment

```
def ethics_assessment(model, tokenizer, device, label_names):
    """
    Conduct an ethical impact assessment for deploying this NER model
    in a clinical setting.

    Implement the following analyses:

    1. Demographic bias test: Run the model on matched sentence pairs
       that differ only in demographic terms. For example:
       - "The [male/female] patient was diagnosed with depression."
       - "The [young/elderly] patient has chronic pain."
       - "The [African-American/Caucasian] patient presents with..."
       Compare entity predictions across demographic variants. Any
       difference indicates potential bias.

    2. Confidence calibration: For all predictions on the test set,
       bin predictions by confidence (softmax probability) and compute
       accuracy within each bin. Plot a reliability diagram. A well-
       calibrated model should have accuracy ~= confidence in each bin.
       Poor calibration is dangerous in clinical settings (overconfident
       wrong predictions).

    3. Failure mode severity ranking: For each error from error_analysis,
       assign a clinical severity score:
       - Critical (3): missed diagnosis, wrong medication
       - Moderate (2): wrong anatomy, missed lab test
       - Minor (1): boundary error, partial entity

       Report the distribution of error severities.

    Print a summary with:
    - Whether demographic bias was detected (yes/no for each pair)
    - Expected Calibration Error (ECE) -- formula:
      ECE = sum over bins of (bin_size/total * |accuracy - confidence|)
    - Distribution of error severities
    - A brief recommendation: is this model safe to deploy? What
      guardrails are needed?

    Args:
        model: trained BertForClinicalNER
        tokenizer: BertTokenizerFast
        device: torch device
        label_names: list of label name strings

    Returns:
        dict with 'bias_results', 'ece', 'severity_distribution',
             'deployment_recommendation'

    Hints:
    - For bias testing, create pairs of sentences that are identical
      except for the demographic term
    - For calibration, use torch.softmax on logits and take the max
      probability as the confidence score
    - Use 10 equally-spaced bins for the reliability diagram
    - ECE below 0.05 is considered well-calibrated
    """
    # YOUR CODE HERE
    pass
```

Section 4: Production and System Design Extension

This section describes how MedScribe would deploy the BERT-based NER system in a production environment serving 12 hospital clients. This is a system design exercise for advanced students.

Architecture Overview

The production system consists of five major components:

```
Hospital EHR System
        |
        v
[Ingestion Service] -- receives HL7/FHIR messages when notes are finalized
        |
        v
```

```
[Preprocessing Pipeline] -- tokenization, sentence splitting, PHI redaction
        |
        v
[NER Inference Service] -- BERT model serving with batched inference
        |
        v
[Post-Processing Pipeline] -- entity consolidation, ICD code mapping,
                              negation resolution, confidence scoring
        |
        v
[Coding Dashboard API] -- serves suggestions to the human coder interface
        |
        v
[Human Coder Review] -- final review, approval, and feedback collection
```

Each component runs as a Kubernetes pod within the hospital's VPC, with no data leaving the network.

API Design

NER Prediction Endpoint

```
POST /api/v1/predict
Content-Type: application/json

Request:
{
    "note_id": "string (UUID)",
    "note_text": "string (clinical note text, max 10,000 chars)",
    "note_type": "string (discharge_summary|progress_note|operative_report)",
    "priority": "string (normal|urgent)",
    "metadata": {
        "encounter_id": "string",
        "department": "string",
        "provider_id": "string"
    }
}

Response:
{
    "note_id": "string",
    "entities": [
        {
            "text": "acute myocardial infarction",
            "type": "DIAGNOSIS",
            "start_char": 34,
            "end_char": 60,
            "confidence": 0.94,
            "icd_codes": [
                {"code": "I21.9", "description": "Acute MI, unspecified",
                 "confidence": 0.87}
            ],
            "is_negated": false,
            "negation_trigger": null
        }
    ],
    "processing_time_ms": 342,
    "model_version": "bert-clinical-ner-v2.3.1"
}
```

Batch Prediction Endpoint

```
POST /api/v1/predict/batch
Content-Type: application/json

Request:
{
    "notes": [
        {"note_id": "string", "note_text": "string", "note_type": "string"}
    ],
    "max_batch_size": 32
}

Response:
{
    "results": [ ... ],  // Array of individual prediction responses
    "total_processing_time_ms": 1240,
    "notes_processed": 32
}
```

Serving Infrastructure

Model Serving Framework: TorchServe with custom handler for BERT tokenization and BIO-to-entity post-processing.

Scaling Strategy: - Each hospital deployment runs a dedicated Kubernetes cluster (EKS on AWS GovCloud) - The NER inference service runs as a Deployment with HPA (Horizontal Pod Autoscaler) - Scale trigger: average GPU utilization > 70% or p95 latency > 400ms - Minimum replicas: 2 (for redundancy) - Maximum replicas: 8 (budget constraint) - Each pod runs on a g4dn.xlarge instance (1x T4 GPU, 4 vCPU, 16 GB RAM)

Model Loading: - Model weights are stored in S3 (within the VPC) and downloaded at pod startup - Warm-up inference runs 10 dummy predictions before the pod is marked as ready - ReadinessProbe: POST to /api/v1/health with a test sentence, expect 200 within 2 seconds

Latency Budget

End-to-end latency budget for a single note (target: < 3 seconds):

Component	Budget	Notes
EHR webhook receipt	50ms	HL7 FHIR message parsing
PHI redaction	100ms	Regex-based, in-memory
Sentence splitting	20ms	SpaCy sentence tokenizer
BERT tokenization	30ms	HuggingFace fast tokenizer
Model inference	400ms	BERT-Base on T4 GPU, 512 tokens
Sliding window overhead	200ms	For notes > 512 tokens (2-3 windows)
Entity consolidation	50ms	Merge cross-window entities
ICD code lookup	80ms	SNOMED-to-ICD mapping, in-memory
Negation resolution	30ms	Context-based negation classification
API response serialization	20ms	JSON encoding
Total	**980ms**	Well within the 3-second budget

The 2-second margin accommodates network latency, queueing delays, and occasional GC pauses.

Monitoring

Application Metrics (Prometheus + Grafana):

Metric	Alert Threshold	Rationale
Inference latency (p95)	> 500ms	Approaching the latency budget
Inference latency (p99)	> 1500ms	Severe performance degradation
Error rate (5xx)	> 1%	Service reliability
GPU utilization	> 85% sustained 10min	Need to scale up
GPU memory usage	> 90%	Memory leak or batch size issue
Prediction confidence (mean)	< 0.70	Model may be encountering OOD data
Entity count per note (mean)	< 2 or > 50	Anomalous notes or model drift
Request throughput	< 10 req/min for 30min	Ingestion pipeline may be down

ML-Specific Metrics (logged per prediction):

- Per-entity confidence scores (stored for post-hoc analysis)
- Token-level entropy (high entropy = uncertain prediction)
- Proportion of O-label predictions per note (sudden increase = model degradation)
- Entity type distribution per note type (should remain stable)

Dashboard Panels: 1. Real-time: throughput, latency percentiles, error rate, GPU utilization 2. Daily: entity type distribution, confidence score distribution, notes processed 3. Weekly: model accuracy (computed against human-reviewed labels), drift metrics

Model Drift Detection

Input Drift (Data Distribution Shift): - Track the vocabulary distribution of incoming notes using a sliding window of 10,000 notes - Compute KL divergence between the current window's unigram distribution and the training set distribution - Alert when KL divergence exceeds 0.15 (calibrated on historical data) - Common causes: new EHR template, new department onboarded, seasonal disease patterns

Prediction Drift (Output Distribution Shift): - Monitor the distribution of predicted entity types weekly - Use a chi-squared test against the expected distribution from the training set - Alert when p-value < 0.01 - Common causes: model degradation, upstream preprocessing changes, new clinical terminology

Label Drift (Ground Truth Shift): - Regularly sample 200 predictions per week for human review by certified coders - Compute entity-level F1 on the sampled set - Alert when F1 drops below 0.83 (5 points below the target of 0.88) - Trigger model retraining when F1 drops below 0.80 on two consecutive weekly samples

Model Versioning

Version Format: `bert-clinical-ner-v{major}.{minor}.{patch}`

- **Major:** Architecture change (e.g., switch from BERT-Base to ClinicalBERT)
- **Minor:** Retraining on new data or hyperparameter changes
- **Patch:** Bug fix, preprocessing change, or post-processing update

Storage: - All model artifacts stored in S3 with versioned paths: `s3://medscribe-models/ner/{version}/` - Each version includes: model weights, tokenizer config, training config, evaluation report, training data hash - Minimum 5 most recent versions retained for rollback

Rollback Strategy: - Canary deployment: new model serves 5% of traffic for 24 hours - If entity-level F1 (measured against human-reviewed samples) is >= current production model, expand to 100% - If F1 is lower, automatic rollback to the previous version - Rollback is a config change (update the S3 path in the serving config) -- no redeployment needed

A/B Testing

Framework: Custom A/B testing built on top of the inference service. Each incoming note is assigned to a variant based on a hash of the note_id.

Statistical Requirements: - Minimum sample size: 1,000 notes per variant (power analysis: 80% power to detect a 2-point F1 difference at alpha = 0.05) - Test duration: minimum 7 days (to capture day-of-week effects) - Primary metric: entity-level F1 (measured against human coder labels for a subset) - Guardrail metrics: latency p95, error rate, confidence score distribution

Decision Criteria: - Proceed with new model if: (1) F1 improvement is statistically significant at $p < 0.05$, AND (2) all guardrail metrics are within 10% of the control - If guardrail metrics regress significantly, reject even if F1 improves - All A/B test results are logged in a decision registry with the rationale for acceptance or rejection

CI/CD for ML

Training Pipeline (triggered manually or by drift alert):

1. **Data validation:** Check for schema consistency, missing fields, label distribution anomalies. Fail the pipeline if > 5% of samples fail validation.
2. **Training:** Fine-tune from the previous best model (not from scratch) using the combined historical + new data. Log all hyperparameters and metrics to MLflow.
3. **Evaluation gate:** Automated evaluation on a held-out test set. The model must meet ALL of the following:
4. Entity-level F1 >= 0.88
5. Per-entity-type F1 >= 0.85
6. Inference latency p95 < 500ms (benchmarked on T4 GPU)

7. Model size < 4 GB
8. **Bias audit:** Run the demographic bias test suite. Flag any new biases introduced by the updated model.
9. **Artifact packaging:** Package model weights, tokenizer, configs into a versioned artifact.
10. **Canary deployment:** Deploy to staging, then 5% canary, then full rollout (as described in Model Versioning).

Code Pipeline (standard CI/CD): - Pre-commit: linting (ruff), type checking (mypy), unit tests - PR merge: integration tests with a small model checkpoint (DistilBERT for speed) - Staging deployment: full model with end-to-end tests against synthetic clinical notes - Production deployment: blue-green deployment with automated smoke tests

Cost Analysis

Training Costs:

Item	Cost	Frequency
4x A100 GPU (p4d.24xlarge) for 48 hours	1,470	Per training run
Data annotation (200 notes/month for drift monitoring)	4,000/month	Monthly
MLflow/experiment tracking infrastructure	200/month	Monthly
S3 storage for model artifacts and training data	50/month	Monthly
Monthly training cost	**5,720**	

Inference Costs (per hospital):

Item	Cost	Notes
2x g4dn.xlarge (baseline)	1,052/month	On-demand pricing, 24/7
Peak scaling (up to 8 instances)	1,580/month	Estimated 30% peak time
Load balancer	18/month	Application Load Balancer
CloudWatch/monitoring	50/month	Custom metrics and dashboards
Per-hospital monthly cost	**2,700**	
Total inference (12 hospitals)	**32,400/month**	

Total Monthly Cost: 38,120 dollars (training + inference for 12 hospitals)

This is well within MedScribe's 15,000 dollars/month GPU budget for training, with inference costs passed through to hospital clients as part of the SaaS pricing. The per-hospital cost of 2,700 dollars/month is a fraction of the 11.6 million dollar annual cost of coding errors that the system is designed to reduce.