Case Study: Teaching an Insurance Claims Engine to Reason Step by Step

ClearClaim AI -- Building a Verifiable Reasoning Model for Multi-Step Claims Adjudication

---

Section 1: Industry Context and Business Problem

Industry: Insurance Technology (InsurTech) -- Property and Casualty Claims

The property and casualty (P&C) insurance industry processes over \$700 billion in claims annually in the United States alone. Claims adjudication -- the process of determining what a policyholder is owed -- is one of the most labor-intensive and error-prone functions in the industry. A single commercial property claim can involve deductible application, depreciation schedules, sub-limit enforcement, co-insurance calculations, and coordination of benefits across multiple policies. Each step depends on the previous one, and a mistake in any step cascades through to the final payout.

Company Profile: ClearClaim AI

ClearClaim AI is a Series B InsurTech startup headquartered in Chicago, Illinois. Founded in 2021 by two former actuaries and a machine learning engineer, the company builds AI-powered claims adjudication software for mid-market P&C insurers.

- **Team size:** 85 employees (22 engineers, 8 ML engineers, 12 domain experts)
- **Funding:** \$42M raised (Series A: \$14M in 2022, Series B: \$28M in 2024)
- **Product:** ClaimFlow -- an AI-assisted claims processing platform used by 18 regional insurance carriers
- **Processing volume:** 1.2 million claims per year across all carrier clients
- **Revenue model:** Per-claim SaaS pricing (\$4.50 per processed claim)

Business Challenge

ClearClaim's current system uses a fine-tuned LLM (based on Llama-3-8B) to read adjuster reports and compute payouts. For simple single-step claims (e.g., a windshield replacement with a fixed deductible), the system performs well, achieving 94% accuracy. However, for multi-step claims -- which constitute 38% of all claims and 67% of total payout dollars -- accuracy drops to 69%.

The failure modes are specific and costly:

1. **Arithmetic cascading errors.** The model computes depreciation correctly but then forgets to subtract the deductible, or applies the deductible before depreciation instead of after.
2. **Sub-limit violations.** The model ignores policy sub-limits (e.g., a \$5,000 cap on jewelry claims within a homeowner's policy) and approves the full replacement cost.
3. **Co-insurance miscalculations.** For commercial policies with 80/20 co-insurance clauses, the model frequently confuses the insurer's share with the policyholder's share.
4. **No audit trail.** The model outputs a single number with no explanation, making it impossible for adjusters to verify the logic or for regulators to audit the decision.

Why It Matters

- **Financial impact:** The 31% error rate on multi-step claims costs ClearClaim's carrier clients an estimated \$2.8M per year in overpayments and \$1.1M in underpayment dispute resolution.
- **Regulatory pressure:** State insurance commissioners are increasingly requiring explainable AI decisions. Three of ClearClaim's carrier clients have received regulatory inquiries about their AI adjudication process. Without auditable reasoning traces, these carriers risk fines or loss of operating licenses.
- **Competitive threat:** Two competitors (SettleAI and PolicyMind) have announced "explainable claims reasoning" features. ClearClaim risks losing its mid-market positioning if it cannot match this capability within 6 months.
- **Customer churn risk:** Two carrier clients (representing \$1.8M ARR) have flagged the multi-step accuracy issue as a renewal blocker.

Constraints

- **Compute budget:** 8x NVIDIA A100 GPUs available for training (cloud). Inference must run on 2x A10G GPUs.
- **Latency:** Claims processing must complete within 15 seconds per claim (including reasoning trace generation).
- **Compliance:** All AI decisions must produce an auditable reasoning trace that references specific policy terms. SOC 2 Type II certified environment.
- **Data:** 240,000 historical claims with adjuster-verified payouts. An additional 50,000 claims have detailed step-by-step adjuster worksheets (showing intermediate calculations).
- **Model size:** Final production model must be 3B parameters or smaller (to fit within inference budget).
- **Team expertise:** The ML team has experience with SFT and LoRA fine-tuning but has not deployed RL-trained models before.

Section 2: Technical Problem Formulation

Problem Type: Conditional Text Generation with Verifiable Numerical Outputs

This is a conditional generation problem: given a claim description and policy terms, generate a step-by-step reasoning trace that arrives at a correct payout amount. We choose generation over classification or regression because:

- **Classification** (bucketing payouts into ranges) loses the precision needed for exact dollar amounts.
- **Regression** (predicting a single number) provides no audit trail and does not decompose the multi-step logic.
- **Generation with verification** gives us both: a human-readable reasoning trace AND a verifiable numerical answer that can be checked against ground truth.

This framing directly maps to the article's core concept: we can train a reasoning model using verifiable rewards, because the correct payout for each claim is known from historical adjuster worksheets.

Input Specification

Each input consists of two components:

1. **Claim description** (natural language, 200-800 tokens): The adjuster's report describing the damage, affected items, and circumstances.
2. **Policy terms** (structured text, 100-400 tokens): Relevant policy provisions including deductible amount, coverage limits, sub-limits, depreciation schedule, and co-insurance clause.

The input is formatted as a single prompt:

```
<claim>
[Adjuster report text]
</claim>

<policy>
Deductible: $1,000
Coverage limit: $250,000
Depreciation: 10% per year
Sub-limits: Jewelry $5,000, Electronics $10,000
Co-insurance: 80% (insurer) / 20% (policyholder)
</policy>

Calculate the claim payout step by step.
```

Output Specification

The model produces a reasoning trace followed by a final answer:

```
<think>
Step 1: Identify claimed items and amounts.
```

```
  - Roof damage: $18,500 replacement cost
  - Water damage (interior): $6,200 repair cost
  - Personal electronics: $3,800 replacement cost

Step 2: Apply depreciation (roof is 8 years old, 10% per year).
  - Roof depreciation: $18,500 x 0.80 = $14,800 (actual cash value)
  - Water damage: no depreciation (repair cost)
  - Electronics: $3,800 (replacement cost, no depreciation for electronics under 2 years)

Step 3: Check sub-limits.
  - Electronics $3,800 < $10,000 sub-limit. Within limit.

Step 4: Sum pre-deductible amounts.
  - Total: $14,800 + $6,200 + $3,800 = $24,800

Step 5: Apply deductible.
  - $24,800 - $1,000 = $23,800

Step 6: Apply co-insurance.
  - Insurer pays 80%: $23,800 x 0.80 = $19,040

Step 7: Verify against coverage limit.
  - $19,040 < $250,000 limit. Within limit.
</think>

The payout is $19,040.
```

We choose this structured output format because: - The `<think>` tags cleanly separate reasoning from the final answer, enabling automated extraction and verification. - Each numbered step corresponds to a specific adjudication rule, making the trace auditable. - The final answer is a single dollar amount that can be programmatically compared to ground truth.

Mathematical Foundation

The training pipeline uses three mathematical frameworks, each building on the previous one.

Framework 1: Maximum Likelihood Estimation (SFT Stage)

During supervised fine-tuning, we train the model to reproduce expert reasoning traces. The objective is standard next-token prediction:

$$\mathcal{L}_{\text{SFT}} = -\sum_{t=1}^{T} \log p_\theta(y_t \mid y_{<t}, x)$$

where x is the input (claim + policy), y_t is the token at position t in the target completion, and T is the total sequence length. This is maximum likelihood estimation: we are finding model parameters θ that maximize the probability of the observed expert traces.

The key limitation of MLE is that it treats all tokens equally. A model that perfectly predicts the word "the" but gets the number "\$19,040" wrong receives nearly the same loss as one that gets the number right but misspells "the". For claims adjudication, the numerical tokens are far more important than the natural language tokens. SFT alone cannot encode this priority.

Framework 2: Policy Gradient Optimization (GRPO Stage)

To teach the model that correctness matters more than fluency, we use reinforcement learning. The reward signal is binary and verifiable:

$$r(y, y^*) = \begin{cases} 1 & \text{if extract\_amount}(y) = y^* \\ 0 & \text{otherwise} \end{cases}$$

where y is the model's completion and y^* is the ground-truth payout amount.

GRPO computes advantages relative to a group of G sampled completions:

$$\hat{A}_i = \frac{r_i - \bar{r}}{\sigma_r + \varepsilon}$$

where $\bar{r}$ and σ_r are the mean and standard deviation of rewards within the group. This eliminates the need for a learned value function (critic), reducing memory requirements by approximately 40% compared to PPO.

The policy is updated using a clipped surrogate objective:

$$\mathcal{L}_{\text{GRPO}} = -\frac{1}{G}\sum_{i=1}^{G} \min\left(\rho_i \hat{A}_i,\ \text{clip}(\rho_i, 1-\epsilon, 1+\epsilon)\hat{A}_i\right)$$

where $\rho_i = \frac{\pi_\theta(y_i|x)}{\pi_{\text{old}}(y_i|x)}$ is the importance sampling ratio. The clipping prevents catastrophic policy updates.

Why clipping matters for claims: Without clipping, if the model discovers one correct reasoning path, it might increase its probability so aggressively that it never explores alternative paths. In claims adjudication, different claim types require different reasoning strategies (depreciation-first vs. sub-limit-first). Clipping preserves exploration.

Framework 3: KL-Penalized Rewards (Stability)

To prevent the policy from degenerating (e.g., outputting only the number without reasoning), we add a KL divergence penalty:

$$R_{\text{total}} = r(y, y^*) - \beta \cdot D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$$

The KL term measures how far the current policy has drifted from the reference (post-SFT) policy. If β is too small, the model may find reward-hacking shortcuts (e.g., memorizing common payout amounts). If β is too large, the model cannot improve beyond the SFT baseline. For claims adjudication, we set $\beta = 0.05$ based on the observation that the SFT model already produces well-formatted traces -- we want to improve accuracy without destroying the formatting.

Loss Function

The final training loss combines three terms:

$$\mathcal{L} = \mathcal{L}_{\text{GRPO}} + \alpha \cdot \mathcal{L}_{\text{format}} + \beta \cdot D_{\text{KL}}$$

- $\mathcal{L}_{\text{GRPO}}$: Drives the model toward correct payouts. Removing this term means the model has no incentive to be accurate -- it would produce plausible but incorrect traces.
- $\mathcal{L}_{\text{format}}$: A small auxiliary loss that rewards traces containing the correct structure (`<think>` tags, numbered steps). Removing this term leads to occasional format violations where the model omits the reasoning trace entirely. Weight $\alpha = 0.1$.
- D_{KL} : Prevents policy collapse. Removing this term causes the model to degenerate within 200 training steps, producing repetitive or nonsensical traces that happen to output common payout amounts.

Evaluation Metrics

Metric	Description	Target	Baseline
Exact Match Accuracy	Payout matches ground truth exactly (dollar-for-dollar)	> 85%	69%
Step Correctness	Each intermediate step is arithmetically correct	> 90%	N/A (no steps)
Format Compliance	Output contains valid `<think>` block with numbered steps	> 98%	N/A
Inference Latency	End-to-end time per claim (including reasoning)	< 15s	2s
Audit Pass Rate	Regulatory auditor accepts the trace as sufficient documentation	> 95%	0%

Baseline

Without the reasoning model, ClearClaim uses a fine-tuned Llama-3-8B that directly predicts the payout amount as a single number. This achieves:

- 94% accuracy on single-step claims
- 69% accuracy on multi-step claims
- 0% audit compliance (no reasoning trace)

The baseline is insufficient because (a) it fails on the most valuable claims, (b) it provides no regulatory compliance, and (c) errors are undetectable until a human reviewer catches them.

Why This Concept

The reasoning model approach from the article is the right solution for three specific reasons:

1. **Verifiable rewards are natural.** Insurance claims have ground-truth payouts verified by human adjusters. We do not need a reward model -- we simply check if the model's answer matches the adjuster's verified amount. This eliminates a major source of complexity and instability in RLHF pipelines.

2. **Chain-of-thought provides auditability.** The `<think>` trace is not just a training artifact -- it directly satisfies the regulatory requirement for explainable AI decisions. Each step in the trace corresponds to a specific policy provision that an auditor can verify.

3. **Distillation enables deployment.** The article's three-stage pipeline (SFT, RL, distillation) lets us train a large model for reasoning quality and then distill to a 3B model that fits within the inference budget. This solves the latency constraint without sacrificing accuracy.

Technical Constraints

- **Model size:** Base model Qwen2.5-3B (fits on 2x A10G for inference)
- **Training compute:** 8x A100 for up to 72 hours (GRPO training is compute-intensive due to multiple completions per prompt)
- **Inference latency:** < 15 seconds per claim (reasoning traces average 200-400 tokens)
- **Data volume:** 50,000 claims with step-by-step worksheets (for SFT), 240,000 claims with verified payouts (for GRPO)
- **Group size:** $G = 8$ completions per prompt during GRPO training

---

Section 3: Implementation Notebook Structure

3.1 Data Acquisition and Preparation

Context: We will use two datasets: (1) the GSM8K math reasoning dataset for initial SFT warm-up (the model first learns to reason about simpler math problems), and (2) a synthetic insurance claims dataset that mirrors real multi-step payout calculations.

What the student will do: Load GSM8K, inspect its structure, then build a synthetic claims dataset with configurable complexity levels.

Setup code:

```
import torch
import torch.nn.functional as F
import json
import re
import random
import numpy as np
from dataclasses import dataclass
from typing import List, Tuple, Optional
from datasets import load_dataset

# Set random seeds for reproducibility
random.seed(42)
np.random.seed(42)
torch.manual_seed(42)

# Load GSM8K for initial warm-up training
gsm8k = load_dataset("openai/gsm8k", "main")
print(f"GSM8K train size: {len(gsm8k['train'])}")
print(f"GSM8K test size: {len(gsm8k['test'])}")

# Inspect a sample
sample = gsm8k['train'][0]
```

```
print(f"\nQuestion: {sample['question'][:200]}...")
print(f"\nAnswer: {sample['answer'][:200]}...")
```

TODO 1: Build the Synthetic Claims Dataset Generator

```
@dataclass
class InsuranceClaim:
    """Represents a single insurance claim with policy terms and ground truth."""
    claim_description: str
    policy_terms: dict
    ground_truth_payout: float
    reasoning_steps: List[str]
    difficulty: str  # 'single_step', 'multi_step', 'complex'

def generate_claims_dataset(n_claims: int = 5000, difficulty_mix: dict = None) -> List[InsuranceClaim]:
    """
    Generate a synthetic insurance claims dataset with verifiable payouts.

    Each claim includes:
    - A natural language claim description
    - Structured policy terms (deductible, limits, depreciation, sub-limits, co-insurance)
    - The correct payout amount (computed deterministically from the terms)
    - Step-by-step reasoning showing how to arrive at the payout
    - Difficulty level

    The difficulty mix controls the distribution:
    - 'single_step': Only deductible subtraction (e.g., damage - deductible)
    - 'multi_step': Deductible + depreciation + sub-limits (3-4 steps)
    - 'complex': All of the above + co-insurance + coverage limit checks (5-7 steps)

    Args:
        n_claims: Total number of claims to generate
        difficulty_mix: Dict like {'single_step': 0.3, 'multi_step': 0.4, 'complex': 0.3}

    Returns:
        List of InsuranceClaim objects with verified ground truth payouts

    Hints:
        1. Start by defining ranges for each policy parameter:
           - Deductibles: [$500, $1000, $2500, $5000]
           - Coverage limits: [$50K, $100K, $250K, $500K]
           - Depreciation rates: [5%, 10%, 15%] per year, item age 1-15 years
           - Sub-limits: electronics $10K, jewelry $5K, art $25K
           - Co-insurance: 80/20, 70/30, 90/10
        2. For each claim, randomly sample parameters, then compute the payout
           step by step (this becomes both the ground truth AND the reasoning trace)
        3. The payout computation order matters:
           a. Start with replacement cost for each item
           b. Apply depreciation to get actual cash value (ACV)
           c. Check each item against sub-limits (cap if needed)
           d. Sum all items
           e. Subtract deductible
           f. Apply co-insurance percentage
           g. Check against coverage limit (cap if needed)
        4. Generate the natural language claim description from the parameters
    """
    # TODO: Implement this function
    pass


# Verification: check that generated payouts are deterministically correct
def verify_dataset(claims: List[InsuranceClaim]) -> dict:
    """Recompute payouts from policy terms and verify they match ground truth."""
    correct = 0
    for claim in claims:
        recomputed = recompute_payout(claim.policy_terms)
        if abs(recomputed - claim.ground_truth_payout) < 0.01:
            correct += 1
    accuracy = correct / len(claims)
    print(f"Dataset verification: {correct}/{len(claims)} ({accuracy:.1%}) payouts verified")
    assert accuracy == 1.0, "Dataset has computation errors!"
    return {"verified": correct, "total": len(claims)}
```

Thought questions: - Why is it critical that the synthetic dataset is deterministically verifiable? What would happen during GRPO training if the ground-truth payouts were sometimes wrong? - How does the difficulty mix affect the training curriculum? Should we train on easy claims first or mix all difficulties from the start?

3.2 Exploratory Data Analysis

Context: Before training, we need to understand the distribution of our data -- both GSM8K and the synthetic claims dataset.

```
import matplotlib.pyplot as plt

# Analyze GSM8K answer distribution
gsm8k_answers = []
for ex in gsm8k['train']:
    match = re.search(r'#### (\-?[\d,]+)', ex['answer'])
    if match:
        gsm8k_answers.append(float(match.group(1).replace(',', '')))

plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.hist(gsm8k_answers, bins=50, edgecolor='black', alpha=0.7)
plt.xlabel('Answer Value')
plt.ylabel('Count')
plt.title('GSM8K Answer Distribution')
plt.subplot(1, 2, 2)
plt.hist([len(ex['answer'].split()) for ex in gsm8k['train']], bins=30, edgecolor='black', alpha=0.7)
plt.xlabel('Reasoning Length (words)')
plt.ylabel('Count')
plt.title('GSM8K Reasoning Trace Length')
plt.tight_layout()
plt.show()
```

TODO 2: Analyze the Claims Dataset

```
def analyze_claims_dataset(claims: List[InsuranceClaim]):
    """
    Produce exploratory data analysis plots for the insurance claims dataset.

    Generate the following visualizations:
    1. Payout distribution by difficulty level (three overlapping histograms)
    2. Number of reasoning steps vs. payout amount (scatter plot)
    3. Distribution of policy parameters (deductibles, limits, depreciation rates)
    4. Correlation heatmap between policy parameters and payout amount

    Also compute and print:
    - Mean and median payout by difficulty level
    - Average number of reasoning steps by difficulty level
    - The fraction of claims where sub-limits affect the payout
    - The fraction of claims where the coverage limit caps the payout

    Hints:
        1. Use matplotlib with 2x2 subplots for the four visualizations
        2. For the correlation heatmap, extract numeric fields from policy_terms
        3. Pay attention to the relationship between number of steps and accuracy --
           this tells us how difficult multi-step reasoning is for the model
    """
    # TODO: Implement this function
    pass
```

Thought questions: - What does the relationship between reasoning length and accuracy tell you about why SFT alone is insufficient? - If the payout distribution is heavily right-skewed (a

few very large claims), how might this affect GRPO training? Would you apply any normalization to the rewards?

3.3 Baseline: Direct Payout Prediction

Context: Before building a reasoning model, we implement the current baseline -- a model that directly predicts the payout amount without intermediate steps.

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Use a small model for the baseline experiment
MODEL_NAME = "Qwen/Qwen2.5-0.5B"
tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForCausalLM.from_pretrained(MODEL_NAME, torch_dtype=torch.float32)

# If pad token is not set, use eos token
if tokenizer.pad_token is None:
    tokenizer.pad_token = tokenizer.eos_token

def format_direct_prompt(claim: InsuranceClaim) -> str:
    """Format a claim as a direct (no reasoning) prompt."""
    return f"""<claim>
{claim.claim_description}
</claim>

<policy>
{json.dumps(claim.policy_terms, indent=2)}
</policy>

What is the payout amount? Answer with just the dollar amount."""

print("Example direct prompt:")
print(format_direct_prompt(claims[0]))
```

TODO 3: Implement and Evaluate the Direct Baseline

```
def evaluate_direct_baseline(model, tokenizer, claims: List[InsuranceClaim], n_eval: int = 200) -> dict:
    """
    Evaluate the direct (no-reasoning) payout prediction baseline.

    For each claim:
    1. Format the claim as a direct prompt (no reasoning requested)
    2. Generate the model's completion (max 50 tokens)
    3. Extract the predicted payout amount from the completion
    4. Compare against ground truth

    Args:
        model: The language model
        tokenizer: The tokenizer
        claims: List of InsuranceClaim objects
        n_eval: Number of claims to evaluate

    Returns:
        Dict with:
        - 'exact_match': fraction of exact matches (within 1 dollar)
        - 'within_10pct': fraction within 10% of ground truth
        - 'mean_absolute_error': average dollar error
        - 'accuracy_by_difficulty': dict mapping difficulty to exact_match rate

    Hints:
        1. Use model.generate() with temperature=0 for deterministic evaluation
        2. Extract dollar amounts using regex: r'\\$?([\\d,]+\\.?\\d*)'
        3. Handle cases where the model outputs text instead of a number
        4. Track accuracy separately for each difficulty level to confirm that
           multi-step claims are harder
    """
    # TODO: Implement this function
```

```
    pass

# Run baseline evaluation
# baseline_results = evaluate_direct_baseline(model, tokenizer, test_claims)
# print(f"Baseline exact match: {baseline_results['exact_match']:.1%}")
# print(f"Baseline by difficulty: {baseline_results['accuracy_by_difficulty']}")
```

Thought questions: - Why is the direct baseline likely to perform worse on multi-step claims? Think about what happens when the model must do multiple arithmetic operations "in its head" without writing them down. - If you increased the model size from 0.5B to 7B, would the direct baseline improve significantly on multi-step claims? Why or why not?

3.4 Model Design: Chain-of-Thought Reasoning Architecture

Context: Our reasoning model uses the same base architecture (Qwen2.5-0.5B) but is trained to produce step-by-step reasoning traces before the final answer. The architecture does not change -- only the training procedure and expected output format change.

```
def format_reasoning_prompt(claim: InsuranceClaim) -> str:
    """Format a claim as a reasoning prompt with <think> tags."""
    return f"""<claim>
{claim.claim_description}
</claim>

<policy>
{json.dumps(claim.policy_terms, indent=2)}
</policy>

Calculate the claim payout step by step."""


def format_reasoning_target(claim: InsuranceClaim) -> str:
    """Format the target completion with reasoning trace."""
    steps = "\n".join(f"Step {i+1}: {step}" for i, step in enumerate(claim.reasoning_steps))
    return f"""<think>
{steps}
</think>

The payout is ${claim.ground_truth_payout:,.2f}."""


# Inspect the format
sample_claim = claims[0]
print("=== PROMPT ===")
print(format_reasoning_prompt(sample_claim))
print("\n=== TARGET ===")
print(format_reasoning_target(sample_claim))
```

TODO 4: Implement the SFT Training Loop

```
def sft_training_step(model, tokenizer, prompt: str, target: str, optimizer) -> float:
    """
    One step of supervised fine-tuning on a chain-of-thought example.

    This implements the SFT loss from the article:
    L_SFT = -sum_{t=1}^{T} log p_theta(y_t | y_{<t}, x)

    Args:
        model: The language model
        tokenizer: The tokenizer
        prompt: The input prompt (claim + policy)
        target: The target completion (reasoning trace + answer)
        optimizer: The optimizer
```

```
    Returns:
        The loss value (float)

    Hints:
        1. Concatenate prompt + target into one sequence
        2. Tokenize the full sequence
        3. Create a labels tensor where prompt tokens are set to -100
           (so the loss is only computed on the target tokens)
        4. Forward pass through the model
        5. Compute cross-entropy loss only on target tokens
        6. Backpropagate and update
    """
    # TODO: Implement this function
    pass


def run_sft_training(model, tokenizer, claims: List[InsuranceClaim],
                     n_epochs: int = 2, lr: float = 2e-5, batch_size: int = 4) -> List[float]:
    """
    Run the full SFT training loop.

    Args:
        model: The language model
        tokenizer: The tokenizer
        claims: Training claims with reasoning traces
        n_epochs: Number of training epochs
        lr: Learning rate
        batch_size: Not used for simplicity (we do single-example updates)

    Returns:
        List of loss values for plotting

    Hints:
        1. Use AdamW optimizer with the specified learning rate
        2. Shuffle claims at the start of each epoch
        3. Log loss every 100 steps
        4. Save a checkpoint at the end of each epoch
    """
    # TODO: Implement this function
    pass
```

Thought questions: - Why do we mask the prompt tokens (set labels to -100) instead of training on the full sequence? What would happen if we included the prompt tokens in the loss? - After SFT, the model can produce well-formatted reasoning traces. But can it produce CORRECT reasoning? What fundamental limitation of maximum likelihood training causes this gap?

3.5 Training Strategy: GRPO with Verifiable Rewards

Context: This is the core of the case study. We implement GRPO to train the model to produce correct payouts, using the ground-truth payout as a verifiable reward signal.

```
# Configuration for GRPO training
GRPO_CONFIG = {
    'group_size': 8,        # G: completions per prompt
    'epsilon': 0.2,         # Clipping parameter
    'beta': 0.05,           # KL penalty weight
    'lr': 1e-6,             # Learning rate (much smaller than SFT)
    'max_new_tokens': 512,  # Max reasoning trace length
    'temperature': 0.7,     # Sampling temperature
}

def extract_payout(completion: str) -> Optional[float]:
    """Extract the dollar payout amount from a model completion."""
    # Look for patterns like "The payout is X,XXX.XX" or "#### X,XXX"
    patterns = [
```

```
        r'payout\s+is\s+\$?([\d,]+\.?\d*)',
        r'####\s+\$?([\d,]+\.?\d*)',
        r'total.*?\$?([\d,]+\.?\d*)\s*$',
    ]
    for pattern in patterns:
        match = re.search(pattern, completion, re.IGNORECASE | re.MULTILINE)
        if match:
            return float(match.group(1).replace(',', ''))
    return None


def compute_reward(completion: str, ground_truth: float) -> float:
    """Binary reward: 1 if extracted payout matches ground truth, 0 otherwise."""
    predicted = extract_payout(completion)
    if predicted is None:
        return 0.0
    return 1.0 if abs(predicted - ground_truth) < 0.01 else 0.0
```

TODO 5: Implement GRPO Training

```
def compute_grpo_loss(model, ref_model, tokenizer, prompt: str,
                      completions: List[str], advantages: torch.Tensor,
                      epsilon: float = 0.2, beta: float = 0.05) -> torch.Tensor:
    """
    Compute the GRPO loss for a group of completions.

    This implements:
    L_GRPO = -(1/G) * sum_i min(rho_i * A_i, clip(rho_i, 1-eps, 1+eps) * A_i)
             + beta * D_KL(pi_theta || pi_ref)

    Args:
        model: Current policy model
        ref_model: Frozen reference model (post-SFT checkpoint)
        tokenizer: The tokenizer
        prompt: The input prompt
        completions: List of G completions from the old policy
        advantages: Group-relative advantages, shape (G,)
        epsilon: Clipping parameter
        beta: KL penalty weight

    Returns:
        Scalar loss tensor

    Hints:
        1. For each completion, compute log p(completion | prompt) under both
           the current model and the reference model
        2. The probability ratio is rho_i = exp(log_p_current - log_p_old)
           Note: log_p_old is the probability under the model that GENERATED
           the completions, which is the current model from the previous step.
           For simplicity, you can use the reference model as the old policy.
        3. Compute the clipped and unclipped objectives
        4. Take the element-wise minimum
        5. Add the KL penalty: beta * mean(log_p_current - log_p_ref)
           (this is a sample-based estimate of the KL divergence)
        6. Return the negative mean (we minimize the loss, which maximizes the objective)
    """
    # TODO: Implement this function
    pass


def grpo_training_step(model, ref_model, tokenizer, claim: InsuranceClaim,
                       optimizer, config: dict) -> dict:
    """
    One complete GRPO training step for a single claim.

    Steps:
    1. Format the claim as a reasoning prompt
    2. Generate G completions from the current model
    3. Compute binary rewards for each completion
    4. Compute group-relative advantages
    5. Compute GRPO loss
    6. Backpropagate and update

    Args:
```

```
        model: Current policy model
        ref_model: Frozen reference model
        tokenizer: The tokenizer
        claim: The insurance claim to train on
        optimizer: The optimizer
        config: GRPO hyperparameters

    Returns:
        Dict with 'loss', 'mean_reward', 'n_correct', 'advantages'

    Hints:
        1. Use model.generate() with do_sample=True and the configured temperature
        2. After computing rewards, check if all rewards are the same (all correct
           or all incorrect). If so, skip the update -- the advantages would be
           all zeros (no learning signal).
        3. Compute advantages as (rewards - mean) / (std + 1e-8)
        4. Detach the generated completions from the computation graph before
           computing the GRPO loss
    """
    # TODO: Implement this function
    pass
```

Thought questions: - What happens when all G completions get the same reward (all correct or all incorrect)? Why must we skip the update in this case? - The KL penalty weight $\beta = 0.05$ is relatively small. What would happen if we set $\beta = 1.0$? What about $\beta = 0.001$? - Why do we use temperature=0.7 for GRPO sampling instead of temperature=0 (greedy)? What role does stochasticity play in the learning process?

3.6 Evaluation

Context: We evaluate the reasoning model on a held-out test set, comparing against the direct prediction baseline.

```
def evaluate_reasoning_model(model, tokenizer, test_claims: List[InsuranceClaim],
                             n_eval: int = 200) -> dict:
    """
    Evaluate the reasoning model on held-out claims.

    For each claim:
    1. Generate a reasoning trace with greedy decoding (temperature=0)
    2. Extract the predicted payout
    3. Verify each intermediate step is arithmetically correct
    4. Check format compliance (proper <think> tags, numbered steps)

    Returns:
        Dict with all evaluation metrics from Section 2
    """
    results = {
        'exact_match': 0,
        'step_correct': 0,
        'format_compliant': 0,
        'total': 0,
        'by_difficulty': {},
        'latencies': [],
    }

    model.eval()
    with torch.no_grad():
        for claim in test_claims[:n_eval]:
            prompt = format_reasoning_prompt(claim)
            # TODO: Generate, extract, evaluate
            pass

    return results
```

TODO 6: Implement Comprehensive Evaluation

```
def verify_reasoning_steps(completion: str, policy_terms: dict) -> dict:
    """
    Verify that each step in the reasoning trace is arithmetically correct.

    Parse the <think> block, extract each Step, identify the arithmetic
    operation performed, and check if the result is correct.

    Args:
        completion: The model's full completion (with <think> block)
        policy_terms: The policy terms for reference

    Returns:
        Dict with:
        - 'n_steps': number of steps found
        - 'n_correct_steps': number of arithmetically correct steps
        - 'step_details': list of dicts with step text, operation, expected, actual
        - 'format_valid': whether the <think> block is properly formatted

    Hints:
        1. Extract text between <think> and </think> tags
        2. Split on "Step N:" patterns
        3. For each step, look for arithmetic expressions (A * B = C, A - B = C, etc.)
        4. Evaluate the expression and compare to the stated result
        5. Track which steps have errors and what kind of errors
    """
    # TODO: Implement this function
    pass


def plot_evaluation_comparison(baseline_results: dict, reasoning_results: dict):
    """
    Create a side-by-side comparison plot of baseline vs reasoning model.

    Generate:
    1. Grouped bar chart: accuracy by difficulty level (baseline vs reasoning)
    2. Scatter plot: payout error vs claim complexity
    3. Histogram: reasoning trace lengths for correct vs incorrect predictions
    4. Line plot: accuracy vs number of reasoning steps

    Hints:
        1. Use matplotlib with a 2x2 subplot layout
        2. Use consistent colors: blue for baseline, orange for reasoning model
        3. Add error bars if you have multiple evaluation runs
    """
    # TODO: Implement this function
    pass
```

Thought questions: - If the model achieves 85% exact match accuracy but only 78% step correctness, what does that imply? Can a model arrive at the right answer through incorrect reasoning? - How would you modify the evaluation to catch "right answer, wrong reasoning" cases? Why does this matter for regulatory compliance?

3.7 Error Analysis

Context: Understanding failure modes is critical for improving the model and building trust with insurance carriers.

TODO 7: Implement Error Analysis

```
def categorize_errors(model, tokenizer, test_claims: List[InsuranceClaim],
                      n_eval: int = 200) -> dict:
    """
    Systematically categorize errors from the reasoning model.
```

```
    Error categories:
    1. 'arithmetic_error': A step contains an incorrect arithmetic computation
    2. 'step_ordering_error': Steps are applied in the wrong order
       (e.g., deductible before depreciation)
    3. 'missing_step': A required step is omitted (e.g., sub-limit not checked)
    4. 'hallucinated_value': The model uses a number not present in the claim
       or policy terms
    5. 'format_error': The output is malformed (missing <think> tags, no answer)
    6. 'extraction_error': The answer is present but could not be parsed

    For each error:
    - Record the claim difficulty level
    - Record which step the error occurred in
    - Record the model's output and the expected output

    Args:
        model: The reasoning model
        tokenizer: The tokenizer
        test_claims: Held-out test claims

    Returns:
        Dict mapping error category to list of error instances

    Hints:
        1. Use verify_reasoning_steps() to identify arithmetic errors
        2. Define the expected step ordering: depreciation -> sub-limits -> sum ->
           deductible -> co-insurance -> coverage limit
        3. Check if all required policy terms are referenced in the trace
        4. Compare numbers in the trace against numbers in the claim/policy
    """
    # TODO: Implement this function
    pass
```

Thought questions: - Which error category do you expect to be most common? Why? - If the model consistently applies deductible before depreciation (wrong order), is this an SFT problem or a GRPO problem? How would you fix it? - How would you incorporate error analysis results back into the training pipeline? (Hint: think about targeted data augmentation or reward shaping.)

3.8 Scalability and Deployment Considerations

Context: ClearClaim needs to serve 1.2 million claims per year (approximately 3,300 per day, or 140 per hour). The model must fit within the inference budget (2x A10G GPUs) and meet the 15-second latency requirement.

```
import time

def benchmark_inference(model, tokenizer, test_claims: List[InsuranceClaim],
                        n_runs: int = 50) -> dict:
    """
    Benchmark inference latency for the reasoning model.

    Measure:
    - Time to first token (TTFT)
    - Tokens per second (TPS)
    - Total generation time per claim
    - Memory usage (peak GPU memory)

    Run multiple times to get stable statistics.
    """
    latencies = []
    token_counts = []

    model.eval()
```

```
    with torch.no_grad():
        for claim in test_claims[:n_runs]:
            prompt = format_reasoning_prompt(claim)
            inputs = tokenizer(prompt, return_tensors="pt")

            start = time.time()
            outputs = model.generate(
                inputs.input_ids,
                max_new_tokens=512,
                temperature=0.0,
                do_sample=False,
            )
            elapsed = time.time() - start

            n_tokens = outputs.shape[1] - inputs.input_ids.shape[1]
            latencies.append(elapsed)
            token_counts.append(n_tokens)

    return {
        'mean_latency_s': np.mean(latencies),
        'p95_latency_s': np.percentile(latencies, 95),
        'mean_tokens': np.mean(token_counts),
        'tokens_per_second': np.mean([t/l for t, l in zip(token_counts, latencies)]),
    }
```

TODO 8: Write an Inference Optimization Plan

```
def plan_inference_optimization(benchmark_results: dict, target_latency_s: float = 15.0) -> str:
    """
    Analyze benchmark results and produce a concrete optimization plan.

    If the current latency exceeds the target, recommend specific optimizations:
    1. KV-cache optimization (quantify expected speedup)
    2. Speculative decoding (explain how it works and expected speedup)
    3. Model quantization (INT8/INT4) (quantify memory and speed tradeoffs)
    4. Batching strategy for throughput (how many claims per batch)
    5. Early stopping (stop generation when answer token is detected)

    For each recommendation:
    - Estimate the expected latency reduction
    - Note any accuracy tradeoffs
    - Provide implementation complexity (low/medium/high)

    Args:
        benchmark_results: Output from benchmark_inference()
        target_latency_s: Target latency per claim

    Returns:
        A formatted string report with the optimization plan

    Hints:
        1. KV-cache is usually already enabled by default in HuggingFace generate()
        2. INT8 quantization typically gives 1.5-2x speedup with < 1% accuracy loss
        3. INT4 quantization gives 2-3x speedup but may degrade reasoning quality
        4. Speculative decoding with a small draft model can give 2-3x speedup
        5. Early stopping after </think> + answer saves generating padding tokens
    """
    # TODO: Implement this function
    pass
```

3.9 Ethical and Regulatory Analysis

Context: Insurance claims adjudication directly affects people's financial well-being. Errors can mean a family cannot rebuild after a disaster, or an insurer pays fraudulent claims. Regulatory compliance is not optional.

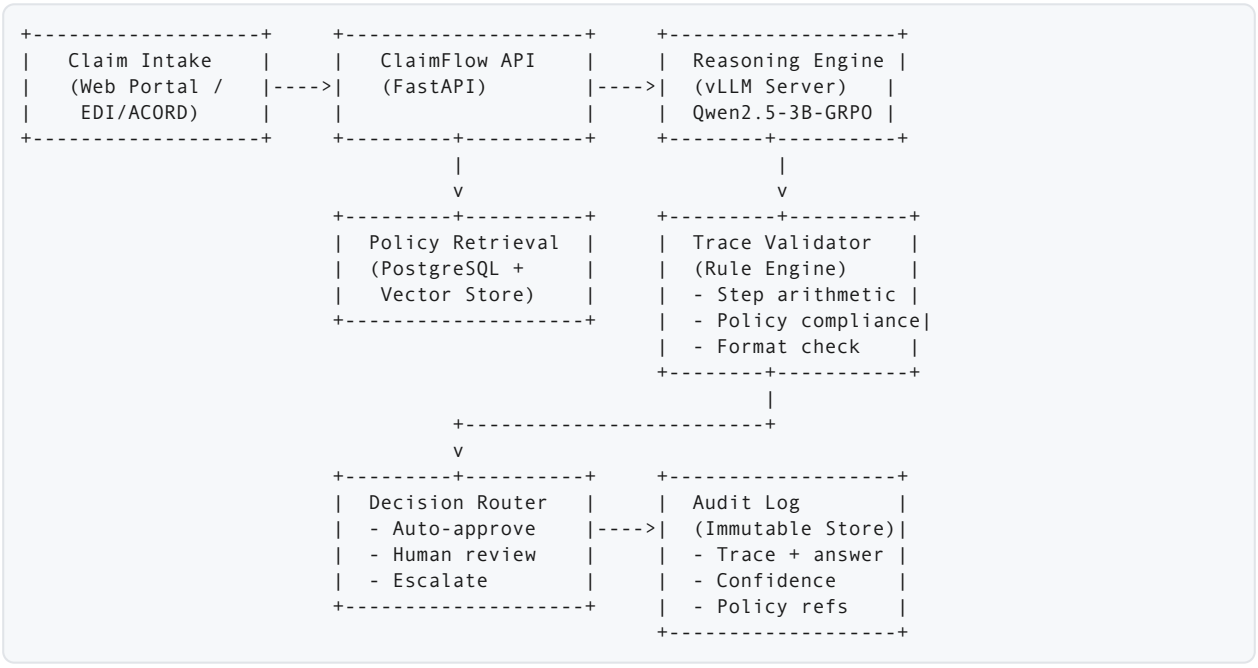
TODO 9: Ethical Impact Assessment

```
def ethical_impact_assessment(model, tokenizer, test_claims: List[InsuranceClaim]) -> str:
    """
    Conduct an ethical impact assessment for the claims reasoning model.

    Address the following:

    1. BIAS ANALYSIS
       - Test whether the model's accuracy varies by claim type (property damage,
         theft, natural disaster, liability)
       - Check if payout errors are systematically biased (does the model
         tend to overpay or underpay?)
       - Analyze if claim description phrasing affects the payout
         (e.g., formal vs. informal language)

    2. FAIRNESS METRICS
       - Compute demographic parity: does the model's error rate differ
         across geographic regions or claim sizes?
       - Compute equalized odds: conditional on the true payout, is the
         model equally accurate for high and low payouts?

    3. REGULATORY COMPLIANCE CHECKLIST
       - NAIC Model Bulletin on AI in Insurance (2024): explainability requirement
       - State insurance commissioner guidelines: audit trail requirement
       - SOC 2 Type II: data handling and access controls
       - GDPR/CCPA: if processing claims with personal data

    4. FAILURE MODE RISKS
       - What is the worst case if the model overpays? (insurer financial loss)
       - What is the worst case if the model underpays? (policyholder harm,
         bad faith litigation)
       - How should the system handle low-confidence predictions?

    5. HUMAN-IN-THE-LOOP DESIGN
       - When should the model's decision be automatically approved?
       - When should a human reviewer be required?
       - How should the reasoning trace be presented to the reviewer?

    Returns:
        A formatted string report addressing all five areas

    Hints:
        1. For bias analysis, group claims by type and compare error rates
        2. For fairness, use the claims' policy_terms to define subgroups
        3. A good threshold for human review: any claim where the model's
           top-2 payout predictions differ by more than 10%
        4. Consider the asymmetry: underpayment causes more harm to individuals
           than overpayment causes to insurers
    """
    # TODO: Implement this function
    pass
```

Thought questions: - Should the model's confidence score affect whether a claim is auto-approved or sent to a human reviewer? How would you compute a meaningful confidence score from a generative model? - If the model is more accurate on large claims (where it has more reasoning steps and thus more signal during GRPO), is this a fairness issue? Small claims affect more people but get less accurate decisions.

Section 4: Production and System Design Extension

Architecture Diagram



API Design

Endpoint: `POST /api/v1/claims/adjudicate`

Request:

```
{
    "claim_id": "CLM-2025-00847",
    "claim_description": "Roof damage from hailstorm...",
    "policy_id": "POL-HC-39201",
    "policy_terms": {
        "deductible": 1000,
        "coverage_limit": 250000,
        "depreciation_rate": 0.10,
        "sub_limits": {"jewelry": 5000, "electronics": 10000},
        "co_insurance": {"insurer": 0.80, "policyholder": 0.20}
    },
    "priority": "standard"
}
```

Response:

```
{
    "claim_id": "CLM-2025-00847",
    "payout_amount": 19040.00,
    "confidence": 0.92,
    "decision": "auto_approved",
    "reasoning_trace": "<think>\nStep 1: Identify claimed items...\n</think>",
    "reasoning_steps": [
        {"step": 1, "description": "Identify items", "result": 28500.00, "verified": true},
        {"step": 2, "description": "Apply depreciation", "result": 24800.00, "verified": true}
    ],
    "policy_references": ["Section 4.2: Depreciation Schedule", "Section 6.1: Deductible"],
    "processing_time_ms": 8420,
    "model_version": "grpo-v2.1.0"
}
```

Serving Infrastructure

- **Model server:** vLLM with continuous batching (handles variable-length reasoning traces efficiently)
- **Hardware:** 2x NVIDIA A10G (24GB each), INT8 quantized model
- **Scaling:** Horizontal pod autoscaler (HPA) based on queue depth, scale from 2 to 8 replicas
- **Load balancing:** Round-robin with session affinity disabled (each claim is independent)

Latency Budget

Component	Target	P50	P95
API ingestion + policy retrieval	200ms	120ms	280ms
Prompt construction	10ms	5ms	15ms
Model inference (reasoning generation)	10,000ms	7,200ms	12,400ms
Trace validation (rule engine)	100ms	60ms	150ms
Decision routing + audit logging	50ms	30ms	80ms
Total	**< 15,000ms**	**~7,400ms**	**~12,900ms**

Monitoring

- **Model metrics:** Mean reward (rolling 1h window), exact match accuracy (daily batch eval), mean reasoning trace length, KL divergence from reference
- **System metrics:** P50/P95/P99 latency, GPU utilization, memory usage, queue depth, error rate
- **Business metrics:** Auto-approval rate, human review rate, escalation rate, payout discrepancy rate (model vs. final adjuster decision)
- **Alerting:** Page on-call if exact match accuracy drops below 80% on daily eval, or if P95 latency exceeds 14 seconds

Model Drift Detection

- Daily evaluation on a held-out "golden set" of 500 claims with verified payouts
- Track distribution of predicted payouts over time (detect shifts using Kolmogorov-Smirnov test)
- Monitor the ratio of format-compliant traces (a drop indicates model degradation)
- Trigger retraining if accuracy drops below 82% for 3 consecutive days

Model Versioning

- All models stored in S3 with semantic versioning (e.g., `grpo-v2.1.0`)
- Each version includes: model weights, tokenizer, GRPO config, training data hash, evaluation results

- Rollback strategy: keep the previous 3 versions warm-loaded, can switch traffic in < 60 seconds via feature flag

A/B Testing

- Shadow mode: new model runs alongside production model, both process the same claims, only production model's decision is used
- Statistical test: two-proportion z-test on exact match accuracy, require $p < 0.01$ and at least 1,000 claims per arm
- Guardrail metrics: new model must not increase overpayment rate by more than 0.5% or decrease format compliance below 97%
- Canary deployment: route 5% of traffic to new model for 48 hours before full rollout

CI/CD for ML

- **Training pipeline:** Airflow DAG triggered weekly or on-demand
- Pull new verified claims from data warehouse
- Run SFT warm-up (if new base model) or skip
- Run GRPO training (8x A100, ~48 hours)
- Evaluate on golden set
- If accuracy > 85%, push to model registry
- **Validation gates:** (a) accuracy gate (> 85% exact match), (b) latency gate (< 15s P95), (c) format gate (> 98% compliance), (d) bias gate (no subgroup accuracy below 80%)
- **Deployment:** Automated canary deployment via ArgoCD

Cost Analysis

Component	Monthly Cost
Training (GRPO, 8x A100, 48h/week)	\$6,400
Inference (2x A10G, 24/7)	\$2,900
Autoscaling (peak: 8 replicas, ~4h/day)	\$1,200
Storage (model versions, logs, traces)	\$400
Monitoring and evaluation pipeline	\$300
Total	**\$11,200/month**

At 100,000 claims/month and \$4.50/claim revenue, infrastructure cost is approximately 2.5% of revenue. The \$2.8M annual savings from reduced payout errors provides a 20x ROI on infrastructure investment.