Masters Research Project

Entropy-Adaptive Branching for Efficient Semantic Uncertainty Estimation

Donia Gasmi

Supervisors:

Dr. Geovani Rizk & Dr. Gauthier Voron

Professor:

Prof. Rachid Guerraoui

EPFL

EPFL

Table of Content

EPFL

1 Introduction

Large language models (LLMs) achieve impressive performance across diverse tasks, yet remain prone to generating plausible but incorrect outputs, a phenomenon known as hallucination. As these models are deployed in high-stakes domains such as healthcare and legal reasoning, reliably quantifying *when* a model is uncertain becomes as important as the predictions themselves.

Existing uncertainty quantification approaches face a fundamental efficiency-quality trade-off. Single-sample methods like token-level entropy are computationally cheap but miss semantic uncertainty: a model may generate contradictory answers (e.g., "Paris" vs. "Lyon" for France's capital) while remaining confident at each token. Multi-sample methods such as semantic entropy [1] address this by generating diverse outputs and measuring disagreement, but require independent forward passes per sample, incurring computational costs that scale linearly with sample count.

The core challenge is generating diverse samples *efficiently*. Naive multi-sample generation redundantly encodes the same prompt multiple times. Beam search prioritizes high-probability paths at the cost of diversity; temperature sampling produces diversity at the cost of coherence. No existing method efficiently explores the model's uncertainty landscape while enabling reliable disagreement detection.

We introduce **Entropy-Adaptive Branching (EAB)**, a sampling algorithm that generates diverse samples by branching *only* at high-entropy token positions. EAB encodes the prompt once, then dynamically creates new generation paths when normalized entropy exceeds a threshold τ, indicating genuine next-token ambiguity. This strategy shares prefix computation across samples while selectively exploring uncertain regions.

EAB forms Layer 1 of a two-layer uncertainty system. It efficiently generates N diverse samples by capturing token-level uncertainty through adaptive branching. Layer 2 applies **Semantic Entropy (SE)**: clustering responses by meaning and computing entropy over the resulting distribution. Together, the layers combine local (token) and global (semantic) signals for comprehensive uncertainty quantification.

Contributions.

- **Entropy-Adaptive Branching:** A novel algorithm that generates diverse samples efficiently by branching only at high-entropy positions, reducing redundant computation while preserving sample diversity.
- **Two-layer uncertainty architecture:** A unified framework combining token-level (EAB) and semantic-level (SE) uncertainty signals.
- **Empirical evaluation:** Comprehensive experiments analyzing EAB's computational efficiency, branching behavior, and uncertainty estimation quality across varying configurations.

The remainder of this report is organized as follows: Section 2 provides background on entropy measures and semantic clustering. Section 3 details EAB and the two-layer architecture. Section 4 describes implementation choices. Section 5 presents experimental results. Section 6 concludes and discusses limitations and future work.

2 Background

This section outlines foundational concepts relevant to our work on efficient multi-sample generation for LLM uncertainty quantification. We cover autoregressive decoding, KV-cache reuse across shared prefixes, and techniques for assessing semantic diversity among generated responses. We review autoregressive decoding, prefix reuse via KV-caching, and methods for quantifying and interpreting output uncertainty, particularly those based on semantic diversity across multiple samples.

2.1 Large Language Models and Autoregressive Generation

Large language models (LLMs) generate text autoregressively by predicting tokens sequentially based on preceding context. Given an input prompt $\mathbf{x} = (x_1, \ldots, x_n)$, the model produces a probability distribution over the vocabulary $\mathcal{V}$ for the next token:

$$P(x_{n+1} \mid x_1, \ldots, x_n; \theta) \tag{1}$$

where θ represents the model parameters. Modern transformer architectures [6] maintain a key-value (KV) cache to store intermediate representations, enabling efficient sequential generation without recomputing past tokens.

2.2 KV-Cache Sharing and Prefix Reuse

The KV-cache stores the key and value vectors computed for each token in the input sequence. When generating multiple responses from the same prompt, these cached representations for the shared prefix can be reused across samples, avoiding redundant computation.

For a prompt of length n and continuations diverging at position t, all samples share the KV-cache for positions 1 to $t-1$. Only the divergent portions from position t onward require separate cache storage. This prefix sharing enables memory-efficient parallel generation: for a model with L layers and hidden dimension d, generating m samples of total length T each without sharing requires $O(m \cdot L \cdot d \cdot T)$ memory. With prefix sharing up to position t, the total memory reduces to $O(L \cdot d \cdot (t + m \cdot (T - t)))$, where the shared prefix of length t is stored once and only the m divergent continuations require separate storage.

The challenge lies in strategically identifying branching points that maximize prefix sharing while maintaining sufficient diversity among generated samples.

2.3 Uncertainty Quantification in LLMs

Uncertainty quantification aims to measure the reliability of model outputs, particularly important for high-stakes applications. We distinguish between two types of uncertainty:

- **Aleatoric uncertainty** arises from inherent randomness or ambiguity in the task itself, such as questions with multiple valid answers.
- **Epistemic uncertainty** reflects the model's lack of knowledge, often due to limited training data or out-of-distribution inputs.

A straightforward uncertainty measure is *predictive entropy*, computed from the model's token probability distribution:

$$H(P) = -\sum_{v \in \mathcal{V}} P(v) \log P(v) \tag{2}$$

However, this single-token metric provides only local uncertainty and does not account for the semantic content of full responses.

An alternative approach generates multiple complete outputs through sampling and measures their disagreement [3]. A key challenge is that naive comparison of token sequences fails to capture semantic equivalence. For example, "The capital is Paris" and "Paris is the capital" express the same meaning despite having different surface forms.

2.4 Semantic Entropy and Entailment-Based Methods

Kuhn et al. [1] exemplify this sampling-based approach by generating multiple responses and measuring disagreement at the semantic level. They introduced *semantic entropy*, which clusters semantically equivalent outputs before computing uncertainty. Their method uses bidirectional entailment to determine equivalence: two responses y_i and y_j are considered equivalent if a natural language inference (NLI) model predicts $y_i \vDash y_j$ and $y_j \vDash y_i$.
Given a set of sampled responses $\mathcal{Y} = \{y_1, \ldots, y_m\}$ partitioned into semantic clusters $\mathcal{C} = \{C_1, \ldots, C_k\}$, semantic entropy is computed as:

$$H_{\text{sem}} = -\sum_{i=1}^{k} p(C_i) \log p(C_i) \tag{3}$$

where $p(C_i) = \sum_{y \in C_i} p(y)$ aggregates the probabilities of responses within each cluster.
While effective, entailment-based methods have limitations. NLI models themselves are imperfect and may introduce errors in clustering decisions, potentially misclassifying semantically distinct responses as equivalent or vice versa.

2.5 Embedding-Based Semantic Similarity

An alternative approach leverages dense embeddings from sentence encoders [4]. Given a response y, an encoder $f : \mathcal{Y} \to \mathbb{R}^d$ maps it to a fixed-dimensional vector $\mathbf{e}_y = f(y)$. Semantic similarity can then be measured using cosine similarity:

$$\text{sim}(y_i, y_j) = \frac{\mathbf{e}_{y_i} \cdot \mathbf{e}_{y_j}}{\|\mathbf{e}_{y_i}\| \|\mathbf{e}_{y_j}\|} \tag{4}$$

Clustering algorithms such as hierarchical agglomerative clustering [7] can group semantically similar responses based on embedding distances. This approach avoids the need for entailment models while maintaining the ability to capture semantic equivalence beyond surface-level token matching.

2.6 Computational Constraints in Sampling

Generating multiple diverse samples for uncertainty estimation is computationally expensive. Each sample requires maintaining its own KV-cache during generation. For a model with hidden dimension d, L layers, and maximum sequence length T, storing KV-caches for m samples requires $O(m \cdot L \cdot d \cdot T)$ memory.
While sequential generation of samples can reduce peak memory usage, it incurs prohibitive time costs, as each sample must wait for previous generations to complete. Conversely, parallel generation of all samples minimizes latency but requires simultaneous storage of all KV-caches. This presents a fundamental tradeoff between time and memory efficiency. Practical uncertainty quantification at scale requires sampling strategies that navigate this tradeoff effectively, reducing computational overhead while maintaining sample diversity.

3 Design

Our system quantifies LLM uncertainty through a two-layer architecture that combines efficient multi-sample generation with semantic analysis. This section describes the design rationale and key components.

3.1 Two-Layer Uncertainty Quantification

We quantify LLM uncertainty through two complementary layers:

1. **Token-level uncertainty (Layer 1):** Measured during generation via next-token entropy. High entropy triggers adaptive branching, producing diverse continuations where the model is uncertain.
2. **Semantic-level uncertainty (Layer 2):** Measured after generation by clustering responses into meaning-equivalent groups and computing entropy over cluster assignments. High semantic entropy indicates disagreement among generated responses.

Core Idea. While token-level entropy captures local ambiguity (e.g., choosing between "Paris" and "Lyon"), it cannot distinguish between meaningful disagreement and superficial variation (e.g., "The capital is Paris." vs. "Paris is the capital."). Semantic entropy resolves this by grouping responses based on meaning, not surface form. Together, the two layers provide a more complete picture of uncertainty:

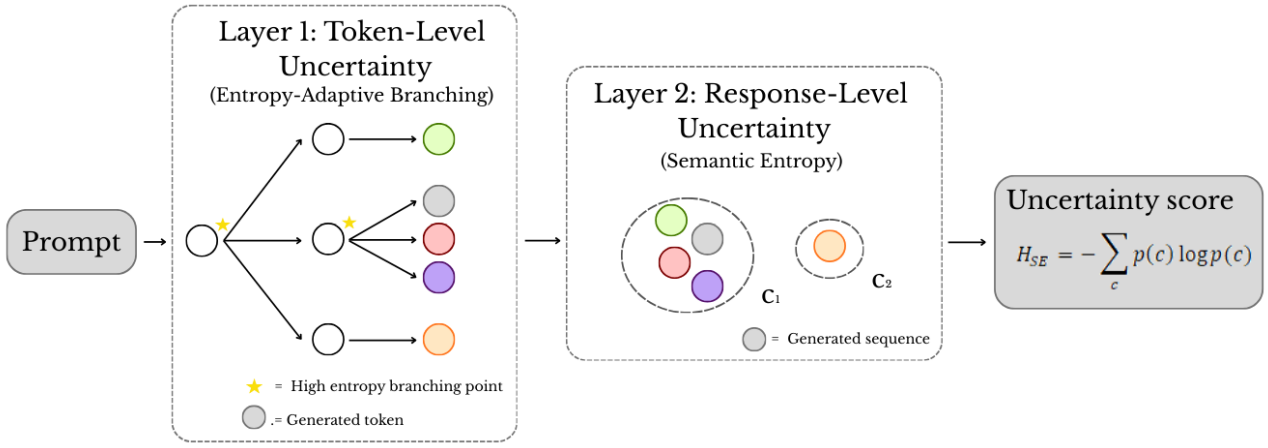

Figure 1: Two-layer uncertainty quantification architecture. Layer 1 generates diverse samples via entropy-adaptive branching; Layer 2 clusters them by semantic similarity and computes response-level entropy to produce a final uncertainty score.

3.2 Layer 1: Entropy-Adaptive Branching (EAB)

EAB generates diverse samples efficiently by dynamically branching only when the model is uncertain, while reusing computation for shared prefixes.

3.2.1 Core Idea

Traditional multi-sample generation processes the prompt independently for each sample, duplicating computation for the shared prefix. EAB eliminates this redundancy by:

- Encoding the prompt **once** and caching its KV states
- Generating tokens autoregressively, **sharing all intermediate states** until a branching decision
- **Branching only at high-entropy positions**, where the model's next-token distribution is ambiguous
- Enforcing a **fixed path budget** to prevent exponential growth

This efficiency is illustrated in Figure 2: EAB computes each token, prompt or generated, exactly once per unique prefix, regardless of how many samples diverge afterward.

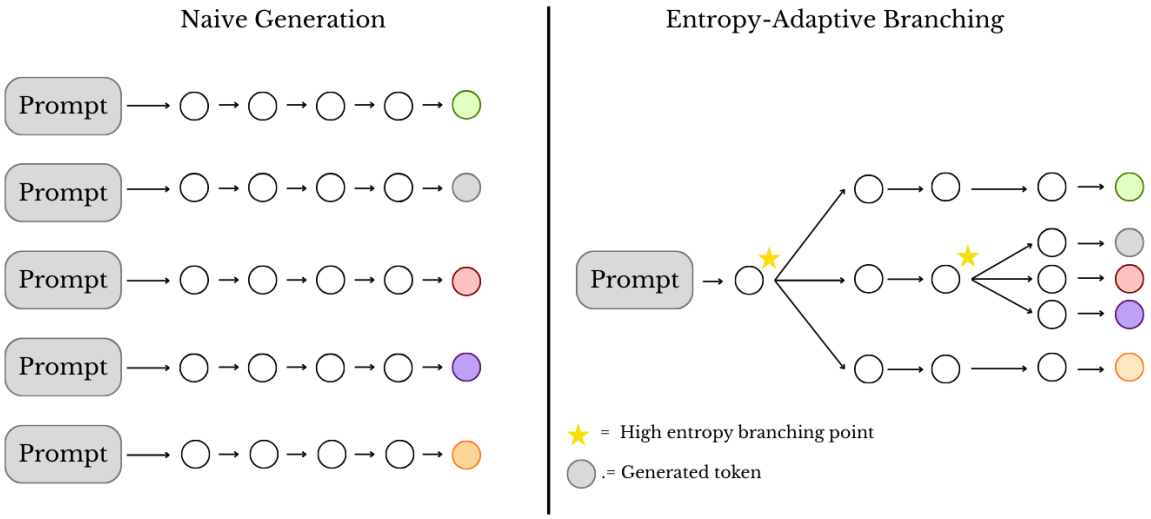

Figure 2: Conceptual comparison of naive generation versus Entropy-Adaptive Branching (EAB). Left: Naive generation re-encodes the prompt for each sample. Right: EAB encodes the prompt once and branches adaptively at high-entropy positions (stars), producing diverse samples with minimal redundant computation.

3.2.2 Entropy as Branching Signal

We use normalized entropy to measure token-level uncertainty during generation:

$$H_{\text{norm}}(p) = -\frac{\sum_{i=1}^{V} p_i \log p_i}{\log V} \tag{5}$$

where p is the next-token probability distribution over vocabulary V. Normalization ensures values lie in $[0, 1]$, enabling consistent thresholds across models and prompts.

To validate entropy as a reliable branching signal and calibrate the threshold τ, we conducted a pilot study on 200 diverse prompts, grouped by expected model confidence: **high-confidence** (70 factual QA, e.g., "What is the capital of France?"), **medium-confidence** (65 advice or opinion prompts), and **low-confidence** (65 creative or open-ended prompts, e.g., "Write a short poem about forbidden love.").

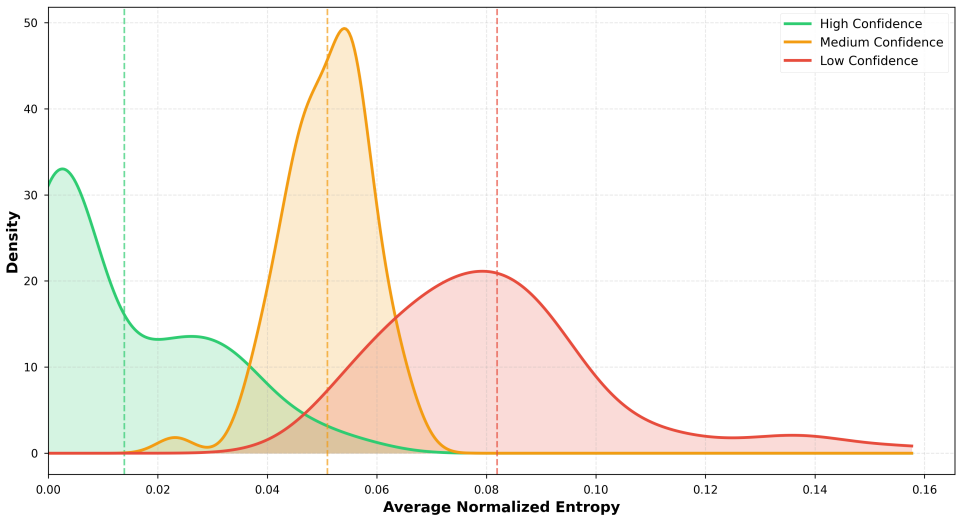

Figure 3: Average normalized entropy distributions by prompt confidence level. High-confidence prompts (green) cluster at low entropy; medium (orange) and low (red) confidence show progressively higher entropy.

For each prompt, we performed standard autoregressive generation (greedy decoding) and recorded the normalized entropy at every token. We then computed the average normalized entropy per prompt and plotted the resulting distributions (Figure 3). Statistical analysis confirmed strong separation between groups:

- One-way ANOVA: $F = 315.85$, $p < 0.001$ → significant differences in mean entropy
- Cohen's d: Large effect sizes (> 1.9) between adjacent groups → meaningful separation

Based on this, we set $\tau = 0.055$; the 75th percentile of medium-confidence prompts. This choice ensures:

- High-confidence prompts branch rarely (only 44% exceed τ) → preserves efficiency
- Medium/low-confidence prompts branch reliably (100% exceed τ) → captures true uncertainty
- Classification accuracy: 96.5% in distinguishing high vs. medium/low confidence based solely on whether branching occurred

This threshold operationalizes our core design principle: **let the number of generated samples reflect true uncertainty**, turning generation dynamics into a reliable signal for downstream uncertainty quantification.

3.2.3 Path Management

EAB maintains a dynamic set of active paths, bounded by a maximum count M (default: 20). At each generation step:

1. Compute next-token entropy for each path.
2. If $H_{\text{norm}} > \tau$ and paths remain within budget, spawn $k = 3$ new paths by sampling distinct tokens.
3. If total paths exceed M, retain only the top-M by cumulative log-probability.

Unlike hard-stop methods, our adaptive strategy ensures exploration throughout generation, capturing more entropy peaks, as shown in Figure 4, without exceeding memory limits, thanks to probability-based pruning.

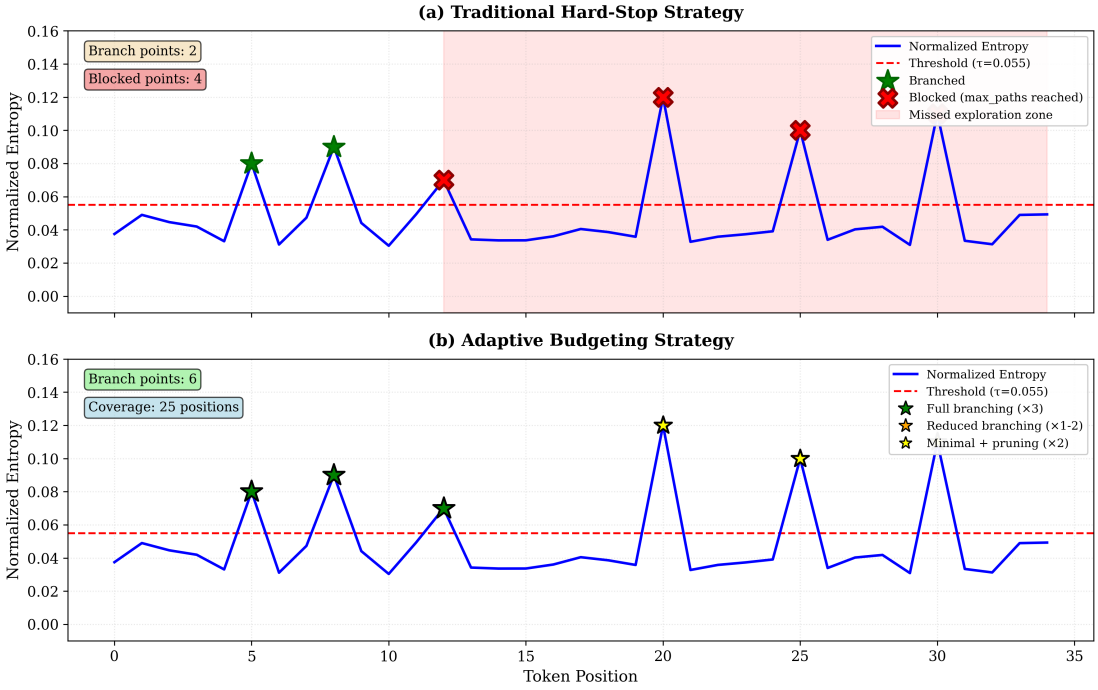

Figure 4: Comparison of path management strategies. (a) Traditional hard-stop blocks branching after M paths are reached, missing later high-entropy positions. (b) Adaptive budgeting continues exploration throughout the sequence (green/yellow stars), achieving 6 branch points vs. 2 while staying under the path limit via pruning.

Adaptive Branch Factor. To prevent exceeding the path budget while still exploring late high-entropy positions, we dynamically adjust the branch factor:

$$k' = \begin{cases} k & \text{if } M - |\mathcal{P}| \geq k \\ M - |\mathcal{P}| & \text{if } 0 < M - |\mathcal{P}| < k \\ 2 & \text{otherwise (minimal branching, rely on pruning)} \end{cases} \tag{6}$$

3.2.4 Efficiency Gains

Let L_p be prompt length and L_g generation length. Naive sampling requires $O(N(L_p + L_g))$ computation for N samples. EAB reduces this to $O(L_p + \sum_i n_i \delta_i)$, where n_i is the number of active paths during segment i of length δ_i. For long prompts or late branching, this yields substantial speedups (see Section 5).

3.2.5 Algorithm

Algorithm 1 presents the complete EAB generation procedure. The algorithm processes prompts through a chat template for instruction-tuned models (line 2), encodes once to obtain the initial KV-cache (line 3), then iteratively generates tokens while adaptively branching at high-entropy positions.

Algorithm 1 Entropy-Adaptive Branching (EAB)

Require: Prompt x, threshold τ, branch factor k, max paths M, max tokens T
Ensure: Set of generated sequences $\mathcal{Y}$

1: $\mathcal{P} \leftarrow \emptyset$ ▷ Active paths
2: $x \leftarrow \textsc{ApplyChatTemplate}(x)$ ▷ Format for instruct models
3: $\text{cache}_0, \text{logits}_0 \leftarrow \textsc{Encode}(x)$ ▷ Single prompt encoding
4: $\mathcal{P} \leftarrow \{(\emptyset, 0, \text{cache}_0)\}$ ▷ Initialize: (tokens, log-prob, cache)
5: **for** $t = 1$ to T **do**
6: $\quad \mathcal{P}' \leftarrow \emptyset$
7: $\quad$ **for** each path $(y, \ell, c) \in \mathcal{P}$ **do**
8: $\quad\quad \text{logits}, c' \leftarrow \textsc{Forward}(y_{-1}, c)$ ▷ Single token forward pass
9: $\quad\quad p \leftarrow \textsc{Softmax}(\text{logits}/\text{temperature})$
10: $\quad\quad H_{\text{norm}} \leftarrow -\sum_v p_v \log p_v \,/\, \log |V|$ ▷ Normalized entropy
11: $\quad\quad$ **if** $H_{\text{norm}} > \tau$ **then** ▷ High uncertainty: branch
12: $\quad\quad\quad k' \leftarrow \textsc{AdaptiveFactor}(k, M, |\mathcal{P}|)$ ▷ Adjust for budget
13: $\quad\quad\quad c_{\text{cow}} \leftarrow \textsc{WrapCOW}(c')$ ▷ Prepare for efficient branching
14: $\quad\quad\quad \{v_1, \ldots, v_{k'}\} \leftarrow \textsc{SampleDistinct}(p, k')$
15: $\quad\quad\quad$ **for** $i = 1$ to k' **do**
16: $\quad\quad\quad\quad \mathcal{P}' \leftarrow \mathcal{P}' \cup \{(y \circ v_i,\ \ell + \log p_{v_i},\ c_{\text{cow}}.\textsc{Branch}())\}$
17: $\quad\quad\quad$ **end for**
18: $\quad\quad$ **else** ▷ Low uncertainty: continue single path
19: $\quad\quad\quad v \leftarrow \textsc{Sample}(p)$
20: $\quad\quad\quad \mathcal{P}' \leftarrow \mathcal{P}' \cup \{(y \circ v,\ \ell + \log p_v,\ c')\}$
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: $\quad$ Move paths ending with EOS to completed set $\mathcal{Y}$
24: $\quad$ **if** $|\mathcal{P}'| > M$ **then** ▷ Prune if over budget
25: $\quad\quad \mathcal{P} \leftarrow \textsc{TopK}(\mathcal{P}', M, \text{key} = \ell)$ ▷ Keep highest probability
26: $\quad$ **else**
27: $\quad\quad \mathcal{P} \leftarrow \mathcal{P}'$
28: $\quad$ **end if**
29: **end for**
30: **return** $\mathcal{Y} \cup \mathcal{P}$

3.3 Layer 2: Semantic Entropy (SE)

While token-level entropy captures local uncertainty during generation, it cannot detect *semantic* disagreement across complete responses. Layer 2 addresses this by clustering generated responses by meaning and computing entropy over the resulting distribution.

3.3.1 Core Idea

Token-level entropy triggers branching whenever the model faces lexical ambiguity, but not all ambiguity reflects genuine uncertainty. Consider two scenarios for the prompt *"What is France's capital?"*:

1. **Meaningful disagreement**: The model generates "The capital is **Paris**." and "The capital is **Lyon**.". → High token entropy at the city name (justifiably), and semantic clustering reveals *contradictory meanings*.
2. **Superficial variation**: The model generates "The capital is **Paris**." and "The capital is **named Paris**.". → Similarly high token entropy at the city name (due to phrasing differences), but semantic clustering groups both into the *same meaning*.

In both cases, Layer 1 (EAB) branches appropriately, but only Layer 2 distinguishes factual uncertainty from stylistic variation. Semantic entropy thus provides a more reliable signal of true model confusion.

3.3.2 Embedding-Based Semantic Similarity

We measure semantic similarity by encoding responses into a shared embedding space using a pretrained sentence encoder. Two responses r_i and r_j are considered semantically similar if their cosine similarity exceeds a threshold:

$$\text{sim}(r_i, r_j) = \frac{\mathbf{e}_i \cdot \mathbf{e}_j}{\|\mathbf{e}_i\|\|\mathbf{e}_j\|} > \theta \tag{7}$$

where $\mathbf{e}_i = \text{encode}(r_i)$ is the embedding of response r_i. This captures semantic equivalence even when responses differ lexically (e.g., "The capital is Paris" ≈ "Paris is France's capital").

3.3.3 Agglomerative Clustering

Given N generated responses $\{r_1, \ldots, r_N\}$, we partition them into semantic clusters using agglomerative clustering with average linkage:

1. Encode all responses: $\mathbf{e}_i = \text{encode}(r_i)$
2. Compute pairwise cosine distances: $d_{ij} = 1 - \text{sim}(r_i, r_j)$
3. Merge clusters hierarchically until distance exceeds threshold δ

The number of clusters K emerges naturally, ranging from $K = 1$ (full agreement) to $K = N$ (complete disagreement).

3.3.4 Semantic Entropy Computation

Let $p_k = |C_k|/N$ be the fraction of responses in semantic cluster k. We compute normalized semantic entropy as:

$$H_{\text{sem}}^{\text{norm}} = -\frac{\sum_{k=1}^{K} p_k \log p_k}{\log K} \tag{8}$$

This yields a value in $[0, 1]$, where 0 indicates full agreement (all responses in one cluster) and 1 indicates uniform disagreement across K clusters.

3.3.5 Combined Uncertainty Score

Normalized entropy alone has a limitation: both 2 uniform clusters and 10 uniform clusters yield $H_{\text{sem}}^{\text{norm}} = 1$, yet 10 clusters indicates higher uncertainty. We address this with a combined score:

$$U = \alpha \cdot H_{\text{sem}}^{\text{norm}} + (1 - \alpha) \cdot \frac{K - 1}{N - 1} \tag{9}$$

where $\alpha = 0.6$ weights entropy and $(K-1)/(N-1)$ measures cluster diversity relative to the maximum possible.

3.3.6 Combining Both Layers

The two layers provide complementary uncertainty signals, enabling fine-grained diagnosis of model behavior:

Scenario	**Token-level**	**Response-level**
Confident & consistent	Low	Low
Confident & contradictory	Low	High
Hesitant & consistent	High	Low
Hesitant & contradictory	High	High

These categories support actionable interpretation:

- **Confident & consistent** → High reliability; suitable for automated use.
- **Confident & contradictory** → Potential hallucination or factual error: the model is certain but generates conflicting answers (e.g., "Paris" vs. "Lyon" for France's capital).
- **Hesitant & consistent** → Stylistic uncertainty: the model explores phrasing variations but agrees on meaning (e.g., "Paris is the capital" vs. "The capital is Paris").
- **Hesitant & contradictory** → Genuine ambiguity: the model is uncertain both locally and globally, suggesting the prompt is ambiguous, out-of-distribution, or requires external knowledge.

EPFL

4 Implementation

We implement our two-layer uncertainty quantification system in Python using Hugging Face Transformers [8]. The full codebase is publicly available at: DoniaGasmii/cost-aware-semantic-uncertainty-llm

This section describes key implementation decisions, data structures, and algorithms.

4.1 Layer 1: Entropy-Adaptive Branching (EAB)

4.1.1 Framework and Model Integration

We use `AutoModelForCausalLM` from the Transformers library with manual control over the generation loop. This provides direct access to:

- Raw logits for entropy computation
- KV-cache states (`past_key_values`) for prefix reuse
- Per-token log-probabilities for path scoring

For instruction-tuned models (e.g., Qwen2.5-Instruct), we apply the model's chat template to format prompts before generation. This ensures proper handling of system prompts and conversation structure, which is critical for eliciting well-calibrated responses from instruct-tuned models.
While this approach forgoes the throughput optimizations of engines like vLLM [2], it offers the fine-grained introspection required for dynamic branching and uncertainty analysis.

4.1.2 Path Representation

Each active generation path is represented by a lightweight data structure:

```
class GenerationPath:
    tokens: List[int]              # Generated token IDs
    log_prob: float                # Cumulative log P(path)
    cache: Tuple[Tensor, ...]      # KV-cache up to current step
    branch_points: List[int]       # Positions where branching occurred
    parent_id: Optional[int]       # Parent path ID (tree tracking)
    path_id: Optional[int]         # Unique path identifier
```

The initial path contains an empty token list, zero log-probability, and the KV-cache from prompt encoding. When branching occurs, child paths copy the parent's state and record the branch position. A `PathManager` maintains separate lists for active and completed paths, moving paths to the completed set upon EOS token generation.

4.1.3 KV-Cache Management

To minimize memory overhead during branching, we implement a **copy-on-write (COW)** strategy:

- When a path branches, child paths store a *reference* to the parent's cache plus their own divergent tokens.
- The full cache for a child is reconstructed on-demand by concatenating the shared prefix (from parent) with its private suffix.
- Actual tensor copying occurs only when a path modifies its cache (lazy allocation).

This reduces peak memory usage by $\sim$50% compared to eager deep copying, critical for maintaining a reasonable path budget ($M = 20$) on a single GPU.

4.1.4 Branching and Sampling Logic

At each generation step, we process all active paths sequentially:

1. For each path, run a forward pass with its cached states to obtain next-token logits.
2. Compute normalized entropy H_{norm} from the softmax distribution.
3. If $H_{\text{norm}} > \tau$ and path budget allows:
   - Sample k *distinct* tokens via multinomial sampling without replacement
   - Create k child paths, each initialized with:
     - Parent's token sequence + sampled token
     - Updated cumulative log-probability
     - COW reference to parent cache
4. Otherwise, extend the path with the highest-probability token.
5. After processing all paths, prune to top-M by log-probability if needed.

Sequential processing of paths simplifies bookkeeping for paths of varying lengths, though it sacrifices batch-level parallelism.

4.1.5 System Configuration

Default hyperparameters:

- Entropy threshold: $\tau = 0.055$
- Branch factor: $k = 3$
- Max paths: $M = 20$
- Temperature: $T = 0.8$
- Max new tokens: 50

All experiments use `Qwen/Qwen2.5-3B-Instruct`[5] unless otherwise noted, running on an NVIDIA A100 with 26GB VRAM.

4.2 Layer 2: Semantic Entropy (SE)

After EAB generates N candidate responses, we compute semantic entropy to quantify meaning-level disagreement.

4.2.1 Sentence Encoding

We use Sentence Transformers [4] to encode responses into dense vectors. Three encoder options are supported:

- `sentence-t5-xxl`: 768-dim embeddings, highest quality (default)
- `all-mpnet-base-v2`: 768-dim, good quality/speed tradeoff
- `all-MiniLM-L6-v2`: 384-dim, fastest, lower quality

Embeddings are L2-normalized to enable cosine similarity via dot product.

4.2.2 Clustering Implementation

We use scikit-learn's `AgglomerativeClustering` with the following configuration:

```
clustering = AgglomerativeClustering(
    n_clusters=None,            # Determined by threshold
    distance_threshold=0.05,    # Merge if similarity > 0.95
    metric="cosine",
    linkage="average"
)
labels = clustering.fit_predict(embeddings)
```

The distance threshold $\delta = 0.05$ corresponds to a similarity threshold of $\theta = 0.95$, requiring near-identical semantic content for cluster membership.

4.2.3 Semantic Entropy Computation

By default, we compute semantic entropy using uniform cluster weights based on response counts. Let C_k be the set of responses in cluster k, and N the total number of responses. The probability of cluster k is:

$$p_k = \frac{|C_k|}{N} \tag{10}$$

and normalized semantic entropy is:

$$H_{\text{sem}}^{\text{norm}} = -\frac{\sum_{k=1}^{K} p_k \log p_k}{\log K} \tag{11}$$

This is implemented as:

```
def compute_semantic_entropy(labels):
    cluster_counts = np.bincount(labels)
    probs = cluster_counts / len(labels)
    probs = probs[probs > 0]  # Avoid log(0)
    return -np.sum(probs * np.log(probs))
```

Optional: Probability-Weighted Clustering When EAB provides path log-probabilities $\log P_i$, we can weight clusters by generation likelihood:

$$p_k = \frac{\sum_{i \in C_k} \exp(\log P_i)}{\sum_{j=1}^{N} \exp(\log P_j)} \tag{12}$$

This down-weights clusters containing low-probability outliers, focusing uncertainty estimation on likely responses, following the standard formulation of semantic entropy [1]. While this variant is implemented, our experiments use the default count-based weighting (each response contributes equally).

4.3 Integration Pipeline

The full uncertainty quantification pipeline:

1. **Generate:** EAB produces N diverse responses with token-level entropy tracked
2. **Encode:** Sentence transformer maps responses to embeddings
3. **Cluster:** Agglomerative clustering groups semantically similar responses into K clusters
4. **Quantify:** Compute normalized semantic entropy $H_{\text{sem}}^{\text{norm}}$ and combined uncertainty score U
5. **Report:** Return token-level statistics and response-level metrics $(H_{\text{sem}}^{\text{norm}}, K, U)$