# Causal Decoding

Complete Decoder-Only Language Model: Embeddings, Transformer Blocks,
Output Projection, and Autoregressive Generation with Sampling

The culmination of Phase 3: a complete CausalLM that takes token IDs and generates token IDs, the same interface as any production LLM. This is the naive version -- full forward pass recompute at every generation step -- deliberately inefficient to motivate KV caching.

This demo covers:
1. Full forward pass walkthrough with shape tracing
2. Causal property verification (THE centerpiece)
3. Sampling strategies: greedy, temperature, top-k, top-p
4. Autoregressive generation loop visualization
5. Computational cost: naive O(n^2) vs KV cache O(n)
6. Model parameter analysis for Llama/Mistral configs

Model config: V=256, d=64, layers=2, h=4, h_kv=2, d_ff=172
Random seed: 42
Number of visualizations: 6
Examples: 6

*Generated by demo.py*

# Summary of Findings

1. Forward Pass: Token IDs (B,L) -> embedding lookup (B,L,d) -> N transformer blocks -> final RMSNorm -> output logits (B,L,V). Weight tying: W_out = E.T shares memory, saving V*d parameters. Softmax of last-position logits gives next-token probabilities.

2. Causal Property (CENTERPIECE): For sequences sharing prefix [0..k-1], logits at positions 0..k-1 are EXACTLY identical regardless of tokens at positions k+. Verified: changing token at position 3 has ZERO effect on positions 0-2 (diff < 1e-12). This enables autoregressive generation.

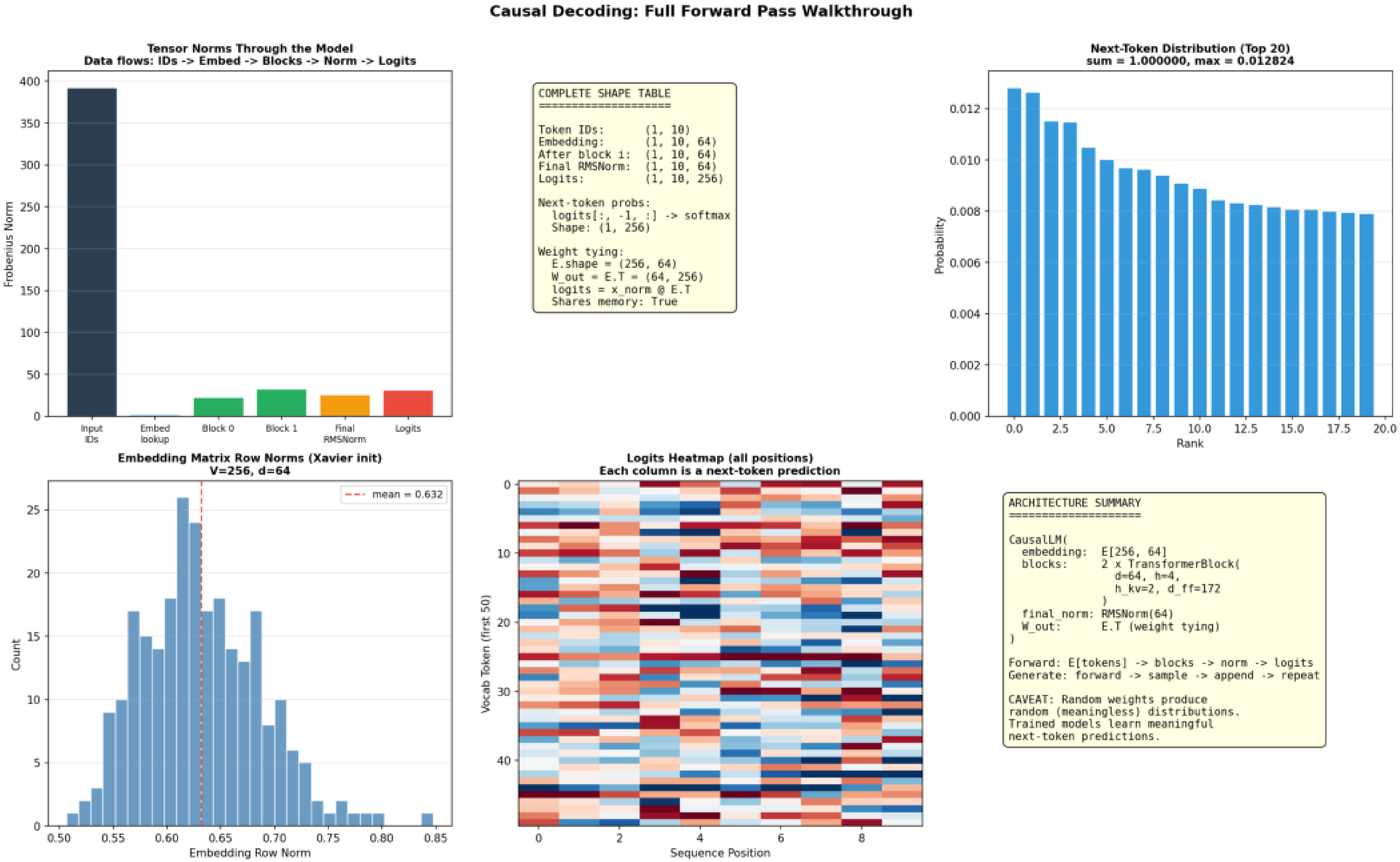
3. Sampling Strategies: Temperature controls sharpness (T<1: deterministic, T>1: random). Top-k keeps only k highest logits. Top-p (nucleus) keeps the smallest set with cumulative probability >= p. Combined pipeline: logits -> /T -> top-k -> top-p -> softmax -> categorical sample.

4. Generation Loop: Naive version does full forward pass at each step, processing the growing sequence (P, P+1, ..., P+n-1 tokens). Total cost: n*P + n(n-1)/2 token-steps. With KV cache: P + n. The redundant recomputation motivates KV caching.

5. Computational Cost: Naive FLOPs grow super-linearly with generation length. KV cache eliminates redundant K/V projections for previous tokens. Speedup ratio grows approximately linearly with n. This is the single most impactful optimization in LLM inference.

6. Parameter Analysis: Transformer blocks are >86% of total parameters. Llama 2 7B: ~6.74B. Llama 3 8B: ~8.03B. Mistral 7B: ~7.24B. None use weight tying; tying would save V*d (131M-525M). FFN dominates per-block params (~67% MHA, ~81% GQA).

# Example 1: Full Forward Pass Walkthrough

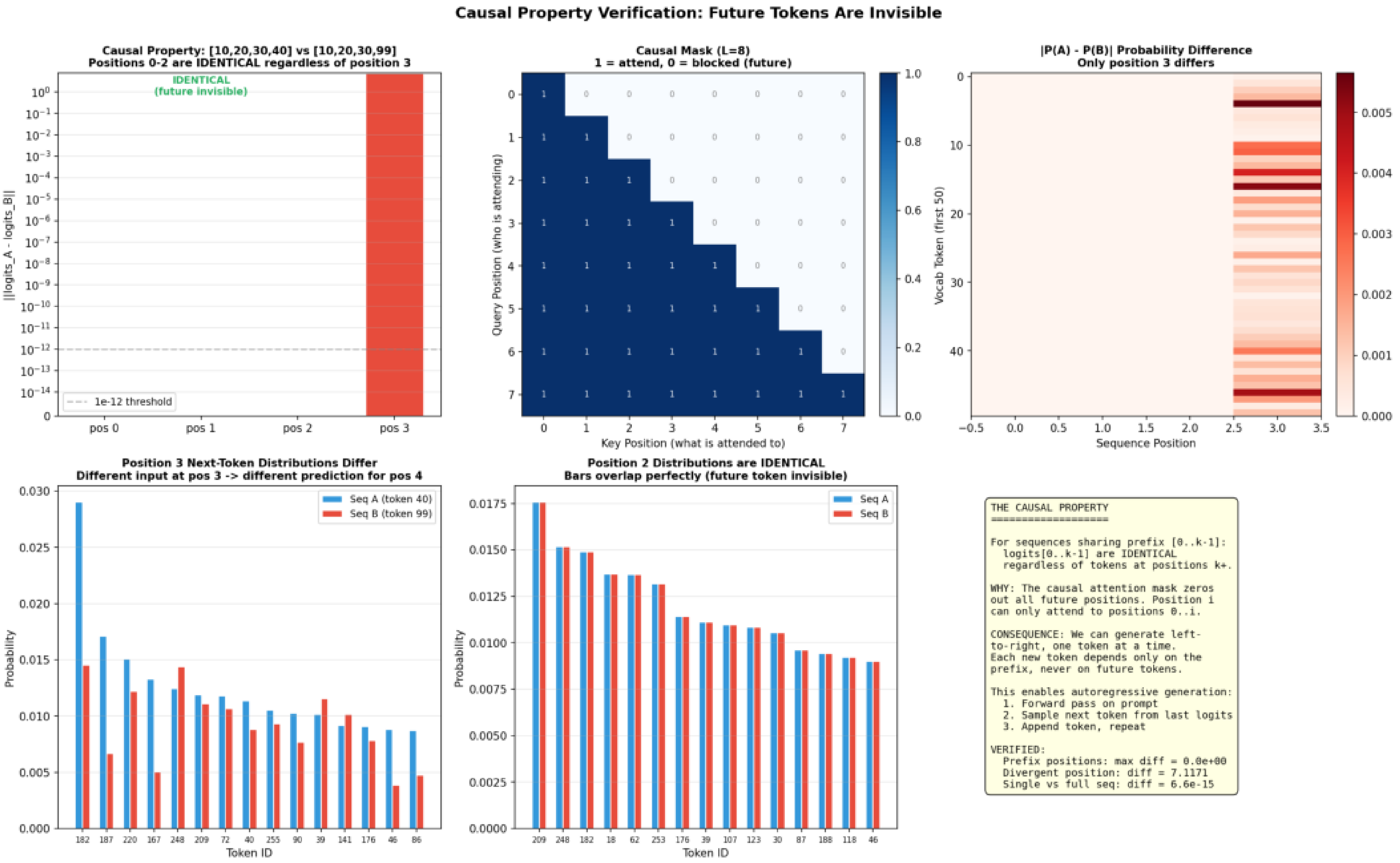


Causal Decoding: Full Forward Pass Walkthrough

```
COMPLETE SHAPE TABLE
====================

Token IDs:       (1, 10)
Embedding:       (1, 10, 64)
After block i:   (1, 10, 64)
Final RMSNorm:   (1, 10, 64)
Logits:          (1, 10, 256)

Next-token probs:
  logits[:, -1, :] -> softmax
  Shape: (1, 256)

Weight tying:
  E.shape = (256, 64)
  W_out = E.T = (64, 256)
  logits = x_norm @ E.T
  Shares memory: True
```

```
ARCHITECTURE SUMMARY
====================

CausalLM(
  embedding:  E[256, 64]
  blocks:     2 x TransformerBlock(
                d=64, h=4,
                h_kv=2, d_ff=172
              )
  final_norm: RMSNorm(64)
  W_out:      E.T (weight tying)
)

Forward: E[tokens] -> blocks -> norm -> logits
Generate: forward -> sample -> append -> repeat

CAVEAT: Random weights produce
random (meaningless) distributions.
Trained models learn meaningful
next-token predictions.
```

# Example 2: Causal Property Verification (CENTERPIECE)



Causal Property Verification: Future Tokens Are Invisible

Causal Property: [10,20,30,40] vs [10,20,30,99]
Positions 0-2 are IDENTICAL regardless of position 3

Causal Mask (L=8)
1 = attend, 0 = blocked (future)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| 3 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 |
| 4 | 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 5 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 7 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

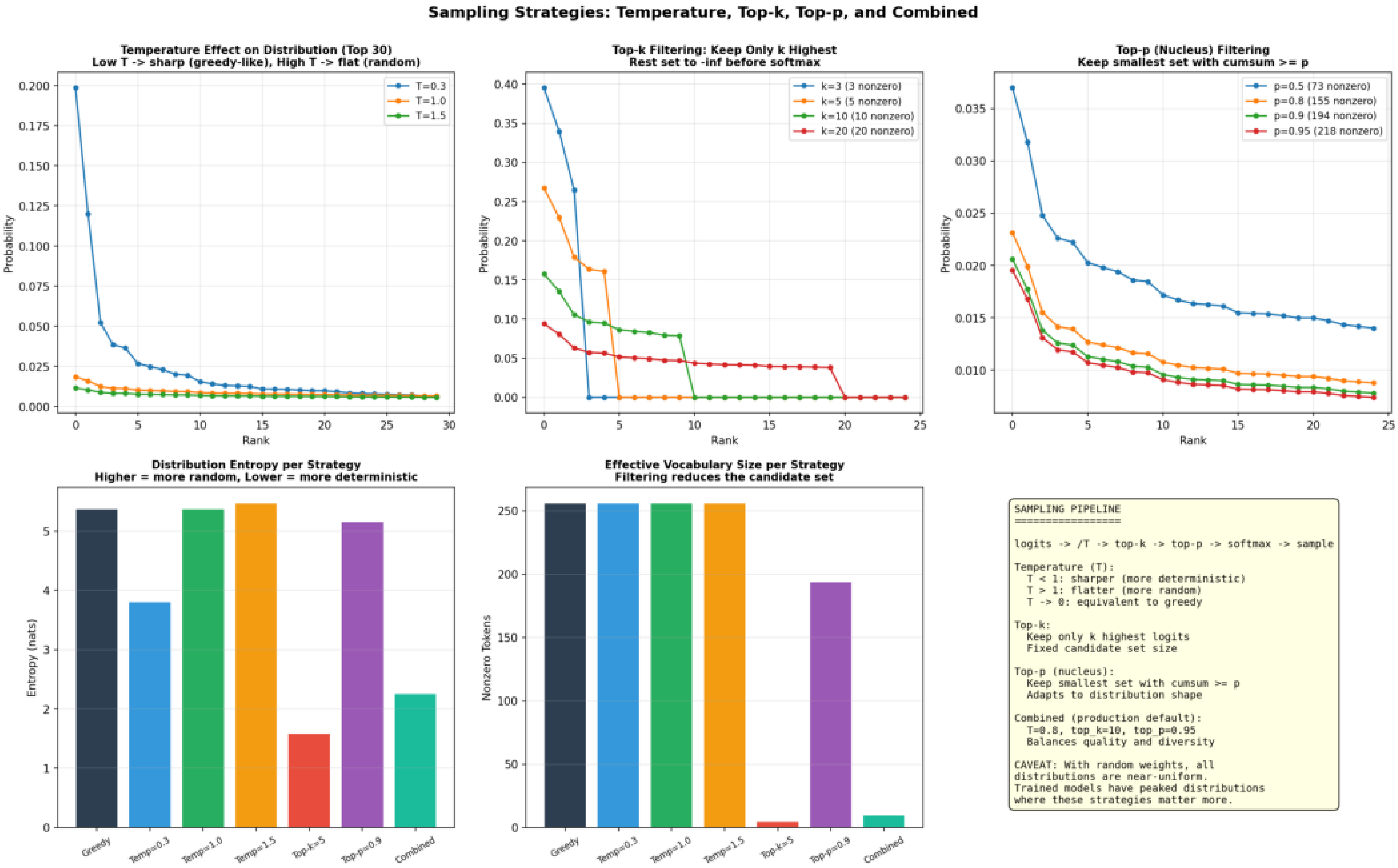
|P(A) - P(B)| Probability Difference
Only position 3 differs

Position 3 Next-Token Distributions Differ
Different input at pos 3 -> different prediction for pos 4

Position 2 Distributions are IDENTICAL
Bars overlap perfectly (future token invisible)

```
THE CAUSAL PROPERTY
===================

For sequences sharing prefix [0..k-1]:
  logits[0..k-1] are IDENTICAL
  regardless of tokens at positions k+.

WHY: The causal attention mask zeros
out all future positions. Position i
can only attend to positions 0..i.

CONSEQUENCE: We can generate left-
to-right, one token at a time.
Each new token depends only on the
prefix, never on future tokens.

This enables autoregressive generation:
  1. Forward pass on prompt
  2. Sample next token from last logits
  3. Append token, repeat

VERIFIED:
  Prefix positions: max diff = 0.0e+00
  Divergent position: diff = 7.1171
  Single vs full seq: diff = 6.6e-15
```

# Example 3: Sampling Strategy Comparison



Sampling Strategies: Temperature, Top-k, Top-p, and Combined

Temperature Effect on Distribution (Top 30)
Low T -> sharp (greedy-like), High T -> flat (random)

Top-k Filtering: Keep Only k Highest
Rest set to -inf before softmax

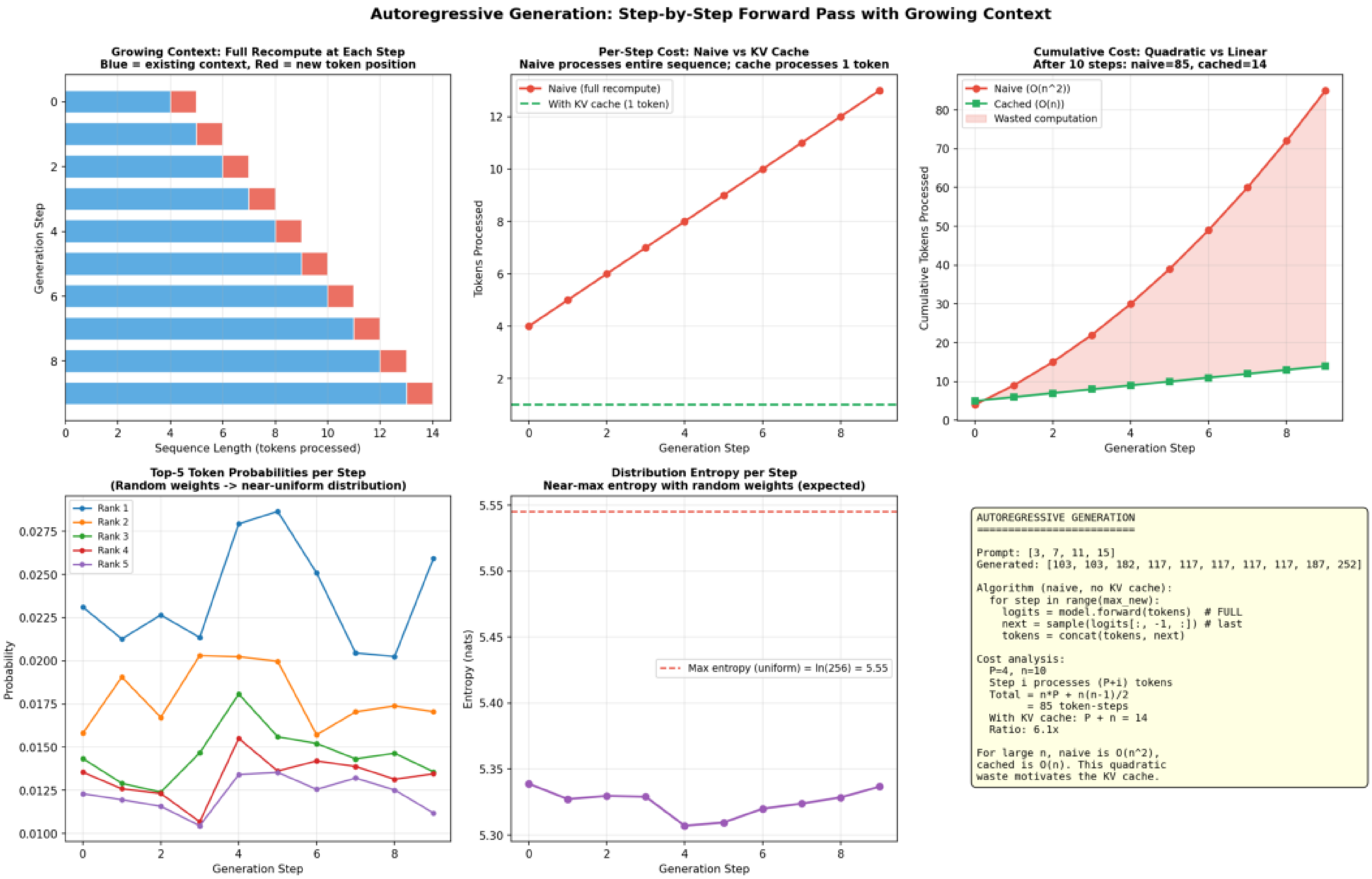
Top-p (Nucleus) Filtering
Keep smallest set with cumsum >= p

Distribution Entropy per Strategy
Higher = more random, Lower = more deterministic

Effective Vocabulary Size per Strategy
Filtering reduces the candidate set

```
SAMPLING PIPELINE
=================

logits -> /T -> top-k -> top-p -> softmax -> sample

Temperature (T):
  T < 1: sharper (more deterministic)
  T > 1: flatter (more random)
  T -> 0: equivalent to greedy

Top-k:
  Keep only k highest logits
  Fixed candidate set size

Top-p (nucleus):
  Keep smallest set with cumsum >= p
  Adapts to distribution shape

Combined (production default):
  T=0.8, top_k=10, top_p=0.95
  Balances quality and diversity

CAVEAT: With random weights, all
distributions are near-uniform.
Trained models have peaked distributions
where these strategies matter more.
```

# Example 4: Autoregressive Generation Loop



Autoregressive Generation: Step-by-Step Forward Pass with Growing Context

Growing Context: Full Recompute at Each Step
Blue = existing context, Red = new token position

Per-Step Cost: Naive vs KV Cache
Naive processes entire sequence; cache processes 1 token

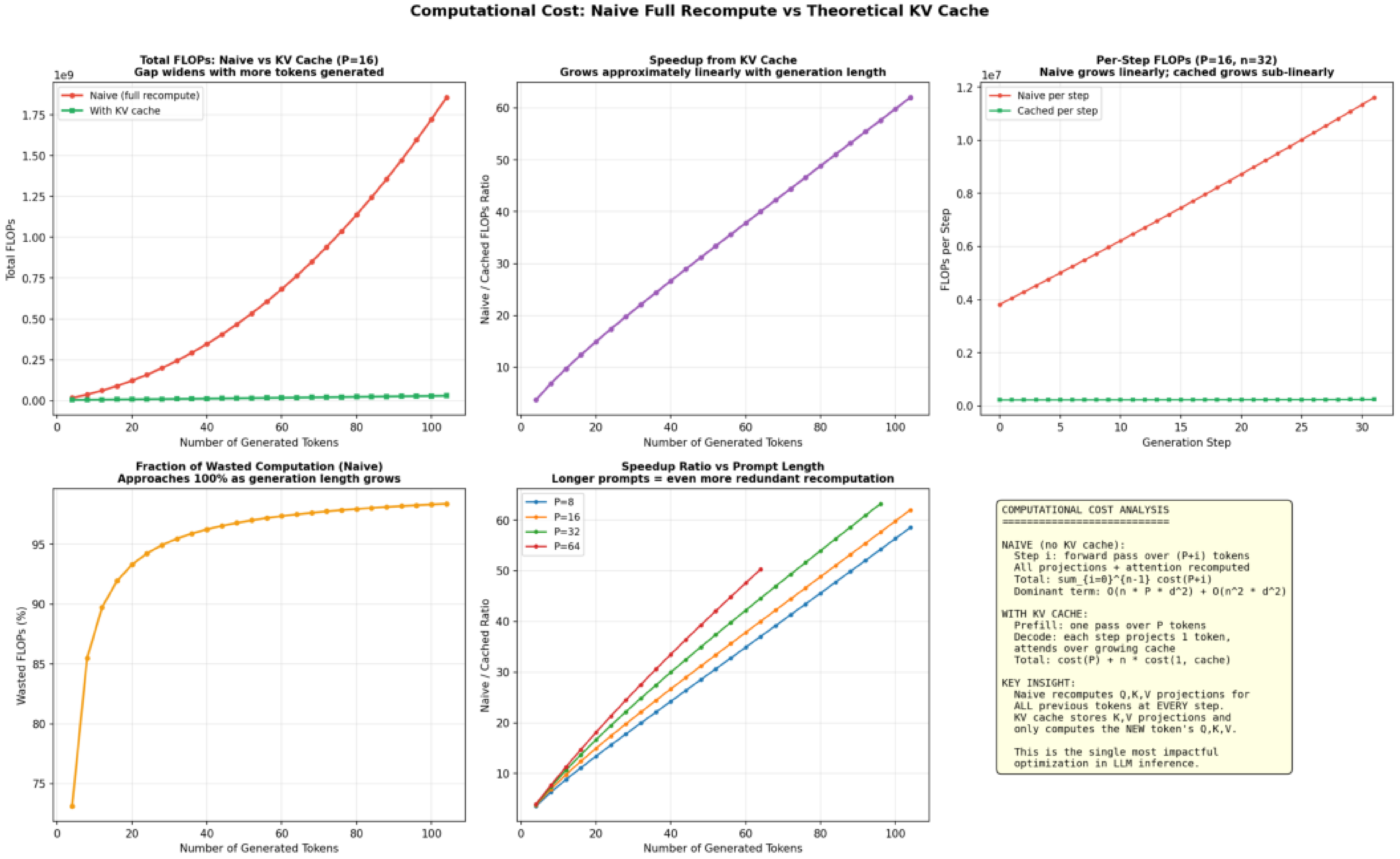
Cumulative Cost: Quadratic vs Linear
After 10 steps: naive=85, cached=14

Top-5 Token Probabilities per Step
(Random weights -> near-uniform distribution)

Distribution Entropy per Step
Near-max entropy with random weights (expected)

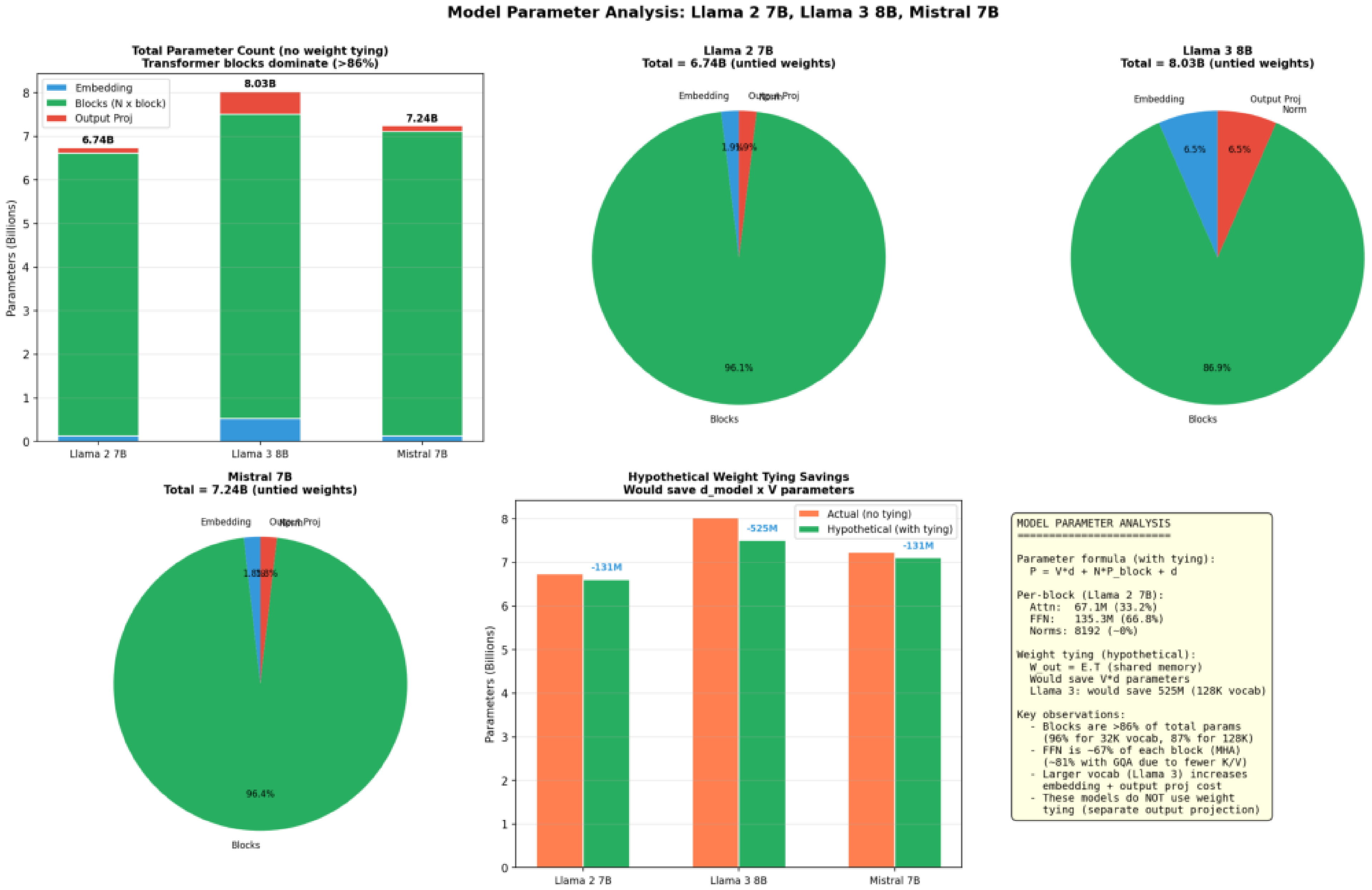
```
AUTOREGRESSIVE GENERATION
=========================

Prompt: [3, 7, 11, 15]
Generated: [103, 103, 182, 117, 117, 117, 117, 117, 187, 252]

Algorithm (naive, no KV cache):
  for step in range(max_new):
    logits = model.forward(tokens)  # FULL
    next = sample(logits[:, -1, :]) # last
    tokens = concat(tokens, next)

Cost analysis:
  P=4, n=10
  Step i processes (P+i) tokens
  Total = n*P + n(n-1)/2
        = 85 token-steps
  With KV cache: P + n = 14
  Ratio: 6.1x

For large n, naive is O(n^2),
cached is O(n). This quadratic
waste motivates the KV cache.
```

# Example 5: Computational Cost (Naive vs KV Cache)



```
COMPUTATIONAL COST ANALYSIS
============================

NAIVE (no KV cache):
  Step i: forward pass over (P+i) tokens
  All projections + attention recomputed
  Total: sum_{i=0}^{n-1} cost(P+i)
  Dominant term: O(n * P * d^2) + O(n^2 * d^2)

WITH KV CACHE:
  Prefill: one pass over P tokens
  Decode: each step projects 1 token,
  attends over growing cache
  Total: cost(P) + n * cost(1, cache)

KEY INSIGHT:
  Naive recomputes Q,K,V projections for
  ALL previous tokens at EVERY step.
  KV cache stores K,V projections and
  only computes the NEW token's Q,K,V.

  This is the single most impactful
  optimization in LLM inference.
```

# Example 6: Model Parameter Analysis



```
MODEL PARAMETER ANALYSIS
==============================

Parameter formula (with tying):
  P = V*d + N*P_block + d

Per-block (Llama 2 7B):
  Attn:  67.1M (33.2%)
  FFN:   135.3M (66.8%)
  Norms: 8192 (~0%)

Weight tying (hypothetical):
  W_out = E.T (shared memory)
  Would save V*d parameters
  Llama 3: would save 525M (128K vocab)

Key observations:
  - Blocks are >86% of total params
    (96% for 32K vocab, 87% for 128K)
  - FFN is ~67% of each block (MHA)
    (~81% with GQA due to fewer K/V)
  - Larger vocab (Llama 3) increases
    embedding + output proj cost
  - These models do NOT use weight
    tying (separate output projection)
```