Case Study: Real-Time Code Completion with Diffusion Language Models

Section 1: Industry Context and Business Problem

Industry: AI-Assisted Developer Tools

The AI code assistant market has grown from a niche curiosity into a \$15 billion industry segment. GitHub Copilot, Cursor, Codeium, and a growing roster of competitors have fundamentally changed how professional software engineers write code. Enterprise adoption has crossed the tipping point — over 60% of Fortune 500 engineering teams now use some form of AI-assisted code completion. The competitive dynamics are ruthless: developers will switch tools in a week if a competitor offers noticeably faster or more accurate suggestions.

Company Profile: Velocode

Velocode is a Series B developer tools startup headquartered in San Francisco, founded in 2022 by two former Google Brain researchers and a senior VS Code engineer. The company has raised \$45M across two rounds (Seed: \$8M from Founders Fund, Series A: \$12M, Series B: \$25M from Andreessen Horowitz). The team has grown to 120 people, with 65 ML engineers and researchers, 30 platform/infrastructure engineers, and 25 in sales, marketing, and operations.

Velocode's core product is **VeloComplete**, an AI-powered code completion plugin for VS Code, JetBrains, and Neovim. VeloComplete currently serves 800,000 monthly active developers, with 2,400 enterprise seats across 18 companies including three Fortune 500 firms. The product generates \$28M ARR, with enterprise contracts accounting for 70% of revenue.

VeloComplete is powered by **VeloLM-7B**, a 7-billion-parameter autoregressive Transformer trained on 500B tokens of permissively licensed code. The model supports Python, JavaScript/TypeScript, Java, Go, Rust, and C++.

Business Challenge

Velocode is losing enterprise deals — and the reason is latency.

VeloLM-7B runs on NVIDIA A100 GPUs behind a custom serving stack. Median end-to-end latency for a 50-token code completion is **350 milliseconds**. For longer completions (100-200 tokens, common when generating full function bodies), latency climbs to 700ms-1.4 seconds. Developers report that anything above 200ms feels sluggish, and above 500ms they start ignoring suggestions entirely.

Three specific problems are compounding the latency issue:

1. **Linear latency scaling.** Because VeloLM-7B is autoregressive, generation time scales linearly with output length. Generating 200 tokens takes roughly 4x longer than generating

50 tokens. There is no way around this within the autoregressive paradigm — each token must wait for the previous one.

2. **No native infilling.** Enterprise customers repeatedly request "fill-in-the-middle" (FIM) capabilities: the developer writes a function signature and return type, then asks the model to complete the body. VeloLM-7B handles this through a FIM hack — rearranging the prompt with special tokens — but quality degrades by 15-20% compared to standard left-to-right completion. The model was not designed for bidirectional context.

3. **Competitive pressure.** Two competitors have announced diffusion-based code models claiming sub-100ms latency for multi-line completions. Velocode's sales team reports that three enterprise prospects in the pipeline have paused negotiations, citing "waiting to evaluate faster alternatives." A lost enterprise cohort at this stage would cost an estimated \$4.2M in annual recurring revenue.

Why It Matters

- **Revenue at risk:** \$4.2M ARR from paused enterprise deals, with potential cascading loss of \$8-12M if the latency gap widens over the next two quarters.
- **Developer retention:** Internal analytics show that developers who experience >400ms median latency have a 45% higher monthly churn rate than those under 200ms. Every 100ms of latency reduction correlates with a 12% increase in suggestion acceptance rate.
- **Market positioning:** Velocode has positioned itself as the "performance-first" code assistant. Losing the speed crown would undermine the core brand promise and make the Series C raise (planned for Q3) significantly harder.

Constraints

- **Compute budget:** Velocode has a \$180K/month GPU budget for training (8x H100 node, reserved instance). Inference runs on a fleet of 40 A100-80GB GPUs across three regions.
- **Latency requirement:** p50 latency must be under 80ms for 50-token completions, under 150ms for 200-token completions. This is a hard requirement from the enterprise sales team.
- **Model size:** The new model must fit within 8GB of GPU memory at FP16 for single-GPU inference (to match current deployment density of 5 model replicas per A100-80GB).
- **Quality bar:** Code completion accuracy (measured by pass@1 on HumanEval) must not regress more than 5% from VeloLM-7B's current score of 48.2%.
- **Training timeline:** The first prototype must be ready in 8 weeks. Production-ready model in 16 weeks.
- **Data compliance:** Training data must be permissively licensed (Apache 2.0, MIT, BSD). No copyleft code. SOC 2 Type II compliance required for the serving infrastructure.
- **Language coverage:** Must support at minimum Python, JavaScript/TypeScript, and Java (the three highest-volume languages in VeloComplete usage data).

Section 2: Technical Problem Formulation

Problem Type: Conditional Sequence Generation via Discrete Diffusion

The core task is **conditional code generation**: given a context (surrounding code — prefix and optionally suffix), generate a code completion that is syntactically correct, semantically meaningful, and stylistically consistent with the codebase.

The natural first instinct is to frame this as autoregressive sequence generation — and indeed, this is what VeloLM-7B does. However, the business constraints expose fundamental limitations of the autoregressive framing:

- Autoregressive generation is inherently sequential (each token depends on the previous), so latency scales linearly with output length. No amount of engineering optimization can break this barrier.
- Autoregressive models process context unidirectionally (left-to-right), making fill-in-the-middle a workaround rather than a native capability.
- Autoregressive models commit to each token irreversibly, with no mechanism for self-correction.

Discrete diffusion — specifically, masked diffusion — resolves all three issues. By generating tokens in parallel through iterative unmasking, latency scales with the number of diffusion steps (a tunable hyperparameter, typically 8-16) rather than the output length. Bidirectional attention is built into the architecture. And the iterative refinement process naturally supports error correction across steps.

We therefore frame the problem as: **train a masked diffusion language model for code that can generate completions through iterative unmasking, achieving sub-80ms p50 latency while maintaining competitive code quality.**

Input Specification

The model receives a partially masked token sequence representing the code context and the completion region:

- **Prefix tokens:** The code before the cursor position, tokenized into subword units. These tokens are never masked — they provide left context.
- **Suffix tokens (optional):** The code after the cursor position, for fill-in-the-middle scenarios. Also never masked.
- **Completion region:** A fixed-length block of [MASK] tokens where the model will generate the completion. Length is set to a maximum of $L_{\max} = 256$ tokens.
- **Timestep embedding:** A scalar $t \in [0, 1]$ indicating the current noise level (fraction of tokens still masked in the completion region).

Formally, the input at diffusion step s is:

$$x_{\text{input}} = [\text{prefix};\ x_t^{(\text{completion})};\ \text{suffix}]$$

where $x_t^{(\text{completion})}$ is the partially unmasked completion region at noise level t.

The tokenizer is a Byte-Pair Encoding (BPE) tokenizer trained on the code corpus with a vocabulary size of 32,768 — small enough for efficient softmax computation but large enough to capture common code patterns (keywords, operators, indentation patterns, common variable names).

Output Specification

At each diffusion step, the model outputs a probability distribution over the vocabulary for every position in the completion region:

$$p_\theta(x_0^{(i)} \mid x_t) \in \mathbb{R}^V \quad \text{for each position } i \text{ in the completion region}$$

where $V = 32{,}768$ is the vocabulary size.

During generation, the model iteratively converts these distributions into concrete token predictions by: 1. Sampling from the predicted distribution (or taking the argmax for greedy decoding). 2. Computing a confidence score for each prediction (the maximum probability in the distribution). 3. Unmasking the top-k most confident predictions, where k is determined by the unmasking schedule.

The final output is a sequence of token IDs that, when decoded, produces the code completion string.

Why this output representation? The alternative would be to predict in a continuous embedding space (as in Diffusion-LM) and round to the nearest token. But rounding introduces errors that compound across the sequence — a problem especially severe for code, where a single wrong character (e.g., `=` vs `==`) changes semantics entirely. By predicting directly in discrete token space, we avoid rounding errors entirely.

Mathematical Foundation

To understand why masked diffusion works and why the training objective is sound, we need to build up from three foundational concepts.

1. The Forward Process as a Markov Chain on Discrete States

Each token position has $V+1$ possible states: one of the V vocabulary tokens, or the [MASK] token. The forward process defines a transition matrix Q_t that independently transitions each token:

$$q(x_t^{(i)} = m \mid x_{t-\Delta t}^{(i)} = v) = \frac{\Delta t}{1-(t-\Delta t)} \quad \text{(transition from token } v \text{ to [MAS}$$

$$q(x_t^{(i)} = v \mid x_{t-\Delta t}^{(i)} = v) = 1 - \tfrac{\Delta t}{1-(t-\Delta t)} \quad \text{(token stays unmasked)}$$

In the continuous-time limit, this simplifies to the marginal:

$$q(x_t^{(i)} \mid x_0^{(i)}) = \begin{cases} x_0^{(i)} & \text{with probability } 1-t \\ [\text{MASK}] & \text{with probability } t \end{cases}$$

This is an absorbing Markov chain — once a token becomes [MASK], it stays [MASK] (in the forward direction). At $t = 0$, the sequence is fully clean. At $t = 1$, the sequence is fully masked. The rate at which tokens are absorbed is uniform across positions, which is critical for the mathematical simplification we need.

2. The Evidence Lower Bound (ELBO) for Discrete Diffusion

We want to maximize $\log p_\theta(x_0)$, the log-likelihood of the training data under our model. Direct computation requires marginalizing over all possible noising trajectories, which is intractable. Instead, we derive a lower bound.

Starting from Jensen's inequality:

$$\log p_\theta(x_0) = \log \mathbb{E}_{q(x_{1:T}|x_0)}\left[\tfrac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}\right] \geq \mathbb{E}_q\left[\log \tfrac{p_\theta(x_{0:T})}{q(x_{1:T}|x_0)}\right]$$

For the masked diffusion process, the ELBO decomposes into a sum of terms, each of which measures how well the model predicts masked tokens at a given noise level. The key result from the MDLM paper is that, in continuous time, this ELBO simplifies to:

$$\text{ELBO} = -\mathbb{E}_{t\sim U(0,1)}\left[\frac{1}{1-t}\sum_{i:x_t^{(i)}=[\text{MASK}]} \log p_\theta(x_0^{(i)} \mid x_t)\right]$$

The factor $\frac{1}{1-t}$ is a time-dependent weight that upweights the loss at high noise levels (when t is close to 1 and most tokens are masked). This makes intuitive sense: predicting a token when almost everything is masked is harder and more informative, so it should contribute more to the training signal.

3. Simplification to Weighted Masked Language Modeling

The ELBO above is a weighted sum of cross-entropy losses on masked positions, where the weight depends on the masking ratio. In practice, training with uniform timestep sampling ($t \sim U(0,1)$) and unweighted cross-entropy loss works nearly as well, because the model sees a uniform distribution of noise levels and the weighting has minimal impact on the gradient signal.

This gives us the practical training objective:

$$\mathcal{L}(\theta) = \mathbb{E}_{t\sim U(0,1)}\,\mathbb{E}_{x_t\sim q(x_t|x_0)}\left[-\sum_{i:x_t^{(i)}=[\mathrm{MASK}]}\log p_\theta(x_0^{(i)} \mid x_t)\right]$$

This is exactly masked language modeling (as in BERT) with a randomly sampled masking ratio. The theoretical derivation from the ELBO confirms that this simple objective is a valid lower bound on the data log-likelihood — we are not just heuristically training a model, we are optimizing a principled generative objective.

Loss Function

The training loss has a single primary term, but understanding its behavior requires examining several aspects:

Primary term — Cross-entropy on masked positions:

$$\mathcal{L}_{\mathrm{MLM}} = -\sum_{i:x_t^{(i)}=[\mathrm{MASK}]}\log p_\theta(x_0^{(i)} \mid x_t)$$

This term penalizes the model for assigning low probability to the correct token at each masked position. If the model predicts the correct token with probability 1.0 at every position, the loss is 0. In practice, the loss converges to a value determined by the inherent ambiguity of code (multiple valid completions exist for any given context).

What if we remove this term? The model would have no training signal at all — this is the only loss term. There are no auxiliary losses, no regularization terms, no adversarial components. The simplicity of the objective is a feature, not a limitation.

Behavior across masking ratios:

- At low t (few tokens masked): The model has abundant context and the predictions are relatively easy. The loss per masked token is low, but there are few masked tokens to sum over.
- At high t (most tokens masked): Predictions are harder (less context), but there are many masked tokens. The total loss is dominated by these high-noise examples.
- The balance between these regimes is automatically handled by uniform sampling of t.

Optional auxiliary term — Length prediction loss (for variable-length generation):

$$\mathcal{L}_{\mathrm{length}} = -\log p_\phi(L \mid \mathrm{prefix}, \mathrm{suffix})$$

In production, we do not always know how many tokens the completion should be. A small auxiliary head can predict the completion length, allowing the system to allocate the right number of [MASK] tokens before generation begins. This term is weighted by $\lambda = 0.1$ relative to the main loss.

Total loss:

$$\mathcal{L} = \mathcal{L}_{\mathrm{MLM}} + \lambda\,\mathcal{L}_{\mathrm{length}}$$

Evaluation Metrics

Metric	Definition	Target	Rationale
pass@1 (HumanEval)	Fraction of problems solved on first attempt	$\geq 45.8\%$ (within 5% of 48.2%)	Primary quality metric; directly measures functional correctness
Exact Match (EM)	Fraction of completions that exactly match the ground truth	$\geq 35\%$	Measures precision of common completions
Edit Similarity	Normalized Levenshtein distance between prediction and ground truth	≥ 0.72	Captures partial credit for near-correct completions
p50 Latency	Median end-to-end latency for 50-token completion	$\leq 80\text{ms}$	Hard business requirement
p99 Latency	99th percentile latency	$\leq 200\text{ms}$	Ensures consistent user experience
FIM Accuracy	pass@1 on fill-in-the-middle benchmarks	$\geq 40\%$	Validates bidirectional context capability
Tokens/ second	Generation throughput on a single A100	≥ 800	Must demonstrate significant speedup over AR baseline

Baseline: Autoregressive VeloLM-7B

The current production model, VeloLM-7B, is a 7B-parameter autoregressive Transformer trained with standard next-token prediction. It achieves:

- pass@1 on HumanEval: 48.2%
- p50 latency (50 tokens): 350ms
- p50 latency (200 tokens): 1.2 seconds
- FIM accuracy (with prefix-suffix-middle rearrangement): 38%
- Throughput: ~140 tokens/second on A100

VeloLM-7B represents a strong but fundamentally constrained baseline. Its quality is competitive, but its latency and infilling limitations are architectural — no amount of systems optimization (quantization, speculative decoding, KV-cache tuning) can achieve the 80ms target for multi-token completions within the autoregressive paradigm.

Why Masked Diffusion Is the Right Approach

The choice of masked diffusion over the autoregressive paradigm — and over alternative diffusion formulations — is grounded in three fundamental properties:

1. **Parallel generation breaks the latency barrier.** In masked diffusion, the number of forward passes equals the number of diffusion steps (typically 8-16), regardless of output length. Generating 50 tokens and 200 tokens takes the same number of steps. Each forward

pass processes all tokens in parallel, and modern GPU hardware is optimized for exactly this kind of batched parallel computation. With 10 diffusion steps and a per-step latency of ~8ms on an A100, the total generation time is ~80ms — matching our target.

2. **Bidirectional attention is architecturally native.** The model uses a standard Transformer encoder (like BERT), where every token attends to every other token. Fill-in-the-middle is not a special mode — it is the default behavior. The prefix and suffix tokens provide fixed context, and the model naturally conditions on both when predicting the masked completion region. This eliminates the quality degradation observed with autoregressive FIM hacks.

3. **Masked diffusion is simpler than alternatives.** Compared to continuous-embedding diffusion (which requires embedding and rounding), masked diffusion operates directly in token space — no approximation errors. Compared to score-based discrete diffusion (SEDD), masked diffusion uses a simpler training objective (cross-entropy vs. score matching) and a more interpretable generation process. For a production system that must be debugged, monitored, and maintained by an engineering team, simplicity is a significant advantage.

Technical Constraints

- **Model size:** Target 1.5B parameters (fits in 3GB at FP16, well within the 8GB budget with room for KV-cache and activations). This is 4.7x smaller than VeloLM-7B, but the diffusion architecture compensates through parallel generation.
- **Inference latency budget:** 80ms total = ~8ms per diffusion step x 10 steps. Each step requires one full forward pass through the 1.5B-parameter Transformer.
- **Training compute:** 8x H100 for 8 weeks. At ~2.5 PFLOPS effective throughput, this yields approximately 8×10^{20} FLOPs — sufficient to train a 1.5B model on ~100B tokens with 3 epochs.
- **Sequence length:** Maximum context of 2048 tokens (prefix + completion + suffix). The completion region is at most 256 tokens.
- **Diffusion steps at inference:** 8-16 steps. Fewer steps increase speed but reduce quality. The optimal tradeoff will be determined empirically.

Section 3: Implementation Notebook Structure

This section provides the detailed structure for a hands-on Google Colab notebook where the student implements a masked diffusion model for code completion. The notebook progresses from data loading through training to generation and evaluation.

3.1 Data Acquisition Strategy

Dataset: CodeSearchNet (Husain et al., 2019) — a curated dataset of 2 million functions extracted from open-source GitHub repositories across six programming languages. We will use the Python subset (~450K functions) for this case study, as it provides a large, realistic corpus of well-structured code.

Why CodeSearchNet? It contains real, production-quality code from popular GitHub repositories. Each example is a complete function with its docstring, providing natural prefix-suffix pairs for fill-in-the-middle training. The dataset is freely available via HuggingFace Datasets and is permissively licensed for research use.

Preprocessing pipeline: 1. Load the Python subset from HuggingFace Datasets 2. Filter functions by length (keep functions with 50-512 tokens after BPE tokenization) 3. Train a BPE tokenizer on the corpus with vocabulary size 8,192 (smaller than production to fit Colab constraints) 4. Tokenize all functions and store as tensors

TODO: Students implement data augmentation

```
def augment_code_sample(tokens: list[int], mask_token_id: int) -> dict:
    """
    Create a training sample by randomly selecting a contiguous span
    within the token sequence to serve as the 'completion region.'

    The function should:
    1. Randomly select a span start position and span length
       (span length between 10% and 50% of the sequence length)
    2. Split the sequence into prefix, completion, and suffix
    3. Return a dictionary with keys:
       - 'prefix': tokens before the span
       - 'completion': the original tokens in the span (ground truth)
       - 'suffix': tokens after the span
       - 'full_sequence': the complete token sequence

    Hint: Use random.randint for start position. The span length
    should be sampled uniformly between 0.1 * seq_len and 0.5 * seq_len.
    Make sure the span does not extend beyond the sequence.

    Args:
        tokens: List of token IDs for a complete function
        mask_token_id: The ID of the [MASK] token

    Returns:
        Dictionary with prefix, completion, suffix, and full_sequence
    """
    # TODO: Implement this function
    pass
```

3.2 Exploratory Data Analysis

Key distributions to analyze: - Token sequence length distribution (histogram) - Vocabulary frequency distribution (log-scale, to verify Zipf's law holds for code tokens) - Distribution of function lengths in lines of code - Most common tokens (verify that code-specific tokens like `def`, `return`, `self` dominate) - Proportion of whitespace/indentation tokens (important for code quality)

Anomalies to investigate: - Functions that are extremely short (<10 tokens) or extremely long (>1000 tokens) - Functions with unusual character distributions (e.g., auto-generated code, data literals)

TODO: Students write EDA code and answer guided questions

```
def plot_token_length_distribution(dataset, tokenizer, max_length: int = 1024) -> None:
    """
    Plot a histogram of token sequence lengths for all functions in the dataset.

    Steps:
    1. Tokenize each function in the dataset using the provided tokenizer
    2. Record the length (number of tokens) for each function
    3. Plot a histogram with 50 bins, x-axis 'Token count', y-axis 'Number of functions'
    4. Add vertical lines at the 25th, 50th, and 75th percentiles
    5. Print the mean, median, and standard deviation of token lengths

    Hint: Use matplotlib for plotting. Use numpy for percentile calculations.

    Args:
        dataset: HuggingFace dataset with 'func_code_string' field
        tokenizer: Trained BPE tokenizer
        max_length: Maximum length to display on x-axis
    """
    # TODO: Implement this function
    pass
```

Thought questions: 1. Why does the token length distribution have a long right tail? What does this imply for choosing a maximum sequence length? 2. If 5% of your functions exceed 512 tokens, what are the tradeoffs between truncating them, splitting them, and discarding them? 3. How does the vocabulary frequency distribution for code compare to natural language? What does this tell you about the information density of code?

3.3 Baseline Approach

Baseline: Bigram Language Model

Before building the diffusion model, implement a simple bigram model as a baseline. The bigram model predicts each token based only on the immediately preceding token — no attention, no neural network, just a lookup table of conditional probabilities.

This baseline is deliberately weak. Its purpose is to establish a performance floor and to make the student appreciate what the diffusion model learns.

TODO: Students implement and evaluate the baseline

```
class BigramCodeModel:
    """
    A bigram language model for code completion.

    The model estimates P(token_i | token_{i-1}) from the training corpus
    using simple counting with add-k smoothing.
    """

    def __init__(self, vocab_size: int, smoothing: float = 0.01):
        """
        Initialize the bigram count matrix.

        Args:
            vocab_size: Size of the token vocabulary
            smoothing: Laplace smoothing parameter (add-k)
        """
        # TODO: Initialize a (vocab_size x vocab_size) count matrix
        # and a total count vector
        pass

    def fit(self, token_sequences: list[list[int]]) -> None:
        """
        Fit the bigram model by counting token pairs in the training data.
        Steps:
        1. For each sequence, iterate over consecutive token pairs (t_{i-1}, t_i)
        2. Increment the count matrix at position [t_{i-1}, t_i]
        3. After counting, convert counts to probabilities using add-k smoothing:
           P(t_i | t_{i-1}) = (count[t_{i-1}, t_i] + k) / (total[t_{i-1}] + k * V)

        Args:
            token_sequences: List of tokenized code sequences
        """
        # TODO: Implement bigram counting and probability estimation
        pass

    def predict_next(self, context_token: int) -> np.ndarray:
        """
        Predict the probability distribution over next tokens given a context token.

        Args:
            context_token: The preceding token ID

        Returns:
            Probability distribution over vocabulary (shape: [vocab_size])
        """
        # TODO: Return the row of the probability matrix for the context token
        pass

    def evaluate_perplexity(self, token_sequences: list[list[int]]) -> float:
        """
        Compute perplexity of the model on a set of sequences.

        Perplexity = exp(-1/N * sum(log P(t_i | t_{i-1})))
        where N is the total number of predicted tokens.

        Hint: Use numpy for log calculations. Handle the case where
        a probability is very small (add a floor of 1e-10 to avoid log(0)).

        Args:
            token_sequences: List of tokenized code sequences

        Returns:
            Perplexity (float). Lower is better.
        """
        # TODO: Implement perplexity calculation
        pass
```

Verification cell:

```
# After implementing, run this to verify
bigram = BigramCodeModel(vocab_size=tokenizer.vocab_size)
bigram.fit(train_token_sequences)
train_ppl = bigram.evaluate_perplexity(train_token_sequences[:100])
val_ppl = bigram.evaluate_perplexity(val_token_sequences[:100])
print(f"Train perplexity: {train_ppl:.1f}")
print(f"Validation perplexity: {val_ppl:.1f}")
assert train_ppl < val_ppl, "Train perplexity should be lower than validation"
assert val_ppl < 10000, "Perplexity seems too high — check your implementation"
print("Baseline implementation verified.")
```

3.4 Model Design

Architecture: Bidirectional Transformer with Timestep Conditioning

The model is a standard Transformer encoder (not decoder) with the following components:

1. **Token embedding layer:** Maps token IDs to d-dimensional vectors.
2. **Positional embedding:** Learned absolute position embeddings for sequence positions 0 through $L_{\max}$.

3. **Timestep embedding:** A small MLP that maps the scalar timestep t to a d-dimensional vector, added to every position's representation. This tells the model the current noise level.
4. **Transformer encoder blocks:** N layers of multi-head self-attention + feed-forward network. Critically, there is **no causal mask** — every position attends to every other position.
5. **Output projection:** A linear layer mapping from d-dimensional hidden states to vocabulary logits.

Key design decisions:

- **Why a Transformer encoder (not decoder)?** The encoder uses bidirectional self-attention, allowing every token to attend to every other token. This is essential for diffusion — the model must use both prefix and suffix context when predicting masked tokens.

- **Why add the timestep to every position?** The model needs to know the noise level to calibrate its predictions. At high noise (many masks), the model should make conservative, high-frequency predictions. At low noise (few masks), it should make precise, context-specific predictions. Without the timestep signal, the model cannot distinguish these regimes.

- **Why learned positional embeddings (not rotary)?** For the notebook exercise, learned embeddings are simpler to implement. In production, rotary positional embeddings (RoPE) would be preferred for their better length generalization.

Model hyperparameters (sized for Colab T4 GPU): - $d_{\text{model}} = 256$ - $n_{\text{heads}} = 8$ (head dimension = 32) - $n_{\text{layers}} = 6$ - $d_{\text{ff}} = 1024$ - Vocabulary size: 8,192 - Max sequence length: 512 - Total parameters: ~15M

TODO: Students implement the core model

```
import torch
import torch.nn as nn
import math


class TimestepEmbedding(nn.Module):
    """
    Embeds a scalar timestep t into a d_model-dimensional vector using
    a small MLP with sinusoidal features.

    Architecture:
    1. Map t -> sinusoidal features (like positional encoding, but for time)
    2. Linear(d_model, d_model) -> SiLU -> Linear(d_model, d_model)

    The sinusoidal features ensure the model can distinguish nearby timesteps.
    """

    def __init__(self, d_model: int):
        super().__init__()
        # TODO: Initialize the sinusoidal frequency table and MLP layers
        # Hint: Use torch.exp(torch.arange(0, d_model, 2) * -(math.log(10000.0) / d_model))
        # for the frequency table. The MLP should be:
        # Linear(d_model, d_model) -> SiLU -> Linear(d_model, d_model)
        pass

    def forward(self, t: torch.Tensor) -> torch.Tensor:
        """
        Args:
            t: Timestep values, shape (batch_size, 1)
        Returns:
            Timestep embeddings, shape (batch_size, d_model)
        """
        # TODO: Compute sinusoidal features from t, then pass through MLP
        # Step 1: t * frequencies -> shape (batch_size, d_model // 2)
        # Step 2: Concatenate sin and cos -> shape (batch_size, d_model)
        # Step 3: Pass through MLP
        pass


class DiffusionCodeLM(nn.Module):
    """
    A masked diffusion language model for code completion.

    Architecture:
    - Token embedding + positional embedding + timestep embedding
    - N Transformer encoder layers (bidirectional attention)
    - Linear output head to vocabulary logits
    """

    def __init__(
        self,
        vocab_size: int,
        d_model: int = 256,
        n_heads: int = 8,
        n_layers: int = 6,
        d_ff: int = 1024,
        max_seq_len: int = 512,
        dropout: float = 0.1,
    ):
        super().__init__()
        # TODO: Initialize all layers:
        # 1. self.token_embed = nn.Embedding(vocab_size, d_model)
        # 2. self.pos_embed = nn.Embedding(max_seq_len, d_model)
        # 3. self.time_embed = TimestepEmbedding(d_model)
        # 4. self.transformer = nn.TransformerEncoder(...)
        #    Use nn.TransformerEncoderLayer with d_model, n_heads, d_ff,
        #    dropout, batch_first=True, and norm_first=True (Pre-LN)
        # 5. self.output_head = nn.Linear(d_model, vocab_size)
        # 6. self.dropout = nn.Dropout(dropout)
        #
        # Hint: Do NOT pass a mask to the TransformerEncoder — we want
        # bidirectional attention (every token sees every other token).
        pass

    def forward(self, x_t: torch.Tensor, t: torch.Tensor) -> torch.Tensor:
        """
        Forward pass of the diffusion model.

        Args:
            x_t: Partially masked token IDs, shape (batch_size, seq_len)
            t: Timestep, shape (batch_size, 1)

        Returns:
            Logits over vocabulary, shape (batch_size, seq_len, vocab_size)

        Steps:
        1. Compute token embeddings from x_t
        2. Add positional embeddings (positions 0, 1, ..., seq_len-1)
        3. Compute timestep embedding from t and ADD it to every position
        4. Apply dropout
        5. Pass through the Transformer encoder (no mask argument!)
        6. Project to vocabulary logits via the output head
        """
        # TODO: Implement the forward pass following the steps above
        pass
```

Verification cell:

```
# Test the model with dummy inputs
model = DiffusionCodeLM(vocab_size=8192)
dummy_tokens = torch.randint(0, 8192, (4, 128))  # batch=4, seq_len=128
dummy_t = torch.rand(4, 1)
logits = model(dummy_tokens, dummy_t)
assert logits.shape == (4, 128, 8192), f"Expected (4, 128, 8192), got {logits.shape}"
```

```
print(f"Model parameters: {sum(p.numel() for p in model.parameters()):,}")
print("Model architecture verified.")
```

3.5 Training Strategy

Optimizer: AdamW with weight decay 0.01. AdamW decouples weight decay from the gradient update, which is important for Transformers where the loss landscape has many sharp local minima. Standard Adam without decoupling tends to under-regularize large models.

Learning rate schedule: Linear warmup over 1,000 steps to peak LR of 3×10^{-4} , followed by cosine decay to 1×10^{-5} . The warmup prevents early training instability (large gradients before the model has learned basic token statistics). Cosine decay provides a smooth annealing that empirically improves final model quality compared to step-wise decay.

Regularization: Dropout of 0.1 on attention and feed-forward layers. No label smoothing (the cross-entropy loss on masked positions already provides sufficient regularization through the varying masking ratio).

Training loop structure: 1. Sample a batch of tokenized code sequences 2. For each sequence, create a prefix-completion-suffix split (using the augmentation function from 3.1) 3. Sample a random timestep $t \sim U(0, 1)$ for each example 4. Apply the forward process: mask tokens in the completion region with probability t 5. Run the model forward to get logits 6. Compute cross-entropy loss only on the masked positions in the completion region 7. Backpropagate and update parameters

TODO: Students implement the training loop

```
def forward_process(x_completion: torch.Tensor, t: torch.Tensor, mask_token_id: int) -> tuple:
    """
    Apply the forward (masking) process to the completion region.

    For each token in x_completion, independently mask it with probability t.

    Args:
        x_completion: Clean tokens in the completion region, shape (batch_size, comp_len)
        t: Masking probability for each example, shape (batch_size, 1)
        mask_token_id: ID of the [MASK] token

    Returns:
        x_t: Masked tokens, shape (batch_size, comp_len)
        mask: Boolean mask indicating which positions were masked, shape (batch_size, comp_len)

    Hint: Generate random values with torch.rand_like(x_completion.float()).
    Compare with t (broadcasted) to create the boolean mask.
    Then clone x_completion and set masked positions to mask_token_id.
    """
    # TODO: Implement the forward masking process
    pass


def train_one_epoch(model, dataloader, optimizer, scheduler, mask_token_id: int, device: str) -> float:
    """
    Train the diffusion model for one epoch.

    For each batch:
    1. Move data to device
    2. Sample random timesteps t ~ U(0, 1), shape (batch_size, 1)
    3. Apply forward_process to get masked sequences and the boolean mask
    4. Concatenate [prefix, masked_completion, suffix] to form the full input
    5. Run model forward: logits = model(full_input, t)
    6. Extract logits only at the masked completion positions
    7. Compute cross-entropy loss between predicted logits and true tokens
    8. Backpropagate and step optimizer and scheduler

    Args:
        model: DiffusionCodeLM instance
        dataloader: DataLoader yielding batches of (prefix, completion, suffix) tuples
        optimizer: AdamW optimizer
        scheduler: Learning rate scheduler
        mask_token_id: ID of the [MASK] token
        device: 'cuda' or 'cpu'

    Returns:
        Average loss for the epoch (float)

    Hint: For step 6, you need to figure out which positions in the full
    concatenated sequence correspond to masked completion tokens. Use the
    mask from forward_process, but offset by the prefix length.
    """
    # TODO: Implement the training loop
    pass
```

Verification cell:

```
# Run one epoch on a small subset and verify loss decreases
model = DiffusionCodeLM(vocab_size=tokenizer.vocab_size).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=3e-4, weight_decay=0.01)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=100)

loss1 = train_one_epoch(model, small_dataloader, optimizer, scheduler, MASK_TOKEN_ID, device)
loss2 = train_one_epoch(model, small_dataloader, optimizer, scheduler, MASK_TOKEN_ID, device)
print(f"Epoch 1 loss: {loss1:.4f}")
print(f"Epoch 2 loss: {loss2:.4f}")
assert loss2 < loss1, "Loss should decrease between epochs — check your training loop"
print("Training loop verified.")
```

3.6 Evaluation

Quantitative evaluation on held-out test set: - Compute cross-entropy loss on the test set at multiple masking ratios ($t = 0.25, 0.5, 0.75$) - Generate completions for test functions and compute exact match, edit similarity - Compare all metrics against the bigram baseline

TODO: Students generate evaluation plots and interpret results

```
def evaluate_model(
    model: nn.Module,
    test_dataloader,
    mask_token_id: int,
    device: str,
    masking_ratios: list[float] = [0.25, 0.5, 0.75],
) -> dict:
    """
    Evaluate the diffusion model on the test set.

    For each masking ratio in masking_ratios:
    1. Apply forward_process with the fixed masking ratio (not random)
    2. Compute cross-entropy loss on masked positions
    3. Also compute token-level accuracy: what fraction of masked tokens
       does the model predict correctly (using argmax)?

    Additionally, for the full test set:
    4. Run the full iterative unmasking generation procedure
    5. Compute exact match rate against ground truth
    6. Compute average edit similarity (1 - normalized Levenshtein distance)

    Args:
        model: Trained DiffusionCodeLM
        test_dataloader: DataLoader for the test set
        mask_token_id: ID of the [MASK] token
        device: 'cuda' or 'cpu'
        masking_ratios: List of masking ratios to evaluate at

    Returns:
        Dictionary with keys:
        - 'loss_by_ratio': {ratio: avg_loss} for each masking ratio
        - 'accuracy_by_ratio': {ratio: avg_accuracy} for each masking ratio
        - 'exact_match': float, fraction of exact matches
        - 'edit_similarity': float, average edit similarity

    Hint: For edit similarity, you can use the python-Levenshtein library
    or implement it with dynamic programming. Normalize by max(len(pred), len(truth)).
    """
    # TODO: Implement evaluation
    pass


def plot_evaluation_results(eval_results: dict, baseline_perplexity: float) -> None:
    """
    Create three plots:
    1. Loss vs masking ratio (bar chart) — shows how loss increases with more masking
    2. Token accuracy vs masking ratio (bar chart) — shows how accuracy drops with more masking
    3. Comparison table: diffusion model vs bigram baseline on all metrics

    Hint: Use matplotlib subplots with 1 row, 3 columns.
    """
    # TODO: Create the evaluation plots
    pass
```

3.7 Error Analysis

Systematic error categorization:

After generating completions, categorize errors into the following taxonomy:

1. **Syntax errors:** Missing colons, unmatched parentheses, incorrect indentation
2. **Semantic errors:** Variable used before definition, wrong function called, type mismatches
3. **Style errors:** Inconsistent naming conventions, unusual formatting
4. **Logic errors:** Correct syntax but wrong algorithm or control flow
5. **Repetition errors:** Token or pattern repetition (a known failure mode of diffusion models)

TODO: Students identify and categorize the top 3 failure modes

```
def analyze_errors(
    model: nn.Module,
    test_samples: list[dict],
    tokenizer,
    mask_token_id: int,
    device: str,
    num_steps: int = 10,
    num_samples: int = 50,
) -> dict:
    """
    Generate completions for test samples and categorize errors.

    For each test sample:
    1. Generate a completion using iterative unmasking
    2. Compare with the ground truth
    3. If they differ, categorize the error type:
       - 'syntax': Try to compile/parse the generated code with ast.parse().
         If it raises SyntaxError, it is a syntax error.
       - 'repetition': Check if any 3-gram appears more than 3 times
         in the generated code.
       - 'semantic': If syntactically correct but different from ground truth,
         classify as semantic.
    4. Collect examples of each error type.
    Args:
        model: Trained DiffusionCodeLM
        test_samples: List of dicts with 'prefix', 'completion', 'suffix'
        tokenizer: BPE tokenizer for decoding
        mask_token_id: [MASK] token ID
        device: 'cuda' or 'cpu'
        num_steps: Number of diffusion steps for generation
        num_samples: Number of samples to analyze

    Returns:
        Dictionary with:
        - 'error_counts': {error_type: count}
        - 'error_examples': {error_type: list of (generated, ground_truth) pairs}
        - 'total_errors': total number of incorrect completions
        - 'total_correct': total number of exact matches

    Hint: Use Python's ast module for syntax checking:
      try:
          ast.parse(generated_code)
          is_syntax_error = False
      except SyntaxError:
          is_syntax_error = True
    """
    # TODO: Implement error analysis
    pass
```

Thought questions: 1. Which error type is most common? Is this a fundamental limitation of the diffusion approach, or a training issue? 2. Does the model make more errors at the beginning or end of the completion region? What does this tell you about the unmasking order? 3. How might you modify the unmasking schedule to reduce the most common error type?

3.8 Scalability and Deployment Considerations

Production deployment analysis:

In production, the model would be served on NVIDIA A100 or H100 GPUs behind a load balancer. Key considerations:

- **Batched inference:** Unlike autoregressive models (where each request is at a different generation step), diffusion model requests can be batched at the same diffusion step, enabling higher GPU utilization.
- **Adaptive step count:** For simple completions (single token, common pattern), fewer diffusion steps suffice. A confidence-based early stopping criterion can reduce average latency.
- **KV-cache:** Standard KV-caching does not apply (no causal attention), but the prefix/suffix embeddings can be cached across diffusion steps since they do not change.

TODO: Students write inference benchmarking

```
def benchmark_inference(
    model: nn.Module,
    tokenizer,
    mask_token_id: int,
    device: str,
    completion_lengths: list[int] = [32, 64, 128, 256],
    num_steps_list: list[int] = [4, 8, 12, 16],
    num_trials: int = 20,
) -> dict:
    """
    Benchmark inference latency across different completion lengths and step counts.
    For each combination of (completion_length, num_steps):
    1. Create a dummy input with a fixed prefix (64 tokens) and the specified
       number of [MASK] tokens as the completion region
    2. Run the full iterative unmasking generation procedure
    3. Measure wall-clock time (use torch.cuda.synchronize() before timing!)
    4. Record the median latency over num_trials runs

    Also measure:
    5. Tokens per second = completion_length / median_latency
    6. GPU memory usage via torch.cuda.max_memory_allocated()

    Create a 2D heatmap (completion_length x num_steps) showing median latency.

    Args:
        model: Trained DiffusionCodeLM
        tokenizer: BPE tokenizer
        mask_token_id: [MASK] token ID
        device: 'cuda' (must be GPU for meaningful results)
        completion_lengths: List of completion region sizes to test
        num_steps_list: List of diffusion step counts to test
        num_trials: Number of timing trials per configuration

    Returns:
        Dictionary with:
        - 'latencies': 2D dict {comp_len: {num_steps: median_latency_ms}}
        - 'throughput': 2D dict {comp_len: {num_steps: tokens_per_second}}
        - 'memory_mb': peak GPU memory in megabytes

    Hint: For accurate GPU timing:
      torch.cuda.synchronize()
      start = time.time()
      # ... run generation ...
      torch.cuda.synchronize()
      elapsed = time.time() - start
    """
    # TODO: Implement inference benchmarking
    pass
```

Thought questions: 1. How does latency scale with completion length? Compare this to autoregressive scaling (linear). What do you observe? 2. What is the relationship between num_steps and generation quality? Is there a "sweet spot" where adding more steps gives diminishing returns? 3. If you needed to deploy this model on a T4 GPU (16GB, lower compute), what modifications would you make?

3.9 Ethical and Regulatory Analysis

Bias considerations for AI code completion:

- **Training data bias:** Code from GitHub over-represents certain programming styles, libraries, and patterns. Code completion models may reinforce these biases, making it harder for developers using less common patterns to get useful suggestions.
- **Security vulnerabilities:** The model may suggest code with security vulnerabilities (SQL injection, XSS, buffer overflows) if such patterns exist in the training data. This is especially concerning because developers may accept suggestions without careful review.
- **License compliance:** Generated code could inadvertently reproduce copyrighted code verbatim. For enterprise deployment, mechanisms to detect and filter memorized code are necessary.
- **Accessibility:** Code completions that default to inaccessible UI patterns (e.g., missing ARIA attributes) can propagate accessibility debt at scale.

TODO: Students write an ethical impact assessment

```
def ethical_assessment(
    model: nn.Module,
    tokenizer,
    mask_token_id: int,
    device: str,
    num_steps: int = 10,
) -> str:
    """
    Conduct a basic ethical assessment of the code completion model.

    Perform the following tests and return a written assessment:

    1. MEMORIZATION TEST: Generate 100 completions from the same prefix.
       Check if any two completions are identical. High duplication rate
       may indicate memorization of training data.

    2. VULNERABILITY TEST: Create 5 prompts that could lead to insecure code:
       - SQL query construction
       - HTML template rendering
       - File path handling
       - Password/credential handling
       - HTTP request construction
       For each, generate a completion and manually inspect whether the
       generated code contains common vulnerability patterns (string
       concatenation for SQL, no input sanitization for HTML, etc.)

    3. DIVERSITY TEST: Generate 20 completions for a function stub that
       could be solved multiple ways. Measure the diversity of solutions
       (e.g., number of unique approaches). Low diversity may indicate
       the model overfits to dominant patterns.

    Return a structured string report with:
    - Findings for each test
    - Risk level (Low/Medium/High) for each category
    - Recommended mitigations

    Args:
        model: Trained DiffusionCodeLM
        tokenizer: BPE tokenizer
        mask_token_id: [MASK] token ID
        device: 'cuda' or 'cpu'
        num_steps: Number of diffusion steps

    Returns:
        String containing the structured ethical assessment report
    """
    # TODO: Implement the ethical assessment
    # This is intentionally open-ended — there is no single right answer.
    # The goal is to develop a systematic process for evaluating ML models.
    pass
```

Thought questions: 1. If the model memorizes a block of GPL-licensed code and suggests it to a developer writing proprietary software, who is responsible? The model developer, the code completion company, or the end user? 2. How would you build a real-time filter that prevents the model from suggesting code with known vulnerability patterns? 3. What fairness metrics make sense for code completion? Is it fair if the model works better for Python than for Rust? What about for English variable names vs. non-English?

Section 4: Production and System Design Extension

This section is for advanced students who want to understand how the prototype from Section 3 would be scaled to a production system serving 800,000 developers.

Architecture Overview

The production system consists of five major components:

1. **Client Plugin** (VS Code / JetBrains / Neovim): Captures code context (prefix, suffix, cursor position), sends completion requests via HTTPS, and renders suggestions in the editor.
2. **API Gateway** (Kong / AWS API Gateway): Handles authentication, rate limiting (1,000 requests/minute per user), request routing, and TLS termination.
3. **Inference Service** (Kubernetes pods on GPU nodes): Runs the diffusion model. Each pod contains the model loaded on a single GPU with a custom serving runtime. Pods are replicated across three regions (us-east, eu-west, ap-southeast) for latency optimization.
4. **Context Service** (gRPC microservice): Preprocesses the code context — tokenization, prefix/suffix truncation to fit the model's context window, and caching of tokenized prefixes for repeat requests from the same file.
5. **Monitoring and Analytics** (Prometheus + Grafana + BigQuery): Tracks latency, throughput, error rates, suggestion acceptance rates, and model quality metrics.

API Design

Completion Request (REST):

```
POST /v1/completions
Content-Type: application/json
Authorization: Bearer <api_key>

{
    "prefix": "def fibonacci(n):\n    ",
    "suffix": "\n    return result",
    "language": "python",
    "max_tokens": 128,
    "num_steps": 10,
    "temperature": 0.8,
    "request_id": "uuid-v4"
}
```

Completion Response:

```
{
    "completion": "if n <= 1:\n        return n\n    a, b = 0, 1\n    for _ in range(2, n + 1):\n        a, b = b, a
    "tokens_generated": 42,
    "latency_ms": 67,
    "confidence": 0.84,
    "request_id": "uuid-v4",
    "model_version": "velodiff-1.5b-v2.3"
}
```

Streaming endpoint for long completions:

```
POST /v1/completions/stream
```

Returns Server-Sent Events (SSE) with partial completions after each diffusion step, allowing the UI to progressively reveal tokens as they are unmasked — similar to how autoregressive models stream token-by-token, but here each event reveals a batch of newly unmasked tokens.

Serving Infrastructure

- **GPU fleet:** 40x NVIDIA A100-80GB across three regions. Each GPU runs 5 model replicas (1.5B params at FP16 = ~3GB per replica, plus KV-cache and activations).
- **Serving framework:** Custom C++ inference runtime with CUDA graphs for the Transformer forward pass. CUDA graphs eliminate kernel launch overhead, which matters when each diffusion step is only ~8ms.
- **Autoscaling:** Horizontal pod autoscaler (HPA) based on GPU utilization. Scale from 40 to 120 GPUs during peak hours (9am-6pm in each region). Scale-up latency target: under 90 seconds.
- **Batching:** Dynamic batching with a 5ms collection window. Requests arriving within 5ms of each other are batched together for a single forward pass. Batch sizes of 16-32 are common during peak traffic.

Latency Budget

Component	Budget	Notes
Network (client to API gateway)	15ms	Varies by region; CDN edge reduces this
API gateway processing	2ms	Authentication, rate limiting
Context service (tokenization)	5ms	Cached for repeat requests from same file
Inference (10 diffusion steps)	55ms	~5.5ms per step with CUDA graphs and FP16
Response serialization	3ms	JSON encoding, compression
Total	**80ms**	Meets the p50 target

For p99, add 40ms for tail latency (cache miss in context service, batching delay, garbage collection), targeting 120ms.

Monitoring

Key metrics to track:

- **Latency percentiles:** p50, p90, p99 per region, per language. Alert if p50 > 100ms or p99 > 250ms.
- **Throughput:** Requests per second per GPU. Alert if below 80% of expected capacity.
- **Error rate:** 5xx responses, timeouts. Alert if error rate > 0.1%.

- **Suggestion acceptance rate:** Fraction of displayed suggestions that users accept (Tab key). This is the primary product quality signal. Alert if 7-day rolling average drops by more than 5% from baseline.
- **GPU utilization:** Target 60-75%. Below 50% indicates over-provisioning; above 85% indicates risk of latency spikes.
- **Model confidence distribution:** Histogram of confidence scores. A shift toward lower confidence may indicate distribution drift.

Dashboard layout: Three panels — (1) Real-time latency and throughput, (2) Daily acceptance rate by language, (3) Weekly model quality trends.

Model Drift Detection

Code evolves over time. New languages gain popularity, new frameworks emerge, and coding styles shift. The model must be monitored for distribution drift.

Detection approach:

1. **Input drift:** Track the distribution of token n-grams in incoming requests. Use a KL-divergence test against the training distribution, computed weekly. If KL > 0.1, flag for investigation.
2. **Output drift:** Monitor the distribution of model confidence scores. A gradual decrease in mean confidence suggests the model is encountering inputs it was not trained on.
3. **Quality drift:** Track suggestion acceptance rate as a proxy for quality. A statistically significant drop ($p < 0.01$ in a two-proportion z-test over a 7-day window) triggers an alert.
4. **Semantic drift:** Periodically run the model on a held-out benchmark (HumanEval, MBPP). If pass@1 drops by more than 2% from the deployment baseline, schedule a retraining run.

Response to drift: Retraining on fresh data every 8 weeks, with continuous fine-tuning on anonymized, high-quality user-accepted completions (with explicit opt-in consent).

Model Versioning

- **Naming convention:** `velodiff-{size}-v{major}.{minor}` (e.g., `velodiff-1.5b-v2.3`)
- **Artifact storage:** Model checkpoints stored in S3 with immutable versioning. Each version includes: model weights, tokenizer files, training config, evaluation results, and a provenance record (training data hash, code commit).
- **Rollback:** Any previous version can be deployed within 5 minutes by updating the Kubernetes deployment to point to the new S3 artifact. A canary deployment (5% traffic to new version) runs for 2 hours before full rollout.
- **Deprecation policy:** Maintain the two most recent major versions in production. Older versions are archived but available for emergency rollback.

A/B Testing

Framework: Each completion request includes a user ID that is deterministically hashed to an experiment bucket. Standard A/B tests run at 50/50 traffic split for 2 weeks.

Primary metric: Suggestion acceptance rate (binary outcome per suggestion).

Statistical requirements: - Minimum detectable effect (MDE): 1% absolute change in acceptance rate - Significance level: $\alpha = 0.05$ (two-sided) - Power: $1 - \beta = 0.80$ - Given a baseline acceptance rate of 28% and ~5M suggestions/day, the required sample size per arm is approximately 250,000 suggestions, achievable in under 3 hours of traffic.

Guardrail metrics: Latency p99, error rate, and user churn rate. If any guardrail metric degrades by more than 10% in the treatment group, the experiment is automatically halted.

CI/CD for ML

Training pipeline (orchestrated by Airflow): 1. **Data collection:** Weekly scrape of new permissively licensed code from GitHub (filtered by license and quality heuristics). 2. **Data validation:** Schema checks, deduplication against training set, toxicity filtering, license verification. 3. **Training:** 8x H100 node, triggered manually or on schedule. Training runs are tracked in Weights and Biases. 4. **Evaluation gate:** Automated evaluation on HumanEval, MBPP, and an internal benchmark of 500 real completion scenarios from VeloComplete logs (anonymized). If pass@1 drops by more than 2% on any benchmark, the pipeline halts and pages the ML team. 5. **Artifact packaging:** Successful models are packaged with their tokenizer and config into a versioned S3 artifact. 6. **Canary deployment:** Automatic canary with 5% traffic for 2 hours. 7. **Full rollout:** If canary metrics are within bounds, gradual rollout over 4 hours (5% -> 25% -> 50% -> 100%).

Cost Analysis

Training costs (one-time per model version): - 8x H100 reserved instance: \$180K/month - Training duration: ~3 weeks for 1.5B model on 100B tokens - Per-training-run cost: ~\$135K - Annual training cost (6 model versions): ~\$810K

Inference costs (ongoing): - 40x A100-80GB reserved instances: ~\$400K/month - Autoscaling peak (additional 80 GPUs for ~8 hours/day): ~\$200K/month - Network and infrastructure: ~\$50K/month - Total monthly inference cost: ~\$650K/month - Annual inference cost: ~\$7.8M

Cost per completion: - ~5M completions/day = ~150M completions/month - Cost per completion: \$650K / 150M = \$0.0043 (less than half a cent) - Revenue per completion (blended): \$28M ARR / (150M x 12) = \$0.0156 - Gross margin per completion: ~72%

Break-even analysis: The diffusion model investment (training + engineering) is estimated at \$2M. At the current margin, this is recovered in approximately 3 months of the recovered enterprise pipeline (\$4.2M ARR at risk).