Critic–Gate Score Distillation for Latent Diffusion on CIFAR-10

From-Scratch Implementation Plan with Heun Predictor–Corrector Sampling

November 16, 2025

1 Goal and Scope

This document specifies a complete, from-scratch implementation plan for critic–gate score distillation in a latent diffusion model (LDM) with a VAE encoder, targeting *unconditional* CIFAR-10 image generation.

Key constraints and goals:

- Dataset: CIFAR-10, images $x \in \mathbb{R}^{3\times 32\times 32}$.
- Latent space: VAE encoder produces latents $z \in \mathbb{R}^{C\times H\times W}$.
- Training distribution in latent space is the *aggregated posterior* $q(z) = \int q_\phi(z \mid x) p_{\text{data}}(x)\, dx$: we **always** use stochastic samples $z \sim q_\phi(z \mid x)$, not just the mean $\mu_\phi(x)$.
- Baseline: a standard VP latent diffusion model trained with an MSE loss on noise ϵ, using a cosine β schedule and Heun predictor–corrector (Heun-PC) sampling for the VP-SDE.
- Critic–gate variant: identical architecture, data, schedules, and sampler, but with a *critic–gate blended* target in the MSE loss.
- VAE: trained with a *small* β_{KL} regularizer. We explicitly *do not* want the latent to be fully Gaussian—we only need enough Gaussianization to prevent the empirical covariance from becoming numerically degenerate, because the critic–gate prior term uses $\Sigma^{-1}(z - \mu)$.
- Sampler: VP-SDE Heun predictor–corrector is the *primary* sampler for generation; we may implement DDPM/DDIM for sanity checks, but all comparison plots and metrics should use Heun-PC.
- Evaluation: FID on CIFAR-10, plus:
  - panels of generated images (baseline vs critic–gate);
  - 2D sample histograms of latent projections (e.g. first two PCA directions) comparing the empirical $q(z)$ and generated $p_\theta(z)$;
  - optionally, histograms of gate values $g(y, t)$ across time/ samples.

The intended audience is another code-focused LLM (e.g. Cursor/Codex) that can implement the described system directly from this specification.

2 Mathematical Framework

2.1 VAE Latent Space and Aggregated Posterior

CIFAR-10 images $x \in \mathbb{R}^{3\times 32\times 32}$ are normalized to $[-1, 1]$. A convolutional VAE encoder $\mathcal{E}_\phi$ maps x to per-pixel Gaussian parameters $(\mu_\phi(x), \log \sigma_\phi^2(x))$:

$$q_\phi(z \mid x) = \mathcal{N}\big(z;\, \mu_\phi(x), \mathrm{diag}(\sigma_\phi^2(x))\big),$$

with $z \in \mathbb{R}^{C\times H\times W}$ (we will set $C = 4$, $H = W = 8$ later).

We always train diffusion on samples from the *aggregated posterior*:

$$z_0 = \mu_\phi(x) + \sigma_\phi(x) \odot \epsilon_0, \qquad \epsilon_0 \sim \mathcal{N}(0, I), \quad x \sim p_{\text{data}}. \tag{1}$$

We do **not** collapse to the mean manifold $\mu_\phi(x)$.

2.2 Forward VP Diffusion and VP-SDE

We use the standard variance-preserving (VP) diffusion in latent space. In discrete DDPM form, the forward noising process is:

$$z_t = \sqrt{\bar\alpha_t}\, z_0 + \sqrt{1 - \bar\alpha_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I), \quad t \in \{0, \dots, T-1\}, \tag{2}$$

with a β-schedule $\{\beta_t\}_{t=0}^{T-1}$ and

$$\alpha_t = 1 - \beta_t, \qquad \bar\alpha_t = \prod_{k=0}^{t} \alpha_k.$$

For training-time forward diffusion, we sample t uniformly from $\{0, \dots, T-1\}$, sample $\epsilon \sim \mathcal{N}(0, I)$, and use (2).

For continuous-time analysis and Heun-PC sampling, we interpret this as a VP SDE in Itô form:

$$dZ_\tau = -\frac{1}{2}\beta(\tau) Z_\tau\, d\tau + \sqrt{\beta(\tau)}\, dW_\tau, \qquad \tau \in [0, 1], \tag{3}$$

with scalar noise schedule $\beta(\tau) \geq 0$. We will discretize this SDE with Heun predictor–corrector (see Section 5.5), using a continuous-time parameterization of the score network $s_\theta(z, \tau) \approx \nabla_z \log p_\tau(z)$.

2.3 Cosine β-Schedule without Symbol Confusion

We use an improved cosine schedule as in Nichol & Dhariwal (2021), but to avoid confusion with the score notation $s(\cdot)$ we will *not* use the traditional symbol s_0 for the offset. Instead we use a distinct scalar offset $c_{\text{off}} > 0$.

Define for $\tau \in [0, 1]$:

$$\bar\alpha(\tau) = \frac{f(\tau)}{f(0)}, \qquad f(\tau) = \cos^2\Big(\frac{\pi}{2} \frac{\tau + c_{\text{off}}}{1 + c_{\text{off}}}\Big), \tag{4}$$

with e.g. $c_{\text{off}} = 0.008$ (typical from the literature). Then discretize by

$$\bar\alpha_t = \bar\alpha\Big(\frac{t+1}{T+1}\Big), \quad \alpha_t = \frac{\bar\alpha_t}{\bar\alpha_{t-1}}, \quad \beta_t = 1 - \alpha_t.$$

We never use the symbol s_0 for the schedule; s is reserved for scores.

2.4 Baseline Noise-Prediction Objective

Let $f_\theta(z_t, t)$ be a UNet denoiser that maps a noisy latent and timestep to a noise prediction $\hat{\epsilon}$. The baseline training loss is the standard MSE on noise:

$$\mathcal{L}_{\text{baseline}}(\theta) = \mathbb{E}_{z_0,t,\epsilon}\big[\|f_\theta(z_t, t) - \epsilon\|_2^2\big], \tag{5}$$

where z_t is given by (2).

Sampling from this baseline will be done via the VP-SDE Heun-PC scheme: we reinterpret f_θ as a score (see below) and integrate (3) backwards.

2.5 Gaussian Prior Approximation and Score Notation

The critic–gate construction uses a global Gaussian approximation to the aggregated latent distribution $q(z)$. We estimate empirical mean μ_x and diagonal covariance $\Sigma_x = \operatorname{diag}(\sigma_x^2)$ from $z_0 \sim q_\phi(z \mid x)$; details in Section 5.3.

Define the global prior score

$$s_{\text{prior}}(z) = \Sigma_x^{-1}(z - \mu_x). \tag{6}$$

We explicitly emphasize: **we use a small $\beta_{\mathbf{KL}}$ VAE regularizer** (Section 3.1), so $q(z)$ is only mildly Gaussianized. We are not trying to enforce $q(z) \approx \mathcal{N}(0, I)$. We care about Gaussianization only insofar as it keeps the empirical covariance well-conditioned, so that Σ_x^{-1} does not explode.

In particular:

- Small β_{KL} ensures good reconstructions and preserves rich latent structure.
- The CSEM-based prior term only needs a *reasonable* diagonal Gaussian approximation, not an exact match.
- A numerically bounded Σ_x^{-1} keeps the CSEM contribution to the critic–gate target from dominating or becoming unstable.

2.6 Two Score Signals and Critic–Gate Blend

Let $p_t(z_t \mid z_0)$ denote the forward VP kernel. We define:

- Conditional (likelihood) score:

$$b(z_0, z_t, t) = \nabla_{z_t} \log p_t(z_t \mid z_0) = -\frac{z_t - \sqrt{\bar{\alpha}_t}\, z_0}{1 - \bar{\alpha}_t}.$$

- Global prior score at time t via OU transport of s_{prior}.

  We match the discrete VP schedule to a continuous OU time u_t by $e^{-u_t} = \sqrt{\bar{\alpha}_t}$, so

$$u_t = -\frac{1}{2} \log \bar{\alpha}_t.$$

  Under OU flow, the transported prior score scales like e^{u_t}:

$$a(z_0, t) = e^{u_t}\, s_{\text{prior}}(z_0).$$

Introduce a scalar gate $g(z_t, t; \psi) \in [0, 1]$ that depends on the noisy latent and time. The critic–gate blended *score* target is:

$$s^\star(z_0, z_t, t) = g(z_t, t)\, a(z_0, t) + \big(1 - g(z_t, t)\big)\, b(z_0, z_t, t). \tag{7}$$

2.7 Mapping Score Target to Noise Target

We keep f_θ in the noise parameterization to reuse standard samplers. Using the relation

$$b(z_0, z_t, t) = -\frac{z_t - \sqrt{\bar{\alpha}_t} z_0}{1 - \bar{\alpha}_t} = -\frac{\epsilon}{\sqrt{1 - \bar{\alpha}_t}},$$

we have

$$\epsilon = -\sqrt{1 - \bar{\alpha}_t}\, b.$$

Then the blended noise target corresponding to $s^\star$ is

$$\begin{aligned} \epsilon^\star &= -\sqrt{1 - \bar{\alpha}_t}\, s^\star && (8) \\ &= -\sqrt{1 - \bar{\alpha}_t}\Big[g\, a + (1 - g) b\Big] && (9) \\ &= \epsilon\,(1 - g) - \sqrt{1 - \bar{\alpha}_t}\, g\, a(z_0, t). && (10) \end{aligned}$$

The critic–gate training loss is therefore:

$$\mathcal{L}_{\text{cg}}(\theta, \psi) = \mathbb{E}_{z_0, t, \epsilon}\big[\|f_\theta(z_t, t) - \epsilon^\star\|_2^2\big], \tag{11}$$

with $\epsilon^\star$ from (10). Gradients flow jointly into the UNet parameters θ and the gate parameters ψ.

3 Architectural Choices (Explicit)

3.1 VAE Architecture and Training

Latent resolution.

- Input: $x \in \mathbb{R}^{3\times32\times32}$, values in $[-1, 1]$.
- Latent: $z \in \mathbb{R}^{4\times8\times8}$: downsample by factor 4 in each spatial dimension and use $C = 4$ channels.

Encoder. Use a simple ResNet-style conv encoder:

```
ConvBlock(in_ch, out_ch, stride=1):
    Conv2d(in_ch, out_ch, kernel_size=3, stride=stride, padding=1)
    GroupNorm(num_groups=min(32, out_ch))
    SiLU()

Encoder:
    x: (B, 3, 32, 32)
    h = ConvBlock(3, 64, stride=1)        # 32x32
    h = ConvBlock(64, 128, stride=2)      # 16x16
    h = ConvBlock(128, 256, stride=2)     # 8x8
    # Optionally 1-2 residual blocks at 8x8
    mu      = Conv2d(256, 4, kernel_size=1)   # (B,4,8,8)
    logvar  = Conv2d(256, 4, kernel_size=1)
```

Decoder. Symmetric structure:

```
Decoder:
    z: (B, 4, 8, 8)
    h = ConvBlock(4, 256, stride=1)       # 8x8
    h = Upsample(scale_factor=2, mode="nearest")  # 16x16
    h = ConvBlock(256, 128, stride=1)
    h = Upsample(scale_factor=2, mode="nearest")  # 32x32
    h = ConvBlock(128, 64, stride=1)
    x_logits = Conv2d(64, 3, kernel_size=3, padding=1)
    x_recon = tanh(x_logits)              # in [-1, 1]
```

VAE forward and reparameterization.

```
class VAE(nn.Module):
    def encode(self, x):
        # returns mu, logvar: each (B, 4, 8, 8)

    def reparameterize(self, mu, logvar):
        eps = torch.randn_like(mu)
        z = mu + torch.exp(0.5 * logvar) * eps
        return z

    def decode(self, z):
        # returns x_recon in [-1, 1]

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.reparameterize(mu, logvar)
        x_recon = self.decode(z)
        return x_recon, mu, logvar
```

VAE loss and *small* KL regularization. We explicitly choose a *small* β_{KL} so that:

- reconstruction quality is high;
- the latent distribution $q(z)$ preserves rich structure, not collapsing to a spherical Gaussian;
- the empirical covariance is still well-conditioned, so Σ_x^{-1} is numerically stable.

Loss:

$$\mathcal{L}_{\mathrm{VAE}}(\phi) = \mathbb{E}_{x\sim p_{\mathrm{data}}}\Big[\|x - x_{\mathrm{recon}}\|_2^2\Big] \tag{12}$$

$$+ \beta_{\mathrm{KL}}\mathbb{E}_x\left[-\frac{1}{2}\sum_i \big(1 + \log\sigma_{\phi,i}^2(x) - \mu_{\phi,i}(x)^2 - \sigma_{\phi,i}(x)^2\big)\right]. \tag{13}$$

Choosing $\beta_{\mathrm{KL}} \in [10^{-3}, 10^{-2}]$ is reasonable for CIFAR-10.

Training schedule for VAE.

- Optimizer: Adam with $\mathrm{lr} = 2 \cdot 10^{-4}$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, no weight decay.
- Batch size: 128.
- Epochs: 200–300 over CIFAR-10 train.
- Learning rate schedule: cosine decay or constant learning rate (either is fine; for simplicity, constant lr is acceptable).
- Gradient clipping: optional, e.g. $\|\nabla\| \leq 1.0$.

We save a checkpoint `checkpoints/vae_cifar10.pt` and freeze ϕ when training the diffusion model.

3.2 UNet Denoiser for Latent Diffusion

We follow "Improved DDPM" best practices but scaled to the small latent resolution 8×8:

- Input channels: $C = 4$ (latent channels).
- Base channels: $C_{\mathrm{base}} = 128$.
- Channel multipliers: $[1, 2, 2]$.
- Number of ResBlocks per resolution: 2.
- Attention: at all resolutions (8x8, 4x4, 2x2).
- Time embedding: sinusoidal embedding + MLP, dimension $4C_{\mathrm{base}} = 512$.

Time embedding.

```
class TimeEmbedding(nn.Module):
    def __init__(self, dim: int):
        super().__init__()
        self.dim = dim
        self.mlp = nn.Sequential(
            nn.Linear(dim, 4 * dim),
            nn.SiLU(),
            nn.Linear(4 * dim, 4 * dim),
        )

    def forward(self, t: torch.Tensor, T: int):
        # t: (B,) integer in [0, T-1]
        half = self.dim // 2
        freqs = torch.exp(
            torch.linspace(
                0, math.log(10000), steps=half
            )
        ).to(t.device)
```

```
        # normalize t to [0, 1]
        t_norm = t.float() / float(max(T - 1, 1))
        args = t_norm[:, None] * freqs[None, :]
        emb = torch.cat([torch.sin(args), torch.cos(args)], dim=-1)
        return self.mlp(emb)  # (B, 4*dim)
```

ResBlock and attention.

```
class ResBlock(nn.Module):
    def __init__(self, in_ch, out_ch, time_dim):
        super().__init__()
        self.norm1 = nn.GroupNorm(32, in_ch)
        self.act1 = nn.SiLU()
        self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)

        self.time_proj = nn.Linear(time_dim, out_ch)

        self.norm2 = nn.GroupNorm(32, out_ch)
        self.act2 = nn.SiLU()
        self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)

        if in_ch != out_ch:
            self.skip = nn.Conv2d(in_ch, out_ch, 1)
        else:
            self.skip = nn.Identity()

    def forward(self, x, t_emb):
        h = self.conv1(self.act1(self.norm1(x)))
        h = h + self.time_proj(t_emb)[:, :, None, None]
        h = self.conv2(self.act2(self.norm2(h)))
        return h + self.skip(x)

class AttentionBlock(nn.Module):
    def __init__(self, ch, num_heads=4):
        super().__init__()
        self.norm = nn.GroupNorm(32, ch)
        self.q = nn.Conv2d(ch, ch, 1)
        self.k = nn.Conv2d(ch, ch, 1)
        self.v = nn.Conv2d(ch, ch, 1)
        self.proj = nn.Conv2d(ch, ch, 1)
        self.num_heads = num_heads

    def forward(self, x):
        B, C, H, W = x.shape
        h = self.norm(x)
        q = self.q(h).view(B, self.num_heads, C // self.num_heads, H * W)
        k = self.k(h).view(B, self.num_heads, C // self.num_heads, H * W)
        v = self.v(h).view(B, self.num_heads, C // self.num_heads, H * W)
```

```
        attn = torch.einsum("bnch,bnck->bnhk", q, k) * (C // self.num_heads) ** -0.5
        attn = attn.softmax(dim=-1)
        out = torch.einsum("bnhk,bnck->bnch", attn, v)
        out = out.view(B, C, H, W)
        out = self.proj(out)
        return x + out
```

UNet top-level. Use a standard UNet with 3 resolution levels (8, 4, 2), ResBlocks, attention, and skip connections. A code LLM can copy the pattern from many public implementations (e.g. `improved-diffusion`) but with the above channel sizes and spatial dimensions.

3.3 Critic Gate Network

We use the `CriticGate` described earlier:

```
class CriticGate(nn.Module):
    def __init__(self, num_timesteps, latent_channels,
                 hidden_dim=128, time_embed_dim=16):
        # (implementation from previous section)
        ...
```

4 Codebase Layout

We propose the following layout:

```
ldm_critic_gate_cifar10/
    configs/
        cifar10_vae.yaml
        cifar10_ldm_baseline.yaml
        cifar10_ldm_critic_gate.yaml

    src/
        data/
            cifar10.py
        models/
            vae.py
            unet.py
            critic_gate.py
        diffusion/
            schedules.py
            gaussian_diffusion.py
            heun_pc_sampler.py
        training/
            vae_trainer.py
            ldm_trainer_baseline.py
            ldm_trainer_critic_gate.py
        metrics/
```

```
        fid.py
        projections.py    # for 2D sample histograms

    scripts/
        train_vae.py
        compute_latent_stats.py
        train_ldm_baseline.py
        train_ldm_critic_gate.py
        sample_and_compare.py
```

5 Implementation Details

5.1 CIFAR-10 Data Loader

File: src/data/cifar10.py.

```
def make_cifar10_dataloaders(batch_size=128, num_workers=4):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.5, 0.5, 0.5],
                             std=[0.5, 0.5, 0.5]),
    ])
    train_ds = torchvision.datasets.CIFAR10(
        root="./data", train=True, download=True, transform=transform
    )
    test_ds = torchvision.datasets.CIFAR10(
        root="./data", train=False, download=True, transform=transform
    )
    train_loader = DataLoader(train_ds, batch_size=batch_size,
                              shuffle=True, num_workers=num_workers)
    test_loader = DataLoader(test_ds, batch_size=batch_size,
                             shuffle=False, num_workers=num_workers)
    return train_loader, test_loader
```

5.2 VAE Training Script

File: scripts/train_vae.py.

```
def train_vae():
    cfg = load_yaml("configs/cifar10_vae.yaml")
    train_loader, _ = make_cifar10_dataloaders(
        batch_size=cfg["batch_size"]
    )
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = VAE(cfg).to(device)

    optimizer = torch.optim.Adam(
        model.parameters(),
        lr=cfg["lr"], betas=(0.9, 0.999)
```

```
    )

    beta_kl = cfg.get("beta_kl", 1e-3)  # SMALL KL!

    for epoch in range(cfg["epochs"]):
        model.train()
        for x, _ in train_loader:
            x = x.to(device)
            x_recon, mu, logvar = model(x)

            recon_loss = F.mse_loss(x_recon, x)
            kl = -0.5 * torch.mean(
                1 + logvar - mu.pow(2) - logvar.exp()
            )
            loss = recon_loss + beta_kl * kl

            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
            optimizer.step()

        save_checkpoint(model, "checkpoints/vae_cifar10.pt")
```

5.3 Latent Stats for CSEM Prior

File: `scripts/compute_latent_stats.py`.

```
def compute_latent_stats():
    cfg = load_yaml("configs/cifar10_vae.yaml")
    train_loader, _ = make_cifar10_dataloaders(
        batch_size=cfg["batch_size"]
    )
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    vae = VAE(cfg).to(device)
    vae.load_state_dict(torch.load("checkpoints/vae_cifar10.pt"))
    vae.eval()

    n = 0
    mean_sum = None
    sq_sum = None

    with torch.no_grad():
        for x, _ in train_loader:
            x = x.to(device)
            mu, logvar = vae.encode(x)
            z = vae.reparameterize(mu, logvar)  # SAMPLE from q(z|x)
```

```
            B = z.shape[0]
            z_flat = z.view(B, -1)

            if mean_sum is None:
                D = z_flat.shape[1]
                mean_sum = torch.zeros(D, device=device)
                sq_sum = torch.zeros(D, device=device)

            mean_sum += z_flat.sum(dim=0)
            sq_sum += (z_flat ** 2).sum(dim=0)
            n += B

    mu_flat = mean_sum / float(n)
    second_moment = sq_sum / float(n)
    var_flat = torch.clamp(second_moment - mu_flat ** 2, min=1e-8)

    os.makedirs("stats", exist_ok=True)
    torch.save({"mu": mu_flat.cpu(), "var": var_flat.cpu()},
               "stats/cifar10_latent_gaussian.pt")
```

Again: small β_{KL} means this Gaussian is a coarse, regularized approximation. We use it only for the CSEM prior term in the critic–gate target; we do *not* expect it to perfectly characterize $q(z)$.

5.4 Diffusion Utilities and Schedule

File: `src/diffusion/schedules.py`.

```
def cosine_beta_schedule(T: int, c_offset: float = 0.008) -> torch.Tensor:
    steps = T + 1
    t = torch.linspace(0, T, steps)  # 0,...,T
    # map to [0, 1]
    tau = t / T
    f = torch.cos(
        ( (tau + c_offset) / (1.0 + c_offset) ) * math.pi / 2.0
    ) ** 2
    alphas_cumprod = f / f[0]
    betas = 1.0 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    betas = torch.clamp(betas, min=1e-8, max=0.999)
    return betas
```

File: `src/diffusion/gaussian_diffusion.py`.

```
class GaussianDiffusion:
    def __init__(self, betas: torch.Tensor):
        self.device = betas.device
        self.betas = betas
        self.alphas = 1.0 - betas
        self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
        self.alphas_cumprod_prev = torch.cat(
```

```
            [torch.tensor([1.0], device=self.device),
             self.alphas_cumprod[:-1]],
            dim=0,
        )
        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod)
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(
            1.0 - self.alphas_cumprod
        )
        self.T = betas.shape[0]

    def q_sample(self, z0: torch.Tensor, t: torch.Tensor,
                 noise: torch.Tensor) -> torch.Tensor:
        sqrt_alpha = self.sqrt_alphas_cumprod[t].view(-1, 1, 1, 1)
        sqrt_one_minus = self.sqrt_one_minus_alphas_cumprod[t].view(
            -1, 1, 1, 1
        )
        return sqrt_alpha * z0 + sqrt_one_minus * noise
```

5.5 Heun Predictor–Corrector for VP-SDE

File: `src/diffusion/heun_pc_sampler.py`.

We use the VP-SDE (3), reparameterized as:

$$dZ_\tau = \left(-\tfrac{1}{2}\beta(\tau)Z_\tau - \beta(\tau)\nabla_z \log p_\tau(Z_\tau)\right) d\tau + \sqrt{\beta(\tau)}\, dW_\tau,$$

so that the drift uses the *score* $s_\theta(z, \tau)$. For the noise parameterization f_θ, we convert to a score:

$$s_\theta(z_t, t) \approx -\frac{f_\theta(z_t, t)}{\sqrt{1 - \bar{\alpha}_t}}.$$

Heun-PC update for a step from τ_k to $\tau_{k+1} = \tau_k - \Delta\tau$ (backward in time):

1. Predictor:
$$\tilde{z} = z_k + \Delta\tau\, f_{\text{drift}}(z_k, \tau_k) + \sqrt{\max(\beta(\tau_k)\Delta\tau, 0)}\, \xi,$$
with $\xi \sim \mathcal{N}(0, I)$ and
$$f_{\text{drift}}(z, \tau) = -\tfrac{1}{2}\beta(\tau)z - \beta(\tau)s_\theta(z, \tau).$$

2. Corrector:
$$z_{k+1} = z_k + \frac{\Delta\tau}{2}\Big[f_{\text{drift}}(z_k, \tau_k) + f_{\text{drift}}(\tilde{z}, \tau_{k+1})\Big] + \sqrt{\max(\beta(\tau_{k+1})\Delta\tau, 0)}\, \xi'.$$

A simple implementation skeleton:

```
class HeunPCSampler:
    def __init__(self, diffusion: GaussianDiffusion, num_steps: int = 50):
        self.diffusion = diffusion
        self.num_steps = num_steps

    def _beta_tau(self, tau: torch.Tensor) -> torch.Tensor:
```

```
        # Approximate continuous beta(tau) by interpolating discrete betas.
        # tau in [0, 1]
        t_cont = tau * (self.diffusion.T - 1)
        t0 = torch.floor(t_cont).long()
        t1 = torch.clamp(t0 + 1, max=self.diffusion.T - 1)
        w = t_cont - t0.float()
        betas = (1 - w) * self.diffusion.betas[t0] + w * self.diffusion.betas[t1]
        return betas

    def sample(self, unet, shape, device):
        B = shape[0]
        z = torch.randn(shape, device=device)  # initial noise at tau=1

        taus = torch.linspace(1.0, 1e-3, self.num_steps + 1, device=device)
        for k in range(self.num_steps):
            tau_k = taus[k].expand(B)
            tau_k1 = taus[k + 1].expand(B)
            dt = tau_k1 - tau_k  # negative

            # map tau -> discrete t index
            t_cont = tau_k * (self.diffusion.T - 1)
            t = torch.round(t_cont).long()

            # score from noise prediction
            alpha_bar_t = self.diffusion.alphas_cumprod[t].view(B, 1, 1, 1)
            sigma_t = torch.sqrt(1.0 - alpha_bar_t)

            eps = unet(z, t)  # noise
            score = -eps / sigma_t

            beta_k = self._beta_tau(tau_k)
            drift_k = -0.5 * beta_k.view(B, 1, 1, 1) * z \
                      - beta_k.view(B, 1, 1, 1) * score

            # predictor
            xi = torch.randn_like(z)
            z_pred = z + dt.view(B, 1, 1, 1) * drift_k \
                       + torch.sqrt(torch.clamp(beta_k * (-dt), min=0.0))\
                         .view(B, 1, 1, 1) * xi

            # corrector
            t_cont1 = tau_k1 * (self.diffusion.T - 1)
            t1 = torch.round(t_cont1).long()
            alpha_bar_t1 = self.diffusion.alphas_cumprod[t1].view(B, 1, 1, 1)
            sigma_t1 = torch.sqrt(1.0 - alpha_bar_t1)

            eps1 = unet(z_pred, t1)
            score1 = -eps1 / sigma_t1
```

```
        beta_k1 = self._beta_tau(tau_k1)
        drift_k1 = -0.5 * beta_k1.view(B, 1, 1, 1) * z_pred \
                   - beta_k1.view(B, 1, 1, 1) * score1

        xi1 = torch.randn_like(z)
        z = z + 0.5 * dt.view(B, 1, 1, 1) * (drift_k + drift_k1) \
              + torch.sqrt(torch.clamp(beta_k1 * (-dt), min=0.0))\
                .view(B, 1, 1, 1) * xi1

    return z
```

We use **Heun-PC as the preferred sampler** for both baseline and critic–gate LDMs when generating samples for evaluation.

5.6 Baseline LDM Training Loop

File: `scripts/train_ldm_baseline.py`.

```
def train_ldm_baseline():
    cfg = load_yaml("configs/cifar10_ldm_baseline.yaml")
    train_loader, _ = make_cifar10_dataloaders(
        batch_size=cfg["batch_size"]
    )
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

    # VAE
    vae = VAE(cfg["vae"]).to(device)
    vae.load_state_dict(torch.load("checkpoints/vae_cifar10.pt"))
    vae.eval()

    # Diffusion
    betas = cosine_beta_schedule(T=cfg["T"]).to(device)
    diffusion = GaussianDiffusion(betas)

    # UNet
    unet = UNetModel(...).to(device)

    optimizer = torch.optim.AdamW(
        unet.parameters(),
        lr=cfg["lr"],
        betas=(0.9, 0.999),
        weight_decay=1e-4,
    )

    # Training schedule: ~800k steps / ~200 epochs
    for epoch in range(cfg["epochs"]):
        unet.train()
```

```
        for x, _ in train_loader:
            x = x.to(device)

            with torch.no_grad():
                mu, logvar = vae.encode(x)
                z0 = vae.reparameterize(mu, logvar)

            B = z0.shape[0]
            t = torch.randint(
                low=0,
                high=diffusion.T,
                size=(B,),
                device=device,
                dtype=torch.long,
            )
            noise = torch.randn_like(z0)
            z_t = diffusion.q_sample(z0, t, noise)

            eps_pred = unet(z_t, t)
            loss = F.mse_loss(eps_pred, noise)

            optimizer.zero_grad()
            loss.backward()
            torch.nn.utils.clip_grad_norm_(unet.parameters(), 1.0)
            optimizer.step()

        save_checkpoint(unet, "checkpoints/ldm_baseline.pt")
```

Hyperparameters:

- $T = 1000$.
- Batch size: 128.
- Epochs: ~ 200 (tunable).
- LR schedule: either:
  - constant lr (e.g. $2 \cdot 10^{-4}$), or
  - linear warmup for 10k steps + cosine decay.
- Use EMA on UNet weights (optional but recommended): keep an EMA copy for sampling.

5.7 Critic–Gate LDM Training Loop

File: `scripts/train_ldm_critic_gate.py`.

```
def train_ldm_critic_gate():
    cfg = load_yaml("configs/cifar10_ldm_critic_gate.yaml")
    train_loader, _ = make_cifar10_dataloaders(
```

```
    batch_size=cfg["batch_size"]
)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# VAE
vae = VAE(cfg["vae"]).to(device)
vae.load_state_dict(torch.load("checkpoints/vae_cifar10.pt"))
vae.eval()

# Diffusion
betas = cosine_beta_schedule(T=cfg["T"]).to(device)
diffusion = GaussianDiffusion(betas)

# UNet
unet = UNetModel(...).to(device)

# Critic gate
gate = CriticGate(
    num_timesteps=diffusion.T,
    latent_channels=cfg["latent_channels"],
    hidden_dim=cfg["gate"]["hidden_dim"],
    time_embed_dim=cfg["gate"]["time_embed_dim"],
).to(device)

# Latent stats for prior score
stats = torch.load("stats/cifar10_latent_gaussian.pt", map_location=device)
mu_flat = stats["mu"].view(1, -1)
var_flat = stats["var"].view(1, -1)
inv_var_flat = 1.0 / var_flat

params = list(unet.parameters()) + list(gate.parameters())
optimizer = torch.optim.AdamW(
    params,
    lr=cfg["lr"],
    betas=(0.9, 0.999),
    weight_decay=1e-4,
)

for epoch in range(cfg["epochs"]):
    unet.train()
    gate.train()
    for x, _ in train_loader:
        x = x.to(device)

        with torch.no_grad():
            mu, logvar = vae.encode(x)
            z0 = vae.reparameterize(mu, logvar)   # (B, C, H, W)
```

```
    B = z0.shape[0]
    z0_flat = z0.view(B, -1)

    # prior score at t=0
    s_prior_flat = inv_var_flat * (z0_flat - mu_flat)   # (B, D)
    s_prior = s_prior_flat.view_as(z0)                 # (B, C, H, W)

    t = torch.randint(
        low=0,
        high=diffusion.T,
        size=(B,),
        device=device,
        dtype=torch.long,
    )
    noise = torch.randn_like(z0)
    z_t = diffusion.q_sample(z0, t, noise)

    alpha_bar = diffusion.alphas_cumprod[t].view(B, 1, 1, 1)
    sigma2_t = 1.0 - alpha_bar

    # Tweedie score
    b = -(z_t - torch.sqrt(alpha_bar) * z0) / sigma2_t

    # OU time and transported prior score
    u_t = -0.5 * torch.log(alpha_bar)
    a = torch.exp(u_t) * s_prior

    # Gate
    g = gate(z_t, t)  # (B,1,1,1)

    # Blended noise target
    eps_star = noise * (1.0 - g) - torch.sqrt(sigma2_t) * a * g

    # Noise prediction
    eps_pred = unet(z_t, t)

    loss = F.mse_loss(eps_pred, eps_star)

    optimizer.zero_grad()
    loss.backward()
    torch.nn.utils.clip_grad_norm_(params, 1.0)
    optimizer.step()

save_checkpoint(unet, "checkpoints/ldm_critic_gate_unet.pt")
save_checkpoint(gate, "checkpoints/ldm_critic_gate_gate.pt")
```

6 Sampling, Metrics, and Plots

6.1 Sampling with Heun-PC and VAE Decoder

File: `scripts/sample_and_compare.py`.

For both baseline and critic–gate models:

1. Load the VAE and the appropriate UNet (and gate, though gate is not used at sampling time; only UNet matters).
2. Instantiate `GaussianDiffusion` and `HeunPCSampler`.
3. Generate latent samples with Heun-PC:

```
sampler = HeunPCSampler(diffusion, num_steps=cfg["num_steps"])
z_samples = sampler.sample(
    unet=unet_ema_or_final,
    shape=(N, latent_channels, 8, 8),
    device=device,
)
```

4. Decode with VAE:

```
with torch.no_grad():
    x_recons = vae.decode(z_samples)
    x_imgs = (x_recons.clamp(-1, 1) + 1) / 2  # back to [0,1]
```

5. Save small grids (e.g. 8x8) as PNGs for visual comparison.

6.2 FID Evaluation

File: `src/metrics/fid.py`.

Use any standard PyTorch FID implementation. Procedure:

- Collect CIFAR-10 test images (10k samples) and precompute their inception activations.
- For each model (baseline and critic–gate), generate $N = 50\,000$ samples with Heun-PC, decode to images, and compute their inception activations.
- Compute FID between model activations and real test activations.

Outputs:

- `metrics/fid_baseline.txt`
- `metrics/fid_critic_gate.txt`

6.3 2D Sample Histograms in Latent Space

File: `src/metrics/projections.py`.

We compare the *latent* distributions:

1. Draw a large batch of $z_0 \sim q(z)$ by encoding CIFAR-10 images through the VAE and sampling from $q_\phi(z \mid x)$.
2. Compute PCA on $\{z_0\}$ latents in flattened space (e.g. first two principal components).
3. Project:
   - real latents z_0 to 2D;
   - baseline LDM samples z_{baseline} to 2D;
   - critic–gate LDM samples z_{cg} to 2D.
4. For each, create 2D histograms / contour plots:
   - Real vs baseline.
   - Real vs critic–gate.

This yields visual evidence of how well each LDM matches the latent distribution geometry.

6.4 Panels of Generated Samples

As part of deliverables, generate:

- **8x8 image grid (64 samples)** from baseline LDM;
- **8x8 grid** from critic–gate LDM;
- optionally, grids at several Heun-PC step counts (e.g. 20, 50, 100) to evaluate sample quality vs number of SDE steps.

Each grid should be saved with clear filenames, e.g.:

```
samples/baseline_heunpc_50steps_grid.png
samples/critic_gate_heunpc_50steps_grid.png
```

6.5 Gate Diagnostics (Optional but Recommended)

We can also visualize how the gate behaves:

- Track the distribution of $g(z_t, t)$ over:
  - training epochs,
  - timesteps t.
- For example, at evaluation time:

  ```
  g_vals = gate(z_t, t).detach().cpu().view(-1)
  # log histogram per timestep or per log(SNR)
  ```

- Plot heatmaps of $\mathbb{E}[g(z_t, t)]$ vs t.

7 Final Deliverables

Once implemented and trained, the full system should produce:

- **Codebase**
  - Complete PyTorch implementation under `ldm_critic_gate_cifar10/` as outlined.
  - Clear configuration files for VAE, baseline LDM, and critic–gate LDM.
- **Models**
  - VAE checkpoint: `checkpoints/vae_cifar10.pt`.
  - Baseline UNet (and optional EMA) checkpoint: `checkpoints/ldm_baseline.pt`.
  - Critic–gate UNet and gate checkpoints: `checkpoints/ldm_critic_gate_unet.pt`, `checkpoints/ldm_criti`
  - Latent stats: `stats/cifar10_latent_gaussian.pt`.
- **Metrics**
  - FID scores for:
    * baseline LDM (Heun-PC samples),
    * critic–gate LDM (Heun-PC samples),

    saved in text/JSON with date, config hash, and step count.
  - Training loss curves:
    * Baseline: $\mathbb{E}[\|\hat{\epsilon} - \epsilon\|^2]$ vs iteration.
    * Critic–gate: $\mathbb{E}[\|\hat{\epsilon} - \epsilon^\star\|^2]$ vs iteration.
- **Plots**
  - Panels of generated samples (8x8 grids) for baseline and critic–gate LDMs at a fixed Heun-PC step count (e.g. 50 steps), decoded to image space.
  - 2D latent histograms / contour plots showing:
    * real aggregated posterior $q(z)$ vs baseline LDM samples;
    * $q(z)$ vs critic–gate LDM samples;

    using PCA projections.
  - Optional: heatmap/histograms of gate values $g(z_t, t)$ across timesteps, to show where the model leans toward the prior vs likelihood score.

This specification is explicit enough in architecture, training schedules, sampler design (VP-SDE Heun-PC), and evaluation protocol that a code LLM can implement the entire critic–gate CIFAR-10 LDM from scratch with no further instructions.