```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import os
import gc
from tqdm.notebook import tqdm
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
from torch.cuda.amp import autocast, GradScaler
from sklearn.manifold import TSNE
from sklearn.decomposition import PCA
from scipy.stats import spearmanr
import time
import json
from datetime import datetime
import warnings
import math
warnings.filterwarnings('ignore')

# Cell 2: Utility Functions
def setup_gpu():
    """Set up GPU for training if available"""
    if torch.cuda.is_available():
        print(f"GPU available: {torch.cuda.get_device_name(0)}")
        print(f"Number of GPUs: {torch.cuda.device_count()}")
        device = torch.device("cuda")
        torch.backends.cudnn.benchmark = True
        torch.backends.cudnn.deterministic = False
        torch.cuda.empty_cache()
        if torch.cuda.device_count() > 1:
            print(f"Using {torch.cuda.device_count()} GPUs")
    else:
        device = torch.device("cpu")
        print("No GPU available, using CPU")
    return device

def set_seed(seed=42):
    """Set seeds for reproducibility"""
    np.random.seed(seed)
    torch.manual_seed(seed)
    if torch.cuda.is_available():
        torch.cuda.manual_seed(seed)
        torch.cuda.manual_seed_all(seed)
    return seed

def create_synthetic_dataset(output_dir, num_samples=1000,
input_dim=400, structured=True):
```

```python
    """
    Create synthetic protein structure data

    Args:
        output_dir: Directory to save the data
        num_samples: Number of samples to generate
        input_dim: Dimension of each sample
        structured: Whether to add structure to the random data
    """
    os.makedirs(output_dir, exist_ok=True)

    for i in range(num_samples):
        if structured:
            # Create more structured data with patterns
            base = np.random.rand(input_dim // 4).astype(np.float32)
            # Add some repeating patterns to simulate protein motifs
            structure = np.tile(base, 4) + 0.05 *
np.random.randn(input_dim).astype(np.float32)
            # Add some correlation between features
            structure += np.sin(np.linspace(0, 8 * np.pi, input_dim))
* 0.1
            # Normalize to 0-1 range
            structure = (structure - structure.min()) /
(structure.max() - structure.min())
        else:
            # Simple random data
            structure = np.random.rand(input_dim).astype(np.float32)

        np.save(os.path.join(output_dir, f"protein_{i}.npy"),
structure)

    print(f"Created {num_samples} synthetic protein structures in
{output_dir}")

# Cell 3: Dataset and DataLoader Classes
class ProteinStructureDataset(Dataset):
    def __init__(self, data_path=None, data=None, transform=None):
        """
        Initialize dataset either from a directory of files or from
provided data

        Args:
            data_path: Path to directory with protein structure data
files
            data: Directly provided data (numpy array)
            transform: Optional transforms to apply
        """
        self.transform = transform

        if data is not None:
```

```
            self.data = data
            self.from_memory = True
        else:
            self.data_path = data_path
            self.data_files = self._get_data_files()
            self.from_memory = False

    def _get_data_files(self):
        """Get all data files from the directory"""
        files = [os.path.join(self.data_path, f) for f in
os.listdir(self.data_path)
                if f.endswith('.npy') or f.endswith('.npz')]
        return files

    def __len__(self):
        """Return the size of the dataset"""
        if self.from_memory:
            return len(self.data)
        return len(self.data_files)

    def __getitem__(self, idx):
        """Get a specific data item"""
        if self.from_memory:
            structure = self.data[idx]
        else:
            structure = np.load(self.data_files[idx])

        if self.transform:
            structure = self.transform(structure)

        return torch.tensor(structure, dtype=torch.float32)

def create_data_loaders(data_path=None, data=None, batch_size=128,
                        num_workers=4, pin_memory=True,
train_ratio=0.7,
                        val_ratio=0.15, test_ratio=0.15,
shuffle=True):
    """
    Create train, validation, and test data loaders

    Args:
        data_path: Path to data directory
        data: Directly provided data array
        batch_size: Batch size for the loaders
        num_workers: Number of workers for loading
        pin_memory: Whether to pin memory
        train_ratio, val_ratio, test_ratio: Dataset split ratios
        shuffle: Whether to shuffle the data
    """
```

```
    if data is not None:
        # Create a dataset from provided data
        dataset = ProteinStructureDataset(data=data)

        # Split data
        total_size = len(dataset)
        train_size = int(train_ratio * total_size)
        val_size = int(val_ratio * total_size)
        test_size = total_size - train_size - val_size

        train_data, val_data, test_data =
torch.utils.data.random_split(
            dataset, [train_size, val_size, test_size]
        )
    else:
        # Load from path
        train_data = ProteinStructureDataset(os.path.join(data_path,
'train'))
        val_data = ProteinStructureDataset(os.path.join(data_path,
'val'))
        test_data = ProteinStructureDataset(os.path.join(data_path,
'test'))

    # Create data loaders
    train_loader = DataLoader(
        train_data,
        batch_size=batch_size,
        shuffle=shuffle,
        num_workers=num_workers,
        pin_memory=pin_memory,
        persistent_workers=(num_workers > 0),
        prefetch_factor=2 if num_workers > 0 else None,
    )

    val_loader = DataLoader(
        val_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
        pin_memory=pin_memory,
        persistent_workers=(num_workers > 0),
        prefetch_factor=2 if num_workers > 0 else None,
    )

    test_loader = DataLoader(
        test_data,
        batch_size=batch_size,
        shuffle=False,
        num_workers=num_workers,
```

```python
        pin_memory=pin_memory,
        persistent_workers=(num_workers > 0),
        prefetch_factor=2 if num_workers > 0 else None,
    )

    return train_loader, val_loader, test_loader

# Cell 4: VAE Model
class ProteinVAE(nn.Module):
    def __init__(self, input_dim, hidden_dim, latent_dim,
dropout_rate=0.1):
        super(ProteinVAE, self).__init__()

        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.latent_dim = latent_dim

        # Encoder network
        self.encoder = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Dropout(dropout_rate),

            nn.Linear(hidden_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Dropout(dropout_rate)
        )

        # Latent space projection
        self.fc_mu = nn.Linear(hidden_dim, latent_dim)
        self.fc_var = nn.Linear(hidden_dim, latent_dim)

        # Decoder network
        self.decoder = nn.Sequential(
            nn.Linear(latent_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Dropout(dropout_rate),

            nn.Linear(hidden_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.LeakyReLU(0.2),
            nn.Dropout(dropout_rate),

            nn.Linear(hidden_dim, input_dim)
        )

    def encode(self, x):
```

```
        """Encode input to latent space parameters"""
        h = self.encoder(x)
        mu = self.fc_mu(h)
        log_var = self.fc_var(h)
        return mu, log_var

    def reparameterize(self, mu, log_var):
        """Reparameterization trick for sampling from latent
distribution"""
        std = torch.exp(0.5 * log_var)
        eps = torch.randn_like(std)
        z = mu + eps * std
        return z

    def decode(self, z):
        """Decode from latent space to original space"""
        return self.decoder(z)

    def forward(self, x):
        """Full forward pass: encode -> sample -> decode"""
        mu, log_var = self.encode(x)
        z = self.reparameterize(mu, log_var)
        x_reconstructed = self.decode(z)
        return x_reconstructed, mu, log_var

def vae_loss_function(recon_x, x, mu, log_var, beta=1.0):
    """
    VAE loss function combining reconstruction loss and KL divergence

    Args:
        recon_x: Reconstructed input
        x: Original input
        mu: Mean of the latent distribution
        log_var: Log variance of the latent distribution
        beta: Weight of the KL divergence term
    """
    # Binary cross entropy for reconstruction
    BCE = F.binary_cross_entropy_with_logits(recon_x, x,
reduction='sum')

    # KL divergence
    KLD = -0.5 * torch.sum(1 + log_var - mu.pow(2) - log_var.exp())

    return BCE + beta * KLD

class DiffusionModel(nn.Module):
    def __init__(self, input_dim, hidden_dim=256, time_embed_dim=128,
dropout_rate=0.1):
        super(DiffusionModel, self).__init__()
```

```
        self.input_dim = input_dim
        self.time_embed_dim = time_embed_dim

        # Time embedding
        self.time_embed = nn.Sequential(
            nn.Linear(1, time_embed_dim),
            nn.SiLU(),
            nn.Linear(time_embed_dim, time_embed_dim),
        )

        # Main network
        self.net = nn.Sequential(
            nn.Linear(input_dim + time_embed_dim, hidden_dim),  # Fix:
input_dim + time_embed_dim
            nn.BatchNorm1d(hidden_dim),
            nn.SiLU(),
            nn.Dropout(dropout_rate),

            nn.Linear(hidden_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.SiLU(),
            nn.Dropout(dropout_rate),

            nn.Linear(hidden_dim, hidden_dim),
            nn.BatchNorm1d(hidden_dim),
            nn.SiLU(),
            nn.Dropout(dropout_rate),

            nn.Linear(hidden_dim, input_dim)
        )

    def forward(self, x, t):
        """
        Forward pass

        Args:
            x: Input data [batch_size, input_dim]
            t: Timesteps [batch_size, 1]
        """
        t_emb = self.time_embed(t)  # Shape: (batch_size,
time_embed_dim)
        x_input = torch.cat([x, t_emb], dim=1)  # Concatenate x and
time embedding
        return self.net(x_input)

class DiffusionTrainer:
    def __init__(self, model, device, n_timesteps=1000, beta_start=1e-
4, beta_end=0.02, noise_schedule='cosine', s=0.008):
        """
```

```
        Diffusion model training controller

        Args:
            model: DiffusionModel
            device: Torch device
            n_timesteps: Number of diffusion steps
            beta_start, beta_end: Noise schedule parameters (for
linear schedule)
            noise_schedule: 'linear' or 'cosine'
            s: Small offset for cosine schedule
        """
        self.model = model
        self.device = device
        self.n_timesteps = n_timesteps
        self.noise_schedule = noise_schedule
        self.s = s

        if noise_schedule == 'linear':
            # Linear noise schedule
            self.betas = torch.linspace(beta_start, beta_end,
n_timesteps).to(device)
            self.alphas = 1. - self.betas
            self.alphas_cumprod = torch.cumprod(self.alphas, dim=0)
            self.alphas_cumprod_prev = F.pad(self.alphas_cumprod[:-1],
(1, 0), value=1.0)

        elif noise_schedule == 'cosine':
            # Cosine noise schedule
            self.alphas_cumprod =
torch.tensor(self._cosine_schedule()).float().to(device)
            self.alphas_cumprod_prev = F.pad(self.alphas_cumprod[:-1],
(1, 0), value=1.0)

            self.betas = 1 - (self.alphas_cumprod[1:] /
self.alphas_cumprod[:-1])
            self.betas = torch.clip(self.betas, 0, 0.999).to(device)
# Clip betas to avoid extreme values

            self.alphas = 1. - self.betas
        else:
            raise ValueError(f"Invalid noise schedule:
{noise_schedule}")

        self.sqrt_recip_alphas = torch.sqrt(1.0 / self.alphas)
        self.sqrt_alphas_cumprod = torch.sqrt(self.alphas_cumprod[1:])
        self.sqrt_one_minus_alphas_cumprod = torch.sqrt(1. -
self.alphas_cumprod[1:])
        self.posterior_variance = self.betas * (1. -
self.alphas_cumprod[:-1]) / (1. - self.alphas_cumprod[1:])
```

```
    def _cosine_schedule(self):
        """
        Generate cosine schedule
        """
        steps = self.n_timesteps + 1
        x = torch.linspace(0, self.n_timesteps, steps)
        alphas_cumprod = torch.cos(((x / self.n_timesteps) + self.s) /
(1 + self.s) * math.pi * 0.5) ** 2
        alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
        return alphas_cumprod

    def q_sample(self, x_start, t, noise=None):
        """Forward diffusion process: add noise to data"""
        if noise is None:
            noise = torch.randn_like(x_start)

        sqrt_alphas_cumprod_t = self.sqrt_alphas_cumprod[t].reshape(-
1, 1)
        sqrt_one_minus_alphas_cumprod_t =
self.sqrt_one_minus_alphas_cumprod[t].reshape(-1, 1)

        return sqrt_alphas_cumprod_t * x_start +
sqrt_one_minus_alphas_cumprod_t * noise

    def p_losses(self, x_start, t, noise=None):
        """Calculate loss for denoising diffusion"""
        if noise is None:
            noise = torch.randn_like(x_start)

        x_noisy = self.q_sample(x_start, t, noise)
        predicted_noise = self.model(x_noisy, t.reshape(-1, 1).float()
/ self.n_timesteps)

        loss = F.mse_loss(predicted_noise, noise)
        return loss

    @torch.no_grad()
    def p_sample(self, x, t):
        """Sample from the model at timestep t"""
        betas_t = self.betas[t].reshape(-1, 1)
        sqrt_one_minus_alphas_cumprod_t =
self.sqrt_one_minus_alphas_cumprod[t].reshape(-1, 1)
        sqrt_recip_alphas_t = self.sqrt_recip_alphas[t].reshape(-1, 1)

        # Use our model (noise predictor) to predict the mean
        model_mean = sqrt_recip_alphas_t * (
            x - betas_t * self.model(x, t.reshape(-1, 1).float() /
self.n_timesteps) / sqrt_one_minus_alphas_cumprod_t
```

```
        )

        # Handle the case where t contains a mix of 0 and non-zero
values
        if (t == 0).any():
            return model_mean
        else:
            posterior_variance_t =
self.posterior_variance[t].reshape(-1, 1)
            noise = torch.randn_like(x)
            # Algorithm 2 line 4:
            return model_mean + torch.sqrt(posterior_variance_t) *
noise

    @torch.no_grad()
    def p_sample_loop(self, shape):
        """Generate samples by sampling backwards through the
diffusion process"""
        self.model.eval()
        device = next(self.model.parameters()).device

        b = shape[0]
        # Start from pure noise
        img = torch.randn(shape).to(device)
        imgs = []

        for i in tqdm(reversed(range(0, self.n_timesteps)),
desc='Sampling', total=self.n_timesteps):
            t = torch.full((b,), i, device=device, dtype=torch.long)
            img = self.p_sample(img, t)
            imgs.append(img.cpu().numpy())

        return img, imgs

    def sample(self, n_samples, shape):
        """Generate new protein samples using the diffusion model"""
        sample_shape = (n_samples, shape)
        samples, diffusion_steps = self.p_sample_loop(sample_shape)

        # Apply sigmoid to map values to 0-1 range
        samples = torch.sigmoid(samples)

        return samples, diffusion_steps

def train_diffusion_model(diffusion_model, train_loader, val_loader,
device,
                         epochs=50, lr=1e-4, weight_decay=1e-5,
use_amp=True, model_dir='models',
                         noise_schedule='cosine', warmup_steps=500):
```

```
    """Train the diffusion model"""
    os.makedirs(model_dir, exist_ok=True)
    model_path = os.path.join(model_dir, 'best_diffusion_model.pt')

    diffusion_trainer = DiffusionTrainer(diffusion_model, device,
noise_schedule=noise_schedule)

    optimizer = torch.optim.AdamW(
        diffusion_model.parameters(),
        lr=lr,
        weight_decay=weight_decay
    )

    # Learning rate warm-up and cosine annealing
    total_steps = len(train_loader) * epochs
    scheduler = WarmupCosineScheduler(optimizer,
warmup_steps=warmup_steps, total_steps=total_steps)

    early_stopping = EarlyStopping(patience=10, save_path=model_path)
    scaler = GradScaler() if use_amp else None

    train_losses = []
    val_losses = []

    start_time = time.time()

    for epoch in range(epochs):
        # Training phase
        diffusion_model.train()
        train_loss = 0
        batch_count = 0

        progress_bar = tqdm(train_loader, desc=f"Epoch
{epoch+1}/{epochs}")
        for batch in progress_bar:
            optimizer.zero_grad()

            x = batch.to(device, non_blocking=True)
            batch_size = x.shape[0]

            # Sample random timesteps
            t = torch.randint(0, diffusion_trainer.n_timesteps,
(batch_size,), device=device).long()

            if use_amp:
                with autocast():
                    loss = diffusion_trainer.p_losses(x, t)

                scaler.scale(loss).backward()
```

```
                scaler.step(optimizer)
                scaler.update()
            else:
                loss = diffusion_trainer.p_losses(x, t)
                loss.backward()
                optimizer.step()

            train_loss += loss.item()
            batch_count += 1

            progress_bar.set_postfix({'loss': loss.item()})

            # Free up memory
            del x, t, loss

            # Update learning rate
            scheduler.step()

        # Validation phase
        diffusion_model.eval()
        val_loss = 0
        val_batches = 0

        with torch.no_grad():
            for batch in val_loader:
                x = batch.to(device, non_blocking=True)
                batch_size = x.shape[0]

                # Sample random timesteps
                t = torch.randint(0, diffusion_trainer.n_timesteps,
(batch_size,), device=device).long()

                if use_amp:
                    with autocast():
                        loss = diffusion_trainer.p_losses(x, t)
                else:
                    loss = diffusion_trainer.p_losses(x, t)

                val_loss += loss.item()
                val_batches += 1

                del x, t, loss

        avg_val_loss = val_loss / val_batches
        val_losses.append(avg_val_loss)

        # Check early stopping and save best model
        early_stopping(avg_val_loss, diffusion_model)
        if early_stopping.early_stop:
```

```
            print(f"Early stopping at epoch {epoch+1}")
            break

        print(f"Epoch {epoch+1}: Train Loss: {avg_train_loss:.6f}, Val
Loss: {avg_val_loss:.6f}")

        # Memory cleanup
        torch.cuda.empty_cache()
        gc.collect()

    end_time = time.time()
    training_time = end_time - start_time
    print(f"Diffusion model training completed in {training_time:.2f}
seconds")

    # Load best model if it exists, otherwise save the current model
    if os.path.exists(model_path):
        diffusion_model.load_state_dict(torch.load(model_path))
    else:
        torch.save(diffusion_model.state_dict(), model_path)
        print(f"Saved final model as best model to {model_path}")

    # Save training history
    history = {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'training_time': training_time,
        'epochs': len(train_losses)
    }

    with open(os.path.join(model_dir,
'diffusion_training_history.json'), 'w') as f:
        json.dump(history, f)

    return diffusion_model, diffusion_trainer, train_losses,
val_losses

# Cell 6: Training Utilities (Updated for Stability)
def train_diffusion_model(diffusion_model, train_loader, val_loader,
device,
                         epochs=50, lr=1e-4, weight_decay=1e-5,
use_amp=True, model_dir='models',
                         noise_schedule='cosine'):
    """Train the diffusion model"""
    os.makedirs(model_dir, exist_ok=True)
    model_path = os.path.join(model_dir, 'best_diffusion_model.pt')

    diffusion_trainer = DiffusionTrainer(diffusion_model, device,
noise_schedule=noise_schedule)
```

```
    optimizer = torch.optim.AdamW(
        diffusion_model.parameters(),
        lr=lr,
        weight_decay=weight_decay
    )

    # Learning rate warm-up and cosine annealing
    total_steps = len(train_loader) * epochs
    scheduler = WarmupCosineScheduler(optimizer,
warmup_steps=warmup_steps, total_steps=total_steps)

    early_stopping = EarlyStopping(patience=10, save_path=model_path)
    scaler = GradScaler() if use_amp else None

    train_losses = []
    val_losses = []

    start_time = time.time()

    for epoch in range(epochs):
        # Training phase
        diffusion_model.train()
        train_loss = 0
        batch_count = 0

        progress_bar = tqdm(train_loader, desc=f"Epoch
{epoch+1}/{epochs}")
        for batch in progress_bar:
            optimizer.zero_grad()

            x = batch.to(device, non_blocking=True)
            batch_size = x.shape[0]

            # Sample random timesteps
            t = torch.randint(0, diffusion_trainer.n_timesteps,
(batch_size,), device=device).long()

            if use_amp:
                with autocast():
                    loss = diffusion_trainer.p_losses(x, t)

                scaler.scale(loss).backward()
                scaler.step(optimizer)
                scaler.update()
            else:
                loss = diffusion_trainer.p_losses(x, t)
                loss.backward()
                optimizer.step()
```

```
            train_loss += loss.item()
            batch_count += 1

            progress_bar.set_postfix({'loss': loss.item()})

            # Free up memory
            del x, t, loss

            # Update learning rate
            scheduler.step()

        # Validation phase
        diffusion_model.eval()
        val_loss = 0
        val_batches = 0

        with torch.no_grad():
            for batch in val_loader:
                x = batch.to(device, non_blocking=True)
                batch_size = x.shape[0]

                # Sample random timesteps
                t = torch.randint(0, diffusion_trainer.n_timesteps,
(batch_size,), device=device).long()

                if use_amp:
                    with autocast():
                        loss = diffusion_trainer.p_losses(x, t)
                else:
                    loss = diffusion_trainer.p_losses(x, t)

                val_loss += loss.item()
                val_batches += 1

                del x, t, loss

        avg_val_loss = val_loss / val_batches
        val_losses.append(avg_val_loss)

        # Check early stopping and save best model
        early_stopping(avg_val_loss, diffusion_model)
        if early_stopping.early_stop:
            print(f"Early stopping at epoch {epoch+1}")
            break

        print(f"Epoch {epoch+1}: Train Loss: {avg_train_loss:.6f}, Val
Loss: {avg_val_loss:.6f}")

        # Memory cleanup
```

```
        torch.cuda.empty_cache()
        gc.collect()

    end_time = time.time()
    training_time = end_time - start_time
    print(f"Diffusion model training completed in {training_time:.2f}
seconds")

    # Load best model if it exists, otherwise save the current model
    if os.path.exists(model_path):
        diffusion_model.load_state_dict(torch.load(model_path))
    else:
        torch.save(diffusion_model.state_dict(), model_path)
        print(f"Saved final model as best model to {model_path}")

    # Save training history
    history = {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'training_time': training_time,
        'epochs': len(train_losses)
    }

    with open(os.path.join(model_dir,
'diffusion_training_history.json'), 'w') as f:
        json.dump(history, f)

    return diffusion_model, diffusion_trainer, train_losses,
val_losses

class EarlyStopping:
    def __init__(self, patience=7, min_delta=0,
save_path='best_model.pt'):
        """
        Early stopping controller

        Args:
            patience: How many epochs to wait for improvement
            min_delta: Minimum change to qualify as improvement
            save_path: Where to save the best model
        """
        self.patience = patience
        self.min_delta = min_delta
        self.counter = 0
        self.best_loss = None
        self.early_stop = False
        self.save_path = save_path

    def __call__(self, val_loss, model):
```

```
        """Check if training should stop and save model if it's the
best so far"""
        if self.best_loss is None:
            self.best_loss = val_loss
            self.save_checkpoint(model)
        elif val_loss > self.best_loss - self.min_delta:
            self.counter += 1
            print(f"Early stopping counter:
{self.counter}/{self.patience}")
            if self.counter >= self.patience:
                self.early_stop = True
        else:
            self.best_loss = val_loss
            self.save_checkpoint(model)
            self.counter = 0

    def save_checkpoint(self, model):
        """Save model checkpoint"""
        torch.save(model.state_dict(), self.save_path)
        print(f'Model saved to {self.save_path}')

def train_model(model, train_loader, val_loader, device, epochs=100,
lr=1e-3,
               beta=1.0, weight_decay=1e-5, use_amp=True,
model_dir='models'):
    """
    Train the VAE model

    Args:
        model: VAE model
        train_loader, val_loader: Data loaders
        device: Torch device
        epochs: Number of epochs
        lr: Learning rate
        beta: Weight of KL divergence in loss function
        weight_decay: L2 regularization
        use_amp: Whether to use automatic mixed precision
        model_dir: Directory to save models
    """
    os.makedirs(model_dir, exist_ok=True)
    model_path = os.path.join(model_dir, 'best_vae_model.pt')

    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=lr,
        weight_decay=weight_decay
    )

    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
```

```
        optimizer, T_max=epochs, eta_min=lr/10
    )

    early_stopping = EarlyStopping(patience=10, save_path=model_path)
    scaler = GradScaler() if use_amp else None

    train_losses = []
    val_losses = []

    start_time = time.time()

    for epoch in range(epochs):
        # Training phase
        model.train()
        train_loss = 0
        progress_bar = tqdm(train_loader, desc=f"Epoch
{epoch+1}/{epochs}")

        for batch_idx, data in enumerate(progress_bar):
            data = data.to(device, non_blocking=True)
            optimizer.zero_grad()

            if use_amp:
                # Fix: Remove device_type parameter
                with autocast():
                    recon_batch, mu, log_var = model(data)
                    loss = vae_loss_function(recon_batch, data, mu,
log_var, beta=beta)

                scaler.scale(loss).backward()
                scaler.step(optimizer)
                scaler.update()
            else:
                recon_batch, mu, log_var = model(data)
                loss = vae_loss_function(recon_batch, data, mu,
log_var, beta=beta)
                loss.backward()
                optimizer.step()

            train_loss += loss.item()
            progress_bar.set_postfix({'loss': loss.item()})

            # Clean up memory
            del data, recon_batch, mu, log_var, loss

        scheduler.step()

        # Calculate average training loss
        avg_train_loss = train_loss / len(train_loader.dataset)
```

```
        train_losses.append(avg_train_loss)

        # Validation phase
        model.eval()
        val_loss = 0

        with torch.no_grad():
            for data in val_loader:
                data = data.to(device, non_blocking=True)

                if use_amp:
                    # Fix: Remove device_type parameter
                    with autocast():
                        recon_batch, mu, log_var = model(data)
                        loss = vae_loss_function(recon_batch, data,
mu, log_var, beta=beta)
                else:
                    recon_batch, mu, log_var = model(data)
                    loss = vae_loss_function(recon_batch, data, mu,
log_var, beta=beta)

                val_loss += loss.item()

                # Clean up memory
                del data, recon_batch, mu, log_var, loss

        # Calculate average validation loss
        avg_val_loss = val_loss / len(val_loader.dataset)
        val_losses.append(avg_val_loss)

        # Check early stopping
        early_stopping(avg_val_loss, model)
        if early_stopping.early_stop:
            print(f"Early stopping at epoch {epoch+1}")
            break

        print(f"Epoch {epoch+1}: Train Loss: {avg_train_loss:.4f}, Val
Loss: {avg_val_loss:.4f}")

        # Clean memory
        torch.cuda.empty_cache()
        gc.collect()

    end_time = time.time()
    training_time = end_time - start_time
    print(f"Training completed in {training_time:.2f} seconds")

    # Load best model
    model.load_state_dict(torch.load(model_path))
```

```
    # Save training history
    history = {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'training_time': training_time,
        'epochs': len(train_losses)
    }

    with open(os.path.join(model_dir, 'vae_training_history.json'),
'w') as f:
        json.dump(history, f)

    return model, train_losses, val_losses


def train_diffusion_model(diffusion_model, train_loader, val_loader,
device,
                         epochs=50, lr=1e-4, weight_decay=1e-5,
use_amp=True, model_dir='models'):
    """Train the diffusion model"""
    os.makedirs(model_dir, exist_ok=True)
    model_path = os.path.join(model_dir, 'best_diffusion_model.pt')

    diffusion_trainer = DiffusionTrainer(diffusion_model, device)

    optimizer = torch.optim.AdamW(
        diffusion_model.parameters(),
        lr=lr,
        weight_decay=weight_decay
    )

    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=epochs, eta_min=lr/10
    )

    early_stopping = EarlyStopping(patience=10, save_path=model_path)
    scaler = GradScaler() if use_amp else None

    train_losses = []
    val_losses = []

    start_time = time.time()

    for epoch in range(epochs):
        # Training phase
        diffusion_model.train()
        train_loss = 0
        batch_count = 0
```

```
        progress_bar = tqdm(train_loader, desc=f"Epoch
{epoch+1}/{epochs}")
        for batch in progress_bar:
            optimizer.zero_grad()

            x = batch.to(device, non_blocking=True)
            batch_size = x.shape[0]

            # Sample random timesteps
            t = torch.randint(0, diffusion_trainer.n_timesteps,
(batch_size,), device=device).long()

            if use_amp:
                with autocast():
                    loss = diffusion_trainer.p_losses(x, t)

                scaler.scale(loss).backward()
                scaler.step(optimizer)
                scaler.update()
            else:
                loss = diffusion_trainer.p_losses(x, t)
                loss.backward()
                optimizer.step()

            train_loss += loss.item()
            batch_count += 1

            progress_bar.set_postfix({'loss': loss.item()})

            # Free up memory
            del x, t, loss

        if scheduler is not None:
            scheduler.step()

        avg_train_loss = train_loss / batch_count
        train_losses.append(avg_train_loss)

        # Validation phase
        diffusion_model.eval()
        val_loss = 0
        val_batches = 0

        with torch.no_grad():
            for batch in val_loader:
                x = batch.to(device, non_blocking=True)
                batch_size = x.shape[0]
```

```
                # Sample random timesteps
                t = torch.randint(0, diffusion_trainer.n_timesteps,
(batch_size,), device=device).long()

                if use_amp:
                    with autocast():
                        loss = diffusion_trainer.p_losses(x, t)
                else:
                    loss = diffusion_trainer.p_losses(x, t)

                val_loss += loss.item()
                val_batches += 1

                del x, t, loss

        avg_val_loss = val_loss / val_batches
        val_losses.append(avg_val_loss)

        # Check early stopping and save best model
        early_stopping(avg_val_loss, diffusion_model)
        if early_stopping.early_stop:
            print(f"Early stopping at epoch {epoch+1}")
            break

        print(f"Epoch {epoch+1}: Train Loss: {avg_train_loss:.6f}, Val
Loss: {avg_val_loss:.6f}")

        # Memory cleanup
        torch.cuda.empty_cache()
        gc.collect()

    end_time = time.time()
    training_time = end_time - start_time
    print(f"Diffusion model training completed in {training_time:.2f}
seconds")

    # Load best model if it exists, otherwise save the current model
    if os.path.exists(model_path):
        diffusion_model.load_state_dict(torch.load(model_path))
    else:
        torch.save(diffusion_model.state_dict(), model_path)
        print(f"Saved final model as best model to {model_path}")

    # Save training history
    history = {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'training_time': training_time,
        'epochs': len(train_losses)
    }
```

```python
    with open(os.path.join(model_dir,
'diffusion_training_history.json'), 'w') as f:
        json.dump(history, f)

    return diffusion_model, diffusion_trainer, train_losses,
val_losses

# Cell 6: Training Utilities (Fixed)
def train_model(model, train_loader, val_loader, device, epochs=100,
lr=1e-3,
               beta=1.0, weight_decay=1e-5, use_amp=True,
model_dir='models'):
    """
    Train the VAE model

    Args:
        model: VAE model
        train_loader, val_loader: Data loaders
        device: Torch device
        epochs: Number of epochs
        lr: Learning rate
        beta: Weight of KL divergence in loss function
        weight_decay: L2 regularization
        use_amp: Whether to use automatic mixed precision
        model_dir: Directory to save models
    """
    os.makedirs(model_dir, exist_ok=True)
    model_path = os.path.join(model_dir, 'best_vae_model.pt')

    optimizer = torch.optim.AdamW(
        model.parameters(),
        lr=lr,
        weight_decay=weight_decay
    )

    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(
        optimizer, T_max=epochs, eta_min=lr/10
    )

    early_stopping = EarlyStopping(patience=10, save_path=model_path)
    scaler = GradScaler() if use_amp else None

    train_losses = []
    val_losses = []

    start_time = time.time()

    for epoch in range(epochs):
        # Training phase
```

```
        model.train()
        train_loss = 0
        progress_bar = tqdm(train_loader, desc=f"Epoch
{epoch+1}/{epochs}")

        for batch_idx, data in enumerate(progress_bar):
            data = data.to(device, non_blocking=True)
            optimizer.zero_grad()

            if use_amp:
                with autocast():
                    recon_batch, mu, log_var = model(data)
                    loss = vae_loss_function(recon_batch, data, mu,
log_var, beta=beta)

                scaler.scale(loss).backward()
                scaler.step(optimizer)
                scaler.update()
            else:
                recon_batch, mu, log_var = model(data)
                loss = vae_loss_function(recon_batch, data, mu,
log_var, beta=beta)
                loss.backward()
                optimizer.step()

            train_loss += loss.item()
            progress_bar.set_postfix({'loss': loss.item()})

            # Clean up memory
            del data, recon_batch, mu, log_var, loss

        scheduler.step()

        # Calculate average training loss
        avg_train_loss = train_loss / len(train_loader.dataset)
        train_losses.append(avg_train_loss)

        # Validation phase
        model.eval()
        val_loss = 0

        with torch.no_grad():
            for data in val_loader:
                data = data.to(device, non_blocking=True)

                if use_amp:
                    with autocast():
                        recon_batch, mu, log_var = model(data)
                        loss = vae_loss_function(recon_batch, data,
```

```
mu, log_var, beta=beta)
                else:
                    recon_batch, mu, log_var = model(data)
                    loss = vae_loss_function(recon_batch, data, mu,
log_var, beta=beta)

                val_loss += loss.item()

                # Clean up memory
                del data, recon_batch, mu, log_var, loss

        # Calculate average validation loss
        avg_val_loss = val_loss / len(val_loader.dataset)
        val_losses.append(avg_val_loss)

        # Check early stopping
        early_stopping(avg_val_loss, model)
        if early_stopping.early_stop:
            print(f"Early stopping at epoch {epoch+1}")
            break

        print(f"Epoch {epoch+1}: Train Loss: {avg_train_loss:.4f}, Val
Loss: {avg_val_loss:.4f}")

        # Clean memory
        torch.cuda.empty_cache()
        gc.collect()

    end_time = time.time()
    training_time = end_time - start_time
    print(f"Training completed in {training_time:.2f} seconds")

    # Load best model
    model.load_state_dict(torch.load(model_path))

    # Save training history
    history = {
        'train_losses': train_losses,
        'val_losses': val_losses,
        'training_time': training_time,
        'epochs': len(train_losses)
    }

    with open(os.path.join(model_dir, 'vae_training_history.json'),
'w') as f:
        json.dump(history, f)

    return model, train_losses, val_losses
```

```python
# Cell 7: Evaluation and Visualization Functions (Fixed for Kaggle)
@torch.no_grad()
def evaluate_model(model, test_loader, device, use_amp=True):
    """
    Evaluate the VAE model on test data

    Args:
        model: VAE model
        test_loader: Test data loader
        device: Torch device
        use_amp: Whether to use automatic mixed precision
    """
    model.eval()
    test_loss = 0
    reconstruction_error = 0
    kl_divergence = 0
    all_mu = []
    all_log_var = []

    with torch.no_grad():
        for data in tqdm(test_loader, desc="Evaluating"):
            data = data.to(device, non_blocking=True)

            if use_amp:
                # Fix: Remove device_type parameter
                with autocast():
                    recon_batch, mu, log_var = model(data)
                    loss = vae_loss_function(recon_batch, data, mu,
log_var)
            else:
                recon_batch, mu, log_var = model(data)
                loss = vae_loss_function(recon_batch, data, mu,
log_var)

            test_loss += loss.item()

            # Component-wise losses
            recon_error =
F.binary_cross_entropy_with_logits(recon_batch, data,
reduction='sum').item()
            reconstruction_error += recon_error

            kld = -0.5 * torch.sum(1 + log_var - mu.pow(2) -
log_var.exp()).item()
            kl_divergence += kld

            # Save latent space statistics
            all_mu.append(mu.cpu().numpy())
            all_log_var.append(log_var.cpu().numpy())
```

```python
            # Clean memory
            del data, recon_batch, mu, log_var

    # Calculate average metrics
    test_loss /= len(test_loader.dataset)
    reconstruction_error /= len(test_loader.dataset)
    kl_divergence /= len(test_loader.dataset)

    # Combine all latent variables
    all_mu = np.concatenate(all_mu, axis=0)
    all_log_var = np.concatenate(all_log_var, axis=0)

    # Calculate statistics on latent space
    mu_mean = np.mean(all_mu, axis=0)
    mu_std = np.std(all_mu, axis=0)
    var_mean = np.mean(np.exp(all_log_var), axis=0)

    # Package metrics
    metrics = {
        'test_loss': test_loss,
        'reconstruction_error': reconstruction_error,
        'kl_divergence': kl_divergence,
        'mu_mean': mu_mean.tolist(),
        'mu_std': mu_std.tolist(),
        'var_mean': var_mean.tolist()
    }

    # Clean memory
    torch.cuda.empty_cache()

    return metrics, all_mu, all_log_var

# Cell 8: Protein Generation and Analysis
class ProteinGenerator:
    def __init__(self, vae_model, diffusion_trainer, device):
        """
        Protein structure generator using both VAE and diffusion
models

        Args:
            vae_model: Trained VAE model
            diffusion_trainer: Trained diffusion model trainer
            device: Torch device
        """
        self.vae_model = vae_model
        self.diffusion_trainer = diffusion_trainer
        self.device = device

    def generate_from_vae(self, num_samples=10, temperature=1.0):
        """Generate protein structures using the VAE model"""
```

```
        self.vae_model.eval()

        with torch.no_grad():
            # Sample from latent space
            z = torch.randn(num_samples,
self.vae_model.latent_dim).to(self.device) * temperature

            # Decode
            samples = self.vae_model.decode(z)
            samples = torch.sigmoid(samples)

        return samples.cpu().numpy()

    def generate_from_diffusion(self, num_samples=10):
        """Generate protein structures using the diffusion model"""
        samples, _ =
self.diffusion_trainer.sample(n_samples=num_samples,
shape=self.vae_model.input_dim)
        return samples.cpu().numpy()

    def interpolate_structures(self, structure1, structure2,
num_steps=10):
        """Interpolate between two protein structures in latent
space"""
        self.vae_model.eval()

        with torch.no_grad():
            # Convert to tensors
            s1 = torch.tensor(structure1,
dtype=torch.float32).unsqueeze(0).to(self.device)
            s2 = torch.tensor(structure2,
dtype=torch.float32).unsqueeze(0).to(self.device)

            # Encode to latent space
            mu1, _ = self.vae_model.encode(s1)
            mu2, _ = self.vae_model.encode(s2)

            # Interpolate in latent space
            alphas = np.linspace(0, 1, num_steps)
            interpolations = []

            for alpha in alphas:
                mu_interp = alpha * mu1 + (1 - alpha) * mu2
                decoded =
torch.sigmoid(self.vae_model.decode(mu_interp))
                interpolations.append(decoded.cpu().numpy()[0])

        return interpolations
```

```
    def analyze_structure(self, structure):
        """Analyze a protein structure"""
        # Calculate basic statistics
        mean = np.mean(structure)
        std = np.std(structure)
        min_val = np.min(structure)
        max_val = np.max(structure)

        # Find peaks (potential binding sites or structural motifs)
        from scipy.signal import find_peaks
        peaks, _ = find_peaks(structure, height=0.5, distance=10)

        # Calculate periodicity using autocorrelation
        from scipy.signal import correlate
        autocorr = correlate(structure, structure, mode='full')
        autocorr = autocorr[len(autocorr)//2:]

        # Package results
        analysis = {
            'mean': float(mean),
            'std': float(std),
            'min': float(min_val),
            'max': float(max_val),
            'num_peaks': len(peaks),
            'peak_positions': peaks.tolist(),
            'autocorrelation': autocorr[:100].tolist()  # First 100
points
        }

        return analysis

def generate_protein_report(generator, num_samples=10,
output_dir='protein_report'):
    """Generate a comprehensive report on protein structures"""
    os.makedirs(output_dir, exist_ok=True)

    # Generate samples
    print("Generating samples from VAE...")
    vae_samples = generator.generate_from_vae(num_samples)

    print("Generating samples from diffusion model...")
    diffusion_samples = generator.generate_from_diffusion(num_samples)

    # Analyze samples
    print("Analyzing generated structures...")
    vae_analyses = [generator.analyze_structure(s) for s in
vae_samples]
    diffusion_analyses = [generator.analyze_structure(s) for s in
diffusion_samples]
```

```
    # Create interpolations
    print("Creating interpolations...")
    interpolations = generator.interpolate_structures(vae_samples[0],
vae_samples[1])

    # Plot samples
    plt.figure(figsize=(15, 10))

    # Plot samples
    plt.figure(figsize=(15, 10))

    # VAE samples
    for i in range(min(5, num_samples)):
        plt.subplot(3, 5, i+1)
        plt.plot(vae_samples[i])
        plt.title(f'VAE Sample {i+1}')
        plt.ylim(0, 1)
        plt.axis('off')

    # Diffusion samples
    for i in range(min(5, num_samples)):
        plt.subplot(3, 5, i+6)
        plt.plot(diffusion_samples[i])
        plt.title(f'Diffusion Sample {i+1}')
        plt.ylim(0, 1)
        plt.axis('off')

    # Interpolations
    for i in range(min(5, len(interpolations))):
        plt.subplot(3, 5, i+11)
        plt.plot(interpolations[i])
        plt.title(f'Interpolation {i+1}')
        plt.ylim(0, 1)
        plt.axis('off')

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, 'protein_samples.png'))
    plt.close()

    # Plot statistics
    plt.figure(figsize=(15, 10))

    # Mean values
    vae_means = [a['mean'] for a in vae_analyses]
    diff_means = [a['mean'] for a in diffusion_analyses]

    plt.subplot(2, 2, 1)
    plt.boxplot([vae_means, diff_means], labels=['VAE', 'Diffusion'])
```

```
    plt.title('Mean Values')

    # Standard deviations
    vae_stds = [a['std'] for a in vae_analyses]
    diff_stds = [a['std'] for a in diffusion_analyses]

    plt.subplot(2, 2, 2)
    plt.boxplot([vae_stds, diff_stds], labels=['VAE', 'Diffusion'])
    plt.title('Standard Deviations')

    # Number of peaks
    vae_peaks = [a['num_peaks'] for a in vae_analyses]
    diff_peaks = [a['num_peaks'] for a in diffusion_analyses]

    plt.subplot(2, 2, 3)
    plt.boxplot([vae_peaks, diff_peaks], labels=['VAE', 'Diffusion'])
    plt.title('Number of Peaks')

    # Autocorrelation
    plt.subplot(2, 2, 4)
    plt.plot(vae_analyses[0]['autocorrelation'], label='VAE')
    plt.plot(diffusion_analyses[0]['autocorrelation'],
label='Diffusion')
    plt.title('Autocorrelation (Sample 1)')
    plt.legend()

    plt.tight_layout()
    plt.savefig(os.path.join(output_dir, 'protein_statistics.png'))
    plt.close()

    # Save analyses to JSON
    analyses = {
        'vae_samples': vae_analyses,
        'diffusion_samples': diffusion_analyses,
        'timestamp': datetime.now().strftime("%Y-%m-%d %H:%M:%S")
    }

    with open(os.path.join(output_dir, 'protein_analyses.json'), 'w')
as f:
        json.dump(analyses, f, indent=4)

    print(f"Protein report generated in {output_dir}")
    return analyses

# Cell 9: Complete Pipeline (Updated for Stability)
def run_helixsynth_pipeline(input_dim=400, hidden_dim=256,
latent_dim=32,
                            batch_size=128, vae_epochs=50,
diffusion_epochs=30,
                            use_synthetic_data=True, num_samples=1000,
```

```
                           output_dir='helixsynth_output',
                           ):
    """
    Run the complete HelixSynth-Pro pipeline

    Args:
        input_dim: Dimension of protein structure vectors
        hidden_dim: Hidden dimension for models
        latent_dim: Latent dimension for VAE
        batch_size: Batch size for training
        vae_epochs: Number of epochs for VAE training
        diffusion_epochs: Number of epochs for diffusion model
training
        use_synthetic_data: Whether to use synthetic data
        num_samples: Number of samples if using synthetic data
        output_dir: Directory for all outputs
        noise_schedule: 'linear' or 'cosine'
        warmup_steps: Number of warm-up steps for learning rate
    """
    # Create output directories
    os.makedirs(output_dir, exist_ok=True)
    model_dir = os.path.join(output_dir, 'models')
    data_dir = os.path.join(output_dir, 'data')
    vis_dir = os.path.join(output_dir, 'visualizations')
    report_dir = os.path.join(output_dir, 'protein_report')

    os.makedirs(model_dir, exist_ok=True)
    os.makedirs(data_dir, exist_ok=True)
    os.makedirs(vis_dir, exist_ok=True)
    os.makedirs(report_dir, exist_ok=True)

    # Set up device and seed
    device = setup_gpu()
    seed = set_seed(42)

    # Create or load data
    if use_synthetic_data:
        print("Creating synthetic dataset...")
        train_dir = os.path.join(data_dir, 'train')
        val_dir = os.path.join(data_dir, 'val')
        test_dir = os.path.join(data_dir, 'test')

        os.makedirs(train_dir, exist_ok=True)
        os.makedirs(val_dir, exist_ok=True)
        os.makedirs(test_dir, exist_ok=True)

        # Create datasets with different sizes
        create_synthetic_dataset(train_dir, int(num_samples * 0.7),
input_dim, structured=True)
```

```
        create_synthetic_dataset(val_dir, int(num_samples * 0.15),
input_dim, structured=True)
        create_synthetic_dataset(test_dir, int(num_samples * 0.15),
input_dim, structured=True)

        # Create data loaders
        train_loader, val_loader, test_loader = create_data_loaders(
            data_path=data_dir,
            batch_size=batch_size,
            num_workers=4
        )
    else:
        # Assume data is already available
        print("Loading existing dataset...")
        train_loader, val_loader, test_loader = create_data_loaders(
            data_path=data_dir,
            batch_size=batch_size,
            num_workers=4
        )

    # Initialize and train VAE model
    print("\n" + "="*50)
    print("Initializing and training VAE model...")
    print("="*50)

    vae_model = ProteinVAE(input_dim, hidden_dim,
latent_dim).to(device)
    vae_model, vae_train_losses, vae_val_losses = train_model(
        vae_model, train_loader, val_loader, device,
        epochs=vae_epochs,
        model_dir=model_dir
    )

    # Plot VAE training history
    plot_training_history(
        vae_train_losses, vae_val_losses,
        save_path=os.path.join(vis_dir, 'vae_training_history.png')
    )

    # Initialize and train diffusion model
    print("\n" + "="*50)
    print("Initializing and training diffusion model...")
    print("="*50)

    diffusion_model = DiffusionModel(input_dim, hidden_dim).to(device)
    diffusion_model, diffusion_trainer, diff_train_losses,
diff_val_losses = train_diffusion_model(
        diffusion_model, train_loader, val_loader, device,
        epochs=diffusion_epochs,
```

```
        lr=1e-4,
        weight_decay=1e-5,
        use_amp=True,
        model_dir=model_dir,
    )

    # Plot diffusion training history
    plot_training_history(
        diff_train_losses, diff_val_losses,
        save_path=os.path.join(vis_dir,
'diffusion_training_history.png')
    )

    # Create visualizations
    print("\n" + "="*50)
    print("Creating model visualizations...")
    print("="*50)

    metrics = create_visualization_pipeline(
        vae_model, diffusion_model, diffusion_trainer,
        test_loader, device, output_dir=vis_dir
    )

    # Generate protein structures and report
    print("\n" + "="*50)
    print("Generating protein structures and analysis report...")
    print("="*50)

    protein_generator = ProteinGenerator(vae_model, diffusion_trainer,
device)
    protein_analyses = generate_protein_report(
        protein_generator, num_samples=10, output_dir=report_dir
    )

    # Final summary
    print("\n" + "="*50)
    print("HelixSynth-Pro Pipeline Complete!")
    print("="*50)
    print(f"Output directory: {output_dir}")
    print(f"VAE model disentanglement score:
{metrics['disentanglement_score']:.4f}")
    print(f"Test reconstruction error:
{metrics['reconstruction_error']:.4f}")
    print(f"Generated {len(protein_analyses['vae_samples'])} protein
structures with VAE")
    print(f"Generated {len(protein_analyses['diffusion_samples'])}
protein structures with diffusion model")

    return {
```

```python
        'vae_model': vae_model,
        'diffusion_model': diffusion_model,
        'diffusion_trainer': diffusion_trainer,
        'metrics': metrics,
        'protein_analyses': protein_analyses
    }

# Cell 10: Execute Pipeline (Fixed for Kaggle)
if __name__ == "__main__":
    # Run the complete pipeline with Kaggle working directory
    results = run_helixsynth_pipeline(
        input_dim=400,
        hidden_dim=256,
        latent_dim=32,
        batch_size=128,
        vae_epochs=30,  # Reduced for faster execution
        diffusion_epochs=20,  # Reduced for faster execution
        use_synthetic_data=True,
        num_samples=1000,
        output_dir='/kaggle/working/'
    )

    # Access models and results
    vae_model = results['vae_model']
    diffusion_model = results['diffusion_model']
    diffusion_trainer = results['diffusion_trainer']
    metrics = results['metrics']
    protein_analyses = results['protein_analyses']

    print("\nPipeline execution complete!")
```