```
#
================================================================================
======
# Cell 1: Setup and Dependencies
#
================================================================================
======

print("--- Part 1: Setup and Dependencies ---")
!pip install torch datasets transformers huggingface_hub tokenizers matplotlib -q

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
from transformers import PreTrainedTokenizerFast
from tokenizers import Tokenizer
from tokenizers.models import BPE
from tokenizers.trainers import BpeTrainer
from tokenizers.pre_tokenizers import Whitespace
from datasets import load_dataset
import copy
from tqdm import tqdm
import math
import os
import numpy as np
import matplotlib.pyplot as plt
from google.colab import drive

# Mount Google Drive for persistent storage
try:
    drive.mount('/content/drive')
    DRIVE_MOUNTED = True
    print("Google Drive mounted successfully.")
except:
    DRIVE_MOUNTED = False
    print("Could not mount Google Drive. Results will be saved to the local Colab runtime.")

# Verify that a GPU is available and set the device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Using device: {device}")

# Enable TF32 for A100 GPUs for a free performance boost
if torch.cuda.is_available() and torch.cuda.get_device_capability(0)[0] >= 8:
```

```
    print("A100 GPU detected. Enabling TF32.")
    torch.set_float32_matmul_precision('high')

# Enable benchmark mode for cuDNN for potentially faster convolutions and RNNs
torch.backends.cudnn.benchmark = True

print("\nSetup complete.")
_____
#
================================================================================
======
# Cell 2: Core Framework - Component Classes
#
================================================================================
======

class Expert(nn.Module):
    """An expert module with configurable random weight initialization."""
    def __init__(self, d_model, d_ffn, init_method='default'):
        super().__init__()
        self.w_down = nn.Linear(d_model, d_ffn)
        self.activation = nn.GELU()
        self.w_up = nn.Linear(d_ffn, d_model)

        if init_method == 'orthogonal':
            nn.init.orthogonal_(self.w_down.weight)
            nn.init.orthogonal_(self.w_up.weight)
        elif init_method == 'sparse':
            nn.init.sparse_(self.w_down.weight, sparsity=0.5)
            nn.init.sparse_(self.w_up.weight, sparsity=0.5)
        elif init_method != 'default':
            raise ValueError(f"Unknown initialization method: {init_method}")

        nn.init.zeros_(self.w_down.bias)
        nn.init.zeros_(self.w_up.bias)

    def forward(self, x):
        return self.w_up(self.activation(self.w_down(x)))

class StandardFFN(nn.Module):
    """A standard Feed-Forward Network block for the baseline Transformer."""
    def __init__(self, d_model):
        super().__init__()
        d_ffn = d_model * 4
```

```
        self.linear1 = nn.Linear(d_model, d_ffn)
        self.activation = nn.GELU()
        self.linear2 = nn.Linear(d_ffn, d_model)

    def forward(self, x):
        return self.linear2(self.activation(self.linear1(x)))

class SOMELayer(nn.Module):
    def __init__(self, d_model, some_config):
        super().__init__()
        self.d_model = d_model
        self.num_experts = some_config['num_experts']
        self.d_ffn = some_config['d_ffn']
        self.top_k = some_config['top_k']

        # Heuristic update parameters
        self.alpha = some_config['alpha']
        self.beta = some_config['beta']
        self.delta = some_config['delta']

        # Key management parameters
        self.theta_percentile = some_config['theta_percentile']
        self.warmup_steps = some_config['warmup_steps']
        self.ema_decay = some_config['ema_decay']

        self.ablation_flags = some_config.get('ablation_flags', {'use_alpha': True, 'use_beta': True,
'use_delta': True})

        self.query_network = nn.Linear(d_model, d_model)

        keys = torch.randn(self.num_experts, d_model)
        self.register_buffer("key_store", F.normalize(keys, p=2, dim=-1))
        self.register_buffer("usage_count", torch.zeros(self.num_experts))
        self.register_buffer("steps", torch.tensor([0], dtype=torch.long))

        self.experts = nn.ModuleList([Expert(d_model, self.d_ffn,
init_method=some_config['init_method']) for _ in range(self.num_experts)])

        for expert in self.experts:
            for param in expert.parameters():
                param.requires_grad = False

        if self.top_k > 1:
```

```
        self.register_buffer("peer_pull_indices", torch.combinations(torch.arange(self.top_k),
r=2))

    def forward(self, x, temperature=1.0):
        batch_size, seq_len, _ = x.shape
        x_flat = x.view(-1, self.d_model)

        queries_raw = self.query_network(x_flat)
        queries = F.normalize(queries_raw, p=2, dim=-1)

        scores = torch.matmul(queries, self.key_store.t())
        top_k_scores, top_k_indices = torch.topk(scores, self.top_k, dim=-1)
        gating_weights = F.softmax(top_k_scores / temperature, dim=-1)

        flat_top_k_indices = top_k_indices.view(-1)

        sorted_indices, permutation_map = torch.sort(flat_top_k_indices)
        unique_expert_ids, counts = torch.unique_consecutive(sorted_indices,
return_counts=True)

        flat_inputs = x_flat.repeat_interleave(self.top_k, dim=0)
        permuted_inputs = flat_inputs[permutation_map]
        split_inputs = torch.split(permuted_inputs, counts.tolist(), dim=0)

        output_chunks = []
        for i, expert_id in enumerate(unique_expert_ids):
            output_chunks.append(self.experts[expert_id](split_inputs[i]))

        concatenated_outputs = torch.cat(output_chunks, dim=0)
        inverse_permutation_map = torch.argsort(permutation_map)
        expert_outputs = concatenated_outputs[inverse_permutation_map]

        weighted_outputs = (expert_outputs.view(-1, self.top_k, self.d_model) *
gating_weights.unsqueeze(-1)).sum(dim=1)
        final_output = weighted_outputs.view(batch_size, seq_len, self.d_model)

        return x + final_output, queries, top_k_indices

    @torch.no_grad()
    def update_keys(self, queries, top_k_indices):
        self.steps += 1
        unique_indices, counts = torch.unique(top_k_indices, return_counts=True)
        self.usage_count.mul_(self.ema_decay)
        self.usage_count.index_add_(0, unique_indices, (1.0 - self.ema_decay) * counts.float())
```

```
        if self.ablation_flags.get('use_alpha', True):
            for i in range(self.top_k):
                indices = top_k_indices[:, i]
                inertia = 1.0 + self.usage_count[indices]
                alpha_effective = self.alpha / inertia.unsqueeze(-1)
                update_vec = queries - self.key_store[indices]
                self.key_store.index_add_(0, indices, alpha_effective * update_vec)

        if self.top_k > 1 and self.ablation_flags.get('use_beta', True):
            indices_i = top_k_indices[:, self.peer_pull_indices[:, 0]].reshape(-1)
            indices_j = top_k_indices[:, self.peer_pull_indices[:, 1]].reshape(-1)
            keys_i, keys_j = self.key_store[indices_i], self.key_store[indices_j]
            inertia_i = (1.0 + self.usage_count[indices_i]).unsqueeze(-1)
            inertia_j = (1.0 + self.usage_count[indices_j]).unsqueeze(-1)
            beta_effective = self.beta / torch.min(inertia_i, inertia_j)

            update_vec_i = beta_effective * (keys_j - keys_i)
            update_vec_j = beta_effective * (keys_i - keys_j)
            self.key_store.index_add_(0, indices_i, update_vec_i)
            self.key_store.index_add_(0, indices_j, update_vec_j)

        if self.steps > self.warmup_steps and self.ablation_flags.get('use_delta', True):
            active_usage_counts = self.usage_count[self.usage_count > 0]
            if active_usage_counts.numel() > 0:
                dynamic_theta = torch.quantile(active_usage_counts.float(), self.theta_percentile)
                low_usage_mask = self.usage_count < dynamic_theta
                self.key_store[low_usage_mask] *= (1.0 - self.delta)

        self.key_store.data = F.normalize(self.key_store.data, p=2, dim=-1)

class TransformerBlock(nn.Module):
    def __init__(self, d_model, num_heads, some_config, architecture_type):
        super().__init__()
        self.attention = nn.MultiheadAttention(d_model, num_heads, batch_first=True)
        self.norm1 = nn.LayerNorm(d_model)
        self.norm2 = nn.LayerNorm(d_model)

        self.architecture_type = architecture_type
        if self.architecture_type == 'some_candidate':
            self.ff_layer = SOMELayer(d_model, some_config)
        else:
            self.ff_layer = StandardFFN(d_model)
```

```
    def forward(self, x, temperature=1.0):
        seq_len = x.size(1)
        mask = torch.triu(torch.ones(seq_len, seq_len, device=x.device) * float('-inf'), diagonal=1)
        attn_output, _ = self.attention(x, x, x, attn_mask=mask, need_weights=False)
        x = self.norm1(x + attn_output)

        if self.architecture_type == 'some_candidate':
            ff_output, queries, top_k_indices = self.ff_layer(x, temperature=temperature)
            x = self.norm2(ff_output)
            return x, queries, top_k_indices
        else:
            ff_output = self.ff_layer(x)
            x = self.norm2(x + ff_output)
            return x, None, None

class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        position = torch.arange(max_len).unsqueeze(1)
        div_term = torch.exp(torch.arange(0, d_model, 2) * (-math.log(10000.0) / d_model))
        pe = torch.zeros(1, max_len, d_model)
        pe[0, :, 0::2] = torch.sin(position * div_term)
        pe[0, :, 1::2] = torch.cos(position * div_term)
        self.register_buffer('pe', pe)

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]

class UniversalTransformer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config
        model_config = config['model']
        some_config = config.get('some_layer')

        self.embedding = nn.Embedding(model_config['vocab_size'], model_config['d_model'])
        self.pos_encoder = PositionalEncoding(model_config['d_model'], model_config['seq_len'])

        self.layers = nn.ModuleList([
            TransformerBlock(
                model_config['d_model'],
                model_config['num_heads'],
                some_config,
                config['architecture_type']
```

```
            ) for _ in range(model_config['num_layers'])
        ])

        self.fc_out = nn.Linear(model_config['d_model'], model_config['vocab_size'])
        self.d_model = model_config['d_model']

    def forward(self, x, temperature=1.0):
        x = self.embedding(x) * math.sqrt(self.d_model)
        x = self.pos_encoder(x)

        all_queries, all_indices = [], []
        for layer in self.layers:
            x, queries, top_k_indices = layer(x, temperature=temperature)
            if queries is not None:
                all_queries.append(queries)
                all_indices.append(top_k_indices)

        return self.fc_out(x), all_queries, all_indices

    @torch.no_grad()
    def update_all_keys(self, all_queries, all_indices):
        if self.config['architecture_type'] != 'some_candidate':
            return

        for i, layer_block in enumerate(self.layers):
            if isinstance(layer_block.ff_layer, SOMELayer):
                queries = all_queries[i].view(-1, layer_block.ff_layer.d_model)
                indices = all_indices[i].view(-1, layer_block.ff_layer.top_k)
                layer_block.ff_layer.update_keys(queries, indices)

print("Core model components defined.")
_____
#
==========================================================================
======
# Cell 3: Data Preparation and Training/Evaluation Functions
#
==========================================================================
======

class LanguageModelDataset(Dataset):
    def __init__(self, tokenized_data):
        self.data = tokenized_data
```

```
    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        item = self.data[idx]
        inputs = torch.tensor(item['input_ids'])
        targets = inputs.clone()
        targets[:-1] = inputs[1:]
        targets[-1] = -100 # Ignore loss for the last token prediction
        return inputs, targets

def prepare_data(config):
    print("\n--- Part 2: Data Preparation & Configuration ---")
    tokenizer_path = "universal_bpe_tokenizer.json"

    if not os.path.exists(tokenizer_path):
        print("Training universal BPE tokenizer...")
        # Use TinyStories as the base for the tokenizer as it's general-purpose
        dataset = load_dataset("roneneldan/TinyStories", split="train", streaming=True)

        def get_training_corpus():
            iterator = iter(dataset)
            for i in range(20000): # Use 20k samples to build tokenizer
                try:
                    yield next(iterator)['text']
                except StopIteration:
                    break

        tokenizer_model = Tokenizer(BPE(unk_token="[UNK]"))
        tokenizer_model.pre_tokenizer = Whitespace()
        trainer = BpeTrainer(special_tokens=["[UNK]", "[PAD]", "[EOS]"],
vocab_size=config['model']['vocab_size'])

        tokenizer_model.train_from_iterator(get_training_corpus(), trainer=trainer)
        tokenizer_model.save(tokenizer_path)
    else:
        print("Universal tokenizer already exists. Loading from file.")

    tokenizer = PreTrainedTokenizerFast(tokenizer_file=tokenizer_path)
    tokenizer.add_special_tokens({'pad_token': '[PAD]', 'eos_token': '[EOS]'})
    print(f"Tokenizer loaded with vocab size: {tokenizer.vocab_size}")

    print(f"\nTokenizing dataset: {config['data']['dataset_name']}...")
    full_dataset = load_dataset(config['data']['dataset_name'])
```

```
    # Ensure train/validation splits exist
    train_split = 'train' if 'train' in full_dataset else list(full_dataset.keys())[0]
    val_split = 'validation' if 'validation' in full_dataset else train_split

    train_subset = full_dataset[train_split].select(range(min(len(full_dataset[train_split]),
config['data']['train_subset_size'])))
    val_subset = full_dataset[val_split].select(range(min(len(full_dataset[val_split]),
config['data']['val_subset_size'])))

    def tokenize_function(examples):
        text_with_eos = [str(s) + tokenizer.eos_token for s in examples["text"]]
        return tokenizer(text_with_eos, truncation=True, padding="max_length",
max_length=config['model']['seq_len'], return_tensors="pt")

    tokenized_train = train_subset.map(tokenize_function, batched=True,
remove_columns=train_subset.column_names, num_proc=os.cpu_count())
    tokenized_val = val_subset.map(tokenize_function, batched=True,
remove_columns=val_subset.column_names, num_proc=os.cpu_count())

    # Set format to PyTorch
    tokenized_train.set_format(type='torch', columns=['input_ids'])
    tokenized_val.set_format(type='torch', columns=['input_ids'])

    train_dataset = LanguageModelDataset(tokenized_train)
    validation_dataset = LanguageModelDataset(tokenized_val)

    num_workers = max(2, os.cpu_count() // 2 if os.cpu_count() else 2)
    train_loader = DataLoader(train_dataset, batch_size=config['data']['batch_size'], shuffle=True,
drop_last=True, num_workers=num_workers, pin_memory=True)
    validation_loader = DataLoader(validation_dataset, batch_size=config['data']['batch_size'],
drop_last=True, num_workers=num_workers, pin_memory=True)

    print(f"\nTrain dataset size (subset): {len(train_dataset)}")
    print(f"Validation dataset size (subset): {len(validation_dataset)}")
    print(f"Using {num_workers} workers for DataLoader.")

    return train_loader, validation_loader, tokenizer

def calculate_gini(usage_counts):
    counts = usage_counts.cpu().to(torch.float32).numpy()
    if np.sum(counts) == 0: return 0.0
    counts = np.sort(counts)
    n = len(counts)
```

```
    index = np.arange(1, n + 1)
    return (np.sum((2 * index - n - 1) * counts)) / (n * np.sum(counts))

def calculate_entropy(usage_counts):
    total_usage = usage_counts.sum()
    if total_usage == 0: return 0.0
    probs = usage_counts / total_usage
    probs = probs[probs > 0]
    return -torch.sum(probs * torch.log2(probs)).item()

def train_epoch(model, dataloader, optimizer, criterion, scheduler, current_temp,
tokenizer_vocab_size):
    model.train()
    total_loss = 0
    scaler = torch.cuda.amp.GradScaler()
    progress_bar = tqdm(dataloader, desc="Training", leave=False)

    for inputs, targets in progress_bar:
        inputs, targets = inputs.to(device, non_blocking=True), targets.to(device,
non_blocking=True)

        optimizer.zero_grad(set_to_none=True)

        with torch.cuda.amp.autocast():
            logits, queries, indices = model(inputs, temperature=current_temp)
            loss = criterion(logits.view(-1, tokenizer_vocab_size), targets.view(-1))

        scaler.scale(loss).backward()
        scaler.unscale_(optimizer)
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        scaler.step(optimizer)
        scaler.update()
        scheduler.step()

        if model.config['architecture_type'] == 'some_candidate':
            model.update_all_keys(queries, indices)

        total_loss += loss.item()
        progress_bar.set_postfix({'loss': f'{loss.item():.4f}', 'lr': f'{scheduler.get_last_lr()[0]:.1e}'})

    return total_loss / len(dataloader)

def evaluate_epoch(model, dataloader, criterion, tokenizer_vocab_size):
    model.eval()
```

```
    total_loss = 0
    progress_bar = tqdm(dataloader, desc="Evaluating", leave=False)

    with torch.no_grad():
        for inputs, targets in progress_bar:
            inputs, targets = inputs.to(device, non_blocking=True), targets.to(device,
non_blocking=True)
            with torch.cuda.amp.autocast():
                logits, _, _ = model(inputs, temperature=0.5) # Sharpen during eval for stability
                loss = criterion(logits.view(-1, tokenizer_vocab_size), targets.view(-1))
            total_loss += loss.item()
            progress_bar.set_postfix({'loss': f'{loss.item():.4f}'})

    return total_loss / len(dataloader)

def plot_and_save_metrics(train_losses, val_losses, some_metrics, config, save_dir):
    epochs = len(train_losses)
    plt.figure(figsize=(12, 10))

    # Plot Loss
    plt.subplot(2, 1, 1)
    plt.plot(range(1, epochs + 1), train_losses, 'b-o', label='Training Loss')
    plt.plot(range(1, epochs + 1), val_losses, 'r-o', label='Validation Loss')
    title = f"Loss Curve for {config['run_name']}"
    plt.title(title)
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.xticks(range(1, epochs + 1))

    # Plot SoME Metrics if available
    if config['architecture_type'] == 'some_candidate':
        gini_coeffs = [m['gini'] for m in some_metrics]
        entropy_vals = [m['entropy'] for m in some_metrics]
        plt.subplot(2, 1, 2)
        ax1 = plt.gca()
        ax2 = ax1.twinx()

        ax1.plot(range(1, epochs + 1), gini_coeffs, 'g-s', label='Gini Coefficient')
        ax2.plot(range(1, epochs + 1), entropy_vals, 'm-^', label='Shannon Entropy')

        plt.title(f"SoME Expert Metrics (Middle Layer)")
        ax1.set_xlabel('Epochs')
```

```
        ax1.set_ylabel('Gini Coefficient', color='g')
        ax2.set_ylabel('Shannon Entropy', color='m')
        ax1.tick_params(axis='y', labelcolor='g')
        ax2.tick_params(axis='y', labelcolor='m')
        ax1.legend(loc='upper left')
        ax2.legend(loc='upper right')
        ax1.grid(True)
        plt.xticks(range(1, epochs + 1))

    plt.tight_layout()
    filename = os.path.join(save_dir, f"metrics_{config['run_name']}.png")
    plt.savefig(filename)
    print(f"\nMetrics plot saved to {filename}")
    plt.show()

print("Helper functions for data, training, and evaluation defined.")
_____
#
================================================================================
======
# Cell 4: Main Execution Function
#
================================================================================
======

def main(config):
    """Main function to run a single, complete experiment."""

    print(f"\n--- Starting Experiment: {config['run_name']} ---")

    # Create a directory for this run's results
    base_save_dir = '/content/drive/MyDrive/SoME_Experiments' if DRIVE_MOUNTED else
'/content/SoME_Experiments'
    run_save_dir = os.path.join(base_save_dir, config['run_name'])
    os.makedirs(run_save_dir, exist_ok=True)
    print(f"Results will be saved in: {run_save_dir}")

    # 1. Data
    train_loader, val_loader, tokenizer = prepare_data(config)

    # 2. Model Initialization
    print("\n--- Part 3: Model Definition ---")
    model = UniversalTransformer(config).to(device)
```

```
    # 3. Training Setup
    print("\n--- Part 4: Training, Evaluation, and Metrics ---")
    trainable_params = sum(p.numel() for p in model.parameters() if p.requires_grad)
    total_params = sum(p.numel() for p in model.parameters())
    print(f"Total Parameters: {total_params/1e6:.2f}M")
    print(f"Trainable Parameters: {trainable_params/1e6:.2f}M ({100 * trainable_params /
total_params:.2f}%)")

    optimizer = torch.optim.AdamW([p for p in model.parameters() if p.requires_grad],
lr=config['training']['learning_rate'], betas=(0.9, 0.95), weight_decay=0.1)
    criterion = nn.CrossEntropyLoss(ignore_index=-100)
    total_steps = len(train_loader) * config['training']['num_epochs']
    scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=total_steps)

    # 4. Training Loop
    train_losses, val_losses = [], []
    some_metrics_log = []
    best_val_loss = float('inf')
    model_save_path = os.path.join(run_save_dir, f"best_model_{config['run_name']}.pth")

    for epoch in range(config['training']['num_epochs']):
        print(f"\n--- Epoch {epoch+1}/{config['training']['num_epochs']} ---")

        train_loss = train_epoch(model, train_loader, optimizer, criterion, scheduler,
config['training']['training_temp'], tokenizer.vocab_size)
        val_loss = evaluate_epoch(model, val_loader, criterion, tokenizer.vocab_size)
        perplexity = math.exp(val_loss)

        train_losses.append(train_loss)
        val_losses.append(val_loss)

        print(f"Epoch {epoch+1}: Train Loss = {train_loss:.4f}, Val Loss = {val_loss:.4f}, Val
Perplexity = {perplexity:.2f}")

        if config['architecture_type'] == 'some_candidate':
            middle_layer_idx = config['model']['num_layers'] // 2
            usage_counts = model.layers[middle_layer_idx].ff_layer.usage_count
            gini_coeff = calculate_gini(usage_counts)
            entropy_val = calculate_entropy(usage_counts)
            some_metrics_log.append({'gini': gini_coeff, 'entropy': entropy_val})
            print(f"  Middle Layer Expert Metrics: Gini = {gini_coeff:.3f}, Entropy = {entropy_val:.3f}")

        if val_loss < best_val_loss:
            best_val_loss = val_loss
```

```
            torch.save(model.state_dict(), model_save_path)
            print(f"  -> New best model saved to {model_save_path}")

    # 5. Finalization
    print(f"\n--- Training Complete for {config['run_name']} ---")
    final_perplexity = math.exp(best_val_loss)
    print(f"Best Validation Loss: {best_val_loss:.4f}")
    print(f"Final Validation Perplexity: {final_perplexity:.2f}")

    # Save a summary file
    summary_path = os.path.join(run_save_dir, "summary.txt")
    with open(summary_path, 'w') as f:
        f.write(f"Run Name: {config['run_name']}\n")
        f.write(f"Best Validation Loss: {best_val_loss:.4f}\n")
        f.write(f"Final Validation Perplexity: {final_perplexity:.2f}\n")

    plot_and_save_metrics(train_losses, val_losses, some_metrics_log, config, run_save_dir)

print("Main execution function defined.")
_____
#
================================================================================
======
# Cell 5: Experiment Configuration and Execution Control Panel
#
================================================================================
======

def get_config(architecture, dataset_name):
    """
    Generates the configuration for a specific experimental run.
    architecture: 'transformer_baseline' or 'some_candidate'
    dataset_name: 'tinystories', 'tiny_textbooks', or 'tiny_codes'
    """
    config = {
        "run_name": f"{architecture}_{dataset_name}",
        "architecture_type": architecture,
        "model": {
            "vocab_size": 8192,
            "d_model": 512,
            "num_layers": 8,
            "num_heads": 8,
            "seq_len": 512,
        },
```

```
        "data": {
            "dataset_name": {
                "tinystories": "roneneldan/TinyStories",
                "tiny_textbooks": "nampdn-ai/tiny-textbooks",
                "tiny_codes": "nampdn-ai/tiny-codes"
            },
            "train_subset_size": 20000,
            "val_subset_size": 5000,
            "batch_size": 24,
        },
        "training": {
            "num_epochs": 4,
            "learning_rate": 4e-4,
            "training_temp": 1.0,
        },
    }

    if architecture == 'some_candidate':
        config["some_layer"] = {
            "num_experts": 128,
            "d_ffn": 1536,
            "top_k": 8,
            "init_method": "sparse",
            "alpha": 0.015,
            "beta": 0.001,
            "delta": 0.001,
            "theta_percentile": 0.05,
            "warmup_steps": 400,
            "ema_decay": 0.995,
            "ablation_flags": {"use_alpha": True, "use_beta": True, "use_delta": True}
        }
    return config

# --- SELECT AND RUN YOUR EXPERIMENT ---
# Uncomment one of the following lines to choose your run, then execute the cell.

# --- TinyStories Runs ---
#config = get_config(architecture='transformer_baseline', dataset_name='tinystories')
#config = get_config(architecture='some_candidate', dataset_name='tinystories')

# --- TinyTextbooks Runs ---
#config = get_config(architecture='transformer_baseline', dataset_name='tiny_textbooks')
#config = get_config(architecture='some_candidate', dataset_name='tiny_textbooks')
```

```
# --- TinyCodes Runs ---
#config = get_config(architecture='transformer_baseline', dataset_name='tiny_codes')
#config = get_config(architecture='some_candidate', dataset_name='tiny_codes')


# --- Execute the selected experiment ---
main(config)
```