# General Inference Acceleration Strategy for Time-Series Data

Speculative Sampling/Decoding paper: https://arxiv.org/pdf/2211.17192
Timer paper: https://arxiv.org/pdf/2402.02368
Timer-XL repo: https://github.com/thuml/Timer-XL
Sundial: https://arxiv.org/abs/2502.00816 (if time permits) (how long does it take for training/finetuning? What GPU? What is the size of the dataset?)

Steps:

1. Pretraining code
2. Finetuning code (for downstream tasks) (smaller model training)

We need to train the smaller model from scratch. Target model would be Timer-XL for example

- Downsample on the size of the model (scaling the size of each layer or the number of layers) (we can decrease embedding size by 2 for example – needs to be compatible with target model)

**Further** tune the weights **after** we get the Speculative Decoding framework (weights can be fixed or not fixed (ablational))

A token in the time-series case is a patch of continuous data

Notes:
Why is the verification step by the target model parallel?
Let's say you have "word + x1 + x2 + x3". Normally you would do a forward pass of "word" then a forward pass of "word x1" then a forward pass of "word x1 x2". In this new method, you have three inputs: "word" "word x1" and "word x1 x2" and run 3 forward passes in parallel to get 3 probability distributions (this is doable with parallel tasks on GPU and parallel Transformer blocks).

## **Tasks to be done:** Speculative Decoding Implementation Plan for Timer-XL

This document outlines the concrete, executable steps to implement speculative decoding for time series prediction using the Timer-XL model as the target model and a smaller draft model for speculation.

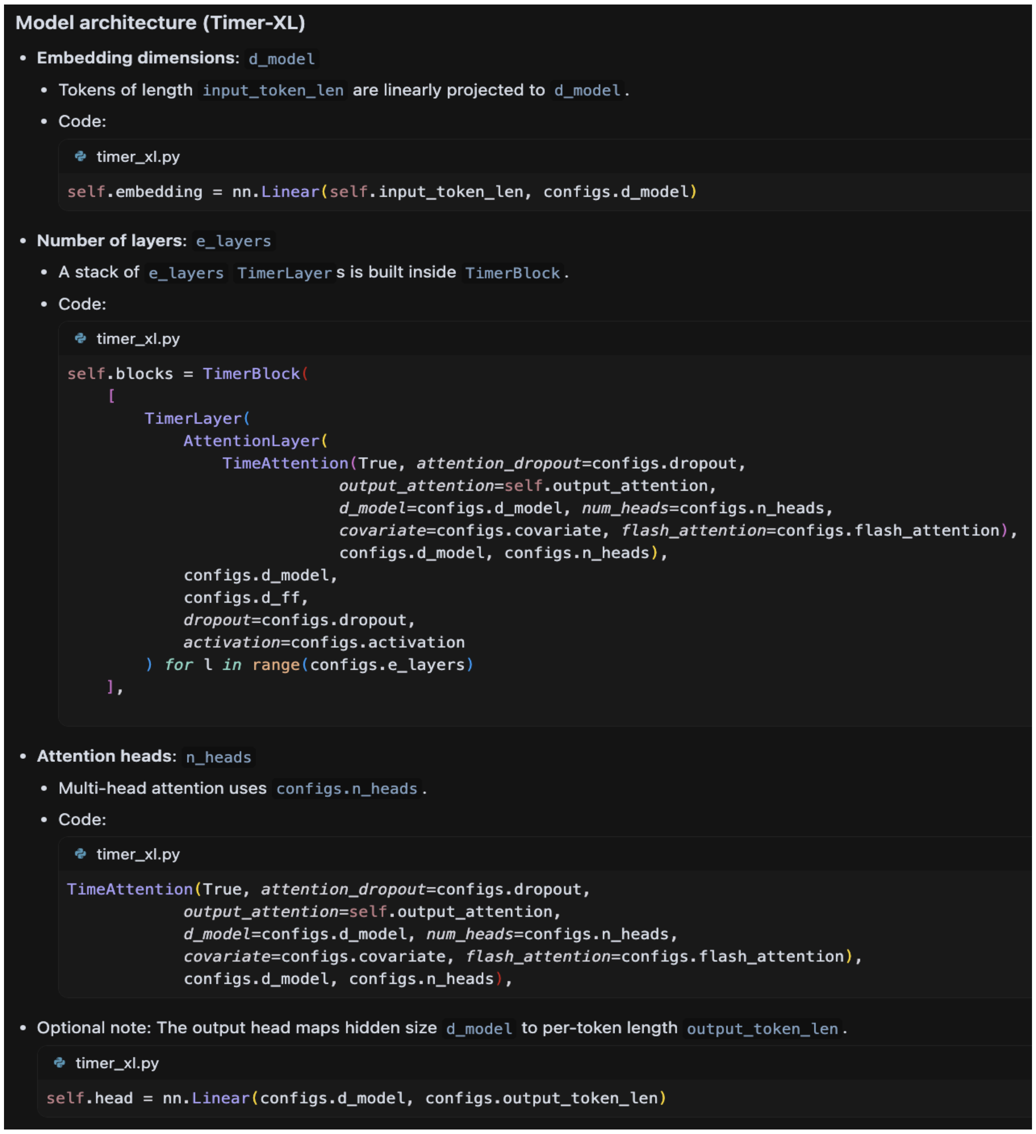
# Phase 1: Environment Setup and Codebase Analysis

## 1.1 Set up development environment and clone necessary repositories

- ☑ Install Python 3.10 environment with CUDA support
- ☑ Clone Timer-XL repository: `git clone https://github.com/thuml/Timer-XL.git`
- ☑ Clone OpenLTM repository: `git clone https://github.com/thuml/OpenLTM.git`
- ☑ Install dependencies: `transformers==4.40.1`, `torch`, `pandas`, `numpy`, `matplotlib`
- ☑ Download pre-trained Timer-XL model from HuggingFace: `thuml/timer-base-84m`
- ☑ Set up development branches for speculative decoding implementation

## 1.2 Analyze Timer-XL architecture and implementation details

- ☑ Study `models/timer_xl.py` in OpenLTM repository
- ☑ Document model architecture: embedding dimensions, number of layers, attention heads



**Model architecture (Timer-XL)**

- **Embedding dimensions:** `d_model`
  - Tokens of length `input_token_len` are linearly projected to `d_model`.
  - Code:

timer_xl.py
```python
self.embedding = nn.Linear(self.input_token_len, configs.d_model)
```

- **Number of layers:** `e_layers`
  - A stack of `e_layers` `TimerLayer`s is built inside `TimerBlock`.
  - Code:

timer_xl.py
```python
self.blocks = TimerBlock(
    [
        TimerLayer(
            AttentionLayer(
                TimeAttention(True, attention_dropout=configs.dropout,
                              output_attention=self.output_attention,
                              d_model=configs.d_model, num_heads=configs.n_heads,
                              covariate=configs.covariate, flash_attention=configs.flash_attention),
                              configs.d_model, configs.n_heads),
            configs.d_model,
            configs.d_ff,
            dropout=configs.dropout,
            activation=configs.activation
        ) for l in range(configs.e_layers)
    ],
```

- **Attention heads:** `n_heads`
  - Multi-head attention uses `configs.n_heads`.
  - Code:

timer_xl.py
```python
TimeAttention(True, attention_dropout=configs.dropout,
              output_attention=self.output_attention,
              d_model=configs.d_model, num_heads=configs.n_heads,
              covariate=configs.covariate, flash_attention=configs.flash_attention),
              configs.d_model, configs.n_heads),
```

- Optional note: The output head maps hidden size `d_model` to per-token length `output_token_len`.

timer_xl.py
```python
self.head = nn.Linear(configs.d_model, configs.output_token_len)
```

☑ Analyze TimeAttention mechanism implementation

A standard Transformer is designed to process a 1D sequence of tokens, like words in a sentence. However, multivariate time series data is inherently 2D: you have the **time dimension** (past to future) and the **variable dimension** (e.g., temperature, humidity, pressure).

Simply flattening this 2D data into a long 1D sequence confuses the Transformer. It doesn't inherently know that token #5 and token #10 might be from the same variable at different times, or that token #5 and token #6 might be from different variables at the same time.

**TimeAttention** is a specialized self-attention mechanism designed to make the Transformer aware of this 2D structure.

TimeAttention's goal is to capture two types of relationships simultaneously:

- **Intra-series dependencies:** How a variable's past values influence its future values (the pattern *within* the temperature series).
- **Inter-series dependencies:** How other variables influence a target variable (how air pressure affects temperature).

It achieves this by cleverly constructing its attention mask using a mathematical operation called the **Kronecker product (**$\otimes$**)**. The final attention mask is created by combining two simpler, smaller masks:

1. **The Variable Dependency Matrix (`C`):** This small square matrix defines which variables are allowed to "pay attention" to which other variables.
    - For a standard multivariate forecasting task where every variable can influence every other variable, this matrix is simply an all-ones matrix.
    - For a covariate task, you could customize it. For example, to predict variable A using covariate B, the matrix would be set up so that A can attend to both A and B, but B can only attend to itself.
2. **The Temporal Causal Mask (`T`):** This is the standard mask used in decoder-only Transformers. It's a triangular matrix that ensures a token at a given time step can only pay attention to previous time steps, not future ones. This preserves the principle of causality.

**The Magic of the Kronecker Product (**$\otimes$**)**

The Kronecker product combines these two matrices to create the final, large attention mask. As illustrated in

**Figure 2 of the `Timer-XL` paper**, this operation essentially takes the small temporal mask `T` and uses it as a "template" to fill in a larger block matrix whose structure is defined by the variable matrix `C`.

The resulting mask has a repeating block-diagonal structure. This structure perfectly encodes the rules for multivariate next-token prediction: a token for a specific variable at time `i` can attend to:

- All *previous* tokens (`< i`) from its **own** variable.
- All *previous* tokens (`< i`) from **all other** variables.

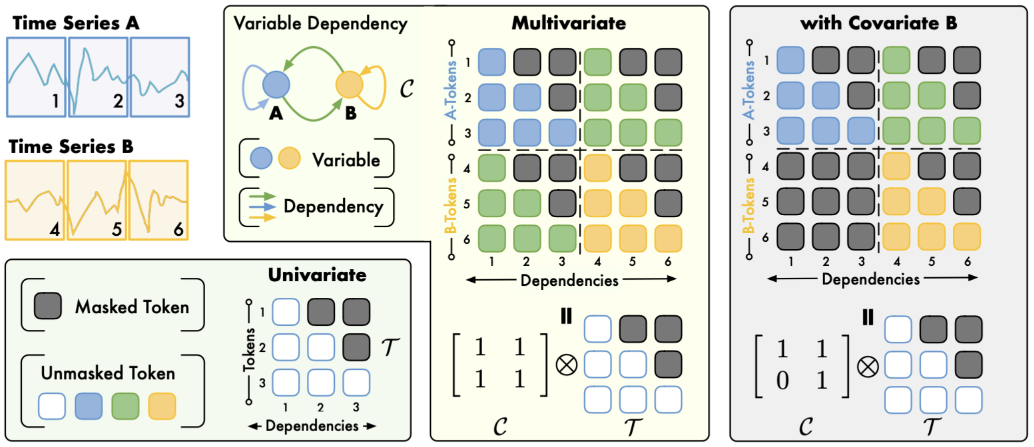
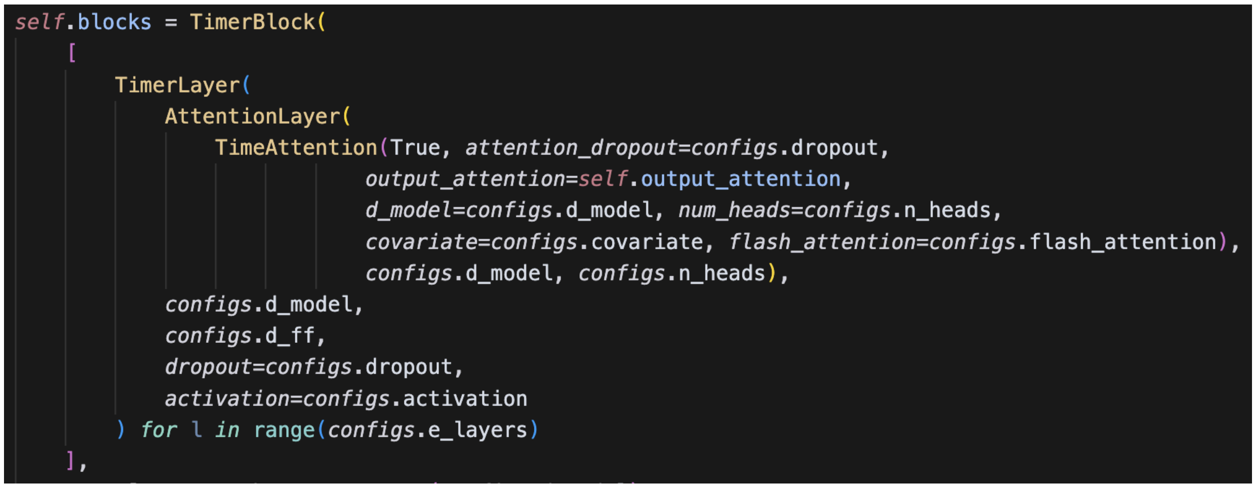


Figure 2: Illustration of TimeAttention. For univariate series, temporal mask $\mathcal{T}$ keeps the causality. Given multivariate patch tokens sorted in a temporal-first order, we adopt the variable dependencies $\mathcal{C}$, an all-one matrix, as the left-operand of Kronecker product, expanding temporal mask to a block matrix, which exactly reflects dependencies of multivariate next token prediction. The formulation is also generalizable to univariate and covariate-informed contexts with pre-defined variable dependency.

To complete the picture, TimeAttention also uses specialized position embeddings to help the model distinguish between the time and variable dimensions:

- **Temporal Dimension:** It uses **RoPE (Rotary Position Embedding)** to encode the chronological order of the patches.
- **Variable Dimension:** It uses two simple learnable scalars to tell a token whether it's looking at another token from its own series (endogenous) or from a different series (exogenous). This clever trick maintains permutation-equivalence, meaning the order in which you feed the variables doesn't matter.

Code implementation has: block structure (TimerLayer), multi-head projections/merging (AttentionLayer), the attention computation itself (TimeAttention).



```
self.blocks = TimerBlock(
    [
        TimerLayer(
            AttentionLayer(
                TimeAttention(True, attention_dropout=configs.dropout,
                              output_attention=self.output_attention,
                              d_model=configs.d_model, num_heads=configs.n_heads,
                              covariate=configs.covariate, flash_attention=configs.flash_attention),
                              configs.d_model, configs.n_heads),
            configs.d_model,
            configs.d_ff,
            dropout=configs.dropout,
            activation=configs.activation
        ) for l in range(configs.e_layers)
    ],
```

- [ ] Understand patch tokenization for time series (how continuous data becomes tokens)

The core idea is to treat a time series not as a sequence of individual data points, but as a sequence of small "chunks" or "patches." This is analogous to how we read a sentence word-by-word instead of letter-by-letter.

## Step 1: Slicing the Time Series into Patches

First, a long, continuous time series is divided into fixed-size, non-overlapping segments. These segments are called

**patches**.

- **Definition**: A time series token (or patch) is defined as `P` consecutive time points from the series.
- **Example**: Imagine you have a time series with 12 data points and a patch size `P=4`.
  - **Original Series**: `[1.2, 1.5, 1.8, 1.6, 2.1, 2.3, 2.2, 2.5, 2.9, 2.8, 2.6, 2.4]`
  - This series would be "chopped" into three patches:
    - Patch 1: `[1.2, 1.5, 1.8, 1.6]`
    - Patch 2: `[2.1, 2.3, 2.2, 2.5]`
    - Patch 3: `[2.9, 2.8, 2.6, 2.4]`

The model now sees a sequence of 3 items instead of 12, making it much more efficient.

## Step 2: Converting Patches into Embeddings (Creating the "Token")

A Transformer can't work with a simple list of numbers like `[1.2, 1.5, 1.8, 1.6]`. It needs a high-dimensional vector representation for each item in its input sequence. This process is called **embedding**.

- **Process**: Each patch (which is a vector of size `P`) is fed through a linear layer or a small MLP (Multi-Layer Perceptron), often called a "PatchEmbed" layer.
- **Transformation**: This embedding layer transforms the patch vector of length `P` into a high-dimensional embedding vector of length `D` (e.g., `D=512`).
- **The Result**: This final `D`-dimensional vector *is the token* that the Transformer's self-attention mechanism processes. The model now has a sequence of rich, high-dimensional vectors, where each vector represents a patch of the original time series.

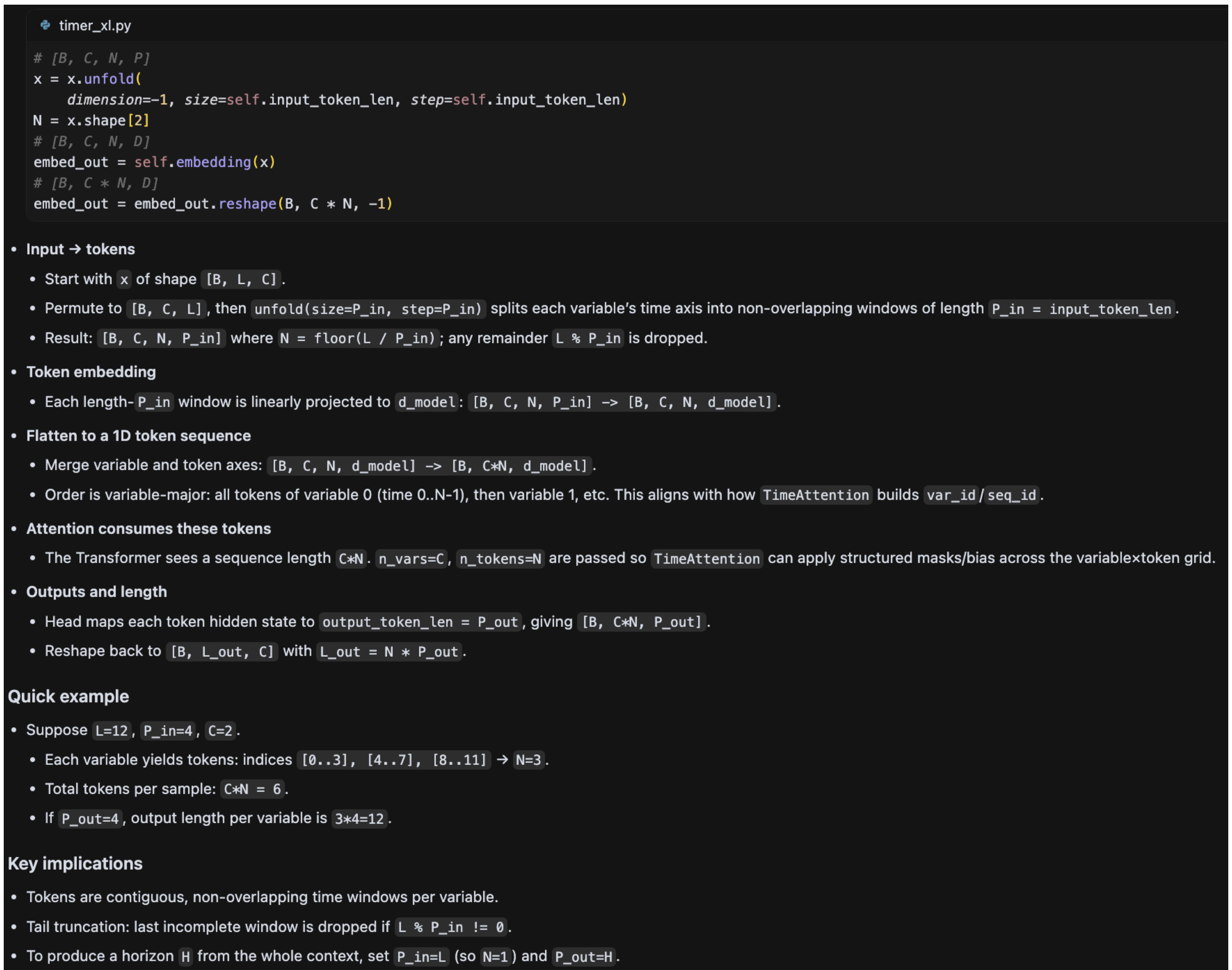
The Sundial paper notes that to handle series whose lengths aren't perfectly divisible by the patch size, the series is padded at the beginning, and a binary mask is concatenated to the patch before the embedding step to let the model know which points are real and which are padding.

## Why is this approach used?

- **"Native" Continuous Representation**: This is a key advantage highlighted in the Sundial paper. Unlike some methods that convert continuous values into discrete integers (e.g., turning
  `8.12` into a token `23`), patch tokenization works directly on the original, continuous-valued data. This avoids information loss that can happen during quantization.
- **Efficiency**: It significantly reduces the length of the sequence the Transformer has to process. Instead of attending to thousands of individual time points, the model might only attend to a few hundred patches, making it much faster and less memory-intensive.
- **Capturing Local Patterns**: Each patch inherently captures local information about the series' shape, trend, and seasonality within that small window. This provides a more meaningful input to the model than a single, isolated data point.

### Patch tokenization code implementation:

- B: batch size (number of sequences in the batch)
- L: sequence length (number of time steps per sequence)
- C: number of variables/features (channels) per time step



timer_xl.py

```python
# [B, C, N, P]
x = x.unfold(
    dimension=-1, size=self.input_token_len, step=self.input_token_len)
N = x.shape[2]
# [B, C, N, D]
embed_out = self.embedding(x)
# [B, C * N, D]
embed_out = embed_out.reshape(B, C * N, -1)
```

- **Input → tokens**
  - Start with `x` of shape `[B, L, C]`.
  - Permute to `[B, C, L]`, then `unfold(size=P_in, step=P_in)` splits each variable's time axis into non-overlapping windows of length `P_in = input_token_len`.
  - Result: `[B, C, N, P_in]` where `N = floor(L / P_in)`; any remainder `L % P_in` is dropped.
- **Token embedding**
  - Each length-`P_in` window is linearly projected to `d_model`: `[B, C, N, P_in] -> [B, C, N, d_model]`.
- **Flatten to a 1D token sequence**
  - Merge variable and token axes: `[B, C, N, d_model] -> [B, C*N, d_model]`.
  - Order is variable-major: all tokens of variable 0 (time 0..N-1), then variable 1, etc. This aligns with how `TimeAttention` builds `var_id`/`seq_id`.
- **Attention consumes these tokens**
  - The Transformer sees a sequence length `C*N`. `n_vars=C`, `n_tokens=N` are passed so `TimeAttention` can apply structured masks/bias across the variable×token grid.
- **Outputs and length**
  - Head maps each token hidden state to `output_token_len = P_out`, giving `[B, C*N, P_out]`.
  - Reshape back to `[B, L_out, C]` with `L_out = N * P_out`.

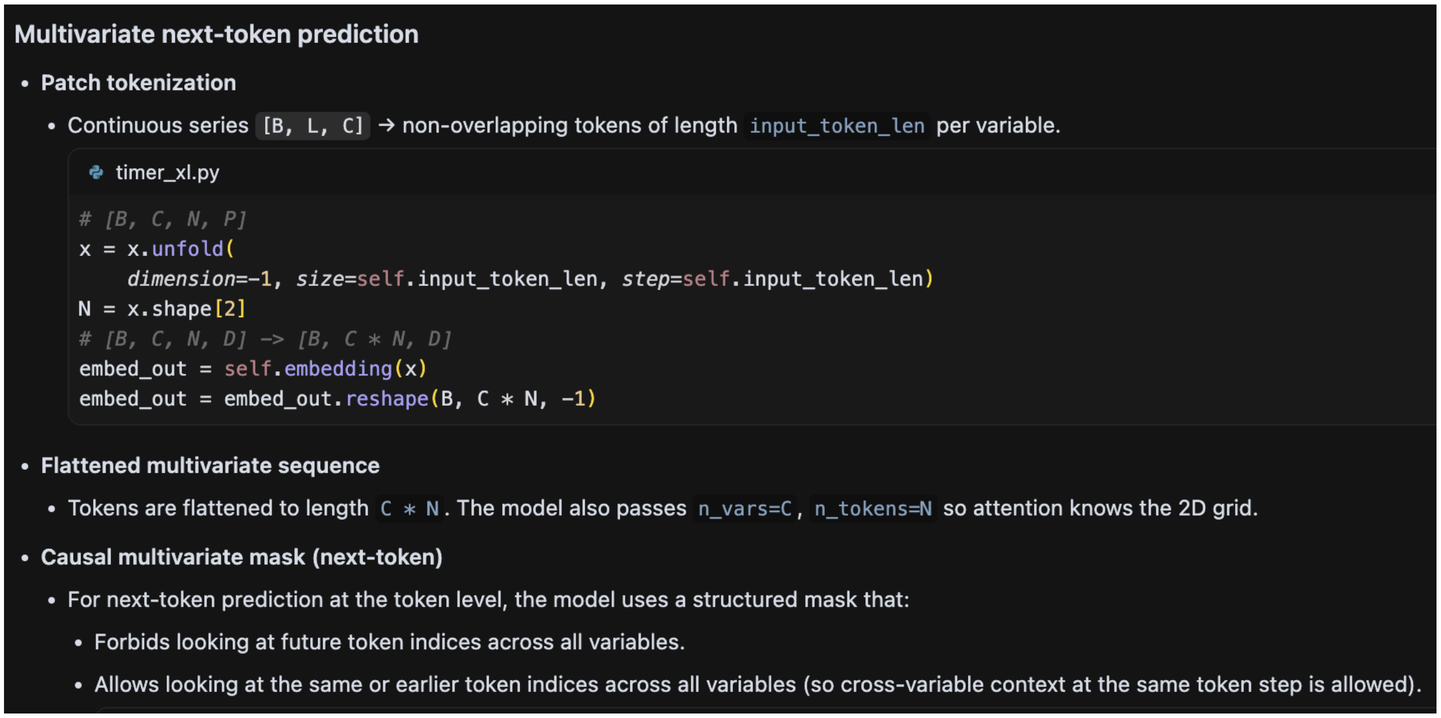
**Quick example**

- Suppose `L=12`, `P_in=4`, `C=2`.
  - Each variable yields tokens: indices `[0..3], [4..7], [8..11]` → `N=3`.
  - Total tokens per sample: `C*N = 6`.
  - If `P_out=4`, output length per variable is `3*4=12`.

**Key implications**

- Tokens are contiguous, non-overlapping time windows per variable.
- Tail truncation: last incomplete window is dropped if `L % P_in != 0`.
- To produce a horizon `H` from the whole context, set `P_in=L` (so `N=1`) and `P_out=H`.

☑ Document multivariate next token prediction implementation



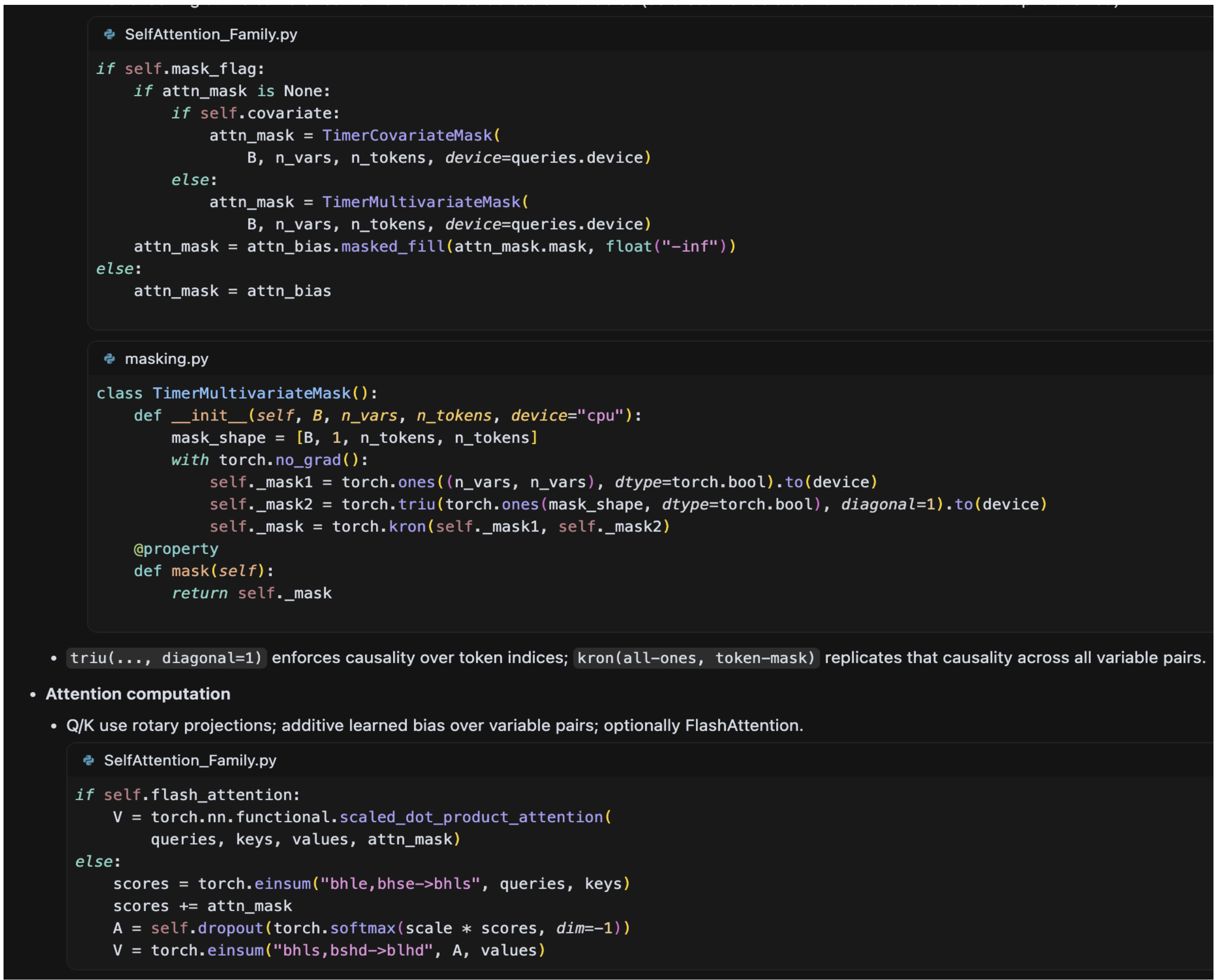
## Multivariate next-token prediction

- **Patch tokenization**
  - Continuous series `[B, L, C]` → non-overlapping tokens of length `input_token_len` per variable.

timer_xl.py

```python
# [B, C, N, P]
x = x.unfold(
    dimension=-1, size=self.input_token_len, step=self.input_token_len)
N = x.shape[2]
# [B, C, N, D] -> [B, C * N, D]
embed_out = self.embedding(x)
embed_out = embed_out.reshape(B, C * N, -1)
```

- **Flattened multivariate sequence**
  - Tokens are flattened to length `C * N`. The model also passes `n_vars=C`, `n_tokens=N` so attention knows the 2D grid.
- **Causal multivariate mask (next-token)**
  - For next-token prediction at the token level, the model uses a structured mask that:
    - Forbids looking at future token indices across all variables.
    - Allows looking at the same or earlier token indices across all variables (so cross-variable context at the same token step is allowed).



SelfAttention_Family.py

```python
if self.mask_flag:
    if attn_mask is None:
        if self.covariate:
            attn_mask = TimerCovariateMask(
                B, n_vars, n_tokens, device=queries.device)
        else:
            attn_mask = TimerMultivariateMask(
                B, n_vars, n_tokens, device=queries.device)
    attn_mask = attn_bias.masked_fill(attn_mask.mask, float("-inf"))
else:
    attn_mask = attn_bias
```

masking.py

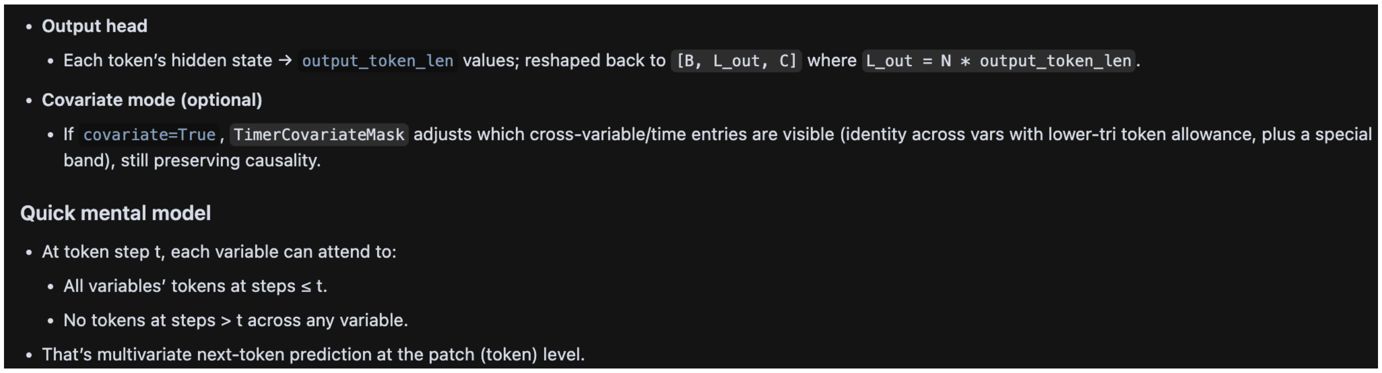
```python
class TimerMultivariateMask():
    def __init__(self, B, n_vars, n_tokens, device="cpu"):
        mask_shape = [B, 1, n_tokens, n_tokens]
        with torch.no_grad():
            self._mask1 = torch.ones((n_vars, n_vars), dtype=torch.bool).to(device)
            self._mask2 = torch.triu(torch.ones(mask_shape, dtype=torch.bool), diagonal=1).to(device)
            self._mask = torch.kron(self._mask1, self._mask2)
    @property
    def mask(self):
        return self._mask
```

- `triu(..., diagonal=1)` enforces causality over token indices; `kron(all-ones, token-mask)` replicates that causality across all variable pairs.

- **Attention computation**
  - Q/K use rotary projections; additive learned bias over variable pairs; optionally FlashAttention.

SelfAttention_Family.py

```python
if self.flash_attention:
    V = torch.nn.functional.scaled_dot_product_attention(
        queries, keys, values, attn_mask)
else:
    scores = torch.einsum("bhle,bhse->bhls", queries, keys)
    scores += attn_mask
    A = self.dropout(torch.softmax(scale * scores, dim=-1))
    V = torch.einsum("bhls,bshd->blhd", A, values)
```

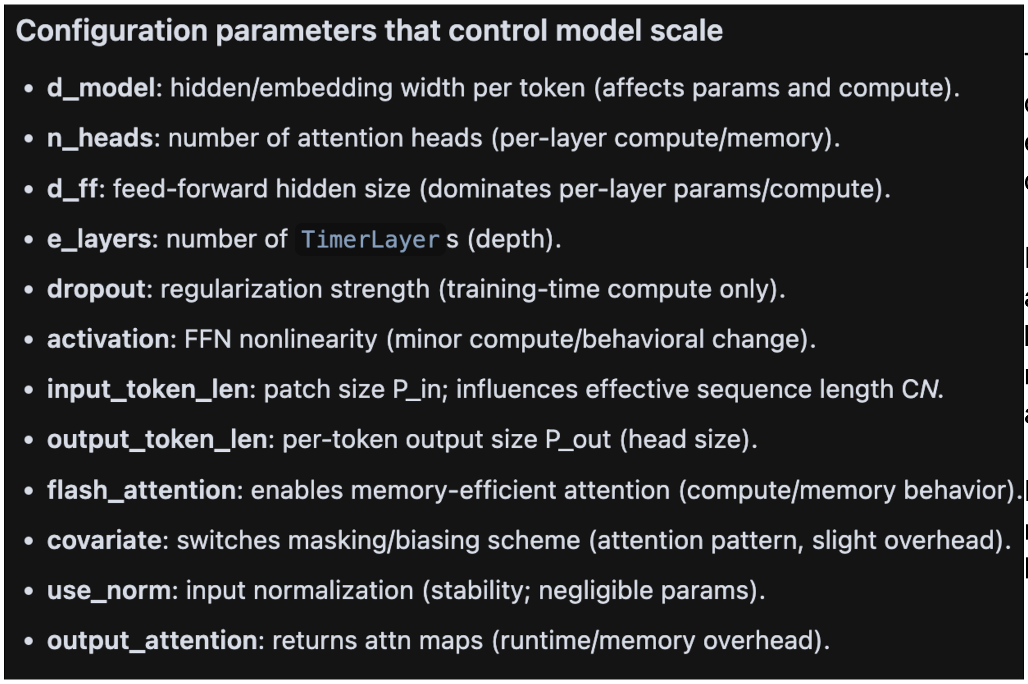


- **Output head**
  - Each token's hidden state → `output_token_len` values; reshaped back to `[B, L_out, C]` where `L_out = N * output_token_len`.
- **Covariate mode (optional)**
  - If `covariate=True`, `TimerCovariateMask` adjusts which cross-variable/time entries are visible (identity across vars with lower-tri token allowance, plus a special band), still preserving causality.

**Quick mental model**

- At token step t, each variable can attend to:
  - All variables' tokens at steps ≤ t.
  - No tokens at steps > t across any variable.
- That's multivariate next-token prediction at the patch (token) level.

- [x] Extract configuration parameters for model scaling



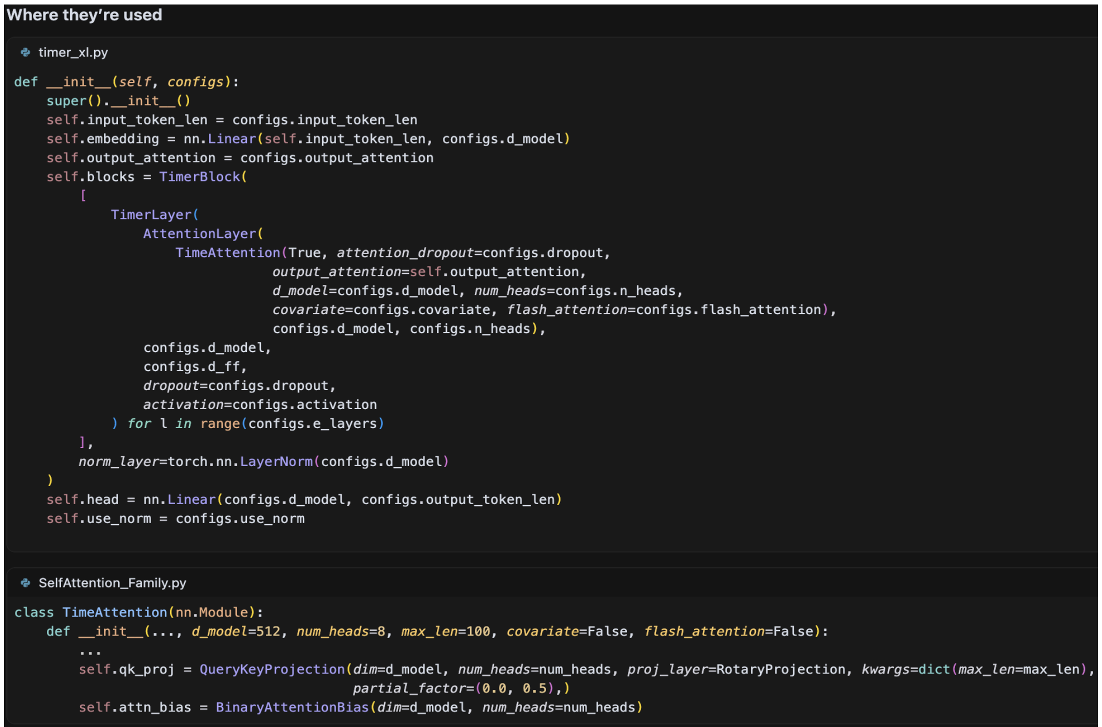
## Configuration parameters that control model scale

- **d_model**: hidden/embedding width per token (affects params and compute).
- **n_heads**: number of attention heads (per-layer compute/memory).
- **d_ff**: feed-forward hidden size (dominates per-layer params/compute).
- **e_layers**: number of `TimerLayer`s (depth).
- **dropout**: regularization strength (training-time compute only).
- **activation**: FFN nonlinearity (minor compute/behavioral change).
- **input_token_len**: patch size P_in; influences effective sequence length *CN*.
- **output_token_len**: per-token output size P_out (head size).
- **flash_attention**: enables memory-efficient attention (compute/memory behavior).
- **covariate**: switches masking/biasing scheme (attention pattern, slight overhead).
- **use_norm**: input normalization (stability; negligible params).
- **output_attention**: returns attn maps (runtime/memory overhead).

Things we can decrease:
e_layers,
d_model, d_ff

Input token length and output token length should remain the same as Target model

Flash Attention may have some hardware issues.



**Where they're used**

timer_xl.py

```python
def __init__(self, configs):
    super().__init__()
    self.input_token_len = configs.input_token_len
    self.embedding = nn.Linear(self.input_token_len, configs.d_model)
    self.output_attention = configs.output_attention
    self.blocks = TimerBlock(
        [
            TimerLayer(
                AttentionLayer(
                    TimeAttention(True, attention_dropout=configs.dropout,
                                  output_attention=self.output_attention,
                                  d_model=configs.d_model, num_heads=configs.n_heads,
                                  covariate=configs.covariate, flash_attention=configs.flash_attention),
                                  configs.d_model, configs.n_heads),
                configs.d_model,
                configs.d_ff,
                dropout=configs.dropout,
                activation=configs.activation
            ) for l in range(configs.e_layers)
        ],
        norm_layer=torch.nn.LayerNorm(configs.d_model)
    )
    self.head = nn.Linear(configs.d_model, configs.output_token_len)
    self.use_norm = configs.use_norm
```

SelfAttention_Family.py

```python
class TimeAttention(nn.Module):
    def __init__(..., d_model=512, num_heads=8, max_len=100, covariate=False, flash_attention=False):
        ...
        self.qk_proj = QueryKeyProjection(dim=d_model, num_heads=num_heads, proj_layer=RotaryProjection, kwargs=dict(max_len=max_len),
                                          partial_factor=(0.0, 0.5),)
        self.attn_bias = BinaryAttentionBias(dim=d_model, num_heads=num_heads)
```

### 1.3 Create inventory of all datasets supported in the codebase

- ☐ List all datasets in OpenLTM data loaders
- ☐ Document dataset characteristics (univariate/multivariate, length, frequency)
- ☐ Identify dataset preprocessing pipelines
- ☐ Create dataset configuration file with paths and parameters
- ☐ Set up data download scripts for all datasets

For implementing the prototype: may want to start with a small multivariate dataset.

## Phase 2: Draft Model Design and Implementation

### 2.1 Design the draft model architecture

- ☐ Define downsampling strategies:
    - Embedding dimension reduction (e.g., embed_dim // 2)
    - Layer reduction (e.g., n_layers // 2)
    - Attention head reduction (e.g., n_heads // 2)
- ☐ Create configuration class for draft model with scaling parameters
- ☐ Ensure compatibility between draft and target model outputs
- ☐ Design adapter layers if needed for dimension matching

Do we need to align the embedding dimension of draft and target models?
We can start by dividing by 2 (for downsizing) for some acceleration (hopefully the quality of predictions don't decrease). We can start by downsizing more conservatively.

### 2.2 Implement the draft model

- ☐ Create `timer_xl_draft.py` based on Timer-XL architecture
- ☐ Implement configurable downsampling in model initialization
- ☐ Ensure forward pass compatibility with Timer-XL interface
- ☐ Implement patch embedding layer with reduced dimensions
- ☐ Implement scaled TimeAttention mechanism
- ☐ Add model checkpointing and saving functionality

The draft model implementation is essentially just the target model with different hyperparameters (scaled down)

## Phase 3: Pre-training Infrastructure

### 3.1 Implement pre-training pipeline for the draft model

- ☐ Adapt Timer-XL pre-training script for draft model
- ☐ Implement data loaders for all identified datasets
- ☐ Create multi-dataset training loop with dataset sampling
- ☐ Implement loss calculation for multivariate next token prediction
- ☐ Add logging and monitoring (wandb/tensorboard integration)
- ☐ Create checkpoint saving strategy

### 3.2 Pre-train draft models

- ☐ Train draft model with embed_dim // 2
- ☐ Train draft model with n_layers // 2
- ☐ Train draft model with combined downsampling
- ☐ Monitor training metrics and convergence
- ☐ Save best checkpoints for each configuration

3 types of training we need to do:

- Fine-tuning of target model
- Pre-training of draft model
- Fine-tuning of draft model (on downstream tasks)

We need to make sure that the draft and target model share the same underlying context/knowledge to make sure that the performance does not deteriorate that much. (finetune on some portion of the dataset and then do inference on the test portion)
Training data is the “upper bound”. Giving the draft model access to a larger training corpus (this corpus is what we used for pre-training the target model) could make the model perform better.

## Phase 4: Speculative Decoding Framework

### 4.1 Implement speculative decoding framework for time series patches

- ☐ Create `speculative_decoder.py` module
- ☐ Implement draft model speculation for K future patches
- ☐ Handle multivariate time series token generation
- ☐ Implement batched speculation for efficiency
- ☐ Create token probability distribution extraction

### 4.2 Implement parallel verification mechanism

- ☐ Implement parallel forward pass for Timer-XL verification

- [ ] Create batch construction for parallel verification:
    - Input: [context], [context, spec_1], [context, spec_1, spec_2], ...
- [ ] Implement acceptance/rejection logic based on probability distributions
- [ ] Calculate acceptance rate metrics
- [ ] Implement fallback to standard generation when speculation fails

### 4.3 Integrate speculative decoding with generation

- [ ] Modify Timer-XL's generate() method to use speculative decoding
- [ ] Implement adaptive K (number of speculated tokens) based on acceptance rate
- [ ] Add temperature and top-p sampling compatibility
- [ ] Ensure proper handling of sequence lengths and padding

## Phase 5: Fine-tuning and Downstream Tasks

### 5.1 Create fine-tuning pipeline for downstream tasks

- [ ] Identify downstream tasks (forecasting horizons, datasets)
- [ ] Implement task-specific data loaders
- [ ] Create fine-tuning script with frozen/unfrozen backbone options
- [ ] Implement task-specific evaluation metrics
- [ ] Add early stopping and best model selection

### 5.2 Fine-tune draft models for each dataset

- [ ] Fine-tune on ETT datasets (ETTh1, ETTh2, ETTm1, ETTm2)
- [ ] Fine-tune on electricity dataset
- [ ] Fine-tune on traffic dataset
- [ ] Fine-tune on weather dataset
- [ ] Fine-tune on ILI dataset
- [ ] Save task-specific checkpoints

## Phase 6: Ablation Studies

### 6.1 Implement ablation study framework

- [ ] Create experiment configuration system
- [ ] Implement weight freezing/unfreezing options during speculative decoding
- [ ] Add draft model architecture ablations (width vs depth scaling)
- [ ] Implement different K values for speculation
- [ ] Create automated experiment runner

### 6.2 Run ablation experiments

- ☐ Test fixed vs. adaptive weights during speculative decoding
- ☐ Compare different draft model architectures
- ☐ Analyze impact of speculation length K
- ☐ Test on different dataset characteristics
- ☐ Document results in structured format

## Phase 7: Evaluation and Optimization

### 7.1 Develop evaluation framework

- ☐ Implement latency measurement tools
- ☐ Calculate effective tokens per second
- ☐ Measure memory usage and GPU utilization
- ☐ Implement accuracy metrics (MAE, MSE, MAPE)
- ☐ Create visualization tools for results

### 7.2 Comprehensive evaluation

- ☐ Evaluate speedup across all datasets
- ☐ Compare accuracy with baseline Timer-XL
- ☐ Analyze acceptance rates by dataset type
- ☐ Profile bottlenecks in the pipeline
- ☐ Generate performance reports

### 7.3 Optimize the complete system

- ☐ Implement kernel fusion for parallel verification
- ☐ Optimize memory allocation and caching
- ☐ Implement dynamic batching for variable-length sequences
- ☐ Add mixed precision training/inference
- ☐ Create production-ready inference server

We want to report efficiency as a metric too.

## Phase 8: Documentation and Release

### 8.1 Create comprehensive documentation

- ☐ Write API documentation for speculative decoder
- ☐ Create usage examples for each dataset

- ☐ Document configuration options and best practices
- ☐ Write performance tuning guide
- ☐ Create troubleshooting guide

### 8.2 Prepare for release

- ☐ Clean up codebase and remove experimental code
- ☐ Create unit tests for all components
- ☐ Package as pip-installable library
- ☐ Create demo notebooks for key use cases
- ☐ Prepare model zoo with pre-trained draft models

## Key Technical Considerations

1. **Token Definition**: In time series, a token is a patch of continuous data points. Ensure consistent patch size between draft and target models.

2. **Parallel Verification**: Leverage GPU parallelism by constructing batches [context], [context + spec_1], [context + spec_1 + spec_2], etc., for simultaneous verification.

3. **Acceptance Criteria**: Use KL divergence or token-wise probability comparison for accepting/rejecting speculated patches.

4. **Memory Efficiency**: Implement gradient checkpointing and efficient attention mechanisms for long-context time series.

5. **Dataset Coverage**: Ensure the implementation works across univariate, multivariate, and covariate-informed time series.