FMTK: A Modular Toolkit for Composable Time Series Foundation Model Pipelines

Anonymous Author(s)
Affiliation
Address
email

Abstract

Foundation models (FMs) have opened new avenues for machine learning applications due to their ability to adapt to new and unseen tasks with minimal or no further training. Time-series foundation models (TSFMs)—FMs trained on time-series data—have shown strong performance on classification, regression, and imputation tasks. Recent pipelines combine TSFMs with task-specific encoders, decoders, and adapters to improve performance; however, assembling such pipelines typically requires ad hoc, model-specific implementations that hinder modularity and reproducibility. We introduce FMTK, an open-source, lightweight and extensible toolkit for constructing and fine-tuning TSFM pipelines via standardized backbone and component abstractions. FMTK enables flexible composition across models and tasks, achieving correctness and performance with an average of seven lines of code. https://anonymous.4open.science/r/FMTK-4F96

1 Introduction

Time Series Foundation Models (TSFMs), such as MOMENT [6], Chronos [2], and TimesFM [4], have emerged as powerful pre-trained architectures for a variety of downstream tasks, including forecasting, classification, and imputation. While these models serve as fixed backbones trained on large-scale temporal data, effective task specialization often requires integrating additional components: input encoders to structure raw data, task-specific decoders to generate predictions, and increasingly, parameter-efficient adapters (e.g., LoRA [8]) to enable lightweight fine-tuning.

This modular design space, though conceptually flexible, has resulted in a fragmented and ad hoc implementation landscape. For example, models such as PaPaGei [13] and Mantis [5] require the development of extensive custom pipelines for simple comparison with state-of-the-art models. Moreover, models like Moment [6] offer distinct modes for different tasks (e.g., forecasting versus classification), which restricts the reuse of the same powerful backbone for a diverse set of downstream tasks during runtime. As a result, three key challenges emerge. ① First, the absence of a unifying abstraction across encoders, backbones, adapters, and decoders significantly increases the engineering burden and inhibits systematic exploration of architectural variants. ② Second, the lack of modular encapsulation complicates the attribution. It becomes difficult to isolate and measure the contribution of individual components to the overall performance of the model. ③ Third, evaluation practices vary widely across studies. Minor differences in data pre-processing, training regimes, or decoder heads can lead to substantial discrepancies in reported results, thereby undermining reproducibility.

Although existing time series libraries such as sktime [10], Darts [7], tsai [12] and GluonTS [1] support classical and deep learning pipelines, they do not address the emerging need for composable, FM-centric evaluation. To this end, we introduce FMTK: an open-source, lightweight, and extensible Time Series Foundation Model Toolkit for constructing, fine-tuning and benchmarking modular TSFM pipelines.

Under review at the NeurIPS 2025 Workshop on Recent Advances in Time Series Foundation Models (BERT^{2}S). Do not distribute.

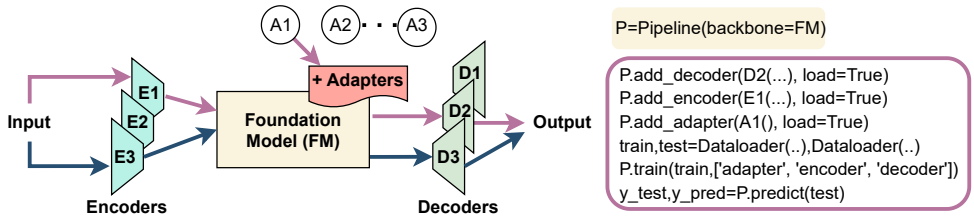

Figure 1: **Modular abstraction of pipeline construction using** FMTK: The framework allows instantiating pipelines by pairing an FM with interchangeable components. Users can dynamically select and load components, specify trainable parts (e.g., decoder), and benchmark pipelines in a unified interface. We illustrate two example configurations using the same FM: **(top)** encoder-decoder-adapter-tuned pipeline with E1, A1 and D2; **(bottom)** encoder-decoder tuned with E3 and D3.

Contributions: Our contributions can be summarized as follows:

1. FMTK proposes a standardized API for TSFM pipelines that defines a common grammar for connecting FM backbones with external encoders, fine-tuning adapters and decoders.
2. FMTK provides reference implementation of commonly used configurations and supports multiple time series tasks under consistent evaluation settings.
3. By decoupling architectural components and enforcing standardized execution semantics, FMTK facilitates reproducible experimentation and controlled comparison across a rapidly growing space of TSFM-based systems.

2 Design

To address the challenges outlined earlier (①–③), the design of FMTK is guided by three core principles: composability, usability, and reproducibility. *Composability* is achieved through standardized interfaces for encoders, backbones, adapters, and decoders, enabling seamless interchangeability and systematic exploration of pipeline configurations. *Usability* is prioritized through a lightweight and declarative API that abstracts away engineering complexity, making it possible for both domain experts and non-specialists to experiment with modular TSFM pipelines without extensive prior familiarity with FM internals. Finally, to promote *reproducibility*, the toolkit enforces consistent pre-processing, component integration, evaluation routines, and monitoring capabilities, allowing researchers to conduct controlled comparisons and ablation studies under uniform conditions.

2.1 Components

These design principles manifest in FMTK through a modular architecture centered around four key components as shown in Figure 1. **Encoders** transform raw time series into representations compatible with the foundation model. These may perform format conversion, context windowing, dimentionality reduction or domain-specific preprocessing. FMTK allows interchangeable encoders to support diverse inputs. **Foundation Model Backbone (FM)** refers to a pre-trained, frozen or partially frozen TSFM that produces task-agnostic latent features. FMTK supports FMs like Chronos, Moment, and TimesFM, by providing a common I/O interfaces. **Adapters** are lightweight modules for efficient adaptation to new tasks or domains without full backbone fine-tuning. FMTK currently supports methods such as LoRA enabling selective training. **Decoders** are task-specific heads (e.g., for forecasting, classification) that operate on FM outputs. They may range from simple MLPs to more complex attention-based or partially fine-tuned heads.

2.2 Pipeline Abstraction

To support flexible experimentation, FMTK exposes a unified `Pipeline` abstraction, which allows any valid combination of encoder, backbone, adapter, and decoder modules. As illustrated in Figure 1, different configurations traverse distinct paths through the component graph depending on the intended usage pattern. We demonstrate via two use-cases how the system handles component integration, fine-tuning and execution behind the scenes, allowing both novice users and expert

		Chronos		Moment		PaPaGei-S	
Task	**Decoder**	Base	FMTK	Base	FMTK	Base	FMTK
Regression (MAE)							
Systolic BP (PPG-BP)		15.84	15.82	15.99	15.99	15.65	15.65
Diastolic BP (PPG-BP)	Ridge	9.43	9.56	8.88	8.88	8.98	8.98
Heart Rate (PPG-BP)		9.04	9.04	5.24	5.24	6.32	6.32
Classification (Acc %)							
Heartbeat (ECG5000)	SVM	–	93.15	93.42	94.02	–	89.88
Forecasting (MAE)							
Energy (ETTh1)	MLP	–	0.76	0.43	0.54	–	0.83

Table 1: MAE and Accuracy comparison between baseline and FMTK across regression, classification, and forecasting tasks using Chronos-large, Moment-large, and PaPaGei-S backbones with Ridge, SVM and MLP decoder.

researchers to build, train, and evaluate pipelines with minimal overhead. (More details provided in Appendix A)

Use-Case 1 (simple usage): Chronos backbone can be combined with SVM decoder to enable Heartbeat Classification task. FMTK allows this by attaching the decoder through `add_decoder()` and restricting training to the decoder via `train(parts_to_train=[...])`.

Use-Case 2 (advanced usage): The Moment backbone can support diverse downstream tasks, such as PPG monitoring and Energy Forecasting, through a shared representation. PPG monitoring is supported by attaching a regression MLP decoder, linear encoder, and LoRA adapter to the backbone, while Energy Forecasting is supported by attaching an MLP decoder alone. After fine-tuning as explained in Use-Case 1, toolkit enables adaptive switching between distinct paths at runtime via `load_encoder()`, `load_adapter()` and `load_decoder()`.

2.3 Runtime Metrics and Benchmarking Support

In addition to traditional accuracy-based or MAE-based evaluation, FMTK supports the collection of runtime metrics for pipeline benchmarking. For each experiment, the framework logs:

1. **Memory Utilization**: Peak GPU/CPU memory usage during inference and training.
2. **Training and Inference Time**: Wall-clock time for each training phase and prediction batch.
3. **Component Loading Overhead**: Time to load and switch between components at runtime.
4. **Energy Consumption**: Optional support for energy profiling using compatible hardware monitors.

These measurements facilitate holistic comparisons between pipeline variants and promote reproducibility by modularizing system-level bottlenecks and trade-offs.

3 Evaluation

We implement FMTK in approximately 1.6k lines of Python code, leveraging PyTorch 2.7.1 for model execution and integration with widely used transformer and fine-tuning libraries [11]. All experiments are conducted on a Linux system equipped with an NVIDIA A100 GPU, using Python 3.10.18 and CUDA 12.6. FM backbones are sourced from publicly available repositories: Chronos and Moment are retrieved via HuggingFace, while Papagei is initialized from locally downloaded checkpoints. We have performed three types of tasks regression, classification and forecating using PPG-BP [9], ECG5000 [3], and ETTh1 [14] datasets.

We assess the capabilities of FMTK along three dimensions aligned with its design goals: (i) usability and interface simplicity; (ii) architectural adaptability across tasks and components; and (iii) performance benchmarking. Together, these evaluations aim to characterize the extent to which FMTK facilitates reproducible, modular experimentation in the TSFM landscape.

3.1 Usability and Interface Simplicity

FMTK prioritizes usability through a set of intuitive APIs. Users can compose, fine-tune and execute complex pipelines via a small number of method calls, abstracting away internal implementation details. To validate FMTK, we first replicate the PPG-based physiological monitoring benchmark

Decoder	Chronos	Moment	PaPaGei-S
SVM	93.15	94.02	89.86
KNN	91.64	92.71	86.91
LR	58.37	92.02	89.51
RF	91.26	91.80	87.53
MLP	90.26	93.00	75.55

	HC Task		EF Task	
Metric	Base	FMTK	Base	FMTK
Time (s)				
Finetune	10.83	11.15	50.38	50.20
Predict	0.03	0.03	0.04	0.04
Peak Memory (MB)				
Finetune	563.11	569.17	804.86	779.08
Predict	562.62	562.62	797.07	740.80
Energy (J)				
Finetune	892.52	923.19	16718.41	16689.45
Predict	7.84	8.14	16.56	16.86

Table 2: **(left)** Accuracy of Chronos-large, Moment-large, and PaPaGei-S backbone with multiple decoders for Heartbeat Classification task. Table 3: **(right)** Performance comparison of FMTK using the Moment-base backbone with SVM and MLP decoders for Heartbeat Classification (HC) and Energy Forecasting (EF), respectively. Fine-tuning time is measured over the entire train dataset, while prediction time is measured per batch.

from the PaPaGei repository [13], which includes tasks such as predicting systolic/diastolic BP and heart rate. As shown in Table 1, our standardized pipeline matches the baseline results, achieving an error rate within 1% of the original implementation across all prediction tasks and FM backbones. In addition, FMTK simplifies the exploration of novel architectural combinations. For instance, constructing a complex pipeline that combines the Moment backbone with a custom encoder, an MLP decoder, and a LoRA adapter for fine-tuning requires only seven lines of code (see Appendix A for details). This demonstrates the toolkit's core design principle: new components can be implemented by sub-classing simple abstract classes and are immediately interoperable within the `Pipeline` interface, enabling rapid and systematic experimentation.

3.2 Architectural Adaptability

One of the central design goals of FMTK is to enable modular experimentation across different architectural choices. To demonstrate this, we first show how a single backbone can be paired with various decoders. As shown in Table 2, the Chronos, Moment and PaPaGei backbone can be seamlessly combined with diverse decoders such as an SVM, MLP, KNN, Logistic Regression (LR) or Random Forest (RF) for Heartbeat Classification task. Furthermore, FMTK simplifies the process of repurposing the entire pipeline for different tasks and backbones with minimal code changes. For instance, Table 1 presents results for both Heartbeat Classification and Energy Forecasting tasks using the Chronos, Moment and PaPaGei backbones, all implemented within our unified toolkit using non-traditional components. FMTK successfully accommodates a rich diversity of components and tasks, fulfilling its core design goal of composability. Notably, the table demonstrates novel compositions, even beyond default use-cases, as in the case of Chronos and PaPaGei.

3.3 Performance Benchmarking

To validate that the modularity and ease of use in FMTK come at an acceptable computational cost, we conduct an extensive performance analysis comparing our toolkit with the baseline implementations. We used Moment-Base model with SVMDecoder for Heartbeat Classification, and with MLPDecoder for Energy Forecasting task. As shown in Table 3, FMTK incurs a minimal $\sim$3% overhead for fine-tuning and prediction time across both tasks. It achieves a $\sim$7% reduction in peak inference GPU memory for the Energy Forecasting task, with negligible memory overhead during training. Furthermore, the toolkit exhibits an energy consumption comparable to the baseline implementation.

4 Conclusion

This work introduces FMTK, a lightweight yet extensible toolkit for modularizing, composing, and benchmarking time series foundation model pipelines. By decoupling architectural components and unifying evaluation semantics, FMTK lowers the barrier to rigorous, reproducible experimentation across a rapidly expanding design space. While instantiated for time series tasks, the framework is broadly applicable to any foundation model workflow exhibiting an encoder–backbone–decoder structure, offering a template for systematic evaluation across modalities. For future work, we are expanding this toolkit to support other types of foundation models, adapters and add support for runtime optimizations.

References

[1] Alexander Alexandrov, Konstantinos Benidis, Michael Bohlke-Schneider, Valentin Flunkert, Jan Gasthaus, Tim Januschowski, Danielle C. Maddix, Syama Rangapuram, David Salinas, Jasper Schulz, Lorenzo Stella, Ali Caner Türkmen, and Yuyang Wang. GluonTS: Probabilistic and Neural Time Series Modeling in Python. *Journal of Machine Learning Research*, 21(116):1–6, 2020.

[2] Abdul Fatir Ansari, Lorenzo Stella, Caner Turkmen, Xiyuan Zhang, Pedro Mercado, Huibin Shen, Oleksandr Shchur, Syama Syndar Rangapuram, Sebastian Pineda Arango, Shubham Kapoor, Jasper Zschiegner, Danielle C. Maddix, Michael W. Mahoney, Kari Torkkola, Andrew Gordon Wilson, Michael Bohlke-Schneider, and Yuyang Wang. Chronos: Learning the language of time series. *Transactions on Machine Learning Research*, 2024.

[3] Y. Chen and E. Keogh. Ecg5000 dataset.

[4] Abhimanyu Das, Weihao Kong, Rajat Sen, and Yichen Zhou. A decoder-only foundation model for time-series forecasting. In *Forty-first International Conference on Machine Learning*, 2024.

[5] Vasilii Feofanov, Songkang Wen, Marius Alonso, Romain Ilbert, Hongbo Guo, Malik Tiomoko, Lujia Pan, Jianfeng Zhang, and Ievgen Redko. Mantis: Lightweight calibrated foundation model for user-friendly time series classification. *arXiv preprint arXiv:2502.15637*, 2025.

[6] Mononito Goswami, Konrad Szafer, Arjun Choudhry, Yifu Cai, Shuo Li, and Artur Dubrawski. Moment: A family of open time-series foundation models. In *International Conference on Machine Learning*, 2024.

[7] Julien Herzen, Francesco LÃ¤ssig, Samuele Giuliano Piazzetta, Thomas Neuer, LÃ©o Tafti, Guillaume Raille, Tomas Van Pottelbergh, Marek Pasieka, Andrzej Skrodzki, Nicolas Huguenin, Maxime Dumonal, Jan KoÅ›cisz, Dennis Bader, FrÃ©dÃ©rick Gusset, Mounir Benheddi, Camila Williamson, Michal Kosinski, Matej Petrik, and GaÃ«l Grosch. Darts: User-Friendly Modern Machine Learning for Time Series. *Journal of Machine Learning Research*, 23(124):1–6, 2022.

[8] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. LoRA: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[9] Yongbo Liang, Guiyong Liu, Zhencheng Chen, and Mohamed Elgendi. PPG-BP Database. 2 2018.

[10] Markus Löning, Anthony Bagnall, Sajaysurya Ganesh, Viktor Kazakov, Jason Lines, and Franz J Király. sktime: A Unified Interface for Machine Learning with Time Series. In *Workshop on Systems for ML at NeurIPS 2019*.

[11] Sourab Mangrulkar, Sylvain Gugger, Lysandre Debut, Younes Belkada, Sayak Paul, and Benjamin Bossan. PEFT: State-of-the-art parameter-efficient fine-tuning methods. `https://github.com/huggingface/peft`, 2022.

[12] Ignacio Oguiza. tsai - a state-of-the-art deep learning library for time series and sequential data. Github, 2023.

[13] Arvind Pillai, Dimitris Spathis, Fahim Kawsar, and Mohammad Malekzadeh. Papagei: Open foundation models for optical physiological signals. *arXiv preprint arXiv:2410.20542*, 2024.

[14] Haoyi Zhou, Shanghang Zhang, Jieqi Peng, Shuai Zhang, Jianxin Li, Hui Xiong, and Wancai Zhang. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *The Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Virtual Conference*, volume 35, pages 11106–11115. AAAI Press, 2021.

Appendices

A Implementation Details

A.1 Code Example for a Modular Pipeline

The following code Listing 1 demonstrates how FMTK can be used to construct a complex pipeline. It loads the Moment backbone, adds a custom encoder, attaches an MLP decoder, and applies a LoRA adapter for the heart rate prediction task using the PPG-BP dataset.

```
lora_config = LoraConfig(r=64, lora_alpha=32, target_modules=["q", "v"], lora_dropout=0.05)
P=Pipeline(MomentModel(model))
P.add_decoder(MLPDecoder(cfg={'input_dim':1024,'output_dim':1,'hidden_dim':128},load=True)
P.add_encoder(LinearChannelCombiner(cfg={num_channels=3,new_num_channels=1}, load=True)
P.add_adapter(lora_config)
P.train(dataloader_train,parts_to_train=['encoder','decoder','adapter'],cfg=task_cfg['train_config'])
y_test,y_pred=P.predict(dataloader_test,cfg=task_cfg['inference_config'])
```

Listing 1: Example to setup pipeline abstraction attaching Moment Backbone with MLP decoder, Linear channel encoder and LORA adapter for heart rate prediction task using PPG Dataset.

A.2 Standardized Interface for Integrating Customized Components

To integrate new components and enable modular composition in the time-series foundation-model (TSFM) pipeline, we adopt a single, minimal interface shared by encoders, backbones, and decoders (Listing 2 (left)). Each component minimally implements the BaseModel interface: preprocess adapts inbound tensors to the component's expected format (shape/device/dtype/reduction) and postprocess standardizes the outbound representation for the next stage. This separation isolates model-specific code from pipeline composition, allowing components to be swapped without code changes in the pipeline abstraction. For example as shown in Listing 2 (right), the Chronos backbone receives inputs shaped [B,C,L]. Internally, the model operates on a flattened view [B*C,L], which is produced in the preprocess. Chronos returns embeddings [B,E,L] (where E is the token/feature dimension); postprocess reshapes these to [B,C,E,L] so that downstream decoders (e.g., the MLP decoder in Listing 2 (right)) can consume a consistent, a channel aware representation. This standardized boundary ensures that alternative encoders/backbones/adapters/decoders can be composed interchangeably within the same pipeline.

```python
class BaseModel:
    def __init__(self, cfg):
        """
        Loading model
        """

    def preprocess(self, batch):
        """
        Match the shape and preprocess
        before sending it to model.
        Args:
            batch
        Returns:
            batch
        """

    def postprocess(self,embedding):
        """
        Postprocess the embedding to
        standard shape for next component
        Args:
            embedding
        Returns:
            embedding
        """

    def forward(self, batch):
        """
        Method for forward pass for
        one batch
        Args:
            batch
        Returns:
            embedding
        """
        ...

    def trainable_parameters(self):
        """
        Get trainable paramters
        out of model
        Returns:
            Iterable[torch.nn.Parameter]
        """
        ...
```

```python
# Encoder
class LinearChannelEncoder(BaseModel):
    def __init__(self,cfg):
        """
        Dimentionality reduction using
        linear layer from number of
        input channel to
        number of output channel.
        """

    def preprocess(self, batch):
        """
        Preprocessing it to acceptable
        input dimension [B,C,L]
        """

    def postprocess(self,embedding):
        """
        Postprocess the embedding to
        standard shape for
        foundation model [B,C,L]
        """


# Backbone wrapper
class ChronosBackbone(BaseModel):
    def __init__(self, cfg):
        """
        Loading Chronos
        """
        ...
    def preprocess(self, batch):
        """
        Preprocessing it to acceptable
        input dimension [B*C,L]
        """
        ...
    def forward(self, batch):
        """
        One forward pass through
        preprocess, backbone and
        postprocess
        """
        ...
    def postprocess(self, embedding):
        """
        Postprocess the embedding to
        standard shape for
        decoder [B,C,T,L] | [B,C,L]
        """
        ...


# Decoder
class MLPDecoder(BaseModel):
    def __init__(self, cfg):
        ...
    def preprocess(self, batch):
        """
        Preprocessing it to acceptable
        input dimension [B,L]
        based on cfg
        """
        ...
```

Listing 2: **(left)** Common interface description for encoder, backbone and decoders. **(right)** Example illustrating the pre-processing and post-processing steps involved in integrating a Linear channel encoder, Chronos backbone, and MLP decoder into a unified pipeline.