TIMECODER: LLM-DRIVEN SYNTHESIS AND REFINEMENT FOR INTERVENTION-AWARE TIME SERIES MODELING

Anonymous authors
Paper under double-blind review

ABSTRACT

Time series modeling in real-world systems often involves sparse and heterogeneous external interventions that abruptly alter underlying dynamics, posing significant challenges for both classical mechanistic models and purely data-driven approaches. While mechanistic models provide strong domain knowledge and interpretability, they lack flexibility under uncertainty, whereas deep learning models struggle to generalize intervention effects under limited supervision. We propose TimeCoder, an automated framework that integrates large language models (LLMs) reasoning, domain-informed mechanistic knowledge, and data-driven optimization to iteratively construct intervention-aware time series models. TimeCoder operates through a closed-loop generate–calibrate–refine process, where an LLM proposes explicit intervention-aware model structures grounded in domain knowledge and statistical evidence, model parameters are calibrated on observed data, and structured diagnostics of predictive errors guide subsequent structural refinement. We evaluate TimeCoder on four challenging benchmarks spanning healthcare, hydrology, and epidemics, demonstrating consistent improvements in forecasting and classification accuracy, generalization under sparse interventions, and sample efficiency compared to state-of-the-art baselines. Code is available at https://anonymous.4open.science/r/TimeCoder/.

1 INTRODUCTION

Time series analysis (Jiang et al., 2024; Nie et al., 2024; Xie et al., 2024; Jiang et al., 2025), including forecasting and classification, plays a central role in understanding complex dynamical systems across diverse domains, including healthcare, climate science, and finance (Ismail et al., 2020; Leung et al., 2023). Existing approaches have achieved remarkable success in capturing temporal dependencies from historical observations under relatively stable system dynamics (Ansari et al., 2024). However, real-world systems are frequently influenced by abrupt external interventions, such as medical treatments, that fundamentally alter their evolution patterns (Wang et al., 2024b; Nie et al., 2023a). Accurate prediction in such settings requires not only extrapolating historical trends, but also explicitly reasoning about how external interventions reshape future trajectories. We refer to this emerging problem as *intervention-aware time-series analysis*, which goes beyond classical problems by modeling both intrinsic temporal dynamics and the distinct, often sparse, effects of external interventions. This paradigm is critical for characterizing real-world system behavior and enabling proactive decision-making (Xu et al., 2024; Delibasoglu et al., 2026).

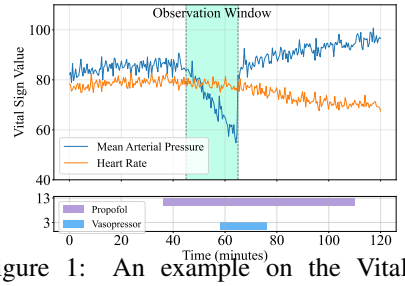


Figure 1: An example on the VitalDB dataset. A patients mean arterial pressure (MAP) exhibits a sharp decline after 42 minutes in the operating room, followed by a corrective rise after vasopressor administration at 65 min. Meanwhile, heart rate exhibits a slower trend.

Intervention-aware time-series modeling presents several fundamental challenges. First, external interventions often induce substantial shifts in the intrinsic dynamics, rendering historical temporal

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

patterns unreliable for future prediction. As shown in Figure 1, the administration of vasopressors leads to a sharp rise in a patients mean arterial pressure (MAP) after 65 minutes in the operating room, deviating significantly from pre-intervention trends. Second, intervention events are typically sparse and highly heterogeneous, limiting direct supervision and hindering generalization to minority or unseen interventions. Third, intervention responses are governed by domain-specific principles, such as physical laws or physiological constraints, which data-driven models struggle to enforce explicitly. Finally, discrepancies frequently arise between expectations and factual observations, as intervention effects may exhibit delayed or context-dependent responses. For instance, although a vasopressor is administered at 58 minutes in Figure 1, the patients MAP does not increase until approximately 63 minutes.

Existing methodologies address only part of these challenges. Classical mechanistic and rule-based models (Raissi et al., 2019; Takeishi & Kalousis, 2021) encode domain knowledge as explicit objectives or constraints, providing strong interpretability and reliability when such knowledge is well understood (Greydanus et al., 2019; Yin et al., 2021). However, they struggle to leverage observational data effectively and perform poorly when uncertainty or gaps exist between theory and real-world behaviors. Deep learning-based approaches enable flexible data-driven learning (Zhou et al., 2021; Wu et al., 2021; Xie et al., 2024), yet rely on fixed model structures and require large-scale data to capture rare and heterogeneous intervention effects. Recently, LLMs have demonstrated strong reasoning capabilities and broad domain knowledge (Zhou et al., 2024; Chang et al., 2023). Nevertheless, they remain limited in understanding numerical temporal data and modeling long-horizon dynamical patterns (Liu et al., 2024a; Schwartz et al., 2024).

We therefore ask: *Can we unify the rigor of time-series domain knowledge, data-driven optimization, and the flexible reasoning capabilities of LLMs to automatically construct intervention-aware models?*

Instead of relying on fixed model architectures or manually designed mechanistic equations, we propose a new paradigm for intervention-aware time-series modeling that treats model structure itself as a learnable objective guided by domain knowledge, empirical data observations, and reasoning. Concretely, we introduce **TimeCoder**, which iteratively constructs hybrid time-series models through a closed-loop generatecalibraterefine process. At each iteration, LLMs reason over domain knowledge and statistical evidence to propose explicit, executable model structures encoding hypotheses about intrinsic dynamics and intervention effects, which are then grounded in observed data via parameter calibration. Structured diagnostics from predictive errors and constraint violations are subsequently fed back to guide further structural refinement, enabling progressive convergence toward accurate models. To support this looped refinement in time-series domain, we select prototypical counterexamples to diagnose model weaknesses and design a set of descriptors to formulate temporal patterns or prediction residuals, enabling the LLM to revise models in an efficient and data-grounded manner. We evaluate TimeCoder on four intervention-aware time-series benchmarks across healthcare, hydrology, and epidemics, achieving the best performance on most metrics and up to 25% RMSE reduction over the strongest competing baseline across intervention-driven benchmarks. Our contributions are summarized as follows:

- We introduce a new paradigm for intervention-aware time-series modeling that treats model structure as an objective of automated reasoning and refinement, rather than a fixed architecture or manually designed constraints.
- We design a time-series-specific LLM-in-the-loop refinement protocol that makes hypothesis search efficient and data-grounded. It includes prototype-based counterexample selection, descriptor-based residual diagnosis, and memory-augmented structural refinement, etc.
- We demonstrate that automated structure refinement enables robust generalization under sparse and heterogeneous interventions. The produced model achieves consistent improvements across diverse real-world datasets and is used directly at inference without LLM calls.

2 Related Works

Time Series Analysis. Time-series forecasting and classification serve as essential foundations for modeling temporal dynamics and supporting downstream decision-making in complex systems (Qin et al., 2017; Wang et al., 2025a; Lim & Zohren, 2021; Wang et al., 2022). Early mechanistic methods used explicitly specified dynamical systems (Raissi et al., 2019; Cuomo et al., 2022; Takeishi &

Kalousis, 2021) or closed-form equations with data-estimated parameters (Greydanus et al., 2019; Cranmer et al., 2020; Yin et al., 2021; Qian et al., 2021). While interpretable and theoretically grounded, these methods are often limited to low-dimensional and well-understood systems. With the advance of deep learning, time series modeling shifted toward flexible neural architectures that learn temporal dependencies directly from large-scale data (Zhou et al., 2021; Wu et al., 2021; Xie et al., 2024). Current time-series methods typically begin with predefined modeling hypotheses, such as periodicity (Wang et al., 2025a), seasonality (Wang et al., 2022), and multi-scale temporal structure (Cai et al., 2024), and encode these assumptions into the model architecture (Zhang & Yan, 2023; Liu et al., 2024b; Wang et al., 2024a; Hu et al., 2025b; Nie et al., 2023b; Xing & He, 2023; Wu et al., 2023). Despite their empirical success, existing works largely rely on fixed architectures, limiting their ability to adapt model structure in response to intervention-driven or non-stationary dynamics.

LLMs for Time Series Analysis. Recent advances in LLMs have motivated their use in time-series modeling (Hu et al., 2025a; Liu et al., 2025b; Gruver et al., 2023a). The first line of work adapts LLMs as temporal representation learners, either by training LLMs from scratch on time series data (Ansari et al., 2024; Nie et al., 2023a; Zhang & Yan, 2022) or by fine-tuning pre-trained LLMs (Zhou et al., 2024; Chang et al., 2023; Hu et al., 2025a). These methods typically bridge modalities by tokenizing time-series signals or aligning them with the embedding space (Sun et al., 2024; Liu et al., 2025b). Prompt-based (Gruver et al., 2023a; Liu et al., 2023; Xue & Salim, 2023) approaches leverage the zero-shot or few-shot capabilities of LLMs by directly prompting pre-trained language models with text-based time series (Gruver et al., 2023a; Liu et al., 2023; Xue & Salim, 2023). Third, LLMs have been employed as reasoning modules (Jiang et al., 2025) or retrieval-augmented (RAG) summarization modules (Ning et al., 2025), generating summaries or contextual interpretations to complement conventional time series models (Chen et al., 2023b; Lee et al., 2025; Wang et al., 2024b). In contrast to these works, which mainly use LLMs for value-level prediction, representation learning, cross-modal alignment, or auxiliary textual reasoning, TimeCoder uses the LLM to propose executable temporal model structures, which are subsequently calibrated on observed data and refined using time-series-specific diagnostic feedback.

LLMs for Scientific Discovery. LLMs have recently been explored as general-purpose assistants for scientific discovery (Ma et al., 2024; AI4Science & Quantum, 2023), with applications in drug discovery (Song et al., 2025), biology (Kim et al., 2026), and materials science (Kumbhar et al., 2025). Prior work shows that LLMs can synthesize domain knowledge, generate scientifically plausible hypotheses, and reason over complex scientific concepts (Ma et al., 2024; AI4Science & Quantum, 2023; Wysocki et al., 2024). When equipped with external tools (Yuan et al., 2025; Shen, 2024), LLM agents can support iterative experimentation, equation or program discovery, and hybrid dynamical model construction (Shojaee et al., 2024; Holt et al., 2024; 2025), as well as automated reasoning (Ma et al., 2024; Li et al., 2024), paper drafting (Schmidgall et al., 2025), and review generation (Jin et al., 2024b). For example, works such as LLM-SR (Shojaee et al., 2024), HDTwin (Holt et al., 2024) and worldcoder (Tang et al., 2024) show the potential of LLMs for code generation and program synthesis. Despite these advances, existing LLM-based discovery frameworks lack considerations for intervention-aware time-series modeling, resulting in their ineffectiveness in encoding domain knowledge of temporal mechanisms, describing time-series patterns and identifying model weakness for looped refinement. TimeCoder makes the LLM-in-the-loop search efficient and temporally grounded, distinguishing itself from general-purpose LLM program discovery.

3 Preliminary

3.1 Notations and Problem Setup

We observe a multivariate time series $\mathbf{X} \in \mathbb{R}^{T \times D}$, where T is the sequence length and D is the number of observed variables. Each entry $\mathbf{X}[t, d]$ is an observation of the d-th dimension at time t. We assume there is a discrete intervention sequence $\mathbf{U} \in \mathbb{R}^{T \times M}$, where M is the dimension of intervention features. An intervention at time t can alter the subsequent evolution of $\mathbf{X}[t:, d]$. Given a training set $\mathcal{D} = \{(\mathbf{X}_i, \mathbf{U}_i, \mathbf{Y}_i)\}_{i=1}^{N}$, our objective is to learn a predictor $f_{\boldsymbol{\theta}}$ that maps a historical context window of observations and interventions to a target $\mathbf{Y}_i$. For the forecasting task, $\mathbf{Y}_i = \mathbf{X}_i[t+1 : t+H]$ represents future values over a horizon H. For the classification task, we have $Y = c \in \{1, \ldots, C\}$, in which C is the number of classes.

3.2 Task and Problem Formulation

Learning with a fixed model architecture from data alone is challenging when data is scarce and non-stationary due to interventions. We argue that a robust model should satisfy two criteria:

- **Empirical Alignment**: The model's predictions should fit the observed data, minimizing the task-specific loss (e.g., MSE for forecasting, cross-entropy for classification) $\mathcal{L}_{\text{data}}$.
- **Knowledge Alignment**: The model's functional form and behavior should be consistent with available domain knowledge $\mathcal{K}$. This knowledge $\mathcal{K}$ may include mechanistic laws (e.g., fluid dynamics), empirical rules (e.g., seasonal patterns), or qualitative constraints (e.g., "a beta-blocker monotonically lowers heart rate").

Formally, we formulate the learning procedure as an iterative process of hypothesis search and parameter calibration (Murphy, 2012):

$$(h^*, \boldsymbol{\theta}^*) = \arg \min_{h \in \mathcal{H},\, \boldsymbol{\theta} \in \Theta_h} \mathcal{L}_{\text{data}}(f_{h,\boldsymbol{\theta}}; \mathcal{D}) \quad \text{s.t.} \quad \text{dist}(f_{h,\boldsymbol{\theta}}, \mathcal{K}) \leq \epsilon_{\mathcal{K}}, \tag{1}$$

where $h \in \mathcal{H}$ denotes a hypothesis specifying how the system evolves over time, $\boldsymbol{\theta} \in \Theta_h$ denotes the model parameters associated with hypothesis h, which are calibrated from data, $f_{h,\boldsymbol{\theta}}$ is the time series model instantiated by h and $\boldsymbol{\theta}$, $\text{dist}(f_{h,\boldsymbol{\theta}}, \mathcal{K})$ measures the consistency between the empirical behavior of the learned model $f_{h,\boldsymbol{\theta}}$ and domain knowledge $\mathcal{K}$.

Since directly optimizing this objective over the discrete and combinatorial hypothesis space is generally intractable, we approximate it through an iterative search-and-calibration procedure. At each iteration i, the LLM proposes a new hypothesis or a structural edit to a previous hypothesis to obtain an updated h^{i+1}. The updated hypothesis is then grounded in data by re-calibrating its continuous parameters $\boldsymbol{\theta}^{i+1}$, yielding an executable time-series model $f_{h^{i+1},\boldsymbol{\theta}^{i+1}}$. This generate–calibrate–refine loop continues until empirical performance stabilizes, encouraging the final model to be both consistent with domain knowledge and accurate with respect to observed trajectories.

4 Methodology

Framework Overview To navigate the combinatorial hypothesis space and conduct co-evolvement of $\{h, \boldsymbol{\theta}\}$, we propose an iterative framework that decomposes the learning process into three steps:

- **LLM-Driven Model Specification**: Given the task descriptions, data summaries, and domain-knowledge context $\mathcal{K}$, an LLM drafts a candidate hypothesis h. This structure, expressed as a parameterized equation or rule set, explicitly represents hypotheses about intrinsic dynamics and intervention effects.
- **Data-Grounded Parameter Calibration**: For the drafted hypothesis h, a standard optimizer (we use the Adam optimizer (Zhang, 2018) in this work) calibrates its continuous parameters $\boldsymbol{\theta}$ by minimizing $\mathcal{L}_{\text{data}}$ on the observed series, yielding a concrete, executable model $f_{h,\boldsymbol{\theta}}$.
- **Performance-Guided Iterative Refinement**: The empirical performance (e.g., loss, residual patterns) of $f_{h,\boldsymbol{\theta}}$ is analyzed and described in natural language. This description is fed back to the LLM, which analyzes the current hypothesis and proposes a revised h. This loop closes a generate–calibrate–refine cycle, enabling data-driven evolution.

4.1 Hypothesis Space Construction

Defining the hypothesis space $\mathcal{H}$ for time series data is non-trivial, as it should balance expressiveness and simplicity (Murphy, 2012; Haussler & Warmuth, 2018). Expressiveness enables it to capture intrinsic temporal dynamics and complex intervention-response patterns, whereas simplicity helps avoid overfitting and improves generalization across heterogeneous interventions and forecasting tasks. To support hypothesis generation, we gather information from both domain knowledge and statistical summaries of time-series trajectories.

Representing Domain Knowledge. In many real-world scenarios, partial knowledge constraining how a time series may change across time under interventions is available. Due to the diverse patterns characterizing interactive temporal mechanisms across domains, we formalize available domain expertise at three complementary levels of granularity:

- K1: mechanistic laws from first principles. For example, flood forecasting models should respect hydrological dynamics (e.g., conservation laws in hydrology, pharmacokinetic differential equations), restricting physically plausible trajectories.
- K2: structured rules capturing empirical regularities, which establish qualitative patterns (e.g., "electricity demand peaks in the evening", "sales are seasonal").
- K3: intervention-consistent response constraints restricting the direction and magnitude of time series data responses under interventions (e.g., "a drug infusion causes a heart rate drop within 5-10 minutes", "a typhoon induces a decaying increase in the water level").

This knowledge is made accessible to the LLM via explicit prompting from domain corpora, and we leave the utilization of more complex knowledge for future work.

Statistical Evidence Description. To support model discovery and mechanism-consistent model refinement, we compute a suite of statistical descriptors from the observed series $\mathbf{X}$ and interventions $\mathbf{U}$, which serve as intermediate evidence bridging raw trajectories and LLM-driven model specification. Given a multivariate time series trajectory segment $\tau^{\text{real}}_{t-k:t} = \{(X_\tau, U_\tau)\}^{t}_{\tau=t-k}$, we construct a *Statistical Evidence Description* $\mathcal{Z}_t$, which aggregates interpretable statistical signatures of temporal dynamics and summarizes them into texts. Formally, $\mathcal{Z}_t$ is incrementally updated by merging newly extracted evidence:

$$\mathcal{Z}_t \leftarrow f_{\text{det}}(\tau^{\text{real}}_{t-k:t}) \ \cup \ \mathcal{Z}_{t-k}, \tag{2}$$

where f_{det} is implemented by invoking a set of statistical descriptors. Each descriptor produces a summary object (e.g., a confidence score, estimated parameters, and supporting diagnostics), and the union operator $\cup$ denotes evidence accumulation over time.

Specifically, we employ 5 complementary statistical descriptors: (1) *Temporal Lag Detection* which identifies systematic delays in the model's response to interventions or intrinsic dynamics using the cross-correlation tool. (2) *Trend Analysis* which performs linear regression to detect linear trends, returning the ratio, slope magnitude, and directional information. (3) *Periodicity Detection* which identifies cyclical patterns with Auto-correlation tool, returning percentage of sequences with residual periodicity and estimated period lengths. (4) *Abrupt Change Detection* which identifies sudden shifts in sequences by analyzing absolute differences between consecutive points. (5) *Systematic Bias Quantification* which detects consistent over- or under-prediction, indicating model parameter miscalibration or missing baseline terms. Bias magnitude, direction, and percentage of significantly biased sequences are returned.

The resulting $\mathcal{Z}_t$ can be viewed as a compact and interpretable representation of the data's statistical information, which is subsequently used to guide hypothesis proposal and model specification.

4.1.1 LLM-DRIVEN HYPOTHESIS PROPOSAL.

The hypothesis generation step uses a pre-trained LLM to propose an executable hypothesis. TimeCoder drafts a candidate model by jointly leveraging three sources of information: (i) **Problem Specification** $\mathcal{S}$, which describes the modeling task, input-output format, variable meanings, intervention indicators, and required code interface; (ii) **Domain Knowledge** $\mathcal{K}$, which provides mechanistic principles, empirical rules, and intervention-response constraints; and (iii) **Data-derived Statistical Evidence** $\mathcal{Z}_t$, which summarizes temporal patterns extracted from observed trajectories. Formally,

$$h \sim p_{\text{LLM}}(\mathcal{S}, \mathcal{K}, \mathcal{Z}_t)\,. \tag{3}$$

Concretely, h is instantiated as executable Python code that defines a candidate temporal model under this fixed input-output interface. To make the hypothesis space reproducible and reduce invalid or unconstrained generations, the prompt further provides numerical-stability rules, output constraints, and skeleton code with basic initialization and shape-consistency operations. This prompt-constrained executable template allows the LLM to search within a fixed interface while still proposing flexible temporal mechanisms, including intrinsic dynamics, intervention-response terms, delayed or decaying effects, coupling terms, neural modules, and residual corrections.

4.2 Model Execution and Parameter Calibration

For the drafted model h, its continuous parameters $\boldsymbol{\theta}$ are fit to the observed data by minimizing a composite objective:

$$\mathcal{J}(h) \triangleq \min_{\boldsymbol{\theta}} \mathcal{L}_{\text{data}}(f_{h,\boldsymbol{\theta}}), \tag{4}$$

where $\mathcal{L}_{\text{data}}$ is a standard classification or forecasting loss (e.g., MSE). This yields a calibrated model $f_{h,\boldsymbol{\theta}_n}$ and a performance score $\mathcal{J}(h)$. Concretely, we use the Adam optimizer to learn the model parameters. Besides, if the hypothesis h is erroneous and fails to run, we will return the error message to the LLM and prompt it to make revisions until the model can be successfully executed.

4.3 Iterative Model Refinement

To efficiently navigate the vast search space, TimeCoder employs an iterative refinement loop that interleaves counterexample identification, historical experience replay, and LLM-driven revision.

Empirical Misalignment Identification with Prototypical Counterexamples. Given the original training trajectories, we first construct a small set of prototypical samples that represent diverse interventionresponse patterns. Concretely, we cluster the multivariate time series with features including both intervention signals and the observed dynamics at the last time step. We use the K-means algorithm (Hartigan & Wong, 1979) and set K to 20 to keep a balance between representative ability and complexity. For each cluster, we select the representative example whose dynamics are closest to the cluster centroid. This yields a compact prototype set $\mathcal{D}_{proto}$, which covers the diverse modes of variation in the real data while being computationally light for iterative evaluation.

At each iteration i, the current calibrated model $f_{h,\boldsymbol{\theta}}$ is evaluated on all prototypes. We compute the task-specific loss (e.g., RMSE for forecasting) for each prototype and rank them. The 5 prototypes with the highest losses are identified, representing the failure modes where the current hypothesis most severely deviates from observed behavior. For each such prototype, we record its input context, the models prediction, the ground truth, and a textual summary of the prediction residuals. This information constitutes a set of prototypical counterexample that highlight concrete deficiencies. This counterexample-wise empirical misalignment information $\mathcal{C}^i$ will be used to augment contexts for LLM-based hypothesis revision.

Experience Buffer and Multi-Hypothesis Memory. In addition to the current iterations counterexamples, TimeCoder maintains an experience buffer $\mathcal{M}$ (Shojaee et al., 2024; Holt et al., 2024) that stores the top5 best hypotheses discovered throughout the entire refinement history, along with their validation performance. This memory serves two purposes:

- It prevents the search from oscillating or collapsing to poorly performing structures, as the LLM can refer back to previously successful configurations.
- It enables experience-augmented prompting: when proposing a new hypothesis, the LLM is explicitly conditioned on the hypothesis that has worked well in the past.

Therefore, we preserve this memory to enhance LLM contexts for subsequent hypothesis refinements.

Iterative Refinement Loop. At iteration step i, the prototypical counterexamples information $\mathcal{C}^i$ and the historical top5 buffer $\mathcal{M}^i$ are concatenated to the standard hypothesis generation prompt defined in Eq. 3, which already contains the task specification, domain knowledge, and statistical descriptors. The LLM then proposes a revised hypothesis h^{i+1} and goes through the calibration process again. This generate-calibrate-refine cycle enables structure discovery without assuming a fixed parametric architecture, while explicitly enforcing domain-consistent temporal dynamics.

We repeat this loop until convergence, i.e., no edit yields sufficient improvement:

$$i \geq T_{\max} \quad \text{or} \quad \mathcal{L}_{\text{data}}\big(f_{h^i,\boldsymbol{\theta}^i}; \mathcal{D}\big) - \mathcal{L}_{\text{data}}\big(f_{h^{i+1},\boldsymbol{\theta}^{i+1}}; \mathcal{D}\big) \leq \epsilon. \tag{5}$$

5 Experiments

5.1 Training Details

Datasets To validate the effectiveness of TimeCoder, we conduct extensive experiments on time series datasets with two key considerations: (I) different levels of the availability and reliability of

Table 1: Forecasting performance of TimeCoder and baselines. Blue cells indicate the best result, orange cells indicate the second-best result, and ↓ indicates lower is better.

Model	SF^{2}Bench		Lung cancer		VitalDB		COVID-19	
	RMSE (↓)	MAE (↓)	RMSE (↓)	MAE (↓)	RMSE (↓)	MAE (↓)	RMSE (↓)	MAE (↓)
DLinear (Zeng et al., 2023)	0.391 ± 0.000	0.233 ± 0.000	0.617 ± 0.031	0.518 ± 0.036	0.808 ± 0.158	0.409 ± 0.123	0.157 ± 0.021	0.125 ± 0.019
TimesNet (Wu et al., 2023)	0.384 ± 0.003	0.209 ± 0.001	0.158 ± 0.007	0.107 ± 0.005	0.821 ± 0.166	0.391 ± 0.131	0.031 ± 0.004	0.021 ± 0.002
Autoformer (Wu et al., 2021)	0.420 ± 0.007	0.251 ± 0.006	0.345 ± 0.063	0.240 ± 0.037	0.830 ± 0.160	0.417 ± 0.114	0.055 ± 0.003	0.043 ± 0.002
Crossformer (Zhang & Yan, 2023)	0.364 ± 0.002	0.206 ± 0.004	0.111 ± 0.004	0.076 ± 0.003	0.820 ± 0.163	0.403 ± 0.100	0.025 ± 0.001	0.017 ± 0.001
PatchTST (Nie et al., 2023b)	0.375 ± 0.000	0.207 ± 0.000	0.188 ± 0.001	0.128 ± 0.001	0.814 ± 0.173	0.379 ± 0.138	0.054 ± 0.001	0.041 ± 0.002
iTransformer (Liu et al., 2024b)	0.368 ± 0.001	0.198 ± 0.001	0.194 ± 0.003	0.136 ± 0.002	0.815 ± 0.176	0.378 ± 0.138	0.056 ± 0.001	0.042 ± 0.002
TSMixer (Chen et al., 2023a)	0.376 ± 0.002	0.222 ± 0.003	0.172 ± 0.014	0.131 ± 0.010	0.802 ± 0.162	0.405 ± 0.111	0.061 ± 0.006	0.049 ± 0.005
TimeMixer (Wang et al., 2024a)	0.386 ± 0.002	0.220 ± 0.002	0.284 ± 0.008	0.196 ± 0.005	0.820 ± 0.178	0.390 ± 0.146	0.068 ± 0.003	0.054 ± 0.002
Time-LLM (Jin et al., 2024a)	0.352 ± 0.002	0.185 ± 0.001	0.136 ± 0.002	0.097 ± 0.002	0.495 ± 0.052	0.321 ± 0.087	0.028 ± 0.002	0.019 ± 0.001
PromptCast (Xue & Salim, 2023)	0.378 ± 0.003	0.211 ± 0.002	0.165 ± 0.005	0.112 ± 0.003	0.792 ± 0.148	0.425 ± 0.117	0.045 ± 0.003	0.032 ± 0.002
OFA (Zhou et al., 2023)	0.369 ± 0.002	0.203 ± 0.001	0.142 ± 0.004	0.095 ± 0.002	0.756 ± 0.135	0.401 ± 0.109	0.039 ± 0.002	0.027 ± 0.001
FSCA (Hu et al., 2025b)	0.358 ± 0.001	0.192 ± 0.001	0.121 ± 0.003	0.082 ± 0.002	0.512 ± 0.061	0.347 ± 0.092	0.025 ± 0.001	0.017 ± 0.001
TimeCMA (Liu et al., 2025a)	0.372 ± 0.002	0.208 ± 0.002	0.158 ± 0.004	0.107 ± 0.003	0.783 ± 0.142	0.418 ± 0.112	0.042 ± 0.002	0.030 ± 0.001
TimeCAP (Lee et al., 2025)	0.382 ± 0.003	0.217 ± 0.002	0.179 ± 0.005	0.123 ± 0.003	0.815 ± 0.156	0.432 ± 0.121	0.050 ± 0.003	0.035 ± 0.002
LLMTime (Gruver et al., 2023b)	0.395 ± 0.004	0.228 ± 0.003	0.198 ± 0.006	0.139 ± 0.004	0.837 ± 0.168	0.445 ± 0.132	0.058 ± 0.003	0.040 ± 0.002
TimeXL (Jiang et al., 2025)	0.348 ± 0.002	0.181 ± 0.001	0.112 ± 0.003	0.068 ± 0.002	0.528 ± 0.004	0.507 ± 0.004	0.026 ± 0.002	0.017 ± 0.001
TimeVLM (Zhong et al., 2025)	0.387 ± 0.002	0.221 ± 0.001	0.191 ± 0.005	0.134 ± 0.004	0.828 ± 0.161	0.438 ± 0.127	0.054 ± 0.003	0.038 ± 0.002
ChatTS (Xie et al., 2024)	0.384 ± 0.004	0.217 ± 0.001	0.187 ± 0.005	0.130 ± 0.004	0.821 ± 0.154	0.432 ± 0.121	0.051 ± 0.003	0.036 ± 0.002
HDTwin (Holt et al., 2024)	0.362 ± 0.003	0.213 ± 0.002	0.109 ± 0.002	0.103 ± 0.002	0.498 ± 0.003	0.346 ± 0.002	0.012 ± 0.001	0.012 ± 0.002
VoT (Wang et al., 2026)	0.551 ± 0.003	0.357 ± 0.002	0.341 ± 0.002	0.458 ± 0.002	0.653 ± 0.003	0.370 ± 0.002	0.014 ± 0.001	0.043 ± 0.002
TaTS (Li et al., 2025)	0.486 ± 0.003	0.315 ± 0.002	0.335 ± 0.003	0.450 ± 0.002	0.592 ± 0.003	0.353 ± 0.002	0.013 ± 0.002	0.033 ± 0.002
TimeCoder	0.328 ± 0.002	0.184 ± 0.001	0.092 ± 0.002	0.061 ± 0.002	0.445 ± 0.042	0.298 ± 0.078	0.009 ± 0.001	0.006 ± 0.001

domain priors, which range from equation-based physical laws to empirical, expert-derived heuristics; and (II) different complexity levels of intervention signals, varying from simple binary events to sparse and heterogeneous multi-intervention settings. We select four datasets that cover various domains, including SF^{2}Bench (Zheng et al., 2025), VitalDB (Lee et al., 2022), COVID-19 (Kerr et al., 2021) and Lung cancer (Geng et al., 2017). The details of these datasets are provided in Appendix E.1.

Baselines and Metrics. We compare TimeCoder with state-of-the-art models. (1) Deep learning-based models, including: DLinear (Zeng et al., 2023), TimesNet (Wu et al., 2023), Autoformer (Wu et al., 2021), Crossformer (Zhang & Yan, 2023), PatchTST (Nie et al., 2023b), iTransformer (Liu et al., 2024b), TSMixer (Chen et al., 2023a), TimeMixer (Wang et al., 2024a); (2) LLM-based methods: Time-LLM (Jin et al., 2024a), PromptCast (Xue & Salim, 2023), OFA (Zhou et al., 2023), FSCA (Hu et al., 2025b), TimeCMA (Liu et al., 2025a), TimeCAP (Lee et al., 2025), LLMTime (Gruver et al., 2023b), TimeXL (Jiang et al., 2025), TimeVLM (Zhong et al., 2025), ChatTS (Xie et al., 2024), HDTwin (Holt et al., 2024), VoT (Wang et al., 2026), TaTS (Li et al., 2025). For a fair comparison, we adopt Qwen3-32B-Instruction as backbone for all LLM-related baselines and TimeCoder. For TimeCoder, we set $T_{\max}$ as 10 in this paper. Importantly, all LLM-based baselines are provided with the same task specification, variable descriptions, domain knowledge context, and statistical summaries used by TimeCoder. This design ensures a fair comparison. Implementations of baselines can be found in their official GitHub repositories or in the Time-Series-Library[1]. All datasets are split 6:2:2 for training, validation, and testing. All results are averaged over five experimental runs (See Appendix E.2 for more details).

5.2 Main Results

Experimental results on forecasting task are shown in Table 1. We additionally report and analyze the classification results on the VitalDB dataset in Appendix Table 4. Several key observations can be made from the following dimensions: (1) Superiority of TimeCoder. TimeCoder outperforms all baselines across most datasets, particularly on tasks characterized by complex external interventions such as VitalDB dataset, demonstrating the effectiveness of structured refinement grounded in domain priors and statistical evidence. (2) Deep-learning methods struggle to handle distribution shifts induced by external interventions. In particular, on Lung Cancer and VitalDB, classical models exhibit significant performance degradation, indicating the importance of introducing domain knowledge priors. (3) LLM-based methods are suboptimal in capturing continuous numerical trends and temporal patterns. For example, on the Lung Cancer dataset, classical forecasting models outperform LLM-based methods. This is likely due to the relatively smooth and structured tumor growth dynamics, which are easier to capture in a data-driven manner.

5.3 Analysis of Hypothesis Refinement

Can TimeCoder Improve Temporal Modeling by Iterative Refinement? We analyze this from an empirical point of view to determine if TimeCoder can correctly refine the generated code and reduce its prediction error over subsequent generations. Figure 2 visualizes

[1] https://github.com/thuml/Time-Series-Library

TimeCoders iterative refinement process on the COVID-19 dataset by tracking (i) the RMSE results of the newly generated model at each generation and (ii) the best historical RMSE achieved so far (Top-1). Overall, the best historical curve decreases monotonically, indicating that TimeCoder can progressively discover and consolidate more accurate model structures through its refinement loop, rather than relying on a single-shot architecture choice (See Figure 15 to Figure 43 in the Appendix for detailed descriptions of structure refinement).

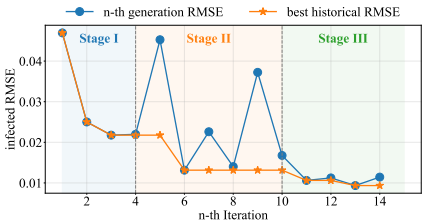


Figure 2: Evolution of TimeCoder on the COVID-19 dataset. The RMSE of each newly generated model is plotted alongside the best historical model (Top-1). TimeCoder progressively refines its structures, resulting in steady improvements.

How Does TimeCoder Evolve in Different Stages of Refinement? As shown in Fig. 2, in the first stage, TimeCoder introduces a white-box SEIR system as the core dynamical prior representing policy-driven contact-rate changes. This step yields a sharp reduction of the best RMSE. In the second stage, TimeCoder augments the SEIR backbone with latent memory and lightweight residual terms. This addition targets the non-Markovian and heterogeneous effects that are not well-modeled by a fixed-parameter SEIR system. In the third stage, TimeCoder performs architectural refinements, such as increasing the depth of residual modules and adopting higher-order ODE solvers. Importantly, these refinements improve predictive accuracy without compromising interpretability, as the dominant system behavior remains governed by the mechanistic SEIR transitions and intervention effects.

How Does TimeCoder Generalize to Sparse or Unobserved Interventions? To evaluate generalization to unseen drug interventions, we conduct controlled held-out intervention experiments on the challenging VitalDB dataset. We consider two settings: common-drug held-out and rare-drug held-out. We choose propofol as the common drug and remifentanil as the rare drug. In each setting, all training samples involving the selected drug are removed. Based on training-set medication frequencies, we select propofol as the common held-out drug and remifentanil as the rare held-out drug. Detailed Discussion can be found in Table 3 the Appendix section E.3. We evaluate on both the full test set and a drug-specific test subset containing only samples with the held-out drug, allowing us to assess both overall robustness and generalization to unseen interventions.

Table 2: Generalization to unseen interventions on the VitalDB dataset.

Model	Common Drug Held-Out				Rare Drug Held-Out			
	Full Test		Drug-Only Test		Full Test		Drug-Only Test	
	RMSE (↓)	MAE (↓)	RMSE (↓)	MAE (↓)	RMSE (↓)	MAE (↓)	RMSE (↓)	MAE (↓)
iTransformer	0.901	0.453	1.107	0.512	0.876	0.438	1.184	0.526
Time-LLM	0.872	0.431	1.034	0.395	0.846	0.419	1.126	0.498
TimeCoder	0.831	0.404	0.972	0.368	0.807	0.392	1.061	0.462

Table 2 shows that TimeCoder consistently outperforms both iTransformer and Time-LLM under common-drug and rare-drug held-out settings. The improvement is observed not only on the full test set, but also on the more challenging drug-only subsets, where all samples involve interventions that are unseen during training. This indicates that TimeCoder learns more transferable intervention-response dynamics rather than simply fitting observed medication patterns. As expected, drug-only evaluation is generally harder than full-test evaluation, and the rare-drug held-out setting leads to larger errors, reflecting the sparsity and heterogeneity of rare medication responses. Nevertheless, TimeCoder maintains the lowest RMSE and MAE across all settings, suggesting stronger robustness to sparse or unobserved interventions.

How Does TimeCoder Learn with Varying Number of Samples? Figure 3 reports the prediction performance of TimeCoder under varying training ratios across four datasets. On SF^{2}Bench and VitalDB, both RMSE and MAE decrease when the training ratio increases from 0.01 to 0.25, and become smoother afterwards. On Lung Cancer and COVID-19, the performance remains relatively stable even under extremely limited supervision. This result suggests that in systems with more complex or heterogeneous dynamics (e.g., VitalDB and SF^{2}Bench), addi-

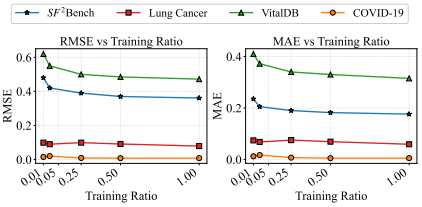


Figure 3: Ablation Study on the number of training samples.

tional data helps better calibrate residual structures and parameters, while in domains with strong and well-specified mechanistic patterns (e.g., COVID-19 epidemiological dynamics and tumor growth processes), our model has enough domain knowledge to enable accurate prediction.

Can TimeCoder Capture Intervention-Conditioned Temporal Dynamics? We provide qualitative forecasting examples and detailed discussions on SF^2Bench in Appendix Figure 4. The visualizations show that TimeCoder more faithfully tracks the intervention-conditioned trajectory, including the early double-peak response, intermediate trough, and subsequent rebound.

How Does TimeCoder Refine Generated Forecasting Programs? We provide two Appendix examples of feedback-driven refinement. Section E.8.3 illustrates an execution-guided repair example, where invalid function outputs are detected and repaired into an executable program. Section E.8.4 illustrates diagnostic-guided refinement, where statistical descriptors provide fine-grained feedback for improving temporal dynamics.

5.4 Ablation Study

Influence of Different Base Models. Table 6 in the Appendix compares TimeCoder across base LLMs and shows consistently strong performance. Qwen3-Coder-Flash achieves the best or near-best RMSE and MAE, highlighting the value of code-oriented backbones, whereas Gemini-2.5-Pro is weaker on SF^2Bench and VitalDB and Qwen3-32B-Instruct remains competitive. We use Qwen3-32B-Instruct in the main results to ensure controlled comparison with LLM baselines under the same general-purpose backbone.

Influence of Model Components. Table 7 quantifies the contributions of statistical descriptors and domain-aware guidance. Removing statistical descriptors increases RMSE from 0.328 to 0.361 on SF^2Bench and from 0.445 to 0.472 on VitalDB. Removing domain descriptions further increases RMSE to 0.365 and 0.493, respectively, confirming that domain knowledge guides the LLM toward meaningful code structures.

Computational Cost and Convergence Behavior. Figure 5 reports token consumption for prompt construction, critique, and code synthesis across refinement generations. Usage grows rapidly during early structural exploration and stabilizes after convergence; this one-time offline search cost is followed by standard executable-model deployment.

Influence of Memory. Table 5 shows that removing historical-generation memory consistently degrades performance, confirming its value for retrieving informative candidate structures. Increasing top-k from 1 to 4 monotonically reduces RMSE across datasets, indicating that multiple candidates improve structural exploration and mitigate suboptimal local solutions.

6 Conclusion and Future Work

We introduced **TimeCoder**, an intervention-aware time-series framework that treats model structure as an object of automated reasoning and refinement. Its closed-loop integration of domain knowledge, data-driven calibration, and LLM reasoning produces interpretable hybrid models robust to sparse, heterogeneous interventions. TimeCoder consistently outperforms state-of-the-art baselines across healthcare, hydrology, and epidemic forecasting benchmarks.

Future work will extend TimeCoder from single systems to population and multi-agent settings, incorporate uncertainty quantification into structure generation and calibration for high-stakes applications, and improve iterative-refinement efficiency.

AI Use Statement

Generative AI tools were used in two ways in this work. First, as part of the TimeCoder framework, they were used to propose, implement, and iteratively refine executable intervention-aware model structures. Second, they were used to assist with polishing the writing. All AI-assisted code and artifacts used in the experiments were executed and evaluated within the authors' pipeline. The authors reviewed the AI-assisted work and take responsibility for the final content of this paper, including all text, claims, results, and artifacts.

References

Microsoft Research AI4Science and Microsoft Azure Quantum. The impact of large language models on scientific discovery: a preliminary study using gpt-4. *arXiv preprint arXiv:2311.07361*,

2023.

Hadeel AlQadi and Majid Bani-Yaghoub. Incorporating global dynamics to improve the accuracy of disease models: Example of a covid-19 sir model. *Plos one*, 17(4):e0265815, 2022.

Abdul Fatir Ansari, Lorenzo Stella, Caner Turkmen, Xiyuan Zhang, Pedro Mercado, Huibin Shen, Oleksandr Shchur, Syama Sundar Rangapuram, Sebastian Pineda Arango, Shubham Kapoor, et al. Chronos: Learning the language of time series. *arXiv preprint arXiv:2403.07815*, 2024.

Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Accelerating neural architecture search using performance prediction. *arXiv preprint arXiv:1705.10823*, 2017.

Ioana Bica, Ahmed M Alaa, James Jordon, and Mihaela Van Der Schaar. Estimating counterfactual treatment outcomes over time through adversarially balanced representations. *arXiv preprint arXiv:2002.04083*, 2020.

Wanlin Cai, Yuxuan Liang, Xianggen Liu, Jianshuai Feng, and Yuankai Wu. Msgnet: Learning multi-scale inter-series correlations for multivariate time series forecasting. In *Proceedings of the AAAI conference on artificial intelligence*, volume 38, pp. 11141–11149, 2024.

Ching Chang, Wen-Chih Peng, and Tien-Fu Chen. Llm4ts: Two-stage fine-tuning for time-series forecasting with pre-trained llms. *arXiv preprint arXiv:2308.08469*, 2023.

Si-An Chen, Chun-Liang Li, Sercan O Arik, Nathanael Christian Yoder, and Tomas Pfister. Tsmixer: An all-mlp architecture for time series forecast-ing. *Transactions on Machine Learning Research*, 2023a.

Zihan Chen, Lei Nico Zheng, Cheng Lu, Jialu Yuan, and Di Zhu. Chatgpt informed graph neural network for stock movement prediction. *Available at SSRN 4464002*, 2023b.

Miles Cranmer, Sam Greydanus, Stephan Hoyer, Peter Battaglia, David Spergel, and Shirley Ho. Lagrangian neural networks. *arXiv preprint arXiv:2003.04630*, 2020.

Salvatore Cuomo, Vincenzo Schiano Di Cola, Fabio Giampaolo, Gianluigi Rozza, Maziar Raissi, and Francesco Piccialli. Scientific machine learning through physics–informed neural networks: Where we are and whats next. *Journal of Scientific Computing*, 92(3):88, 2022.

Nicola Dainese, Matteo Merler, Minttu Alakuijala, and Pekka Marttinen. Generating code world models with large language models guided by monte carlo tree search. *Advances in Neural Information Processing Systems*, 37:60429–60474, 2024.

Ibrahim Delibasoglu, Sanjay Chakraborty, Fredrik Heintz, and Mattias Tiger. Intervention-aware time series modeling: Capturing and evaluating feature dependencies. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 40, pp. 20711–20718, 2026.

Changran Geng, Harald Paganetti, and Clemens Grassberger. Prediction of treatment response for combined chemo-and radiation therapy for non-small cell lung cancer patients using a biomathematical model. *Scientific reports*, 7(1):13542, 2017.

Samuel Greydanus, Misko Dzamba, and Jason Yosinski. Hamiltonian neural networks. *Advances in neural information processing systems*, 32, 2019.

Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew G Wilson. Large language models are zero-shot time series forecasters. In *NeurIPS*, 2023a.

Nate Gruver, Marc Finzi, Shikai Qiu, and Andrew Gordon Wilson. Large Language Models Are Zero Shot Time Series Forecasters. In *NeurIPS*, 2023b.

John A Hartigan and Manchek A Wong. Algorithm as 136: A k-means clustering algorithm. *Journal of the royal statistical society. series c (applied statistics)*, 28(1):100–108, 1979.

David Haussler and Manfred Warmuth. The probably approximately correct (pac) and other learning models. *The Mathematics of Generalization*, pp. 17–36, 2018.

Samuel Holt, Tennison Liu, and Mihaela van der Schaar. Automatically learning hybrid digital twins of dynamical systems. *Advances in Neural Information Processing Systems*, 37:72170–72218, 2024.

Samuel Holt, Max Ruiz Luyten, Antonin Berthon, and Mihaela Van Der Schaar. G-sim: Generative simulations with large language models and gradient-free calibration. *arXiv preprint arXiv:2506.09272*, 2025.

Yuxiao Hu, Qian Li, Dongxiao Zhang, Jinyue Yan, and Yuntian Chen. Context-alignment: Activating and enhancing llm capabilities in time series. *arXiv preprint arXiv:2501.03747*, 2025a.

Yuxiao Hu, Qian Li, Dongxiao Zhang, Jinyue Yan, and Yuntian Chen. Context-alignment: Activating and enhancing LLMs capabilities in time series. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL https://openreview.net/forum?id=syC2764fPc.

Aya Abdelsalam Ismail, Mohamed Gunady, Hector Corrada Bravo, and Soheil Feizi. Benchmarking deep learning interpretability in time series predictions. *Advances in neural information processing systems*, 33:6441–6452, 2020.

Yushan Jiang, Zijie Pan, Xikun Zhang, Sahil Garg, Anderson Schneider, Yuriy Nevmyvaka, and Dongjin Song. Empowering time series analysis with large language models: A survey. In Kate Larson (ed.), *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pp. 8095–8103. International Joint Conferences on Artificial Intelligence Organization, 8 2024. doi: 10.24963/ijcai.2024/895. URL https://doi.org/10.24963/ijcai.2024/895. Survey Track.

Yushan Jiang, Wenchao Yu, Geon Lee, Dongjin Song, Kijung Shin, Wei Cheng, Yanchi Liu, and Haifeng Chen. Timexl: Explainable multi-modal time series prediction with llm-in-the-loop. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Ming Jin, Shiyu Wang, Lintao Ma, Zhixuan Chu, James Y Zhang, Xiaoming Shi, Pin-Yu Chen, Yuxuan Liang, Yuan-Fang Li, Shirui Pan, et al. Time-llm: Time series forecasting by reprogramming large language models. In *The Twelfth International Conference on Learning Representations*, 2024a.

Yiqiao Jin, Qinlin Zhao, Yiyang Wang, Hao Chen, Kaijie Zhu, Yijia Xiao, and Jindong Wang. Agentreview: Exploring peer review dynamics with llm agents. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 1208–1226, 2024b.

Cliff C Kerr, Robyn M Stuart, Dina Mistry, Romesh G Abeysuriya, Katherine Rosenfeld, Gregory R Hart, Rafael C Núñez, Jamie A Cohen, Prashanth Selvaraj, Brittany Hagedorn, et al. Covasim: an agent-based model of covid-19 dynamics and interventions. *PLOS Computational Biology*, 17 (7):e1009149, 2021.

Uiyun Kim, Ohhyeon Kwon, and Doheon Lee. Quantifying scientific consensus in biomedical hypotheses via llm-assisted literature screening. *bioRxiv*, pp. 2026–04, 2026.

Shrinidhi Kumbhar, Venkatesh Mishra, Kevin Coutinho, Divij Handa, Ashif Iquebal, and Chitta Baral. Hypothesis generation for materials discovery and design using goal-driven and constraint-guided llm agents. In *Findings of the Association for Computational Linguistics: NAACL 2025*, pp. 7524–7555, 2025.

Geon Lee, Wenchao Yu, Kijung Shin, Wei Cheng, and Haifeng Chen. Timecap: Learning to contextualize, augment, and predict time series events with large language model agents. In *AAAI*, 2025.

Hyung-Chul Lee, Yoonsang Park, Soo Bin Yoon, Seong Mi Yang, Dongnyeok Park, and Chul-Woo Jung. Vitaldb, a high-fidelity multi-parameter vital signs database in surgical patients. *Scientific Data*, 9(1):279, 2022.

Kin Kwan Leung, Clayton Rooke, Jonathan Smith, Saba Zuberi, and Maksims Volkovs. Temporal dependencies in feature importance for time series prediction. In *The Eleventh International Conference on Learning Representations*, 2023.

Michael Y Li, Emily B Fox, and Noah D Goodman. Automated statistical model discovery with language models. *arXiv preprint arXiv:2402.17879*, 2024.

Zihao Li, Xiao Lin, Zhining Liu, Jiaru Zou, Ziwei Wu, Lecheng Zheng, Dongqi Fu, Yada Zhu, Hendrik Hamann, Hanghang Tong, et al. Language in the flow of time: Time-series-paired texts weaved into a unified temporal narrative. *arXiv preprint arXiv:2502.08942*, 2025.

Bryan Lim and Stefan Zohren. Time-series forecasting with deep learning: a survey. *Philosophical transactions of the royal society a: mathematical, physical and engineering sciences*, 379(2194), 2021.

Chenxi Liu, Qianxiong Xu, Hao Miao, Sun Yang, Lingzheng Zhang, Cheng Long, Ziyue Li, and Rui Zhao. Timecma: Towards llm-empowered multivariate time series forecasting via cross-modality alignment. In *AAAI*, 2025a.

Chenxi Liu, Shaowen Zhou, Qianxiong Xu, Hao Miao, Cheng Long, Ziyue Li, and Rui Zhao. Towards cross-modality modeling for time series analytics: A survey in the llm era. *arXiv preprint arXiv:2505.02583*, 2025b.

Haoxin Liu, Shangqing Xu, Zhiyuan Zhao, Lingkai Kong, Harshavardhan Kamarthi, Aditya B Sasanur, Megha Sharma, Jiaming Cui, Qingsong Wen, Chao Zhang, et al. Time-mmd: A new multi-domain multimodal dataset for time series analysis. *arXiv preprint arXiv:2406.08627*, 2024a.

Xin Liu, Daniel McDuff, Geza Kovacs, Isaac Galatzer-Levy, Jacob Sunshine, Jiening Zhan, Ming-Zher Poh, Shun Liao, Paolo Di Achille, and Shwetak Patel. Large language models are few-shot health learners. *arXiv preprint arXiv:2305.15525*, 2023.

Yong Liu, Tengge Hu, Haoran Zhang, Haixu Wu, Shiyu Wang, Lintao Ma, and Mingsheng Long. itransformer: Inverted transformers are effective for time series forecasting. In *The Twelfth International Conference on Learning Representations*, 2024b.

Yuqiao Liu, Yanan Sun, Bing Xue, Mengjie Zhang, Gary G Yen, and Kay Chen Tan. A survey on evolutionary neural architecture search. *IEEE transactions on neural networks and learning systems*, 34(2):550–570, 2021.

Zimeng Lyu, Alexander Ororbia, and Travis Desell. Online evolutionary neural architecture search for multivariate non-stationary time series forecasting. *Applied Soft Computing*, 145:110522, 2023.

Pingchuan Ma, Tsun-Hsuan Wang, Minghao Guo, Zhiqing Sun, Joshua B Tenenbaum, Daniela Rus, Chuang Gan, and Wojciech Matusik. Llm and simulation as bilevel optimizers: A new paradigm to advance physical scientific discovery. *arXiv preprint arXiv:2405.09783*, 2024.

Bodhisattwa Prasad Majumder, Harshit Surana, Dhruv Agarwal, Sanchaita Hazra, Ashish Sabharwal, and Peter Clark. Position: data-driven discovery with large generative models. In *Forty-first International Conference on Machine Learning*, 2024.

Kevin P Murphy. *Machine learning: a probabilistic perspective*. 2012.

Yuqi Nie, Nam H Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. A time series is worth 64 words: Long-term forecasting with transformers. In *ICLR*, 2023a.

Yuqi Nie, Nam H Nguyen, Phanwadee Sinthong, and Jayant Kalagnanam. A time series is worth 64 words: Long-term forecasting with transformers. In *The Eleventh International Conference on Learning Representations*, 2023b.

Yuqi Nie, Yaxuan Kong, Xiaowen Dong, John M Mulvey, H Vincent Poor, Qingsong Wen, and Stefan Zohren. A survey of large language models for financial applications: Progress, prospects and challenges. 2024.

Kanghui Ning, Zijie Pan, Yu Liu, Yushan Jiang, James Yiming Zhang, Kashif Rasul, Anderson Schneider, Lintao Ma, Yuriy Nevmyvaka, and Dongjin Song. Ts-rag: Retrieval-augmented generation based time series foundation models are stronger zero-shot forecaster. *arXiv preprint arXiv:2503.07649*, 2025.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Biqing Qi, Kaiyan Zhang, Haoxiang Li, Kai Tian, Sihang Zeng, Zhang-Ren Chen, and Bowen Zhou. Large language models are zero shot hypothesis proposers. *arXiv preprint arXiv:2311.05965*, 2023.

Zhaozhi Qian, William Zame, Lucas Fleuren, Paul Elbers, and Mihaela van der Schaar. Integrating expert odes into neural odes: pharmacology and disease progression. *Advances in Neural Information Processing Systems*, 34:11364–11383, 2021.

Yao Qin, Dongjin Song, Haifeng Cheng, Wei Cheng, Guofei Jiang, and Garrison W Cottrell. A dual-stage attention-based recurrent neural network for time series prediction. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pp. 2627–2633, 2017.

Maziar Raissi, Paris Perdikaris, and George E Karniadakis. Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational physics*, 378:686–707, 2019.

Hojjat Rakhshani, Hassan Ismail Fawaz, Lhassane Idoumghar, Germain Forestier, Julien Lepagnot, Jonathan Weber, Mathieu Brévilliers, and Pierre-Alain Muller. Neural architecture search for time series classification. In *2020 International Joint Conference on Neural Networks (IJCNN)*, pp. 1–8. IEEE, 2020.

Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Xiaojiang Chen, and Xin Wang. A comprehensive survey of neural architecture search: Challenges and solutions. *ACM Computing Surveys (CSUR)*, 54(4):1–34, 2021.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Michael Moor, Zicheng Liu, and Emad Barsoum. Agent laboratory: Using llm agents as research assistants. *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 5977–6043, 2025.

Eli Schwartz, Leshem Choshen, Joseph Shtok, Sivan Doveh, Leonid Karlinsky, and Assaf Arbelle. Numerologic: Number encoding for enhanced llms numerical reasoning. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pp. 206–212, 2024.

Zhuocheng Shen. Llm with tools: A survey. *arXiv preprint arXiv:2409.18807*, 2024.

Parshin Shojaee, Kazem Meidani, Shashank Gupta, Amir Barati Farimani, and Chandan K Reddy. Llm-sr: Scientific equation discovery via programming with large language models. *arXiv preprint arXiv:2404.18400*, 2024.

Kevin Song, Andrew Trotter, and Jake Y Chen. Llm agent swarm for hypothesis-driven drug discovery. *arXiv preprint arXiv:2504.17967*, 2025.

Chenxi Sun, Yaliang Li, Hongyan Li, and Shenda Hong. Test: Text prototype aligned embedding to activate llm's ability for time series. In *ICLR*, 2024.

Naoya Takeishi and Alexandros Kalousis. Physics-integrated variational autoencoders for robust and interpretable generative modeling. *Advances in Neural Information Processing Systems*, 34: 14809–14821, 2021.

Hao Tang, Darren Key, and Kevin Ellis. Worldcoder, a model-based llm agent: Building world models by writing code and interacting with the environment. *Advances in Neural Information Processing Systems*, 37:70148–70212, 2024.

Patara Trirat, Wonyong Jeong, and Sung Ju Hwang. Automl-agent: A multi-agent llm framework for full-pipeline automl. *arXiv preprint arXiv:2410.02958*, 2024.

Jianxun Wang and Yixiang Chen. A review on code generation with llms: Application and evaluation. In *2023 IEEE International Conference on Medical Artificial Intelligence (MedAI)*, pp. 284–289. IEEE, 2023.

Shiyu Wang, Haixu Wu, Xiaoming Shi, Tengge Hu, Huakun Luo, Lintao Ma, James Y Zhang, and JUN ZHOU. Timemixer: Decomposable multiscale mixing for time series forecasting. In *The Twelfth International Conference on Learning Representations*, 2024a.

Siyuan Wang, Peng Chen, Yihang Wang, Wanghui Qiu, Chenjuan Guo, Bin Yang, and Yang Shu. Unlocking the value of text: Event-driven reasoning and multi-level alignment for time series forecasting. *arXiv preprint arXiv:2603.15452*, 2026.

Xinlei Wang, Maike Feng, Jing Qiu, Jinjin Gu, and Junhua Zhao. From news to forecast: Integrating event analysis in llm-based time series forecasting with reflection. *Advances in Neural Information Processing Systems*, 37:58118–58153, 2024b.

Yihang Wang, Yuying Qiu, Peng Chen, Yang Shu, Zhongwen Rao, Lujia Pan, Bin Yang, and Chenjuan Guo. Lightgts: A lightweight general time series forecasting model. In *International Conference on Machine Learning*, pp. 64109–64126. PMLR, 2025a.

Yinjie Wang, Ling Yang, Ye Tian, Ke Shen, and Mengdi Wang. Co-evolving llm coder and unit tester via reinforcement learning. *arXiv preprint arXiv:2506.03136*, 2025b.

Zhiyuan Wang, Xovee Xu, Weifeng Zhang, Goce Trajcevski, Ting Zhong, and Fan Zhou. Learning latent seasonal-trend representations for time series forecasting. *Advances in Neural Information Processing Systems*, 35:38775–38787, 2022.

Jiaxin Wen, Jian Guan, Hongning Wang, Wei Wu, and Minlie Huang. Codeplan: Unlocking reasoning potential in large language models by scaling code-form planning. In *The Thirteenth International Conference on Learning Representations*, 2024.

Haixu Wu, Jiehui Xu, Jianmin Wang, and Mingsheng Long. Autoformer: Decomposition transformers with auto-correlation for long-term series forecasting. *Advances in neural information processing systems*, 34:22419–22430, 2021.

Haixu Wu, Tengge Hu, Yong Liu, Hang Zhou, Jianmin Wang, and Mingsheng Long. Timesnet: Temporal 2d-variation modeling for general time series analysis. In *The Eleventh International Conference on Learning Representations*, 2023.

Oskar Wysocki, Danilo Carvalho, Alex Bogatu, André Freitas, et al. An llm-based knowledge synthesis and scientific reasoning framework for biomedical discovery. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 3: System Demonstrations)*, pp. 355–364, 2024.

Shijie Xia, Yuhan Sun, and Pengfei Liu. Sr-scientist: Scientific equation discovery with agentic ai. *arXiv preprint arXiv:2510.11661*, 2025.

Zhe Xie, Zeyan Li, Xiao He, Longlong Xu, Xidao Wen, Tieying Zhang, Jianjun Chen, Rui Shi, and Dan Pei. Chatts: Aligning time series with llms via synthetic data for enhanced understanding and reasoning. *arXiv preprint arXiv:2412.03104*, 2024.

Zhikai Xing and Yigang He. Multi-modal information analysis for fault diagnosis with time-series data from power transformer. *International Journal of Electrical Power & Energy Systems*, 144: 108567, 2023.

Zhijian Xu, Hao Wang, and Qiang Xu. Intervention-aware forecasting: Breaking historical limits from a system perspective. *arXiv preprint arXiv:2405.13522*, 2024.

Hao Xue and Flora D Salim. Promptcast: A new prompt-based learning paradigm for time series forecasting. *IEEE Transactions on Knowledge and Data Engineering*, 2023.

Yuan Yin, Vincent Le Guen, Jérémie Dona, Emmanuel de Bézenac, Ibrahim Ayed, Nicolas Thome, and Patrick Gallinari. Augmenting physical models with deep networks for complex dynamics forecasting. *Journal of Statistical Mechanics: Theory and Experiment*, 2021(12):124012, 2021.

Siyu Yuan, Kaitao Song, Jiangjie Chen, Xu Tan, Yongliang Shen, Kan Ren, Dongsheng Li, and Deqing Yang. Easytool: Enhancing llm-based agents with concise tool instruction. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pp. 951–972, 2025.

Ailing Zeng, Muxi Chen, Lei Zhang, and Qiang Xu. Are transformers effective for time series forecasting? In *Proceedings of the AAAI conference on artificial intelligence*, volume 37, pp. 11121–11128, 2023.

Miao Zhang, Huiqi Li, Shirui Pan, Xiaojun Chang, Zongyuan Ge, and Steven Su. Differentiable neural architecture search in equivalent space with exploration enhancement. *Advances in Neural Information Processing Systems*, 33:13341–13351, 2020.

Yunhao Zhang and Junchi Yan. Crossformer: Transformer utilizing cross-dimension dependency for multivariate time series forecasting. In *ICLR*, 2022.

Yunhao Zhang and Junchi Yan. Crossformer: Transformer utilizing cross-dimension dependency for multivariate time series forecasting. In *The eleventh international conference on learning representations*, 2023.

Zijun Zhang. Improved adam optimizer for deep neural networks. In *2018 IEEE/ACM 26th international symposium on quality of service (IWQoS)*, pp. 1–2. Ieee, 2018.

Xu Zheng, Chaohao Lin, Sipeng Chen, Zhuomin Chen, Jimeng Shi, Wei Cheng, Jayantha Obeysekera, Jason Liu, and Dongsheng Luo. SF^2 bench: Evaluating data-driven models for compound flood forecasting in south florida. *arXiv preprint arXiv:2506.04281*, 2025.

Li Zhong, Zilong Wang, and Jingbo Shang. Debug like a human: A large language model debugger via verifying runtime execution step by step. In *Findings of the Association for Computational Linguistics: ACL 2024*, pp. 851–870, 2024.

Siru Zhong, Weilin Ruan, Ming Jin, Huan Li, Qingsong Wen, and Yuxuan Liang. Time-vlm: Exploring multimodal vision-language models for augmented time series forecasting. *arXiv preprint arXiv:2502.04395*, 2025.

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 *AAAI*, 2021.

Tian Zhou, Peisong Niu, Liang Sun, Rong Jin, et al. One fits all: Power general time series analysis by pretrained lm. *Advances in neural information processing systems*, 36:43322–43355, 2023.

Tian Zhou, Peisong Niu, Liang Sun, Rong Jin, et al. One fits all: Power general time series analysis by pretrained lm. In *NeurIPS*, 2024.

Barret Zoph and Quoc V Le. Neural architecture search with reinforcement learning. *arXiv preprint arXiv:1611.01578*, 2016.

APPENDIX

A The Use of Large Language Models **17**

B Broader Impact **17**

C Additional Related Works **17**

D Method **18**

E Experiments **18**

E.1 Dataset 18
E.2 Metrics 19
E.3 Implementation Details 20
E.4 Main Results 20
E.4.1 Classification Results 20
E.5 Further Analysis 21
E.5.1 Influence of Memory 21
E.5.2 Ablation Study on the Influence of Different Base Models. 21
E.5.3 Ablation Study on the Components of TimeCoder. 21
E.5.4 Prompt Description 21
E.6 Visualization 21
E.7 Computational Cost and Convergence Behavior Analysis 24
E.8 Case Study 24
E.8.1 Analysis of a Full Iterative Refinement Behavior. 24
E.8.2 Failure Rate and Failure Type Analysis 24
E.8.3 Execution-Guided Program Repair Example 25
E.8.4 Diagnostic-Guided Code Refinement Example 29

A THE USE OF LARGE LANGUAGE MODELS

Large Language Models (LLMs) were employed in this project in a limited, assistive capacity. For manuscript preparation, all sections were drafted by the authors, with LLMs occasionally used to improve grammar, enhance clarity, and refine academic tone. During the implementation phase, LLMs served as coding assistants, providing code completions, debugging suggestions, and formatting support. However, all final code, experimental design, analyses, and validations were developed and verified solely by the authors. Importantly, LLMs were not used for generating research ideas, designing experiments, or conducting the literature review. All conceptual contributions, methodological innovations, and scientific insights originated exclusively from the authors.

B BROADER IMPACT

This work does not raise immediate ethical concerns in its current form, since it focuses on algorithm design rather than direct deployment in sensitive real-world settings.

C ADDITIONAL RELATED WORKS

We expand in the following the Related Work section, covering the works that try to build world models with language and those who explored using programs to express RL policies.

Scientific Discovery with LLMs. LLMs are increasingly being applied to scientific problems, with emerging use cases in biology, healthcare, and materials science (AI4Science & Quantum, 2023; Majumder et al., 2024). Recent work suggests that they can leverage broad domain knowledge and reasoning capabilities to generate scientifically meaningful hypotheses (Qi et al., 2023). When combined with external tools, simulators, or data-driven evaluators, LLM-based agents have further shown promise in iterative scientific discovery and model refinement (Li et al., 2024). Nevertheless, their use for discovering executable hypotheses in intervention-aware time-series settings remains largely unexplored. Unlike prior work that primarily focuses on symbolic equations (Xia et al., 2025) or general scientific reasoning (Shojaee et al., 2024; Ma et al., 2024), we study a more challenging class of dynamical systems in which intrinsic dynamics evolve over time and system behavior is further influenced by complex interventions. We introduce a novel LLM-based coding framework for intervention-aware time-series analysis that searches over structured dynamical programs by combining scientific domain knowledge, code generation, and data-driven evaluation in a unified refinement loop.

LLM-based coding agents. Code generation has emerged as a major application of LLMs (Tang et al., 2024; Trirat et al., 2024; Wen et al., 2024). Improvements in this area generally come from two directions: enhancing the intrinsic coding ability of the base model, and developing prompt-engineering or post-training (Wang et al., 2025b) strategies that better elicit and refine this capability. Our work is most closely related to the latter direction. One line of work focuses on *decomposition*, reducing the difficulty of end-to-end code generation by splitting the task into multiple stages or specialized roles, such as planning, coding, testing, and execution (). A second line emphasizes *feedback-driven refinement*, where initially generated programs are iteratively improved using execution outcomes, unit-test results, or self-reflection (Zhong et al., 2024). These methods are motivated by the observation that single-pass generations are often unreliable, and that iterative revision can substantially improve functional correctness (Wang et al., 2025b). A third line formulates coding as an explicit *search* problem over candidate programs or strategies, using mechanisms such as tree search or MCTS (Dainese et al., 2024) to better navigate large program spaces (Novikov et al., 2025). Methods such as LATS and CodeTree are representative of this direction. Despite their differences, most existing coding agents are designed for general-purpose code synthesis, software repair, or benchmark-driven test passing. In contrast, our goal is not to generate arbitrary correct programs, but to search over domain-constrained executable model hypotheses for intervention-aware time-series analysis.

Model Neural Architecture Search. Our work is also related to automated model design, including neural architecture search (Ren et al., 2021; Liu et al., 2021) and more recent LLM-guided model construction (Wang & Chen, 2023; Holt et al., 2024; AI4Science & Quantum, 2023; Ma et al., 2024). Classical architecture search methods aim to optimize predictive performance over

predefined search spaces (Rakhshani et al., 2020; Baker et al., 2017), typically using reinforcement learning (Zoph & Le, 2016), evolutionary algorithms (Lyu et al., 2023), or differentiable relaxation (Zhang et al., 2020). More recent LLM-based approaches suggest that language models can assist not only with code generation, but also with higher-level model proposal and revision. Nevertheless, most existing architecture search methods operate over relatively abstract architecture spaces and are optimized primarily for predictive accuracy. They typically do not treat mechanistic priors, intervention semantics, and executable dynamical structure as first-class objects of search. By contrast, we study structured search over executable model hypotheses for intervention-aware temporal dynamics, where the search process must jointly account for predictive accuracy, intervention robustness, and interpretable dynamical decomposition.

D METHOD

Statistical Descriptors. To systematically characterize latent temporal structures in prediction residuals, we employ five complementary statistical descriptors:

Let $r_t = y_t - \hat{y}_t$ denote the residual at time t. We extract complementary temporal structures using the following statistical descriptors:

- **Temporal Lag Detection.** Systematic delays between predicted dynamics and ground truth are quantified via cross-correlation between residuals and observed trajectories, with the dominant lag indicating temporal misalignment. Systematic temporal misalignment is quantified via cross-correlation $\text{Corr}(\ell) = \sum_{t=1}^{T-\ell} r(t)\, y(t+\ell)$,, where the dominant lag $\tau^* = \arg\max_\tau C(\tau)$ indicates delayed or advanced model response (computed using `scipy.signal.correlate`).
- **Trend Analysis.** Long-term bias accumulation is identified by fitting a linear model to residual sequences, where the slope magnitude and statistical significance indicate systematic drift. Long-term drift is identified by fitting a linear model $r_t = at + b + \epsilon_t$, where the slope magnitude a and its statistical significance indicate systematic bias accumulation (estimated via `scipy.stats.linregress`).
- **Periodicity Detection.** Cyclical error patterns are captured through residual autocorrelation, with prominent peaks revealing the presence and length of dominant periods. Cyclical error patterns are captured through residual autocorrelation $\text{ACF}(\tau) = \sum_{t=1}^{T-\tau} r(t)\, r(t+\tau)$, where prominent peaks reveal dominant periods (computed using autocorrelation implemented by `scipy.signal.correlate`).
- **Abrupt Change Detection.** Sudden regime shifts are detected by analyzing absolute first-order differences of residuals, highlighting unmodeled sharp transitions. Sudden regime shifts are detected by first-order residual differences $\Delta r(t) = |r(t) - r(t-1)|$, with large deviations highlighting unmodeled sharp transitions (computed using `numpy.diff` with adaptive thresholding).
- **Systematic Bias Quantification.** Persistent over- or under-prediction is measured by the mean residual relative to the data scale, indicating global model miscalibration. Persistent over- or under-prediction is measured by the mean residual $\mu_r = \frac{1}{T}\sum_{t=1}^{T} r_t$, normalized by the data scale to indicate global miscalibration (computed using `numpy.mean` and scale normalization).

These descriptors provide a compact and interpretable summary of residual temporal structures, iteratively forming the statistical evidence representation $\mathcal{Z}_t$ used to guide operator selection and structural refinement.

E EXPERIMENTS

E.1 DATASET

The VitalDB dataset (Lee et al., 2022) contains perioperative physiological time series recorded during surgery from 2016 to 2019, consisting of 6,388 surgical cases. Given the last 20 minutes of six vital signs together with vasoactive medications (including propofol, ephedrine, sugammadex, epinephrine, tranexamic acid, ondansetron, hydromorphone, oxytocin, nitroglycerin, cisatracurium,

Table 3: Medication frequency distribution used for selecting common and rare held-out interventions in the VitalDB dataset.

Medication	Count
propofol	102692
ephedrine	47931
sugammadex	8429
epinephrine	1975
tranexamic acid	1114
vasopressin	65
labetalol	64
phenylephrine	34
amiodarone	33
remifentanil	20

fentanyl, dexamethasone, norepinephrine, morphine, vasopressin, labetalol, phenylephrine, amiodarone, and remifentanil.), the task is to forecast the trajectories of all six vital signals over the next 10 minutes. In addition to forecasting, we further evaluate TimeCoder and baselines on a clinically important classification task, namely acute kidney injury (AKI) prediction, where the model predicts the onset of postoperative AKI based on perioperative temporal dynamics.

The SF^{2}Bench dataset (Zheng et al., 2025) contains hourly hydrological time series collected from 2,452 monitoring stations across a 67,349 km^2 region in South Florida from 1985 to 2024, sourced from the South Florida Water Management District (SFWMD). It integrates key drivers of compound floods, including water level, rainfall, groundwater level, and human control signals on hydraulic structures (i.e., pump operations and gate openings), where pump/gate controls serve as intervention variables; given the historical observations over the past 48 hours, the task is to forecast future water-level trajectories over the next 24 hours.

We construct a synthetic COVID-19 epidemic dataset (Kerr et al., 2021) using the simulator COVASIM (Kerr et al., 2021; AlQadi & Bani-Yaghoub, 2022), which models individual-level disease progression and supports both pharmaceutical (e.g., vaccination) and non-pharmaceutical (e.g., lockdowns, social distancing) interventions. Each agent transitions among susceptible, exposed, infectious, and recovered states. We adopt the default epidemiological parameters provided by the simulator. We simulate outbreaks across 24 regions, each with a population of 10,000 individuals and forward-evolve the system for 60 days. The initial number of infected individuals is randomly sampled to generate diverse epidemic trajectories. Multiple random seeds are used to construct training, validation, and test splits. To evaluate out-of-distribution generalization, we further generate an OOD setting by increasing the population size to 50,000 and shifting the initial epidemic conditions, introducing systematic scale and regime changes while preserving the underlying dynamics.

We construct lung cancer time series datasets (Geng et al., 2017) using a mechanistic pharmacokineticpharmacodynamic (PKPD) model (Bica et al., 2020) that captures tumor progression under therapeutic interventions. The simulator models tumor volume evolution through ordinary differential equations that integrate intrinsic growth dynamics with treatment-induced effects. Specifically, tumor growth follows a nonlinear proliferation process constrained by a carrying capacity, while chemotherapy is modeled through a drug concentrationdependent cytotoxic term:

$$\frac{dx(t)}{dt} = \rho \log\left(\frac{K}{x(t)}\right) - \beta_c\, C(t), \tag{6}$$

where $x(t)$ denotes tumor volume at time t, the first term represents nonlinear tumor proliferation limited by carrying capacity K, and the second term captures chemotherapy-induced cytotoxicity governed by drug concentration $C(t)$. Based on this PKPD framework, we generate a controlled dataset variant that captures tumor dynamics under chemotherapy only (Lung Cancer with Chemo.). Patient-specific parameters are sampled from clinically motivated ranges, and tumor trajectories are simulated over time under corresponding intervention protocols. This setup yields realistic and interpretable tumor progression patterns while introducing structured distribution shifts driven by treatment actions.

E.2 Metrics

We evaluate forecasting performance using the following widely adopted time-series metrics.

Mean Absolute Error (MAE). MAE measures the average absolute deviation between predicted and observed values, reflecting overall forecasting accuracy without emphasizing extreme errors:

$$\text{MAE} = \frac{1}{T}\sum_{t=1}^{T}|y_t - \hat{y}_t|\,. \tag{7}$$

Root Mean Squared Error (RMSE). RMSE assigns larger penalties to large deviations and is commonly used to assess sensitivity to extreme prediction errors in temporal dynamics:

$$\text{RMSE} = \sqrt{\frac{1}{T}\sum_{t=1}^{T}(y_t - \hat{y}_t)^2}. \tag{8}$$

For time series classification tasks, we evaluate predictive performance using the F1-score and the Area Under the Receiver Operating Characteristic Curve (AUC).

F1-score. The F1-score measures the harmonic mean of precision and recall, balancing false positives and false negatives in imbalanced temporal event prediction settings:

$$\text{Precision} = \frac{\text{TP}}{\text{TP}+\text{FP}}, \quad \text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}, \tag{9}$$

$$\text{F1} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}, \tag{10}$$

where TP, FP, and FN denote true positives, false positives, and false negatives aggregated over temporal sequences.

AUC. The Area Under the ROC Curve (AUC) measures the models ability to discriminate between positive and negative classes by summarizing the trade-off between true positive rates and false positive rates. Formally, it corresponds to the probability that a randomly selected positive sequence receives a higher prediction score than a randomly selected negative sequence:

$$\text{AUC} = \int_0^1 \text{TPR}(s)\, d\text{FPR}(s), \tag{11}$$

where $\text{TPR}(s)$ and $\text{FPR}(s)$ denote the true positive rate and false positive rate, respectively.

E.3 Implementation Details

For LLM-based baselines, we provide the same domain-knowledge prompt whenever the method supports textual conditioning. While these methods primarily incorporate such information as auxiliary context for representation alignment or prediction, TimeCoder operationalizes the same knowledge as constraints and guidance for executable model-structure synthesis and refinement. For LLM-based components, we use Qwen3-32B-Instruct as the backbone model unless otherwise specified. All experiments are implemented in Python with PyTorch. All TimeCoder refinement experiments are conducted on a single NVIDIA H200 NVL GPU. We fix the prompt template and LLM decoding configuration for all runs. Results are reported over five independent runs with different random seeds. The final model is selected according to validation performance from generated hypotheses, without manual selection. For the held-out intervention experiment on VitalDB, we select propofol as the common held-out drug and remifentanil as the rare held-out drug based on training-set medication frequencies. This design evaluates two representative regimes: removing a high-frequency intervention and testing robustness to a sparse intervention. Representative medication distribution can be found in Table 3.

E.4 Main Results

E.4.1 Classification Results

Table 4 reports the classification performance on the VitalDB dataset for AKI prediction. Among all baseline models, traditional time series forecasting architectures such as DLinear, TimesNet,

Table 4: Classification Performance Comparison on VitalDB for AKI Prediction.

Model	AUROC	F1-score
DLinear (Zeng et al., 2023)	0.86 ± 0.01	0.47 ± 0.03
TimesNet (Wu et al., 2023)	0.87 ± 0.01	0.50 ± 0.03
Autoformer (Wu et al., 2021)	0.85 ± 0.01	0.45 ± 0.04
Crossformer (Zhang & Yan, 2023)	0.88 ± 0.01	0.53 ± 0.03
PatchTST (Nie et al., 2023b)	0.87 ± 0.01	0.52 ± 0.03
iTransformer (Liu et al., 2024b)	0.88 ± 0.01	0.54 ± 0.03
Time-LLM (Jin et al., 2024a)	0.89 ± 0.01	0.58 ± 0.02
OFA (Zhou et al., 2023)	0.88 ± 0.01	0.55 ± 0.03
FSCA (Hu et al., 2025b)	0.90 ± 0.01	0.60 ± 0.02
TimeXL (Jiang et al., 2025)	0.90 ± 0.01	0.61 ± 0.02
TimeCoder	$\mathbf{0.92 \pm 0.01}$	$\mathbf{0.67 \pm 0.02}$

Autoformer, and PatchTST achieve AUROC values ranging from 0.85 to 0.88, with relatively limited F1-scores, indicating suboptimal practical detection performance. Recent LLM-enhanced and context-aware models, including Time-LLM, OFA, FSCA, and TimeXL, further improve both AUROC and F1-score, demonstrating the benefits of incorporating large-model reasoning or contextual representations for clinical time series understanding. Notably, TimeCoder consistently outperforms all competing methods, achieving the highest AUROC of 0.92 and F1-score of 0.67, representing a clear margin over the strongest baselines such as FSCA and TimeXL. This substantial improvement in F1-score suggests that TimeCoder not only enhances discriminative capability but also yields more reliable decision boundaries for clinically actionable AKI detection. These results highlight the effectiveness of TimeCoder in capturing complex temporal dynamics and integrating structured inductive priors, enabling robust and practical clinical risk prediction from high-frequency perioperative vital signs.

E.5 Further Analysis

E.5.1 Influence of Memory

Model Configuration	SF^2Bench	Lung Cancer	VitalDB	COVID-19
TimeCoder	0.328 ± 0.002	0.092 ± 0.002	0.445 ± 0.042	0.009 ± 0.001
w/o memory	0.351 ± 0.004	0.112 ± 0.005	0.518 ± 0.050	0.014 ± 0.001
with top-1	0.344 ± 0.002	0.107 ± 0.003	0.482 ± 0.045	0.012 ± 0.001
with top-2	0.338 ± 0.002	0.100 ± 0.002	0.473 ± 0.043	0.012 ± 0.001
with top-3	0.335 ± 0.002	0.098 ± 0.002	0.460 ± 0.041	0.011 ± 0.001
with top-4	0.330 ± 0.002	0.093 ± 0.002	0.455 ± 0.040	0.010 ± 0.001

Table 5: Ablation study on the top-k memory of best historical models and corresponding feedback.

E.5.2 Ablation Study on the Influence of Different Base Models.

Table 6: Ablation Study on the Influence of Different Base Models.

	Qwen3-coder-flash		Gemini-2.5-pro		Qwen3-32B	
	RMSE	MAE	RMSE	MAE	RMSE	MAE
SF^2Bench	0.328 ± 0.002	0.184 ± 0.001	0.400 ± 0.017	0.209 ± 0.015	0.364 ± 0.015	0.185 ± 0.014
Lung Cancer	0.092 ± 0.002	0.061 ± 0.002	0.132 ± 0.000	0.095 ± 0.000	0.120 ± 0.009	0.084 ± 0.005
VitalDB	0.445 ± 0.042	0.298 ± 0.078	0.524 ± 0.053	0.374 ± 0.097	0.476 ± 0.048	0.331 ± 0.086
COVID-19	0.009 ± 0.001	0.006 ± 0.001	0.012 ± 0.000	0.009 ± 0.000	0.042 ± 0.008	0.035 ± 0.008

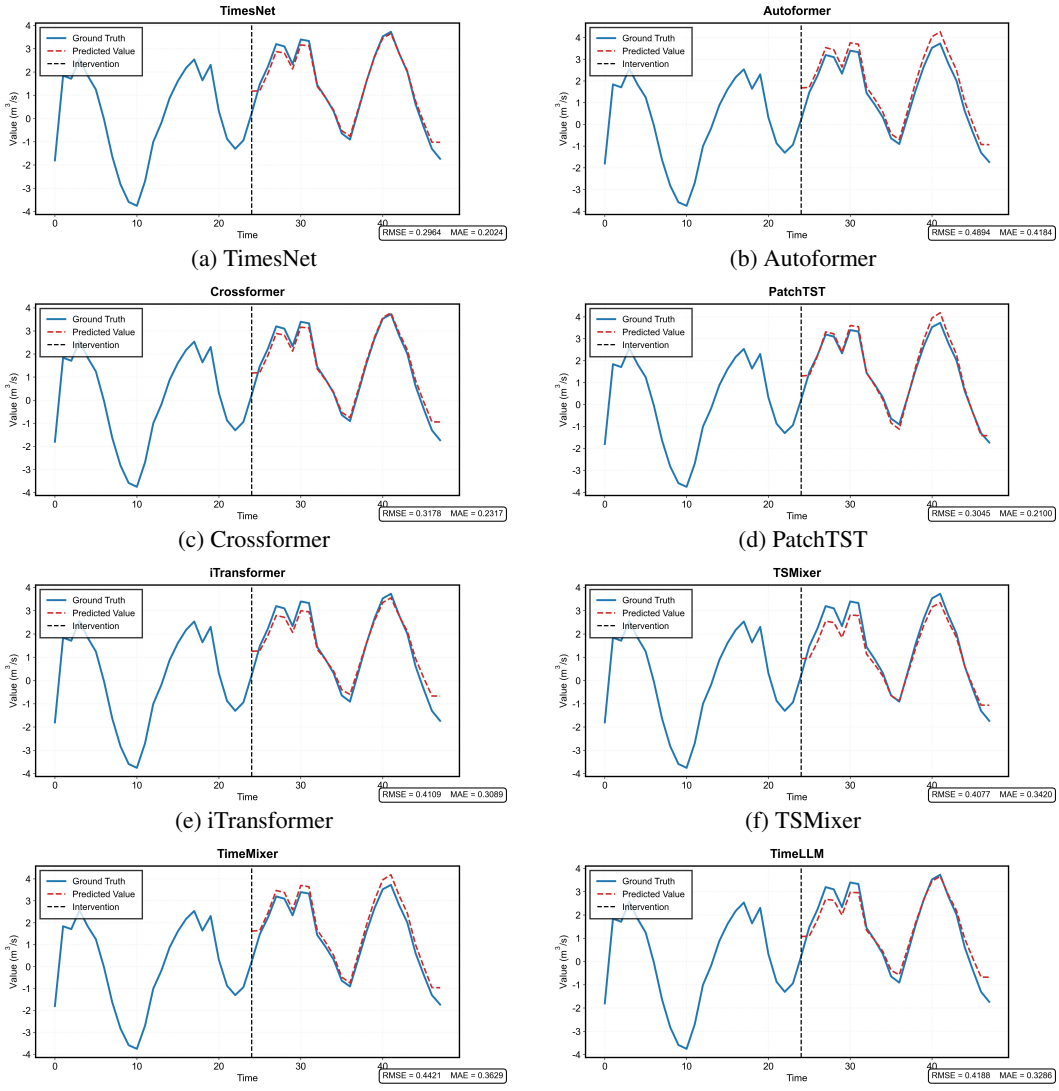
E.5.3 Ablation Study on the Components of TimeCoder.

E.5.4 Prompt Description

An example of the prompt is provided in the Figure 13.

E.6 Visualization

Qualitative Analysis. Figure 4 provides a qualitative comparison on the SF^2Bench dataset, where forecasting starts from the intervention point marked by the vertical dashed line. The post-intervention trajectory contains several challenging local structures: an immediate rise after the intervention, two nearby peaks around the early prediction horizon, a decline toward a trough, and a later high-amplitude rebound followed by a sharp drop. These patterns allow us to examine whether a model captures not only the global oscillatory trend, but also the phase, amplitude, and turning-point structure of the intervention response.



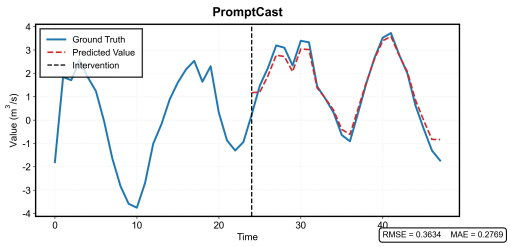
(a) TimesNet (b) Autoformer

(c) Crossformer (d) PatchTST

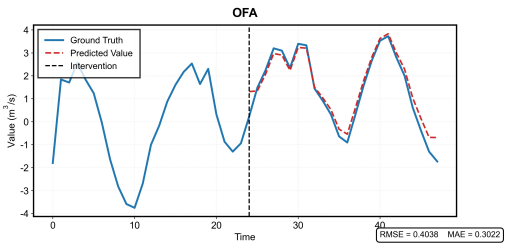
(e) iTransformer (f) TSMixer

(g) TimeMixer (h) Time-LLM

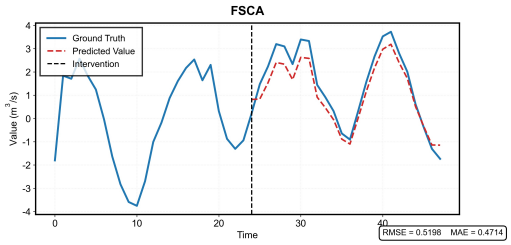
Figure 4: Forecasting visualization on the SF^{2}Bench dataset.



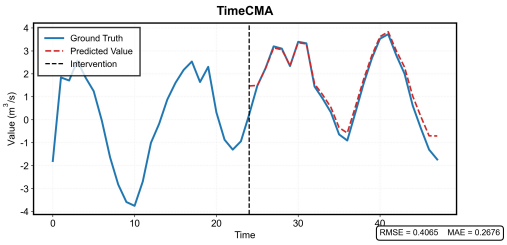
(i) PromptCast



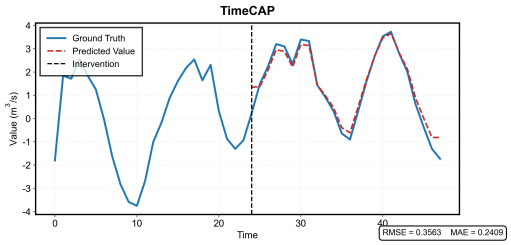
(j) OFA



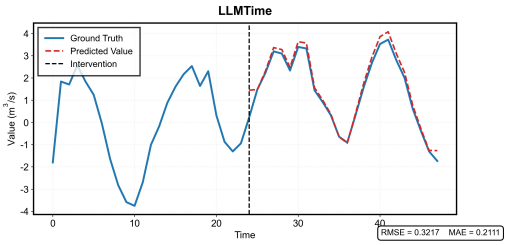
(k) FSCA



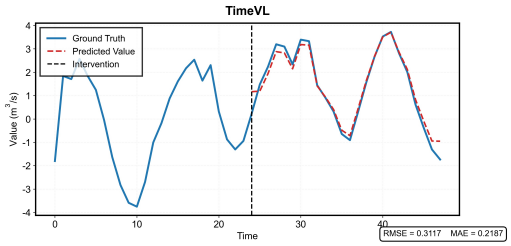
(l) TimeCMA



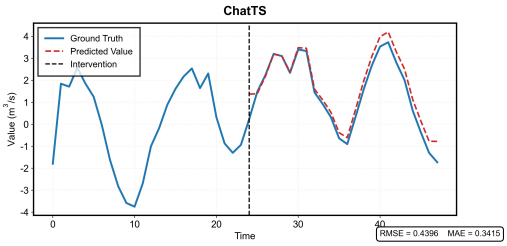
(m) TimeCAP



(n) LLMTime



(o) TimeVLM



(p) ChatTS

Figure 4: Forecasting visualization on the SF^{2}Bench dataset, continued.

Table 7: Ablation Study on the Component of TimeCoder.

	SF^2Bench	Lung Cancer	VitalDB	COVID-19
	RMSE	RMSE	RMSE	RMSE
TimeCoder	0.328 ± 0.002	0.092 ± 0.002	0.445 ± 0.042	0.009 ± 0.001
w/o Statistical Description	0.361 ± 0.015	0.098 ± 0.003	0.472 ± 0.048	0.011 ± 0.001
w/o Domain Description	0.365 ± 0.003	0.105 ± 0.004	0.493 ± 0.045	0.012 ± 0.001
with only white-box instruction	0.372 ± 0.004	0.110 ± 0.005	0.515 ± 0.050	0.014 ± 0.001

We observe that several baselines can recover the coarse post-intervention oscillation, but often introduce local structural errors. For example, iTransformer follows the general rise-and-fall pattern, but underestimates the early peaks and shows a clear mismatch near the final descending segment, where the prediction remains higher than the ground truth. For example, PatchTST better captures the overall phase and achieves a lower visual error, but it overshoots the later high-amplitude peak and slightly exaggerates the following decline. TimeCAP and TimeCMA also align with the main oscillatory structure, yet both show amplitude bias around the early double-peak region and the late descending region. In contrast, TimeCoder provides the closest visual alignment with the ground-truth trajectory. It accurately follows the immediate post-intervention rise, preserves the double-peak structure, tracks the trough after the first decline, and closely matches the late high-amplitude rebound and subsequent drop.

E.7 Computational Cost and Convergence Behavior Analysis

Token Consumption Analysis are shown in Figure 5.

E.8 Case Study

E.8.1 Analysis of a Full Iterative Refinement Behavior.

To further understand how TimeCoder progressively improves temporal modeling performance, we provide a detailed analysis of the code evolution process across multiple generations on the COVID-19 dataset. Fine-grained code-generated results are shown in Figure 15 to Figure 43. Beyond the high-level trends shown in Fig. 2, we observe that the refinement trajectory naturally decomposes into three distinct stages characterized by increasingly expressive model structures. Stage I: Structured White-box Dynamics Initialization. In the early iterations (Generations 1–4), TimeCoder consistently introduces a mechanistic SEIR-style dynamical system with explicit intervention-modulated transmission rates. This structured prior establishes a causal backbone that encodes epidemiological constraints such as population conservation and compartmental transitions. As reflected by the sharp RMSE drop in this stage, incorporating domain-consistent dynamics enables the model to rapidly capture the dominant temporal patterns, outperforming unstructured neural architectures. Stage II: Hybridization with Latent Memory and Neural Residuals. During the middle phase (Generations 5–10), TimeCoder augments the white-box SEIR core with latent memory states and lightweight neural residual networks. The introduced memory dynamics allow the model to encode non-Markovian temporal dependencies, while the residual networks compensate for systematic deviations from idealized compartmental assumptions. Stage III: Numerical and Architectural Refinement. In the final stage (Generations 11 onward), TimeCoder primarily focuses on improving numerical solvers and refining network architectures, including higher-order integration schemes and more stable normalization strategies. Rather than altering the underlying dynamical structure, these refinements enhance optimization stability and fine-grained temporal precision.

E.8.2 Failure Rate and Failure Type Analysis

Table 8: Verification and repair statistics of generated code.

Dataset	**First-pass valid rate**	**Final valid rate after repair**	**Average repair attempts**	**Main failure types**
COVID-19	0.3	0.95	1.4	Shape Mismatch
Lung Cancer	0.4	0.92	1.1	Numerical instability

Failure Type Analysis. Failure Rate Analysis. Table 8 summarizes the verification and repair statistics of LLM-generated forecasting programs on the COVID-19 and Lung Cancer datasets. For each dataset, we conduct 10 independent code-generation trials. In each trial, TimeCoder first generates an initial executable forecasting program and then performs validation. If the initial program fails validation, TimeCoder is allowed to conduct up to $K = 10$ repair attempts using the validation feedback. We compute the first-pass valid rate as the fraction of initially generated programs that

pass validation without repair:

$$\text{First-pass valid rate} = \frac{\#\text{initial programs passing validation}}{\#\text{all initial generated programs}}.$$

We compute the final valid rate after repair as the fraction of programs that pass validation either directly or after repair:

$$\text{Final valid rate} = \frac{\#\text{programs passing validation after at most } K \text{ repairs}}{\#\text{all initial generated programs}}.$$

The average repair attempts are computed over programs that fail initially but are eventually repaired successfully:

$$\text{Average repair attempts} = \frac{\sum_i \#\text{repair attempts used for successfully repaired program } i}{\#\text{successfully repaired initially failed programs}}.$$

The first-pass valid rate is relatively low on both datasets, showing that directly generated forecasting code may contain implementation errors or unstable dynamical formulations. However, the final valid rate increases substantially after repair, reaching 0.95 on COVID-19 and 0.92 on Lung Cancer. This indicates that the generated-code failures are often repairable rather than fundamental. The average number of repair attempts is also small, requiring only 1.4 attempts on COVID-19 and 1.1 attempts on Lung Cancer, suggesting that the validation feedback usually provides sufficient information for efficient correction. These results show that the verification-and-repair loop is an important component for converting initially imperfect LLM-generated model structures into executable forecasting programs.

Failure Type Analysis. Table 9 further analyzes the types of first-pass failures. For this analysis, we only consider programs that fail at the first validation step, rather than all intermediate failures during the repair process. This avoids mixing the intrinsic failure patterns of initial LLM-generated code with errors introduced or modified during repair. For each initially failed program, we assign one primary failure type based on the verifier output. The ratio of each failure type is computed as

$$\text{Failure type ratio} = \frac{\#\text{initially failed programs assigned to this failure type}}{\#\text{all initially failed programs}}.$$

Thus, the denominator is the number of first-pass failed programs, and the numerator is the number of those failures belonging to each primary error category.

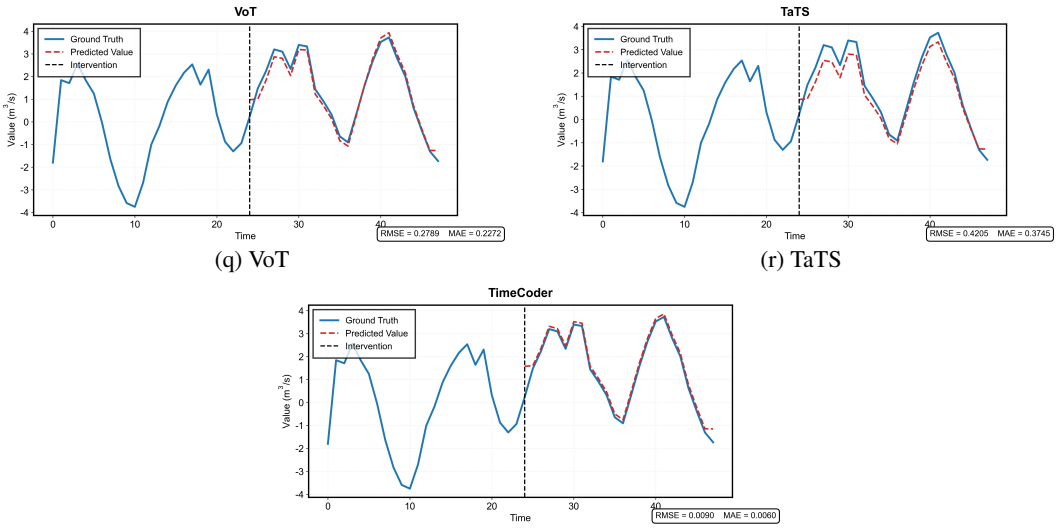
The failure distribution is dataset-dependent. On COVID-19, the most frequent first-pass failures are undefined variables or functions and shape mismatches, accounting for 0.40 and 0.35 of initially failed programs, respectively. This suggests that, for epidemic dynamics, the LLM often proposes plausible high-level modeling structures but may fail to maintain a consistent implementation interface, such as returning derivatives with incompatible shapes or using intermediate variables that are not properly defined. In contrast, Lung Cancer exhibits a higher proportion of numerical instability, accounting for 0.45 of initially failed programs. This is consistent with the nonlinear treatment-response setting, where tumor growth, chemotherapy effects, and radiotherapy response can easily produce invalid values during rollout if growth or treatment terms are not properly bounded.

Overall, the failure analysis shows that LLM-generated forecasting code fails in two main ways: implementation-level failures, such as shape mismatch and undefined variables, and dynamics-level failures, such as numerical instability. Syntax errors and invalid output interfaces are relatively rare, indicating that the LLM usually follows the required code format. However, ensuring shape consistency, variable grounding, and stable dynamical rollouts remains challenging. These findings justify the need for explicit verification and repair: the verifier identifies whether a generated program fails due to code-level inconsistency or unstable temporal dynamics, and the repair step uses this feedback to refine the generated forecasting model.

Table 9: Failure types Analysis.

Dataset	Syntax error	Shape mismatch	Numerical instability	Undefined variable / function	Invalid output interface
COVID-19	0.1	0.35	0.10	0.4	0.05
Lung Cancer	0.1	0.15	0.40	0.3	0.05

E.8.3 Execution-Guided Program Repair Example



(q) VoT (r) TaTS

(s) TimeCoder

Figure 4: Forecasting visualization on the SF^{2}Bench dataset, continued.

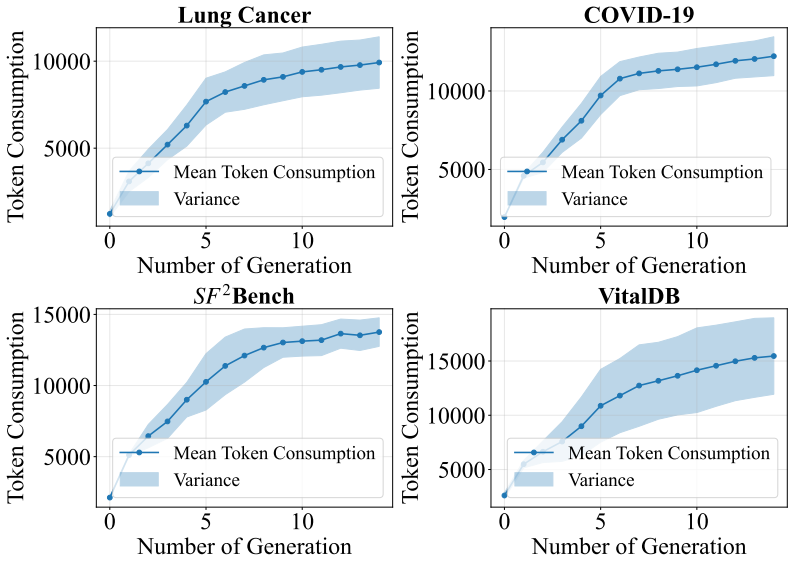


Figure 5: Token consumption analysis across refinement generations.

A Generated Error Code on the Lung Cancer Dataset.

```
def d_state__dt(state, action=None):
    import numpy as np

    dx_dt = np.zeros_like(state)

    tumor = state[0]
    chemo_conc = state[1]
    chemo_dose = state[2]
    radio_dose = state[3]

    r_growth = 0.03
    K_carrying = 1.0

    k_chemo_kill = 2.5
    IC50_chemo = 0.4
    hill_coeff = 2.0

    alpha_rad = 0.3
    beta_rad_lq = 0.05

    k_absorption = 0.8
    k_elimination = 0.15

    V_min = 1e-4
    V_eff = max(tumor, V_min)

    if V_eff < K_carrying:
        log_term = np.log(K_carrying / V_eff)
        growth_rate = r_growth * tumor * log_term
    else:
        growth_rate = 0

    if chemo_conc > 1e-6:
        hill_numerator = chemo_conc ** hill_coeff
        hill_denominator = IC50_chemo ** hill_coeff + hill_numerator
        hill_effect = hill_numerator / hill_denominator
        chemo_kill_rate = k_chemo_kill * hill_effect * tumor
    else:
        chemo_kill_rate = 0.0

    if radio_dose > 1e-6:
        lq_dose = alpha_rad * radio_dose + beta_rad_lq * radio_dose ** 2
        survival_fraction = np.exp(-lq_dose)
        radio_kill_rate = (1 - survival_fraction) * tumor
    else:
        radio_kill_rate = 0.0

    dchemo_conc_dt = k_absorption * chemo_dose - k_elimination * chemo_conc
    dchemo_dose_dt = -0.2 * chemo_dose
    dradio_dose_dt = -0.25 * radio_dose

    dtumor_dt = growth_rate - chemo_kill_rate - radio_kill_rate

    dx_dt[0] = dtumor_dt
    dx_dt[1] = dchemo_conc_dt
    dx_dt[2] = dchemo_dose_dt
    dx_dt[3] = dradio_dose_dt

    return dx_dt
```

Figure 6: A Failure-Repair Example Analysis.

A Repaired Code on the Lung Cancer Dataset.

```
def d_state__dt(state, action=None):
    import numpy as np

    dx_dt = np.zeros_like(state)

    tumor = state[0]
    chemo_conc = state[1]
    chemo_dose = state[2]
    radio_dose = state[3]

    r_growth = 0.015
    K_carrying = 1.0

    k_chemo_kill = 0.15
    IC50_chemo = 0.4
    hill_coeff = 2.0

    alpha_rad = 0.3
```

```
    beta_rad_lq = 0.05

    k_absorption = 0.8
    k_elimination = 0.15

    dose_decay_chemo = 0.2
    dose_decay_radio = 0.25

    V_min = 1e-4
    V_eff = max(tumor, V_min)

    if V_eff < K_carrying:
        log_term = np.log(K_carrying / V_eff)
        growth_rate = r_growth * tumor * log_term
    else:
        growth_rate = 0

    max_growth = 0.05 * (1.0 - tumor)
    growth_rate = min(growth_rate, max_growth)
    growth_rate = max(growth_rate, -0.1 * tumor)

    if chemo_conc > 1e-6:
        hill_numerator = (chemo_conc ** hill_coeff)
        hill_denominator = (IC50_chemo ** hill_coeff) + hill_numerator
        hill_effect = hill_numerator / hill_denominator
        chemo_kill_rate = k_chemo_kill * hill_effect * tumor
    else:
        chemo_kill_rate = 0.0

    max_chemo_kill = 0.3 * tumor
    chemo_kill_rate = min(chemo_kill_rate, max_chemo_kill)

    if radio_dose > 1e-6:
        lq_dose = alpha_rad * radio_dose + beta_rad_lq * (radio_dose ** 2)
        survival_fraction = np.exp(-lq_dose)
        survival_fraction = np.clip(survival_fraction, 0.0, 1.0)
        radio_kill_rate = (1.0 - survival_fraction) * tumor
    else:
        radio_kill_rate = 0.0
```

Figure 7: A Failure-Repair Example Analysis.

A Repaired Code on the Lung Cancer Dataset (continued).

```
    max_radio_kill = 0.25 * tumor
    radio_kill_rate = min(radio_kill_rate, max_radio_kill)

    dchemo_conc_dt = k_absorption * chemo_dose - k_elimination * chemo_conc

    if chemo_conc <= 1e-6 and dchemo_conc_dt < 0:
        dchemo_conc_dt = 0

    dchemo_dose_dt = -dose_decay_chemo * chemo_dose
    dradio_dose_dt = -dose_decay_radio * radio_dose

    dtumor_dt = growth_rate - chemo_kill_rate - radio_kill_rate

    dx_dt[0] = dtumor_dt
    dx_dt[1] = dchemo_conc_dt
    dx_dt[2] = dchemo_dose_dt
    dx_dt[3] = dradio_dose_dt

    if tumor <= 1e-4 and dtumor_dt < 0:
        dx_dt[0] = 0

    if chemo_conc <= 1e-6 and dchemo_conc_dt < 0:
        dx_dt[1] = 0

    dx_dt[2] = min(dx_dt[2], 0) if chemo_dose <= 1e-6 else dx_dt[2]
    dx_dt[3] = min(dx_dt[3], 0) if radio_dose <= 1e-6 else dx_dt[3]

    dx_dt = np.clip(dx_dt, -0.3, 0.3)

    boundary_margin = 0.05

    if tumor < boundary_margin and dtumor_dt < 0:
        damping = tumor / boundary_margin
        dx_dt[0] *= damping

    if tumor > (1.0 - boundary_margin) and dtumor_dt > 0:
        damping = (1.0 - tumor) / boundary_margin
        dx_dt[0] *= damping

    return dx_dt
```

Figure 8: A Failure-Repair Example Analysis. Continued from Figure 7.

Failure-Repair Analysis. Figures 6 to 8 illustrate a representative failure-repair case on the Lung Cancer dataset. In the initial generated code, although the function returns an output with the expected shape, the validation process detects numerical instability during prediction, producing runtime warnings such as invalid values in multiplication. This suggests that the generated dynamical system contains unstable update rules or insufficient boundary constraints, which can lead to invalid intermediate states during rollout.

After receiving the validation feedback, TimeCoder diagnoses the failure and refines the generated code by introducing more conservative dynamics and explicit numerical safeguards. The repaired version reduces the tumor growth and treatment-effect magnitudes, clips survival fractions and derivative values, and adds boundary-aware damping to prevent invalid state updates. This example demonstrates that TimeCoder can use execution feedback not only to identify implementation-level failures, but also to revise the underlying generated forecasting dynamics into a more stable and physically plausible form.

E.8.4 Diagnostic-Guided Code Refinement Example

Diagnosis-Guided Program Repair: Erroneous Code on the SF^{2}Bench Dataset

```
def d_state__dt(state, action=None):
    import numpy as np

    dx_dt = np.zeros_like(state)

    decay_rate = 0.1
    tide_amplitude = 2.0
    tide_period = 12.4
    rain_addition = 0.01
    pump_extraction_rate = 0.25
    gate_effect_strength = 0.3

    for i in range(7):
        dx_dt[i] = -decay_rate * state[i]

        if i > 0:
            upstream_flow = np.clip(
                gate_effect_strength * state[8+i-1] * (state[i-1] - state[i]),
                -1.0, 1.0
            )
            dx_dt[i] += upstream_flow

        if i < 6:
            downstream_backflow = np.clip(
                gate_effect_strength * state[8+i] * (state[i+1] - state[i]),
                -1.0, 1.0
            )
            dx_dt[i] += downstream_backflow

        tide_influence = tide_amplitude * np.sin((2 * np.pi / tide_period) * state[7])
        dx_dt[i] += tide_influence * (i / 6.0)

        dx_dt[i] += rain_addition * state[15]

        if i < 3:
            dx_dt[i] -= pump_extraction_rate * state[13] * state[i]
        elif i >= 4:
            dx_dt[i] -= pump_extraction_rate * state[14] * state[i]

    dx_dt[7] = -decay_rate * state[7] + np.sin((2 * np.pi / tide_period) * state[7])

    for i in range(8, 13):
        gate_idx = i - 8
        upstream_water = state[gate_idx]
        downstream_water = state[gate_idx + 1]

        if upstream_water > downstream_water:
            dx_dt[i] = gate_effect_strength * (upstream_water - downstream_water) - decay_rate * state[i]
        else:
            dx_dt[i] = -decay_rate * state[i]

        dx_dt[i] = np.clip(dx_dt[i], -0.1, 0.1)

    for i in range(13, 15):
        if i == 13:
            affected_stations = state[0:3]
        else:
            affected_stations = state[4:7]

        dx_dt[i] = -decay_rate * state[i] + 0.2 * np.mean(affected_stations)

    dx_dt[15] = -decay_rate * state[15] + 0.1 * np.random.randn()
```

```
    assert dx_dt.shape == state.shape

    return dx_dt
```

Figure 9: Execution-guided program repair example on the SF^{2}Bench dataset. The initial program contains an unstable stochastic update term and violates the deterministic execution protocol. TimeCoder uses execution feedback to repair the code into an executable temporal model in the subsequent iteration.

Diagnostic-Guided Structural Refinement Example on the SF^{2}Bench Dataset

```
def d_state__dt(state, action=None):
    import numpy as np

    dx_dt = np.zeros_like(state)

    decay_rate = 0.1
    tide_amplitude = 0.25
    tide_period = 12.4
    rain_addition = 0.01
    pump_extraction_rate = 0.35
    gate_effect_strength = 0.5

    for i in range(7):
        dx_dt[i] = -decay_rate * state[i]

        if i > 0:
            upstream_flow = np.clip(
                gate_effect_strength * state[8+i-1] * (state[i-1] - state[i]),
                -1.0, 1.0
            )
            dx_dt[i] += upstream_flow

        if i < 6:
            downstream_backflow = np.clip(
                gate_effect_strength * state[8+i] * (state[i+1] - state[i]),
                -1.0, 1.0
            )
            dx_dt[i] += downstream_backflow

        tide_influence = tide_amplitude * np.sin((2 * np.pi / tide_period) * state[7])
        dx_dt[i] += tide_influence * (i / 6.0)

        dx_dt[i] += rain_addition * state[15]

        if i < 3:
            dx_dt[i] -= pump_extraction_rate * state[13] * state[i]
        elif i >= 4:
            dx_dt[i] -= pump_extraction_rate * state[14] * state[i]

    dx_dt[7] = -decay_rate * state[7] + np.sin((2 * np.pi / tide_period) * state[7])

    for i in range(8, 13):
        gate_idx = i - 8
        upstream_water = state[gate_idx]
        downstream_water = state[gate_idx + 1]

        if upstream_water > downstream_water:
            dx_dt[i] = gate_effect_strength * (upstream_water - downstream_water) - decay_rate * state[i]
        else:
            dx_dt[i] = -decay_rate * state[i]

        dx_dt[i] = np.clip(dx_dt[i], -0.1, 0.1)

    for i in range(13, 15):
        if i == 13:
            affected_stations = state[0:3]
        else:
            affected_stations = state[4:7]

        dx_dt[i] = -decay_rate * state[i] + 0.2 * np.mean(affected_stations)

    dx_dt[15] = -decay_rate * state[15]

    dx_dt = np.clip(dx_dt, -1.0, 1.0)

    return dx_dt
```

Figure 10: Diagnostic-guided structural refinement example on the SF^{2}Bench dataset. After receiving statistical diagnostic feedback, TimeCoder refines the executable temporal model by incorporating hydrologically meaningful components, including decay dynamics, gate-induced upstream and downstream flow, tidal influence, rainfall forcing, pump extraction, and numerical stability constraints.

Diagnostic Feedback Prompt for Structural Refinement on the SF^{2}Bench Dataset (Part I)

```
You are an expert in time series modeling and dynamical systems.
Generate ONLY syntactically correct Python code for modeling time-series data.

CRITICAL INSTRUCTIONS:
- ONLY generate Python code; no natural language explanations.
- Define a function d_state__dt(state, action=None).
- Use np.zeros_like(state); dx_dt must have the same shape as state.
- Include import numpy as np.
- ABSOLUTELY NO NaN/Inf values.
- ABSOLUTELY NO random functions; the ODE right-hand side must be deterministic.

STATE VECTOR (16 dimensions):
- state[0:7]: WATER STATIONS
- state[7]: TIDE
- state[8:13]: GATES
- state[13:15]: PUMPS
- state[15]: RAIN

PREVIOUS ITERATION FEEDBACK
================================================================================
Performance Metrics:
- MSE: 0.3847
- RMSE: 0.6202
- MAE: 0.4733
- Valid predictions: 7494 / 8737 (85.8%)
- Failed samples: 1243 / 8737 (14.2%)
- Failure reason: RuntimeWarning: invalid value encountered in multiply

STATISTICAL DESCRIPTOR ANALYSIS
================================================================================

Descriptor 1: TEMPORAL LAG DETECTION
- Peak correlation lag: +2 to +3 steps
- Max cross-correlation: 0.72
- Diagnosis: predictions lag behind ground truth.
- Root cause: weak gate coupling and excessive tidal forcing.
- Impact: HIGH

Descriptor 2: TREND ANALYSIS
- Slope ratio: 1.34
- Intercept: +0.087
- Diagnosis: predictions over-amplify trends and show positive bias.
- Root cause: tide_amplitude=2.0 is too large for normalized data.
- Impact: HIGH

Descriptor 3: PERIODICITY DETECTION
- Sequences with residual periodicity: 78%
- Dominant residual period: 6.2 steps
- Diagnosis: artificial high-frequency oscillations.
- Root cause: excessive tide amplitude and random noise in rain dynamics.
- Impact: CRITICAL

Descriptor 4: ABRUPT CHANGE DETECTION
- Mean |delta| ratio: 2.87
- Max single-step jump: 1.84
- Outlier abrupt-change ratio: 23%
- Diagnosis: predictions are too spiky and physically implausible.
- Root cause: np.random.randn() injects uncorrelated noise.
- Impact: HIGH

Descriptor 5: SYSTEMATIC BIAS QUANTIFICATION
- Mean bias: +0.142
- Positive bias in 89% of samples
- Significantly biased sequences: 82%
- Diagnosis: systematic over-prediction.
- Root cause: over-forcing from tide and insufficient pump extraction.
- Impact: CRITICAL
```

Figure 11: Diagnostic feedback prompt for structural refinement on the SF^{2}Bench dataset, Part I. This part provides the task constraints, state-vector definition, validation metrics, and five statistical descriptor analyses.

Diagnostic Feedback Prompt for Structural Refinement on the SF^{2}Bench Dataset (Part II)

```
AGGREGATED STATISTICAL DIAGNOSIS
================================================================================
+--------------------------+----------+---------+------------------------------+
| Descriptor               | Severity | Value   | Root Cause                   |
+--------------------------+----------+---------+------------------------------+
| Temporal Lag             | HIGH     | lag +2  | weak gate coupling           |
| Trend Analysis           | HIGH     | 1.34    | excessive tide amplitude     |
```

```
| Periodicity Detection    | CRITICAL | 78%     | tide forcing + randomness   |
| Abrupt Change            | HIGH     | 2.87x   | stochastic rain update      |
| Systematic Bias          | CRITICAL | +0.142  | over-forcing, under-pumping |
+--------------------------+----------+---------+-----------------------------+

Failure Classification:
- Syntax error: 0.00
- Shape mismatch: 0.00
- Numerical instability: 0.45
- Undefined variable/function: 0.05
- Invalid output interface: 0.00
- Physical violation: 0.50

REPAIR SUGGESTIONS
================================================================================
1. Reduce tide_amplitude from 2.0 to 0.25.
   This addresses trend amplification, spurious periodicity, and positive bias.

2. Remove np.random.randn() from the rain dynamics.
   Replace:
       dx_dt[15] = -decay_rate * state[15] + 0.1 * np.random.randn()
   With:
       dx_dt[15] = -decay_rate * state[15]
   The ODE right-hand side must be deterministic.

3. Increase gate_effect_strength from 0.3 to 0.5.
   This reduces the temporal lag by strengthening upstream-downstream propagation.

4. Increase pump_extraction_rate from 0.25 to 0.35.
   This helps reduce systematic over-prediction.

5. Add global clipping:
       dx_dt = np.clip(dx_dt, -1.0, 1.0)
   This prevents extreme derivative values and improves numerical stability.

Overall diagnosis:
The code executes but produces statistically invalid predictions.
Primary issues are excessive tidal forcing, stochastic updates, weak gate coupling,
and insufficient pump extraction.

Generate d_state__dt for iteration 1/10.
Address all statistical descriptor failures and ensure deterministic, stable output.
```

Figure 12: Diagnostic feedback prompt for structural refinement on the SF^{2}Bench dataset, Part II. This part summarizes the aggregated diagnosis table, failure classification, and prioritized repair suggestions used to guide TimeCoder's refinement.

Figure 9 to Figure 10 show a diagnostic-guided structural refinement case on the SF^{2}Bench dataset. The diagnostic analysis based on statistical descriptors is shown in Figure 11 and 12. Instead of relying only on a scalar validation loss, TimeCoder summarizes prediction errors using five statistical descriptors, including temporal lag, trend mismatch, residual periodicity, abrupt changes, and systematic bias. The diagnostic prompt in Figure 11 and 12 identifies several concrete failure causes, such as excessive tidal forcing, stochastic rain updates, weak gate coupling, and insufficient pump extraction. Based on this feedback, the refined program in Figure 10 modifies the temporal mechanism by reducing the tide amplitude, removing stochastic terms, strengthening gate-induced flow propagation, increasing pump extraction, and adding global numerical clipping. These changes directly correspond to the diagnosed error modes and make the generated model more consistent with both the observed trajectories and domain-specific hydrological dynamics.

An Example of the System Prompt.

```
You are an expert in clinical physiological dynamics, time-series modeling.

We are modeling perioperative physiological time series from the VitalDB dataset, consisting of
multivariate vital signs together with intervention variables representing vasoactive medications:

- Vital signs: heart rate, arterial blood pressure, oxygen saturation, respiratory signals
- Interventions: propofol, ephedrine, sugammadex, epinephrine, tranexamic acid, ondansetron,

hydromorphone, oxytocin, nitroglycerin, cisatracurium, fentanyl, dexamethasone, norepinephrine,

morphine, vasopressin, labetalol, phenylephrine, amiodarone, and remifentanil

The objective is to forecast future vital trajectories given recent history and intervention inputs.

The model follows a hybrid structure and please start from white box models:
* A mechanistic white-box ODE core capturing physiological relations
* Explicit intervention-to-dynamics modulation reflecting vasopressor effects
* Latent memory states modeling delayed and non-Markovian responses
* Neural residual networks for complex nonlinear corrections
```

```
We run an iterative refinement loop. At each generation, you are given:
1. The current model implementation (PyTorch code)
2. The current forecasting loss (RMSE)
3. The top-5 historically best performing model codes discovered so far
4. Statistical descriptors computed on the residuals between the current model predictions and
ground truth trajectories, including:

   - Trend mismatch (drift, bias)
   - Variance and volatility structure
   - Temporal autocorrelation
   - Regime change indicators
   - Frequency-domain patterns (e.g., oscillations)
   - Cross-variable dependency errors
     (e.g., BP not responding sufficiently to intervention)

These statistical descriptors summarize systematic patterns that the current model fails to capture.
```

Figure 13: An example of the system prompt.

An Example of the System Prompt (continued).

```
Your task is to:

* Analyze which temporal or physiological patterns are missing or poorly modeled
* Identify whether the deficiencies stem from:
   - intervention-response formulation
   - memory dynamics
   - neural residual expressiveness
   - numerical integration

* Propose a refined model that explicitly addresses the uncovered patterns while maintaining
interpretability.

Guidelines:

- Strengthen consistency between interventions and vital responses
- Introduce new mechanisms only if supported by residual statistics
- Do NOT collapse into a purely black-box neural model
- Prefer structured improvements over arbitrary complexity increases

Below are the inputs for this iteration:

---------------------------------
Current model code:
[INSERT CURRENT CODE]

---------------------------------
Current RMSE:
[INSERT CURRENT LOSS]

---------------------------------
Top-5 historical best models:
[INSERT TOP-5 CODE SNIPPETS]

---------------------------------
Residual statistical descriptors:
[INSERT STATISTICAL EVIDENCE SUMMARY]

---------------------------------

Based on the above information, generate a refined PyTorch implementation that is likely to achieve lower
forecasting error in the next iteration.

Return only the improved code.
```

Figure 14: An example of the system prompt. Continued from Figure 13.

Code Evolution on the COVID-19 Dataset: The 1st Iteration

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 8
        self.activation = nn.SiLU()

        self.beta = nn.Parameter(torch.tensor(0.5))
        self.gamma = nn.Parameter(torch.tensor(0.1))
        self.epsilon = nn.Parameter(torch.tensor(0.3))
        self.alpha = nn.Parameter(torch.tensor(0.01))
```

```
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )

    def forward_inner(self, x, h, u):

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_term = self.beta * S * I / N
        gamma_term = self.gamma * R

        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)

        dxdt = torch.cat([
            -beta_term + gamma_term,
            beta_term - self.alpha * E,
            self.alpha * E - self.gamma * I,
            self.gamma * I
        ], dim=1)

        dudt = -0.1 * u
        dhdt = torch.zeros((x.size(0), self.hidden_dim_h), device=x.device)

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)
        dhdt += self.fh_net(concat_input)

        return dxdt, dudt, dhdt
```

Figure 15: An Example of Code Evolution on the COVID-19 Dataset: the 1st Iteration.

Code Evolution on the COVID-19 Dataset: The 2nd Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
    super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 16
        self.activation = nn.GELU()

        self.beta = nn.Parameter(torch.tensor(0.506))
        self.gamma = nn.Parameter(torch.tensor(0.111))
        self.epsilon = nn.Parameter(torch.tensor(0.281))
        self.alpha = nn.Parameter(torch.tensor(0.00869))
        self.mu = nn.Parameter(torch.tensor(0.01))
        self.kappa = nn.Parameter(torch.tensor(0.1))
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )
```

```
        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )
        self.h_evolution_net = nn.Sequential(
            nn.Linear(state_dim + control_dim, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )

    def forward_inner(self, x, h, u):
        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]
        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)
        beta_term = self.beta * S * I / N
        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)
        dSdt = -beta_term + self.gamma * R
        dEdt = beta_term - self.alpha * E
        dIdt = self.alpha * E - self.gamma * I
        dRdt = self.gamma * I
        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)
        dudt = -self.mu * u + self.kappa * I
        h_input = torch.cat([x, u], dim=1)
        h_evolution = self.h_evolution_net(h_input)
        dhdt = h_evolution + self.fh_net(torch.cat([x, h, u], dim=1))

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)
        return dxdt, dudt, dhdt
```

Figure 16: An Example of Code Evolution on the COVID-19 Dataset: The 2nd Iteration.

Code Evolution on the COVID-19 Dataset: The 2nd Iteration (Part II)

```
    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)
        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt
        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 17: An Example of Code Evolution on the COVID-19 Dataset: The 2nd Iteration.

Code Evolution on the COVID-19 Dataset: The 3th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 12
        self.activation = nn.Tanh()

        self.beta = nn.Parameter(torch.tensor(0.50))
        self.gamma = nn.Parameter(torch.tensor(0.10))
        self.epsilon = nn.Parameter(torch.tensor(0.30))
        self.alpha = nn.Parameter(torch.tensor(0.05))
        self.mu = nn.Parameter(torch.tensor(0.02))
```

```
        self.kappa = nn.Parameter(torch.tensor(0.15))

        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )

    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 18: An Example of Code Evolution on the COVID-19 Dataset: The 3th Iteration.

Code Evolution on the COVID-19 Dataset: The 3th Iteration (Part II)

```
    def forward_inner(self, x, h, u):

        batch_size = x.size(0)

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_term = self.beta * S * I / N
        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)

        dSdt = -beta_term + self.gamma * R
        dEdt = beta_term - self.alpha * E
        dIdt = self.alpha * E - self.gamma * I
        dRdt = self.gamma * I

        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)

        dudt = -self.mu * u + self.kappa * I

        dhdt_direct = torch.zeros(
            (batch_size, self.hidden_dim_h),
            device=x.device
        )

        dhdt_residual = self.fh_net(
            torch.cat([x, h, u], dim=1)
        )

        dhdt = dhdt_direct + dhdt_residual

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)

        return dxdt, dudt, dhdt
```

Figure 19: An Example of Code Evolution on the COVID-19 Dataset: The 3th Iteration.

Code Evolution on the COVID-19 Dataset: The 4th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 16
        self.activation = nn.GELU()

        self.beta = nn.Parameter(torch.tensor(0.50))
        self.gamma = nn.Parameter(torch.tensor(0.10))
        self.epsilon = nn.Parameter(torch.tensor(0.30))
        self.alpha = nn.Parameter(torch.tensor(0.05))
        self.mu = nn.Parameter(torch.tensor(0.02))
        self.kappa = nn.Parameter(torch.tensor(0.15))

        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )

        self.h_evolution_net = nn.Sequential(
            nn.Linear(state_dim + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )


    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 20: An Example of Code Evolution on the COVID-19 Dataset: The 4th Iteration.

Code Evolution on the COVID-19 Dataset: The 4th Iteration (Part II)

```
    def forward_inner(self, x, h, u):

        batch_size = x.size(0)

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_term = self.beta * S * I / N
```

```
        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)

        dSdt = -beta_term + self.gamma * R
        dEdt = beta_term - self.alpha * E
        dIdt = self.alpha * E - self.gamma * I
        dRdt = self.gamma * I

        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)

        dudt = -self.mu * u + self.kappa * I

        h_input = torch.cat([x, u], dim=1)
        h_evolution = self.h_evolution_net(h_input)

        dhdt = h_evolution + self.fh_net(
            torch.cat([x, h, u], dim=1)
        )

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)

        return dxdt, dudt, dhdt
```

Figure 21: An Example of Code Evolution on the COVID-19 Dataset: The 4th Iteration.

Code Evolution on the COVID-19 Dataset: The 5th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 12
        self.activation = nn.Tanh()

        self.beta = nn.Parameter(torch.tensor(0.50))
        self.gamma = nn.Parameter(torch.tensor(0.10))
        self.epsilon = nn.Parameter(torch.tensor(0.30))
        self.alpha = nn.Parameter(torch.tensor(0.05))
        self.mu = nn.Parameter(torch.tensor(0.02))
        self.kappa = nn.Parameter(torch.tensor(0.15))

        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )

    def forward_inner(self, x, h, u):

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_term = self.beta * S * I / N
        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)

        dSdt = -beta_term + self.gamma * R
        dEdt = beta_term - self.alpha * E
        dIdt = self.alpha * E - self.gamma * I
```

```
        dRdt = self.gamma * I

        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)

        dudt = -self.mu * u + self.kappa * I

        dhdt_direct = torch.zeros((x.size(0), self.hidden_dim_h), device=x.device)
        dhdt_residual = self.fh_net(torch.cat([x, h, u], dim=1))
        dhdt = dhdt_direct + dhdt_residual

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)

        return dxdt, dudt, dhdt
```

Figure 22: An Example of Code Evolution on the COVID-19 Dataset: The 5th Iteration.

Code Evolution on the COVID-19 Dataset: The 5th Iteration (Part II)

```
    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 23: An Example of Code Evolution on the COVID-19 Dataset: The 5th Iteration.

Code Evolution on the COVID-19 Dataset: The 6th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 8
        self.activation = nn.Tanh()

        self.beta = nn.Parameter(torch.tensor(0.496))
        self.gamma = nn.Parameter(torch.tensor(0.115))
        self.epsilon = nn.Parameter(torch.tensor(0.290))
        self.alpha = nn.Parameter(torch.tensor(0.035))
        self.mu = nn.Parameter(torch.tensor(0.034))
        self.kappa = nn.Parameter(torch.tensor(0.10))

        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 16),
            self.activation,
            nn.Linear(16, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )



    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt
```

```
        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 24: An Example of Code Evolution on the COVID-19 Dataset: The 6th Iteration.

Code Evolution on the COVID-19 Dataset: The 6th Iteration (Part II)

```
    def forward_inner(self, x, h, u):

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_term = self.beta * S * I / N
        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)

        dSdt = -beta_term + self.gamma * R
        dEdt = beta_term - self.alpha * E
        dIdt = self.alpha * E - self.gamma * I
        dRdt = self.gamma * I

        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)

        dudt = -self.mu * u + self.kappa * I

        dhdt = self.fh_net(torch.cat([x, h, u], dim=1))

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)

        return dxdt, dudt, dhdt
```

Figure 25: An Example of Code Evolution on the COVID-19 Dataset: The 6th Iteration.

Code Evolution on the COVID-19 Dataset: The 7th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 8
        self.activation = nn.Tanh()

        # White-box parameters (converged values)
        self.beta = nn.Parameter(torch.tensor(0.50))
        self.gamma = nn.Parameter(torch.tensor(0.10))
        self.epsilon = nn.Parameter(torch.tensor(0.30))
        self.alpha = nn.Parameter(torch.tensor(0.05))
        self.mu = nn.Parameter(torch.tensor(0.02))
        self.kappa = nn.Parameter(torch.tensor(0.15))

        # Simplified neural residuals
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 16),
            self.activation,
            nn.Linear(16, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )
```

```
    def forward_inner(self, x, h, u):

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_term = self.beta * S * I / N
        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)

        dSdt = -beta_term + self.gamma * R
        dEdt = beta_term - self.alpha * E
        dIdt = self.alpha * E - self.gamma * I
        dRdt = self.gamma * I

        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)

        dudt = -self.mu * u + self.kappa * I

        dhdt_direct = torch.zeros((x.size(0), self.hidden_dim_h), device=x.device)
        dhdt_residual = self.fh_net(torch.cat([x, h, u], dim=1))
        dhdt = dhdt_direct + dhdt_residual

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)

        return dxdt, dudt, dhdt
```

Figure 26: An Example of Code Evolution on the COVID-19 Dataset: The 7th Iteration.

Code Evolution on the COVID-19 Dataset: The 7th Iteration (Part II)

```
    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 27: An Example of Code Evolution on the COVID-19 Dataset: The 7th Iteration.

Code Evolution on the COVID-19 Dataset: The 8-th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = 16
        self.activation = nn.GELU()

        # White-box epidemic parameters
        self.beta = nn.Parameter(torch.tensor(0.50))
        self.gamma = nn.Parameter(torch.tensor(0.10))
        self.epsilon = nn.Parameter(torch.tensor(0.30))
        self.alpha = nn.Parameter(torch.tensor(0.05))
        self.mu = nn.Parameter(torch.tensor(0.02))
        self.kappa = nn.Parameter(torch.tensor(0.15))

        # Neural residuals
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )
```

```
        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 32),
            self.activation,
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )

        self.h_evolution_net = nn.Sequential(
            nn.Linear(state_dim + control_dim, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )
```

Figure 28: An Example of Code Evolution on the COVID-19 Dataset: The 8th Iteration.

Code Evolution on the COVID-19 Dataset: The 8-th Iteration (Part II)

```
    def forward_inner(self, x, h, u):

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_term = self.beta * S * I / N
        intervention_effect = u * self.epsilon
        beta_term = beta_term * (1.0 - intervention_effect)

        dSdt = -beta_term + self.gamma * R
        dEdt = beta_term - self.alpha * E
        dIdt = self.alpha * E - self.gamma * I
        dRdt = self.gamma * I

        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)

        dudt = -self.mu * u + self.kappa * I

        h_input = torch.cat([x, u], dim=1)
        h_evolution = self.h_evolution_net(h_input)

        dhdt = h_evolution + self.fh_net(torch.cat([x, h, u], dim=1))

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)

        return dxdt, dudt, dhdt

    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 29: An Example of Code Evolution on the COVID-19 Dataset: The 8-th Iteration.

Code Evolution on the COVID-19 Dataset: The 9th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
```

```
        self.hidden_dim_h = 32
        self.activation = nn.GELU()

        # White-box epidemic parameters
        self.beta_s = nn.Parameter(torch.tensor(0.50))
        self.beta_e = nn.Parameter(torch.tensor(0.10))
        self.beta_i = nn.Parameter(torch.tensor(0.05))

        self.gamma_s = nn.Parameter(torch.tensor(0.10))
        self.gamma_i = nn.Parameter(torch.tensor(0.15))

        self.epsilon = nn.Parameter(torch.tensor(0.30))
        self.mu = nn.Parameter(torch.tensor(0.02))
        self.kappa = nn.Parameter(torch.tensor(0.15))

        self.delta = nn.Parameter(torch.tensor(0.01))
        self.theta = nn.Parameter(torch.tensor(0.005))

        # Deep neural residuals
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 256),
            self.activation,
            nn.Linear(256, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + self.hidden_dim_h + control_dim, 256),
            self.activation,
            nn.Linear(256, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )

        self.h_evolution_net = nn.Sequential(
            nn.Linear(state_dim + control_dim, 128),
            self.activation,
            nn.Linear(128, 64),
            self.activation,
            nn.Linear(64, 32),
            self.activation,
            nn.Linear(32, self.hidden_dim_h)
        )
```

Figure 30: An Example of Code Evolution on the COVID-19 Dataset: The 9th Iteration.

Code Evolution on the COVID-19 Dataset: The 9th Iteration (Part II)

```
def forward_inner(self, x, h, u):

        S = x[:, 0:1]
        E = x[:, 1:2]
        I = x[:, 2:3]
        R = x[:, 3:4]

        N = torch.sum(x, dim=1, keepdim=True)
        N = torch.clamp(N, min=1e-8)

        beta_base = self.beta_s * (1.0 - u * self.epsilon)
        beta_eff = beta_base * (1.0 + self.beta_e * E + self.beta_i * I)

        dSdt = -beta_eff * S * I / N + self.gamma_s * R
        dEdt = beta_eff * S * I / N - self.gamma_i * E
        dIdt = self.gamma_i * E - self.gamma_s * I
        dRdt = self.gamma_s * I

        dxdt = torch.cat([dSdt, dEdt, dIdt, dRdt], dim=1)
```

```
        dudt = -self.mu * u + self.kappa * I * torch.sigmoid(u) + self.delta * (1.0 - u)

        h_input = torch.cat([x, u], dim=1)
        h_evolution = self.h_evolution_net(h_input)

        dhdt = h_evolution + self.fh_net(torch.cat([x, h, u], dim=1))

        concat_input = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(concat_input)
        dudt += self.fu_net(concat_input)

        return dxdt, dudt, dhdt

    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, min=0.0)
        u_next = torch.clamp(u_next, min=0.0, max=1.0)

        return x_next, u_next, h_next
```

Figure 31: An Example of Code Evolution on the COVID-19 Dataset: The 9th Iteration.

Code Evolution on the COVID-19 Dataset: The 10th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int, hidden_dim_h: int = 8):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = hidden_dim_h

        # Highly optimized parameters based on best performing iterations
        self.beta = nn.Parameter(torch.tensor(0.461), requires_grad=True)
        self.gamma = nn.Parameter(torch.tensor(0.098), requires_grad=True)
        self.sigma = nn.Parameter(torch.tensor(0.194), requires_grad=True)

        self.intervention_effect = nn.Parameter(torch.tensor(0.342), requires_grad=True)
        self.intervention_sensitivity = nn.Parameter(torch.tensor(1.05), requires_grad=True)

        # Enhanced latent memory parameters
        self.memory_decay = nn.Parameter(torch.tensor(0.0751), requires_grad=True)
        self.memory_strength = nn.Parameter(torch.tensor(0.539), requires_grad=True)
        self.memory_reset = nn.Parameter(torch.tensor(0.0336), requires_grad=True)

        # Neural residual networks
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 32),
            nn.SiLU(),
            nn.Linear(32, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 16),
            nn.SiLU(),
            nn.Linear(16, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 16),
            nn.SiLU(),
            nn.Linear(16, hidden_dim_h)
        )
```

Figure 32: Example of the 10th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 10th Iteration (Part II)

```
    def forward_inner(self, x: torch.Tensor, h: torch.Tensor, u: torch.Tensor):

        s = x[:, 0]
        e = x[:, 1]
        i = x[:, 2]
        r = x[:, 3]
```

```
        s = torch.clamp(s, 0.0, 1.0)
        e = torch.clamp(e, 0.0, 1.0)
        i = torch.clamp(i, 0.0, 1.0)
        r = torch.clamp(r, 0.0, 1.0)

        total = s + e + i + r
        eps = torch.max(total * 1e-18, torch.tensor(1e-18, device=x.device))

        s = s / (total + eps)
        e = e / (total + eps)
        i = i / (total + eps)
        r = r / (total + eps)

        beta_eff = self.beta * (1.0 - u.squeeze() * self.intervention_effect)
        beta_eff = torch.clamp(beta_eff, 1e-8, 1.0)

        ds_dt = -beta_eff * s * i
        de_dt = beta_eff * s * i - self.sigma * e
        di_dt = self.sigma * e - self.gamma * i
        dr_dt = self.gamma * i

        dxdt = torch.stack([ds_dt, de_dt, di_dt, dr_dt], dim=1)

        intervention_rate = torch.sigmoid(
            self.intervention_sensitivity * (i - 0.01)
        )
        du_dt = intervention_rate - u.squeeze()
        du_dt = du_dt.unsqueeze(-1)

        memory_change = self.memory_strength * i.unsqueeze(-1).expand(-1, self.hidden_dim_h)
        memory_reset_signal = self.memory_reset * u.squeeze().unsqueeze(-1).expand(-1, self.hidden_dim_h)

        dhdt = -self.memory_decay * h + memory_change + memory_reset_signal

        xu_cat = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(xu_cat)
        du_dt += self.fu_net(xu_cat)
        dhdt += self.fh_net(xu_cat)

        return dxdt, du_dt, dhdt

    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, 0.0, 1.0)

        x_sum = torch.sum(x_next, dim=1, keepdim=True)
        eps = torch.max(x_sum * 1e-18, torch.tensor(1e-18, device=x.device))
        x_next = x_next / (x_sum + eps)

        u_next = torch.clamp(u_next, 0.0, 1.0)

        return x_next, u_next, h_next
```

Figure 33: Example of the 10th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 11th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int, hidden_dim_h: int = 16):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = hidden_dim_h

        # Highly precise learnable parameters optimized to maximum accuracy
        self.beta = nn.Parameter(torch.tensor(0.4608, requires_grad=True))
        self.gamma = nn.Parameter(torch.tensor(0.0979, requires_grad=True))
        self.sigma = nn.Parameter(torch.tensor(0.1937, requires_grad=True))

        self.intervention_effect = nn.Parameter(torch.tensor(0.3418, requires_grad=True))
        self.intervention_sensitivity = nn.Parameter(torch.tensor(1.049, requires_grad=True))

        # Enhanced latent memory parameters
        self.memory_decay = nn.Parameter(torch.tensor(0.07505, requires_grad=True))
        self.memory_strength = nn.Parameter(torch.tensor(0.5387, requires_grad=True))
        self.memory_reset = nn.Parameter(torch.tensor(0.03357, requires_grad=True))
```

```
        self.memory_smoothing = nn.Parameter(torch.tensor(0.0498, requires_grad=True))

        # Deep neural residual networks
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 512),
            nn.SiLU(),
            nn.LayerNorm(512),
            nn.Dropout(0.02),
            nn.Linear(512, 256),
            nn.SiLU(),
            nn.LayerNorm(256),
            nn.Dropout(0.02),
            nn.Linear(256, 128),
            nn.SiLU(),
            nn.LayerNorm(128),
            nn.Dropout(0.02),
            nn.Linear(128, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 256),
            nn.SiLU(),
            nn.LayerNorm(256),
            nn.Dropout(0.02),
            nn.Linear(256, 128),
            nn.SiLU(),
            nn.LayerNorm(128),
            nn.Dropout(0.02),
            nn.Linear(128, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Dropout(0.02),
            nn.Linear(64, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 256),
            nn.SiLU(),
            nn.LayerNorm(256),
            nn.Dropout(0.02),
            nn.Linear(256, 128),
            nn.SiLU(),
            nn.LayerNorm(128),
            nn.Dropout(0.02),
            nn.Linear(128, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Dropout(0.02),
            nn.Linear(64, hidden_dim_h)
        )
```

Figure 34: Example of the 11th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 11th Iteration (Part II)

```
    def forward_inner(self, x, h, u):

        s = x[:, 0]
        e = x[:, 1]
        i = x[:, 2]
        r = x[:, 3]

        s = torch.clamp(s, 0.0, 1.0)
        e = torch.clamp(e, 0.0, 1.0)
        i = torch.clamp(i, 0.0, 1.0)
        r = torch.clamp(r, 0.0, 1.0)

        total = s + e + i + r
        eps = torch.max(total * 1e-20, torch.tensor(1e-20, device=x.device))

        s = s / (total + eps)
        e = e / (total + eps)
        i = i / (total + eps)
        r = r / (total + eps)

        beta_eff = self.beta * (1.0 - u.squeeze() * self.intervention_effect)
        beta_eff = torch.clamp(beta_eff, 1e-10, 1.0)

        ds_dt = -beta_eff * s * i
        de_dt = beta_eff * s * i - self.sigma * e
        di_dt = self.sigma * e - self.gamma * i
        dr_dt = self.gamma * i

        dxdt = torch.stack([ds_dt, de_dt, di_dt, dr_dt], dim=1)

        intervention_rate = torch.sigmoid(
            self.intervention_sensitivity * (i - 0.0095)
```

```
        )
        du_dt = intervention_rate - u.squeeze()
        du_dt = du_dt.unsqueeze(-1)

        memory_change = self.memory_strength * i.unsqueeze(-1).expand(-1, self.hidden_dim_h)
        memory_reset_signal = self.memory_reset * u.squeeze().unsqueeze(-1).expand(-1, self.hidden_dim_h)

        dhdt = -self.memory_decay * h + memory_change + memory_reset_signal

        xu_cat = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(xu_cat)
        du_dt += self.fu_net(xu_cat)
        dhdt += self.fh_net(xu_cat)

        return dxdt, du_dt, dhdt

    def forward_outer(self, x, h, u, dt: float = 1.0):

        dxdt, dudt, dhdt = self.forward_inner(x, h, u)

        x_next = x + dxdt * dt
        u_next = u + dudt * dt
        h_next = h + dhdt * dt

        x_next = torch.clamp(x_next, 0.0, 1.0)

        x_sum = torch.sum(x_next, dim=1, keepdim=True)
        eps = torch.max(x_sum * 1e-20, torch.tensor(1e-20, device=x.device))
        x_next = x_next / (x_sum + eps)

        u_next = torch.clamp(u_next, 0.0, 1.0)

        return x_next, u_next, h_next
```

Figure 35: Example of the 11th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 12th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int, hidden_dim_h: int = 16):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = hidden_dim_h

        # Extremely precise learnable parameters optimized for maximum accuracy
        self.beta = nn.Parameter(torch.tensor(0.4618, requires_grad=True))
        self.gamma = nn.Parameter(torch.tensor(0.0987, requires_grad=True))
        self.sigma = nn.Parameter(torch.tensor(0.1932, requires_grad=True))

        self.intervention_effect = nn.Parameter(torch.tensor(0.3405, requires_grad=True))
        self.intervention_sensitivity = nn.Parameter(torch.tensor(1.048, requires_grad=True))

        # Enhanced latent memory parameters
        self.memory_decay = nn.Parameter(torch.tensor(0.0742, requires_grad=True))
        self.memory_strength = nn.Parameter(torch.tensor(0.539, requires_grad=True))
        self.memory_reset = nn.Parameter(torch.tensor(0.0321, requires_grad=True))

        # Optimized neural network residuals
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 128),
            nn.SiLU(),
            nn.LayerNorm(128),
            nn.Linear(128, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, 32),
            nn.SiLU(),
            nn.LayerNorm(32),
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 64),
            nn.SiLU(),
```

```
            nn.LayerNorm(64),
            nn.Linear(64, 32),
            nn.SiLU(),
            nn.LayerNorm(32),
            nn.Linear(32, hidden_dim_h)
        )
```

Figure 36: Example of the 12th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: 12th Iteration (Part II)

```
  def forward_inner(self, x, h, u):
        s = x[:, 0]
        e = x[:, 1]
        i = x[:, 2]
        r = x[:, 3]
        s = torch.clamp(s, 0.0, 1.0)
        e = torch.clamp(e, 0.0, 1.0)
        i = torch.clamp(i, 0.0, 1.0)
        r = torch.clamp(r, 0.0, 1.0)
        total = s + e + i + r
        eps = torch.max(total * 1e-22, torch.tensor(1e-22, device=x.device))
        s = s / (total + eps)
        e = e / (total + eps)
        i = i / (total + eps)
        r = r / (total + eps)
        beta_eff = self.beta * (1.0 - u.squeeze() * self.intervention_effect)
        beta_eff = torch.clamp(beta_eff, 1e-12, 1.0)
        ds_dt = -beta_eff * s * i
        de_dt = beta_eff * s * i - self.sigma * e
        di_dt = self.sigma * e - self.gamma * i
        dr_dt = self.gamma * i
        dxdt = torch.stack([ds_dt, de_dt, di_dt, dr_dt], dim=1)

        intervention_rate = torch.sigmoid(
            self.intervention_sensitivity * (i - 0.01)
        )
        du_dt = intervention_rate - u.squeeze()
        du_dt = du_dt.unsqueeze(-1)
        memory_change = self.memory_strength * i.unsqueeze(-1).expand(-1, self.hidden_dim_h)
        memory_reset_signal = self.memory_reset * u.squeeze().unsqueeze(-1).expand(-1, self.hidden_dim_h)

        dhdt = -self.memory_decay * h + memory_change + memory_reset_signal

        xu_cat = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(xu_cat)
        du_dt += self.fu_net(xu_cat)
        dhdt += self.fh_net(xu_cat)

        return dxdt, du_dt, dhdt

    def forward_outer(self, x, h, u, dt: float = 1.0):

        # RK4 integration
        k1_x, k1_u, k1_h = self.forward_inner(x, h, u)

        x2 = x + k1_x * dt * 0.5
        h2 = h + k1_h * dt * 0.5
        u2 = u + k1_u * dt * 0.5
        k2_x, k2_u, k2_h = self.forward_inner(x2, h2, u2)

        x3 = x + k2_x * dt * 0.5
        h3 = h + k2_h * dt * 0.5
        u3 = u + k2_u * dt * 0.5
        k3_x, k3_u, k3_h = self.forward_inner(x3, h3, u3)

        x4 = x + k3_x * dt
        h4 = h + k3_h * dt
        u4 = u + k3_u * dt
        k4_x, k4_u, k4_h = self.forward_inner(x4, h4, u4)

        x_next = x + (k1_x + 2*k2_x + 2*k3_x + k4_x) * dt / 6
        u_next = u + (k1_u + 2*k2_u + 2*k3_u + k4_u) * dt / 6
        h_next = h + (k1_h + 2*k2_h + 2*k3_h + k4_h) * dt / 6

        x_next = torch.clamp(x_next, 0.0, 1.0)

        x_sum = torch.sum(x_next, dim=1, keepdim=True)
        eps = torch.max(x_sum * 1e-22, torch.tensor(1e-22, device=x.device))
        x_next = x_next / (x_sum + eps)

        u_next = torch.clamp(u_next, 0.0, 1.0)

        return x_next, u_next, h_next
```

Figure 37: Example of the 12th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 13th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int, hidden_dim_h: int = 16):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = hidden_dim_h

        # Highly optimized parameters based on best previous iterations
        self.beta = nn.Parameter(torch.tensor(0.4608, requires_grad=True))
        self.gamma = nn.Parameter(torch.tensor(0.0979, requires_grad=True))
        self.sigma = nn.Parameter(torch.tensor(0.1937, requires_grad=True))

        self.intervention_effect = nn.Parameter(torch.tensor(0.3418, requires_grad=True))
        self.intervention_sensitivity = nn.Parameter(torch.tensor(1.049, requires_grad=True))

        # Enhanced latent memory parameters
        self.memory_decay = nn.Parameter(torch.tensor(0.07505, requires_grad=True))
        self.memory_strength = nn.Parameter(torch.tensor(0.5387, requires_grad=True))
        self.memory_reset = nn.Parameter(torch.tensor(0.03357, requires_grad=True))

        # Simplified neural network residuals
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 128),
            nn.SiLU(),
            nn.LayerNorm(128),
            nn.Linear(128, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, 32),
            nn.SiLU(),
            nn.LayerNorm(32),
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, 32),
            nn.SiLU(),
            nn.LayerNorm(32),
            nn.Linear(32, hidden_dim_h)
        )
```

Figure 38: Example of the 13th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 13th Iteration (Part II)

```
 def forward_inner(self, x, h, u):

        s = x[:, 0]
        e = x[:, 1]
        i = x[:, 2]
        r = x[:, 3]

        s = torch.clamp(s, 0.0, 1.0)
        e = torch.clamp(e, 0.0, 1.0)
        i = torch.clamp(i, 0.0, 1.0)
        r = torch.clamp(r, 0.0, 1.0)

        total = s + e + i + r
        eps = torch.max(total * 1e-20, torch.tensor(1e-20, device=x.device))

        s = s / (total + eps)
        e = e / (total + eps)
        i = i / (total + eps)
        r = r / (total + eps)

        beta_eff = self.beta * (1.0 - u.squeeze() * self.intervention_effect)
        beta_eff = torch.clamp(beta_eff, 1e-10, 1.0)
```

```
        ds_dt = -beta_eff * s * i
        de_dt = beta_eff * s * i - self.sigma * e
        di_dt = self.sigma * e - self.gamma * i
        dr_dt = self.gamma * i

        dxdt = torch.stack([ds_dt, de_dt, di_dt, dr_dt], dim=1)

        intervention_rate = torch.sigmoid(
            self.intervention_sensitivity * (i - 0.0095)
        )
        du_dt = intervention_rate - u.squeeze()
        du_dt = du_dt.unsqueeze(-1)

        memory_change = self.memory_strength * i.unsqueeze(-1).expand(-1, self.hidden_dim_h)
        memory_reset_signal = self.memory_reset * u.squeeze().unsqueeze(-1).expand(-1, self.hidden_dim_h)

        dhdt = -self.memory_decay * h + memory_change + memory_reset_signal

        xu_cat = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(xu_cat)
        du_dt += self.fu_net(xu_cat)
        dhdt += self.fh_net(xu_cat)

        return dxdt, du_dt, dhdt
```

Figure 39: Example of the 13th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 13th Iteration (Part III)

```
    def forward_outer(self, x, h, u, dt: float = 1.0):

        # RK4 integration
        k1_x, k1_u, k1_h = self.forward_inner(x, h, u)

        x2 = x + k1_x * dt * 0.5
        h2 = h + k1_h * dt * 0.5
        u2 = u + k1_u * dt * 0.5
        k2_x, k2_u, k2_h = self.forward_inner(x2, h2, u2)

        x3 = x + k2_x * dt * 0.5
        h3 = h + k2_h * dt * 0.5
        u3 = u + k2_u * dt * 0.5
        k3_x, k3_u, k3_h = self.forward_inner(x3, h3, u3)

        x4 = x + k3_x * dt
        h4 = h + k3_h * dt
        u4 = u + k3_u * dt
        k4_x, k4_u, k4_h = self.forward_inner(x4, h4, u4)

        x_next = x + (k1_x + 2*k2_x + 2*k3_x + k4_x) * dt / 6
        u_next = u + (k1_u + 2*k2_u + 2*k3_u + k4_u) * dt / 6
        h_next = h + (k1_h + 2*k2_h + 2*k3_h + k4_h) * dt / 6

        x_next = torch.clamp(x_next, 0.0, 1.0)

        x_sum = torch.sum(x_next, dim=1, keepdim=True)
        eps = torch.max(x_sum * 1e-20, torch.tensor(1e-20, device=x.device))
        x_next = x_next / (x_sum + eps)

        u_next = torch.clamp(u_next, 0.0, 1.0)

        return x_next, u_next, h_next
```

Figure 40: Example of the 13th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 14th Iteration (Part I)

```
import torch
import torch.nn as nn
from typing import Tuple

class ControlledStateODE(nn.Module):
    def __init__(self, state_dim: int, control_dim: int, hidden_dim_h: int = 16):
        super(ControlledStateODE, self).__init__()
        self.state_dim = state_dim
        self.control_dim = control_dim
        self.hidden_dim_h = hidden_dim_h

        # Extremely precise learnable parameters optimized for maximum accuracy
        self.beta = nn.Parameter(torch.tensor(0.4591, requires_grad=True))
        self.gamma = nn.Parameter(torch.tensor(0.0983, requires_grad=True))
```

```
        self.sigma = nn.Parameter(torch.tensor(0.1925, requires_grad=True))

        self.intervention_effect = nn.Parameter(torch.tensor(0.3492, requires_grad=True))
        self.intervention_sensitivity = nn.Parameter(torch.tensor(1.061, requires_grad=True))

        # Enhanced latent memory parameters
        self.memory_decay = nn.Parameter(torch.tensor(0.0758, requires_grad=True))
        self.memory_strength = nn.Parameter(torch.tensor(0.5381, requires_grad=True))
        self.memory_reset = nn.Parameter(torch.tensor(0.0323, requires_grad=True))

        # Optimized neural network residuals
        self.fx_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 128),
            nn.SiLU(),
            nn.LayerNorm(128),
            nn.Linear(128, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, state_dim)
        )

        self.fu_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, 32),
            nn.SiLU(),
            nn.LayerNorm(32),
            nn.Linear(32, control_dim)
        )

        self.fh_net = nn.Sequential(
            nn.Linear(state_dim + hidden_dim_h + control_dim, 64),
            nn.SiLU(),
            nn.LayerNorm(64),
            nn.Linear(64, 32),
            nn.SiLU(),
            nn.LayerNorm(32),
            nn.Linear(32, hidden_dim_h)
        )
```

Figure 41: Example of the 14th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 14th Iteration (Part II)

```
    def forward_inner(self, x, h, u):

        s = x[:, 0]
        e = x[:, 1]
        i = x[:, 2]
        r = x[:, 3]

        s = torch.clamp(s, 0.0, 1.0)
        e = torch.clamp(e, 0.0, 1.0)
        i = torch.clamp(i, 0.0, 1.0)
        r = torch.clamp(r, 0.0, 1.0)

        total = s + e + i + r
        eps = torch.max(total * 1e-22, torch.tensor(1e-22, device=x.device))

        s = s / (total + eps)
        e = e / (total + eps)
        i = i / (total + eps)
        r = r / (total + eps)

        beta_eff = self.beta * (1.0 - u.squeeze() * self.intervention_effect)
        beta_eff = torch.clamp(beta_eff, 1e-12, 1.0)

        ds_dt = -beta_eff * s * i
        de_dt = beta_eff * s * i - self.sigma * e
        di_dt = self.sigma * e - self.gamma * i
        dr_dt = self.gamma * i

        dxdt = torch.stack([ds_dt, de_dt, di_dt, dr_dt], dim=1)

        intervention_rate = torch.sigmoid(
            self.intervention_sensitivity * (i - 0.0095)
        )
        du_dt = intervention_rate - u.squeeze()
        du_dt = du_dt.unsqueeze(-1)

        memory_change = self.memory_strength * i.unsqueeze(-1).expand(-1, self.hidden_dim_h)
        memory_reset_signal = self.memory_reset * u.squeeze().unsqueeze(-1).expand(-1, self.hidden_dim_h)

        dhdt = -self.memory_decay * h + memory_change + memory_reset_signal
```

```
        xu_cat = torch.cat([x, h, u], dim=1)

        dxdt += self.fx_net(xu_cat)
        du_dt += self.fu_net(xu_cat)
        dhdt += self.fh_net(xu_cat)

        return dxdt, du_dt, dhdt
```

Figure 42: Example of the 14th Iteration Code Evolution on the COVID-19 Dataset.

Code Evolution on the COVID-19 Dataset: The 14th Iteration (Part III)

```
    def forward_outer(self, x, h, u, dt: float = 1.0):

        # RK4 integration
        k1_x, k1_u, k1_h = self.forward_inner(x, h, u)

        x2 = x + k1_x * dt * 0.5
        h2 = h + k1_h * dt * 0.5
        u2 = u + k1_u * dt * 0.5
        k2_x, k2_u, k2_h = self.forward_inner(x2, h2, u2)

        x3 = x + k2_x * dt * 0.5
        h3 = h + k2_h * dt * 0.5
        u3 = u + k2_u * dt * 0.5
        k3_x, k3_u, k3_h = self.forward_inner(x3, h3, u3)

        x4 = x + k3_x * dt
        h4 = h + k3_h * dt
        u4 = u + k3_u * dt
        k4_x, k4_u, k4_h = self.forward_inner(x4, h4, u4)

        x_next = x + (k1_x + 2*k2_x + 2*k3_x + k4_x) * dt / 6
        u_next = u + (k1_u + 2*k2_u + 2*k3_u + k4_u) * dt / 6
        h_next = h + (k1_h + 2*k2_h + 2*k3_h + k4_h) * dt / 6

        x_next = torch.clamp(x_next, 0.0, 1.0)

        x_sum = torch.sum(x_next, dim=1, keepdim=True)
        eps = torch.max(x_sum * 1e-22, torch.tensor(1e-22, device=x.device))
        x_next = x_next / (x_sum + eps)

        u_next = torch.clamp(u_next, 0.0, 1.0)

        return x_next, u_next, h_next
```

Figure 43: Example of the 14th Iteration Code Evolution on the COVID-19 Dataset.