Contents

Forecasting Long-term Spatial-temporal Dynamics with Generative Transformer Networks

Donggeun Park, Hugon Lee, and Seunghwa Ryu*

Department of Mechanical Engineering, Korea Advanced Institute of Science and Technology (KAIST), Daejeon 34141, Republic of Korea

*Corresponding author: ryush@kaist.ac.kr

A Background

Spatiotemporal dynamics are at the heart of modern science and engineering, shaping our understanding of complex systems and their behavior over time and space. From climate simulations to the intricate analysis of turbulent flows and material failure mechanisms, computational simulations and sensing technologies have become essential tools. These approaches provide invaluable insights into specific scenarios; however, they face significant challenges when it comes to real-time forecasting. The vast amount of data, the substantial computational costs, and the inherent uncertainties associated with dynamic systems make real-time prediction a formidable task.

These limitations are especially pronounced in real-world applications, where it is impractical to perfectly model every aspect of a system. Real-world data often exhibits nonlinear and chaotic interactions among numerous variables, making precise prediction a challenging endeavor. To address these issues, deep learning has emerged as a game-changing approach. With its capacity to learn complex, nonlinear relationships and process data at multiple scales, deep learning has shown remarkable potential in overcoming many of the limitations faced by traditional methods.

Over time, the integration of AI and simulation techniques has evolved significantly, as illustrated in **Figure S1**. This evolution can be divided into four distinct generations, each building on the strengths of its predecessors while attempting to address their weaknesses:

- **Generation 1** introduced Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs) for processing spatiotemporal data. While these models were effective for learning simpler patterns, they struggled to capture long-term dependencies, which limited their predictive capabilities.

- **Generation 2** saw the rise of more sophisticated models, including Physics-informed Neural Networks (PINNs), ConvLSTM (a combination of CNNs and RNNs), and Transformers.
  - PINNs integrated governing physical equations into their training processes, enhancing the realism of simulations. However, they required a deep understanding of these governing equations, which limited their flexibility for diverse real-world problems.
  - ConvLSTM brought a hybrid approach, combining spatial feature extraction and temporal sequence modeling to improve spatiotemporal dependency handling. Despite this, it faced limitations in managing ultra-long term dynamics, constraining its use in more extended predictive tasks.
  - Transformers provided a breakthrough by handling long-term dependencies more effectively than prior architectures. However, they still encountered issues with spatial information loss, particularly when processing complex spatiotemporal data due to their reliance on single-scale processing.
- **Generation 3** featured the development of Deep Generative Models, offering unprecedented capabilities for handling intricate spatiotemporal patterns and uncertainties. Despite their promise, challenges related to computational expense and the complexity of the models persisted, hindering their application in real-time scenarios.
- **Generation 4** emerged to tackle these remaining obstacles. In this context, we introduce DynamicGPT, a model built upon a Vision Transformer (ViT)-based architecture that preserves multi-scale spatial features and effectively models long-term temporal dependencies. This design strikes an optimal balance between high-accuracy predictions and real-time feasibility.

In this supplementary information, we will provide an in-depth look at the mechanisms of DynamicGPT, including its inference process and architectural components. Additionally, we will outline how we selected and utilized baseline models to benchmark DynamicGPT's performance in a fair and rigorous manner.

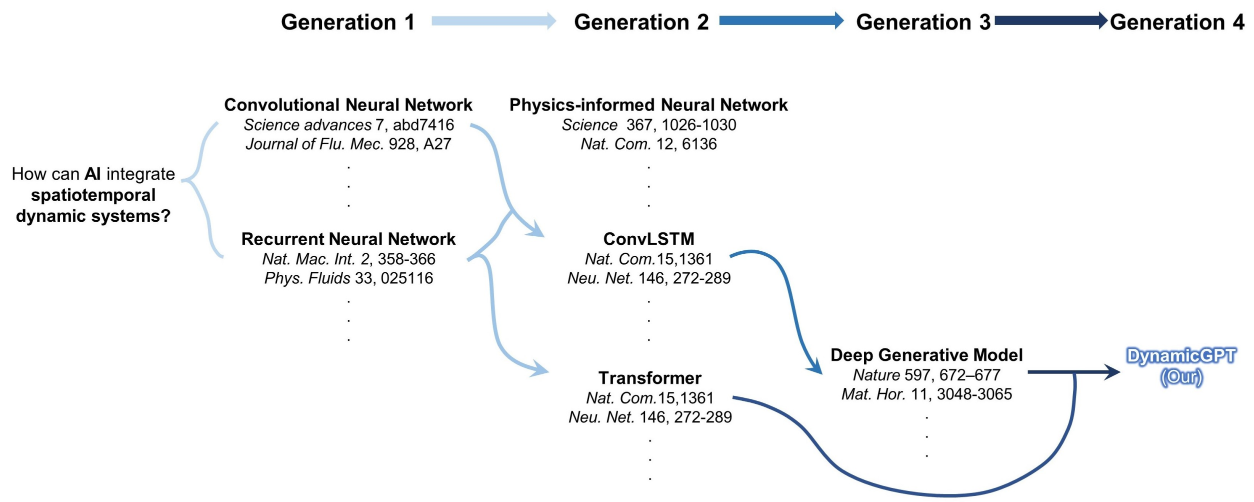


Figure S1: Evolution of neural network architectures for integrating spatiotemporal dynamic systems, progressing from basic models (Generation 1) to advanced models like Deep Generative Models and DynamicGPT (Generation 4).

B DynamicGPT

In this section, we provide detailed insights aimed at reinforcing the explanations presented in **DynamicGPT Implementation Details and Training Process** of the **Main Text**. We describe the data preprocessing required for training and inference in DynamicGPT, delve into the architecture's key mechanisms, and explain its training strategy, including the rationale behind specific design choices. Additionally, we outline the optimized hyperparameters used for validating DynamicGPT on real-world problem datasets, adding further depth to understanding its performance and applicability.

B.1 DynamicGPT Data process

As illustrated in **Figure S2**, DynamicGPT's training strategy begins with processing spatiotemporal data using a sliding window approach. The input consists of past dynamic fields with a length $\boldsymbol{T}$, and the target is the subsequent frame. This preprocessing method effectively augments the dataset by sliding the window frame-by-frame, ensuring that the model learns from various temporal contexts across the entire dataset. Once trained, DynamicGPT iteratively predicts frames, where each predicted frame is fed back into the model as input for subsequent predictions. This process allows for accurate long-term spatiotemporal forecasting, providing continuity and consistency over extended periods.

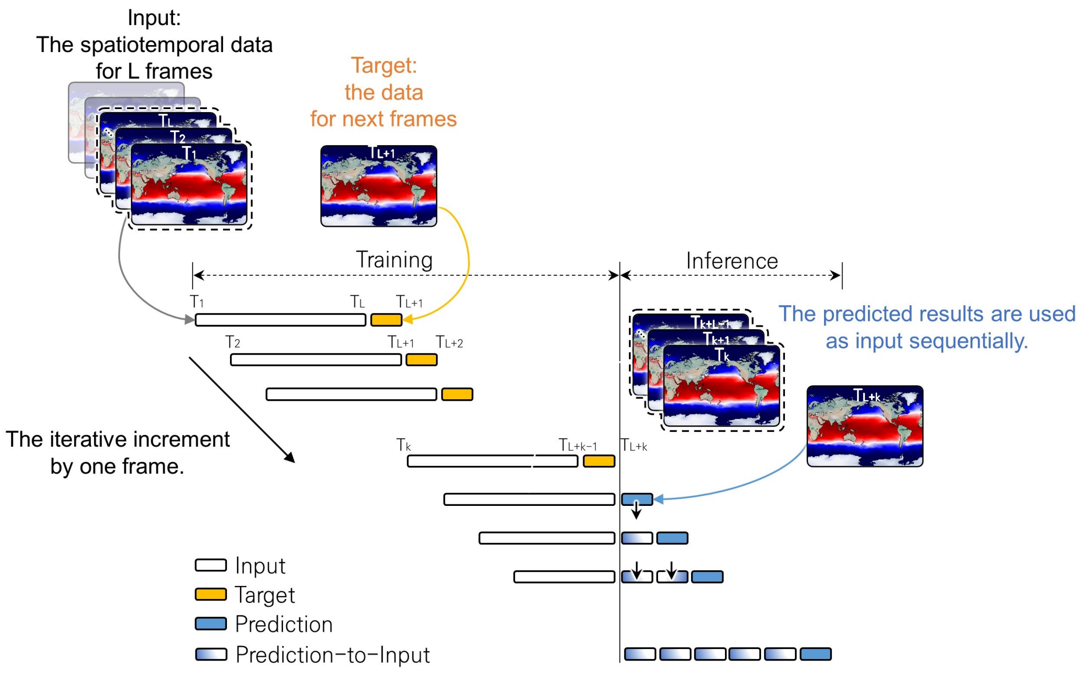

Input:
The spatiotemporal data for L frames
Target: the data for next frames
Training
Inference
The predicted results are used as input sequentially.
The iterative increment by one frame.
Input
Target
Prediction
Prediction−to−Input

Figure S2: The process of training and inference in DynamicGPT. The input consists of spatiotemporal data for 1-L frames, where the target is the data for the next frame L+1. During training, the model learns to predict the target from the input frames. In the inference phase, the predicted results are used sequentially as input to forecast subsequent frames, demonstrating the iterative increment by one frame.

B.2 DynamicGPT Architecture

DynamicGPT's architecture features powerful components that address the limitations of previous **generations 2** and **3**.

- **First Key Mechanism – Multi-spatial Feature Extraction**: As seen in **Figure S3**, the first step of DynamicGPT involves extracting comprehensive spatial features from preprocessed spatiotemporal data pairs using a multi-kernel encoder. The encoder applies multiple kernel sizes (e.g., 2x2, 4x4, 8x8) to capture both fine local details and broader global patterns. This multi-scale approach ensures that the model maintains rich spatial information throughout the feature extraction process. For each input X and kernel F_k , the feature maps F_k are generated as follows (Eq. 1):

$$F_k = X * K_k \quad \forall k \in \{2, 4, 8\} \tag{1}$$

The extracted features F_2, F_4, F_8 are fused through a 1x1 convolutional layer, which combines them efficiently while reducing learning parameters (Eq. 2):

$$F_{\text{fused}} = \sigma\left(W_{\text{fusion}} \cdot [F_2; F_4; F_8]\right) \tag{2}$$

where W_{fusion} denotes the weights of the 1×1 convolution, σ is a non-linear activation function (e.g., ReLU), and $[\cdot]$ represents concatenation of feature maps. This convolution operation not only enhances model performance but also maintains computational efficiency. Next, the resulting latent patches are then processed with a variational sampling technique, modeling the inherent uncertainty present in real-world spatiotemporal dynamics, where the mean μ and variance σ are calculated as follows:

$$\mu = W_\mu F_{\text{fused}} + b_\mu \tag{3}$$

$$\sigma = \exp\left(W_\sigma F_{\text{fused}} + b_\sigma\right) \tag{4}$$

The final latent vector z is sampled using the reparameterization trick:

$$z = \mu + \sigma \odot \epsilon, \quad \epsilon \sim \mathcal{N}(0, I) \tag{5}$$

where $\odot$ denotes element-wise multiplication. This probabilistic approach ensures that the model can generalize effectively, making the method robust in handling unseen scenarios.

- **Second Key Mechanism – Temporal Modeling with Latent Patches**: **Figure S4** illustrates how DynamicGPT enhances temporal modeling using the latent patches produced by the multi-kernel encoder. Unlike Generations 2 and 3, which struggled to preserve spatiotemporal characteristics, DynamicGPT employs a Vision Transformer (ViT) to strengthen temporal connections between latent patches. To capture varying temporal dependencies (short, medium, and long-term), a padding technique aligns latent patches of different input lengths $\boldsymbol{T}$. The core of this ViT is the Multi-Head Self-Attention (MHSA) mechanism, which allows the model to focus on different temporal aspects by processing each latent patch independently across multiple attention heads.

For each latent patch z_i, linear projections are created to form the query (Q), key (K), and value (V) matrices:

$$Q_i = W_Q z_i, \quad K_i = W_K z_i, \quad V_i = W_V z_i \tag{6}$$

where W_Q, W_K, and W_V are learned weights. The attention score for each head is computed as a scaled dot-product between queries and keys, followed by a softmax operation to ensure probabilities:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right) V \tag{7}$$

Here, d_k is the dimension of the key vectors, used to scale the dot-product for stable gradient updates. The outputs from each attention head are concatenated and passed through a final linear layer to integrate the multi-head outputs:

$$\text{MHSA\_output} = W_O \cdot [\text{head}_1, \text{head}_2, \ldots, \text{head}_h] \tag{8}$$

where W_O is the learned weight of the output projection. This MHSA mechanism enables the model to capture short, medium, and long-term temporal dependencies, effectively maintaining temporal integrity across varying input lengths $\boldsymbol{T}$.

Furthermore, to enhance the model's temporal modeling capability, DynamicGPT incorporates a ConvGRU module, which is integrated into the multi-scale temporal network. This ConvGRU helps capture sequential dependencies more effectively within the temporal sequence, reinforcing the latent patch representations. The ConvGRU operates as follows:

- Update Gate:

$$z_t = \sigma\left(W_{xz} * z_{t-1} + U_{hz} * h_{t-1} + b_z\right) \tag{9}$$

- Reset Gate:

$$r_t = \sigma\left(W_{xr} * z_{t-1} + U_{hr} * h_{t-1} + b_r\right) \tag{10}$$

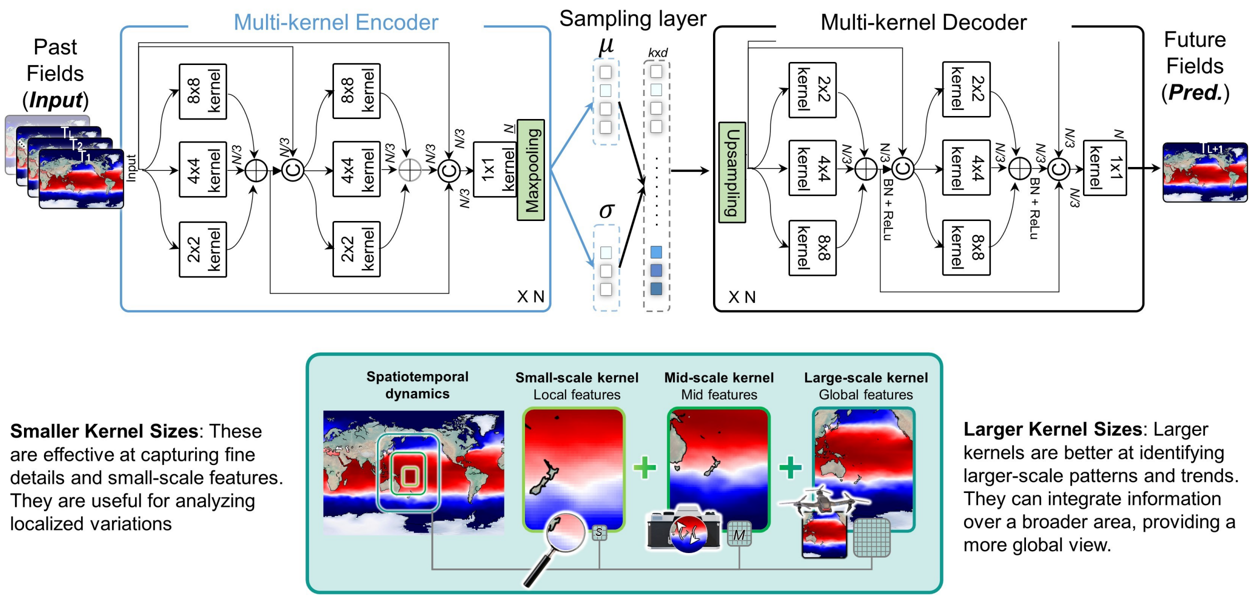
- New Memory Content:

$$\tilde{h}_t = \tanh\left(W_{x\tilde{h}} * z_{t-1} + r_t \odot (U_{h\tilde{h}} * h_{t-1}) + b_{\tilde{h}}\right) \tag{11}$$

- Final Output:

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t \tag{12}$$

where h_t represents the final output of the ConvGRU, reinforcing the temporal dependencies among latent patches. This combined approach enables DynamicGPT to model complex temporal sequences while preserving spatial integrity and incorporating uncertainty—a significant advancement over previous approaches.



Smaller Kernel Sizes: These are effective at capturing fine details and small-scale features. They are useful for analyzing localized variations

Larger Kernel Sizes: Larger kernels are better at identifying larger-scale patterns and trends. They can integrate information over a broader area, providing a more global view.

Figure S3: Architecture of the Multi-Kernel Encoder-Decoder model for spatiotemporal data processing. The model utilizes various kernel sizes to capture features at different scales: smaller kernels for local features, mid-scale kernels for intermediate features, and larger kernels for global patterns. The encoding process analyzes past fields (input) to derive latent representations, while the decoding process predicts future fields. This architecture enhances the model's ability to integrate detailed and large-scale information effectively.

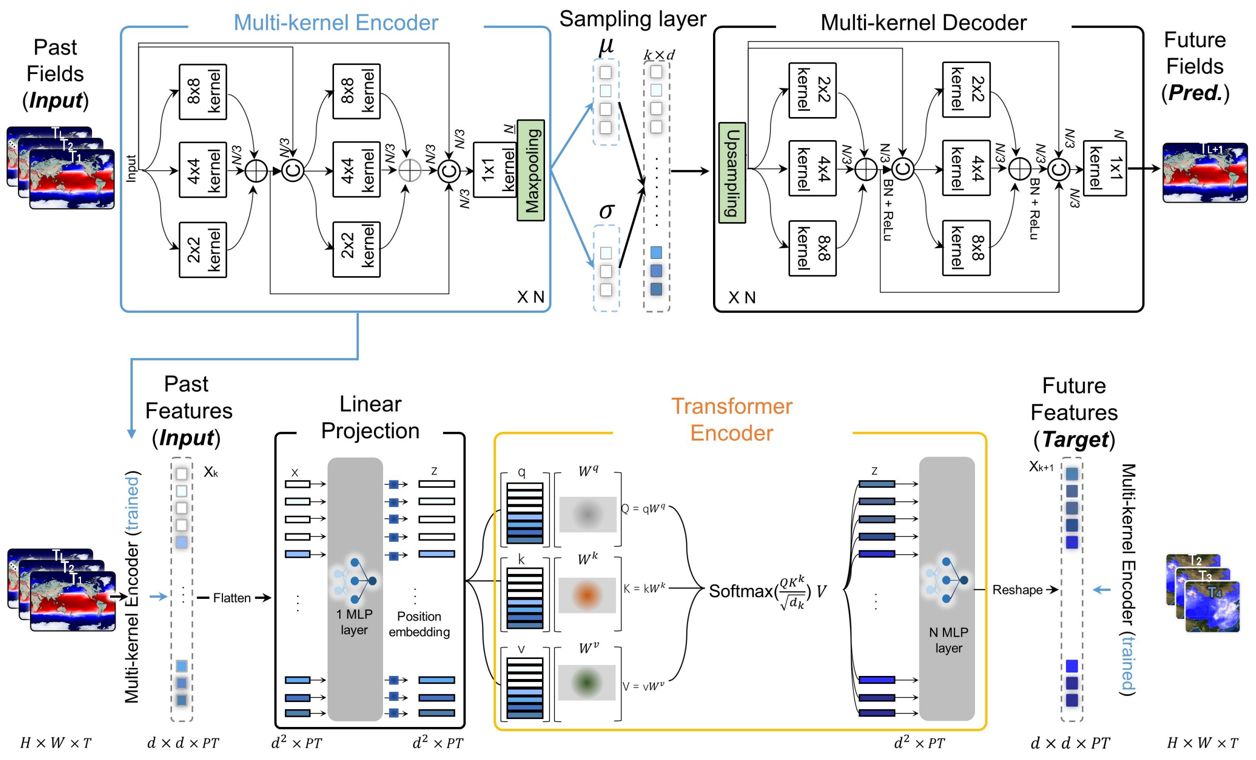


Figure S4: Architecture of the Multi-Kernel Encoder-Decoder model with Transformer integration for spatiotemporal data processing. The model consists of a multi-kernel encoder that extracts features from past fields (input) using various kernel sizes, followed by a sampling layer that captures latent representations. The Transformer encoder processes these features through a linear projection, allowing for effective learning of relationships. The multi-kernel decoder then generates future fields (predictions) based on the encoded information, enhancing the model's ability to predict future dynamics.

B.3 DynamicGPT Training Strategy

As shown in **Figure S5**, DynamicGPT's training employs a split network training approach. This approach separates the training of the multi-kernel encoder and the temporal modeling network, addressing the challenge of scale differences between components that can complicate convergence and hinder learning efficiency. By training these components independently, DynamicGPT achieves better convergence rates and optimizes parameter learning more effectively. This strategy is grounded in the need for stability during training and the goal of utilizing model parameters more efficiently. In other words, this split training method is not only practical computaional efficiency but also essential for maximizing model performance.

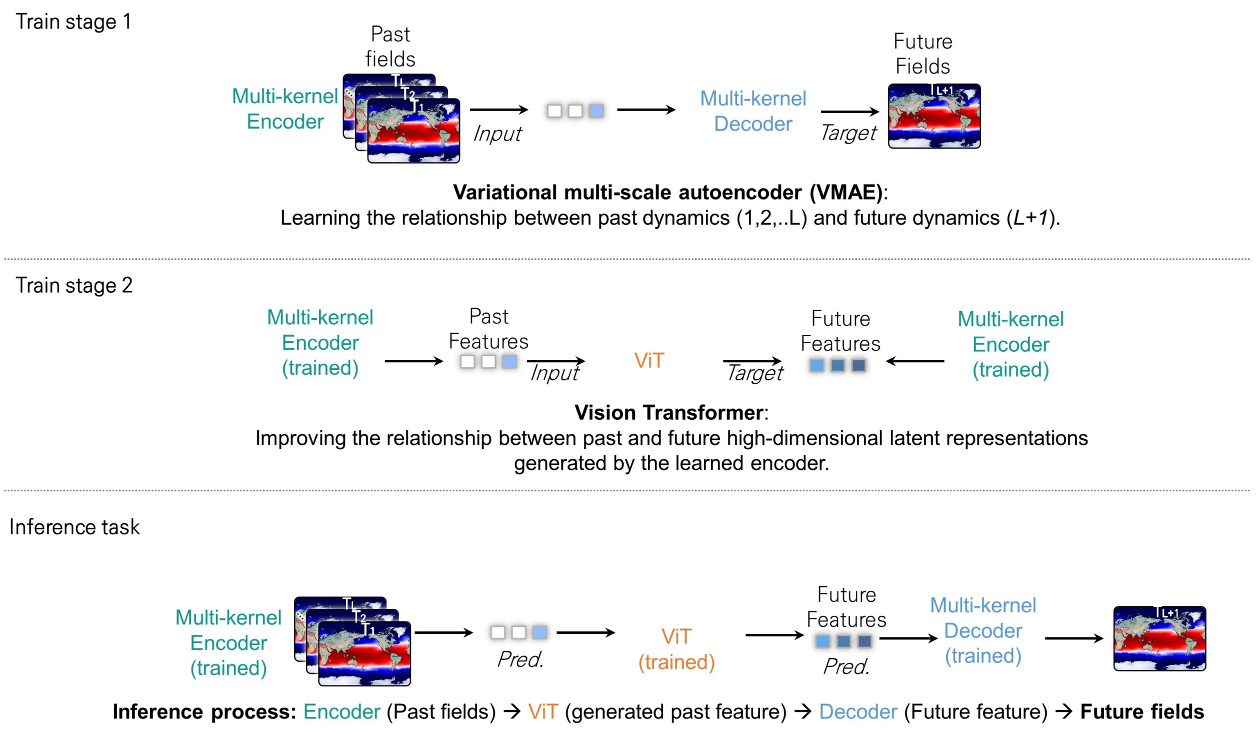

Train stage 1
Past fields
Multi-kernel Encoder
Input
Multi-kernel Decoder
Target
Future Fields
Variational multi-scale autoencoder (VMAE):
Learning the relationship between past dynamics (1,2,..L) and future dynamics (L+1).
Train stage 2
Multi-kernel Encoder (trained)
Past Features
Input
ViT
Target
Future Features
Multi-kernel Encoder (trained)
Vision Transformer:
Improving the relationship between past and future high-dimensional latent representations generated by the learned encoder.
Inference task
Multi-kernel Encoder (trained)
Pred.
ViT (trained)
Future Features
Pred.
Multi-kernel Decoder (trained)
Inference process: Encoder (Past fields) → ViT (generated past feature) → Decoder (Future feature) → Future fields

Figure S5: Overview of the training stages and inference process in the model architecture. Train Stage 1 involves the Variational Multi-Scale Autoencoder (VMAE), which learns the relationship between past dynamics (T_1, T_2, ..., T_L) and future dynamics (T_{L+1}) using a multi-kernel encoder and decoder. Train Stage 2 utilizes the Vision Transformer (ViT) to improve the relationship between high-dimensional past and future features generated by the trained multi-kernel encoder. The Inference Task illustrates how the trained model processes past fields to generate predictions, utilizing the ViT and multi-kernel decoder to output future fields.

B.4 Summary of the optimized DynamicGPT's archtectures

To provide a comprehensive understanding of how DynamicGPT apply to various real-world problems, we highlighted the impact of different hyperparameters (e.g. beta-Parameter, Latent dimension, Head layer, Kernel operation) on model performancein the section **Guidelines for Tuning DynamicGPT Across Diverse Spatiotemporal Dynamics** of the **Main Text**. In this supplementary information, we summarize the final architecture details, including the layers and feature maps, for each validation scenario (**Figure S6**). This overview is essential for demonstrating the adaptability and robustness of the model across different applications.

In the context of validating DynamicGPT, we investigated the performance across four representative real-world problems: crack propagation in materials, 3D reaction-diffusion processes, flow past a cylinder, and climate science predictions. Each scenario presents unique challenges and requires tailored network configurations to achieve optimal performance. By systematically adjusting and analyzing the hyperparameters in the multi-spatial embedding network ($\boldsymbol{E}$), multi-scale temporal network ($\boldsymbol{D}$), multi-scale spatial decoder ($\boldsymbol{F}$), and the discriminator, we identified the configurations that yielded the most accurate predictions.

The provided supplementary figure serves as a detailed reference, illustrating how each network component and feature map size is specifically configured for the scenarios tested. For example, the multi-spatial embedding network ($\boldsymbol{E}$) is tailored with varying numbers of encoding layers and feature dimensions depending on the input data structure and problem complexity. Similarly, the multi-scale temporal network ($\boldsymbol{D}$) incorporates self-attention mechanisms and position layers to handle different temporal dependencies effectively.

By presenting this structured summary, we aim to make it clear why these particular architectural choices were made and how they contribute to the performance of DynamicGPT in diverse spatiotemporal prediction tasks. This supplementary information underscores the thorough experimentation process and provides clarity on the adjustments made for each validation dataset

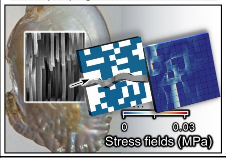
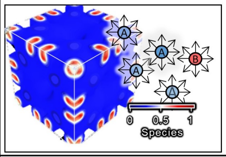
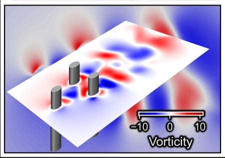
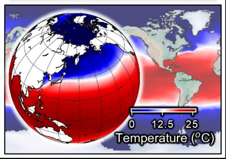
Validation Scenarios	Multi-spatial embedding network (*E*)		Multi-scale temporal network (*D*)		Multi-scale spatial decoder (*F*)		Discriminator	
Crack propagation in materials 	*Input layer*	(None, 132, 132, 3)	*Input*	(None, 16, 16, 16, 3)	*Decode layer 1*	(None, 16, 16, 16)	*Concatenate*	(None, 132, 132, 2)
	Encode layer 1	(None, 66, 66, 32)	*Embed layer*	(None, 3, 2048)	*Decode layer 2*	(None, 33, 33, 32)	*Conv2D*	(None, 66, 66, 64)
	Encode layer 2	(None, 33, 33, 32)	*Position layer*	(None, 3, 2048)	*Decode layer 3*	(None, 66, 66, 32)	*Conv2D*	(None, 33, 33, 128)
	Encode layer 3	(None, 16, 16, 16)	*Self-attention*	(None, 3, 2048)	*Decode layer 4*	(None, 132, 132, 32)	*Conv2D*	(None, 16, 16, 256)
			Forward layer	(None, 3, 2048)	*Output layer*	(None, 132, 132, 1)	*Conv2D*	(None, 15, 15, 512)
			Forward layer	(None, 1, 2048)			*Output layer*	(None, 14, 14, 1)
			Reshape layer	(None, 16, 16, 16)				
3D Reaction-diffusion process 	*Input layer*	(None, 80, 80, 80, 30)	*Input*	(None, 8, 8, 16, 30)	*Dense + Reshape*	(None, 8, 8, 8, 16)	*Concatenate*	(None, 80, 80, 80, 2)
	Encode layer 1	(None, 40, 40, 40, 64)	*Embed layer*	(None, 30, 1024)	*Upsampling*	(None, 10, 10, 10, 16)	*Conv3D*	(None, 40, 40, 40, 64)
	Encode layer 2	(None, 20, 20, 20, 32)	*Position layer*	(None, 30, 1024)	*Decode layer 1*	(None, 20, 20, 20, 32)	*Conv3D*	(None, 20, 20, 20, 128)
	Encode layer 3	(None, 10, 10, 10, 16)	*Self-attention*	(None, 30, 1024)	*Decode layer 2*	(None, 40, 40, 40, 64)	*Conv3D*	(None, 10, 10, 10, 256)
	Encode layer 4	(None, 8, 8, 8, 16)	*Forward layer*	(None, 30, 1024)	*Decode layer 3*	(None, 80, 80, 80, 64)	*Conv3D*	(None, 9, 9, 9, 512)
	Average pooling	(None, 8, 8, 16)	*Forward layer*	(None, 1, 1024)	*Output layer*	(None, 80, 80, 80, 1)	*Output layer*	(None, 8, 8, 8, 1)
			Reshape layer	(None, 8, 8, 16)				
Flow past cylinder 	*Input layer*	(None, 256, 384, 10)	*Input*	(None, 4, 4, 64, 10)	*Decode layer 1*	(None, 8, 12, 64)	*Concatenate*	(None, 256, 384, 2)
	Encode layer 1	(None, 128, 192, 64)	*Embed layer*	(None, 10, 1024)	*Decode layer 2*	(None, 16, 24, 64)	*Conv2D*	(None, 128, 192, 64)
	Encode layer 2	(None, 64, 96, 64)	*Position layer*	(None, 10, 1024)	*Decode layer 3*	(None, 32, 48, 64)	*Conv2D*	(None, 64, 96, 128)
	Encode layer 3	(None, 32, 48, 64)	*Self-attention*	(None, 10, 1024)	*Decode layer 4*	(None, 64, 96, 64)	*Conv2D*	(None, 32, 48, 256)
	Encode layer 4	(None, 16, 24, 64)	*Forward layer*	(None, 10, 1024)	*Decode layer 5*	(None, 128, 192, 64)	*Conv2D*	(None, 31, 47, 512)
	Encode layer 5	(None, 8, 12, 64)	*Forward layer*	(None, 1, 1024)	*Output layer*	(None, 256, 384, 1)	*Output layer*	(None, 30, 46, 1)
	Encode layer 6	(None, 4, 4, 64)	*Reshape layer*	(None, 4, 4, 64)				
Climate science 	*Input layer*	(None, 180, 360, 12)	*Input*	(None, 2, 2, 64, 12)	*Decode layer 1*	(None, 3, 6, 64)	*Concatenate*	(None, 180, 360, 2)
	Encode layer 1	(None, 90, 180, 64)	*Embed layer*	(None, 10, 256)	*Decode layer 2*	(None, 6, 12, 64)	*Conv2D*	(None, 90, 180, 64)
	Encode layer 2	(None, 45, 90, 64)	*Position layer*	(None, 10, 256)	*Decode layer 3*	(None, 12, 23, 64)	*Conv2D*	(None, 45, 90, 128)
	Encode layer 3	(None, 23, 45, 64)	*Self-attention*	(None, 10, 256)	*Decode layer 4*	(None, 23, 45, 64)	*Conv2D*	(None, 23, 45, 256)
	Encode layer 4	(None, 12, 23, 64)	*Forward layer*	(None, 10, 256)	*Decode layer 5*	(None, 45, 90, 64)	*Conv2D*	(None, 22, 44, 512)
	Encode layer 5	(None, 6, 12, 64)	*Forward layer*	(None, 1, 256)	*Decode layer 6*	(None, 90, 180, 64)	*Output layer*	(None, 21, 43, 1)
	Encode layer 6	(None, 3, 6, 64)	*Reshape layer*	(None, 2, 2, 64)	*Output layer*	(None, 180, 360, 1)		
	Encode layer 7	(None, 2, 2, 64)						

Figure S6: DynamicGPT's architecture for various validation scenarios. This table summarizes the layers and dimensions used in the Multi-spatial Embedding Network ($\boldsymbol{E}$), Multi-scale Temporal Network ($\boldsymbol{D}$), Multi-scale Spatial Decoder ($\boldsymbol{F}$), and Discriminator across different tasks, including crack propagation in materials, the 3D reaction-diffusion process, flow past a cylinder, and climate science. Each scenario is detailed with the input and output dimensions, highlighting the complexity and design of the neural network components tailored for specific spatiotemporal challenges.

C Validation Against Baseline Models

To thoroughly assess the performance of DynamicGPT, we conducted a comprehensive comparison against established baseline models, including Physics-Informed Neural Networks (PINNs)[1], Transformers[2], and Deep Generative Models (DGMs)[3]. Each of these models has been widely utilized in spatiotemporal dynamic prediction tasks, and their selection as baselines allows for a robust evaluation of DynamicGPT's advantages.

C.1 Physics-Informed Neural Networks (PINNs)

PINNs integrate physical governing equations directly into the neural network's loss function (Left in **Figure S7**), enabling the model to enforce known physical laws during training. This mechanism allows PINNs to model spatiotemporal phenomena by embedding partial differential equations (PDEs) that describe system dynamics. The advantage of PINNs lies in their ability to maintain physical consistency and provide interpretability. However, their performance is often limited by the need for accurate mathematical descriptions of complex systems and high computational costs, which can hinder real-time prediction capabilities.

Why use PINNs as a baseline? PINNs are included as a baseline to highlight DynamicGPT's ability to achieve comparable or superior accuracy without the constraints of embedding explicit physical equations. This comparison underscores DynamicGPT's flexibility and efficiency when applied to complex systems where exact physical models are difficult to derive or computationally intensive to implement.

C.2 Transformer

Transformers, known for their self-attention mechanisms, excel at modeling long-range dependencies within sequential data. In spatiotemporal dynamic predictions, they provide a way to capture relationships across time steps while processing spatial features. A common approach involves using an autoencoder architecture to effectively reduce the high-dimensional spatiotemporal data into lower-dimensional latent vectors (Center in **Figure S7**). The encoder compresses the input data into a compact 1-dimensional vector representation, preserving essential temporal and spatial features. This dimensionality reduction is critical for making the subsequent dynamic modeling more computationally efficient. The self-attention mechanism within the Transformer then processes these 1-dimensional vectors, enabling the model to weigh different temporal relationships and capture diverse patterns. This allows for sophisticated dynamic modeling over extended sequences. However, despite these strengths, standard Transformers can suffer from spatial information loss when dealing with high-dimensional spatiotemporal data due to their focus on 1-dimensional representations and single-scale processing.

Why use Transformers as a baseline? Including Transformers as a baseline allows us to demonstrate DynamicGPT's ability to overcome limitations related to spatial information loss. While Transformers excel at managing long-term temporal dependencies, the comparison highlights how DynamicGPT's multi-scale feature extraction and vision-based temporal modeling can lead to more accurate and detailed spatiotemporal predictions by preserving rich spatial details alongside temporal modeling.

C.3 Deep Generative Models (DGMs)

Deep Generative Models, especially those built upon conditional GANs (cGANs), are well-known for their capability to learn complex distributions and generate realistic samples. In our DGM baseline, an encoder maps the input data to a latent space representation, followed by ConvLSTM layers that model temporal dependencies. The decoder reconstructs the final spatiotemporal output from the latent space (Right in **Figure S7**). This mechanism allows for modeling uncertainty and non-linear interactions in data. However, due to the sequential structure involving encoder-ConvLSTM-decoder modules, errors can propagate through the model, impacting the stability and accuracy of predictions. In addition, the combined use of cGAN, ConvLSTM, and multiple ecoding layers leads to significant computational expense. The high complexity can hinder real-time performance, particularly with large spatiotemporal datasets.

Why use DGMs as a baseline?: DGMs are included to demonstrate DynamicGPT's advantages in managing complex spatiotemporal dynamics without the drawbacks of excessive computational cost and instability associated with generative models. This comparison showcases how DynamicGPT's structured approach, which incorporates probabilistic modeling and multi-scale feature extraction, achieves stable training and effective inference, overcoming DGM limitations.

C.4 Hyperparameter setting

For all baseline models, hyperparameters were selected based on reported optimal configurations in peer-reviewed studies:

- **PINN**: **https://github.com/stephenbaek/parc**
- **Transformer**: **https://github.com/KTH-FlowAI/beta-Variational-autoencoders-and-transformers-for-reduced-order-modelling-of-fluid-flows**

- **DGM**: **https://github.com/google-deepmind/deepmind-research /tree/master/now-casting**

This ensures that the comparison remains fair and rooted in previous validation studies that have rigorously tested these architectures. Each model was tuned to achieve its best performance in similar spatiotemporal tasks, making the comparison with DynamicGPT reflective of real-world applicability.

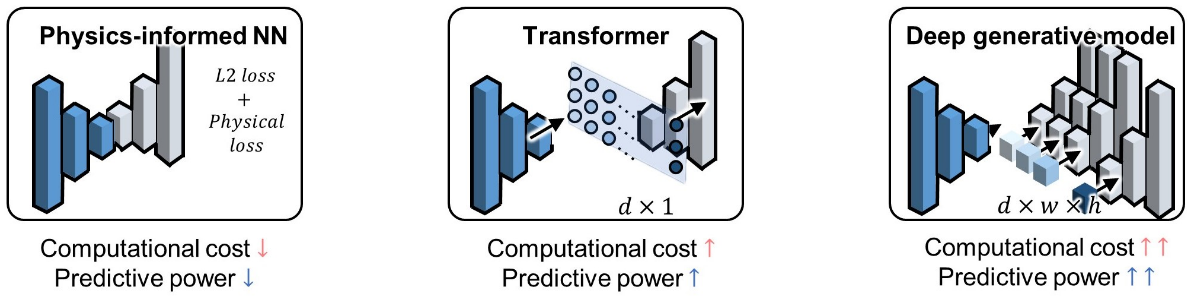


Figure S7: Baseline model architectures: Physics-informed Neural Networks (PINN), Transformer, and Deep Generative Model.

D Detailed Methods for Assessing the Physical Consistency and Computational Efficiency of DynamicGPT

In modern machine learning, evaluating the physical consistency and computational efficiency of predictive models is crucial, especially in domains that involve complex spatiotemporal dynamics. The **Performance Assessment of DynamicGPT: Physical Consistency and Computational Efficiency** section in the **Main Text** underscores the importance of these metrics and highlights DynamicGPT's superior performance compared to baseline models. This section provides a comprehensive explanation of the methods and code used for this assessment, supplementing the main text.

D.1 Methodology for Assessing Physical Consistency

Physical consistency ensures that model predictions adhere to the governing physical equations of the system. For the composite design scenario, we evaluated physical consistency by predicting the stress evolution along different directions and calculating the gradient values (∇) using finite difference methods to verify convergence to zero, satisfying the equilibrium equations of force.

To compute directional derivatives and gradients, we utilized Python libraries like Korina and Torch, which support finite difference-based differentiation. This enabled us to develop a function that calculates deviations from the force equilibrium equations for both simulation-based and model-predicted results (**Figure S8**). The following code snippet demonstrates how this function is structured:

```python
def ForceEquilibriumCalculation(stress_data):
    # Input data: batch_size * output_steps * stress components * H * W
    # stress_data is the results of 'stress tensor' from a model

    sigma_xx = stress_data[:, :, 0]  # xx stress component (shape: B * T * H * W)
    sigma_yy = stress_data[:, :, 1]  # yy stress component (shape: B * T * H * W)
    sigma_xy = stress_data[:, :, 2]  # xy stress component (shape: B * T * H * W)
    sigma_yx = stress_data[:, :, 3]  # yx stress component (shape: B * T * H * W)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Convert numpy arrays to torch tensors and move to device
    sigma_xx = torch.from_numpy(sigma_xx).float().to(device)
    sigma_yy = torch.from_numpy(sigma_yy).float().to(device)
    sigma_xy = torch.from_numpy(sigma_xy).float().to(device)
    sigma_yx = torch.from_numpy(sigma_yx).float().to(device)

    # Add a batch and channel dimension -> (H, W) -> (1, 1, H, W)
    sigma_xx = sigma_xx.unsqueeze(0).unsqueeze(0)
    sigma_yy = sigma_yy.unsqueeze(0).unsqueeze(0)
    sigma_xy = sigma_xy.unsqueeze(0).unsqueeze(0)
    sigma_yx = sigma_yx.unsqueeze(0).unsqueeze(0)

    # Sobolev-function gradients (Kornia's Spatial Gradient)
    field_grad = kornia.filters.SpatialGradient()

    # Calculate gradients
    sigma_xx_grad = field_grad(sigma_xx)
    sigma_yy_grad = field_grad(sigma_yy)
    sigma_xy_grad = field_grad(sigma_xy)
    sigma_yx_grad = field_grad(sigma_yx)

    # Extract x and y components of gradients
    sigma_xx_x = sigma_xx_grad[:, :, 0]  # sigma_xx's x-gradient
    sigma_xy_y = sigma_xy_grad[:, :, 1]  # sigma_xy's y-gradient

    sigma_yy_y = sigma_yy_grad[:, :, 1]  # sigma_yy's y-gradient
    sigma_yx_x = sigma_yx_grad[:, :, 0]  # sigma_yx's x-gradient

    # Force equilibrium equations
    rho = 1.0  # Assume density is 1 for simplicity
    bx, by = 0.0, 0.0  # External forces (e.g., gravity), can be set to non-zero
        if needed

    # x-direction force equilibrium
    force_x_eq = sigma_xx_x + sigma_xy_y + rho * bx

    # y-direction force equilibrium    force_y_eq = sigma_yy_y + sigma_yx_x + rho
        * by

    # Compute mean absolute value of the violations in force equilibrium
    force_x_mean = torch.mean(torch.abs(force_x_eq)).item()
    force_y_mean = torch.mean(torch.abs(force_y_eq)).item()
    f = force_x_mean + force_y_mean
    return f
```

This method was applied to both simulation results and predictions from DynamicGPT and baseline models to assess physical consistency. To test DynamicGPT's robustness and generalizability, we evaluated its performance on 1,000 composite configurations with strength and toughness averages 1.35 and 1.42 times higher than those in the training set. This strategy addresses a common challenge where conventional models struggle to predict unseen configurations accurately, especially when trained on limited data. For the unsteady flow problem, we used a similar approach to assess mass conservation by calculating the divergence of the predicted flow field and verifying its convergence to zero (**Figure S8**). DynamicGPT was trained on a 256 × 256 spatial domain and tested on a larger 1,536 × 256 domain to assess its generalization to unseen, larger computational domains, a critical factor for real-world applications. The code snippet below outlines the divergence calculation:

```python
def DivergenceCalculation(data):
    # Input data: batch_size * output_steps * flow length * H * W
    # The data is the results by 'DynamicGPT and Baseline models'

    preds_u = data[:, :, 0]  # Extract u field (shape: B * T * H * W)
    preds_v = data[:, :, 1]  # Extract v field (shape: B * T * H * W)

    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Convert numpy arrays to torch tensors and move to device
    u = torch.from_numpy(preds_u).float().to(device)
    v = torch.from_numpy(preds_v).float().to(device)

    # Add a batch and channel dimension -> (H, W) -> (1, 1, H, W)
    u = u.unsqueeze(0).unsqueeze(0)  # Add batch and channel dimensions
    v = v.unsqueeze(0).unsqueeze(0)  # Add batch and channel dimensions

    # Sobolev-function gradients (Kornia's Spatial Gradient)
    # Note: the operation use "Central differential method"!
    field_grad = kornia.filters.SpatialGradient()

    # Calculate gradients
    u_grad = field_grad(u)
    v_grad = field_grad(v)

    # Extract x and y components of gradients
    u_x = u_grad[:, :, 0]  # u's x-gradient
    v_y = v_grad[:, :, 1]  # v's y-gradient

    # Compute divergence (u_x + v_y)
    div = u_x + v_y

    # Convert back to numpy and compute mean of the absolute divergence
    div_mean = np.mean(np.abs(div.cpu().data.numpy()), axis=(0, 2, 3))  # Mean
        over H, W

    return div_mean
```

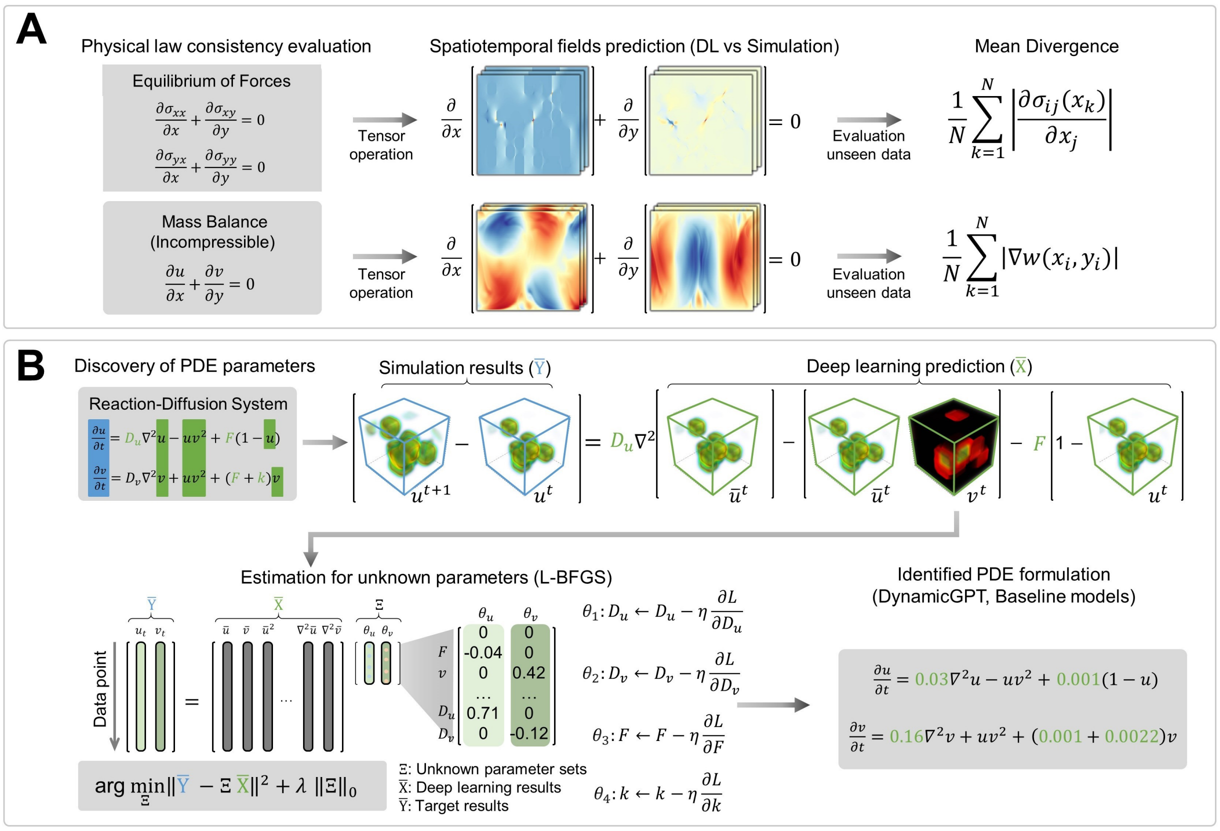

A
Physical law consistency evaluation
Equilibrium of Forces
$\frac{\partial \sigma_{xx}}{\partial x}+\frac{\partial \sigma_{xy}}{\partial y}=0$
$\frac{\partial \sigma_{yx}}{\partial x}+\frac{\partial \sigma_{yy}}{\partial y}=0$
Mass Balance (Incompressible)
$\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}=0$
Tensor operation
Spatiotemporal fields prediction (DL vs Simulation)
Evaluation unseen data
Mean Divergence
$\frac{1}{N}\sum_{k=1}^{N}\left|\frac{\partial \sigma_{ij}(x_k)}{\partial x_j}\right|$
$\frac{1}{N}\sum_{k=1}^{N}|\nabla w(x_i,y_i)|$
B
Discovery of PDE parameters
Reaction-Diffusion System
$\frac{\partial u}{\partial t}=D_u\nabla^2u-uv^2+F(1-u)$
$\frac{\partial v}{\partial t}=D_v\nabla^2v+uv^2+(F+k)v$
Simulation results ($\bar{Y}$)
Deep learning prediction ($\bar{X}$)
Estimation for unknown parameters (L-BFGS)
Data point
$\arg\min_{\Xi}\|\bar{Y}-\Xi\bar{X}\|^2+\lambda\|\Xi\|_0$
Ξ: Unknown parameter sets
X̄: Deep learning results
Ȳ: Target results
$\theta_1: D_u \leftarrow D_u-\eta\frac{\partial L}{\partial D_u}$
$\theta_2: D_v \leftarrow D_v-\eta\frac{\partial L}{\partial D_v}$
$\theta_3: F \leftarrow F-\eta\frac{\partial L}{\partial F}$
$\theta_4: k \leftarrow k-\eta\frac{\partial L}{\partial k}$
Identified PDE formulation (DynamicGPT, Baseline models)
$\frac{\partial u}{\partial t}=0.03\nabla^2u-uv^2+0.001(1-u)$
$\frac{\partial v}{\partial t}=0.16\nabla^2v+uv^2+(0.001+0.0022)v$

Figure S8: (**a**) Flowchart of physical Consistency evaluation, (**b**) Flowchart of PDE parameter estimation

D.2 Comprehensive Evaluation of Physical Consistency for Real-world Validation Sets

Equilibrium of Force

To evaluate the equilibrium of force, we calculated and averaged the stress evolution over 47 time steps for each model in the test set (**Supplementary Table 1**). DynamicGPT achieved a score of 0.0083, closely aligning with the simulation benchmark of 0.0072, highlighting its superior ability to maintain physical consistency in stress evolution. In comparison, conventional models such as PINN, Transformer, and DGM showed greater deviation from the simulation with scores of 0.0181, 0.0148, and 0.0125, respectively.

For a more detailed analysis, we focused on the configuration with the highest strength in the test set—representing the most challenging case and the primary objective of exploring unseen spaces in mechanical design. **Figure S9** illustrates the results for composite design, showing the equilibrium of forces over time. DynamicGPT's predictions closely follow the simulation curve, demonstrating superior physical consistency even under extreme conditions. In contrast, other models such as PINN, DGM, and Transformer exhibit more significant deviations, particularly at later time steps, indicating a loss of accuracy when dealing with high-strength, unseen configurations. The ability to predict the stress evolution in configurations with unprecedented mechanical properties is crucial for advancing material design and engineering practices. These results underline the importance of having models capable of accurately extrapolating to such high-strength designs. By maintaining physical consistency and accurately modeling the long-term stress evolution, DynamicGPT proves to be not only an effective predictive tool but also a potentially transformative asset for real-world engineering, where predicting behavior under novel and extreme conditions is vital for safety and innovation. These findings suggest that while previous models can approximate physical consistency, DynamicGPT's architecture, which integrates multi-scale spatial and temporal modeling with probabilistic components, significantly enhances its ability to capture the fine details necessary for adhering to the governing force equilibrium equations.

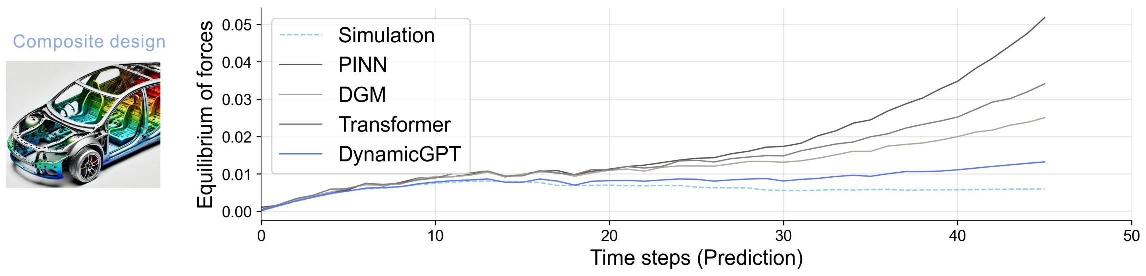


Figure S9: Comparative analysis of physical consistency in predictions made by different models. The equilibrium of forces over time steps is calculated by baseline models, DynamicGPT, and Simulation. DynamicGPT's predictions align more closely with the simulation results, showcasing its ability to maintain physical consistency even in high-strength configurations that were not present in the training data.

Mass Balance

For the mass balance assessment, DynamicGPT also demonstrates superior performance with a deviation of 20.8, compared to the benchmark simulation value of 15.5. While this result is not as close to the simulation as in the case of the force equilibrium, it is notably better than the scores of other models, such as DGM (27.5), Transformer (42.9), and PINN (36.6). This indicates that DynamicGPT is not only capable of maintaining force equilibrium but also effectively manages the conservation of mass in unsteady flow scenarios (**Figure S10**).

D.3 Comparative Analysis and Implications

The results in **Supplementary Table 1** indicate that DynamicGPT consistently outperforms baseline models in physical law adherence for both force equilibrium and mass balance. This performance is attributed to its advanced architecture, which integrates multi-scale feature extraction and temporal modeling while

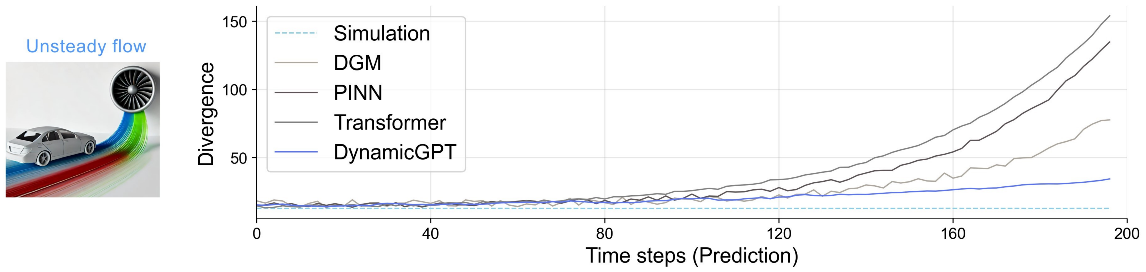


Figure S10: Comparative analysis of physical consistency in predictions made by different models. The divergence over time steps is shown for each model, including PINN, DGM, Transformer, and the proposed DynamicGPT. DynamicGPT demonstrates lower divergence and maintains closer adherence to the simulation benchmark, indicating superior mass conservation in unsteady flow predictions.

incorporating physical constraints. Unlike conventional models that struggle with extrapolating to unseen scenarios and maintaining physical consistency, DynamicGPT demonstrates a clear advantage.

These results highlight the potential of DynamicGPT to bridge the gap between traditional computational simulations and data-driven models. Achieving near-simulation-level accuracy in long-term spatiotemporal predictions, DynamicGPT showcases its capability for reliable and computationally efficient predictions, reinforcing its role as a powerful tool in applications where physical consistency is critical.

In conclusion, these findings demonstrate that DynamicGPT has been designed and optimized to overcome the limitations of previous models, maintaining physical consistency in real-world validation scenarios. This capability suggests that DynamicGPT could augment or even replace traditional simulation methods for complex physical systems, paving the way for new applications in science and engineering.

Table S1: Summary of the evaluation of physical conservation laws across different models. The table compares the performance of various models (PINN, Transformer, DGM, DynamicGPT, and Simulation) in terms of two key metrics: Equilibrium of force and Mass balance. The DynamicGPT model is highlighted in blue, showing the best performance in both metrics, with the lowest values for Equilibrium of force and Mass balance compared to the other models.

Type	PINN	Transformer	DGM	**DynamicGPT**	Simulation
Equilibrium of force	0.0181	0.0148	0.0125	**0.0083**	0.0072
Mass balance	36.6	42.9	27.5	**20.8**	15.5

D.4 Methodology for PDE Parameter Estimation

To expand on the evaluation of physical conservation laws, we employed a method for estimating PDE (Partial Differential Equation) parameters based on the results of deep learning predictions. This approach is significant because it bridges the gap between data-driven predictions and theoretical physical models by allowing for the extraction and refinement of unknown PDE parameters, enhancing the interpretability and reliability of the model.

Process Overview

The process for PDE parameter estimation is depicted in **Figure S8(b)**, which illustrates the workflow for analyzing the reaction-diffusion system as validation scenario. The detailed process with Python code is as follow:

1. **Simulation Results (Y) and Deep Learning Predictions** ($\hat{X}$): We compare simulation results with predictions from models like DynamicGPT. These outputs approximate time derivatives and spatial features used for parameter estimation.

2. **Deriving Time Derivatives**: The time derivatives of the predicted fields, $\frac{\partial u}{\partial t}$ and $\frac{\partial v}{\partial t}$, are calculated using finite differences based on the simulation results:

```
u_pred = np.load('Simulation_u.npy')
v_pred = np.load('Simulation_v.npy')
u_t_pred = (u_pred[:, 1:] - u_pred[:, :-1]) / dt
v_t_pred = (v_pred[:, 1:] - v_pred[:, :-1]) / dt
```

3. **Initial Parameter Setup**: Initial guesses for PDE parameters are established and refined during training:

```
ini1 = np.random.uniform(-1, 1)
ini2 = np.random.uniform(-1, 1)
ini3 = np.random.uniform(-1, 1)
ini4 = np.random.uniform(-1, 1)

D_U = torch.tensor(ini1, requires_grad=True)
D_V = torch.tensor(ini2, requires_grad=True)
F = torch.tensor(ini3, requires_grad=True)
k = torch.tensor(ini4, requires_grad=True)
```

4. **Optimization Strategy**: We use an optimizer, such as LBFGS, for parameter updates:

```
optimizer = torch.optim.LBFGS([D_U, D_V, F, k], lr=1e-4)
```

5. **Laplacian Calculation**: The Laplacian operator for 3D spatial data is defined:

```
def laplacian_3d(U):
    return (
        -6 * U +
        torch.roll(U, 1, dims=2) + torch.roll(U, -1, dims=2) +
        torch.roll(U, 1, dims=3) + torch.roll(U, -1, dims=3) +
        torch.roll(U, 1, dims=4) + torch.roll(U, -1, dims=4)
    )
```

6. **Loss Function Definition**: The loss function compares predicted time derivatives (Right side of equation) with PDE-driven time derivatives (Left side of equation):

```
def gray_scott_loss_multi(U_pred, V_pred, U_t_pred, V_t_pred, D_U, D_V, F, k,
    lambda_reg=1e-3):
    loss_total = 0
    for i in range(U_pred.shape[0]):
        lap_U = laplacian_3d(U_pred[i:i+1])
        lap_V = laplacian_3d(V_pred[i:i+1])
        lap_U = lap_U[:, :-1]
        lap_V = lap_V[:, :-1]

        U_eqn = D_U * lap_U - U_pred[i:i+1, :-1] * V_pred[i:i+1, :-1]**2 + F *
            (1 - U_pred[i:i+1, :-1])
        V_eqn = D_V * lap_V + U_pred[i:i+1, :-1] * V_pred[i:i+1, :-1]**2 - (F
            + k) * V_pred[i:i+1, :-1]

        U_t_pred_sliced = U_t_pred[i:i+1, :-1]
        V_t_pred_sliced = V_t_pred[i:i+1, :-1]

        loss_data = torch.mean((U_t_pred_sliced - U_eqn)**2) + torch.mean((
            V_t_pred_sliced - V_eqn)**2)
        loss_total += loss_data

    loss_total /= U_pred.shape[0]
    return loss_total
```

7. **Training Loop**: The training loop iteratively updates parameters to minimize the loss. The learning rate are the number of update are 1e-04 and 1,200, respectively:

```
for epoch in range(1200):
    optimizer.zero_grad()
    loss = gray_scott_loss_multi(u_pred[:, :-1], v_pred[:, :-1], u_t_pred,
        v_t_pred, D_U, D_V, F, k)
    loss.backward()
    optimizer.step()

    if epoch % 2 == 0:
        print(f'Epoch␣{epoch},␣Loss:␣{loss.item():.6f}')
        print(f'Estimated␣D_U:␣{D_U.item():.6f},␣Actual␣D_U:␣{actual_DU}')
        print(f'Estimated␣D_V:␣{D_V.item():.6f},␣Actual␣D_V:␣{actual_DV}')
        print(f'Estimated␣F:␣{F.item():.6f},␣Actual␣F:␣{actual_F}')
        print(f'Estimated␣k:␣{k.item():.6f},␣Actual␣k:␣{actual_k}')
        print('-'*50)
```

D.5 Detailed Results on PDE Parameter Estimation

Figure S11 illustrates the comparative results of parameter estimation for the reaction-diffusion system across different models: PINN, DGM, and DynamicGPT. Each subplot represents the estimated values of the diffusion coefficients D_u and D_v, feed rate F, and decay rate k over training epochs.

1. D_u **Estimation**: DynamicGPT (blue line) demonstrates rapid convergence towards the target coefficient (dashed line), achieving stability and accurate estimation earlier than both PINN and DGM.

While DGM shows significant fluctuations and a slower approach, PINN struggles to align with the true value, indicating less robust estimation in this parameter.

2. D_v **Estimation**: For the diffusion coefficient D_v, DynamicGPT again shows a more consistent trajectory towards the target compared to other models. PINN exhibits considerable instability, and DGM, although it approaches the target value, does so with a more erratic pattern. This highlights DynamicGPT's ability to estimate this parameter with greater precision and reliability.

3. F **Estimation**: In the estimation of the feed rate F, DynamicGPT displays smooth convergence and aligns closely with the actual target value after the initial training phase. DGM reaches a closer estimate but with higher variance, while PINN diverges away from the target, showing its limitations in accurately estimating this parameter.

4. k **Estimation**: The decay rate k further underscores DynamicGPT's performance advantage. The model converges effectively and stabilizes near the target coefficient. In contrast, DGM initially oscillates significantly before slowly settling, and PINN demonstrates the least effective estimation, deviating throughout the training.

These comparative results underscore DynamicGPT's strength in reliably estimating reaction-diffusion parameters, maintaining accuracy across long-term spatiotemporal dynamic. The superior performance of DynamicGPT over PINN and DGM emphasizes its robustness and precision, positioning it as a powerful tool for predictive modeling and parameter discovery in complex spatiotemporal systems.

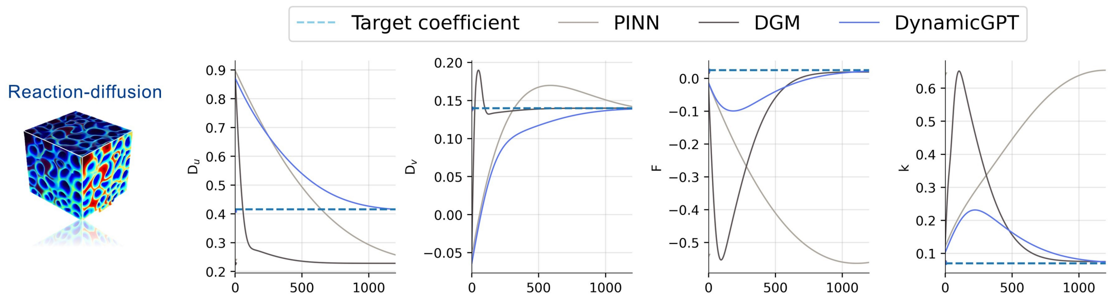


Figure S11: Comparative analysis of physical consistency in predictions made by different models. The divergence over time steps is shown for each model, including PINN, DGM, Transformer, and the proposed DynamicGPT. DynamicGPT demonstrates lower divergence and maintains closer adherence to the simulation benchmark, indicating superior mass conservation in unsteady flow predictions.

Table S2: Summary of the results of parameter estimation for partial differential equations (PDEs). The table compares the estimated values of four key components (D_u, D_v, F, and k) obtained using three different models: PINN, DGM, and DynamicGPT. The true values of the unknown parameters are also included for comparison. Notably, the results obtained with DynamicGPT, highlighted in blue, show the closest estimates to the actual unknown parameter values.

Component	D_u	D_v	F	k
PINN	0.237	0.137	-0.54	0.0642
DGM	0.228	0.02	0.0753	0.075
DynamicGPT	**0.411**	**0.14**	**0.0243**	**0.071**
Unknown Parameters	**0.416**	**0.14**	**0.025**	**0.07**

E References

[1] Nguyen, P.C.H. et al. PARC: Physics-aware recurrent convolutional neural networks to assimilate meso scale reactive mechanics of energetic materials. Sci. Adv. 9, eadd6868 (2023).

[2] Solera-Rico, A. et al. β-Variational autoencoders and transformers for reduced-order modelling of fluid flows. Nat. Commun. 15, 1361 (2024).

[3] Ravuri, S. et al. Skilful precipitation nowcasting using deep generative models of radar. Nature 597, 672–677 (2021).