Automated Kernel Discovery Towards Understanding High-dimensional Bayesian Optimization

Anonymous Author(s)
Affiliation
Address
email

Abstract

Gaussian Process (GP) kernels are central to Bayesian optimization (BO), yet designing effective kernels for high-dimensional problems still relies on extensive manual engineering. Existing automated approaches struggle in high dimensions for two bottlenecks: their kernel search space is limited to additions and multiplications of base kernels, and LLM-based approaches require conditioning on raw observations, which becomes infeasible due to context-length limits and the difficulty of extracting meaningful patterns. We introduce **Kernel Discovery**, a LLM-driven evolutionary framework for high-dimensional BO that searches a broader kernel space beyond predefined composition rules and does not require conditioning on observations. Motivated by the observation that directly prompting an LLM to generate kernel code yields syntactically varied but functionally identical kernels, we adopt a two-stage approach: an LLM first proposes novel mathematical forms, then a second LLM call converts each form into validated, executable code. We also propose a leave-one-out continuous ranked probability score (LOO-CRPS) as a selection criterion that penalizes overfitted kernels. On five high-dimensional BO benchmarks, our method achieves an average rank of **1.2 out of 17**, outperforming competitive baselines. We further analyze the discovered kernels to identify which kernels lead to improvements in high-dimensional BO.

1 Introduction

Optimizing high-dimensional black-box functions spans a wide range of machine learning problems, including hyperparameter optimization [1, 2], neural architecture search [3, 4], biological sequence design [5, 6], and control tasks [7, 8]. Bayesian optimization (BO) is the de facto paradigm for black-box optimization, which iteratively selects promising candidates based on surrogate models fitted to observations so far [9, 10]. However, it often suffers from the curse of dimensionality, as large distances between observations make accurate surrogate modeling difficult and exacerbate overexploration toward the boundary of the search space.

Prior works in high-dimensional BO have exploited structural assumptions such as additive decompositions [11, 12, 13, 14], low-dimensional subspaces [15, 16, 17], or sparsity [18, 19]. However, these approaches perform well only on synthetic benchmarks with explicit underlying structures, which rarely hold in practice. Another family of methods employs trust regions to effectively search high-dimensional spaces [7, 20]. While locality achieves promising results across diverse tasks [5, 21, 22], it requires a large number of evaluations and may struggle to escape local optima.

Recently, Hvarfner et al. [23] and Xu et al. [24] showed that even with basic kernels, such as RBF or Matérn52 kernels, we can match or exceed specialized methods in high-dimensional BO by proper scaling of the lengthscale prior with dimensionality. Moreover, Oh et al. [25] and Doumont et al. [26] demonstrated that geometric input warpings (e.g., hypercylinder or hypersphere) with simple

Submitted to ICML Decision-Making from Offline Datasets to Online Adaptation (DEMO) Workshop. Do not distribute.

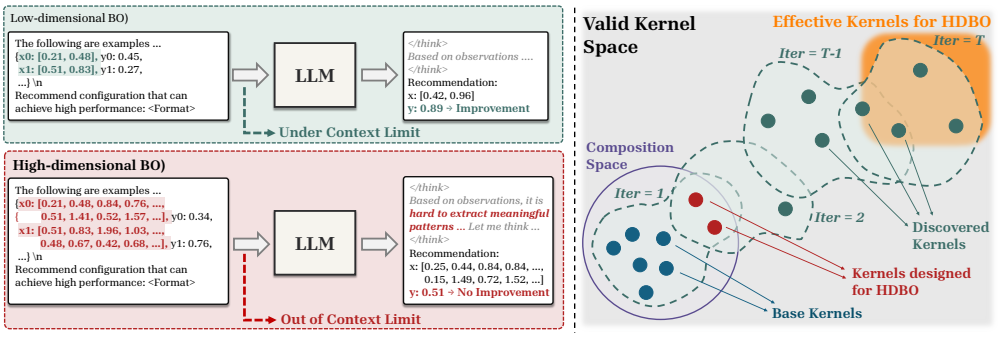


Figure 1: Motivation Figure. **(Left):** In high-dimensional BO, a conventional LLM-based BO does not work due to the out-of-context limits and the difficulty of extracting meaningful patterns. **(Right):** Our pipeline enables us to discover effective kernels for high-dimensional BO beyond compositions.

models can surpass sophisticated baselines on high-dimensional, real-world benchmarks. Although these breakthroughs are promising, identifying such effective priors or transformations still requires extensive manual engineering by domain experts. This suggests that automating kernel design with a system capable of mathematical reasoning could substantially reduce this barrier.

This naturally prompts us to leverage Large Language Models (LLMs, 27, 28, 29), which have internalized vast mathematical knowledge and can propose novel functional forms when incorporated into an evolutionary algorithm (EA) pipeline [30, 31, 32, 33, 34]. However, existing LLM-based BO methods do not directly scale to high-dimensional problems due to two bottlenecks. First, an **expressive bottleneck**: Existing kernel search methods are limited to additive and multiplicative compositions of base kernels, thereby limiting their ability to discover novel kernels suitable for high-dimensional BO. Second, an **interface bottleneck**: most LLM-based BO methods require conditioning on raw observations, which becomes infeasible in high dimensions due to context-length limits and the difficulty of extracting patterns from a long stream of numbers, as shown in Figure 1.

To address these limitations, we introduce **Kernel Discovery**, a novel framework for designing effective kernel structures for high-dimensional BO. First, we **initialize** the population with kernels tailored for high-dimensional BO. At each BO iteration, we **discover** new kernels through a two-stage approach: an LLM first proposes novel mathematical forms, and a second LLM converts the form into executable code. Through this decomposition, we encourage LLMs to explore the broader space of valid kernels by leveraging mathematical reasoning capabilities. We then **validate** the discovered kernels via agnostic execution and PSD checks, retaining only valid GP kernels. To **choose** the most promising kernel, we introduce LOO-CRPS, which is less prone to in-sample overfitting than the marginal log-likelihood metric. Finally, we evaluate the objective at the selected point and **update** the dataset and population, repeating until the evaluation budget is exhausted.

We conduct experiments on five high-dimensional BO benchmarks and achieve an average rank of **1.2 out of 17**, outperforming competitive baselines. We also conduct several ablation studies on the design choices of our framework. Finally, we analyze the discovered kernels and uncover that unexpected kernels (e.g., compositions of geometric warping or non-stationary components) might lead to improvements, providing insights into what makes a kernel effective in high dimensions.

2 Related Work

High-dimensional Bayesian Optimization. Several approaches have been proposed to push the limit of BO in high dimensions. One line of work exploits explicit structural assumptions, such as low-dimensional subspaces [15, 16, 17, 35], sparsity [18, 19], variable selection [7, 36, 37], and additive decompositions [11, 12, 13, 14]. However, such assumptions rarely hold in real-world problems. Another line of work employs trust regions, in which we constrain the search space to prevent the search among boundary points in high-dimensional space [5, 20, 22, 38, 39, 40]. While it achieves superior performance across various domains, particularly in scalable settings, it requires too large a number of evaluations and remains susceptible to being trapped in local optima.

More recently, a series of surprising findings have challenged longstanding intuitions about high-dimensional BO. Hvarfner et al. [23] and Xu et al. [24] showed that simple kernels with well-chosen

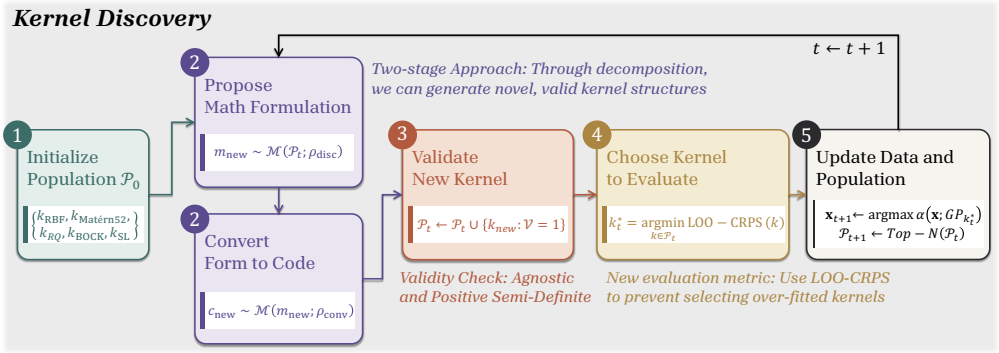


Figure 2: Overview of the method. Given an initial population, we instruct an LLM to generate novel kernels via a two-stage approach: mathematical formulation and code conversion. We then perform a sanity check to retain only valid kernels and select the kernel with the lowest LOO-CRPS value. Finally, we propose a next query using the chosen kernel and update the dataset and population.

lengthscale priors can match or exceed sophisticated methods. Papenmeier et al. [41] also claimed that scaling the initial lengthscale and random axis-aligned subspace perturbation sampling (RAASP, 42) can mitigate the vanishing gradient issue in acquisition function optimization in high dimensions. Doumont et al. [26] further demonstrated that the smoothness of surrogate models is a critical factor, and that spherical input mappings with a simple linear basis suffice for competitive performance. Together, these results reveal that the community's prevailing assumptions about what drives high-dimensional BO may be incomplete. This underscores the need for a more in-depth investigation of kernel design in high-dimensional BO, which is precisely the gap our work aims to fill.

LLMs for Black-box Optimization. Recent advancements in LLMs have demonstrated strong capabilities for complex reasoning [27, 28, 29], prompting several works to integrate them into the BO pipeline. Aglietti et al. [32] and Ngo et al. [43] utilize LLM to discover novel acquisition functions, while Li et al. [44] and Liu et al. [45] replace the entire BO loop with an LLM-driven system. However, these methods are primarily evaluated in low-dimensional settings and rely on prepending prior observations as context, which becomes infeasible in high dimensions due to context-length constraints. Furthermore, extracting meaningful structure from dense observations can be challenging even for recent, closed-source LLMs.

Most closely related to our work is CAKE [46], which employs an LLM to select and combine base kernels to improve BO surrogates. While it shares our motivation for LLM-guided kernel design, it also relies on prior trials as context and restricts the search space to additive and multiplicative compositions of base kernels. In contrast, our framework does not condition on raw observations and actively explores a broader space of valid kernels beyond compositions, thereby enabling more expressive surrogate modeling for high-dimensional problems.

3 Preliminaries

Bayesian Optimization. In BO, we aim to find an input $\mathbf{x} \in \mathcal{X}$ that maximizes the unknown black-box function $f : \mathcal{X} \to \mathbb{R}$, which is typically expensive to evaluate and not differentiable. BO can be modularized into three orthogonal components: a surrogate model, an acquisition function, and an optimizer for the acquisition function [47]. At each iteration, we fit the surrogate model to the observations so far and select the next query that maximizes the chosen acquisition function using the optimizer. While there are several possible options for each component, the conventional approach is to use a Gaussian Process (GP, 48) as a surrogate model, Expected Improvement (`EI`, 49) as an acquisition function, and `L-BFGS-B` [50] as an optimizer.

Gaussian Processes and Kernels. A GP defines a distribution over functions and is specified by a mean function (constant in most cases) and a covariance matrix $\mathbf{K}$. The covariance matrix can be derived by using a kernel function $k(\mathbf{x}, \mathbf{x}') : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$. The most frequently used kernel for BO is the RBF kernel [51], which can be defined as follows:

$$k_{\text{RBF}}(\mathbf{x}, \mathbf{x}') := \sigma^2 \exp\left(-\frac{1}{2}(\mathbf{x} - \mathbf{x}')^\top \mathbf{L}^{-1}(\mathbf{x} - \mathbf{x}')\right) \tag{1}$$

Kernel discovery Prompt

You are an expert in Gaussian process kernel design for high-dimensional Bayesian optimization.

TOP {n} kernels in our population:
Rank {i}({metric}: {score}, {lower, higher} is better)

Generate {n_discovery} DISTINCT, CREATIVE, mathematically innovative DISCOVERED kernels.

Each discovered kernel should be derived from one or more of the above kernels, but with meaningful mathematical changes.

[HDBO guidance prompt]
[Constraint guidance prompt]

OUTPUT FORMAT:
Return EXACTLY {n_ discovery} kernel descriptions.
{kernel description format}

Each MUST be fundamentally distinct (different transform, covariance structure, mathematical idea).
...

HDBO guidance Prompt

HIGH-DIMENSIONAL STRATEGIES (not limited to):

1. Input projections: stereographic, cylindrical, tanh warping, linear embedding to lower-dim subspace
2. Dimensionality scaling: scale distances by sqrt(D) to prevent distance collapse
3. Structural assumptions: additive decompositions, subspace embeddings, sparse attention/gating
4. Novel distance metrics: geodesic, learned metrics, projected distances

...

Constraint guidance Prompt

POSITIVE SEMI-DEFINITENESS (CRITICAL):

1. Safe building blocks: RBF, Matérn, RQ, linear k=x^{T}y, polynomial (x^{T}y+c)^p with positive coefficients
2. Safe operations: sum of PSD kernels, product of PSD kernels, $\phi(x)^T\phi(x')$ for any feature map ϕ
3. Risky: custom formulas not built from these - the system will reject non-PSD kernels

...

Figure 3: Prompts for Kernel Discovery. To discover novel kernels for high-dimensional BO, we start with a prompt that simulates the reasoning of a human expert. Then, we instruct the LLM to understand prior strategies for high-dimensional BO and the constraints it should satisfy.

where $\sigma^2 > 0$ is the output scale and $\mathbf{L} = \text{diag}(\ell_1^2, \ldots, \ell_d^2)$ is a diagonal matrix of per-dimension lengthscale hyperparameters $\ell_i > 0$.

Kernels can be combined or constructed in several ways. First, the sum and product of two valid kernels are also valid kernels, enabling the construction of richer covariance structures. Second, for any feature map $\phi : \mathcal{X} \to \mathbb{R}^m$, the inner product $k(\mathbf{x}, \mathbf{x}') = \phi(\mathbf{x})^\top \phi(\mathbf{x}')$ defines a positive semi-definite kernel by construction. By injecting these compositional and feature-map properties as inductive biases into an LLM-driven kernel construction process, we can discover novel kernel structures tailored for high-dimensional BO.

4 Methods

In this section, we introduce **Kernel Discovery**, a novel framework for discovering effective kernels for high-dimensional BO by leveraging LLMs as an evolutionary operator. We first initialize the population with well-known base kernels tailored for high dimensions. Then, we guide the LLM to generate novel kernel structures as mathematical forms and convert them into executable Python code. We conduct a sanity check to determine whether the generated kernel is valid. We also introduce a new selection criterion to choose the most promising kernel for acquisition function optimization. We repeat the process until convergence. We summarize the overview of our framework in Figure 2.

Notations. We represent each kernel $k = (m, c)$, where m is its mathematical expression and c is the corresponding code. Let $\mathcal{P}_t = \{k_1^{(t)}, \ldots, k_N^{(t)}\}$ denote the population of size N at iteration t. We denote by $\mathcal{M}(\cdot\,; \rho)$ a call to the LLM conditioned on a role-specific prompt ρ. We also introduce an auxiliary verifier $\mathcal{V}$ to check whether the discovered kernel is valid.

Initialize Population. Similar to several approaches that integrate LLMs into the evolutionary loop [31], we initialize the base population with existing kernels that are widely used for BO. As our focus is on high-dimensional BO, we need to carefully select base kernels that will encourage LLMs to extract useful knowledge from them and give insights into how to propose better kernels. To this end, we set the initial population as follows:

$$\mathcal{P}_0 = \{k_{\text{RBF}},\ k_{\text{Matérn52}},\ k_{\text{RQ}},\ k_{\text{BOCK}},\ k_{\text{SL}}\}, \tag{2}$$

For the RBF, Matérn52, and RQ, we use the lengthscale prior $\ell_i \sim \mathcal{LN}(\sqrt{2} + \log(\sqrt{D}), \sqrt{3})$, proposed by Hvarfner et al. [23]. BOCK [25] and SL [26] propose geometric mappings to avoid overexploration. We do not include Linear and Periodic as they perform poorly in high dimensions.

Discover Kernels with LLMs. Starting from the initial population $\mathcal{P}_0$, we leverage LLMs to propose new kernel structures. Specifically, our discovery pipeline consists of two stages: mathematical formulation and code generation.

While one can directly prompt an LLM to generate novel kernels in the code generation space, this is undesirable, as LLMs often produce functionally redundant kernel codes (e.g., by renaming variables or substituting `x-y` with `x.sub(y)`) rather than genuinely novel structures. To mitigate this, we first express the existing kernels in mathematical form $\{m_i^{(t)}\}_{i=1}^N$ and prompt the LLM to generate new mathematical forms that satisfy the constraints and are suitable for high-dimensional BO:

$$m_{\text{new}} \sim \mathcal{M}\Big(\{m_i^{(t)}\}_{i=1}^N;\, \rho_{\text{disc}}\Big) \tag{3}$$

where ρ_{disc} is a prompt for discovery stage as depicted in Figure 3. After discovering a new mathematical form, we introduce another LLM to convert the formulation into executable code:

$$c_{\text{new}} \sim \mathcal{M}(m_{\text{new}};\, \rho_{\text{conv}}) \tag{4}$$

where ρ_{conv} is a prompt for conversion stage as depicted in Figure 9. Rather than conditioning on raw observations, we provide only the mathematical forms from previous iterations as context. This dramatically reduces the context length, allowing the LLM to focus on structural novelty rather than sifting through high-dimensional numerical data for patterns.

Following prior works, we find that composing kernels is the simplest way to discover new valid kernels. We therefore add a composition stage in which the LLM takes the top-k kernels as input and combines them via addition or multiplication. You can find the prompt for composition in Figure 10.

Validate Discovered Kernels. Given a new candidate $k_{\text{new}} = (m_{\text{new}}, c_{\text{new}})$, we filter it with two empirical checks: that the code runs correctly across varying input shapes ($\mathcal{V}_{\text{agn}}$) and that the Cholesky decomposition succeeds on random test inputs ($\mathcal{V}_{\text{psd}}$). Formally:

$$\mathcal{V}(k_{\text{new}}) = \mathcal{V}_{\text{agn}}(k_{\text{new}}) \wedge \mathcal{V}_{\text{psd}}(k_{\text{new}}) \tag{5}$$

where $\mathcal{V}_{\text{agn}}$ tests the `forward` function on inputs of varying batch size and dimensionality, and $\mathcal{V}_{\text{psd}}$ checks for a successful Cholesky decomposition. Only candidates with $\mathcal{V}(k_{\text{new}}) = 1$ are added to the current population pool $\mathcal{P}_t$.

$$\mathcal{P}_t \leftarrow \mathcal{P}_t \cup \{k^{\text{new}} : \mathcal{V}(k^{\text{new}}) = 1\} \tag{6}$$

Choose the Promising Kernel. After generating new kernels, we need to select which kernel $k \in \mathcal{P}_t$ should be used for the acquisition function maximization. While the marginal log-likelihood (MLL) is a straightforward approach, it may prefer overly complex kernels that overfit the current observations and sacrifice predictive calibration. To this end, we introduce the Leave-one-out Continuous Ranked Probability Score (LOO-CRPS), a selection criterion that scores held-out predictive distributions and thereby penalizes overconfident fits more directly than MLL, defined as follows:

$$\text{CRPS}\big(\mathcal{N}(\mu, \sigma^2),\, y\big) = \sigma\left[\frac{y-\mu}{\sigma}\left(2\Phi\left(\frac{y-\mu}{\sigma}\right) - 1\right) + 2\,\phi\left(\frac{y-\mu}{\sigma}\right) - \frac{1}{\sqrt{\pi}}\right] \tag{7}$$

where ϕ and Φ are the standard normal PDF and CDF, respectively. While CRPS is more robust than MLL, it can still overfit when evaluated on training data. To alleviate this, we aggregate CRPS over leave-one-out predictives across all n observations:

$$\text{LOO-CRPS}(\mathbf{K}_k, \mathbf{y}) = \frac{1}{n}\sum_{i=1}^{n} \text{CRPS}\big(\mathcal{N}(\mu_{-i},\, \sigma_{-i}^2),\, y_i\big)\,, \quad \mu_{-i} = y_i - \frac{[\mathbf{K}_k^{-1}\mathbf{y}]_i}{[\mathbf{K}_k^{-1}]_{ii}},\; \sigma_{-i}^2 = \frac{1}{[\mathbf{K}_k^{-1}]_{ii}} \tag{8}$$

where $\mathbf{K}_k$ is a kernel matrix derived from the kernel k. By leveraging the standard LOO identity for GP, the predictive mean μ_{-i} and variance σ_{-i}^2 can be computed analytically from a single matrix inversion. At each round, we select

$$k_t^\star = \arg\min_{k \in \mathcal{P}_t} \text{LOO-CRPS}(\mathbf{K}_k, \mathbf{y}) \tag{9}$$

Finally, we use the selected kernel $k_t^\star$ and fit its hyperparameters (e.g., lengthscale, scale, variance), maximize the acquisition function $\alpha(\mathbf{x};\, \text{GP}_{k_t^\star})$ to obtain the next query point $\mathbf{x}_{t+1}$, evaluate the corresponding objective $y_{t+1} = f(\mathbf{x}_{t+1})$, and augment the dataset $\mathcal{D}_{t+1} \leftarrow \mathcal{D}_t \cup \{(\mathbf{x}_{t+1}, y_{t+1})\}$.

Table 1: Performance comparison of various methods across standard benchmarks. D denotes the dimensionality of the task. **Blue** denotes the best entry in the column, and **Violet** denotes the second best. Experiments are conducted with 4 random seeds.

Method	Rover (↑) ($D = 100$)	Mopta08 (↓) ($D = 124$)	Lasso-DNA (↓) ($D = 180$)	SVM (↓) ($D = 388$)	Humanoid (↑) ($D = 6392$)	Average Rank
Base Kernels						
RBF	3.552 ± 0.304	217.75 ± 2.33	0.291 ± 0.001	**0.061 ± 0.004**	503.00 ± 67.82	6.6 / 17
Matérn52	3.453 ± 0.366	**215.77 ± 0.57**	0.292 ± 0.003	0.063 ± 0.003	509.79 ± 102.32	6.8 / 17
Linear	3.950 ± 0.441	274.67 ± 8.87	0.312 ± 0.004	0.226 ± 0.003	435.49 ± 42.00	13.0 / 17
Periodic	3.664 ± 0.231	309.10 ± 5.01	0.332 ± 0.005	0.226 ± 0.002	413.83 ± 71.30	15.0 / 17
BOCK	3.725 ± 0.593	225.36 ± 1.82	0.291 ± 0.003	0.068 ± 0.007	669.52 ± 50.15	6.1 / 17
SL	4.096 ± 0.473	246.78 ± 3.92	0.297 ± 0.001	0.112 ± 0.007	637.72 ± 37.86	8.2 / 17
RQ	3.858 ± 0.515	217.53 ± 4.25	0.294 ± 0.002	0.069 ± 0.017	600.80 ± 152.49	6.6 / 17
Search-based						
Greedy Search	3.606 ± 0.270	225.36 ± 1.82	0.291 ± 0.003	0.066 ± 0.004	687.12 ± 119.52	6.1 / 17
Compositional Search	3.869 ± 0.531	218.76 ± 1.17	**0.288 ± 0.003**	0.067 ± 0.007	**700.58 ± 81.55**	**4.4 / 17**
CAKE	3.412 ± 0.586	231.40 ± 15.67	0.300 ± 0.011	0.131 ± 0.036	667.49 ± 24.99	9.8 / 17
LLM-based						
ARM	3.981 ± 0.378	238.17 ± 3.65	0.319 ± 0.003	0.228 ± 0.003	457.14 ± 44.96	12.0 / 17
ATRBO	3.054 ± 0.816	267.65 ± 4.63	0.328 ± 0.010	0.221 ± 0.016	428.80 ± 295.37	15.2 / 17
TREvol	3.398 ± 0.620	236.41 ± 3.96	0.301 ± 0.007	0.079 ± 0.007	469.39 ± 144.43	11.6 / 17
TROpt	**4.114 ± 0.409**	234.23 ± 1.86	0.300 ± 0.006	0.110 ± 0.005	520.38 ± 65.42	7.8 / 17
TRPareto	3.872 ± 0.467	253.45 ± 4.73	0.296 ± 0.001	0.185 ± 0.008	509.90 ± 114.06	10.0 / 17
LMABO	3.953 ± 0.369	240.70 ± 5.24	0.304 ± 0.006	0.226 ± 0.001	390.23 ± 44.78	12.6 / 17
Kernel Discovery (Ours)	**4.353 ± 0.319**	**216.81 ± 0.87**	**0.286 ± 0.001**	**0.056 ± 0.003**	**762.78 ± 72.39**	**1.2 / 17**

Update the Population. To maintain a fixed population size N across rounds, we then truncate $\mathcal{P}_{t+1}$ by retaining the N members with the lowest LOO-CRPS:

$$\mathcal{P}_{t+1} \leftarrow \text{Top-N}(\mathcal{P}_t;\ \text{LOO-CRPS})\,. \tag{10}$$

This elitist update ensures that the population progressively favors kernels with more calibrated predictives while preventing unbounded growth. We additionally discard kernels that fail to improve performance over consecutive rounds. We provide full details of the algorithm in Algorithm 1.

5 Experiments

We evaluate our framework on five high-dimensional BO benchmarks and compare against 16 baselines, including base kernels, search-based methods, and LLM-based methods. Our code is here.

Benchmarks. Following prior works on high-dimensional BO [23, 26], we evaluate our framework on five standard benchmarks with dimensionality ranging from $D = 100$ to $D = 6392$: (i) **Rover** ($D = 100$, 52), (ii) **Mopta08** ($D = 124$, 18), (iii) **Lasso-DNA** ($D = 180$, 53), (iv) **SVM** ($D = 388$, 18), (v) **Humanoid** ($D = 6392$, 7). We exclude **Ant** since the function can be easily exploited by random initialization. Please refer to Appendix A for the detailed description of each task.

Baselines. We compare against 16 baselines across three categories: (1) BO methods that rely on a single fixed kernel, (2) search-based kernel methods, and (3) LLM-based BO methods. Please refer to Appendix B for the detailed description of each baseline.

- **Base kernels**: BO methods that rely on a single kernel: RBF, Matérn52, Linear, Periodic, RQ, BOCK, and SL.
- **Search-based Methods**: Methods that search over kernel candidates, including Greedy Search [54], Compositional Search [55], and CAKE [46]. For a fair comparison, all search-based methods share the same initial population $\mathcal{P}_0$ defined in Equation (2).
- **LLM-based Methods**: LLM-based BO methods, including end-to-end approaches (ARM, ATRBO, TREvol, TROpt, and TRPareto) from the LLAMEA-BO [44], and LMABO [43], which adaptively selects the acquisition function.

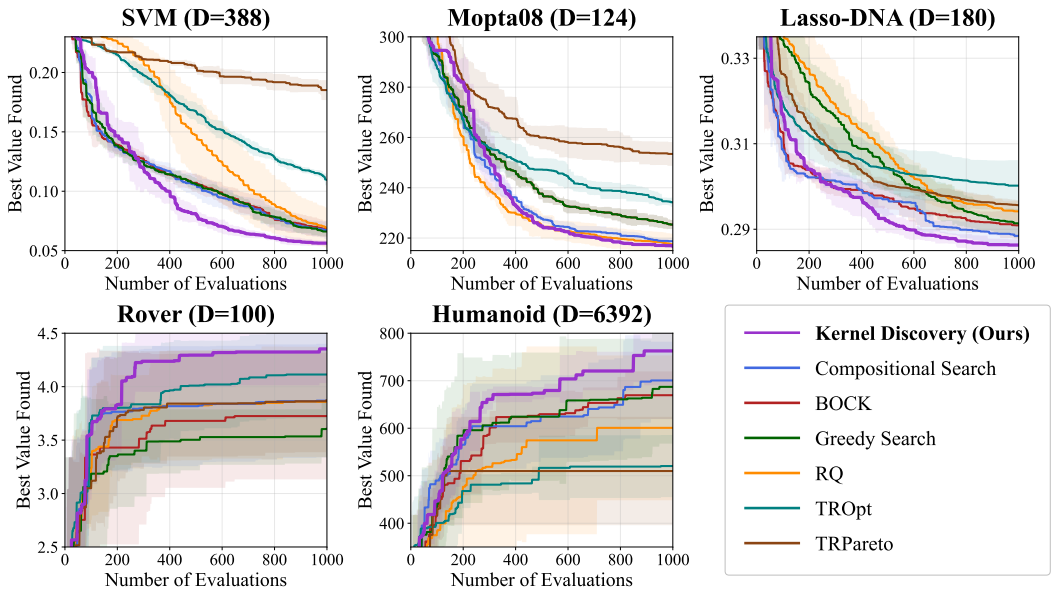


Figure 4: Learning curve of various methods across standard benchmarks. We visualize the top-2 baselines by average rank per category for clarity. Please refer to Figure 11 for the full results.

Implementation. For all methods (except for LLM-based approaches that change other parts of the BO pipeline), we use `LogEI` [56] as an acquisition function and `L-BFGS-B` as an optimizer. We use the Sobol sequence [57] to generate initial samples with size $|\mathcal{D}_0| = 20$, use a batch size of $q = 20$, and set the evaluation budget $T = 1000$. For our method, we set the population size to $N = 10$, generate two new kernel candidates (one from the discovery stage and one from the composition stage) per BO iteration. For a fair comparison, we use `GPT-4o` as the LLM backbone for both our method and all LLM-based baselines. Please refer to Appendix C for more details on implementation.

Main Results. We summarize the results of all methods in Table 1. As shown in the table, Kernel Discovery achieves an average rank of **1.2** across benchmarks, the best among all **17** methods. The runner-up is Compositional Search (rank 4.4), confirming that actively searching over kernel candidates is beneficial. However, due to the restricted search space, it is insufficient to discover effective kernels in high dimensions. Notably, CAKE (rank 9.8) performs substantially worse than competitive base kernels, indicating that injecting observations as context for the LLM is ineffective and degrades performance in high dimensions. Similarly, other LLM-based baselines underperform, indicating that leveraging LLMs to discover kernels within the BO pipeline is more effective than leveraging the LLM itself as a black-box optimizer.

We visualize the learning curve of kernel discovery against several baselines in Figure 4. Our approach consistently improves throughout training by discovering novel kernel functions at each BO iteration. Compositional Search and BOCK sometimes perform well in early rounds but stagnate in suboptimal regions. Our method avoids this by continually evolving the kernel population. We also conduct a runtime analysis of our method and the baselines in Appendix F.

6 Ablation Studies

We discuss the key design choices in our pipeline, justifying the need for the two-stage approach and analyzing the importance of each component.

Importance of Two-Stage Approach. We observed that when we directly instruct an LLM to generate code for novel kernel structures, it sometimes changes the variable names and replaces operators with equivalent operators, as shown in Figure 5a. We further demonstrate this phenomenon by measuring the functional diversity between generated kernels from direct code generation and those from our two-stage approach. As depicted in Figure 5b, our approach consistently produces more genuinely diverse kernel candidates. We also found that this functional diversity translates to consistently better performance (See Table 8), confirming that the two-stage approach discovers genuinely novel and effective kernel structures. Further details on the definition of the diversity metric, performance comparison, and functionally redundant code examples are provided in Appendix G.

Code Snippet 1

```
x1s = x1 / self.lengthscale
x2s = x2 / self.lengthscale

dist = torch.cdist(x1s, x2s, p=2).clamp(
    min=1e-15)
sqrt5_d = math.sqrt(5.0) * dist

exp_comp = torch.exp(-sqrt5_d)
covar = (1.0 + sqrt5_d + (5.0/3.0) *
    (dist**2)) * exp_comp
```

Code Snippet 2

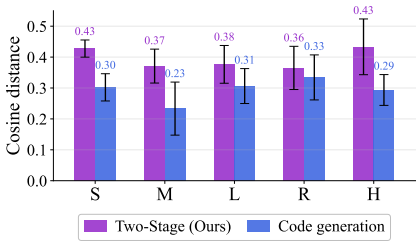
```
x1, x2 = x1 / self.lengthscale, x2 /
self.lengthscale

dist = torch.cdist(x1, x2, p=2) .clamp(
    min=1e-15)
scaled_d = math.sqrt(5.0) * dist

covar = (1.0 + scaled_dist +
    scaled_dist.pow(2) / 3.0) *
    torch.exp(-scaled_d)
```

Variable Renaming — Syntax Variation — Algebraic Rewriting

(a) Example code snippets from direct code generation. Both snippets result in the same function.



(b) Cosine distance among kernels from direct code generation vs two-stage approach.

Figure 5: Ablation studies on the two-stage approach for kernel discovery.

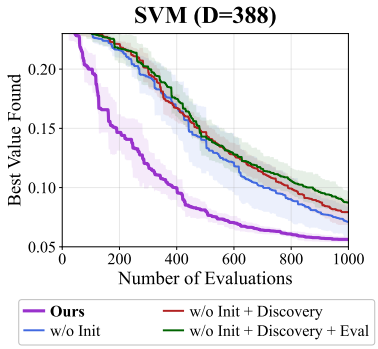


Figure 6: Ablation on each component of Kernel Discovery.

Table 2: Analysis on prompt template. Experiments are conducted with four random seeds in the SVM benchmark.

	Best Value Found	Failure Prob. (%)
Original Prompt (Ours)	0.056 ± 0.002	39.5
w/o HDBO Guidance	0.068 ± 0.007	35.7
w/o Constraint Guidance	0.064 ± 0.003	44.9

Table 3: Robustness on different LLMs. Experiments are conducted with four random seeds in the SVM benchmark.

	Best Value Found	Failure Prob. (%)
GPT-4o (Ours)	0.056 ± 0.002	39.5
GPT-4o-mini	0.060 ± 0.003	43.1
GPT-5-mini	0.056 ± 0.003	30.4

Ablation on Each Component. We conduct a systematic ablation study by removing each component. For the initial population, we replace BOCK and SL with Linear and Periodic kernels, matching the base kernel set used in CAKE. For the evaluation metric, we replace LOO-CRPS with the standard marginal log-likelihood (MLL). As shown in Figure 6, performance consistently drops as each component is removed. In particular, the initial population significantly affects sample efficiency, validating that kernels specifically designed for high-dimensional BO encode inductive biases that are valuable for effective surrogate modeling. We extend this ablation with more variants in Appendix H.

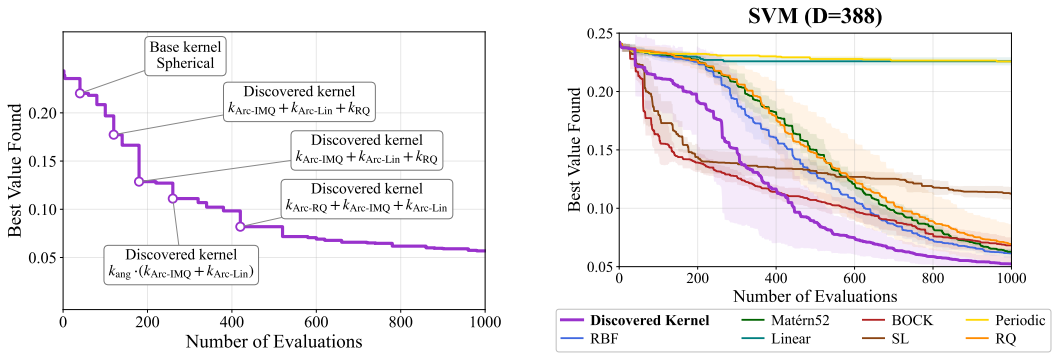
Analysis on Prompt Template. As shown in Figure 3, our prompt includes two key guidelines for the LLM: HDBO and Constraint guidelines. We ablate each by removing it in turn. As shown in Table 2, removing the HDBO guideline degrades performance, whereas removing the Constraint guideline leads to a high failure rate.

Robustness to Different LLMs. By default, we use `GPT-4o` for kernel formulation and code generation. To assess robustness across different LLMs, we run experiments with different LLMs: `GPT-4o-mini` and `GPT-5-mini`. As shown in Table 3, we find no significant performance gap across models, though `GPT-4o-mini` has a slightly higher kernel rejection rate due to weaker code reasoning, while `GPT-5-mini` generates fewer invalid kernels. We also conduct ablation studies with open-source LLMs and different LLM backbones for other LLM-based methods in Appendix I.

7 Analysis of Discovered Kernels

In this section, we characterize which kernels our pipeline discovers, how they evolve over training iterations, and whether they transfer across benchmarks.

Evolution of Discovered Kernels. A key question is whether our pipeline discovers genuinely novel structures or merely recovers known base kernels. To investigate this, we conduct a systematic study of the kernels selected by our pipeline over training iterations in Figure 7a. In the initial rounds, the pipeline favors simple base kernels with geometric input warpings (e.g., Spherical). As optimization progresses, more advanced structures emerge: the pipeline introduces geometric transformations such as `arctan` warping and composes them with existing kernels (e.g., RQ $\times$ Linear with `arctan`,



(a) Summary of selected kernels for the top-5 performance improvements.

(b) Evaluation of discovered kernel from SVM benchmark directly without kernel search.

Figure 7: Post-analysis on kernel discovery. Both experiments are conducted in SVM benchmark.

further combined with BOCK or IMQ). This suggests that compositions of multiple geometrically-warped kernels are a promising direction for high-dimensional BO, going beyond the single-warping strategies explored by prior works. We provide detailed explanations of the discovered kernels, along with the same evolution analysis across all benchmarks, in Appendix J.

Transferability of Discovered Kernels. While our method performs kernel discovery for a single black-box function, we can extract kernels discovered by one benchmark and determine whether they can be transferred to other benchmarks. To verify this, we choose one kernel discovered from the SVM benchmark as follows:

$$k_{\text{discover}}(\mathbf{x}, \mathbf{x}') = k_{\text{Matérn52}}(\mathbf{x}, \mathbf{x}') \cdot [k_{\text{tanh-Poly}}(\mathbf{x}, \mathbf{x}') + k_{\text{RQ}}(\mathbf{x}, \mathbf{x}')] \tag{11}$$

Note that this kernel cannot be obtained via naive compositions since k_{poly} does not exist in the base population and the $\texttt{tanh}(\cdot)$ operation is not allowed. As shown in Figure 7b, the discovered kernel outperforms base kernels on the SVM benchmark, and achieves an average rank of 2.8 across 8 fixed-kernel methods (full results in Table 12). To understand why the discovered kernel achieves superior performance on the SVM benchmark, we analyze the boundary hit ratio and the observation traveling salesman distance (OTSD) and find that the discovered kernel does not exhibit boundary-seeking behavior (See Figure 24). It is noteworthy that $k_{\text{tanh-Poly}}$ is a non-stationary kernel that is not typically preferred for BO, yet it performs well in high dimensions. This suggests that the role of non-stationary kernels in high-dimensional BO warrants further investigation. We provide detailed explanations of the discovered kernel and also present other frequently discovered kernels in Appendix K.

8 Conclusion

We presented Kernel Discovery, an LLM-driven evolutionary framework for designing effective GP kernels in high-dimensional BO. We adopt a two-stage pipeline in which one LLM proposes novel mathematical kernel forms and the other LLM converts them into executable code. To select among discovered kernels without overfitting to current observations, we introduce LOO-CRPS, an overfit-resistant criterion. On five high-dimensional BO benchmarks, our method achieves an average rank of 1.2 out of 17 methods. Beyond performance, our analysis of the discovered kernels provides new insights into what drives effective surrogate modeling in high-dimensional BO.

Limitations and Future Work. Our current framework runs the discovery process independently for each benchmark, optimizing kernels for each benchmark. While we demonstrate that kernels discovered on one benchmark can transfer to others, a natural extension is to develop a general discovery strategy that operates across multiple benchmarks simultaneously, leveraging shared structure to amortize the search cost. Additionally, some kernels generated by LLMs do not pass the verifiers for positive semi-definiteness or dimension-agnosticity. We believe that the failure rate could be further reduced by detailed prompt engineering or by imposing constraints during generation, although we view these as engineering directions orthogonal to our main contribution, which is to demonstrate that LLM-driven kernel discovery is a promising strategy for high-dimensional BO.

References

[1] Ryan Turner, David Eriksson, Michael McCourt, Juha Kiili, Eero Laaksonen, Zhen Xu, and Isabelle Guyon. Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *NeurIPS 2020 Competition and Demonstration Track*, 2021.

[2] Zi Wang, George E Dahl, Kevin Swersky, Chansoo Lee, Zelda Mariet, Zachary Nado, Justin Gilmer, Jasper Snoek, and Zoubin Ghahramani. Pre-training helps bayesian optimization too. In *ICML Workshop on Adaptive Experimental Design and Active Learning in the Real World*, 2022.

[3] Kevin Swersky, David Duvenaud, Jasper Snoek, Frank Hutter, and Michael A Osborne. Raiders of the lost architecture: Kernels for bayesian optimization in conditional parameter spaces. *arXiv preprint arXiv:1409.4011*, 2014.

[4] Kirthevasan Kandasamy, Willie Neiswanger, Jeff Schneider, Barnabas Poczos, and Eric P Xing. Neural architecture search with bayesian optimisation and optimal transport. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.

[5] Natalie Maus, Haydn Jones, Juston Moore, Matt J Kusner, John Bradshaw, and Jacob Gardner. Local latent space bayesian optimization over structured inputs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[6] Seunghun Lee, Jinyoung Park, Jaewon Chu, Minseo Yoon, and Hyunwoo J Kim. Latent bayesian optimization via autoregressive normalizing flows. In *International Conference on Learning Representations (ICLR)*, 2025.

[7] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. Learning search space partition for black-box optimization using monte carlo tree search. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[8] Felix Berkenkamp, Andreas Krause, and Angela P Schoellig. Bayesian optimization with safety constraints: safe and automatic parameter tuning in robotics. *Machine Learning*, 2023.

[9] Harold J Kushner. A new method of locating the maximum point of an arbitrary multipeak curve in the presence of noise. *Journal of Basic Engineering*, 1964.

[10] Roman Garnett. *Bayesian optimization*. Cambridge University Press, 2023.

[11] David K Duvenaud, Hannes Nickisch, and Carl Rasmussen. Additive gaussian processes. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2011.

[12] Kirthevasan Kandasamy, Jeff Schneider, and Barnabás Póczos. High dimensional bayesian optimisation and bandits via additive models. In *International Conference on Machine Learning (ICML)*, 2015.

[13] Jacob Gardner, Chuan Guo, Kilian Weinberger, Roman Garnett, and Roger Grosse. Discovering and exploiting additive structure for bayesian optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017.

[14] Xiaoyu Lu, Alexis Boukouvalas, and James Hensman. Additive gaussian processes revisited. In *International Conference on Machine Learning (ICML)*, 2022.

[15] Ziyu Wang, Frank Hutter, Masrour Zoghi, David Matheson, and Nando De Feitas. Bayesian optimization in a billion dimensions via random embeddings. *Journal of Artificial Intelligence Research*, 2016.

[16] Amin Nayebi, Alexander Munteanu, and Matthias Poloczek. A framework for bayesian optimization in embedded subspaces. In *International Conference on Machine Learning (ICML)*, 2019.

[17] Ben Letham, Roberto Calandra, Akshara Rai, and Eytan Bakshy. Re-examining linear embeddings for high-dimensional bayesian optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[18] David Eriksson and Martin Jankowiak. High-dimensional bayesian optimization with sparse axis-aligned subspaces. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2021.

[19] Leonard Papenmeier, Luigi Nardi, and Matthias Poloczek. Increasing the scope as you learn: Adaptive bayesian optimization in nested subspaces. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[20] David Eriksson, Michael Pearce, Jacob Gardner, Ryan D Turner, and Matthias Poloczek. Scalable global optimization via local bayesian optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2019.

[21] David Eriksson and Matthias Poloczek. Scalable constrained bayesian optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2021.

[22] Natalie Maus, Zhiyuan Jerry Lin, Maximilian Balandat, and Eytan Bakshy. Joint composite latent space bayesian optimization. In *International Conference on Machine Learning (ICML)*, 2024.

[23] Carl Hvarfner, Erik Orm Hellsten, and Luigi Nardi. Vanilla bayesian optimization performs great in high dimensions. In *International Conference on Machine Learning (ICML)*, 2024.

[24] Zhitong Xu, Haitao Wang, Jeff M Phillips, and Shandian Zhe. Standard gaussian process is all you need for high-dimensional bayesian optimization. In *International Conference on Learning Representations (ICLR)*, 2025.

[25] ChangYong Oh, Efstratios Gavves, and Max Welling. Bock: Bayesian optimization with cylindrical kernels. In *International Conference on Machine Learning (ICML)*, 2018.

[26] Colin Doumont, Donney Fan, Natalie Maus, Jacob R Gardner, Henry Moss, and Geoff Pleiss. We still don't understand high-dimensional bayesian optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2026.

[27] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[28] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[29] Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, et al. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*, 2025.

[30] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 2024.

[31] 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.

[32] Virginia Aglietti, Ira Ktena, Jessica Schrouff, Eleni Sgouritsa, Francisco Ruiz, Alan Malek, Alexis Bellot, and Silvia Chiappa. Funbo: Discovering acquisition functions for bayesian optimization with funsearch. In *International Conference on Machine Learning (ICML)*, 2025.

[33] Zhikai Zhao, Chuanbo Hua, Federico Berto, Kanghoon Lee, Zihan Ma, Jiachen Li, and Jinkyoo Park. Trajevo: Trajectory prediction heuristics design via llm-driven evolution. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2026.

[34] Sein Kim, Sangwu Park, Hongseok Kang, Wonjoong Kim, Jimin Seo, Yeonjun In, Kanghoon Yoon, and Chanyoung Park. Self-evolverec: Self-evolving recommender systems with llm-based directional feedback. *arXiv preprint arXiv:2602.12612*, 2026.

[35] Johannes Kirschner, Mojmir Mutny, Nicole Hiller, Rasmus Ischebeck, and Andreas Krause. Adaptive and safe bayesian optimization in high dimensions via one-dimensional subspaces. In *International Conference on Machine Learning (ICML)*, 2019.

[36] Lei Song, Ke Xue, Xiaobin Huang, and Chao Qian. Monte carlo tree search based variable selection for high dimensional bayesian optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[37] Erik Orm Hellsten, Carl Hvarfner, Leonard Papenmeier, and Luigi Nardi. High-dimensional bayesian optimization with group testing. *arXiv preprint arXiv:2310.03515*, 2023.

[38] Samuel Daulton, David Eriksson, Maximilian Balandat, and Eytan Bakshy. Multi-objective bayesian optimization over high-dimensional search spaces. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, 2022.

[39] Natalie Maus, Kaiwen Wu, David Eriksson, and Jacob Gardner. Discovering many diverse solutions with bayesian optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2023.

[40] Paolo Ascia, Elena Raponi, Thomas Bäck, and Fabian Duddeck. Feasibility-driven trust region bayesian optimization. In *International Conference on Automated Machine Learning (AutoML)*, 2025.

[41] Leonard Papenmeier, Matthias Poloczek, and Luigi Nardi. Understanding high-dimensional bayesian optimization. In *International Conference on Machine Learning (ICML)*, 2025.

[42] Bahador Rashidi, Kerrick Johnstonbaugh, and Chao Gao. Cylindrical thompson sampling for high-dimensional bayesian optimization. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2024.

[43] Giang Ngo, Dat Phan Trong, Dang Nguyen, Sunil Gupta, and Svetha Venkatesh. Adaptive acquisition selection for bayesian optimization with large language models. In *International Conference on Learning Representations (ICLR)*, 2026.

[44] Wenhu Li, Niki van Stein, Thomas Bäck, and Elena Raponi. Llamea-bo: A large language model evolutionary algorithm for automatically generating bayesian optimization algorithms. *arXiv preprint arXiv:2505.21034*, 2025.

[45] Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Large language models to enhance bayesian optimization. In *International Conference on Learning Representations (ICLR)*, 2024.

[46] Richard Cornelius Suwandi, Feng Yin, Juntao Wang, Renjie Li, Tsung-Hui Chang, and Sergios Theodoridis. Adaptive kernel design for bayesian optimization is a piece of cake with llms. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025.

[47] Austin Tripp. We are underselling the modularity of Bayesian optimization. `https://www.austintripp.ca/blog/2026-02-09-bo-modularity/`, 2026. Accessed: 2026-04-14.

[48] Carl Edward Rasmussen. Gaussian processes in machine learning. In *Summer School on Machine Learning*. Springer, 2003.

[49] Jonas Mockus. The application of bayesian methods for seeking the extremum. *Towards Global Optimization*, 1998.

[50] Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 1995.

[51] Christopher Williams and Carl Rasmussen. Gaussian processes for regression. In *Advances in Neural Information Processing Systems (NeurIPS)*, 1995.

[52] Zi Wang, Clement Gehring, Pushmeet Kohli, and Stefanie Jegelka. Batched large-scale bayesian optimization in high-dimensional spaces. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2018.

[53] Kenan Šehić, Alexandre Gramfort, Joseph Salmon, and Luigi Nardi. Lassobench: A high-dimensional hyperparameter optimization benchmark suite for lasso. In *International Conference on Automated Machine Learning (AutoML)*, 2022.

[54] Ibai Roman, Roberto Santana, Alexander Mendiburu, and Jose A Lozano. An experimental study in adaptive kernel selection for bayesian optimization. *IEEE Access*, 2019.

[55] David Duvenaud, James Lloyd, Roger Grosse, Joshua Tenenbaum, and Ghahramani Zoubin. Structure discovery in nonparametric regression through compositional kernel search. In *International Conference on Machine Learning (ICML)*, 2013.

[56] Sebastian Ament, Samuel Daulton, David Eriksson, Maximilian Balandat, and Eytan Bakshy. Unexpected improvements to expected improvement for bayesian optimization. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

[57] Ilya M Sobol', Danil Asotsky, Alexander Kreinin, and Sergei Kucherenko. Construction and comparison of high-dimensional sobol'generators. *Wilmott*, 2011.

Appendix

A Task Details

We evaluate our method on five standard high-dimensional BO benchmarks covering a range of problem types and dimensionalities.

Rover ($D = 100$). The Rover benchmark [52] is a trajectory optimization task in which a rover must navigate from a fixed start position to a fixed goal on a 2D terrain. The trajectory is parameterized by 50 intermediate 2D waypoints, yielding a $D = 100$-dimensional continuous search space in $[0, 1]^D$. The objective rewards proximity to the goal while penalizing path irregularity; higher values are better. The reward landscape contains many local optima corresponding to different trajectories.

Mopta08 ($D = 124$). The Mopta08 benchmark is a structural car body engineering design problem [18]. The task is to minimize the mass of a car structural component while satisfying 68 physical safety and manufacturing constraints. The 124-dimensional search space encodes continuous geometric design parameters. Constraint violations are aggregated into a penalty term, making the effective objective landscape highly non-smooth. Lower values are better.

Lasso-DNA ($D = 180$). Lasso-DNA is from the LassoBench suite [53], a benchmark collection for high-dimensional hyperparameter optimization of regularized regression models on genomic data. The input is a $D = 180$-dimensional vector of group-wise LASSO regularization coefficients; the objective is the mean squared prediction error on a validation set from a DNA microarray gene expression study (lower is better). The structured sparsity of the optimal regularizer makes this benchmark representative of practical bioinformatics hyperparameter search.

SVM ($D = 388$). The SVM benchmark [18] involves hyperparameter optimization of a Support Vector Machine classifier on a high-dimensional tabular dataset. The $D = 388$-dimensional search space encodes regularization and kernel hyperparameters of the SVM. The objective is the test misclassification error (lower is better). The high sensitivity of SVM performance to specific hyperparameter regions makes this a particularly challenging task.

Humanoid ($D = 6392$). The Humanoid benchmark [7] is a continuous locomotion control task from the MuJoCo physics simulator. A simulated humanoid robot is controlled by a linear policy that maps the 376-dimensional state observation vector to 17 joint torques, yielding a weight matrix of size $17 \times 376 = D = 6392$ parameters. The objective is the cumulative reward accumulated over a fixed-length simulation episode (higher is better).

Ant ($D = 888$, excluded from main evaluation). We first considered the Ant locomotion benchmark from MuJoCo, in which a quadruped robot is controlled by a linear policy mapping its state observations to joint torques. However, we excluded Ant from the main evaluation due to its abnormal reward structure. Specifically, we found out that the trivial zero-initialization baseline, which sets all policy parameters to $\mathbf{x} = \mathbf{0}$, achieves a cumulative reward of $+1000.40 \pm 4.40$. By contrast, all kernel-based GP-BO methods obtain dramatically lower or negative rewards, as shown in Table 4. This results makes the benchmark unsuitable for comparing kernel quality in BO.

Table 4: Final best values across base kernels compared to the zero-init-parameter policy baseline. Experiments are conducted with 4 random seeds.

Arm	Best Value Found
$\mathbf{x} = \mathbf{0}$ (zero-init-parameter policy)	1000.40 ± 4.40
RBF	$-\ 778.67 \pm 176.89$
Matérn52	$-\ 810.20 \pm 112.79$
RQ	-1011.04 ± 261.02
SL	$-\ 660.03 \pm 86.11$
BOCK	151.70 ± 358.00

B Baseline Details

We compare against 16 baselines across three categories: base kernels, search-based methods, and LLM-based methods.

Base Kernels

The following methods use the standard GP-BO setup with a single fixed kernel. For RBF, Matérn52, and RQ, we apply the data-dependent lengthscale prior of Hvarfner et al. [23]. Let $\mathbf{L} = \mathrm{diag}(\ell_1, \ldots, \ell_D)$ denote the ARD lengthscale matrix and $r_{\mathbf{L}} = \sqrt{(\mathbf{x} - \mathbf{x}')^\top \mathbf{L}^{-1} (\mathbf{x} - \mathbf{x}')}$.

- **RBF.** The squared exponential kernel with ARD:
$$k_{\mathrm{RBF}}(\mathbf{x}, \mathbf{x}') = \exp\!\left(-\tfrac{1}{2}\, r_{\mathbf{L}}^2\right). \tag{12}$$
- **Matérn52.** The Matérn kernel with smoothness $\nu = 5/2$ and ARD:
$$k_{\text{Matérn52}}(\mathbf{x}, \mathbf{x}') = \left(1 + \sqrt{5}\, r_{\mathbf{L}} + \tfrac{5}{3}\, r_{\mathbf{L}}^2\right) \exp\!\left(-\sqrt{5}\, r_{\mathbf{L}}\right). \tag{13}$$
Models twice mean-square differentiable functions; often better suited than RBF to practical optimization landscapes.
- **Linear.** The dot-product kernel with an unconstrained scale parameter $v \in \mathbb{R}$:
$$k_{\mathrm{Lin}}(\mathbf{x}, \mathbf{x}') = v\, \mathbf{x}^\top \mathbf{x}'. \tag{14}$$
- **Periodic.** The ARD periodic kernel with per-dimension lengthscales $\ell_i > 0$ and periods $p_i > 0$:
$$k_{\mathrm{Per}}(\mathbf{x}, \mathbf{x}') = \exp\left(-2 \sum_{i=1}^{D} \frac{\sin^2(\pi(x_i - x_i')/p_i)}{\ell_i^2}\right). \tag{15}$$
- **RQ (Rational Quadratic).** A scale mixture of RBF kernels with Gamma-distributed inverse lengthscales:
$$k_{\mathrm{RQ}}(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{r_{\mathbf{L}}^2}{2\alpha}\right)^{-\alpha}, \quad \alpha > 0. \tag{16}$$
The parameter α controls the relative contribution of short-range vs. long-range variation.
- **BOCK.** Bayesian Optimization with Cylindrical Kernels (BOCK) [25] decomposes each normalized input $\bar{\mathbf{x}} = (\mathbf{x} - \mathbf{c})/R$ into a radial component $r = \|\bar{\mathbf{x}}\|$ and an angular component $\hat{\mathbf{x}} = \bar{\mathbf{x}}/r$. The radial coordinate is transformed by the Kumaraswamy CDF $\kappa(r; \alpha, \beta) = 1 - (1 - r^\alpha)^\beta$ before applying a Matérn52 kernel, and the angular component is modeled by a polynomial kernel in the inner product:
$$k_{\mathrm{BOCK}}(\mathbf{x}, \mathbf{x}') = k_{\text{Matérn52}}(\kappa(r), \kappa(r')) \cdot \sum_{p=0}^{P-1} w_p\, (\hat{\mathbf{x}}^\top \hat{\mathbf{x}}')^p, \quad w_p > 0. \tag{17}$$
This product structure separates the smooth radial variation from the angular geometry, mitigating boundary-seeking behavior.
- **SL (Spherical Linear).** The kernel of Doumont et al. [26] first applies ARD lengthscale scaling followed by a global scale g, then maps the result to the unit sphere via the stereographic projection $\psi : \mathbb{R}^D \to S^D$:
$$\psi(\mathbf{z}) = \frac{1}{1 + \|\mathbf{z}\|^2} \begin{pmatrix} 2\mathbf{z} \\ \|\mathbf{z}\|^2 - 1 \end{pmatrix} \in S^D. \tag{18}$$
Letting $\tilde{\mathbf{x}} = (\mathbf{x} - \mathbf{c}) \,/\, (\boldsymbol{\ell} \cdot g)$, the kernel is a softmax-weighted mixture of a linear term on the sphere and a constant bias:
$$k_{\mathrm{SL}}(\mathbf{x}, \mathbf{x}') = \lambda_1\, \psi(\tilde{\mathbf{x}})^\top \psi(\tilde{\mathbf{x}}') + \lambda_0, \quad \lambda_0 + \lambda_1 = 1,\ \lambda_0, \lambda_1 \geq 0. \tag{19}$$
The stereographic projection ensures that inputs at any distance from the origin are mapped to a bounded region on the sphere, avoiding the unbounded extrapolation that afflicts linear kernels in high dimensions.

Search-based Methods

All search-based baselines share the same initial population $\mathcal{P}_0$ as our method for a fair comparison.

- **Greedy Search.** At each BO iteration, it evaluates every kernel in $\mathcal{P}_0$ and selects the one with the lowest Bayesian Information Criterion (BIC). No new kernels are created.
- **Compositional Search.** At each BO iteration, use greedy search to discover kernel structures that best explain the observed data via pairwise additive and multiplicative compositions of kernels in the current population. Following the paper, we use BIC as a selection metric and directly use hyperparameters fitted in a single kernel to composite kernels without refitting. We use the max depth of the composition as 3.
- **CAKE.** Context-Aware Kernel Evolution [46] uses an LLM to select and compose base kernels conditioned on prior observations. Its search is restricted to additive and multiplicative compositions of a fixed base set, and passing raw observations to the LLM induces context-length issues in high dimensions. If the dimensions are too high, we truncate the observations to prevent context limit violation. We use the max depth of the composition as 3. While CAKE uses `GPT-4o-mini` for the LLM API, we evaluate the baseline with `GPT-4o` for a fair comparison.

LLM-based Methods

For all LLM-based baselines, we initialize the dataset with Sobol sequences of size of 20 as our method for a fair comparison.

- **LLaMEA-BO (ARM, ATRBO, TREvol, TROpt, TRPareto).** Five end-to-end LLM-driven BO algorithms produced by the LLaMEA-BO pipeline [44], which uses an LLM in an evolutionary loop to generate and refine BO algorithm code. Each variant is a distinct algorithm discovered by the pipeline. These methods may not use a GP surrogate and directly propose candidates from prior observations.
- **LMABO.** An LLM-based BO method that adaptively selects the acquisition function at each iteration [43]. For a fair comparison, we use GP with an RBF as the base kernel, following Hvarfner et al. [23], and use `L-BFGS-B` as the optimizer.
- **LLAMBO (excluded from main evaluation).** LLAMBO [45] uses LLMs to propose and evaluate candidates conditioned on historical evaluations, making its prompt size scale with both history length and input dimensionality. On Rover ($D = 100$), the full history of 980 evaluations already requires approximately 600K input tokens, exceeding the 128k context window of GPT-4o. Even after truncating the history to 20 observations, our GPT-4o probe costs about $\$1.05$ per BO iteration, corresponding to about $\$1{,}024$ for a single full run. We therefore exclude LLAMBO due to context-length and budget limitations.

Additional Baselines: TuRBO and SAASBO. We compare against two additional baselines, TuRBO [20] and SAASBO [18], which are not included in the main table as they employ different acquisition function maximization strategies and inference procedures (TuRBO uses a trust-region approach and SAASBO uses sparse axis-aligned priors with MCMC-based inference). However, we compare the results of those baselines as they are widely used in high-dimensional BO. As shown in Table 5, Kernel Discovery outperforms both methods across all benchmarks.

Table 5: Comparison with additional baselines across standard benchmarks. SAASBO results use $T = 500$ due to its higher computational cost. Experiments are conducted with 4 random seeds.

Method	**Rover** ($\uparrow$) ($D = 100$)	**Mopta08** ($\downarrow$) ($D = 124$)	**Lasso-DNA** ($\downarrow$) ($D = 180$)	**SVM** ($\downarrow$) ($D = 388$)	**Humanoid** ($\uparrow$) ($D = 6392$)
Kernel Discovery (Ours)	**4.353 ± 0.319**	**216.81 ± 0.87**	**0.286 ± 0.001**	**0.056 ± 0.003**	**762.78 ± 72.39**
TuRBO	3.584 ± 0.175	242.39 ± 3.60	0.289 ± 0.001	0.142 ± 0.015	449.26 ± 85.56
SAASBO ($T = 500$)	3.981 ± 0.362	223.30 ± 3.44	0.290 ± 0.001	0.148 ± 0.012	449.08 ± 53.58

C Implementation Details

Shared BO Setup. For all methods except end-to-end LLM-based approaches (ARM, ATRBO, TREvol, TROpt, TRPareto), we use a shared GP-based BO framework built on GPyTorch and BoTorch. The surrogate is an exact Gaussian Process with homoscedastic Gaussian observation noise. Kernel hyperparameters are optimized by maximizing the marginal log-likelihood (MLL) using `L-BFGS-B` with 3 random restarts. Inputs are normalized to $[0, 1]^D$ and outputs are standardized to zero mean and unit variance before fitting.

We use `LogEI` as the acquisition function and optimize it using `L-BFGS-B` with 4 random restarts initialized from 512 Sobol quasi-random candidates. RAASP sampling is implemented by tuning the option `sample_around_best=True`. The initial dataset $\mathcal{D}_0$ is generated by a Sobol sequence of size $|\mathcal{D}_0| = 20$. The total evaluation budget is $T = 1000$ function queries.

All experiments use a batch BO setting with batch size $q = 20$. At each BO round, the acquisition function is maximized to select a batch of q candidate points, which are evaluated simultaneously. This yields $T/q = 50$ BO rounds per run. All baselines also use the same $q = 20$ batch size to ensure a fair comparison. Please refer to Table 6 for the comparison of the performance of base kernels under different batch sizes. All experiments are done with a single RTX NVIDIA L40S GPU.

Full Prompt Templates. We present the full prompt templates for each stage, mathematical form discovery, code conversion, and composition, in Figure 8 to 10, respectively. In addition to generating solely mathematical expressions or code blocks, we instruct LLM to justify its answer for exploring its reasoning capabilities and the robustness of the pipeline.

```
You are an expert in Gaussian process kernel design for high-dimensional
Bayesian optimization.

TOP {len(parent_formulas_with_loss)} kernels in our population:
{context}

Generate {n_discoveries} DISTINCT, CREATIVE, mathematically innovative
DISCOVERED kernels.
Each discovery should be derived from one or more of the above kernels, but
with meaningful mathematical changes.
{highdim}{psd}{simplicity}
OUTPUT FORMAT:
Return EXACTLY {n_discoveries} kernel descriptions. Enclose EACH in a ```
formula block:

```formula
KERNEL: [name]

PARAMETERS:
- [name] ([shape], [constraint]): [description]

INPUT TRANSFORM:
  [step-by-step math]

COVARIANCE FUNCTION:
  [k(x1, x2) = ...]

PSD GUARANTEE:
  [why this kernel is PSD]
```

Each MUST be fundamentally distinct (different transform, different
covariance structure, different mathematical idea)."""
```

Figure 8: Mathematical form discovery prompt template

```
You are a GPyTorch kernel code generator. Convert this kernel formula into a GPyTorch EvolvedKernel class.

{formula}

REFERENCE TEMPLATE A - Distance-based kernel {RBF code}

REFERENCE TEMPLATE B - Feature-map kernel {SL code}

CRITICAL RULES:
1. __init__ SIGNATURE: `__init__` MUST accept `ard_num_dims: int` as the ONLY required parameter. The system calls `EvolvedKernel(ard_num_dims=D)`. All other args MUST have defaults. NEVER add required args like `q: int`, `center: Tensor`, `M: int`.
2. FORWARD SIGNATURE: `def forward(self, x1, x2, diag=False, **params)` - always accept **params.
3. BATCH-SAFE SHAPES (CRITICAL): The kernel is tested with these exact shapes - your code MUST handle ALL of them:
   - 2D: x1=(5, D) vs x2=(1, D) -> output must be (5, 1), NOT (5, 5)
   - 2D: x1=(3, D) vs x2=(7, D) -> output must be (3, 7)
   - 3D batched (BoTorch acqf optimization): x1=(1, 4, D) vs x2=(1, 3, D) -> output must be (1, 4, 3)
   Therefore: ALL dim indexing must be relative (-1 for D, -2 for N). NEVER use `.size(0)`, `.expand(x.size(0), ...)`, or `.shape[0]`. For broadcasting scalars to match batch+N dims, use `torch.ones_like(x[..., :1])`.
4. OUTPUT SHAPE: return (..., N1, N2) for ANY N1 != N2. If your output is (N1, N1) instead of (N1, N2), you have a bug.
5. DISTANCE: `torch.cdist(x1s, x2s, p=2)` - NEVER `x1.unsqueeze(-2) - x2.unsqueeze(-3)` (OOM).
6. INNER PRODUCT: `x1 @ x2.transpose(-1, -2)`.
7. DEVICE: new tensors in forward use `device=x1.device`. Use `register_buffer` in __init__.
8. NO IN-PLACE: `x = x + y` not `x += y`. `x = x.clamp(...)` not `x.clamp_(...)`.
9. NO DETACH/NUMPY: never `.detach()`, `.numpy()`, `.item()`, `torch.no_grad()` in forward.
10. NUMERICAL: `.clamp(min=1e-15)` before sqrt/log, `.clamp(max=20.0)` for exp.
11. LENGTHSCALE: if `has_lengthscale=True`, do NOT `register_parameter("raw_lengthscale", ...)`.
12. PARAM ORDER: `register_parameter(name, ...)` BEFORE `register_constraint(name, ...)`.
13. DIAG: `if diag: return covar.diagonal(dim1=-2, dim2=-1)`.
14. IMPORTS: include `import math` if using math.sqrt/pi/log.
15. NO SELF-REASSIGNMENT in forward(): never `self.x = ...` - use local variables.
16. ONLY USE EXISTING gpytorch classes.
17. NO SUB-KERNELS: do NOT instantiate gpytorch kernel objects (e.g. `MaternKernel()`, `RBFKernel()`) as sub-components. They produce LazyEvaluatedKernelTensor with unpredictable shapes. Instead, implement the formula directly (e.g. for Matern-5/2: `(1 + sqrt5*d + 5/3*d**2) * exp(-sqrt5*d)`).
18. NO DATA-DEPENDENT BOUNDS in forward(): never compute bounds/center from the input data (e.g. `x1.max(dim=-2)` or `x1.min(dim=-2)`). These change with N and break shape invariance. Store fixed bounds via `register_buffer` in __init__, or use constants (e.g. 0 and 1).
19. SCALAR PAIRWISE DISTANCE: when computing distance between scalar values (e.g. warped radii of shape (..., N, 1)), use `torch.cdist(r1, r2, p=2)` which gives (..., N1, N2). NEVER use `r1 - r2` or `torch.abs(r1 - r2)` - this gives (..., N1, 1) not (..., N1, N2) and breaks cross-pair computation.

Return ONLY the Python code in a ```python block."""
```

Figure 9: Code conversion prompt template.

```
You are combining GP kernels for high-dimensional Bayesian optimization.

Population of {len(population_formulas_with_loss)} kernels:
{context}

Generate {n_compositions} DISTINCT composed kernels. For EACH composition:
- Pick exactly 2 kernels from the population that COMPLEMENT each other
mathematically
- Combine them: sum their covariance functions, multiply them, share a
transform with different distance metrics, or any other valid composition
- Explain WHY these two kernels complement each other
{psd}{simplicity}
OUTPUT FORMAT:
Return EXACTLY {n_compositions} kernel descriptions. Enclose EACH in a ```
formula block:

```formula
KERNEL: [name]
COMPOSED FROM: [Kernel X] + [Kernel Y]  (or *, or other operation)
WHY: [1 sentence on why these complement each other]

PARAMETERS:
- [param] ([shape], [constraint]): [description]

INPUT TRANSFORM:
  [step-by-step math]

COVARIANCE FUNCTION:
  k(x1, x2) = ...

PSD GUARANTEE:
  [why this kernel is PSD]
```"""
```

Figure 10: Composition prompt template.

Kernel Validation Failure Rates. The failure probability in Tables 2 and 3 denotes the empirical fraction of LLM-generated kernels rejected by our structural validator. A generated kernel k_{new} is accepted only when

$$\mathcal{V}(k_{\text{new}}) = \mathcal{V}_{\text{agn}}(k_{\text{new}}) \wedge \mathcal{V}_{\text{psd}}(k_{\text{new}}) = 1.$$

Here, $\mathcal{V}_{\text{agn}}$ checks whether the generated kernel implementation is dimension- and shape-agnostic, as required by the BO pipeline. Concretely, we instantiate the kernel with different input dimensionalities and evaluate its `forward` function on several synthetic input configurations, including self-covariance inputs (N, D), cross-covariance inputs (N_1, D) and (N_2, D) with $N_1 \neq N_2$, and batched inputs used during acquisition optimization. The kernel is rejected if any call raises a runtime error, assumes a hard-coded dimensionality, fails to broadcast correctly, or returns a covariance tensor with an invalid shape. The PSD check $\mathcal{V}_{\text{psd}}$ then evaluates the kernel on random inputs and tests whether the resulting Gram matrix is numerically positive semi-definite via a Cholesky decomposition with a small jitter. We report the fraction of generated kernels for which at least one of these checks fails, aggregated over BO iterations and random seeds.

Effect of Batch Size (q). The main experiments use $q = 20$, reducing the number of BO rounds to $T/q = 50$. To assess whether this setting disproportionately favors our method, we compare base kernels under $q = 20$ vs. $q = 1$ on the SVM benchmark. Table 6 shows that the relative ordering of kernels is largely preserved across both settings, and that the optimal kernel (RBF in the $q = 1$ setting) is the same as in the $q = 20$ setting. This confirms that the batch setting does not systematically alter which kernels are competitive, and that our comparison is not confounded by the choice of q.

Table 6: Comparison of final best values across base kernels under different oracle query batch sizes.

	RBF	Matérn52	RQ	BOCK	SL	Linear	Periodic
$q = 20$	0.061 ± 0.003	0.062 ± 0.002	0.069 ± 0.014	0.068 ± 0.006	0.111 ± 0.006	0.226 ± 0.003	0.226 ± 0.001
$q = 1$	0.078 ± 0.003	0.117 ± 0.021	0.078 ± 0.007	0.070 ± 0.009	0.064 ± 0.002	0.226 ± 0.002	0.223 ± 0.002

D Algorithm Details

Discard Logic. The difference between the standard evolutionary pipeline and our task is that one kernel that suits a certain round may not suit the other rounds, as the data keeps changing. To mitigate this, we discard the kernel if it is selected but does not improve the best value found so far. For kernels in the initial population, which are the basic building blocks of our discovery pipeline, we discard them if they are selected and no improvement is observed after $P = 3$ iterations. If all kernels in the population are discarded, we reset the population to the initial population.

Humanoid Benchmark. In the humanoid benchmark, the dimension is too high, and we observed that even with the LOO-CRPS evaluation metric, it sometimes leads to selecting too complex kernels that may not lead to an improvement. To remedy this, we regularize the LOO-CRPS score by the BIC metric as follows:

$$\text{LOO-CRPS-BIC}(\mathbf{K}_k, \mathbf{y}) \;=\; \frac{1}{n}\sum_{i=1}^{n} \text{CRPS}\big(\mathcal{N}(\mu_{-i},\, \sigma^2_{-i}),\, y_i\big) \;+\; \frac{|\boldsymbol{\theta}_k| \log n}{n}, \tag{20}$$

where $|\boldsymbol{\theta}_k|$ is the number of hyperparameters in the kernel k. We found that this regularization works well, especially in extremely high dimensions, $D \gg n$.

Algorithm 1 Kernel Discovery for High-Dimensional BO

Require: Objective f, initial dataset $\mathcal{D}_0$, population size N, BO budget T, initial-population patience P, LLM $\mathcal{M}$ with prompts $\rho_{\text{disc}}, \rho_{\text{conv}}, \rho_{\text{comp}}$, verifier $\mathcal{V}$

Initialize $\mathcal{P}_0 \leftarrow \{k_{\text{RBF}},\, k_{\text{Mat52}},\, k_{\text{RQ}},\, k_{\text{BOCK}},\, k_{\text{SL}}\}$, $\quad \mathcal{P} \leftarrow \mathcal{P}_0$
$\mathcal{D} \leftarrow \mathcal{D}_0$, $\quad y^* \leftarrow \max_{(\mathbf{x},y)\in\mathcal{D}_0} y$, $\quad \text{fail}[k] \leftarrow 0$ for all $k \in \mathcal{P}_0$
for $t = 0, 1, \ldots, T-1$ **do**
  // Discovery stage
  $m^{\text{new}} \sim \mathcal{M}(\{m_i\}_{k_i\in\mathcal{P}};\, \rho_{\text{disc}})$, $\quad c^{\text{new}} \leftarrow \mathcal{M}(m^{\text{new}};\, \rho_{\text{conv}})$, $\quad k^{\text{new}} \leftarrow (m^{\text{new}}, c^{\text{new}})$
  if $\mathcal{V}(k^{\text{new}}) = 1$ **and** GP fitting time ≤ 60s **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{k^{\text{new}}\}$
  end if
  // Composition stage
  $m^{\text{comp}} \sim \mathcal{M}\big(\{m_{(i)}\}_{i=1}^{N};\, \rho_{\text{comp}}\big)$, $\quad c^{\text{comp}} \leftarrow \mathcal{M}(m^{\text{comp}};\, \rho_{\text{conv}})$, $\quad k^{\text{comp}} \leftarrow (m^{\text{comp}}, c^{\text{comp}})$
  if $\mathcal{V}(k^{\text{comp}}) = 1$ **and** GP fitting time ≤ 60s **then**
    $\mathcal{P} \leftarrow \mathcal{P} \cup \{k^{\text{comp}}\}$
  end if
  // Kernel selection and BO step
  $k^\star \leftarrow \arg\min_{k\in\mathcal{P}} \text{LOO-CRPS}(\mathbf{K}_k, \mathbf{y})$
  Fit GP with $k^\star$ on $\mathcal{D}$; $\quad \mathbf{x}_{t+1} \leftarrow \arg\max_{\mathbf{x}} \alpha(\mathbf{x};\, \text{GP})$
  $y_{t+1} \leftarrow f(\mathbf{x}_{t+1})$; $\quad \mathcal{D} \leftarrow \mathcal{D} \cup \{(\mathbf{x}_{t+1},\, y_{t+1})\}$
  // Discard logic
  if $y_{t+1} \leq y^*$ **then**
    $\text{fail}[k^\star] \leftarrow \text{fail}[k^\star] + 1$
    if $k^\star \notin \mathcal{P}_0$ **or** $\text{fail}[k^\star] \geq P$ **then**
      $\mathcal{P} \leftarrow \mathcal{P} \setminus \{k^\star\}$
    end if
  end if
  if $\mathcal{P} = \emptyset$ **then**
    $\mathcal{P} \leftarrow \mathcal{P}_0$; $\quad \text{fail}[k] \leftarrow 0$ for all $k \in \mathcal{P}_0$
  end if
  // Population update
  $\mathcal{P} \leftarrow \text{Top-N}(\mathcal{P};\, \text{LOO-CRPS}(\cdot, \mathbf{y}))$, $y^* \leftarrow \max(y^*, y_{t+1})$
end for
return $\arg\max_{(\mathbf{x},y)\in\mathcal{D}} y$

E Extended Experiment Results

Full Learning Curves. Figure 11 extends Figure 4 to show the full learning curves for all 16 baselines across all five benchmarks.

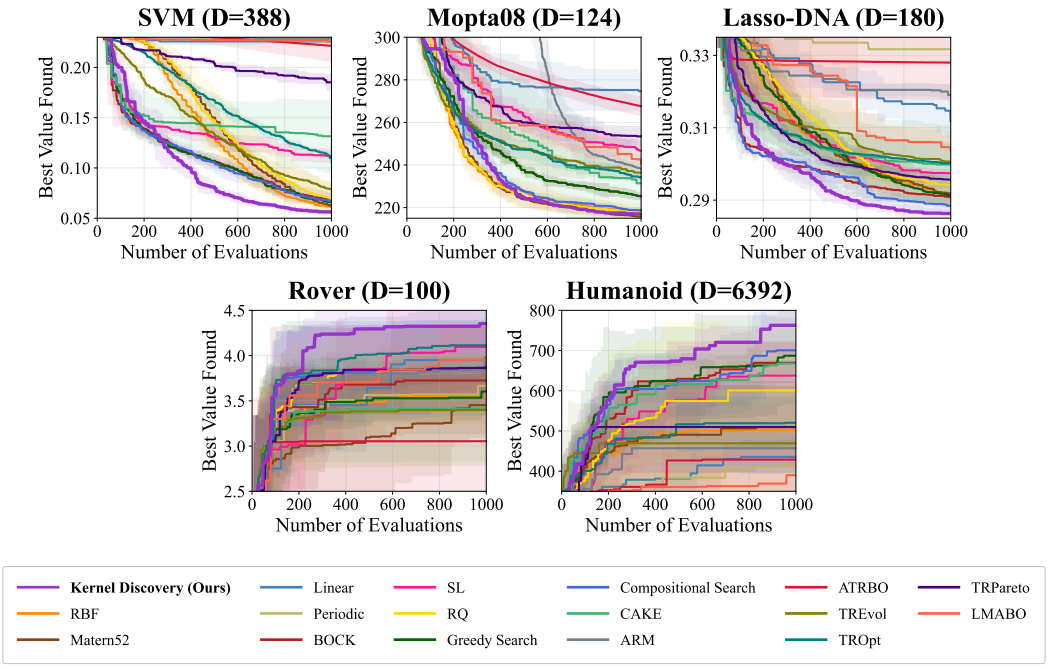


Figure 11: Full learning curves for all methods across five benchmarks.

F Time Complexity Analysis

We analyze the per-iteration wall-clock time of Kernel Discovery and compare it against all baselines in Table 7. To systematically identify the bottleneck of our running time, we decompose each iteration into the following phases:

- **GP Fitting**: Fitting all kernels in the current population in parallel using a process pool. The kernel selection score (e.g., LOO-CRPS) is computed within this phase, so the cost of choosing the best candidate is already included.
- **LLM Inference**: Two API calls per iteration - one for generating a novel mathematical kernel form and one for converting that form into executable code.
- **Kernel Validation**: Lightweight structural checks on each newly generated kernel before it is admitted to the population.
- **Search**: An additional kernel search procedure that applies only to search-based methods. Compositional Search enumerates pairwise additive and multiplicative compositions, while CAKE employs an LLM-guided composition with a BAKER-based selection procedure. Our method and Greedy Search select the best candidate directly from the evaluated pool, so this cost is negligible.
- **Acquisition Function Optimization**: Maximizing the acquisition function via `L-BFGS-B` to obtain the next query point.
- **Oracle Evaluation**: Querying the black-box benchmark function at the selected point, shared identically across all methods.

For LLM-based end-to-end methods, the pipeline does not follow the standard BO decomposition, so we report only the total per-iteration time and oracle evaluation cost. As shown in Table 7, our method incurs longer time than fixed base kernel methods but achieves comparable total running time to search-based methods. Among LLM-based baselines, TREvol and TRPareto require significantly more time per iteration. We also observe that LLM inference is the primary bottleneck in our pipeline, partially due to the two-stage approach that issues separate API calls for formula generation and code conversion. While this decomposition is critical for producing functionally diverse kernels (Section 6), developing more efficient inference strategies is a promising direction for future work.

Table 7: Per-iteration wall-clock time breakdown (mean $\pm$ std, seconds) on the SVM benchmark with 4 random seeds. For LLM-based methods, individual phases are not separately tracked; we report only the total time and oracle evaluation time.

Method	GP Fit	LLM	Validation	Search	Acqf Opt	Oracle	Total
Base Kernels							
RBF	0.68 ± 0.02	N/A	N/A	0.00 ± 0.00	2.70 ± 0.97	0.28 ± 0.04	5.18 ± 1.00
Matérn52	0.69 ± 0.04	N/A	N/A	0.00 ± 0.00	2.76 ± 0.99	0.28 ± 0.04	5.26 ± 1.00
Linear	0.95 ± 0.58	N/A	N/A	0.00 ± 0.00	3.14 ± 0.88	0.29 ± 0.04	6.49 ± 2.38
Periodic	0.85 ± 0.20	N/A	N/A	0.00 ± 0.00	1.35 ± 1.52	0.25 ± 0.04	4.28 ± 1.40
BOCK	1.14 ± 0.44	N/A	N/A	0.00 ± 0.00	4.03 ± 1.62	0.31 ± 0.06	7.02 ± 1.87
SL	1.41 ± 0.85	N/A	N/A	0.00 ± 0.00	3.57 ± 1.10	0.30 ± 0.04	6.83 ± 1.69
Search-based							
Greedy Search	5.38 ± 1.80	N/A	N/A	0.00 ± 0.00	5.70 ± 0.56	0.35 ± 0.05	11.51 ± 2.04
Compositional Search	4.90 ± 1.55	N/A	N/A	13.80 ± 6.78	5.88 ± 1.33	0.34 ± 0.05	20.70 ± 8.87
CAKE	7.15 ± 3.61	9.09 ± 3.23	N/A	17.44 ± 3.73	19.34 ± 4.02	0.35 ± 0.06	37.14 ± 4.47
LLM-based							
ARM	N/A	N/A	N/A	N/A	N/A	0.30 ± 0.04	4.54 ± 0.02
ATRBO	N/A	N/A	N/A	N/A	N/A	0.28 ± 0.05	14.51 ± 0.06
TREvol	N/A	N/A	N/A	N/A	N/A	0.31 ± 0.02	209.88 ± 3.40
TROpt	N/A	N/A	N/A	N/A	N/A	0.29 ± 0.05	3.67 ± 0.10
TRPareto	N/A	N/A	N/A	N/A	N/A	0.30 ± 0.02	66.29 ± 0.26
LMABO	3.56 ± 0.50	1.92 ± 0.45	N/A	0.00 ± 0.00	1.92 ± 0.28	0.27 ± 0.04	5.76 ± 0.61
Kernel Discovery (Ours)	4.02 ± 1.22	14.24 ± 2.05	1.90 ± 0.78	0.00 ± 0.00	4.56 ± 0.11	0.30 ± 0.01	28.05 ± 1.81

G Direct Code Generation Analysis

Details on Cosine Distance Metric. To quantify the functional diversity of generated kernels, we compare their induced Gram matrices on a shared reference input set. For each benchmark and each method, we collect all valid LLM-generated kernels produced during the BO run. We then sample a fixed reference batch $\mathbf{X}_{\text{ref}} \in [0,1]^{N \times D}$ with $N = 80$, where D is the dimensionality of the corresponding benchmark. The same reference batch is used for all kernels within a benchmark.

For each generated kernel k_i, we compute its Gram matrix $K_i = k_i(\mathbf{X}_{\text{ref}}, \mathbf{X}_{\text{ref}}) \in \mathbb{R}^{N \times N}$, and flatten it into a vector $\mathbf{v}_i = \text{vec}(K_i) \in \mathbb{R}^{N^2}$. We then compute the mean pairwise cosine similarity among all generated kernels in the pool $\mathcal{P}$:

$$\bar{s} = \frac{2}{|\mathcal{P}|(|\mathcal{P}|-1)} \sum_{i<j} \frac{\mathbf{v}_i^\top \mathbf{v}_j}{\|\mathbf{v}_i\|_2 \|\mathbf{v}_j\|_2}.$$

Finally, we report the cosine distance as $\text{CosineDistance} = 1 - \bar{s}$. A higher value indicates that the generated kernels induce more diverse covariance structures on the same reference inputs.

Performance Comparison. We further compare direct code generation with our two-stage generation under the same experimental setup. As shown in Table 8, the two-stage approach achieves better final performance on all benchmarks. This suggests that generating kernels through an explicit mathematical formulation step leads to more functionally meaningful and consistently effective kernel candidates than directly generating code.

Table 8: Performance comparison between direct code generation and our two-stage generation. Results are averaged over 4 random seeds. Bold denotes the best entry in each column.

Method	**Rover** (↑) ($D = 100$)	**Mopta08** (↓) ($D = 124$)	**Lasso-DNA** (↓) ($D = 180$)	**SVM388** (↓) ($D = 388$)	**Humanoid** (↑) ($D = 6392$)
Two-Stage Generation (Ours)	**4.353 ± 0.319**	**216.81 ± 0.87**	**0.286 ± 0.001**	**0.056 ± 0.003**	**762.78 ± 72.39**
Direct Code Generation	3.872 ± 0.321	217.32 ± 0.75	0.288 ± 0.001	0.057 ± 0.004	651.85 ± 85.93

Additional Examples of Functional Redundancy in Direct Code Generation. As shown in Figure 5a, direct code generation often produces syntactically different implementations that correspond to the same kernel function. This is a common pitfall of searching directly in code space: changes such as variable renaming, syntax-level variation, or algebraic rewriting can be counted as new candidates. In Figures 12 to 14, we provide examples of this behavior. Across these pairs, the two forward() implementations look different at the code level but share the same kernel function.

```
def forward(self, x1, x2, diag=False, **
params):
    x1s = x1_scaled / self.lengthscale
    x2s = x2_scaled / self.lengthscale
    dist_sq = torch.cdist(x1s, x2s, p=2).
pow(2)

    covar1 = torch.exp(-0.5 * dist_sq).
clamp(min=1e-15)
    covar2 = (1.0 + dist_sq / (2.0 * alpha))
.pow(-alpha).clamp(min=1e-15)
    covar = covar1 + covar2

    if diag:
        return covar.diagonal(dim1=-2,
dim2=-1)
    return covar
```

```
def forward(self, x1, x2, diag=False, **
params):
    x1s = x1 / self.lengthscale
    x2s = x2 / self.lengthscale
    dist_sq = torch.cdist(x1s, x2s, p=2).
pow(2)

    covar_exp = torch.exp(-0.5 * dist_sq).
clamp(min=1e-15)
    covar_matern = (1 + dist_sq / (2 * self.
alpha)).pow(-self.alpha).clamp(min=1e-15)
    covar_combined = covar_exp +
covar_matern
    if diag:
        return covar_combined.diagonal(
dim1=-2, dim2=-1)
    return covar_combined
```

Figure 12: Functional redundancy example from direct code generation. Both snippets compute $k(x, x') = \exp(-d^2/2) + (1 + d^2/(2\alpha))^{-\alpha}$. Green denotes variable renaming, and red denotes syntax-level variation.

```
def forward(self, x1, x2, diag=False, **
params):
    x1s = x1 / self.lengthscale
    x2s = x2 / self.lengthscale
    sq_dist = torch.cdist(x1s, x2s).pow(2)

    covar1 = torch.exp(-0.5 * sq_dist)
    sqrt5_d = math.sqrt(5) * torch.cdist(
x1s, x2s).clamp(min=1e-15)


    covar2 = (1.0 + sqrt5_d + (5.0 / 3.0) *
 sq_dist) * torch.exp(-sqrt5_d)

    covar = self.alpha * covar1 + (1 -
self.alpha) * covar2


    if diag:
        return covar.diagonal(dim1=-1,
dim2=-2)
    return covar
```

```
def forward(self, x1, x2, diag=False, **
params):
    x1s = x1 / self.lengthscale
    x2s = x2 / self.lengthscale
    dist2 = torch.cdist(x1s, x2s).pow(2)

    exp_neg_half_dist2 = torch.exp(-0.5 *
dist2)
    scaled_dist = torch.cdist(x1s, x2s).
clamp(min=1e-15)
    covar1 = exp_neg_half_dist2
    covar2 = (1.0 + math.sqrt(5) *
scaled_dist + (5.0 / 3.0) * dist2) * torch
.exp(-math.sqrt(5) * scaled_dist)

    covar = self.alpha * covar1 + (1 -
self.alpha) * covar2

    if diag:
        return covar.diagonal(dim1=-1,
dim2=-2)
    return covar
```

Figure 13: Functional redundancy example from direct code generation. Both snippets compute $k(x,x') = \alpha\exp(-d^2/2) + (1-\alpha)(1+\sqrt{5}d+\frac{5}{3}d^2)\exp(-\sqrt{5}d)$. Green denotes variable renaming, and red denotes syntax-level variation.

```
def forward(self, x1, x2, diag=False, **
params):

    x1s = x1 / self.lengthscale
    x2s = x2 / self.lengthscale

    dist2 = torch.cdist(x1s, x2s, p=2).pow
(2)

    radial_covar = torch.exp(-dist2 / self
.alpha).clamp(max=1e15)
    negative_power = (1 + dist2).pow(-self.
beta).clamp(min=1e-15)

    covar = radial_covar * negative_power


    if diag:
        return covar.diagonal(dim1=-2,
dim2=-1)
    return covar
```

```
def forward(self, x1, x2, diag=False, **
params):
    device = x1.device
    x1s = x1 / self.lengthscale.to(device)
    x2s = x2 / self.lengthscale.to(device)

    dist2 = torch.cdist(x1s, x2s, p=2).pow
(2)

    radial_covar = torch.exp(-dist2 / self
.alpha.to(device))
    reciprocal_power = 1.0 / (1.0 + dist2).
pow(self.beta.to(device))

    combined_covar = radial_covar *
reciprocal_power

    if diag:
        return combined_covar.diagonal(
dim1=-2, dim2=-1)
    return combined_covar
```

Figure 14: Functional redundancy example from direct code generation. Both snippets compute $k(x,x') = \exp(-d^2/\alpha)(1+d^2)^{-\beta}$. Green denotes variable renaming, red denotes syntax-level variation, and blue denotes algebraic rewriting.

H Extended Ablation Studies

Component Ablation on Other Benchmarks In Figure 6, we conduct an ablation study on each component of our method on the SVM benchmark. In this section, we summarize the ablation studies on other benchmarks, extending several combinations of our methods. As shown in Table 9, removing any single component consistently degrades performance. Notably, omitting the Discovery stage or both the Discovery and the Initialization stage results in the most severe performance drops, second only to removing all components. This suggests that these components are crucial design choices for the efficacy of the kernel discovery pipeline in high-dimensional BO.

Table 9: Ablation study of Kernel Discovery components across standard benchmarks. D denotes the dimensionality of the task. **Bold** denotes the best entry in the column. Experiments are conducted with 4 random seeds.

Method	**Rover** (↑) ($D = 100$)	**Mopta08** (↓) ($D = 124$)	**Lasso-DNA** (↓) ($D = 180$)	**SVM388** (↓) ($D = 388$)	**Humanoid** (↑) ($D = 6392$)	**Avg Rank** (↓)
Kernel Discovery (Ours)	**4.353 ± 0.319**	**216.81 ± 0.87**	**0.286 ± 0.001**	**0.056 ± 0.003**	**762.78 ± 72.39**	**1.0 / 8**
w.o. Init	3.837 ± 0.390	217.60 ± 2.34	0.287 ± 0.001	0.071 ± 0.011	608.40 ± 63.92	4.9 / 8
w.o. Discovery	3.810 ± 0.448	220.42 ± 0.95	0.288 ± 0.001	0.064 ± 0.003	720.41 ± 41.77	5.2 / 8
w.o. Eval	3.928 ± 0.402	219.42 ± 2.37	0.289 ± 0.001	0.057 ± 0.003	712.91 ± 99.62	4.0 / 8
w.o. Init + Discovery	4.153 ± 0.471	217.47 ± 0.65	0.293 ± 0.003	0.079 ± 0.009	557.07 ± 117.39	5.4 / 8
w.o. Init + Eval	4.132 ± 0.493	218.70 ± 1.55	0.291 ± 0.004	0.065 ± 0.010	610.89 ± 73.68	5.0 / 8
w.o. Discovery + Eval	3.908 ± 0.415	220.20 ± 0.59	0.287 ± 0.001	0.063 ± 0.002	705.71 ± 10.51	4.3 / 8
w.o. Init + Discovery + Eval	3.905 ± 0.321	218.56 ± 1.83	0.290 ± 0.001	0.087 ± 0.009	569.78 ± 88.46	6.2 / 8

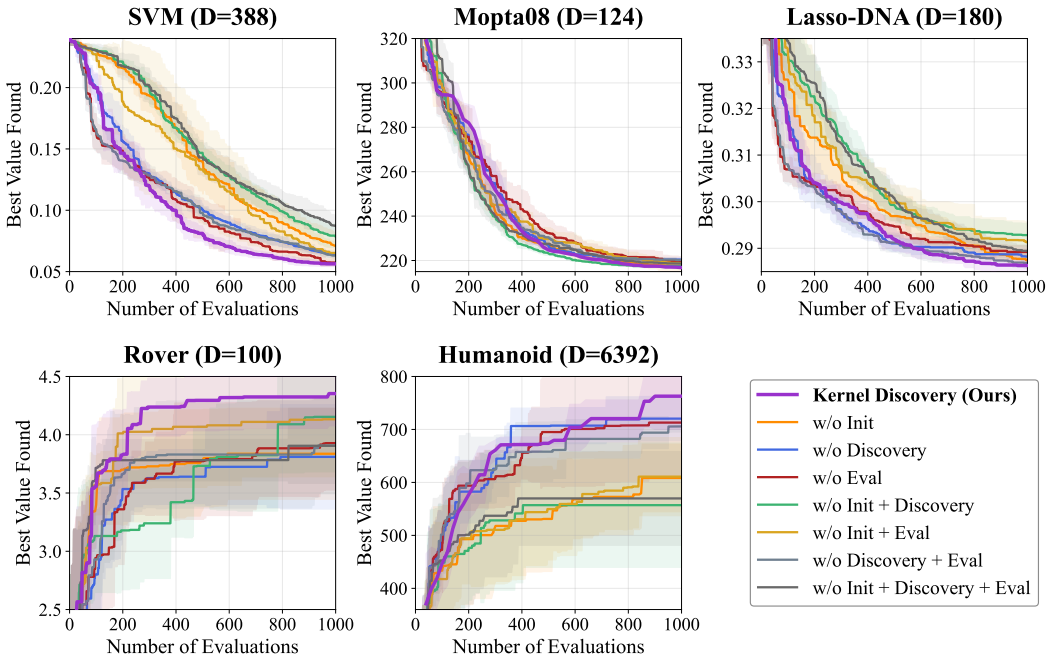


Figure 15: Ablation studies on each component of our method across other benchmarks.

More Ablations on Base Population. For a fair comparison, we also initialize the base populations of the search-based methods, Compositional Search and CAKE, in the same way as in our method. We also conduct ablation studies on the base population for those baselines. As shown in Figure 16, the performance of those methods significantly degrades when we remove BOCK and SL kernels from the base population. While our method also exhibits low sample efficiency without those kernels, it consistently improves the performance through discovering novel kernel structures.

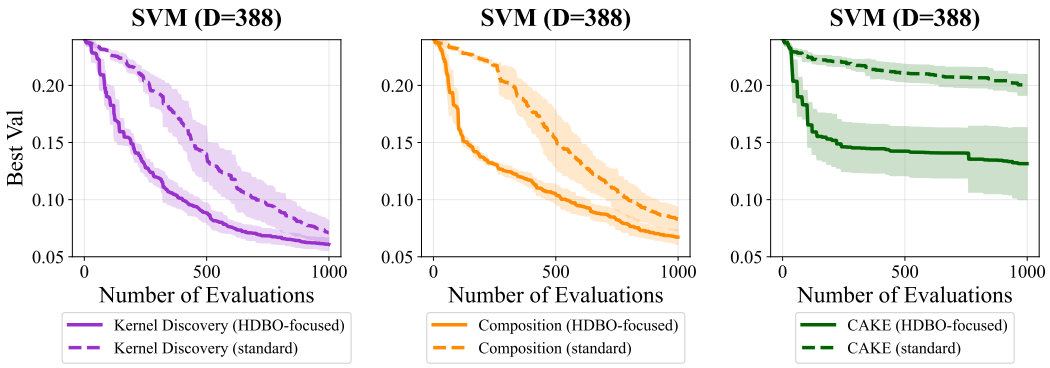


Figure 16: Ablation studies on the base population for Compositional Search and CAKE baselines.

More Ablations on Evaluation Metric. To follow the original implementation, we use the marginal log-likelihood (MLL) to select a kernel for Compositional Search and BIC-Acquisition Kernel Ranking (BAKER) to select a kernel for CAKE. To analyze the effect of the proposed evaluation metric. We compare performance across different evaluation metrics for both our kernel discovery pipeline and search-based baselines. As shown in Figure 17, performance degrades when we replace the evaluation metric with MLL, which tends to favor overly complex kernels that are likely to overfit to the current dataset. We also observe that LOO-CRPS improves the performance of search-based baselines, indicating that it is a powerful metric for selecting kernels in high-dimensional BO.

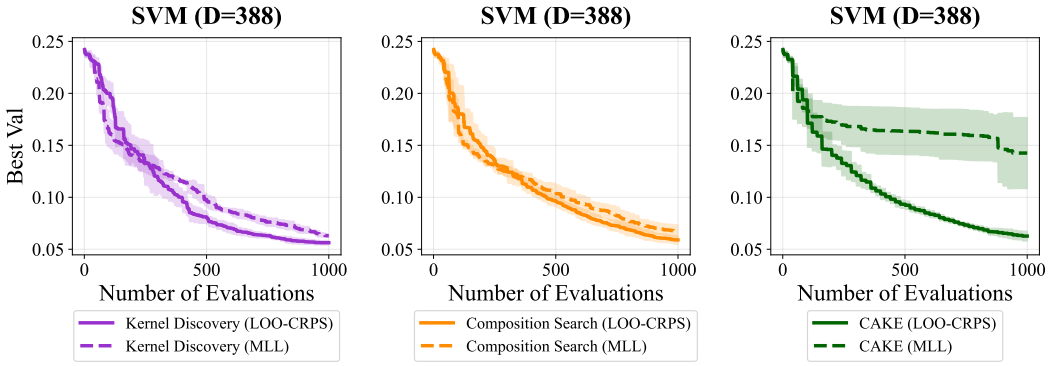


Figure 17: Ablation studies on the evaluation metric for Compositional Search and CAKE baselines.

Representative Kernel Validation Failures. For $\mathcal{V}_{\text{agn}}$, we test whether a generated implementation satisfies the covariance shape requirements. GP fitting only requires the self-covariance matrix $K(\mathbf{X}, \mathbf{X})$, but posterior prediction and acquisition evaluation also require the generally rectangular cross-covariance matrix $K(\mathbf{X}_*, \mathbf{X})$. Accordingly, a common failure mode is that the generated code works for self-covariance calls but breaks under cross-covariance inputs. Figure 18 presents two representative examples.

```
def forward(self, x1, x2, diag=False, **
params):
    # x1: (N1, D), x2: (N2, D)
    # ... input normalization, sphere
projection,
    # and tanh warping ...
    geodesic_kernel = ...       # (N1, N2)
    rq_component_TW = ...       # (N1, N2)
    base_kernel = geodesic_kernel *
rq_component_TW  # (N1, N2)

    I = torch.eye(x1.size(-2), device=x1.
device)[None]  # (1, N1, N1)
    regularized_kernel = base_kernel +
self.regularization_strength * I
    # ERROR: shape mismatch when N1 != N2

    if diag:
        return regularized_kernel.diagonal
(dim1=-2, dim2=-1)
    return regularized_kernel
```

(a) Identity-matrix regularization. The added `torch.eye(N1)` creates a square $N_1 \times N_1$ term, which is only compatible with the covariance tensor shape $N_1 \times N_2$ when $N_1 = N_2$.

```
def forward(self, x1, x2, diag=False, **
params):
    # x1: (N1, D), x2: (N2, D)
    x1 = (x1 - center) / lengthscale
    x2 = (x2 - center) / lengthscale

    dist2 = torch.cdist(x1, x2).square()
    # dist2: (N1, N2)

    r1 = dist2.diagonal(dim1=-2, dim2=-1)
    # r1: (min(N1, N2),)

    angular_kernel = ...
    # angular_kernel: (N1, N2)

    radial_kernel = self.
radial_base_kernel(r1.sqrt(), **params)
    # radial_kernel : (min, min)

    return angular_kernel * radial_kernel
    # ERROR: shape mismatch when N1 != N2
```

(b) Implicit square term. The diagonal extraction and sub-kernel call produce an $M \times M$ block with $M = \min(N_1, N_2)$, which is incompatible with the $N_1 \times N_2$ angular term.

Figure 18: Representative cross-covariance failures caught by $\mathcal{V}_{\text{agn}}$. Both implementations pass $K(\mathbf{X}, \mathbf{X})$ but fail on rectangular $K(\mathbf{X}_*, \mathbf{X})$ calls. Problematic lines are highlighted in red.

For $\mathcal{V}_{\text{psd}}$, a generated implementation may satisfy the required output shape but still fail to define a valid covariance function. A valid GP kernel must produce a positive semi-definite Gram matrix for any finite input set. A common failure mode is that the generated code adds terms that are not generally PSD-preserving, such as distance-increasing components or oscillatory transformations of pairwise distances. Such terms can yield indefinite Gram matrices, as illustrated in Figure 19.

```
def forward(self, x1, x2, diag=False, **
params):
    x1s = x1 / self.lengthscale
    x2s = x2 / self.lengthscale

    dist_sq = torch.cdist(x1s, x2s).pow(2)

    rbf_term = torch.exp(-dist_sq)
    dist_term = torch.sqrt(dist_sq)

    covar = self.alpha * rbf_term + self.
beta * dist_term + self.gamma

    if diag:
        return covar.diagonal(dim1=-2,
dim2=-1)
    return covar
```

(a) Distance-increasing term. This corresponds to $k(x, x') = \alpha e^{-d^2} + \beta d + \gamma$. The added distance term $d = \|x - x'\|$ is not generally PSD-preserving.

```
def forward(self, x1, x2, diag=False, **
params):
    x1s = x1 / self.lengthscale
    x2s = x2 / self.lengthscale

    dist_sq = torch.cdist(x1s, x2s).pow(2)

    base_term = torch.exp(-0.5 * dist_sq)
    osc_term = torch.cos(math.pi*dist_sq)

    covar = self.alpha * base_term + \
        (1 - self.alpha) * osc_term

    if diag:
        return covar.diagonal(dim1=-2,
dim2=-1)
    return covar
```

(b) Oscillatory distance transform. This corresponds to $k(x, x') = \alpha e^{-d^2/2} + (1-\alpha)\cos(\pi d^2)$. The oscillatory term $\cos(\pi d^2)$ is not generally PSD-preserving.

Figure 19: Representative PSD failures caught by $\mathcal{V}_{\text{psd}}$. Both include terms that do not generally preserve positive semi-definiteness and can yield indefinite Gram matrices. Problematic terms are highlighted in red.

I Robustness to Different LLMs

Different LLMs for Other Baselines. We additionally evaluate LLM-based baselines under different LLM backbones to examine whether the performance gap is sensitive to the choice of LLM. Specifically, we compare CAKE and LMABO using GPT-4o and GPT-4o-mini, and report the results together with our method in Table 10. Across the evaluated settings, our method consistently achieves stronger BO performance, suggesting that the performance gain comes from the kernel discovery framework rather than merely from the choice of LLM backbone.

Table 10: Performance of LLM-based methods under different LLM backbones. Results are averaged over 4 random seeds. Bold denotes the best entry in each column.

Method	**Rover** (↑) ($D = 100$)	**Mopta08** (↓) ($D = 124$)	**Lasso-DNA** (↓) ($D = 180$)	**SVM388** (↓) ($D = 388$)	**Humanoid** (↑) ($D = 6392$)
Kernel Discovery (GPT-4o, Ours)	**4.353 ± 0.319**	**216.81 ± 0.87**	**0.286 ± 0.001**	**0.056 ± 0.003**	**762.78 ± 72.39**
Kernel Discovery (GPT-4o-mini, Ours)	3.582 ± 0.230	219.16 ± 1.54	0.289 ± 0.002	0.061 ± 0.004	589.28 ± 107.87
CAKE (GPT-4o)	3.412 ± 0.586	231.40 ± 15.67	0.300 ± 0.011	0.131 ± 0.036	667.49 ± 24.99
CAKE (GPT-4o-mini)	3.863 ± 0.497	229.43 ± 3.88	0.290 ± 0.001	0.103 ± 0.058	648.13 ± 29.11
LMABO (GPT-4o)	3.953 ± 0.369	240.70 ± 5.24	0.304 ± 0.006	0.226 ± 0.001	390.23 ± 44.78
LMABO (GPT-4o-mini)	4.078 ± 0.299	244.21 ± 3.97	0.302 ± 0.005	0.218 ± 0.016	410.09 ± 42.34

Open-source LLMs for Kernel Discovery. We further evaluate whether our kernel discovery pipeline can operate with a smaller open-source LLM. To this end, we replace the kernel-proposing LLM with Qwen3-8B while keeping the rest of the pipeline unchanged. As shown in Table 11, GPT-4o achieves the best performance, but the Qwen3-8B variant still obtains competitive results across benchmarks. Compared with the main results in Table 1, the open-source variant remains comparable to strong baseline methods, indicating that our framework does not rely exclusively on a proprietary LLM. These results suggest that stronger LLMs improve the quality of discovered kernels, while the proposed pipeline remains effective even with a relatively small open-source model.

Table 11: Performance of Kernel Discovery with proprietary and open-source LLMs. Results are averaged over 4 random seeds. Bold denotes the best entry in each column.

Method	**Rover** (↑) ($D = 100$)	**Mopta08** (↓) ($D = 124$)	**Lasso-DNA** (↓) ($D = 180$)	**SVM388** (↓) ($D = 388$)	**Humanoid** (↑) ($D = 6392$)
Kernel Discovery (GPT-4o, Ours)	**4.353 ± 0.319**	**216.81 ± 0.87**	**0.286 ± 0.001**	**0.056 ± 0.003**	**762.78 ± 72.39**
Kernel Discovery (Qwen3-8B, Ours)	3.838 ± 0.293	221.67 ± 1.53	0.288 ± 0.002	0.058 ± 0.003	622.15 ± 107.11

J Evolution of Discovered Kernels

Detailed Explanation of Discovered Kernels in Figure 7a. We provide an explanation of the discovered kernels highlighted in the figure. We first define the kernel components used in the discovered kernels, and then describe each discovered kernel by its decomposition and its interpretations.

Kernel components. We first define the kernel components that appear in the discovered kernels. For the RQ component evaluated in the original input space, let $\boldsymbol{\ell}_B \in \mathbb{R}^d_{>0}$ be an ARD lengthscale, $r_{\mathrm{RQ}} = \|\mathbf{x} \oslash \boldsymbol{\ell}_B - \mathbf{x}' \oslash \boldsymbol{\ell}_B\|_2$, and $\alpha > 0$. Then

$$k_{\mathrm{RQ}}(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{r_{\mathrm{RQ}}^2}{2\alpha}\right)^{-\alpha}.$$

This component captures multi-scale smooth variation in a separately ARD-scaled input space.

Several other components are evaluated after an arctangent-based input transformation. Given an input $\mathbf{x} \in \mathbb{R}^d$ and an ARD lengthscale $\boldsymbol{\ell} \in \mathbb{R}^d_{>0}$, define

$$\mathbf{z}_0(\mathbf{x}) = \mathbf{x} \oslash \boldsymbol{\ell}, \qquad \mathbf{z}_i(\mathbf{x}) = \frac{1}{\sqrt{i}} \arctan(sw\, \mathbf{z}_{i-1}(\mathbf{x})), \quad i = 1, \dots, D,$$

where $s, w > 0$ are scalar transformation parameters and $D \in \mathbb{N}$ is the number of transformation layers. We denote the final transformed feature by $t(\mathbf{x}) = \mathbf{z}_D(\mathbf{x})$. Using this transformed feature, we define

$$k_{\text{Arc-IMQ}}(\mathbf{x}, \mathbf{x}') = \left(1 + \|t(\mathbf{x}) - t(\mathbf{x}')\|_2^2\right)^{-1}, \quad k_{\text{Arc-Lin}}(\mathbf{x}, \mathbf{x}') = t(\mathbf{x})^\top t(\mathbf{x}'),$$

and

$$k_{\text{Arc-RQ}}(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\|t(\mathbf{x}) - t(\mathbf{x}')\|_2^2}{2\alpha}\right)^{-\alpha}.$$

Here, the Arc-IMQ component captures heavy-tailed distance-based similarity in the arctangent-transformed feature space, the Arc-Linear component captures global alignment in the same transformed space, and the Arc-RQ component captures multi-scale smooth variation after the arctangent transformation.

One discovered kernel additionally uses an angular similarity component. Let

$$\tilde{\mathbf{x}} = \frac{\mathbf{x} - \mathbf{c}}{\mathbf{r}}, \qquad \rho(\mathbf{x}) = \|\tilde{\mathbf{x}}\|_2, \qquad \mathbf{a}(\mathbf{x}) = \frac{\tilde{\mathbf{x}}}{\rho(\mathbf{x})}.$$

Here $\mathbf{a}(\mathbf{x})$ is the unit direction vector of the normalized input. The polynomial angular kernel is

$$k_{\mathrm{ang}}(\mathbf{x}, \mathbf{x}') = \sum_{p=0}^{3} w_p \left(\mathbf{a}(\mathbf{x})^\top \mathbf{a}(\mathbf{x}')\right)^p, \qquad w_p > 0.$$

This component captures directional similarity relative to the normalized center.

Kernel 1:

$$k_1(\mathbf{x}, \mathbf{x}') = k_{\text{Arc-IMQ}}(\mathbf{x}, \mathbf{x}') + k_{\text{Arc-Lin}}(\mathbf{x}, \mathbf{x}') + k_{\mathrm{RQ}}(\mathbf{x}, \mathbf{x}').$$

This kernel combines heavy-tailed distance-based similarity and linear alignment in the arctangent-transformed feature space with multi-scale smooth variation in a separate ARD-scaled input space.

Kernel 2:

$$k_2(\mathbf{x}, \mathbf{x}') = k_{\mathrm{ang}}(\mathbf{x}, \mathbf{x}') \cdot \left[k_{\text{Arc-IMQ}}(\mathbf{x}, \mathbf{x}') + k_{\text{Arc-Lin}}(\mathbf{x}, \mathbf{x}')\right].$$

This kernel is a product of a polynomial angular kernel and an arc-transformed additive kernel. The angular component captures directional similarity relative to a normalized center, while the arc component captures heavy-tailed distance-based similarity and linear alignment in the arctangent-transformed feature space.

Kernel 3:

$$k_3(\mathbf{x}, \mathbf{x}') = k_{\text{Arc-RQ}}(\mathbf{x}, \mathbf{x}') + k_{\text{Arc-IMQ}}(\mathbf{x}, \mathbf{x}') + k_{\text{Arc-Lin}}(\mathbf{x}, \mathbf{x}').$$

This kernel evaluates all three components in the same arctangent-transformed feature space. It combines multi-scale smooth variation, heavy-tailed distance-based similarity, and global transformed-space alignment under a shared nonlinear input warping.

We also provide the corresponding `forward` implementations for the discovered kernels.

```
def forward(self, x1, x2, diag=False, **params):
    if x1.dim() == 1: x1 = x1.unsqueeze(-1)
    if x2.dim() == 1: x2 = x2.unsqueeze(-1)

    # Kernel A: Arc-transformed IMQ + Arc-transformed Linear
    x1_scaled_a = x1 / self.lengthscale_a
    x2_scaled_a = x2 / self.lengthscale_a
    t1, t2 = x1_scaled_a, x2_scaled_a
    for i in range(1, self.depth.item() + 1):
        t1 = torch.atan(self.global_scale * self.arc_weight * t1) / math.sqrt(i)
        t2 = torch.atan(self.global_scale * self.arc_weight * t2) / math.sqrt(i)
    linear_term = t1 @ t2.transpose(-1, -2)
    r_squared_a = torch.cdist(t1, t2, p=2).pow(2)
    k_a = (1 + r_squared_a).pow(-1) + linear_term

    # Kernel B: Rational Quadratic
    x1_scaled_b = x1 / self.lengthscale_b
    x2_scaled_b = x2 / self.lengthscale_b
    r_squared_b = torch.cdist(x1_scaled_b, x2_scaled_b, p=2).pow(2)
    k_b = (1 + r_squared_b / (2 * self.alpha)).pow(-self.alpha)

    covar = k_a + k_b
    if diag:
        return covar.diagonal(dim1=-2, dim2=-1)
    return covar
```

Figure 20: Forward code for the discovered kernel k_1.

```
def forward(self, x1, x2, diag=False, **params):
    if x1.dim() == 1: x1 = x1.unsqueeze(-1)
    if x2.dim() == 1: x2 = x2.unsqueeze(-1)
    arc_weight = self.raw_arc_weight_constraint.transform(self.raw_arc_weight)
    global_scale = self.raw_global_scale_constraint.transform(self.raw_global_scale)
    angular_weights = self.raw_angular_weights_constraint.transform(self.raw_angular_weights)

    x1_normalized = (x1 - self.center) / self.radius
    x2_normalized = (x2 - self.center) / self.radius
    r1 = x1_normalized.norm(dim=-1, keepdim=True)
    r2 = x2_normalized.norm(dim=-1, keepdim=True)
    a1 = x1_normalized / r1.clamp(min=1e-15)
    a2 = x2_normalized / r2.clamp(min=1e-15)

    t1 = x1_normalized / self.lengthscale
    t2 = x2_normalized / self.lengthscale
    for i in range(1, self.depth + 1):
        t1 = torch.atan(global_scale * arc_weight * t1) / math.sqrt(i)
        t2 = torch.atan(global_scale * arc_weight * t2) / math.sqrt(i)

    angular_cov = self._angular_kernel(a1, a2, angular_weights)
    t1_dist = torch.cdist(t1, t2, p=2)
    arc_cov = (1 + t1_dist.pow(2)).pow(-1) + (t1 @ t2.transpose(-1, -2))

    covar = angular_cov * arc_cov
    if diag:
        return covar.diagonal(dim1=-2, dim2=-1)
    return covar
```

Figure 21: Forward code for the discovered kernel k_2.

```python
def forward(self, x1, x2, diag=False, **params):
    if x1.dim() == 1: x1 = x1.unsqueeze(-1)
    if x2.dim() == 1: x2 = x2.unsqueeze(-1)

    # Kernel A: Arc-transformed IMQ + Arc-transformed Linear
    x1_scaled_a = x1 / self.lengthscale_a
    x2_scaled_a = x2 / self.lengthscale_a
    t1, t2 = x1_scaled_a, x2_scaled_a
    for i in range(1, self.depth.item() + 1):
        t1 = torch.atan(self.global_scale * self.arc_weight * t1) / math.sqrt(i)
        t2 = torch.atan(self.global_scale * self.arc_weight * t2) / math.sqrt(i)
    linear_term = t1 @ t2.transpose(-1, -2)
    r_squared_a = torch.cdist(t1, t2, p=2).pow(2)
    k_a = (1 + r_squared_a).pow(-1) + linear_term

    # Kernel B: Rational Quadratic
    x1_scaled_b = x1 / self.lengthscale_b
    x2_scaled_b = x2 / self.lengthscale_b
    r_squared_b = torch.cdist(x1_scaled_b, x2_scaled_b, p=2).pow(2)
    k_b = (1 + r_squared_b / (2 * self.alpha)).pow(-self.alpha)

    covar = k_a + k_b
    if diag:
        return covar.diagonal(dim1=-2, dim2=-1)
    return covar
```

Figure 22: Forward code for the discovered kernel k_3.

Evolving Population on Other Benchmarks Figure 23 extends the post-analysis of Figure 7a to four additional benchmarks: Rover, Mopta08, Lasso-DNA, and Humanoid. For each benchmark, we visualize the best-so-far value trajectory and annotate the kernels selected at the top-5 largest improvements. The components of the annotated kernels are described using the same kernel vocabulary introduced above.

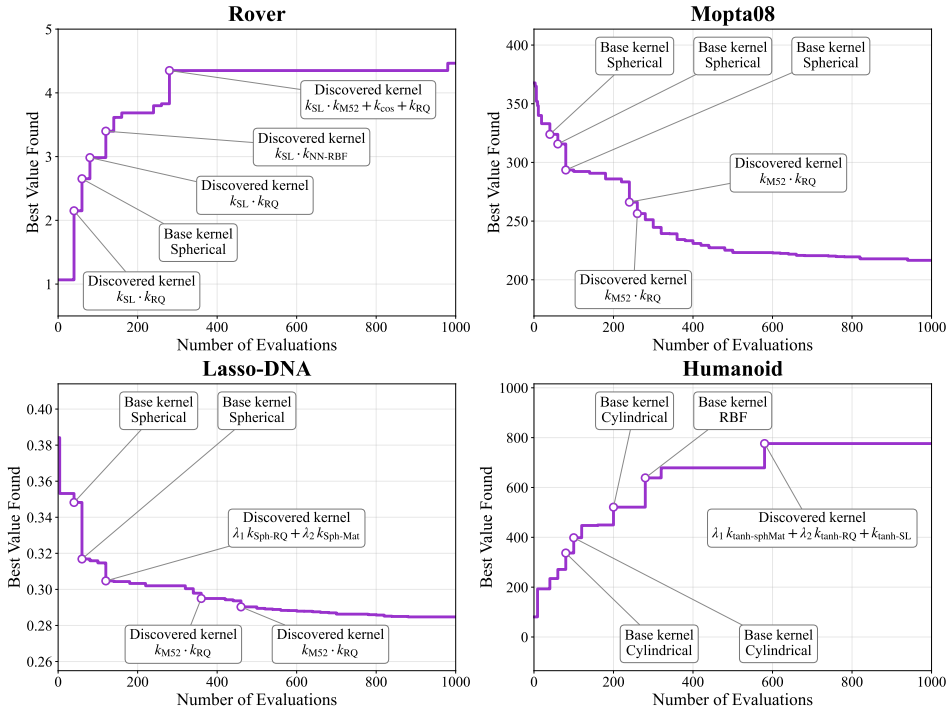


Figure 23: Post-analysis of kernel discovery on four additional benchmarks. For each benchmark, we annotate the kernels selected at the top-5 largest improvements in the best-so-far value.

For the additional discovered kernels annotated in Figure 23, we provide the mathematical definitions of the component kernels used in the decompositions.

Matérn-$5/2$ component. Let

$$r_{\text{Matérn52}} = \|\mathbf{x} \oslash \boldsymbol{\ell}_{\text{Matérn52}} - \mathbf{x}' \oslash \boldsymbol{\ell}_{\text{Matérn52}}\|_2 .$$

Then

$$k_{\text{Matérn52}}(\mathbf{x}, \mathbf{x}') = \left(1 + \sqrt{5} r_{\text{Matérn52}} + \frac{5}{3} r^2_{\text{Matérn52}}\right) \exp(-\sqrt{5} r_{\text{Matérn52}}).$$

Spherical lifted linear component. Given a scaled input $\mathbf{u}(\mathbf{x})$, define the stereographic projection

$$\psi(\mathbf{u}(\mathbf{x})) = \frac{\left[2\mathbf{u}(\mathbf{x}),\ \|\mathbf{u}(\mathbf{x})\|_2^2 - 1\right]}{\|\mathbf{u}(\mathbf{x})\|_2^2 + 1}.$$

Then

$$k_{\text{SL}}(\mathbf{x}, \mathbf{x}') = \lambda_0 + \lambda_1 \psi(\mathbf{u}(\mathbf{x}))^\top \psi(\mathbf{u}(\mathbf{x}')), \qquad \lambda_0 + \lambda_1 = 1,\ \lambda_0, \lambda_1 \geq 0.$$

Neural-network RBF component. Let

$$\mathbf{h}(\mathbf{x}) = \text{NN}(\mathbf{x} \oslash \boldsymbol{\ell}_{\text{NN}}) .$$

Then

$$k_{\text{NN-RBF}}(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{\|\mathbf{h}(\mathbf{x}) - \mathbf{h}(\mathbf{x}')\|_2^2}{2\sigma^2_{\text{NN}}}\right).$$

Cosine warping component. Define

$$\mathbf{v}(\mathbf{x}) = \tanh(\mathbf{x}/r_{\cos}), \qquad \rho_{\cos}(\mathbf{x}) = \|\mathbf{v}(\mathbf{x})\|_2.$$

Then

$$k_{\cos}(\mathbf{x}, \mathbf{x}') = \cos\left(2\pi\alpha_{\cos} \left|\rho_{\cos}(\mathbf{x}) - \rho_{\cos}(\mathbf{x}')\right|\right).$$

Spherical RQ and spherical Matérn-like components. Using the stereographic projection $\psi(\mathbf{u}(\mathbf{x}))$, define

$$d_{\text{sph}} = \|\psi(\mathbf{u}(\mathbf{x})) - \psi(\mathbf{u}(\mathbf{x}'))\|_2.$$

The spherical RQ component is

$$k_{\text{sphRQ}}(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{d^2_{\text{sph}}}{2\alpha}\right)^{-\alpha},$$

and the spherical Matérn-like component is

$$k_{\text{sphMat}}(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\sqrt{3}}{\nu} d_{\text{sph}} + \frac{3}{\nu^2} d^2_{\text{sph}}\right) \exp\left(-\frac{\sqrt{3}}{\nu} d_{\text{sph}}\right).$$

where $\nu > 0$ is a learnable parameter that controls the effective length scale and smoothness of the Matérn-like component.

Tanh-transformed components. Define the tanh embedding

$$\mathbf{h}_{\tanh}(\mathbf{x}) = \tanh\left(\alpha(\mathbf{x} \oslash \boldsymbol{\ell})\right).$$

For the spherical branch, define

$$\mathbf{u}_{\tanh}(\mathbf{x}) = (\mathbf{h}_{\tanh}(\mathbf{x}) - \mathbf{c}) \oslash \boldsymbol{\ell},$$

and let $\psi_{\tanh}(\mathbf{x}) = \psi(\mathbf{u}_{\tanh}(\mathbf{x}))$ be the stereographic projection of $\mathbf{u}_{\tanh}(\mathbf{x})$. Define

$$r_{\tanh\text{-sph}} = \|\psi_{\tanh}(\mathbf{x}) - \psi_{\tanh}(\mathbf{x}')\|_2.$$

Then

$$k_{\text{tanh-sphMat}}(\mathbf{x}, \mathbf{x}') = \left(1 + \sqrt{2\nu} r_{\tanh\text{-sph}} + \frac{2\nu}{3} r^2_{\tanh\text{-sph}}\right) \exp\left(-\sqrt{2\nu} r_{\tanh\text{-sph}}\right),$$

$$k_{\text{tanh-RQ}}(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{\|\mathbf{h}_{\tanh}(\mathbf{x}) - \mathbf{h}_{\tanh}(\mathbf{x}')\|_2^2}{2\gamma}\right)^{-\gamma},$$

and

$$k_{\text{tanh-SL}}(\mathbf{x}, \mathbf{x}') = \psi_{\tanh}(\mathbf{x})^\top \psi_{\tanh}(\mathbf{x}').$$

K Transferability of Discovered Kernels

Experiment Results on Other Benchmarks. In Figure 7b, we visualize the results of the kernel discovered by our pipeline on the SVM benchmark. In this section, we report the kernel's results across the other benchmarks. As shown in Table 12, it consistently achieves competitive and often outperforms other base kernels. We also visualize the code snippet of the kernel in Figure 25.

Table 12: Transferability of the discovered kernel across other benchmarks. We compare the kernel discovered on the SVM benchmark, when transferred to the other four standard benchmarks, against base kernels. D denotes the dimensionality of the task. **Blue** denotes the best entry in the column, and **Violet** denotes the second best. Experiments are conducted with 4 random seeds.

Method	**Rover** (↑) ($D = 100$)	**Mopta08** (↓) ($D = 124$)	**Lasso-DNA** (↓) ($D = 180$)	**SVM** (↓) ($D = 388$)	**Humanoid** (↑) ($D = 6392$)	**Average Rank**
Base Kernels						
RBF	3.552 ± 0.304	217.75 ± 2.33	**0.291 ± 0.001**	**0.061 ± 0.004**	503.00 ± 67.82	3.8 / 8
Matérn52	3.453 ± 0.366	**215.77 ± 0.57**	0.292 ± 0.003	0.063 ± 0.003	509.79 ± 102.32	4.0 / 8
Linear	3.950 ± 0.441	274.67 ± 8.87	0.312 ± 0.004	0.226 ± 0.003	435.49 ± 42.00	6.4 / 8
Periodic	3.664 ± 0.231	309.10 ± 5.01	0.332 ± 0.005	0.226 ± 0.002	413.83 ± 71.30	7.4 / 8
BOCK	3.725 ± 0.593	225.36 ± 1.82	**0.291 ± 0.003**	0.068 ± 0.007	**669.52 ± 50.15**	**3.4 / 8**
SL	**4.096 ± 0.473**	246.78 ± 3.92	0.297 ± 0.001	0.112 ± 0.007	**637.72 ± 37.86**	4.4 / 8
RQ	3.858 ± 0.515	**217.53 ± 4.25**	0.294 ± 0.002	0.069 ± 0.017	600.80 ± 152.49	3.8 / 8
Discovered Kernel (Ours)	**4.238 ± 0.640**	218.30 ± 1.00	0.296 ± 0.004	**0.052 ± 0.006**	629.12 ± 122.84	**2.8 / 8**

Analysis on Boundary-seeking Behavior. To understand why the discovered kernel k_{discover} in Equation (11) achieves superior performance on the SVM benchmark, we analyze whether its non-stationary component $k_{\text{tanh-Poly}}$ induces boundary-seeking behavior. We measure two complementary statistics over BO iterations: (1) the boundary hit ratio, i.e., the fraction of queried points lying on the boundary of the search domain, and (2) the observation traveling salesman distance (OTSD), which measures the total path length traversed across sequential queries. As shown in Figure 24, the discovered kernel maintains a boundary hit ratio comparable to RBF throughout optimization, and substantially lower than the Linear kernel, on both the SVM and Mopta08 benchmarks. The OTSD of the discovered kernel similarly remains low and closely tracks that of RBF, indicating that consecutive queries tend to be spatially concentrated rather than widely scattered. Together, these results suggest that the non-stationary component is regularized during GP hyperparameter optimization, and that the combination of several kernel components encourages a locally concentrated query behavior, which is a key factor enabling the discovered kernel to consistently identify higher-quality candidates.

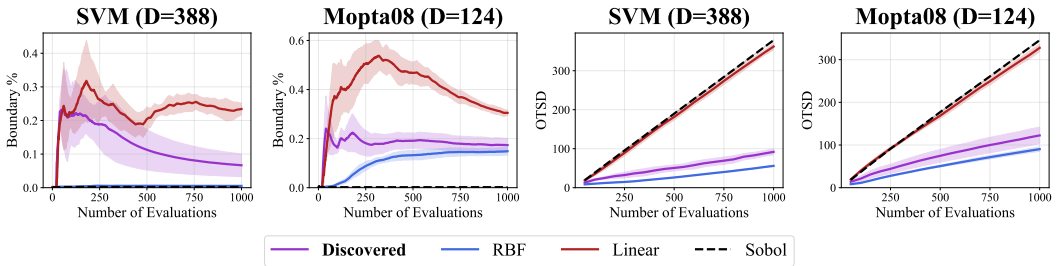


Figure 24: Analysis on boundary-seeking behavior of several kernels.

Detailed Explanation of the Discovered Kernel in Equation (11). We provide a detailed explanation of the discovered kernel in Equation (11). This kernel combines a Matérn-$5/2$ component with an additive composition of a tanh-polynomial component and an RQ component:

$$k_{\text{discover}}(\mathbf{x}, \mathbf{x}') = k_{\text{Matérn52}}(\mathbf{x}, \mathbf{x}') \cdot \left[k_{\text{tanh-Poly}}(\mathbf{x}, \mathbf{x}') + k_{\text{RQ}}(\mathbf{x}, \mathbf{x}')\right].$$

Matérn-$5/2$ component. Let $\boldsymbol{\ell} \in \mathbb{R}^d_{>0}$ be an ARD lengthscale and define

$$r_{\text{Matérn52}}(\mathbf{x}, \mathbf{x}') = \left\|\mathbf{x} \oslash \boldsymbol{\ell} - \mathbf{x}' \oslash \boldsymbol{\ell}\right\|_2.$$

The Matérn-$5/2$ component is

$$k_{\text{Matérn52}}(\mathbf{x}, \mathbf{x}') = \left(1 + \sqrt{5} r_{\text{Matérn52}} + \frac{5}{3} r^2_{\text{Matérn52}}\right) \exp\left(-\sqrt{5} r_{\text{Matérn52}}\right).$$

This component captures moderately smooth local correlation in the ARD-scaled input space.

Tanh-polynomial component. Using the same ARD-scaled input, define the tanh-transformed feature

$$\mathbf{h}_{\tanh}(\mathbf{x}) = \tanh\left(\sigma\left(\mathbf{x} \oslash \boldsymbol{\ell}\right)\right),$$

where $\sigma > 0$ controls the strength of the tanh projection. The tanh-polynomial component is

$$k_{\text{tanh-Poly}}(\mathbf{x}, \mathbf{x}') = \sum_{n=0}^{2} w_n \left(\mathbf{h}_{\tanh}(\mathbf{x})^\top \mathbf{h}_{\tanh}(\mathbf{x}')\right)^n, \qquad w_n > 0.$$

This component captures polynomial similarity after mapping inputs into a bounded nonlinear feature space.

Rational Quadratic component. Let

$$r_{\text{RQ}}(\mathbf{x}, \mathbf{x}') = \left\|\mathbf{x} \oslash \boldsymbol{\ell} - \mathbf{x}' \oslash \boldsymbol{\ell}\right\|_2.$$

The RQ component is

$$k_{\text{RQ}}(\mathbf{x}, \mathbf{x}') = \left(1 + \frac{r^2_{\text{RQ}}}{2\alpha}\right)^{-\alpha}, \qquad \alpha > 0.$$

This component captures multi-scale smooth variation in the ARD-scaled input space.

Overall, the product structure modulates the additive tanh-polynomial and RQ similarities by the Matérn-$5/2$ local smoothness component.

Illustrative Example of Discovered Kernel. As a representative example, we present the full implementation of the kernel discovered from the SVM benchmark, which corresponds to k_{discover} in Equation (11).

```
def forward(self, x1, x2, diag=False, **params):
    if x1.dim() == 1: x1 = x1.unsqueeze(-1)
    if x2.dim() == 1: x2 = x2.unsqueeze(-1)
    x1s = x1 / self.lengthscale
    x2s = x2 / self.lengthscale

    # Matern-5/2 Component
    r = torch.cdist(x1s, x2s, p=2).clamp(min=1e-15)
    sqrt5_r = math.sqrt(5.0) * r
    k_matern = (1 + sqrt5_r + 5.0/3.0 * r.pow(2)) \
                * torch.exp(-sqrt5_r)

    # Tangential Polynomial Component
    t1 = torch.tanh(self.projection_sigma * x1s)
    t2 = torch.tanh(self.projection_sigma * x2s)
    t_ip = t1 @ t2.transpose(-1, -2)
    k_tanh_poly = sum(
        self.tangent_weights[n] * t_ip.pow(n)
        for n in range(3))

    # Rational Quadratic Component
    r2 = torch.cdist(x1s, x2s, p=2).pow(2)
    k_rq = (1 + r2 / (2 * self.alpha)).pow(-self.alpha)

    covar = k_matern * (k_tanh_poly + k_rq)
    if diag:
        return covar.diagonal(dim1=-2, dim2=-1)
    return covar
```

Figure 25: Code snippet of the discovered kernel from SVM benchmark.

We additionally present two discovered kernels that show strong transferability. Using the same fixed-kernel evaluation protocol as in Table 12, both achieve an overall average rank of 2.8 across five benchmarks, outperforming all standard base kernels in aggregate.

```
def forward(self, x1, x2, diag=False, **params):
    if x1.dim() == 1: x1 = x1.unsqueeze(-1)
    if x2.dim() == 1: x2 = x2.unsqueeze(-1)

    # Rational Quadratic Transform
    x_centered_RQ1 = x1 / self.lengthscale_RQ
    x_centered_RQ2 = x2 / self.lengthscale_RQ
    x_scaled_RQ1 = x_centered_RQ1 / math.sqrt(self.D)
    x_scaled_RQ2 = x_centered_RQ2 / math.sqrt(self.D)

    # Bezier Path Projection
    t = self.harmonic_alpha
    path_bezier = sum((1-t)**(self.control_point_count-i) * t**i * self.
bezier_control_points[..., i, :]
                      for i in range(self.control_point_count))
    x_path_BZ1 = x1 / self.lengthscale_BZ
    x_path_BZ2 = x2 / self.lengthscale_BZ
    x_proj_BZ1 = self._inv_stereographic((x_path_BZ1 / x_path_BZ1.norm(dim
=-1, keepdim=True) + path_bezier) / 2)
    x_proj_BZ2 = self._inv_stereographic((x_path_BZ2 / x_path_BZ2.norm(dim
=-1, keepdim=True) + path_bezier) / 2)
    x_scaled_BZ1 = x_proj_BZ1 / math.sqrt(self.D)
    x_scaled_BZ2 = x_proj_BZ2 / math.sqrt(self.D)

    # Rational Quadratic Kernel
    r_RQ2 = torch.cdist(x_scaled_RQ1, x_scaled_RQ2, p=2).pow(2)
    k_RQ = (1 + r_RQ2 / (2 * self.alpha_RQ)).pow(-self.alpha_RQ)

    # Bezier-Projected Polynomial Kernel
    k_BZ = (x_proj_BZ1 @ x_proj_BZ2.transpose(-1, -2) + 1).pow(3)

    # Hybrid Kernel Covariance
    k = k_RQ + self.harmonic_alpha * k_BZ

    if diag:
        return k.diagonal(dim1=-2, dim2=-1)
    return k
```

Figure 26: Code snippet of an additional discovered kernel with strong transferability. This kernel combines an RQ component with a Bézier-projected polynomial component.

```
def forward(self, x1, x2, diag=False, **params):
    if x1.dim() == 1: x1 = x1.unsqueeze(-1)
    if x2.dim() == 1: x2 = x2.unsqueeze(-1)

    # Rational Quadratic component
    x1_rq = x1 / self.lengthscale
    x2_rq = x2 / self.lengthscale
    r_rq2 = torch.cdist(x1_rq, x2_rq, p=2).pow(2)
    k_rq = (1 + r_rq2 / (2 * self.alpha)).pow(-self.alpha)

    # Spherical component
    x1_sphere = self._project_to_sphere(x1 / self.lengthscale_sphere)
    x2_sphere = self._project_to_sphere(x2 / self.lengthscale_sphere)
    coeffs = self.poly_coeffs_sphere
    phi1 = torch.cat([x1_sphere * coeffs[1].sqrt(), coeffs[0].sqrt() * torch.
ones_like(x1_sphere[..., :1])], dim=-1)
    phi2 = torch.cat([x2_sphere * coeffs[1].sqrt(), coeffs[0].sqrt() * torch.
ones_like(x2_sphere[..., :1])], dim=-1)
    k_sphere = phi1 @ phi2.transpose(-1, -2)

    # Matern-5/2 component
    x1_m = x1 / self.lengthscale_matern52
    x2_m = x2 / self.lengthscale_matern52
    r_m = torch.cdist(x1_m, x2_m, p=2).clamp(min=1e-15)
    k_m = (1 + math.sqrt(5) * r_m + 5 / 3 * r_m.pow(2)) * torch.exp(-math.
sqrt(5) * r_m)

    # Cosmic Warping component
    x1_cosmic = torch.tanh(x1 / self.radius_cosmic)
    x2_cosmic = torch.tanh(x2 / self.radius_cosmic)
    cosmic_phase1 = x1_cosmic.norm(dim=-1, keepdim=True)
    cosmic_phase2 = x2_cosmic.norm(dim=-1, keepdim=True)
    k_cosmic = torch.cos(2 * math.pi * self.alpha_cosmic * torch.cdist(
cosmic_phase1, cosmic_phase2, p=2))

    # Combined Kernel
    covar = k_rq + (k_sphere * k_m) + k_cosmic

    if diag:
        return covar.diagonal(dim1=-2, dim2=-1)
    return covar
```

Figure 27: Code snippet of another additional discovered kernel with strong transferability. This kernel combines RQ, spherical, Matérn-5/2, and warped periodic-like components.