CoTune: Co-evolutionary Configuration Tuning

Gangda Xiong[1] and Tao Chen[2*]
[1] School of Computer Science and Engineering, University of Electronic Science and Technology of China, China
[2] IDEAS Lab, School of Computer Science, University of Birmingham, United Kingdom
gangdaxiong0207@gmail.com, t.chen@bham.ac.uk

***Abstract*—To automatically tune configurations for the best possible system performance (e.g., runtime or throughput), much work has been focused on designing intelligent heuristics in a tuner. However, existing tuner designs have mostly ignored the presence of complex performance requirements (e.g., "`the latency shall ideally be 2 seconds`"), but simply assume that better performance is always more preferred. This would not only waste valuable information in a requirement but might also consume extensive resources to tune for a goal with little gain. Yet, prior studies have shown that simply incorporating the requirement as a tuning objective is problematic since the requirement might be too strict, harming convergence; or its highly diverse satisfactions might lead to premature convergence.**

In this paper, we propose `CoTune`, a tool that takes the information of a given target performance requirement into account through co-evolution. `CoTune` is unique in the sense that it creates an auxiliary performance requirement to be co-evolved with the configurations, which assists the target performance requirement when it becomes ineffective or even misleading, hence allowing the tuning to be guided by the requirement while being robust to its harm. Experiment results on 162 cases (nine systems and 18 requirements) reveal that `CoTune` considerably outperforms existing tuners, ranking as the best for $\approx 90\%$ cases (against the 0%–35% for other tuners) with up to $2.9\times$ overall improvements, while doing so under a much better efficiency.

***Index Terms*—SBSE, compiler/database optimization, performance/hyperparameter optimization, requirement satisfactions**

I. Introduction

Modern software systems expose an ever-growing number of configuration options—ranging from thread/cache sizes to algorithm/architecture choices—offering users unprecedented control over the performance needs, e.g., runtime or throughput [1]–[5]. Yet this flexibility comes at a steep price: in 2017-2018 alone, severe violations of performance requirements caused by poor configuration have occurred in more than 50% of the software companies worldwide, costing $\approx 400,000$ USD per hour in average [6]. As such, configuration tuning is an extremely important task for software quality assurance.

Tuning configuration is not easy, because even a single configuration measurement is expensive, e.g., it can take up to 166 minutes to measure one configuration on a database system [7], let alone the exponentially growing configuration space [1], [2], e.g., the recent version of system Jump3R has 6.87×10^{10} configurations. This renders brute-force tuning infeasible. To automatically tune the configurations for better performance, the tuner design has focused on smart heuristics,

[*] Tao Chen is the corresponding author. Gangda Xiong is also supervised in the IDEAS Lab.

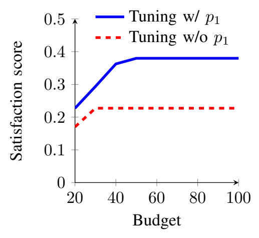
Requirement 4.3.1:	The system should support at least 1000 concurrent users.
Description:	This statement provides a general sense of reliability when the system is under load. It is important that a substantial number of actors be able to access the system at the same time, since a courseware system is important to the courses that employ it. The times when the system will be under the most stress are likely during midterm and finals weeks. Therefore, it must be able to handle at least 1,000 concurrent users.

Fig. 1: A real-world performance requirement from the Puget Sound Enhancements System in the PURE dataset [19].

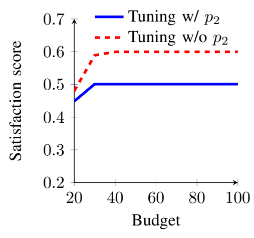
e.g., iterated local search [8], Genetic Algorithm [9]–[15], and Bayesian optimization [16], together with its variants [3], [17].

Despite the rapidly developing tuners, they mostly rely on a simplified assumption: the better the performance, the more preferred, and configuration tuning is all about finding the optimal configuration that achieves the best possible performance [1]–[3], [11], [13], [14]. Indeed, the above is cognitively not incorrect, but the practical scenarios are far more complicated due to the presence of performance requirements [18]: it is not uncommon to see a similar performance need as the one in Figure 1 documented for a system. There are a couple of reasons that it is insufficient to simply assume the better is constantly more preferred: for example, since the configuration measurement is expensive, it might not always be ideal to find the best possible runtime as the improvement when the configuration is already good enough might be minor but achieving so would consume much more budgets. On the other hand, there could be cases where a runtime worse than a certain point is equally unacceptable, although a better runtime than that point might be preferred. The rich information in a given performance requirements can serve as a valuable source to guide the tuning, better finding what is truly satisfied by stakeholders. Indeed, a study from Chen and Li [18] reveals that fitting the requirement as the objective for configuration tuning generally leads to much higher satisfaction than tuning without (see Figure 2a). Further, ignoring the requirement is risky to spend extra resources for tuning more than necessary.

However, directly using a performance requirement as an objective for a tuner, as what has been done in some existing works [13], [14], can be problematic. As hinted by Chen and Li [18], together with our own experiment observations, requirements can sometimes be ineffective or even harmful to the tuning (see Figure 2b): too strict requirements can lead to the loss of search pressure, negatively affecting the convergence of tuning; the highly diverse satisfactions in the requirement might cause stagnation, yielding premature convergence. All



(a) Requirement is helpful



(b) Requirement is harmful

Fig. 2: The satisfaction scores of two cases for tuning with and without a requirement on SQLITE via a tuner based on Genetic Algorithm. p_1 and p_2 denote two different requirements.

those are inevitable since at the requirement elicitation stage, it is difficult to know the feasibility of expectations nor how the performance requirement impacts the tuning.

To address the above gap, in this paper, we present `CoTune`, the first tuner that dynamically "co-evolves" a given performance requirement and configurations for better satisfaction. Unlike others, `CoTune` creates an ***auxiliary*** performance requirement, together with the given ***target*** one, to guide the tuning: the former is co-evolved with the configurations and assists the latter when it becomes useless or harmful. In this way, we neither statically rely on the fixed target performance requirement nor completely ignore it, but take advantage of the rich information therein while catering for its potential limits and harms on-the-fly during tuning. Our contributions are:

- We represent performance requirements in a quantifiable manner via fuzzy logic [20] that fits with the tuning context, enabling intuitive specification by the developers.
- Through cooperative co-evolution [21], we co-evolve requirements and configurations based on the tuning status.
- We exploit differential entropy to measure the discriminative power of the requirement on configurations—the key to mitigating the loss of pressure and stagnation—paired with the corresponding co-evolution mechanisms.
- We evaluate `CoTune` against several state-of-the-art tuners, with and without guidance of performance requirement, under nine systems and 18 possible target performance requirements, leading to 162 cases.

The results demonstrate that `CoTune` is ranked the best for $\approx 90\%$ cases, which is significantly better than the overall best counterpart, with up to $2.9\times$ overall satisfaction improvement and a much superior efficiency. The Code/data can be found at: `https://github.com/ideas-labo/CoTune`.

The paper is organized as follows: Section II introduces preliminaries/motivations. Section III explains `CoTune`. Section IV describes our experimental setup. Section V analyzes the results followed by a discussion in Section VI. Section VII reviews related work and Section VIII concludes the paper.

II. PRELIMINARIES

A. *Configuration Tuning*

Software configuration tuning seeks to automatically find a configuration that optimizes a performance metric of the system (e.g., runtime, throughput, energy). Formally, the goal is to find (assuming minimizing performance metric):

$$\underset{\boldsymbol{c}\in\mathcal{C}}{\arg\min}\ f(\boldsymbol{c}) \tag{1}$$

where $\boldsymbol{c} = (o_1, o_2, \dots, o_n)$ is a configuration such that o_n is a configuration option, which might be binary or enumerate. $\mathcal{C}$ is the configuration space. f denotes measuring the actual system for the performance value of $\boldsymbol{c}$, which is time-consuming.

B. *Tuning with Performance Requirements*

It is possible that one would prefer the best possible performance, e.g., the smaller the runtime, the better, hence we only need to optimize Equation 1. However, there are several reasons in practice that it is not always a desired situation. For example, since configuration tuning is expensive, seeking the best possible performance might consume a significant amount of resources and time for little gain [22]. Indeed, we have seen tuning that achieves a 2 seconds runtime under around one hour budget, but pushing it towards 1.8 seconds requires another 12 hours to do so: the benefit does not seem worth the cost [23]. Some recent works have also revealed this [24]. Yet, in another case, certain ranges of high latency would simply cause too severe penalties, and hence should certainly be avoided; but since this is not reflected in Equation 1, the tuning might consider any latency improvement as beneficial. As a result, practically one often express a performance requirement with some expectations [18], [22], [25], e.g., "`the runtime shall ideally be 2 seconds`", which could imply that there is certain tolerance for runtime higher than 2 seconds while anything lower than 2 seconds is fully satisfied, meaning a 2 second runtime is already the preferred optimality, even if a lower runtime is achievable with more tuning budget.

To incorporate the performance requirement, a natural way is to combine it with the tuning objective formulated as:

$$\underset{\boldsymbol{c}\in\mathcal{C}}{\arg\max}\ p(v) \mapsto 0 \leq p(v) \leq 1 \text{ where } v = f(\boldsymbol{c}) \tag{2}$$

whereby $\boldsymbol{c}$ denotes a configuration, v is the measured performance; we seek to maximize $p(v)$ in which p is a preference mapping function, rather than a constraint, that quantifies and converts the measured performance into a satisfaction score of range $[0, 1]$ (e.g., $p(v) = 0.7$ means 70% satisfied; $p(v) = 1$ and $p(v) = 0$ denote fully satisfied and fully unsatisfied, respectively). The mapping is specified and defined by the stakeholders according to their preferences (see Section III-A). Here, we are mainly interested in to what extent the configuration's performance satisfies the requirement.

Indeed, Chen and Li [18] have proved that, compared with optimizing Equation 1, optimizing Equation 2 is more beneficial if the expectation (e.g., runtime of 20 seconds) is reasonable. For example, a real world requirement is "`must be able to retrieve and display within 20 seconds`" from the *Criminal Tracking Network and Systems* in the PURE dataset [19], for which one interpretation could be that any higher than 20 seconds runtime are fully unsatisfied ($p(v) = 0$) while anything lower than 20 seconds receives a satisfaction score in $(0, 1]$ via a linearly increasing slope. In this case, tuning for the satisfaction score could enable a good number of configurations to be discriminative while a fair amount of the others are not (i.e., $p(v) = 0$), which better balances the exploitation and exploration in the tuning to enforce higher satisfaction.

C. Motivating Issues

Unfortunately, optimizing Equation 2 is not always effective. According to the findings of Chen and Li [18] and our own experiments, we discovered two major issues if the given performance requirement is directly used to guide the tuning:

- **Issue 1:** No pressure to push for convergence. For example, the performance requirement might be too strict, i.e., all configurations explored are fully unsatisfied, hence we do not know what to preserve, leading to weak pressure that causes slow convergence. This makes sense as the developers do not always understand whether a performance expectation is feasible.
- **Issue 2:** Stagnation with premature convergence. The performance requirement might too strongly discriminate the configurations, exacerbating the tuning to be trapped at local optima—the sub-optimal satisfaction never change. Indeed, how exactly the performance requirement impacts the tuning is unknown at requirement elicitation.

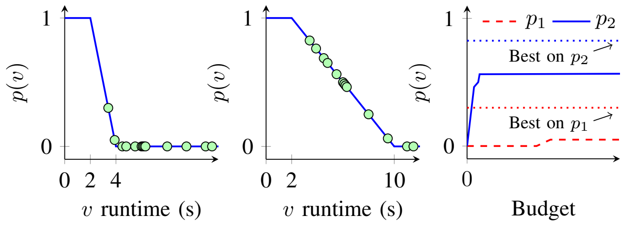
Figure 3a shows an example for **Issue 1**, in which the tuning would most likely perform exploration in the landscape with limited guidance, especially under a diversity preservation strategy [26], since it very likely that all configurations that a tuner explored are indistinguishable on satisfying the performance requirement p_1 (i.e., $p(v) = 0$). In contrast, Figure 3b is a likely case that causes **Issue 2** where the tuning mostly exploits the performance requirement p_2 for guidance since many configurations can be discriminated, but such guidance might be highly strong during the tuning, causing it to quickly converge (prematurely) but easy to stuck at local optima—a typical case of over-emphasizing on exploitation [27]. It is well known that biasing to either exploration or exploitation is devastating to configuration tuning [14], hence the requirement satisfaction of both is struggling to improve, as in Figure 3c.

Our motivation is intuitive: if we know what is required by the stakeholders, then using such information to guide the tuning should lead to better results quicker (which is indeed possible [18]). However, due to **Issues 1-2**, the above is not always true. The key question is *how do we exploit a performance requirement to guide configuration tuning while being able to adapt to its ineffective and/or misleading aspects?* CoTune is designed precisely to mitigate those.

III. Tuning via Co-evolving Performance Requirement and Configurations

CoTune exploits the guidance provided by the given target performance requirement, represented as a proposition[1], while catering for the chance that it can be ineffective or

[1]Proposition denotes the quantification of performance requirement.



(a) p_1 (no pressure) (b) p_2 (stagnation) (c) Tuning trajectory

Fig. 3: The issues when tuning is directly guided by two alternative performance requirements and their quantification p_1 and p_2 (for the same system). p_1 in (a) indicates that most configurations cannot be discriminated by the satisfactions (expectation too strict: we probably can only find $p(v) = 0$), resulting in limited pressure to steer the tuning toward optimality in (c) when using p_1 to guide the tuning. p_2 in (b) shows that nearly all configurations can be discriminated by the satisfactions (expectation too relaxed), but this can easily trap the tuning in local optima, causing stagnation that is far from the optimality when using p_2 as the objective in (c).

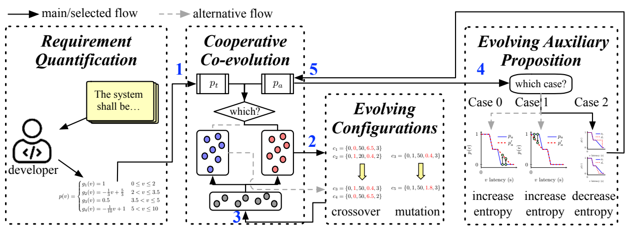


Fig. 4: Architecture overview of CoTune.

even misleading. Drawing on the co-evolution theory [21], CoTune creates a new ***auxiliary proposition*** (p_a) alongside the ***target proposition*** (p_t). p_t would be fixed to serve as the "golden rule" but p_a is co-evolved together with the configurations while allowing both the p_a and p_t to guide and evolve the configurations (addressing **Issues 1-2**). We are not interested in satisfying the auxiliary proposition but merely leverage it to assist the satisfaction of the target one. CoTune has several phases as in Figure 4 and Algorithm 1:

- **Requirements Quantification:** Developers firstly quantify the performance requirement that needs to be satisfied under the framework in CoTune (Section III-A).
- **Co-evolution with Cooperation:** This determines which proposition to guide the evolution of configurations, and how the newly explored configurations are preserved (Section III-B; lines 8-16 and 27-36).
- **Configuration Evolution:** Configurations are evolved based on the guidance from the selected proposition (Section III-C; lines 10 and 16).
- **Requirement Evolution:** The auxiliary proposition is evolved depending on whether convergence is too weak or suffering stagnation (Section III-D; lines 17-26).

CoTune terminates when either a configuration of $p_t(v) = 1$ has been found or the budget has been exhausted.

Algorithm 1: Pseudo code for `CoTune`

```
1  Input: Budget $B$; population size $n$; stagnation cap $k$; target proposition $p_t$
2  Declare: Auxiliary proposition $p_a$; new auxiliary proposition $p'_a$; temporary
       auxiliary proposition $p''_a$; set of auxiliary propositions $\mathcal{G}$; $\mathcal{P}_t$ and $\mathcal{P}_a$ are
       configuration populations for $p_t$ and $p_a$, respectively; the consumed budget
       $b$; newly evolved configurations set $\mathcal{O}$; configuration $\boldsymbol{c}$ and $\boldsymbol{c'}$; stagnation
       counter $i$; the best configuration on $p_t$ from the last iteration $\boldsymbol{c}_{pre-best}$
   Output: The best configuration $\boldsymbol{c}_{best}$ on $p_t$
3  $p_a = p_t$
4  $\mathcal{P}_t, \mathcal{P}_a \leftarrow$ randomly generate $n$ configurations
5  Measure $n$ configurations on the system and fitness via $p_t$ and $p_a$
6  while $b + n < B$ do
7      $\theta \leftarrow$ compute via Equation 4
8      /* Evolve new configurations via standard mating
          selection, mutation, and crossover at each
          iteration in Genetic Algorithm via $p_t$ or $p_a$.   */
9      if random $\alpha \in [0, 1] < \theta$ or $p_a$ was updated last iteration then
10         $\mathcal{O} \leftarrow$ generate $n$ configurations from $\mathcal{P}_a$ using $p_a$
11     else
12         $\mathcal{O} \leftarrow$ generate $n$ configurations from $\mathcal{P}_t$ using $p_t$
13     /* Preserve promising configurations.   */
14     Measure and evaluate configurations in $\mathcal{O}$ under both $p_t$ and $p_a$
15     $\mathcal{P}_t \leftarrow$ top $n$ configurations from $\mathcal{P}_t \cup \mathcal{O}$ on $p_t$
16     $\mathcal{P}_a \leftarrow$ top $n$ configurations from $\mathcal{P}_a \cup \mathcal{O}$ on $p_a$
17     $b = b + n$
18     /* Evolve $p_a$ when Case 0 is detected.   */
19     if $\forall p_a(f(\boldsymbol{c})) = 0$ and $\forall p_t(f(\boldsymbol{c'})) = 0, \boldsymbol{c} \in \mathcal{P}_a; \boldsymbol{c'} \in \mathcal{P}_t$ then
20         $p_a \leftarrow$ change $p_a$ by relaxing a boundary point that increases
             entropy $h$ via Equation 5 and 6
21     /* Evolve $p_a$ when Case 1 is detected.   */
22     else if $\forall p_a(f(\boldsymbol{c})) = 1, \boldsymbol{c} \in \mathcal{P}_a$ then
23         $p_a \leftarrow$ change $p_a$ by tightening a boundary point that increases
             entropy $h$ via Equation 5 and 7
24     /* Evolve $p_a$ when Case 2 is detected.   */
25     else if $i \geq k$ then
26         while $\nexists h(f(\boldsymbol{c}), p'_a) < h(f(\boldsymbol{c}), p_a), p'_a \in \mathcal{G}$ or $|\mathcal{G}| < n$ do
27             $\mathcal{G} \leftarrow \mathcal{G} \cup$ generate a new proposition from $p_a$ by changing
                 fragments and/or boundary points
28             if $|\mathcal{G}| > n$ then
29                 $\mathcal{G} \leftarrow \mathcal{G} - p''_a \in \mathcal{G}$ of the highest $h$ via Equation 5
30         $p_a \leftarrow p''_a \in \mathcal{G}$ of the lowest $h$ via Equation 5
31     if $p_a$ has changed then
32         Reevaluate the fitness of configurations in $\mathcal{P}_a$ on $p_a$
33     $\boldsymbol{c}_{best} \leftarrow$ the best configuration on $p_t$ from $\mathcal{P}_t \cup \mathcal{P}_a$
34     if $\boldsymbol{c}_{best}$ is better than $\boldsymbol{c}_{pre-best}$ on $p_t$ then
35         Set $\boldsymbol{c}_{pre-best} = \boldsymbol{c}_{best}$ and $i = 0$
36         if $p_t(f(\boldsymbol{c}_{best})) = 1$ then return $\boldsymbol{c}_{best}$
37     else
38         $i = i + 1$
39 return $\boldsymbol{c}_{best}$
```

A. Quantifying Performance Requirements

To quantify performance requirements for configuration tuning, we analyze the patterns from the real-world requirements datasets: PROMISE [28], PURE [19], [29], and SRS [30], [31].

1) Fragment: In `CoTune`, we formally specify performance requirements with fragments as these points essentially represent distinct preferences based on the intervals of metric values. Given an interval $[v_i, v_{i+1}]$ over the value v of a performance metric (e.g., $[1.5s, 3s]$ for latency), the fragment and its implied preference of the performance requirement, denoted ψ, can be represented via Backus-Naur notations [32]:

$$\langle\psi\rangle ::= \mathbf{G} \mid \mathbf{S} \mid \mathbf{E}$$
$$\langle\mathbf{G}\rangle ::= \forall v \in [v_i, v_{i+1}], \textit{ a greater } v \textit{ is preferred at } [s_i, s_{i+1}]$$
$$\langle\mathbf{S}\rangle ::= \forall v \in [v_i, v_{i+1}], \textit{ a smaller } v \textit{ is preferred at } [s_i, s_{i+1}]$$
$$\langle\mathbf{E}\rangle ::= \forall v \in [v_i, v_{i+1}] \textit{ is equally preferred at } s_i$$

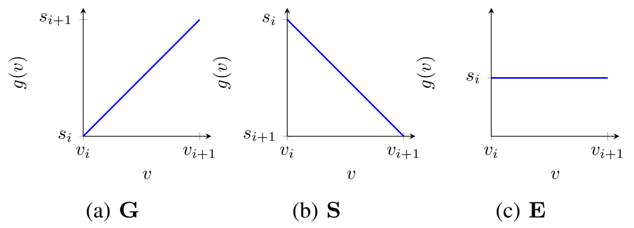


Fig. 5: Fragment types quantified by fuzzy functions $g(v)$ as satisfaction scores. (a): $g(v) = (\frac{s_i - s_{i+1}}{v_i - v_{i+1}})v + \frac{s_{i+1}v_i - s_i v_{i+1}}{v_i - v_{i+1}}$; (b): $g(v) = (\frac{s_i - s_{i+1}}{v_i - v_{i+1}})v + \frac{s_{i+1}v_i - s_i v_{i+1}}{v_i - v_{i+1}}$; (c): $g(v) = s_i$.

where s_i denotes the satisfaction score for that interval ($s_i \in [0, 1]$), which is adapted depending on the preference of the adjacent intervals in a performance requirement. The first two are ***distinguishable fragments*** while the last is an ***indistinguishable fragment***. Clearly, a score of 1 and 0 represent fully satisfied and fully non-satisfied requirement, respectively. Note that while some bounds for a performance metric are clear, e.g., the lower bound for runtime and throughput can only be 0, in other cases, the bounds may need to be set based on the domain knowledge of developers, e.g., the maximum runtime.

The satisfaction score of the above fragments can be quantified by a function $g(\cdot)$ using fuzzy logic [20] (e.g., we quantify the extent to which a requirement can be partially satisfied). As shown in Figures 5a and 5b, since we do not know to what extent a greater (or smaller) v is sufficient, the membership function can be linearly specified as a slop that monotonically increases (or decreases) the satisfaction score gradually from v_i to v_{i+1}. Both fragments reach the best and worst satisfaction at the interval bounds[2]. The fragment "$\forall v \in \theta$ is equally preferred at s_i" is a special case of the fuzzy logic, such that all values of the performance metric between v_i and v_{i+1} are equally satisfied at s_i (Figure 5c).

2) Proposition: In theory, the fragments can be arbitrarily combined, in any number or order, to form a complex yet quantifiable ***proposition*** for the performance requirement. Here, any two fragments are connected by a ***boundary point***. In `CoTune`, using the Backus-Naur notations, a proposition p of n fragments (with $n - 1$ boundary points) is:

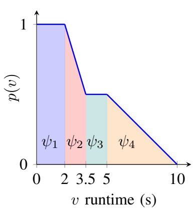
$$\langle p\rangle ::= \psi \mid \psi \ \& \ p$$

in which there are n intervals, i.e., $\{[v_1, v_2], [v_2, v_3]$, ..., $[v_n, v_{n+1}]\}$, and a vector of satisfaction scores, e.g., $\overline{s} = \{[s_1, s_2], s_2, ..., [s_{n-1}, s_n]\}$. Clearly, p is quantified as a piecewise function of several functions g for the fragments.

In this work, we assume that the above proposition of performance requirements can be elicited and quantified by the developers via formal analysis tools as in prior work [33].

3) Example: Consider the performance requirement: "`The system should respond in 5 seconds and ideally less than 2 seconds`". One might interpret that with four fragments and their piecewise function

[2]For maximizing metric, $s_i \leq s_{i+1}$; otherwise, $s_i \geq s_{i+1}$.



$\psi_1 = \mathbf{E} ::= \forall v \in [0, 2]$ is equally preferred at 1.
$\psi_2 = \mathbf{S} ::= \forall v \in [2, 5]$, a smaller v is preferred at $[1, 0.5]$.
$\psi_3 = \mathbf{E} ::= \forall v \in [2, 5]$ is equally preferred at 0.5.
$\psi_4 = \mathbf{S} ::= \forall v \in [5, 10]$, a smaller v is preferred at $[0.5, 10]$.

$$p(v) = \begin{cases} g_1(v) = 1 & 0 \leq v \leq 2 \\ g_2(v) = -\frac{1}{3}v + \frac{5}{3} & 2 < v \leq 3.5 \\ g_3(v) = 0.5 & 3.5 < v \leq 5 \\ g_4(v) = -\frac{1}{10}v + 1 & 5 < v \leq 10 \end{cases} \tag{3}$$

Fig. 6: Exampled function of a proposition $p(v)$; $v = f(\boldsymbol{c})$.

as shown in Figure 6 (suppose that we know the runtime ≤ 10 seconds, e.g., due to the timeout setting[3]). Here, since there is a conflict on the intervals of v for ψ_2 and ψ_3, we set each of them to share half as $[2, 3.5]$ and $[3.5, 5]$, respectively.

Our formalization provides a way for developers to reason about and quantify the performance requirements, which can be readily co-evolved with the configurations.

B. Cooperative Co-evolution

`CoTune` operates on a cooperative co-evolution [21] between performance requirements and configurations under heterogeneous representations. To that end, we use the auxiliary proposition p_a to assist the target proposition p_t, for which we have $p_t = p_a$ initially. During the tuning, p_t is fixed and it is what the tuner needs to satisfy; p_a, which will be evolved from p_t, guides the tuning when p_t is ineffective or even misleading. These two propositions dynamically and alternatively steer the evolution of the configurations, which, in turn, also impacts how the auxiliary proposition would evolve.

1) When to Evolve?: Configurations would be evolved at each iteration, which is a standard procedure. For the auxiliary proposition, its evolution might be triggered at an iteration under three cases (with decreasing commonality):

- The convergence is too weak (too much exploration) due to all explored configurations on the target and auxiliary propositions are fully unsatisfied—over strict proposition.
- The convergence is too weak (too much exploration) due to all explored configurations on the auxiliary proposition being fully satisfied—the proposition is too relaxed.
- Suffering stagnation (too much exploitation) where the best configuration on the target proposition has not changed over some iterations.

2) How to Evolve?: `CoTune` evolves configurations following standard Genetic Algorithm [7], [11], [14], i.e., by mating selection, mutation and crossover, to update the configurations according to the target proposition or the auxiliary one if the target becomes misleading. The promising configurations according to the target and auxiliary propositions are preserved separately. The auxiliary proposition is evolved by changing the boundary point and/or switching the fragments ($\mathbf{G}$, $\mathbf{S}$, and $\mathbf{E}$), i.e., replacing the proposition with another similar yet different quantification. The direction and extent of change depend on the aforementioned cases, which are determined by the configurations found and their overall discriminative power, as we will elaborate in Section III-D. As such, the evolution of the configurations and auxiliary propositions cooperatively influences each other, forming a co-evolution.

3) What Benefits do Co-evolution Bring?: This cooperative co-evolution ensures that the evolution of configurations and propositions can positively influence each other. The target proposition still serves a primary role in the tuning when it is effective, and we can leverage the evolving auxiliary proposition to eliminate its negative impact. As such, the benefits of such a co-evolution in `CoTune` are two-fold:

- The useful information embedded in the target proposition can still be leveraged to boost the tuning;
- while misleading cases (**Issues 1-2**) are mitigated by the auxiliary proposition co-evolved with the configurations.

4) Co-evolution Procedure: As can be seen from Figure 4, we design two populations of configurations, $\mathcal{P}_t$ and $\mathcal{P}_a$, for evolving under p_t and p_a, respectively. These two populations are important to keep track of the evolution via each individual proposition, detecting when the target proposition is misleading while co-evolving the auxiliary one with new configurations generated by the more important proposition. At each iteration, the co-evolution cooperates as follows:

(a) Compute the θ, which reflects the probability for selecting p_a to guide the tuning, as below ($\theta = 1$ if $w_a = w_t = 0$):

$$\theta = \frac{w_a}{w_a + w_t} \quad \text{s.t.} \quad \begin{aligned} w_a &= f_{\mathcal{P}_a \to p_t} + l_{\mathcal{P}_a \to p_a} \\ w_t &= f_{\mathcal{P}_t \to p_t} + l_{\mathcal{P}_t \to p_t} \end{aligned} \tag{4}$$

whereby $f_{\mathcal{P}_a \to p_t}$ and $f_{\mathcal{P}_t \to p_t}$ respectively denote the average fitness[4] of $\mathcal{P}_a$ and $\mathcal{P}_t$ evaluated by p_t. This assesses how well the two propositions help to guide the tuning towards satisfying the target proposition. $l_{\mathcal{P}_a \to p_a}$ and $l_{\mathcal{P}_t \to p_t}$ respectively denote the change of the best configuration in $\mathcal{P}_a$ and $\mathcal{P}_t$ with respect to the fitness on p_a and p_t. This tracks the tuning progress achieved by the independent guidance from the two propositions. Thus, the more effective proposition that can encourage good progress would be used to guide the tuning.

(b) To guide the mating selection, new configurations will be evolved based on the fitness evaluated by p_a if a random number $< \theta$ or if the last iteration has evolved p_a; p_t would be selected otherwise (Section III-C).

(c) After adding the new configurations to both populations, we preserve the top n configurations (n is the population size) from each population using their corresponding proposition. This helps to track the evolution affected by each proposition via their corresponding population.

(d) p_a will be evolved according to different situations of the configurations evolved, see Section III-D.

This cooperative co-evolution ensures that the evolution of configurations and propositions can positively influence each other. The p_t still serves as a primary role for the tuning while the evolving p_a eliminates its misleading impacts.

[3]The worst performance metric can often be set based on experience.

[4]From here, we use fitness and satisfaction score interchangeably.

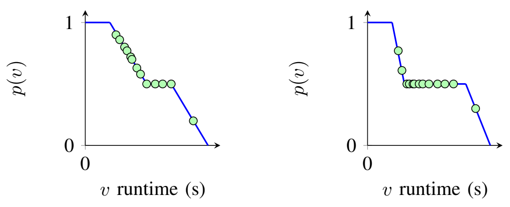
$$\begin{aligned} \boldsymbol{c}_1 &= \{0, 0, 50, 6.5, 3\} \\ \boldsymbol{c}_2 &= \{0, 1, 20, 0.4, 2\} \end{aligned} \qquad \boldsymbol{c}_3 = \{0, 1, 50, 0.4, 3\}$$

$$\Downarrow \qquad\qquad\qquad \Downarrow$$

$$\begin{aligned} \boldsymbol{c}_3 &= \{0, 1, 50, 0.4, 3\} \\ \boldsymbol{c}_4 &= \{0, 0, 50, 6.5, 2\} \end{aligned} \qquad \boldsymbol{c}_5 = \{0, 1, 50, 1.8, 3\}$$

(a) Crossover (b) Mutation

Fig. 7: Evolving configurations in `CoTune`.



(a) High entropy (p_1) (b) Low entropy (p_2)

Fig. 8: A proposition p_1 with high entropy in (a) and another p_2 with low entropy in (b) for the same set of configurations.

C. Evolving Configurations

The configurations are evolved following the standard procedure in using genetic algorithm for configuration tuning [11], [13]–[15]. As from Figure 7, a configuration is represented as a vector (see Equation 1). The reproduction starts by using a binary tournament for mating selection, i.e., randomly choosing two configurations and the one with a better fitness value, which is evaluated by either the target or auxiliary proposition, would be selected (or anyone if they have equal fitness). Two configurations are selected to generate offspring new configurations via the standard random mutation and uniform crossover [11], [14]. The top n configurations from both the current population and newly generated ones are preserved according to the fitness of the corresponding proposition.

D. Evolving Performance Requirements

To evolve the auxiliary proposition p_a, it is represented as $\{x_1, x_2, ..., x_m\}$ where x_m denotes either a fragment or adjustable boundary point (exclude the minimum v_{min} and maximum v_{max} performance value), interleaving each other from left to right. Taking the example in Figure 6, it would become $\{\mathbf{E}, 2, \mathbf{S}, 3.5, \mathbf{E}, 5, \mathbf{S}\}$. The fragment can be switched (e.g., from $\mathbf{S}$ to $\mathbf{E}$) while the boundary value can be changed within the bound of its pieced function; the satisfaction score follows from the left-sided boundary point. p_a would be evolved differently at each iteration depending on three alternative situations to be mitigated, aiming to better assist the target proposition p_t in guiding the tuning, especially when it tends to be useless or misleading:

- **Case 0:** $\forall p_a(f(\boldsymbol{c})) = 0$ and $\forall p_t(f(\boldsymbol{c'})) = 0, \boldsymbol{c} \in \mathcal{P}_a; \boldsymbol{c'} \in \mathcal{P}_t$. This means that both the target and auxiliary propositions (p_t and p_a) cannot provide sufficient pressure for the tuning to converge, because they are too strict to guide the tuning, making all configurations fully unsatisfied.
- **Case 1:** $\forall p_a(f(\boldsymbol{c})) = 1, \boldsymbol{c} \in \mathcal{P}_a$. This reflects that the auxiliary proposition p_a, which is initially set as $p_a = p_t$ and hence evolved from p_t, has become too relaxed and will not provide much convergence pressure if selected, making all configurations fully satisfied.
- **Case 2:** The best $\boldsymbol{c}$ on p_t from both $\mathcal{P}_t$ and $\mathcal{P}_a$ has not changed in k iterations (we use k=3 which is the reasonable setting). This implies that although p_t and/or p_a might still offer good search pressure for the tuning, the pressure could be too strong, causing the tuning traps at local optima—a typical stagnation [26], [27].

The key cause of the above cases is related to the ability to discriminate the configurations through the (auxiliary) proposition, which is a key to comparing and evolving the better or worse configuration. In `CoTune`, we leverage differential entropy to quantify such a discriminative power achieved by a proposition p over the current configurations $\boldsymbol{c}$ in $\mathcal{P}_a$, whose (continuous) performance value $v = f(\boldsymbol{c})$ is converted into the continuous satisfaction score $p(v) \in [0, 1]$:

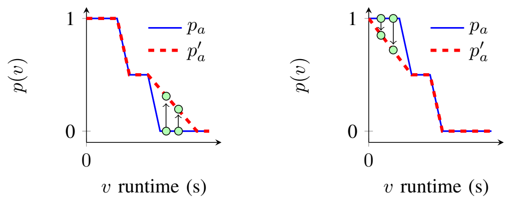
$$h(v, p) = -\int_{\boldsymbol{c} \in \mathcal{P}_a} \beta(p(v)) \log \beta(p(v)) dv \tag{5}$$

where β is the probability density function of the satisfactions from p for $\boldsymbol{c}$ via Kernel Density Estimation (KDE) [34].

As we can see from Figure 8, in general, differential entropy measures the "spread", or uncertainty, of the satisfactions over certain configurations evaluated by a proposition. When h is high (Figure 8a), most part of the satisfaction distribution would have more deviated/fluctuated values (hence uncertain), making it easier to distinguish good from poor configurations, which strengthens the ability of discrimination. Such strong guidance might exacerbate the chance of trapping at local optima in **Issue 2**. Conversely, when h is low (Figure 8b), configurations in the distribution tend to have the same, or highly similar, satisfaction scores, hence it is harder to state which configurations are clearly better/worse, weakening the discriminative power. This might cause **Issue 1** in the tuning. In what follows, we will delineate how we leverage differential entropy to steer the evolution of auxiliary proposition.

1) Encouraging Convergence: The issue behind **Case 0** and **Case 1** is that they have no discrimination ability on the configurations, hence losing the pressure for convergence. Thus, the goal of evolving the p_a in `CoTune` is to strengthen the discrimination by increasing the differential entropy of the current auxiliary proposition, encouraging exploitation.

Suppose that the performance objective has been converted to minimization, to evolve a new auxiliary proposition p'_a under **Case 0**, `CoTune` finds the first fragment from the right that is distinguishable and move its right boundary point rightward by Δ times, where Δ is randomly chosen from $[0.5, 1]$: $b_{right} = \min\{b_{right} + b_{right} \times \Delta, v_{max}\}$ such that v_{min} is the maximum value of the performance metric, if any. The process stops when the current set of configurations $\boldsymbol{c}$ has $h(v = f(\boldsymbol{c}), p'_a) > h(v = f(\boldsymbol{c}), p_a)$: most previously fully



(a) Case 0, p_a to p'_a (b) Case 1, p_a to p'_a

Fig. 9: Evolving the auxiliary proposition for Case 0 and 1.

unsatisfied configurations might become partially satisfied with varying satisfactions. p_a and p'_a will have:

$$\int_{v_{min}}^{v_{max}} p_a(v)\,dv < \int_{v_{min}}^{v_{max}} p'_a(v)\,dv \Rightarrow p_a \sqsubseteq p'_a \tag{6}$$

From the example in Figure 9a, this means that the area of p'_a will be larger than that of p_a, i.e., we have relaxed the auxiliary proposition of the performance requirement.

Similarly, to evolve a new auxiliary proposition p'_a under **Case 1**, `CoTune` finds the first fragment from the left that is distinguishable and move its left boundary point b_{left} leftward by Δ times, where Δ is randomly chosen from $[0.5, 1]$: $b_{left} = \max\{b_{left} - b_{left} \times \Delta, v_{min}\}$ such that v_{min} is the minimum value of the performance metric, if any. Again, the process stops when the current set of configurations $\boldsymbol{c}$ has $h(v = f(\boldsymbol{c}), p'_a) > h(v = f(\boldsymbol{c}), p_a)$: most previously fully satisfied configurations might become partially satisfied with varying satisfactions. In this way, p_a and p'_a will meet:

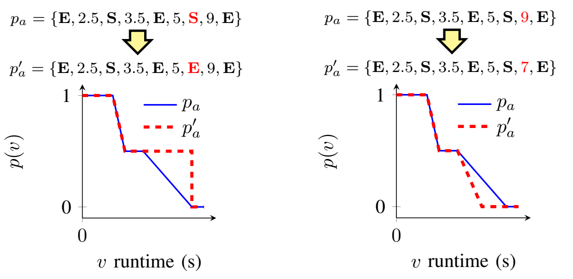
$$\int_{v_{min}}^{v_{max}} p_a(v)\,dv > \int_{v_{min}}^{v_{max}} p'_a(v)\,dv \Rightarrow p_a \sqsupseteq p'_a \tag{7}$$

From Figure 9b, in this case, it means that the area of p'_a will be smaller than that of p_a, i.e., we have tightened the auxiliary proposition of the performance requirement.

2) Overcoming Stagnation and Local Optima: **Case 2** differs from the other two in the sense that the propositions can have strong pressure for convergence by discriminating the configurations, but the tuning might be struggling to escape from some local optima [26], [27]. To mitigate that, `CoTune` seeks to evolve a new auxiliary proposition p_a that exhibits much lower differential entropy, as it is known that weakening the ability to discriminate the configurations is the key to overcome local optima [14] by encouraging exploration.

Since there is no clear direction to evolve p_a, we follow a random search strategy to mutate p_a in the space of possible propositions, guided by differential entropy using all configurations in population $\mathcal{P}_a$ (lines 21-26 in Algorithm 1):

(a) Mutate a new auxiliary proposition p'_a from p_a by randomly changing the fragments and/or boundary points.
(b) Get the entropy h for the generated propositions against all configurations from $\mathcal{P}_a$ by KDE and Equation 5.
(c) Repeat from (a) until the required number of propositions has been met and at least one new proposition has lower entropy h than the current auxiliary proposition p_a.
(d) The proposition with the lowest h is the new p_a.



(a) Switching fragment (b) Moving boundary point

Fig. 10: Evolving auxiliary proposition for Case 2.

TABLE I: Real-world configurable systems studied. $|\mathcal{B}|$ and $|\mathcal{N}|$ denote the number of binary and numeric/enumerate options, respectively; $\mathcal{S}_{space}$ denotes the configuration space.

System	Benchmark	Domain	Lang.	Perf. Metric	$\lvert\mathcal{B}\rvert/\lvert\mathcal{N}\rvert$	$\mathcal{S}_{space}$	Ref.
7Z [35]	A 3 GB directory	File Compressor	C++	Runtime (ms)	11/3	4.39×10^{5}	[36]
KANZI [37]	Default benchmark	File Compressor	Java	Runtime (ms)	31/0	5.36×10^{8}	[2]
EXASTENCILS [38]	Default benchmark	Code Generator	Scala	Runtime (ms)	7/5	6.55×10^{5}	[36]
APACHE [39]	Web server benchmark	Web Server	C	Throughput (req/s)	14/2	3.28×10^{4}	[2]
SQLITE [40]	Default benchmark	Database Engine	Various	Latency (ms)	39/0	5.50×10^{11}	[2]
DCONVERT [41]	Default images	Image Scaling	Java	Runtime (s)	17/1	2.62×10^{5}	[42]
DEEPARCH [43]	UCR Archive dataset	AI Tool	Python	Runtime (min)	12/0	4.10×10^{3}	[44]
JUMP3R [45]	Jump3r codebase	Static Analysis	Java	Runtime (ms)	37/0	6.87×10^{10}	[2]
HSMGP [46]	V-cycle solver bench	Multigrid Solver	C++	Runtime (ms)	11/3	1.00×10^{5}	[2]

As in Figure 10, the mutations ensure that we do not only change the boundary points (Figure 10b), but can also switch the fragments (Figure 10a)—the proposition can either be overall relaxed or tightened, as long as the differential entropy over the current configurations is minimized, weakening the discriminative power of tuning to jump out of local optima.

IV. Experiment Setup

To assess `CoTune`, we study four research questions (RQs):

- **RQ1:** How can the co-evolution help in `CoTune`?
- **RQ2:** How does `CoTune` perform against the others?
- **RQ3:** What is the contribution of the key component to `CoTune`'s performance?
- **RQ4:** What is the sensitivity of `CoTune` to k?

All experiments were implemented in Python and performed on a server with Intel(R) CPU (224 cores) and 500GB RAM.

A. Subject Configurable Systems

To ensure a fair evaluation, we use systems and their performance data measured under real-world benchmarks. We prioritize systems that have been widely used in the literature:

- For multiple versions of a system, we use the one with more options to ensure relevance and complexity.
- We exclude systems that lack consistent benchmarking protocols or have been used by < 2 study for reliability.

Table I shows all the systems studied, covering a wide range of domains/types, scales, metrics, and programming languages. For simplicity, we convert all maximizing performance metrics into minimization via additive inversion.

B. Compared Tuners

We consider several state-of-the-art tuners to fulfil the RQs, including the widely used Genetic Algorithm (`GA`) based tuner [11], [13]–[15] (i.e., `CoTune` without co-evolving the performance requirement) and those rely on the model-based Bayesian Optimization, i.e., `HEBO` [47], `Flash` [3], and `SMAC` [17], together with the most recent ones: `TurBO` [48] and `Bounce` [49]. We omitted those that have been discarded, e.g., `BaxUS` [50] has been upgraded to `Bounce`.

We compare two variants of each compared tuner X (including all variants of Bayesian Optimization):

- X_r: The origin that tunes without knowing the performance requirements, i.e., optimizing Equation 1.
- X_p: The tuning is guided by the requirement—the satisfaction scores of configurations' actual/predicted performance (via p_t)—instead of the raw performance in acquisition/selection, i.e., optimizing Equation 2.

Their evaluations are the same: through the given target proposition p_t, we compare the satisfaction score of the performance of the returned configuration from the tuner.

Note that we do not consider multi-fidelity tuners [51]–[53], because they leverage a well-defined fidelity for AutoML, i.e., using more training data will have higher-fidelity accuracy but is more costly, which is unclear for configurable systems, e.g., for an image rescaling system, it is unclear how the images can be changed to influence the system performance/cost.

C. Requirement, Budget, and Parameter Settings

Since we found no requirement datasets that are specifically related to the studied systems under tuning, in this work, we synthetically generate requirements in the evaluation according to the systems. However, the generated ones are not arbitrary, but rather they are instantiated based on patterns extracted from real-world requirements datasets with 100+ performance requirements [28]–[30], including PROMISE [28], PURE [19], [29], and SRS [30], [31], and tailor them to be system specific:

- We generate each requirement with five fragments, which is the most complex case identified from those datasets.
- For each requirement, we randomly and proportionally assign the fragment (**G**, **S**, and **E**) according to the datasets, while ensuring monotonic satisfaction, e.g., a lower latency would never be less preferred than a higher value; they are at most equally preferred/non-preferred.
- Set the expectation/boundary point according to the empirical ranges for the corresponding systems/metrics.

The above is also a similar strategy adopted by Chen and Li [18]. For each system, we generate the target proposition p_t with the procedure below:

(a) Randomly generate a p_t with five fragments[5].
(b) Randomly and finely replace the fragments and/or change the boundary points such that $d\%$ of the configurations fully or partially satisfy p_t, where $d \in \{0.1\%, 1\%, 5\%, 20\%, 50\%, 90\%\}$. A lower $d\%$ implies a more strict requirement, which is harder to satisfy.
(c) Repeat (b) until all % caps are covered.
(d) Repeat from (a) for three times.

In total, there are three types of requirements ($p_{t,1}$, $p_{t,2}$, and $p_{t,3}$) $\times$ six levels of satisfiability $=$ 18 performance requirements per system. As such, different systems would be given target performance requirements of diverse difficulties, shapes, and implications to the tuning if used therein.

We use the number of system measurements as tuning budget which is robust to the noises of the implementation and hardware [2], [3], [14]. We set a maximum budget of 300 since it is a most common setting from the literature[6] [2], [14]. For `CoTune` and `GA`, we use 30 generations and a population size of 10, which is again the default setting for configuration tuning [14]. For mitigating stagnation in `CoTune`, we set $k = 3$ and this will be evaluated in **RQ4**. The mutation and crossover rate for configuration is 0.1 and 0.9, respectively, as with existing work [14]. For model-based tuners, the initial sample size is 10 and all other settings are left as default [3].

All experiments run 30 repeats with different random seeds.

D. Metrics and Statistical Test

The metric to be evaluated is simply the satisfaction score of the given target performance requirement p_t.

To compare multiple tuners over 30 runs, we adopt the non-parametric Scott-Knott ESD [54] as the statistical test, which ranks tuners based on their average satisfaction and partitions them into statistically distinct groups by maximizing inter-group variance, e.g., if three tuners A, B, and C are evaluated, the ranking result may yield $\{A, B\}$ as 1 and $\{C\}$ as 2, thus A and B perform similarly, but both outperform C.

Compared with other multi-groups tests with post-hoc corrections [55], [56], Scott-Knott ESD overcomes the confounding factor of overlapping groups [54] with interpretable results.

V. EVALUATION

A. Importance of Co-evolution for Tuning

1) Method: For **RQ1**, we evaluate `CoTune` against the two variants of `GA` on all nine systems and 18 target performance requirements, leading to 162 cases tested by Scott-Knott ESD. Since the configuration evolution part in `CoTune` is essentially a `GA`, this helps to examine the necessity of co-evolution.

2) Results: As shown in Table II, we see that `CoTune` significantly outperforms the others in general, regardless of the systems and requirements. Within the 162 cases, `CoTune` has the best Scott-Knott ESD rank in 90% of them (50% sole best) against the 35% (3% sole best) and 35% (5% sole best) for `GA`$_p$ and `GA`$_r$, respectively. `CoTune` also has the best overall ranks and satisfaction with up to $1.78\times$ overall improvement. Interestingly, we see that `GA`$_p$ is generally better than `GA`$_r$ on easier requirements ($d \geq 5\%$), but worse on harder ones. This makes sense, as the requirement is useful when it is reasonable and hence `GA`$_p$ is advantaged. However, the requirement can most commonly mislead the tuning when it becomes too strict,

[5]In fact, with random generation we can have two fragments joined by a boundary point, where both of them have monotonically decreasing, increasing or consistent satisfactions. This reduces the number of fragments as the two adjacent fragments are quantified in the same way as if there are no boundary points in between, hence also emulating the cases of simpler propositions.

[6]For fair comparisons, we do not stop any tuner even if $p_t = 1$ has been achieved before all budget has been used.

TABLE II: Comparing `CoTune` with `GA` on a budget of 300 over 30 runs. Each cell (except the last row) reports the mean±standard deviation of the satisfaction (Scott-Knott ESD rank). A green cell denotes the best ranked tuner for a case.

d%	System	$p_{t,1}$			$p_{t,2}$			$p_{t,3}$		
		CoTune	GA$_p$	GA$_r$	CoTune	GA$_p$	GA$_r$	CoTune	GA$_p$	GA$_r$
0.1%	7z	.28±.34 (1)	.10±.25 (2)	.34±.37 (1)	.35±.39 (2)	.09±.27 (3)	.44±.43 (1)	.21±.33 (1)	.08±.26 (2)	.26±.35 (1)
	Kanzi	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.01±.05 (1)	.00±.00 (2)	.00±.00 (2)	.00±.00 (2)	.02±.10 (1)	.00±.00 (2)
	ExaStencils	.69±.46 (1)	.00±.00 (2)	.90±.30 (1)	.69±.46 (2)	.03±.18 (3)	.90±.30 (1)	.62±.49 (2)	.07±.25 (3)	.83±.38 (1)
	Apache	.00±.01 (1)	.00±.00 (2)	.00±.00 (2)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)
	SQLite	.00±.01 (2)	.00±.00 (2)	.02±.11 (1)	.03±.18 (1)	.03±.14 (1)	.04±.13 (1)	.01±.03 (1)	.00±.00 (2)	.01±.06 (1)
	DConvert	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.03±.18 (1)	.00±.00 (2)	.00±.00 (2)
	DeepArch	.66±.44 (1)	.24±.39 (2)	.59±.45 (1)	.73±.40 (1)	.15±.31 (3)	.62±.43 (2)	.72±.42 (1)	.26±.41 (3)	.58±.45 (2)
	Jump3r	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (1)	.00±.00 (2)	.00±.00 (2)	.00±.00 (1)	.00±.00 (2)	.00±.00 (2)
	HSMGP	.92±.23 (1)	.14±.34 (2)	.89±.30 (1)	.75±.40 (2)	.10±.30 (3)	.84±.32 (1)	.76±.42 (2)	.03±.18 (3)	.85±.33 (1)
1%	7z	.54±.39 (2)	.66±.34 (1)	.60±.37 (2)	.16±.18 (2)	.21±.18 (1)	.23±.18 (1)	.34±.31 (1)	.38±.30 (1)	.36±.28 (1)
	Kanzi	.10±.28 (1)	.06±.19 (2)	.03±.14 (2)	.05±.18 (2)	.12±.28 (1)	.09±.27 (1)	.06±.21 (1)	.02±.12 (2)	.05±.18 (1)
	ExaStencils	1.0±.00 (1)	.76±.41 (3)	.96±.15 (2)	.80±.26 (2)	.52±.44 (3)	.92±.07 (1)	.92±.07 (1)	.61±.43 (2)	.93±.06 (1)
	Apache	.03±.12 (1)	.00±.00 (2)	.00±.00 (2)	.01±.06 (1)	.00±.00 (2)	.00±.00 (2)	.01±.07 (1)	.00±.00 (2)	.00±.00 (2)
	SQLite	.20±.28 (1)	.10±.24 (2)	.19±.33 (1)	.19±.30 (1)	.19±.34 (1)	.12±.27 (2)	.11±.23 (1)	.04±.13 (2)	.05±.16 (2)
	DConvert	.41±.14 (1)	.19±.21 (3)	.31±.22 (2)	.25±.14 (1)	.14±.16 (3)	.18±.15 (2)	.24±.12 (1)	.07±.12 (2)	.24±.13 (1)
	DeepArch	.91±.21 (1)	.75±.31 (3)	.83±.20 (2)	.93±.13 (1)	.64±.39 (2)	.94±.12 (1)	.84±.26 (1)	.79±.36 (1)	.71±.31 (2)
	Jump3r	.08±.20 (1)	.09±.23 (1)	.00±.00 (2)	.08±.21 (1)	.02±.10 (2)	.03±.15 (2)	.04±.13 (1)	.02±.11 (1)	.00±.00 (2)
	HSMGP	.96±.18 (1)	.58±.49 (3)	.82±.38 (2)	1.0±.01 (1)	.48±.50 (3)	.89±.30 (2)	.93±.25 (1)	.69±.46 (3)	.82±.38 (2)
5%	7z	.72±.20 (1)	.66±.22 (2)	.55±.33 (3)	.62±.41 (2)	.78±.30 (1)	.71±.36 (1)	.59±.36 (1)	.66±.33 (1)	.61±.37 (1)
	Kanzi	.24±.32 (1)	.13±.28 (2)	.07±.21 (3)	.14±.25 (2)	.29±.36 (1)	.17±.31 (2)	.29±.32 (1)	.18±.27 (2)	.09±.21 (3)
	ExaStencils	.95±.14 (1)	.92±.23 (2)	.95±.13 (1)	.95±.07 (1)	.93±.11 (2)	.96±.05 (1)	.91±.12 (1)	.75±.31 (3)	.85±.22 (2)
	Apache	.13±.06 (1)	.12±.06 (1)	.12±.07 (1)	.34±.16 (1)	.34±.08 (1)	.28±.12 (2)	.22±.13 (1)	.21±.08 (1)	.18±.09 (2)
	SQLite	.21±.22 (1)	.17±.21 (1)	.11±.18 (2)	.42±.39 (1)	.36±.37 (1)	.20±.31 (2)	.23±.27 (1)	.23±.25 (1)	.21±.30 (1)
	DConvert	.75±.21 (1)	.72±.22 (1)	.71±.29 (1)	.77±.16 (1)	.78±.16 (1)	.79±.07 (1)	.48±.08 (1)	.38±.20 (3)	.44±.14 (2)
	DeepArch	.99±.01 (1)	.96±.18 (3)	.99±.02 (2)	.99±.01 (1)	.99±.01 (1)	.99±.01 (1)	.99±.01 (1)	.99±.01 (1)	.99±.01 (1)
	Jump3r	.15±.31 (1)	.12±.29 (1)	.08±.24 (2)	.18±.34 (1)	.09±.26 (2)	.04±.17 (3)	.08±.20 (1)	.00±.01 (2)	.08±.24 (1)
	HSMGP	1.0±.00 (1)	.93±.25 (2)	.90±.30 (2)	1.0±.00 (1)	.93±.25 (2)	.86±.34 (3)	.96±.18 (1)	.93±.25 (1)	.86±.34 (2)
20%	7z	.91±.16 (1)	.75±.33 (3)	.85±.27 (2)	.70±.15 (1)	.70±.13 (1)	.63±.18 (2)	.82±.21 (2)	.86±.15 (1)	.83±.17 (2)
	Kanzi	.72±.29 (1)	.59±.38 (2)	.32±.36 (3)	.64±.23 (1)	.50±.29 (2)	.27±.30 (3)	.37±.19 (1)	.37±.16 (1)	.26±.20 (2)
	ExaStencils	.98±.05 (1)	.99±.04 (1)	.99±.04 (1)	.99±.04 (1)	.98±.07 (2)	.98±.07 (2)	.96±.07 (2)	.95±.09 (2)	.97±.04 (1)
	Apache	.67±.08 (1)	.62±.09 (2)	.63±.10 (2)	.14±.02 (1)	.14±.02 (1)	.13±.02 (2)	.14±.07 (1)	.13±.05 (2)	.10±.04 (3)
	SQLite	.74±.26 (1)	.64±.28 (2)	.48±.34 (3)	.64±.19 (1)	.52±.24 (2)	.46±.26 (3)	.24±.16 (1)	.25±.19 (1)	.18±.22 (2)
	DConvert	.91±.03 (1)	.90±.05 (2)	.79±.27 (3)	.95±.02 (1)	.93±.08 (2)	.94±.03 (2)	.89±.05 (1)	.85±.15 (2)	.83±.20 (2)
	DeepArch	1.0±.00 (1)	1.0±.01 (2)	1.0±.00 (1)	1.0±.00 (1)	1.0±.01 (2)	1.0±.01 (2)	1.0±.00 (1)	1.0±.00 (1)	1.0±.01 (2)
	Jump3r	.27±.40 (1)	.04±.17 (2)	.04±.17 (2)	.15±.32 (2)	.25±.40 (1)	.02±.02 (3)	.10±.23 (1)	.03±.02 (2)	.04±.14 (2)
	HSMGP	1.0±.00 (1)	1.0±.00 (1)	.92±.23 (2)	.98±.13 (1)	.98±.13 (1)	.98±.13 (1)	.97±.14 (1)	.95±.19 (1)	.92±.23 (2)
50%	7z	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	.64±.28 (1)	.63±.29 (1)	.67±.27 (1)	.93±.18 (1)	.82±.28 (2)	.80±.34 (2)
	Kanzi	.64±.14 (1)	.43±.22 (2)	.24±.24 (3)	.64±.22 (1)	.46±.24 (2)	.42±.29 (2)	.50±.15 (1)	.32±.12 (2)	.22±.17 (3)
	ExaStencils	.99±.04 (1)	.98±.04 (1)	.96±.15 (2)	.88±.15 (1)	.63±.39 (3)	.74±.34 (2)	.99±.02 (2)	.98±.02 (3)	.99±.00 (1)
	Apache	.70±.03 (1)	.67±.04 (2)	.68±.04 (2)	.34±.17 (1)	.24±.08 (3)	.26±.09 (2)	.69±.07 (1)	.64±.10 (2)	.63±.08 (2)
	SQLite	.74±.16 (1)	.60±.22 (2)	.55±.23 (3)	.68±.19 (1)	.60±.15 (2)	.57±.15 (2)	.80±.21 (1)	.62±.25 (2)	.56±.25 (3)
	DConvert	.89±.04 (1)	.84±.17 (2)	.88±.11 (1)	.94±.02 (1)	.92±.11 (2)	.89±.15 (3)	.93±.03 (1)	.92±.03 (2)	.83±.23 (3)
	DeepArch	1.0±.00 (1)	1.0±.01 (2)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	1.0±.01 (2)
	Jump3r	.45±.33 (1)	.28±.22 (2)	.14±.17 (3)	.42±.25 (1)	.30±.15 (2)	.23±.11 (3)	.35±.24 (1)	.24±.11 (2)	.16±.10 (3)
	HSMGP	1.0±.00 (1)	.99±.07 (2)	.96±.13 (3)	1.0±.00 (1)	.97±.10 (3)	.99±.07 (2)	.98±.11 (1)	.96±.15 (1)	.96±.15 (1)
90%	7z	.77±.31 (1)	.74±.35 (1)	.73±.34 (1)	.81±.26 (1)	.83±.30 (1)	.77±.30 (1)	.94±.15 (1)	.87±.24 (2)	.69±.29 (3)
	Kanzi	.46±.19 (1)	.32±.20 (2)	.20±.18 (3)	.30±.22 (1)	.21±.19 (2)	.09±.16 (3)	.30±.28 (1)	.19±.23 (2)	.09±.11 (3)
	ExaStencils	1.0±.02 (1)	1.0±.02 (1)	.98±.04 (2)	.91±.14 (1)	.91±.15 (1)	.92±.12 (1)	.98±.04 (1)	.97±.06 (2)	.97±.06 (2)
	Apache	.99±.00 (1)	.99±.01 (2)	.99±.01 (2)	.80±.00 (1)	.80±.01 (2)	.80±.01 (2)	.98±.00 (1)	.98±.01 (2)	.98±.01 (3)
	SQLite	.63±.18 (1)	.51±.19 (2)	.46±.18 (3)	.68±.17 (1)	.57±.14 (2)	.53±.11 (3)	.78±.22 (1)	.50±.22 (3)	.60±.21 (2)
	DConvert	.88±.04 (1)	.88±.10 (2)	.87±.10 (2)	.94±.03 (1)	.91±.15 (2)	.89±.19 (2)	.88±.13 (1)	.86±.19 (1)	.89±.14 (1)
	DeepArch	1.0±.00 (1)	1.0±.00 (1)	1.0±.01 (2)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)	1.0±.00 (1)
	Jump3r	.32±.17 (1)	.25±.13 (2)	.20±.15 (3)	.27±.38 (1)	.05±.01 (2)	.04±.01 (3)	.09±.20 (1)	.05±.13 (2)	.02±.01 (3)
	HSMGP	1.0±.00 (1)	.88±.27 (2)	.93±.22 (2)	1.0±.00 (1)	.89±.27 (2)	.92±.24 (2)	.98±.13 (1)	.93±.21 (2)	.93±.22 (2)
Average p_t score/rank		.62/1.09	.52/1.87	.55/1.94	.57/1.20	.48/1.83	.53/1.85	.56/1.13	.47/1.81	.51/1.87

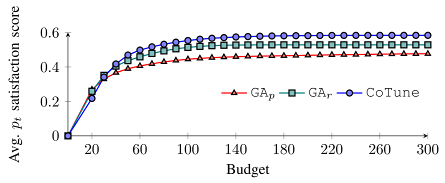


Fig. 11: Trajectories of `CoTune` and `GA` variants on different budgets over all cases/runs.

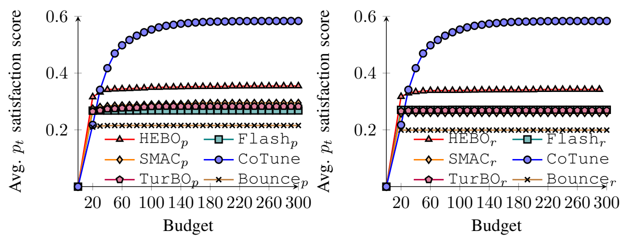
while GA$_r$ is not affected. `CoTune`, in contrast, performs the best in both situations, thanks to the co-evolution.

The same can also be observed in Figure 11, where the tuning trajectory of `CoTune` is generally better/steeper than the other two. This means that even if we have a different budget, `CoTune` would have still maintained its superiority.

RQ1: *`CoTune`, with its co-evolution, significantly outperforms the single-evolution `GA` with and without requirement, having the best rank on 90% of the cases (vs. 35%), boosting the overall satisfaction up to 1.78× with better efficiency.*

B. Improvements over State-of-the-art Tuners

1) Method: We compare `CoTune` with the other state-of-the-art tuners under two of their variants as part of **RQ2**. We follow the same procedure for **RQ1** except that we aggregate results for the three requirement types due to space constraint.



(a) `CoTune` vs. w/ requirement (b) `CoTune` vs. w/o requirement

Fig. 12: Trajectories of `CoTune` and state-of-the-art tuners (w/ and w/o requirement) on different budgets over all cases/runs.

2) Results: From Table III, we clearly observe the superiority of CoTune over the others with/without using requirement as the tuning objective: out of the 54, it has 89% best-ranked cases (87% sole best) compared with the 2%/2% (2%/2% sole best) for HEBO$_p$/HEBO$_r$; none as best for Flash$_p$/Flash$_r$ (`Flash` has failed to terminate reasonably for some systems with large space due to its exhaustive nature); none as best for SMAC$_p$/SMAC$_r$; 6%/0% (4%/0% sole best) for TurBO$_p$/TurBO$_r$; and 4%/4% (4%/4% sole best) for Bounce$_p$/Bounce$_r$. CoTune also has the best average satisfaction and ranks with an improvement up to 2.9× on satisfaction (against Bounce$_p$). For existing tuners, tuning with and without a requirement do not differ much. This is because of their model-based nature: the uncertain model prediction has weakened the positive impact that could have been brought by the requirement as well as the harm raised from the loss of convergence and stagnating at local optima.

From Figure 12, we see that `CoTune` needs some limited resources to correctly co-evolve the auxiliary proposition initially (≈ 20 budget), but it then performs constantly better than the others over different budgets, which confirms its efficiency.

RQ2: *`CoTune` considerably outperforms the state-of-the art tuners: it is ranked first for 89% cases (87% sole first) against the 2% cases of the overall second best `HEBO`, with up to 2.9× improved satisfaction and higher efficiency.*

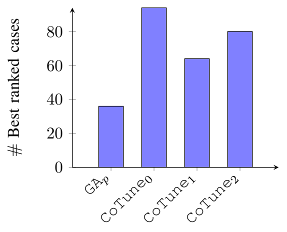
C. Ablation Study

1) Method: To answer **RQ3**, we assess several variants of `CoTune`: CoTune$_0$, CoTune$_1$, and CoTune$_2$ that only evolves p_a under **Case 0**, **Case 1**, and **Case 2**, respectively. We compare them with GA$_p$, i.e., `CoTune` without any co-evolution. All other settings are the same as **RQ1** and **RQ2**.

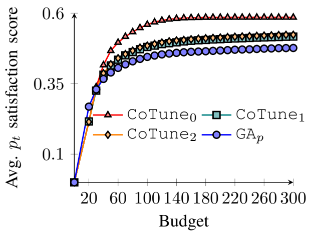
2) Results: As shown in Figure 13a, all variants of `CoTune` performs generally better than GA$_p$: they have 94 (CoTune$_0$), 64 (CoTune$_1$), and 80 (CoTune$_2$) best ranked cases against the 36 for GA$_p$—up to 2.61× better. The results also imply that **Case 0** and **Case 2** are more devastating than **Case 1**. For the trajectory in Figure 13b, all variants need some resources to adapt to the right direction at 20 budget, after which they perform constantly better than the one without co-evolution—a more promising efficiency.

TABLE III: Comparing `CoTune` with state-of-the-art tuners under 300 budgets/30 runs by averaging all three types of requirements $p_{t,1}$, $p_{t,2}$, and $p_{t,3}$. ✗ denotes failed to complete in a reasonable time. The format follows Table II. More detailed data can be found at: https://github.com/ideas-labo/CoTune/blob/main/rqSupplementary/RQ2.pdf.

d%	**System**	CoTune	HEBO$_p$	HEBO$_r$	Flash$_p$	Flash$_r$	SMAC$_p$	SMAC$_r$	TurBO$_p$	TurBO$_r$	Bounce$_p$	Bounce$_r$
	7Z	.28±.35 (1.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.01 (1.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)
	KANZI	.00±.02 (1.00)	.00±.00 (1.33)	.00±.00 (1.33)	✗	✗	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)
	EXASTENCILS	.67±.47 (1.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)
	APACHE	.00±.00 (1.00)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)
0.1%	SQLITE	.01±.07 (1.00)	.00±.00 (2.00)	.00±.00 (2.00)	✗	✗	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)	.00±.00 (2.00)
	DCONVERT	.01±.06 (1.00)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)	.00±.00 (1.33)
	DEEPARCH	.70±.42 (1.00)	.10±.23 (2.00)	.00±.00 (3.67)	.00±.00 (3.67)	.00±.00 (3.67)	.10±.21 (2.00)	.00±.00 (3.67)	.00±.00 (3.67)	.00±.00 (3.67)	.01±.03 (3.00)	.00±.00 (3.67)
	JUMP3R	.00±.00 (1.00)	.00±.00 (1.67)	.00±.00 (1.67)	✗	✗	.00±.00 (1.67)	.00±.00 (1.67)	.00±.00 (1.67)	.00±.00 (1.67)	.00±.00 (1.67)	.00±.00 (1.67)
	HSMGP	.81±.35 (1.00)	.00±.00 (2.00)	.00±.00 (2.33)	.00±.00 (2.33)	.00±.00 (2.33)	.00±.01 (1.67)	.00±.00 (2.33)	.00±.00 (2.33)	.00±.00 (2.33)	.00±.00 (2.33)	.00±.00 (2.33)
	7Z	.35±.29 (1.00)	.01±.03 (3.33)	.01±.03 (3.33)	.01±.03 (3.33)	.01±.03 (3.33)	.01±.03 (3.33)	.01±.03 (3.33)	.05±.13 (2.00)	.01±.03 (3.33)	.01±.08 (2.67)	.01±.05 (3.33)
	KANZI	.07±.22 (1.00)	.03±.14 (1.67)	.03±.14 (1.67)	✗	✗	.03±.14 (1.67)	.03±.14 (1.67)	.03±.14 (1.67)	.03±.14 (1.67)	.00±.00 (2.67)	.00±.00 (2.67)
	EXASTENCILS	.91±.11 (1.00)	.08±.17 (3.33)	.27±.33 (2.00)	.03±.12 (4.00)	.03±.12 (4.00)	.11±.20 (3.00)	.03±.11 (4.00)	.01±.03 (4.67)	.04±.12 (4.00)	.01±.03 (5.33)	.00±.00 (6.33)
	APACHE	.02±.08 (4.33)	.33±.35 (1.33)	.38±.39 (1.33)	.03±.10 (4.00)	.02±.10 (4.33)	.15±.26 (2.67)	.05±.18 (3.67)	.28±.33 (1.67)	.02±.10 (4.33)	.29±.33 (1.67)	.24±.32 (1.67)
1%	SQLITE	.17±.27 (1.00)	.04±.13 (2.33)	.05±.15 (2.00)	✗	✗	.04±.14 (2.33)	.05±.15 (2.00)	.05±.16 (2.00)	.05±.15 (2.00)	.00±.00 (3.33)	.00±.00 (3.33)
	DCONVERT	.30±.13 (1.00)	.03±.07 (2.00)	.03±.07 (2.00)	.03±.07 (2.00)	.03±.07 (2.00)	.03±.07 (2.00)	.03±.07 (2.00)	.03±.07 (2.00)	.03±.07 (2.00)	.02±.06 (2.00)	.02±.06 (2.00)
	DEEPARCH	.89±.20 (1.00)	.57±.33 (2.33)	.22±.07 (4.00)	.00±.00 (6.67)	.00±.00 (6.67)	.48±.36 (2.67)	.00±.00 (6.67)	.00±.00 (6.67)	.00±.00 (6.67)	.08±.22 (4.67)	.00±.01 (5.67)
	JUMP3R	.07±.18 (1.00)	.02±.12 (1.67)	.02±.12 (1.67)	✗	✗	.02±.12 (1.67)	.02±.12 (1.67)	.02±.12 (1.67)	.02±.12 (1.67)	.00±.00 (2.67)	.00±.00 (2.67)
	HSMGP	.96±.15 (1.00)	.07±.20 (2.00)	.06±.20 (2.00)	.06±.20 (2.00)	.06±.20 (2.00)	.06±.20 (2.00)	.06±.20 (2.00)	.06±.20 (2.00)	.06±.20 (2.00)	.03±.15 (2.33)	.00±.00 (3.33)
	7Z	.64±.32 (1.00)	.18±.28 (3.33)	.19±.29 (3.33)	.19±.29 (3.33)	.19±.29 (3.33)	.33±.33 (2.00)	.18±.29 (3.33)	.25±.35 (2.67)	.19±.29 (3.33)	.03±.15 (4.67)	.03±.07 (4.33)
	KANZI	.22±.30 (1.33)	.20±.28 (1.33)	.21±.29 (1.33)	✗	✗	.21±.29 (1.33)	.21±.29 (1.33)	.22±.29 (1.33)	.21±.29 (1.33)	.09±.16 (2.33)	.09±.16 (2.33)
	EXASTENCILS	.94±.11 (1.00)	.26±.23 (2.67)	.33±.36 (2.00)	.07±.17 (5.33)	.07±.17 (5.33)	.21±.25 (3.00)	.06±.17 (5.33)	.13±.15 (4.00)	.11±.19 (4.33)	.02±.07 (6.33)	.00±.01 (7.33)
	APACHE	.23±.12 (3.67)	.48±.27 (1.00)	.26±.26 (3.00)	.06±.14 (5.00)	.06±.14 (5.00)	.22±.21 (3.67)	.08±.18 (5.00)	.42±.22 (2.00)	.07±.14 (5.00)	.42±.22 (1.67)	.39±.21 (2.00)
5%	SQLITE	.29±.29 (1.00)	.13±.20 (2.00)	.12±.20 (2.00)	✗	✗	.14±.20 (2.00)	.14±.20 (2.00)	.14±.20 (2.00)	.14±.20 (2.00)	.00±.00 (3.00)	.00±.00 (3.00)
	DCONVERT	.67±.15 (1.00)	.19±.27 (2.00)	.17±.25 (2.00)	.18±.25 (2.00)	.18±.25 (2.00)	.18±.25 (2.00)	.18±.25 (2.00)	.18±.25 (2.00)	.17±.25 (2.00)	.07±.17 (3.00)	.07±.17 (3.00)
	DEEPARCH	.99±.01 (1.00)	.93±.03 (2.00)	.92±.03 (3.00)	.15±.33 (6.00)	.15±.33 (6.00)	.75±.33 (4.00)	.15±.33 (6.00)	.15±.33 (6.00)	.15±.33 (6.00)	.30±.41 (5.00)	.05±.18 (7.00)
	JUMP3R	.14±.28 (1.00)	.06±.21 (1.67)	.06±.21 (1.67)	✗	✗	.06±.21 (1.67)	.06±.21 (1.67)	.06±.21 (1.67)	.06±.21 (1.67)	.03±.13 (2.00)	.03±.13 (2.00)
	HSMGP	.99±.06 (1.00)	.29±.34 (2.00)	.23±.32 (2.67)	.21±.32 (3.00)	.20±.31 (3.33)	.29±.33 (2.00)	.20±.31 (3.33)	.20±.32 (3.00)	.21±.32 (3.00)	.17±.30 (3.67)	.12±.25 (4.00)
	7Z	.81±.17 (1.00)	.34±.31 (3.67)	.39±.31 (3.00)	.40±.30 (3.00)	.38±.31 (3.00)	.48±.26 (2.00)	.38±.31 (3.00)	.40±.35 (2.67)	.40±.31 (3.00)	.24±.31 (4.67)	.19±.21 (5.00)
	KANZI	.58±.24 (1.00)	.52±.25 (1.33)	.52±.25 (1.33)	✗	✗	.52±.26 (1.33)	.52±.25 (1.33)	.52±.25 (1.33)	.52±.25 (1.33)	.29±.29 (2.33)	.28±.29 (2.33)
	EXASTENCILS	.98±.05 (1.00)	.61±.16 (3.00)	.70±.18 (2.00)	.25±.22 (6.00)	.24±.22 (6.00)	.30±.21 (5.00)	.22±.21 (6.00)	.35±.22 (4.00)	.28±.24 (5.00)	.07±.14 (7.00)	.03±.08 (8.00)
	APACHE	.32±.06 (3.00)	.39±.19 (2.00)	.27±.09 (4.00)	.16±.14 (5.33)	.15±.14 (5.33)	.21±.15 (5.00)	.17±.18 (5.33)	.45±.17 (1.00)	.17±.14 (5.33)	.44±.16 (1.00)	.43±.15 (1.00)
20%	SQLITE	.54±.20 (1.00)	.37±.18 (2.00)	.37±.18 (2.00)	✗	✗	.37±.19 (2.00)	.37±.19 (2.00)	.37±.19 (2.00)	.37±.19 (2.00)	.08±.13 (3.00)	.08±.13 (3.00)
	DCONVERT	.92±.03 (1.00)	.49±.30 (2.00)	.33±.23 (3.00)	.34±.23 (3.00)	.31±.24 (3.00)	.34±.23 (3.00)	.31±.24 (3.00)	.34±.23 (3.00)	.32±.24 (3.00)	.21±.23 (4.00)	.17±.22 (4.33)
	DEEPARCH	1.0±.00 (1.00)	1.0±.00 (2.33)	1.0±.00 (2.33)	.56±.29 (5.00)	.56±.28 (5.00)	.70±.18 (3.67)	.55±.29 (5.00)	.59±.29 (4.67)	.56±.29 (5.00)	.57±.25 (4.67)	.34±.30 (6.00)
	JUMP3R	.17±.32 (1.00)	.08±.22 (1.67)	.08±.22 (1.67)	✗	✗	.08±.22 (1.67)	.08±.22 (1.67)	.08±.22 (1.67)	.07±.20 (2.00)	.03±.15 (2.67)	.03±.15 (2.67)
	HSMGP	.98±.09 (1.00)	.80±.27 (2.00)	.72±.33 (3.00)	.60±.39 (4.33)	.59±.40 (4.33)	.68±.34 (3.33)	.60±.39 (4.33)	.58±.40 (4.33)	.66±.35 (3.33)	.57±.37 (4.33)	.53±.38 (4.33)
	7Z	.86±.15 (1.00)	.54±.22 (2.33)	.54±.23 (2.33)	.56±.23 (2.33)	.54±.23 (2.33)	.54±.23 (2.33)	.54±.23 (2.33)	.49±.28 (3.33)	.56±.24 (2.00)	.36±.11 (4.67)	.41±.15 (3.67)
	KANZI	.59±.17 (1.00)	.52±.16 (2.00)	.52±.16 (2.00)	✗	✗	.52±.16 (2.00)	.51±.16 (2.33)	.52±.16 (2.00)	.52±.16 (2.00)	.37±.18 (3.33)	.37±.18 (3.33)
	EXASTENCILS	.95±.07 (1.00)	.54±.14 (2.67)	.58±.11 (2.00)	.29±.16 (4.67)	.29±.16 (4.67)	.29±.16 (4.67)	.28±.17 (4.67)	.37±.16 (3.67)	.34±.18 (3.67)	.12±.07 (5.67)	.11±.09 (6.33)
	APACHE	.58±.09 (2.67)	.59±.19 (2.33)	.59±.15 (2.67)	.40±.15 (4.67)	.39±.15 (4.67)	.42±.16 (4.33)	.41±.18 (4.67)	.69±.17 (1.00)	.40±.15 (4.67)	.72±.17 (1.00)	.69±.17 (1.00)
50%	SQLITE	.74±.19 (1.00)	.53±.17 (2.33)	.52±.17 (2.33)	✗	✗	.52±.17 (2.33)	.53±.17 (2.33)	.52±.16 (2.33)	.54±.17 (2.00)	.27±.14 (3.33)	.27±.14 (3.33)
	DCONVERT	.92±.03 (1.00)	.52±.28 (2.00)	.41±.24 (3.00)	.43±.23 (3.00)	.41±.24 (3.00)	.43±.23 (3.00)	.41±.24 (3.00)	.44±.22 (2.67)	.41±.23 (3.00)	.31±.20 (4.00)	.29±.20 (4.33)
	DEEPARCH	1.0±.00 (1.00)	1.0±.00 (2.33)	1.0±.00 (2.33)	.84±.10 (3.67)	.83±.12 (3.67)	.84±.10 (3.67)	.83±.11 (3.67)	.83±.12 (3.67)	.83±.12 (3.67)	.82±.06 (4.00)	.76±.15 (5.00)
	JUMP3R	.41±.27 (1.00)	.27±.18 (2.00)	.28±.18 (2.00)	✗	✗	.28±.18 (2.00)	.28±.18 (2.00)	.28±.18 (2.00)	.27±.18 (2.00)	.18±.14 (3.00)	.18±.14 (3.00)
	HSMGP	.99±.04 (1.00)	.84±.17 (2.00)	.81±.19 (2.33)	.74±.23 (3.67)	.73±.24 (4.00)	.82±.16 (2.33)	.74±.23 (4.00)	.76±.22 (3.67)	.75±.22 (3.33)	.71±.22 (4.67)	.70±.22 (4.67)
	7Z	.84±.24 (1.00)	.44±.27 (2.00)	.44±.27 (2.00)	.44±.27 (2.00)	.44±.27 (2.00)	.44±.27 (2.00)	.44±.27 (2.00)	.41±.30 (2.33)	.44±.27 (2.00)	.23±.10 (4.33)	.28±.14 (3.33)
	KANZI	.35±.23 (1.00)	.30±.19 (1.67)	.29±.19 (1.67)	✗	✗	.29±.19 (1.67)	.29±.19 (1.67)	.29±.19 (1.67)	.29±.19 (1.67)	.15±.14 (2.67)	.15±.14 (2.67)
	EXASTENCILS	.96±.07 (1.00)	.58±.16 (2.33)	.58±.13 (2.33)	.35±.14 (4.67)	.35±.13 (4.67)	.34±.15 (4.67)	.34±.15 (4.67)	.39±.12 (3.67)	.37±.15 (4.00)	.20±.11 (5.67)	.18±.07 (6.00)
	APACHE	.92±.00 (3.33)	.93±.02 (3.33)	.95±.04 (1.00)	.92±.01 (4.33)	.92±.01 (4.33)	.92±.01 (4.33)	.92±.02 (4.00)	.92±.01 (4.33)	.92±.01 (4.33)	.94±.03 (2.00)	.94±.03 (2.00)
90%	SQLITE	.70±.19 (1.00)	.52±.16 (2.00)	.52±.16 (2.00)	✗	✗	.53±.16 (2.00)	.53±.16 (2.00)	.54±.16 (2.00)	.53±.16 (2.00)	.32±.08 (3.00)	.32±.08 (3.00)
	DCONVERT	.90±.07 (1.00)	.55±.29 (2.00)	.52±.27 (2.33)	.52±.27 (2.33)	.52±.27 (2.33)	.52±.27 (2.33)	.52±.27 (2.33)	.53±.27 (2.33)	.53±.27 (2.33)	.36±.25 (3.33)	.36±.25 (3.33)
	DEEPARCH	1.0±.00 (1.00)	1.0±.00 (2.00)	1.0±.00 (2.33)	.83±.12 (3.33)	.83±.12 (3.33)	.83±.12 (3.33)	.83±.12 (3.33)	.84±.11 (3.33)	.83±.12 (3.33)	.80±.10 (4.33)	.76±.15 (5.00)
	JUMP3R	.23±.25 (1.00)	.14±.19 (1.67)	.15±.21 (1.67)	✗	✗	.15±.21 (1.67)	.15±.21 (1.67)	.15±.21 (1.67)	.15±.21 (1.67)	.09±.15 (2.67)	.09±.15 (2.67)
	HSMGP	.99±.04 (1.00)	.81±.24 (2.00)	.77±.27 (2.33)	.66±.34 (4.00)	.65±.35 (4.33)	.66±.34 (4.00)	.65±.34 (4.33)	.68±.33 (3.67)	.72±.30 (3.33)	.59±.34 (5.00)	.58±.34 (5.00)
Average p_t score/rank		.58/1.23	.36/2.09	.34/2.22	.29/3.57	.28/3.61	.30/2.56	.26/2.99	.29/2.59	.26/2.87	.21/3.27	.20/3.48

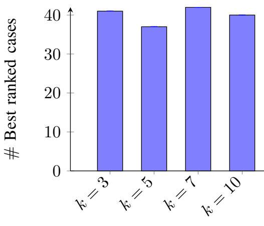


(a) Full budget

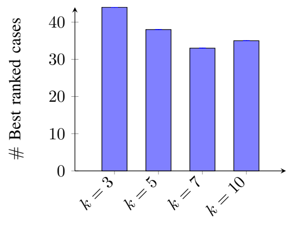


(b) Tuning trajectory

Fig. 13: Ablation analysis of `CoTune` over all cases/runs.



(a) $d \in \{0.1\%, 1\%, 5\%\}$



(b) $d \in \{20\%, 50\%, 90\%\}$

Fig. 14: Sensitivity of `CoTune` to stagnation indicator k.

RQ3: *The co-evolution strategies that handles all three cases in* `CoTune` *are beneficial while handling **Case 0** and **Case 2** are more impactful than that of **Case 1**.*

D. RQ4: Sensitivity of `CoTune` *to k*

1) Method: The key parameter for `CoTune` is k, which sets how many iterations without improvement we need to observe to confirm stagnation. We examine `CoTune` under different k values—$k \in \{3, 5, 7, 10\}$—with the same settings as before.

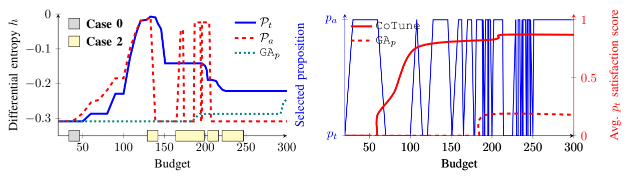
2) Results: In Figure 14, we see that $k = 3$ is generally better than the others due to its more excessive stagnation mitigation. Yet, for harder requirements with $d \in \{0.1\%, 1\%, 5\%\}$ (Figure 14a), the benefit of $k = 3$ become more blurred against the results of easier requirements (Figure 14b). This is because in those cases the stagnation in **Case 2** is much less likely to occur, since most commonly **Case 0** and **Case 1** are the main issues caused by the target performance requirement.

RQ4: *$k = 3$ is generally better for more excessive stagnation mitigation, especially on hard requirements.*

VI. DISCUSSION

A. Behind the Scenes: Why `CoTune` *Works?*

As from Figure 15a, `CoTune` initially encounters **Case 0** and hence increases the entropy by co-evolving p_a; from 145 budget onward it faces several occasions of **Case 2**, and



(a) Entropy value (b) Selected proposition/p_t values

Fig. 15: A randomly chosen example run for SQLite.

hence it seeks to keep the entropy minimized via the co-evolution. This influences the evolution of both $\mathcal{P}_a$ and $\mathcal{P}_b$. Therefore, in Figure 15b, the proposition used to guide the tuning is also more frequently switched at the later half than the first half budget. The co-evolution becomes stable from around 250 budgets onward. Without the co-evolution, the GA$_p$ (Figure 15a) exhibits low entropy throughout since most configurations have $p_t = 0$ for a long period, and this is only changed at 193 budget point, after which it suffers stagnation. As a result, GA$_p$ achieves only $p_t = 0.18$ against the $p_t = 0.87$ for CoTune, which also has a much steeper/better trajectory.

B. *Threats to Validity*

To mitigate threats to **internal validity**, the parameters are pragmatically set based on their implication (e.g., k in CoTune) or follow prior works [2], [3], [14] (e.g., mutation rate and budget), but indeed these can be consolidated.

Since our goal is to satisfy the requirement, we use the systematically quantified satisfaction score as the main metric to ensure **construct validity**. We have also examined the trajectories under different budget settings with Scott-Knott ESD test. However, unintentional ignorance might still exist.

To ensure **external validity**, we evaluate CoTune against four state-of-the-art tuners (with and without requirements) over nine systems from diverse domains and 18 different target performance requirements, leading to 162 cases. Yet, we agree that more subjects might improve the generalizability.

VII. Related Work

General Configuration Tuning: Tuning configurations have been generally done assuming that the better performance would always be more preferred. Genetic Algorithm has been widely used as the core of a tuner [11], [13]–[15]. Recently, MMO [14], [57] is an optimization model that multi-objectvizes configuration tuning, overcoming local optima for the target performance objective via an extra performance metric. Other tuners leverage a surrogate model [58]–[62] to expedite the tuning via a variant of Bayesian optimization [3], [16], [17], [47], e.g., PromiseTune [23], with diverse internal designs.

All the above tuners have ignored the valuable information in a given performance requirement—the key in CoTune.

Performance Requirements Understandings: Languages and notations exist for formulating performance requirements [18], [25], [63]. For example, Eckhardt et al. [25] present a framework of patterns to systematically interpret the performance requirements. However, they have not systematically quantified them. Chen and Li [18] study the impacts of using performance requirements as an objective for configuration tuning. Their findings suggest that reasonably elicited requirements can help, but poorly specified ones are harmful. Yet, they have neither defined to what extent a performance requirement is appropriate nor proposed any automated tools. CoTune, on the other hand, is an automated tuner that can robustly leverage the performance requirement.

Requirement-guided Configuration Tuning: There exist some tuners that incorporate the hard constraint of performance requirement as the objective to guide the configuration tuning using GA [13], [15], [64]. Ghanbari et al. [65] adopt the same and they assume quadratically quantified requirements. Yet, those tuners differ from CoTune in the following:

- They work on a fixed type of performance requirement while CoTune has no such a constraint.
- They assume a given requirement is always elicited perfectly/reasonably, which is unrealistic. CoTune works well even if the given target one is harmful to the tuning.

A recent tool [66] permits interactive configuration tuning, allowing developers to specify preferences between performance metrics at tuning. Although like CoTune, it does gradually adjust the requirements, the process is however relies on human intervention; CoTune does so automatically.

Constrained Tuning: There exist approaches for constrained configuration tuning [67], [68] or general constrained optimization [69], [70]. However, those differ from CoTune in the sense that their constraint definitions are mostly on the dependency constraints/extra constraints in the solution/configuration space[7], while our problem places constraint on the same single-objective/performance metric to be optimized, which is complementary to the definition in prior works.

VIII. Conclusion

This paper presents CoTune, an automated tool that co-evolves a given performance requirement and configurations. Instead of purely leveraging the target performance requirement as an objective or completely ignoring it, CoTune builds an auxiliary performance requirement and co-evolve it with the configurations to mitigate different situations based on the discriminative power of the tuning. We show that CoTune can:

- achieve significantly superior results than tuning with static target performance requirement and without;
- considerably outperform state-of-the-art tuners regardless if they use the requirement as the objective;
- doing so with generally better resource efficiency.

The future work extending from CoTune is fruitful, including the multi-objective cases [71], [72] and incorporating the information of configuration landscape [73] into the co-evolution.

Acknowledgement

This work was supported by a NSFC Grant (62372084) and a UKRI Grant (10054084).

[7]CoTune sets a configuration that violates dependency constraint with the worst possible value, and hence it would be naturally eliminated during tuning iterations. This is the simplest and standard method used in prior work [7].

References

[1] M. Sayagh, N. Kerzazi, B. Adams, and F. Petrillo, "Software configuration engineering in practice interviews, survey, and systematic literature review," *IEEE Trans. Software Eng.*, vol. 46, no. 6, pp. 646–673, 2020.

[2] P. Chen, J. Gong, and T. Chen, "Accuracy can lie: On the impact of surrogate model in configuration tuning," *IEEE Transactions on Software Engineering*, vol. 51, no. 2, pp. 548–580, 2025.

[3] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding faster configurations using FLASH," *IEEE Trans. Software Eng.*, vol. 46, no. 7, pp. 794–811, 2020. [Online]. Available: https://doi.org/10.1109/TSE.2018.2870895

[4] H. Liang, Y. Huang, and T. Chen, "The same only different: On information modality for configuration performance analysis," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, Canada*. IEEE, 2025, pp. 2522–2534. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00212

[5] Y. Ma, T. Chen, and K. Li, "Faster configuration performance bug testing with neural dual-level prioritization," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 988–1000. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00201

[6] https://www.evolven.com/blog/downtime-outages-and-failures-understanding-their-true-costs.html, retrieved on May 1st, 2025.

[7] T. Chen and M. Li, "Adapting multi-objectivized software configuration tuning," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 539–561, 2024. [Online]. Available: https://doi.org/10.1145/3643751

[8] Y. Zhu, J. Liu, M. Guo, Y. Bao, W. Ma, Z. Liu, K. Song, and Y. Yang, "Bestconfig: tapping the performance potential of systems via automatic configuration tuning," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 338–350.

[9] T. Chen, "Lifelong dynamic optimization for self-adaptive systems: Fact or fiction?" in *IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2022, Honolulu, HI, USA, March 15-18, 2022*. IEEE, 2022, pp. 78–89. [Online]. Available: https://doi.org/10.1109/SANER53432.2022.00022

[10] T. Chen, K. Li, R. Bahsoon, and X. Yao, "FEMOSAA: feature-guided and knee-driven multi-objective optimization for self-adaptive software," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 2, pp. 5:1–5:50, 2018. [Online]. Available: https://doi.org/10.1145/3204459

[11] J. Lee, S. Seo, J. Choi, and S. Park, "K2vtune: A workload-aware configuration tuning for rocksdb," *Inf. Process. Manag.*, vol. 61, no. 1, p. 103567, 2024. [Online]. Available: https://doi.org/10.1016/j.ipm.2023.103567

[12] Y. Ye, T. Chen, and M. Li, "Distilled lifelong self-adaptation for configurable systems," in *47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025*. IEEE, 2025, pp. 1333–1345. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00094

[13] S. Gerasimou, R. Calinescu, and G. Tamburrelli, "Synthesis of probabilistic models for quality-of-service software engineering," *Autom. Softw. Eng.*, vol. 25, no. 4, pp. 785–831, 2018. [Online]. Available: https://doi.org/10.1007/s10515-018-0235-8

[14] P. Chen, T. Chen, and M. Li, "MMO: meta multi-objectivization for software configuration tuning," *IEEE Trans. Software Eng.*, vol. 50, no. 6, pp. 1478–1504, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3388910

[15] A. Martens, H. Koziolek, S. Becker, and R. H. Reussner, "Automatically improve software architecture models for performance, reliability, and cost using evolutionary algorithms," in *Proceedings of the first joint WOSP/SIPEW International Conference on Performance Engineering, San Jose, California, USA, January 28-30, 2010*. ACM, 2010, pp. 105–116. [Online]. Available: https://doi.org/10.1145/1712605.1712624

[16] P. Jamshidi and G. Casale, "An uncertainty-aware approach to optimal configuration of stream processing systems," in *24th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS 2016, London, United Kingdom, September 19-21, 2016*. IEEE Computer Society, 2016, pp. 39–48. [Online]. Available: https://doi.org/10.1109/MASCOTS.2016.17

[17] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *Learning and Intelligent Optimization: 5th International Conference, LION 5, Rome, Italy*. Springer, 2011, pp. 507–523.

[18] T. Chen and M. Li, "Do performance aspirations matter for guiding software configuration tuning? an empirical investigation under dual performance objectives," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 3, pp. 68:1–68:41, 2023. [Online]. Available: https://doi.org/10.1145/3571853

[19] "The pure reqruierment dataset," https://zenodo.org/records/1414117, 2005, retrieved on Jan 01, 2025.

[20] L. A. Zadeh, "Fuzzy logic," *Computer*, vol. 21, no. 4, pp. 83–93, 1988.

[21] X. Ma, X. Li, Q. Zhang, K. Tang, Z. Liang, W. Xie, and Z. Zhu, "A survey on cooperative co-evolutionary algorithms," *IEEE Trans. Evol. Comput.*, vol. 23, no. 3, pp. 421–441, 2019. [Online]. Available: https://doi.org/10.1109/TEVC.2018.2868770

[22] A. Fekry, L. Carata, T. F. J. Pasquier, A. Rice, and A. Hopper, "Towards seamless configuration tuning of big data analytics," in *39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019*. IEEE, 2019, pp. 1912–1919. [Online]. Available: https://doi.org/10.1109/ICDCS.2019.00189

[23] P. Chen and T. Chen, "Promisetune: Unveiling causally promising and explainable configuration tuning," in *48th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2026.

[24] K. K. Ganguly and T. Menzies, "Bingo! simple optimizers win big if problems collapse to a few buckets," *CoRR*, vol. abs/2506.04509, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2506.04509

[25] J. Eckhardt, A. Vogelsang, H. Femmer, and P. Mager, "Challenging incompleteness of performance requirements by sentence patterns," in *24th IEEE International Requirements Engineering Conference, RE 2016, Beijing, China*. IEEE Computer Society, 2016, pp. 46–55.

[26] A. D. Bull, "Convergence rates of efficient global optimization algorithms," *J. Mach. Learn. Res.*, vol. 12, pp. 2879–2904, 2011. [Online]. Available: https://dl.acm.org/doi/10.5555/1953048.2078198

[27] G. D. Ath, R. M. Everson, A. A. Rahat, and J. E. Fieldsend, "Greed is good: Exploration and exploitation trade-offs in bayesian optimisation," *ACM Trans. Evol. Learn. Optim.*, vol. 1, no. 1, pp. 1:1–1:22, 2021. [Online]. Available: https://doi.org/10.1145/3425501

[28] "The promise repository of software engineering databases," https://zenodo.org/records/268542, 2005, retrieved on Jan 01, 2025.

[29] A. Ferrari, G. O. Spagnolo, and S. Gnesi, "PURE: A dataset of public requirements documents," in *25th IEEE International Requirements Engineering Conference, RE 2017, Lisbon, Portugal, September 4-8, 2017*. IEEE Computer Society, 2017, pp. 502–505.

[30] Z. S. Shaukat, R. Naseem, and M. Zubair, "A dataset for software requirements risk prediction," in *2018 IEEE International Conference on Computational Science and Engineering, CSE 2018, Bucharest, Romania, October 29-31, 2018*. IEEE Computer Society, 2018, pp. 112–118. [Online]. Available: https://doi.org/10.1109/CSE.2018.00022

[31] "The reqruierment dataset mined by shaukat et al." https://zenodo.org/records/1209601, 2005, retrieved on Jan 01, 2025.

[32] D. E. Knuth, "backus normal form vs. backus naur form," *Commun. ACM*, vol. 7, no. 12, pp. 735–736, 1964. [Online]. Available: https://doi.org/10.1145/355588.365140

[33] H. Meth, M. Brhel, and A. Maedche, "The state of the art in automated requirements elicitation," *Inf. Softw. Technol.*, vol. 55, no. 10, pp. 1695–1709, 2013.

[34] G. R. Terrell and D. W. Scott, "Variable kernel density estimation," *The Annals of Statistics*, pp. 1236–1265, 1992.

[35] https://www.7-zip.org/.

[36] M. Weber, C. Kaltenecker, F. Sattler, S. Apel, and N. Siegmund, "Twins or false friends? A study on energy consumption and performance of configurable software," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2023, pp. 2098–2110. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00177

[37] https://github.com/flanglet/kanzi.

[38] https://www.exastencils.fau.de/.

[39] https://httpd.apache.org/.

[40] https://www.sqlite.org/.

[41] C. S. Mühlbauer, F. Sattler, and N. Siegmund, "Analyzing the impact of workloads on modeling the performance of configurable software systems," in *Proceedings of the International Conference on Software Engineering (ICSE), IEEE*, 2023.

[42] S. Mühlbauer, F. Sattler, C. Kaltenecker, J. Dorn, S. Apel, and N. Siegmund, "Analysing the impact of workloads on modeling the performance of configurable software systems," in *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023,*

Melbourne, Australia, May 14-20, 2023. IEEE, 2023, pp. 2085–2097. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00176

[43] I. Kök and S. Özdemir, "Deepmdp: A novel deep-learning-based missing data prediction protocol for iot," *IEEE Internet Things J.*, vol. 8, no. 1, pp. 232–243, 2021. [Online]. Available: https://doi.org/10.1109/JIOT.2020.3003922

[44] P. Jamshidi, M. Velez, C. Kästner, and N. Siegmund, "Learning to sample: exploiting similarities across environments to learn performance models for configurable systems," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*. ACM, 2018, pp. 71–82. [Online]. Available: https://doi.org/10.1145/3236024.3236074

[45] https://mvnrepository.com/artifact/de.sciss/jump3r.

[46] S. Kuckuk, B. Gmeiner, H. Köstler, and U. Rüde, "A generic prototype to benchmark algorithms and data structures for hierarchical hybrid grids," in *Proceedings of the International Conference on Parallel Computing, ParCo 2013*, vol. 25. IOS Press, 2013, pp. 813–822. [Online]. Available: https://doi.org/10.3233/978-1-61499-381-0-813

[47] A. I. Cowen-Rivers, W. Lyu, R. Tutunov, Z. Wang, A. Grosnit, R. R. Griffiths, A. M. Maraval, H. Jianye, J. Wang, J. Peters *et al.*, "Hebo: Pushing the limits of sample-efficient hyper-parameter optimisation," *Journal of Artificial Intelligence Research*, vol. 74, pp. 1269–1349, 2022.

[48] D. Eriksson, M. Pearce, J. R. Gardner, R. Turner, and M. Poloczek, "Scalable global optimization via local bayesian optimization," in *Advances in Neural Information Processing Systems, Vancouver, BC, Canada*, 2019, pp. 5497–5508.

[49] L. Papenmeier, L. Nardi, and M. Poloczek, "Bounce: Reliable high-dimensional bayesian optimization for combinatorial and mixed spaces," in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/05d2175de7ee637588d1b5ced8b15b32-Abstract-Conference.html

[50] ——, "Increasing the scope as you learn: Adaptive bayesian optimization in nested subspaces," in *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. [Online]. Available: http://papers.nips.cc/paper_files/paper/2022/hash/4b7439a4ab0b8e4bcb4e2412c6a10a58-Abstract-Conference.html

[51] N. H. Awad, N. Mallik, and F. Hutter, "DEHB: evolutionary hyberband for scalable, robust and efficient hyperparameter optimization," in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*. ijcai.org, 2021, pp. 2147–2153. [Online]. Available: https://doi.org/10.24963/ijcai.2021/296

[52] N. Mallik, E. Bergman, C. Hvarfner, D. Stoll, M. Janowski, M. Lindauer, L. Nardi, and F. Hutter, "Priorband: Practical hyperparameter optimization in the age of deep learning," in *Advances in Neural Information Processing Systems*, 2023. [Online]. Available: http://papers.nips.cc/paper_files/paper/2023/hash/1704fe7aaff33a54802b83a016050ab8-Abstract-Conference.html

[53] S. Falkner, A. Klein, and F. Hutter, "BOHB: robust and efficient hyperparameter optimization at scale," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. Proceedings of Machine Learning Research, vol. 80. PMLR, 2018, pp. 1436–1445. [Online]. Available: http://proceedings.mlr.press/v80/falkner18a.html

[54] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Trans. Software Eng.*, vol. 45, no. 7, pp. 683–711, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2018.2794977

[55] M. L. McHugh, "Multiple comparison analysis testing in anova," *Biochemia medica*, vol. 21, no. 3, pp. 203–209, 2011.

[56] P. E. McKight and J. Najab, "Kruskal-wallis test," *The corsini encyclopedia of psychology*, pp. 1–1, 2010.

[57] T. Chen and M. Li, "Multi-objectivizing software configuration tuning," in *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 2021, pp. 453–465. [Online]. Available: https://doi.org/10.1145/3468264.3468555

[58] Z. Xiang, J. Gong, and T. Chen, "Dually hierarchical drift adaptation for online configuration performance learning," in *48th IEEE/ACM International Conference on Software Engineering (ICSE)*. ACM, 2026.

[59] J. Gong, T. Chen, and R. Bahsoon, "Dividable configuration performance learning," *IEEE Trans. Software Eng.*, vol. 51, no. 1, pp. 106–134, 2025. [Online]. Available: https://doi.org/10.1109/TSE.2024.3491945

[60] J. Gong and T. Chen, "Predicting software performance with divide-and-learn," in *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, 2023, pp. 858–870. [Online]. Available: https://doi.org/10.1145/3611643.3616334

[61] ——, "Predicting configuration performance in multiple environments with sequential meta-learning," *Proceedings of ACM Software Engineering*, vol. 1, no. FSE, pp. 359–382, 2024. [Online]. Available: https://doi.org/10.1145/3643743

[62] T. Chen and R. Bahsoon, "Self-adaptive and online qos modeling for cloud-based software services," *IEEE Trans. Software Eng.*, vol. 43, no. 5, pp. 453–475, 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2608826

[63] J. Whittle, P. Sawyer, N. Bencomo, B. H. C. Cheng, and J. Bruel, "RELAX: a language to address uncertainty in self-adaptive systems requirement," *Requir. Eng.*, vol. 15, no. 2, pp. 177–196, 2010. [Online]. Available: https://doi.org/10.1007/s00766-010-0101-0

[64] T. Chen and R. Bahsoon, "Self-adaptive trade-off decision making for autoscaling cloud-based services," *IEEE Trans. Serv. Comput.*, vol. 10, no. 4, pp. 618–632, 2017. [Online]. Available: https://doi.org/10.1109/TSC.2015.2499770

[65] H. Ghanbari, B. Simmons, M. Litoiu, and G. Iszlai, "Feedback-based optimization of a private cloud," *Future Gener. Comput. Syst.*, vol. 28, no. 1, pp. 104–111, 2012. [Online]. Available: https://doi.org/10.1016/j.future.2011.05.019

[66] V. Cortellessa, J. A. Diaz-Pace, D. Di Pompeo, S. Frank, P. Jamshidi, M. Tucci, and A. van Hoorn, "Introducing interactions in multi-objective optimization of software architectures," *ACM Trans. Softw. Eng. Methodol.*, 2025. [Online]. Available: https://doi.org/10.1145/3712185

[67] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li, "Gptune: multitask learning for autotuning exascale applications," in *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*. ACM, 2021, pp. 234–246. [Online]. Available: https://doi.org/10.1145/3437801.3441621

[68] E. O. Hellsten, A. L. F. Souza, J. Lenfers, R. Lacouture, O. Hsu, A. Ejjeh, F. Kjolstad, M. Steuwer, K. Olukotun, and L. Nardi, "Baco: A fast and portable bayesian compiler optimization framework," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 2023, pp. 19–42. [Online]. Available: https://doi.org/10.1145/3623278.3624770

[69] G. D'Angelo and F. Palmieri, "GGA: A modified genetic algorithm with gradient-based local search for solving constrained optimization problems," *Inf. Sci.*, vol. 547, pp. 136–162, 2021. [Online]. Available: https://doi.org/10.1016/j.ins.2020.08.040

[70] D. Eriksson and M. Poloczek, "Scalable constrained bayesian optimization," in *The 24th International Conference on Artificial Intelligence and Statistics, AISTATS 2021, April 13-15, 2021, Virtual Event*, ser. Proceedings of Machine Learning Research, vol. 130. PMLR, 2021, pp. 730–738. [Online]. Available: http://proceedings.mlr.press/v130/eriksson21a.html

[71] M. Li, T. Chen, and X. Yao, "How to evaluate solutions in pareto-based search-based software engineering: A critical review and methodological guidance," *IEEE Trans. Software Eng.*, vol. 48, no. 5, pp. 1771–1799, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2020.3036108

[72] T. Chen and M. Li, "The weights can be harmful: Pareto search versus weighted search in multi-objective search-based software engineering," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 1, pp. 5:1–5:40, 2023. [Online]. Available: https://doi.org/10.1145/3514233

[73] T. Chen, "Planning landscape analysis for self-adaptive systems," in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2022, Pittsburgh, PA, USA, May 22-24, 2022*. ACM/IEEE, 2022, pp. 84–90. [Online]. Available: https://doi.org/10.1145/3524844.3528060