Accurate Modeling of Performance Histories for Evolving Software Systems

Stefan Mühlbauer
Bauhaus-University Weimar,
Germany

Sven Apel
Saarland University,
Germany

Norbert Siegmund
Bauhaus-University Weimar,
Germany

***Abstract*—Learning from the history of a software system's performance behavior does not only help discovering and locating performance bugs, but also identifying evolutionary performance patterns and general trends, such as when technical debt accumulates. Exhaustive regression testing is usually impractical, because rigorous performance benchmarking requires executing a realistic workload per revision, which results in large execution times. In this paper, we propose a novel *active revision sampling* approach, which aims at tracking and understanding a system's performance history by approximating the performance behavior of a software system across all of its revisions. In a nutshell, we iteratively sample and measure the performance of specific revisions that help us building an exact performance-evolution model, and we use Gaussian Process models to assess in which revision ranges our model is most uncertain with the goal to sample further revisions for measurement. We have conducted an empirical analysis of the evolutionary performance behavior modeled as a time series of the histories of six real-world software systems. Our evaluation demonstrates that Gaussian Process models are able to accurately estimate the performance-evolution history of real-world software systems with only few measurements and to reveal interesting behaviors and trends.**

I. Introduction

For most software systems, performance (e.g., response time) is a key success factor [1]. Performance is not only critical in real-time applications [2], but performance bottlenecks may render any kind of software unusable. Despite its relevance for software quality and user experience, performance assessment is all to often postponed and exercised too little and too late in the development process [3].

Performance is shaped by the software system's architecture and evolves along with code added, removed, and refactored in individual revisions[1]. The performance influence of evolutionary changes to a system can emerge cumulatively, such as when software evolves with little respect to its original architecture. This pattern of undisciplined architectural evolution has been described as "architectural erosion" [4] resulting in "technical debt" [5]. It often implies degrading performance, also known as *performance regression* [6].

Performance-related problems have been addressed from different perspectives. Most work on performance anomalies aims at identifying performance regression and at pinpointing the cause to an individual revision of the source code. The detection criteria for performance regression are manifold and comprise statistical significance tests [7], correlation analyses among performance metrics [8], and absolute or relative deviation thresholds that, when exceeded [9], indicate a possible regression. Performance regression is determined by the performance measurements of either performance benchmarks [10, 11] or unit tests [9, 12, 13].

All this work is able to detect performance regressions with high accuracy. Work by Heger et al. complements this task with a consecutive root-cause analysis [12]: it bisects the evolution history recursively until the revision that introduced the performance anomaly is isolated. The scope of this analysis is limited to the *local* performance evolution around a potential performance anomaly. While this is effective for understanding a single performance anomaly, it tells not much about whether there are notable, long-term regression trends, patterns in performance evolution, and possible indicators for future performance degradation. These can be learned only from the *global* performance evolution of a software system, and they are the basis to take proactive measures to direct maintenance tasks, apply refactorings, and postpone addition of new functionality.

The naive approach to obtain a model of the global performance-evolution history requires exhaustive performance assessment of every single revision. Take, for example, the highly influential Python package NUMPY. Its performance is constantly assessed for every code change by means of a microbenchmark suite[2]. Clearly, this huge effort is justified since the package is widely used as an integral part of most industrial-relevant Python libraries, such as SCIPY or TENSORFLOW. However, not every project exercises this level of performance assessment or has the resources to do so.

To obtain a good approximation of the performance evolution history of a software project, assessing all of its revisions is only rarely an option in practice. If the revision history is sampled along time, such as sampling all milestones/releases or flagged commits, this might indeed expose performance changes, but prohibits pinpointing interesting trends and anomalies to a distinct revision (rater than a segment of revisions) in an unbiased manner.

An assumption that underlies our approach is that a revision sampling strategy should be *adaptive* in that it *actively* searches for performance changes and that it should be agnostic of a software system's properties. For instance, it should not rely on the developers commit message discipline. Clearly, the goal

[1]We consider each commit to a repository as a new revision of the system.

[2]https://pv.github.io/numpy-bench/

is to model a software system's performance evolution history with *as few revision measurements as possible*. To this end, we propose *active revision sampling*, an active learning approach to model and estimate performance of arbitrary revisions of a software system. Starting with an initial sample of revisions, our approach interpolates the performance evolution history and iteratively expands the sample set with new revisions. The exploration of new sample revisions is guided by the uncertainty of the current interpolation model.

By means of an empirical analysis of the history of six real-world software systems, we demonstrate that performance evolution histories can be approximated with only a small fraction of all available revisions and that interesting performance behavior and trends can be revealed this way. The main contributions of this paper are:

- an analysis of the presence and frequency of notable performance changes of a selection of six real-world software systems; the analysis reveals common performance evolution characteristics and guides us in the choice of methods for estimating performance across arbitrary revisions (cf. Section II);
- an approach to efficiently learn and estimate the performance of a software system across arbitrary revisions with only few measurements (cf. Section III), along with an evaluation of feasibility and accuracy on six real-world systems (cf. Section IV).

II. Characterizing Performance Evolution

As software evolves, with every incremental modification of the code, performance can change as well. Performance changes may result from purposeful optimizations or as a byproduct of functional modifications of the software.

Our goal is to provide an efficient method to identify substantial performance changes over the life time of a software, which can be the basis of a root-cause analysis of performance degradation or a method to quantify the technical debt and development process of the software system with respect to performance.

Before we present our method in the next section, we first require a better understanding of how performance evolves, that is, what are common characteristics of time series of performance measurements across multiple commits. Identifying *time series properties* of performance evolution is key for developing an appropriate modeling approach and provides insights into *how* performance changes over a system's life time. For time series data, there exists a variety of prediction methods, especially in the category of auto-regressive models [14]. Most traditional methods, however, require knowledge of what characteristics the process to be modeled has. These characteristics usually include constant behavior (stationarity), continuous or gradual changes (trends), and periodic changes (seasonality). Moreover, the time series can show abrupt changes, also called *change points*, which suggest to model the time series' signal piecewise.

In what follows, we analyze the performance evolution of two configurable real-world software systems as well as four Python libraries with respect to change points. Knowing whether performance changes abruptly or continuously allows us to understand the limitations of traditional time series prediction techniques.

A. Change Point Detection with CUSUM

Our first step in assessing the characteristics of performance evolution is to systematically test our corpus of software systems for the presence of change points. The corresponding problem is often referred to as *change point detection*, and there is a variety of statistical methods available for this task. For our analysis, we use CUSUM, a method from statistical quality control [15]. CUSUM is a sequential algorithm that uses the cumulative sum of deviations between successive measurements. The basic idea is to maintain a cumulative sum of changes in positive and negative direction. A change point is indicated when the cumulative sum exceeds a given threshold. In this case, the cumulative sum of the respective positive or negative change is reset to zero. The algorithm behind CUSUM is defined by Page et al. as follows [15]: First, it initializes the positive g_0^+ and negative g_0^- cumulative sum with zero:

$$g_0^+ = g_0^- = 0 \tag{1}$$

Next, for each data point s_t of the time series at time t, the corresponding cumulative sums, g_t^+ and g_t^-, are defined as:

$$g_t^+ = \max(g_{t-1}^+ + s_t - v, 0) \tag{2}$$

$$g_t^- = \max(g_{t-1}^- - s_t - v, 0) \tag{3}$$

That is, the deviation of the current measurement s_t from a target value v (drift) is added/subtracted to the positive and negative cumulative sum. For every point in time, the cumulative sums are compared against a threshold value h, which, when exceeded, indicates a possible change point:

$$g_t^\pm = \begin{cases} 0 & \text{iff } g_t^\pm > h > 0 \\ g_t^\pm & \text{else} \end{cases} \tag{4}$$

The cumulative sum $g_t^\pm$ is then reset to zero to enable detection of following change points.

B. Parameterization of CUSUM

The threshold h describes a budget of change that, when exceeded, indicates a change point. Changes in the data set can emerge gradually or abruptly. The CUSUM method does not discriminate between these two types, as change can accumulate either way. This is an important property for our analysis not to be biased toward a certain expectation. However, along with the change points reported by CUSUM, one can consider only change points for which the absolute change compared to the preceding data point was substantial.

The CUSUM algorithm can be configured with two parameters, the drift v and the threshold h. A guideline of how to tune the parameters for a given data set is given by Gustafsson [16]. The guideline suggests to choose a drift v of "one half of the expected change". In our analysis, we employ a drift of

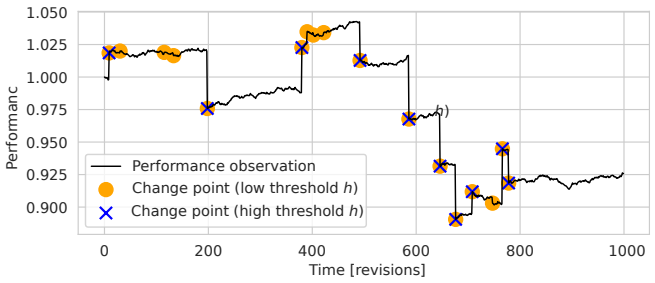


Figure 1: Influence of threshold parameter h on the detection of change points for a synthetic time series.

5 percent of the range of the time series since we consider a change of 10 percent in performance substantial.

$$v = 0.05 \cdot |\max(s) - \min(s)| \tag{5}$$

Next, we estimate and tune the threshold parameter h. The guideline suggests starting with a large value for h, which is then continuously decreased until the algorithm reports sufficiently few false positives. We adopt this procedure by using the average standard deviation of the time series, or more precisely, five times the average standard deviation over a sliding window. Instead of directly changing the threshold parameter directly, we tweak the size of the sliding window as described in Equation 6. The rationale behind this approach is to capture both global and local changes. For a small sliding window, the average standard deviation corresponds to a shorter time frame and, hence, is more sensitive to local changes. By contrast, for a large sliding window, the standard deviation incorporates a wider time frame and the threshold becomes less sensitive to local changes, as these are diluted by large-scale effects. We exemplify this influence of the sliding window size in Figure 1, where, for a smaller sliding window, more small change point candidates are reported (17 in total), whereas a larger sliding window size results in a more robust and large-scale view of possible change points (10 in total).

For an initial window size equal to the total number of data points, the threshold parameter corresponds to the standard deviation of the whole time series and draws a global picture of variation over the entire performance evolution history. We start with this initial case and decrease the window size iteratively. As we approach smaller values of the sliding window size w, the threshold parameter h approaches a more local measure of variation. Hence, with a decreasing window size the threshold becomes more sensitive to change points with smaller amplitudes, but also false positives. We summarize our definition of threshold h, where s denotes the overall number of revisions, w the sliding window size:

$$h = 5 \cdot \frac{1}{s-w} \sum_{p=0}^{s-w} \sum_{i=p}^{p+w} \sqrt{\text{Var}[s_p, \ldots, s_{p+w}]} \tag{6}$$

C. Change Points in Performance Data

In Figure 2, we report our findings of using CUSUM on six different software systems. We provide a detailed description of how we selected our subject systems and respective benchmarks in Section IV. In a nutshell, we have selected two configurable software systems, XZ and LRZIP, because recent findings suggest that performance bugs are often due to issues related to configurability and occur only in certain configurations [17]. That is, for each configuration, we derive a dedicated variant (as discussed in Section IV) with possibly different performance histories. For XZ, LRZIP, we use 47 and 71 different variants with identical workload. For the remaining four software systems, performance measurement was conducted at a finer grain with method-level microbenchmarks. Again, per method, we derived a specific performance history for these software systems. By untangling the selected systems in this manner, we obtain a more generalizable picture which includes whole system performance evolution and micro-system performance evolution.

We have aggregated the change-point analysis results in two different ways. For configurable systems, we report the change points across all variants; for microbenchmarks, we treat the sum of all method execution times as one single benchmark. Both decisions are due to space limitations and affect only the reporting in this paper. We have also analyzed all undiscussed benchmarks and provide the corresponding results online[3]. We consider the more condensed visualization of the sum of all microbenchmarks as representative for the entire software systems because, after analyzing all results, the same conclusions can be drawn from this.

For each subject system, we varied the sliding window size as a percentage of the total number of revisions and measured both the number of reported change points (in red) as well as the distribution of change point amplitudes as box plots. The range of the sliding window size presented in Figure 2 is chosen such that it includes a range where the number of reported change points is constant. For this range, the number of change points is least sensitive to the influence of the sliding window size. Each box plot was normalized to the mean of the time series to provide a context for interpreting this measure of "effect size". That is, the amplitude of a change point puts it in relation to the mean performance of the time series.

As expected, the number of change points decreases for a greater window size. For smaller window sizes, the reported change points include also smaller amplitudes. The rationale is that there should be fewer changes points on a global (i.e., whole history) scale than local changes points, and we can confirm this. However, our subject systems show substantial change points for different ranges of sliding window sizes (e.g., for 10 % sliding window size, change points are reported for XZ, but not for SCIPY).

This is due to the difference in sizes of the data sets: XZ, LRZIP, and ULTRAJSON do not contain as many revisions as the remaining systems, as shown in Table I. The results of Figure 2

[3]https://smba.github.io/active_revision_sampling/

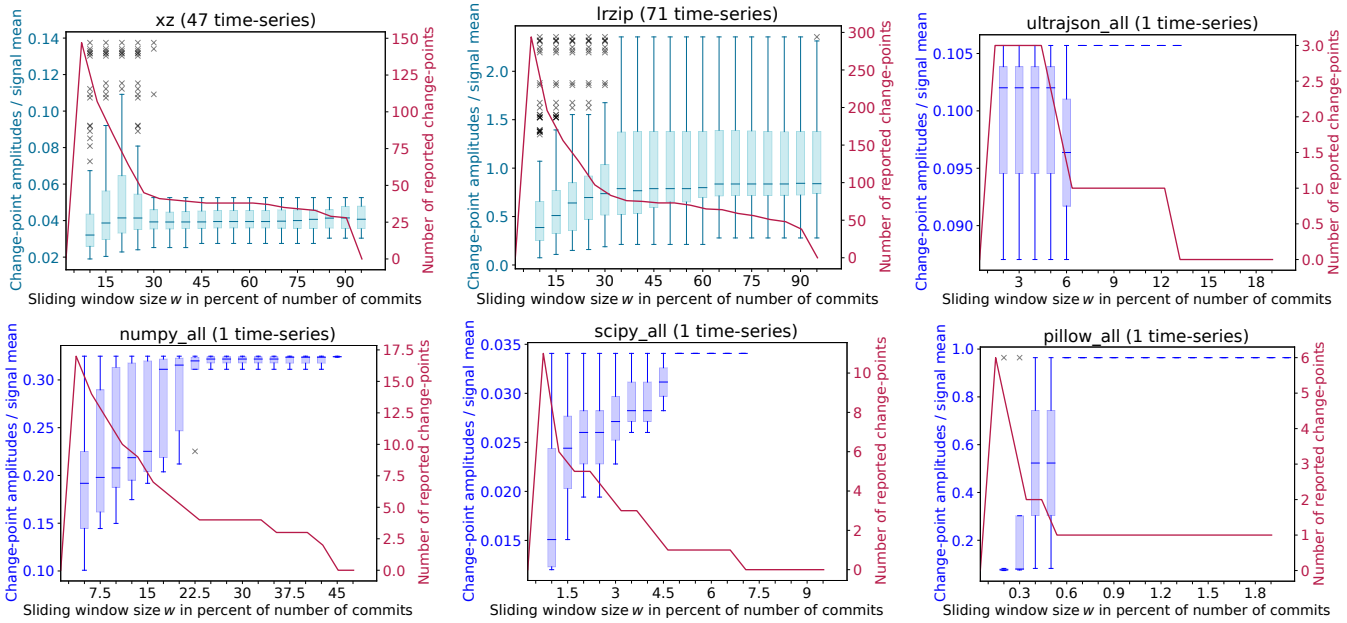


Figure 2: Frequency of reported change points vs sliding window size for threshold h. We present the analysis results for XZ, LRZIP, ULTRAJSON, NUMPY, SCIPY, and PILLOW (from top left to bottom right).

suggest that we are able to show and quantify the presence of substantial changes in performance introduced abruptly by single change points.

D. Results

The question we address in this section is whether performance histories can be modeled with traditional time series prediction techniques [14]. Across the configurable software systems of our sample, we have identified various change points. About one in each variant, on average, with an effect size between 4 and 6 percent for XZ and around 50 percent for LRZIP. The accumulated time series of the microbenchmarks for the remaining five subject systems exhibit larger numbers of change points with similar effect-size ranges, which can be attributed to the larger revision history (with the exception of ULTRAJSON, which has a comparably smaller revision history). That is, we can confirm the presence of substantial change points in all subject systems. Given this, it is important to note that the presence of change points poses an obstacle in modeling performance evolution since the time series data are segmented and possible trends (or other patterns) are superimposed by abrupt changes. We argue that missing knowledge of location and amplitude of these change points *hinders* prediction of performance evolution using traditional models, such as trend or seasonality extrapolation from the realm of auto-regressive models [14]. Since our intention is to efficiently estimate performance for arbitrary versions, we conclude that any such approach has not to only estimate performance, but especially search and pinpoint abrupt changes in performance evolution.

III. Active Performance Approximation

Our empirical findings on the ground truth data of performance-evolution measurements of six software systems in Section II suggests that software performance evolution exhibits abrupt changes that are hard to pinpoint without exhaustive measurements. While one could employ simple search heuristics, such as binary search, to locate abrupt changes, this modeling strategy is neither able to pinpoint possible performance patterns nor an arbitrary number of change points. To analyze performance time series in a way that incorporates possible patterns as well as an arbitrary number of abrupt changes, we propose an *adaptive* revision sampling approach to obtain accurate performance-estimation models. The workflow in Figure 3 outlines its key ingredients and procedure, which we will discuss next in detail.

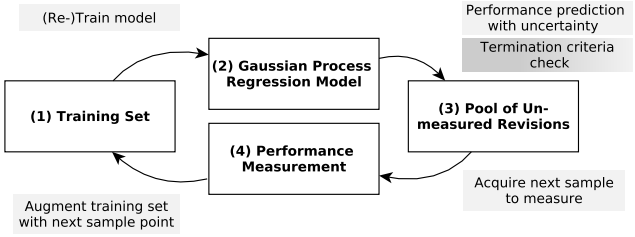


Figure 3: Workflow of adaptive revision sampling

A. Active Learning

A key feature in the workflow presented in Figure 3 is the iterative cycle of actions. In that sense, our approach follows a design strategy and family of machine learning

algorithms called *active learning*. Unlike ordinary machine learning algorithms, which are trained once and then evaluated, active learning trains repeatedly with gradually augmented training sets. This family of algorithms is intended to choose by itself which data are included in the training set. For instance, if an estimation model performs poorly on the influence of a specific parameter, it will suggest obtaining additional training samples with emphasis on variance of this very parameter. The motivation for this strategy is to minimize measurement effort by continuously querying specific data to be measured [18]. In our context, the pool of unseen observations are performance measurements. We opted for an active learning strategy since it minimizes the size of the training data set and guides the process of obtaining new observations. As performance measurement is typically an expensive task, we emphasize the incentive of minimizing costs.

B. Gaussian Process Regression

We use Gaussian Processes (GP) to learn a model that can accurately estimate performance for unobserved revisions (step 2 in Figure 3). The main motivation is that a GP makes estimations in the form of (multivariate) Gaussian distributions rather than scalar values. These distributions provide a measure of variance or, more importantly, a measure of confidence about our estimation accuracy for each revision of our time line [19]. That is, given a small confidence interval, the model is confident in its estimation, whereas a large confidence interval suggests to use the estimation with precaution and to refine the model. This facet of GPs is especially beneficial in the context of active learning, as we will explain in Section IV in more detail.

Formally, a GP assumes that, for a target function $f(x)$, each value $y = f(x)$ can be expressed as a Gaussian distribution with a mean function $\mu : \mathbb{X} \to \mathbb{R}$ and a covariance function $K : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$

$$f(x) \sim \mathcal{N}(\mu(x), K(x, x)). \tag{7}$$

The covariance function K, commonly called *kernel*, encodes the relationship between response values, depending on the *distance* of the input values. That is, for a pair of input values $(x_1, x_2) \in \mathbb{X}$, the covariance between $f(x_1)$ and $f(x_2)$ is defined as $K(x_1, x_2)$ [19]. Although we do not require any internal knowledge about the system studied to build a GP model, we can incorporate domain knowledge such as in the choice or construction of the used kernel function, which is an important success factor for our approach. For instance, for modeling time series with seasonality, a periodic kernel function can be used. For an overview on different kernel functions, we refer the interested reader to the literature [19, 20].

Once we have build a GP model for a set of given observations $f(x_1), f(x_2), \ldots$, we can predict the posterior distribution, that is, the mean m_*, and variance σ_*^2, for an unobserved data point x_* as

$$\begin{aligned} m_* &= \mu(x_*) + K(x_*, x) \cdot K(x, x)^{-1} \cdot (y - \mu(x)) \\ \sigma_*^2 &= K(x_*, x_*) - K(x_*, x) \cdot K(x, x)^{-1} \cdot K(x, x_*). \end{aligned} \tag{8}$$

With these two equations, we can obtain performance estimates of unseen revisions and obtain a measure of uncertainty about this estimate [21].

C. Active Data Selection

The key component of our active learning approach is the repeated augmentation of the training set. The procedure that is used to determine which observation to add to the training set (cf. step 3 in Figure 3) is defined by an *acquisition function* [18]. Generally, a data point is selected such that it is most likely to increase the prediction accuracy compared to the model built in the previous iteration. For different purposes, a variety of acquisition functions have been proposed [18], yet we follow an uncertainty-aware approach and let the data acquisition be guided by the prediction variance of the GP model directly. To minimize prediction uncertainty and, hopefully, increase prediction accuracy in the next iteration, we augment the training set with the data point x_{next} exhibiting the maximum prediction uncertainty σ^2 in the current iteration.

$$x_{\text{next}} = \arg\max_{x \in \mathbb{X}} \sigma_x^2 \tag{9}$$

D. Termination Criterion

The last piece of the puzzle is to decide in each iteration whether our cycle yields a model sufficiently confident in its estimations, or whether we need to continue refinement. Much like for data acquisition, we employ the prediction uncertainty. We consider a model to be sufficiently confident, if the uncertainty measure for every revision does not exceed a threshold specified by the user. This threshold t_{stop} can be estimated empirically, for instance, using the inherent measurement bias of the performance-measurement setup. In other words, we can measure a single revision a couple times and report the resulting variance as the threshold beyond which we cannot obtain more precise results. Thus, in each iteration, we evaluate whether the following formula holds or further refinement is necessary.

$$\max_{x \in \mathbb{X}} \sigma_x^2 \leq t_{\text{stop}} \tag{10}$$

E. Approach Summary

The workflow of our approach is as follows:

1) Initialize training set $\mathcal{T} = \{f(x_1), f(x_2), \ldots\}$ with performance measurements for a small number of revisions $\mathbb{X}_\mathcal{T} = \{x_1, x_2, \ldots\}$ and select a kernel function K.
2) Train a GP model $\mathcal{M}(T, K)$ and estimate the performance $\hat{f}(x) = \mathcal{N}(\mu_x, \sigma_x^2)$ for the remaining unobserved revisions $x \in \mathbb{X} \setminus \mathbb{X}_\mathcal{T}$.
   Check whether Equation 10 holds for the GP model. If so, *return* the prediction model $\mathcal{M}(T, K)$. If not, continue.
3) Determine x_{next} according to Equation 9 as $\arg\max_{x \in \mathbb{X} \setminus \mathbb{X}_\mathcal{T}} \sigma_x^2$.
4) Measure $f(x_{\text{next}})$, add x_{next} to $\mathbb{X}_\mathcal{T}$, $f(x_{\text{next}})$ to the training set $\mathcal{T}$, and go back to step 2.

We illustrate our approach in Figure 4, where we attempt to approximate a performance time series with about 1,000

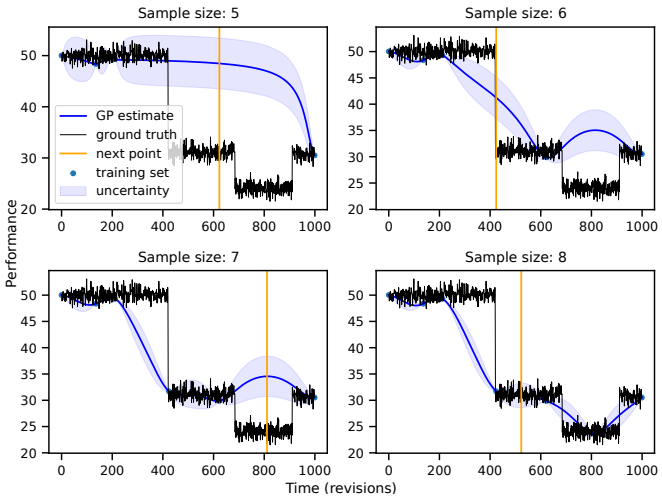


Figure 4: Four iterations of our adaptive approximation approach with 5 to 8 revisions as training sets. The black curve depicts the ground truth performance, the blue curve depicts the GP estimate with its respective uncertainty. The orange line highlights the revision which is next to be included in to the training set.

revisions. In the top left subfigure, we initially train a GP model with a training set of size five (i.e., 0.5 % of all revisions). In all four subfigures, the black line depicts the ground truth performance, the blue line the GP estimation, and the blue-shaded area the corresponding confidence interval. In the initial iteration, we are not satisfied with the maximum uncertainty and instead select the revision with the maximum uncertainty (depicted by an orange vertical line) to be included into the training set in the following iteration. We proceed in this manner with six and seven training samples. Finally, the GP estimation in the bottom right subfigure with eight training samples in total (i.e., 0.8 %) now clearly approaches the ground truth.

Let us summarize the benefits of our approach:

- Instead of finding a single change point as for bisect search, we can approximate the performance history of a software system as a whole.
- We can use different kernel functions at different time intervals to account for different time series properties (as found in Section II) in an appropriate way.

IV. Evaluation

Next, we evaluate our approach with respect to efficacy and efficiency.

A. Subject Systems

1) Software System Selection: We have selected six real-world, actively maintained software systems from different domains, including file compression (XZ and LRZIP), JSON parsing (ULTRAJSON), image manipulation (PILLOW), and scientific computing (NUMPY and SCIPY). The selection along with project metrics is presented in Table I. The subject

Table I: Meta-data for our subject systems. We report the number of different variants and methods per subject respectively. In total, we consider 8,307,640 revisions in our analysis.

Software system	Methods or variants	#Revs.	SLOC
XZ	47 variants	1,151	85,036
LRZIP	71 variants	850	45,049
PILLOW	59 methods	7,928	57,411
ULTRAJSON	4 methods	327	327,948
NUMPY	204 methods	19,914	271,976
SCIPY	174 methods	21,046	338,631

SLOC: Source lines of code were collected using the tool `cloc`.

systems can be divided into two categories: XZ and LRZIP are software systems that can be configured at load time to obtain functionally differing variants. We analyze multiple variants because performance bugs are often configuration-related and emerge only under certain configurations [17]. The second category includes Python libraries for various tasks. Here, we obtained performance history data for individual methods as these subjects represent libraries that are embedded in other applications. Considering both stand-alone systems and libraries, we obtain a more general picture about performance evolution.

2) Deriving Performance Histories: For the two load-time configurable software systems, we assess performance for 47 and 71 different configurations and, at least, 5 repeated runs, resulting in more than 500,000 measurements. The selection of system variants follows known feature-wise sampling for binary options [22] and Plackett-Burman sampling for numeric options [23]. Each configuration results in a different execution for which performance can differ and evolve independently.

We selected benchmarks such that each represents a real-world execution scenario. For XZ and LRZIP, we use the Silesia compression corpus[4], which contains about 200 MB of files of different types. For the Python libraries (except ULTRAJSON), we reused performance benchmarks at method-level provided and used by the respective software projects. These microbenchmarks are used for optimizing the respective libraries, so they reflect performance behavior relevant for maintenance. Based on these microbenchmarks, we derive a number of performance histories by considering each microbenchmark as a variant of the whole software system. This way, we consider macro- and microbenchmarks in our evaluation avoiding bias (e.g., the controversy of whether microbenchmarks are representative for the performance of a whole system).

Except PILLOW, the Python projects measure, track, and publish performance measurements for their microbenchmark using the tool airspeed velocity[5]. We extracted the performance data for the three Python libraries from their respective GitHub

[4]The Silesia corpus can be found at http://mattmahoney.net/dc/silesia.html.
[5]https://github.com/airspeed-velocity/asv/

repositories[6], whereas for PILLOW, ULTRAJSON, XZ, and LRZIP, we conducted the performance measurements on our own. For ULTRAJSON, we assembled four tasks on our own, which consist of parsing and serializing a small (size: 617KB) and a large (size: 7.4MB) JSON file, respectively, as this is the main purpose and most performance relevant task of a JSON library.

All measurements were conducted for all revisions on different commercial-off-the-shelf compute clusters. However, each software system was assessed with an identical hard- and software setup. XZ and LRZIP have been assessed on machines with a Core 2 Quad CPU (2.83 GHz)/16 GB RAM, PILLOW on machines with an i7 CPU (3.40 GHz)/32 GB RAM, and ULTRAJSON on machines with an Xeon CPU (3.00 GHz)/64 GB RAM. Per revision, we repeated each experiment five times and reported the median measurement to mitigate measurement bias. The coefficient of variation reported on all machines was well below ten percent.

B. Research Questions

Our approach can be customized via the choice of the kernel function. Different kernel functions result in possibly different shapes of the estimation of the GP model. For the use of GP regression models, there exists a wide variety of kernel functions, which can be used, or composed to obtain more complex ones [19]. For our evaluation, we selected the five kernel functions presented in Table II, which are commonly provided by (or can be added to) GP libraries, such as SCIKIT-LEARN or GPY for Python. Four of the five kernels are parametric and stationary, since they represent functions of the distance between the input vectors, whereas the Brownian kernel is inspired by a stochastic process, whose growth is normally distributed, so that it is not stationary [19]. This selection is by no means exhaustive, but we selected kernels that are widely used in practice. In our implementation, we use the GPY library[7] for Gaussian Processes in Python and the respective kernels. The kernel hyperparameters were not explicitly tuned per subject system, yet the default optimizer selects the optimal fit of, at most, 1,000 iterations. In what follows, we motivate our research questions and describe how we operationalized them.

RQ1: Which choice of a kernel function performs best at estimating performance histories?

To start our evaluation, for our subject system corpus, we compare two indicators for a setup of the five different kernel functions: maximum uncertainty and performance estimation error. The first indicator serves as an upper bound for model confidence, whereas the *mean absolute percentage error (MAPE)* is a measure of how well an estimation fits the ground truth observations on a global scale. As these two indicators change with every iteration, and, due to the lack of space, we have decided to incorporate these dynamics in our visualization and present it in an animation provided online.

[6]The measurements can be obtained for: NUMPY at https://pv.github.io/numpy-bench/; SCIPY at https://pv.github.io/scipy-bench/.

[7]https://gpy.readthedocs.io/en/deploy/index.html

In addition, we evaluate our results on both macrobenchmarks and microbenchmarks. The four Python libraries represent microbenchmarks as the performance has been measured using the method execution time. For macrobenchmarks, we use (a) the whole systems of XZ and LRZIP including their variants and (b) the sum of method execution times of all methods of a respective library. The rationale behind this is to obtain an overview of the performance behavior of a whole library one would need to create a benchmark executing all methods. Hence, we interpret the performance value of a revision as the sum of all performance values of all method execution times of this revision.

RQ2: Does active revision sampling improve modeling performance evolution histories with GP models?

To evaluate whether selecting the next sample (i.e., revision) based on the uncertainty of the performance estimate is reasonable, we compare active revision sampling against random sampling. In particular, we compare how many iterations are required to obtain accurate estimation models and whether, with each iteration, the estimation error converges robustly. For this comparison, we evaluate on the best-performing kernel of RQ1 on the micro- and macrobenchmarks.

RQ3: Can we estimate global change points in a performance evolution history?

Reporting MAPE for a GP estimation provides a good picture of how accurate global effects, such as trends, have been modeled. However, even a small average error could mean that abrupt changes might not be spotted in the model, as the high error at change points can be diluted by small error of the vast number of remaining revisions. Since change points represent interesting events in the performance history of a software system, we quantify whether our model is capable of detecting them at the macrobenchmark level.

To assess whether we can derive the locations of global change points, we apply two different off-line change point detection algorithms to the ground truth as well as our estimations. In this experiment, the independent variables are the kernel, size of the sample set, and, in addition, the choice of the change point algorithm. We decided not to reuse the CUSUM algorithm from Section II as it is sensitive to its parametrization. This was useful for exploring the presence of change points of different magnitudes, yet in this context, we focus on global change points. Instead, we employ a top-down binary segmentation strategy as well as a bottom-up merge strategy. The first algorithm recursively segments the given time series at the point with the highest variation, whereas the second algorithm merges smaller segments to larger segments as long as the constituent segments exhibit little variation. For both algortihms, we refer to respective implementations and parameterization from the Python library `ruptures`[8], where a more detailed description of both algorithms can be found.

[8]https://github.com/deepcharles/ruptures/

Table II: Selection of five widely used kernel functions. The hyperparameters ϑ describes the lengthscale of the kernel function, σ^2 describes the variance.

Name	Definition
Radial Basis Function (RBF)	$k(x,y) = \sigma^2 \exp\left(-\frac{\lvert x-y\rvert^2}{2\vartheta^2}\right)$
Rational Quadratic Kernel (RQF)	$k(x,y) = \sigma^2 \left(1 - \frac{\lvert x-y\rvert^2}{2\alpha\vartheta}\right)^{\alpha}$
Brownian Kernel (BK)	$k(x,y) = \sigma^2 \min(x,y)$
Matérn Kernel (MK3/2)	$k(x,y) = \sigma^2 \left(1 + \frac{\sqrt{3}\lvert x-y\rvert}{\theta}\right) \exp\left(\frac{-\sqrt{3}\lvert x-y\rvert}{\theta}\right)$
Matérn Kernel (MK5/2)	$k(x,y) = \sigma^2 \left(1 + \frac{\sqrt{5}\lvert x-y\rvert}{\theta} + \frac{5\lvert x-y\rvert^2}{3\ \theta^2}\right) \exp\left(\frac{-\sqrt{5}\lvert x-y\rvert}{\theta}\right)$

C. Results

1) Kernel Selection Efficacy: In Table III, we report for different sample sizes both the uncertainty of the different kernels as well as the respective MAPE. For the configurable systems, we average the uncertainties and errors rates over all variants; for the libraries, we average over all methods. We highlight for each sample size the best kernel with respect to MAPE in green.

As can be seen in Table III, the Brownian Kernel (BK) achieves the lowest MAPE for nearly all subjects. Only for ULTAJASON, we observe that the Radial Basis Function (RBF) and Rational Quadratic Kernel (RQF) perform slightly better. Nevertheless, for most systems, the difference is substantial, partially resulting in a one or to two orders of magnitude lower MAPE. Interestingly, macrobenchmarks seem to be easier to model than microbenchmarks, which might be because macrobenchmarks do not fluctuate in the performance behavior over the history so much. The rationale might be that changes at method level will have a more severe effect on execution time at microbenchmark level than for the whole system.

A further observation is that prediction accuracy increases when the sample grows. Although this is plausible, we argue that suboptimally selected samples might lead to false approximations of the GP model (e.g., in the case of SCIPY when comparing 3 % of revisions vs. 5% of revisions). Since BK is vastly superior, we use this kernel for the next research question.

Summary: The Brownian Kernel (BK) produces the most accurate performance history estimations with single digit prediction error rates, except for NUMPY (MAPE: 26.074).

2) Revision Sampling Efficiency: To learn whether actively selecting samples improves over random sampling, we compare the MAPE of the BK learned with active sampling against random sampling of revisions. We repeated random sampling ten times and use the mean. Table IV shows the corresponding MAPE values, highlighting better values in green. The presented numbers are aggregated MAPEs over multiple variants and methods (compare Table I). To verify whether active revision sampling is more efficient than random sampling, we conducted a significance test.

First, we need to calculate the weighted mean for XZ and LRZIP, because these values represent mean values from multiple variants (cf. Table I). Then, we use a Shapiro-Wilk normality test to determine which kind of significance test is applicable. Since the data are not normally distributed, we use a one-sided non-parametric paired Wilcoxon signed-rank test. The p-value is 0.077, which is not significant at α-level 0.95. In other words, there seems to be a favor toward active sampling, but the difference is not significant.

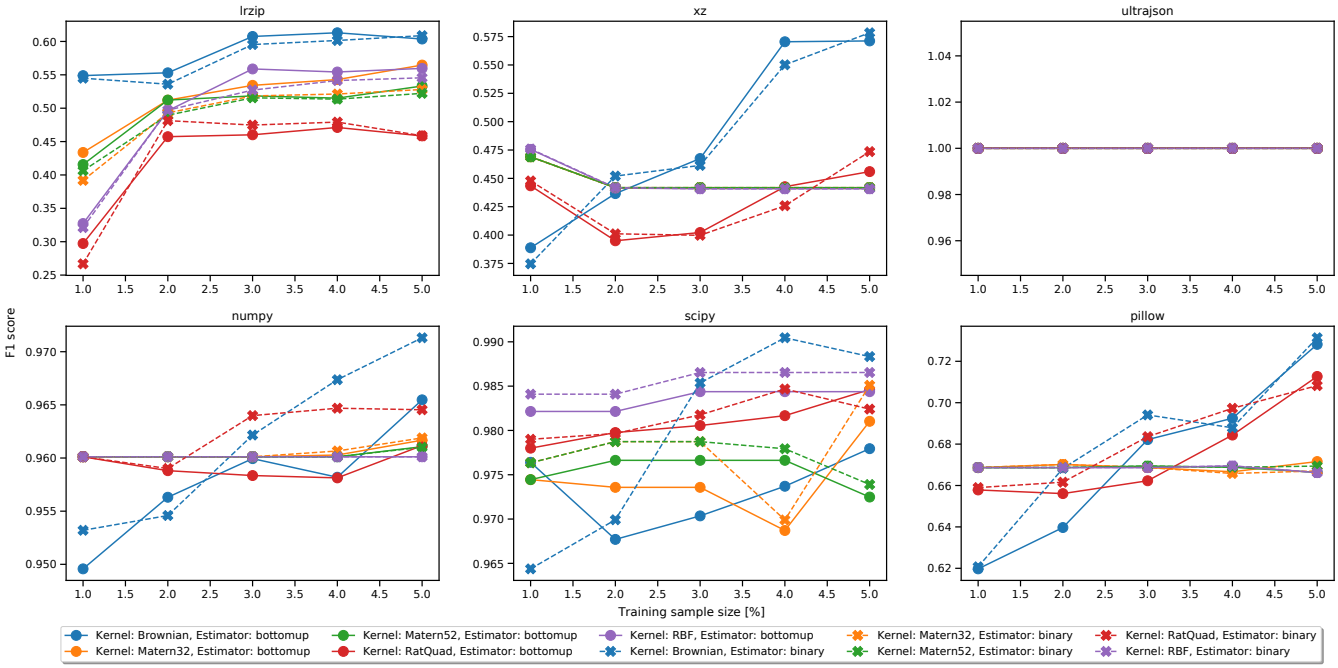
Summary: Active sampling performed slightly better than random sampling, but the effect is not statistically significant (p-value: 0.077).

3) Global Change Point Estimation: Although the estimations with a BK kernel yielded the most accurate estimation, in this analysis, we consider estimations with all kernels since the kernels exhibit different properties[9] which can influence the shape of the GP estimation. For all combinations of algorithms and kernels, we compare the time-series' ground truth against GP estimations trained with 1 %, 3 %, and 5 % of all revisions. We consider a change point a true positive, if a change point inferred from a GP estimation matches a change point reported in the corresponding ground truth within a range of $\pm$ 5 revisions.

In Figure 5 we report the F1 score, a measure that summarizes precision and recall of a binary classificator for all subject systems, kernels, training set sizes, and change point detection algorithms. Following the results from RQ1, the accuracy of pinpointing global change points for the majority of experiment configurations is greater for larger training sets. The most accurate estimations of change point locations are achieved again using the BK. ULTRAJSON exhibits only global few change points, resulting in high accuracy. While the BK outperformed the other kernels for RQ1, the effect here is less pronounced.

Summary: From our GP estimates, we are able to accurately derive the locations of global change points.

[9]Unlike the other four kernels, the Brownian kernel is non-stationary, i.e. corresponding estimated do not exhibit mean-reverting dynamics. Moreover, the two variants of the Matérn kernel are only differentiable one or twice, respectively



The F1 score is the harmonic mean of precision P and recall R. These are defined as P = $TP/(TP * FP)$ and R = $TP/(TP + FN)$ respectively where TP = true positives, FP = false positives, and FN = false negatives.

Figure 5: Comparing change points reported by two different algorithms (binary segmentation and bottom-up merging) from the observed ground thruth and from GP estimations for five different kernels and varying training set sizes. The training sample size refers to the percentage of total commits per software system.

V. Discussion

A. Kernel Selection Efficacy

Among our selection of kernel functions, estimations using the Brownian kernel (BK) outperformed the others in terms of the lowest error rate in most scenarios, only exceeding a MAPE of 10 % for NUMPY. The error rates reported for the BK across our selection of subject system range from below 1 % to 26 %, and the highest among them are reported for microbenchmarks. This discrepancy suggests that macrobenchmarks are easier to learn than microbenchmarks. The estimations obtained using the BK kernel show the applicability of this kernel to obtain *accurate* estimation models using GP regression with relatively small training sample sizes. This high accuracy was achieved by measuring only a tiny portion of the performance history (i.e., 1 % to 5 % of all revisions).

We attribute the accuracy of estimates using BK partly to their general shape. Estimations obtained using BK resemble a piece-wise linear function. Considering a segment of observations between two revisions t_1 and t_2, with $t_1 < t_2$, the covariance for any pair of revisions between t_1 and a point t' of the segment, where $t_1 < t' < t_2$, is constant and equal to t_1 since $K_{\text{BK}}(x, y) = \min(x, y)$. Therefore, each segment between observations is estimated with a linear function. Given the presence of change points, the estimates obtained using BK provide a good fit of the observed performance histories that are segmented by change points since those can be approximated by narrow segments with a high slope.

B. Revision Sampling Efficiency

The comparison of models trained with our active sampling strategy with randomly trained models has shown only a slight, yet not statistically significant improvement in terms of reported error in different scenarios. Random sampling is usually be the best sampling strategy [24], but we provide a more systematic approach, which works slightly better. The work of Roberts et al. [21] on using GP for time series data does not compare active data selection with a random baseline, so our findings complement the understanding of the combination of active learning and GP regression. In addition, since we let GP's uncertainty guide the acquisition of new observations, the training sample is uniformly distributed rather than being concentrated around change points. Therefore, a balanced acquisition strategy might be a better extension, because it explores the revisions guided by uncertainty and exploits the gradient of the GP estimate to accurately approach and pinpoint change points.

C. Global Change Point Estimation

We were able to extract most change points from the estimated models, which we validated against an established approach to locate change points in time series. We acknowledge that our change point analysis is more exploratory nature as this is not our main objective (we thrive for accurate modeling of whole performance histories). Overall, our results suggest that, using adaptive revision sampling, extensive performance analyses over large performance histories with just a few

Table III: For each kernel and subject system, $\overline{U}$ denotes the mean uncertainty of a kernel over all revisions and MAPE denotes the mean average percentage error over all revisions. 1 %, 3 %, and 5 % are the sizes of the sample set with respect to the whole performance history. For all subjects, we report the average over all variants and methods respectively.

		Kernels	1 %		3 %		5 %	
			$\overline{U}$	MAPE	$\overline{U}$	MAPE	$\overline{U}$	MAPE
Macrobenchmarks	XZ	RBF	0.036	2.007	0.015	1.998	0.011	1.991
		RQF	1.991	1.295	0.011	0.624	0.006	0.487
		BK	0.021	0.632	0.006	0.352	0.003	0.308
		MK3/2	0.036	2.007	0.016	1.999	0.04	1.385
		MK5/2	0.036	2.008	0.016	1.997	0.076	1.881
	LRZIP	RBF	7.304	41.954	10.725	10.725	5.743	17.903
		RQF	5.245	16.253	3.675	10.099	3.288	9.502
		BK	1.634	4.008	1.053	3.049	0.565	2.575
		MK3/2	8.772	26.338	2.183	12.285	2.326	9.882
		MK5/2	7.988	26.778	5.032	14.712	4.068	11.469
	PILLOW	RBF	1.679	77.043	0.574	77.028	0.645	76.302
		RQF	0.083	0.93	0.115	0.396	0.159	0.476
		BK	0.301	0.831	0.205	0.32	0.066	0.299
		MK3/2	1.679	77.042	0.935	76.537	0.028	0.547
		MK5/2	1.679	77.04	0.837	76.663	0.036	0.747
	ULTRAJSON	RBF	0.008	1.723	0.004	1.736	0.003	1.764
		RQF	0.004	1.789	0.003	1.806	0.003	1.805
		BK	0.002	2.58	0.003	1.818	0.003	1.818
		MK3/2	0.005	1.762	0.003	1.779	0.003	1.774
		MK5/2	0.005	1.772	0.002	1.789	0.004	1.795
	NUMPY	RBF	0.0	7.36	0.0	7.36	0.0	7.36
		RQF	0.001	7.192	0.0	0.361	0.0	0.309
		BK	0.0	0.644	0.0	0.591	0.0	0.275
		MK3/2	0.0	7.36	0.0	7.36	0.0	7.36
		MK5/2	0.0	7.36	0.0	7.36	0.0	7.36
	SCIPY	RBF	0.0	3.102	0.0	3.102	0.0	3.091
		RQF	0.003	3.065	0.0	0.041	0.0	0.029
		BK	0.0	0.087	0.0	0.032	0.0	0.021
		MK3/2	0.0	3.102	0.0	3.083	0.0	3.057
		MK5/2	0.0	3.102	0.0	3.096	0.0	3.07
Microbenchmarks	PILLOW	RBF	0.057	77.289	0.02	77.281	0.029	77.267
		RQF	0.11	35.902	0.188	23.156	0.133	18.86
		BK	0.017	1.238	0.011	0.788	0.007	0.007
		MK3/2	0.007	77.279	0.043	77.261	0.067	76.151
		MK5/2	0.057	77.282	0.036	77.268	0.056	76.153
	ULTRAJASON	RBF	0.007	5.717	0.004	5.718	0.003	5.601
		RQF	0.005	5.683	0.005	5.008	0.003	3.364
		BK	0.005	5.788	0.005	5.788	0.005	5.788
		MK3/2	0.004	5.69	0.002	5.623	0.004	5.42
		MK5/2	0.005	5.721	0.003	5.712	0.003	5.492
	NUMPY	RBF	0.002	110.66	0.018	107.699	0.024	35.903
		RQF	0.026	80.038	0.026	29.095	0.026	27.677
		BK	0.005	26.074	0.002	25.82	0.001	25.138
		MK3/2	0.015	109.642	0.021	47.049	0.027	45.75
		MK5/2	0.01	109.961	0.022	48.199	0.025	43.986
	SCIPY	RBF	0.185	136.563	0.266	140.011	0.366	64.83
		RQF	0.062	128.77	0.059	121.112	0.065	144.937
		BK	0.015	8.115	0.007	8.04	0.007	8.928
		MK3/2	0.269	135.829	0.323	139.226	0.113	159.222
		MK5/2	0.282	135.747	0.084	138.174	0.088	159.689

measurements is possible. To the best of our knowledge, this could not have been done before. Our approach enables a wide range of further applications, such as tracking accumulated technical debt and performance evolution.

Table IV: For the Brownian kernel, we report the mean average percentage error over all revisions and compare actively (act.) and randomly (rand.) sampled setups.

		1%		3%		5%	
		act.	rand.	act.	rand.	act.	rand.
Macrobenchmarks	XZ	0.632	0.502	0.352	0.355	0.308	0.302
	LRZIP	4.008	5.731	3.049	3.370	2.575	3.149
	PILLOW	0.831	1.023	0.32	0.631	0.299	0.344
	ULTRAJSON	2.58	1.836	1.818	1.732	1.818	1.693
	NUMPY	0.644	0.806	0.591	0.479	0.275	0.323
	SCIPY	0.087	0.099	0.032	0.046	0.021	0.034

D. Threats to Validity

Threats to *internal validity* include measurement noise that may leak into when learning performance models. We mitigate this threat by using the average over five repeated runs. We quantified the deviation of measurements and made sure with additional measurements that the standard deviation is below 10 % with respect to the mean performance of the repeated runs. The data sets we obtained from the library developers have been repeated 40 times for NUMPY and SCIPY and for ULTRAJSON and PILLOW about thousand times on multiple machines. Hence, we are confident that our raw data are robust against measurement outliers. Moreover, our learning and estimation pipeline could contain implementation errors. We carefully reviewed all intermediate results to mitigate this threat. Also, we make all scripts available at our repository.

Regarding *external validity*, we cannot claim that our approach works for all kinds of software systems. With our evaluation, we selected systems from different domains and made sure that we consider both macro- and microbenchmarks. Moreover, we selected popular real-world systems to test our approach on realistic data. Nevertheless, there are still types of system that we have not considered so far: We did not include client-server systems nor other performance metrics than response time.

VI. Related Work

A. Performance Anomaly Detection

The question of what is a relevant performance change is crucial for both applications in industry and academia. We revisit a selection of approaches to identify or pinpoint performance anomalies.

Most prior work focuses on a single performance indicator, such as execution time. Nguyen et al. [13, 25, 26], Malik et al. [27], and Lee et al. [11] use statistical process control (SPC) charts, a technique derived from quality management, to detect possible performance regressions. SPC charts dynamically provide a simple definition of thresholds that, when exceeded, indicate quality regression.

A more rigorous statistical approach is to formalize performance regression as a hypothesis testing problem applying statistical tests to ask whether a revision's performance is significantly different from a preceeding one, as proposed by Reichelt et al. [7]. Their work assesses a set of different statistical tests regarding the applicability to this problem. Heger et al. employ continuous measurement of unit tests as performance benchmarks [12]. They use analysis of variance (ANOVA) as an rigorous alternative [28] to statistical testing and compare performance distributions of different revisions. A more sophisticated notion of performance is to consider complex indicators that are aggregated from single indicator. Foo et al. use the correlation among performance indicators [8, 29] and Malik et al. use performance signatures to aggregate multiple performance indicators, for instance by first performing dimensionality reduction on a vector of performance indicators with PCA and tracking the evolution of PCA weights with SPC charts [27]. Our approach incorporates only single metrics, but aggregated performance indicators provide a possible extension. Cito et al. apply on-line change point detection to detect performance degradation in Web applications and pinpoint possible root causes [30]. While our work uses fractions of a batch of performance observations, their strategy uses a stream of continuous performance measurements at runtime. The application of (off-line) change point detection is a promising further research direction to infer performance anomalies in performance evolution histories.

B. Gaussian Process for Time Series Analysis

The application of Gaussian Process models to learn real-world time series data is extensively, but not exclusively discussed by Roberts et al. [21]. They employ a similar active learning approach of uncertainty-guided active sampling to minimize observation effort. Their proposed algorithm not only includes new actively sampled data points with each iteration, but, in contrast to our approach, has the opportunity to exclude data points for which the model is highly confident. Garnett et al. and Osbourne et al. approach the problem of learning time series data with change points by proposing a kernel, which includes the location of a single change point as a hyperparameter [31, 32]. We cannot resort to such kernels since, for a performance evolution history, we do not know if, and, if so, how many change points there are.

C. Performance of Configurable Software Systems

Another area of research aims at learning performance models on configurable software systems. The idea is to model configuration options as features in a machine-learning setting and learn a corresponding prediction function. There are different learning techniques, such as Classification and Regression Trees [33–35], multi-variable regression [22], and deep neural networks [36]. Although we consider also configurable software systems, we do not model features, but revisions and predict performance not for variants, but revisions, which is an orthogonal task. Nevertheless, there are some ideas related to ours. Siegmund et al. propose an active learning approach based on heuristics to select variants that exhibit certain interactions of configuration options [37]. Nair et al. use also the idea of GP models' acquisition function to sample the configuration first, which point to the fasted configuration [38]. Oh et al. use active sampling to find an optimal configuration without building a performance model [39]. They do so by recursively shrinking the search space towards the fastest configurations. Although these techniques are orthogonal (learning and sampling in space) to ours (in time), we see potential in combining both toward a combined learning and sampling approach.

VII. Conclusion

We propose an approach to accurately estimate the performance evolution history for software systems using GP regression with an active sampling strategy. Guided by the uncertainty provided along estimations, we iteratively expand the training set until a desired model confidence is reached. In an exploratory analysis, we confirm the presence of abrupt and substantial performance changes for six real-world software systems. We investigate the choice of five different kernels for learning GP models, compare our active sampling against a random sampling baseline, and estimate the locations of abrupt changes for six different software systems.

Our work has confirmed the presence of global change points, and we were to able to pinpoint those with high accuracy. As global change points contribute to performance evolution, we believe this aspect of performance evolution in particular deserves further attention. It is an avenue of further research to investigate the causes of change points and triangulate inferred change point locations with insights from development and documentation artifacts. In addition, the search for change points in performance evolution time series can be disassociated from the framework of Gaussian Processes provided in this work.

VIII. Acknoledgements

Apel's work has been supported by the Deutsche Forschungsgemeinschaft (DFG) under the contract AP 206/11-1. Siegmund's work has been supported by the DFG under the contracts SI 2171/2 and SI 2171/3-1.

References

[1] S. Egger, P. Reichl, T. Hoßfeld, and R. Schatz, "Time is bandwidth? Narrowing the gap between subjective time perception and Quality of Experience," in *Proceedings of the IEEE International Conference on Communications (ICC)*. IEEE, Jun. 2012, pp. 1325–1330.

[2] F. F.-H. Nah, "A Study on Tolerable Waiting Time: How Long Are Web Users Willing to Wait?" *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, May 2004.

[3] I. Molyneaux, *The Art of Application Performance Testing*, 2nd ed., ser. Theory in Practice. Beijing: O'Reilly, 2015.

[4] H. P. Breivold, I. Crnkovic, and M. Larsson, "A Systematic Review of Software Architecture Evolution Research,"

Journal of Information and Software Technology, vol. 54, no. 1, pp. 16–40, Jan. 2012.

[5] J. Guo, J. White, G. Wang, J. Li, and Y. Wang, "A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2208–2221, Dec. 2011.

[6] F. I. Vokolos and E. J. Weyuker, "Performance Testing of Software Systems," in *Proceedings of the First International Workshop on Software and Performance (WOSP)*. ACM, 1998, pp. 80–87.

[7] D. G. Reichelt and S. Kühne, "How to Detect Performance Changes in Software History: Performance Analysis of Software System Versions," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2018, pp. 183–188.

[8] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "Mining Performance Regression Testing Repositories for Automated Performance Analysis," in *Proceedings of the International Conference on Quality Software (QSIC)*. IEEE, 2010, pp. 32–41.

[9] F. Pinto, U. Kulesza, L. Silva, and E. Guerra, "Automating the Assessment of the Performance Quality Attribute for Evolving Software Systems: An Exploratory Study," in *Proceedings of the Hawaii International Conference System Sciences (HICSS)*. IEEE, 2015, pp. 5144–5153.

[10] L. Bulej, T. Kalibera, and P. Tma, "Repeated Results Analysis for Middleware Regression Benchmarking," *Performance Evaluation*, vol. 60, no. 1-4, pp. 345–358, May 2005.

[11] D. Lee, S. K. Cha, and A. H. Lee, "A Performance Anomaly Detection and Analysis Framework for DBMS Development," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 24, no. 8, pp. 1345–1360, Aug. 2012.

[12] C. Heger, J. Happe, and R. Farahbod, "Automated Root Cause Isolation of Performance Regressions During Software Development," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2013, pp. 27–38.

[13] T. H. D. Nguyen, M. Nagappan, A. E. Hassan, M. Nasser, and P. Flora, "An Industrial Case Study of Automatically Identifying Performance Regression-Causes," in *Proceedings of the Conference on Mining Software Repositories (MSR)*. ACM, 2014, pp. 232–241.

[14] W. A. Fuller, *Introduction to Statistical Time Series*, 2nd ed., ser. Wiley Series in Probability and Statistics. Wiley, 1996.

[15] E. S. Page, "Continuous Inspection Schemes," *Biometrika*, vol. 41, no. 1/2, pp. pages 100–115, Jun. 1954.

[16] F. Gustafsson, *Adaptive Filtering and Change Detection*. John Wiley & Sons, Ltd, Oct. 2001.

[17] X. Han and T. Yu, "An Empirical Study on Performance Bugs for Highly Configurable Software Systems," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ACM, 2016, pp. 1–10.

[18] B. Settles, "Active Learning Literature Survey," University of Wisconsin, Madison, Tech. Rep. 1648, 2010.

[19] C. K. Williams and C. E. Rasmussen, *Gaussian processes for machine learning*. MIT Press Cambridge, 2006, vol. 2, no. 3.

[20] D. Duvenaud, "Automatic Model Construction with Gaussian Processes," Doctoral Thesis, University of Cambridge, Nov. 2014.

[21] S. Roberts, M. Osborne, M. Ebden, S. Reece, N. Gibson, and S. Aigrain, "Gaussian Processes for time-series modelling," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1984, p. 20110550, 2013.

[22] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-Influence Models for Highly Configurable Systems," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 284–294.

[23] J. Burman and R. Plackett, "The Design of Optimum Multifactorial Experiments," *Biometrika*, vol. 33, pp. 305–325, Jan. 1946.

[24] C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel, "Distance-based Sampling of Software Configuration Spaces," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1084–1094.

[25] T. H. Nguyen, "Using Control Charts for Detecting and Understanding Performance Regressions in Large Software," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*. Montreal, QC, Canada: IEEE, Apr. 2012, pp. 491–494.

[26] T. H. Nguyen, B. Adams, Z. M. Jiang, A. E. Hassan, M. Nasser, and P. Flora, "Automated Detection of Performance Regressions Using Statistical Process Control Techniques," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*. ACM, 2012, p. 299.

[27] H. Malik, H. Hemmati, and A. E. Hassan, "Automatic Detection of Performance Deviations in the Load Testing of Large Scale Systems," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 1012–1021.

[28] A. Georges, D. Buytaert, and L. Eeckhout, "Statistically Rigorous Java Performance Evaluation," in *Proceedings of the Conference on Object-oriented Programming Systems and Applications (OOPLSA)*. ACM, 2007, pp. 57–76.

[29] K. C. Foo, Z. M. Jiang, B. Adams, A. E. Hassan, Y. Zou, and P. Flora, "An Industrial Case Study on the Automated Detection of Performance Regressions in Heterogeneous Environments," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, May 2015, pp. 159–168.

[30] J. Cito, D. Suljoti, P. Leitner, and S. Dustdar, "Identifying Root Causes of Web Performance Degradation Using Changepoint Analysis," in *Proceedings of the International Conference on Web Engineering (ICWE)*. Springer, 2014, pp. 181–199.

[31] R. Garnett, M. A. Osborne, S. Reece, A. Rogers, and S. J. Roberts, "Sequential Bayesian Prediction in the Presence of Changepoints and Faults," *The Computer Journal*, vol. 53, no. 9, pp. 1430–1446, Nov. 2010.

[32] M. A. Osborne, S. J. Roberts, A. Rogers, and N. R. Jennings, "Real-time Information Processing of Environmental Sensor Network Data Using Bayesian Gaussian Processes," *ACM Transactions on Sensor Networks (TOSN)*, vol. 9, no. 1, pp. 1–32, Nov. 2012.

[33] J. Guo, K. Czarnecki, S. Apely, N. Siegmund, and A. Wasowski, "Variability-aware Performance Prediction: A Statistical Learning Approach," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 301–311.

[34] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-Efficient Sampling for Performance Prediction of Configurable Systems (T)," in *Proceedings of the International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 342–352.

[35] V. Nair, T. Menzies, N. Siegmund, and S. Apel, "Using Bad Learners to Find Good Configurations," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, pp. 257–267.

[36] H. Ha and H. Zhang, "DeepPerf: Performance Prediction for Configurable Software with Deep Sparse Neural Network," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 1095–1106.

[37] N. Siegmund, S. S. Kolesnikov, C. Kastner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake, "Predicting Performance via Automated Feature-Interaction Detection," in *Proceedings of the International Conference on Software Engineering (ICSE)*. IEEE, Jun. 2012, pp. 167–177.

[38] V. Nair, Z. Yu, T. Menzies, N. Siegmund, and S. Apel, "Finding Faster Configurations using FLASH," *IEEE Transactions on Software Engineering (TSE)*, 2018, online First.

[39] J. Oh, D. Batory, M. Myers, and N. Siegmund, "Finding Near-Optimal Configurations in Product Lines by Random Sampling," in *Proceedings of the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM Press, 2017, pp. 61–71.