Adapting a System with Noisy Outputs with Statistical Guarantees

Ilias Gerostathopoulos
Technical University Munich
Boltzmannstr. 3, 85748 Garching
Germany
ilias.gerostathopoulos@tum.de

Christian Prehofer
Technical University Munich & fortiss GmbH
Boltzmannstr. 3, 85748 Garching
Germany
christian.prehofer@tum.de

Tomas Bures
Charles University in Prague
Malostranske namesti 25,11800 Prague
Czech Republic
bures@d3s.mff.cuni.cz

ABSTRACT

Many complex systems are intrinsically stochastic in their behavior which complicates their control and optimization. Current self-adaptation and self-optimization approaches are not tailored to systems that have (i) complex internal behavior that is unrealistic to model explicitly, (ii) noisy outputs, (iii) high cost of bad adaptation decisions, i.e. systems that are both hard *and* risky to adapt at runtime. In response, we propose to model the system to be adapted as black box and apply state-of-the-art optimization techniques combined with statistical guarantees. Our main contribution is a framework that combines runtime optimization with guarantees obtained from statistical testing and with a method for handling cost of bad adaptation decisions. We evaluate the feasibility of our approach by applying it on an existing traffic navigation self-adaptation exemplar.

CCS CONCEPTS

• **Software and its engineering ~ Designing software**
• Theory of computation ~ Mathematical optimization

KEYWORDS

self-adaptation, statistical guarantees, experimentation cost

1 INTRODUCTION

Modern software-intensive systems become more and more complex to model, develop, test, and optimize. The major source of complexity is the uncertainty that creeps in both at development time, in the form of unknown requirements and uncertain environment assumptions, and at runtime, in the form of runtime conditions that have not been fully anticipated and thus specifically designed for. Uncertainty handling is the main motivation behind equipping such systems with self-adaptivity, i.e. the ability of a system to observe its own functioning and adapt itself at runtime in order to achieve better performance, recover from faults, or strengthen itself in face of safety or security threats.

Intuitively, certain systems are both *harder* and *riskier* to adapt at runtime than others. With current techniques, adapting a system with predictable behavior and well-known boundaries, such as a web application, is easier than an open-ended system with largely stochastic behavior, such as an emergency coordination system. Adaptation risk is orthogonal to adaptation difficultly and pertains to the magnitude of effect and the frequency of bad runtime adaptation decisions: when adapting a video encoder at runtime, a bad decision may influence the adaptation speed or the quality of the encoding; when adapting a live system being used by a large number of users, such as a traffic navigation system, bad decisions may incur serious cost in terms of business, e.g. dissatisfied customers.

In this paper, we focus on a class of systems which have (i) complex behavior that is unrealistic to model explicitly, (ii) noisy outputs, (iii) high cost of bad adaptation decisions; they are, therefore, both hard *and* risky to adapt at runtime. We assume that the system-to-be-adapted is abstracted as a black-box model of the essential input and output parameters. Noisy outputs refer to observations of system metrics that can be observed and used for self-adaptation but possess high variance. Cost refers to the negative impact of bad adaptation decisions on the system-to-be-adapted.

We aim to provide a framework for performing self-optimization in this challenging class of systems. We are focusing on optimizing application-level performance goals. The main questions we explore examine (i) how to build system models out of observations of noisy system outputs; (ii) how to re-use these models to optimize the system at runtime, even in the face of newly encountered situations; and (iii) how to incorporate the notion of cost, referring to the negative impact of adaptation decisions, in the above processes.

To provide statistical guarantees, our self-optimization framework relies upon statistical procedures (t-tests, analyses of variance, binomial tests) at different phases. These procedures can be configured, e.g. by picking different thresholds for statistical significance levels, to increase or decrease the statistical guarantees that come in the form of confidence intervals and observed effect sizes.

For runtime optimization, we employ the state-of-the-art technique of Bayesian Optimization with Gaussian Processes

SEAMS '18, May 28–29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5715-9/18/05…$15.00
https://doi.org/10.1145/3194133.3194152

(BOGP). This technique supports multi-variate optimization of input parameters and effectively deals with high noise (variance) of system outputs. However, it needs a considerable number of iterations to converge when exploring large configuration spaces, i.e. large number or value ranges of input parameters. We provide a hybrid approach where BOGP is still used but applied only on a subset of the configuration space. This subset is prescribed by the system models learned from prior observations. We also incorporate a technique to handle cost in case of a bad configuration choice made by the BOGP optimizer.

The novel contribution of our work is that it provides the first framework, to the best of our knowledge, that combines runtime optimization with guarantees obtained from statistical testing and with a statistically grounded method for handling cost of bad adaptation decisions.

We have evaluated our framework by implementing it within the RTX tool for self-adaptation based on Big Data analytics [18] and applying it on a traffic navigation self-adaptation exemplar—CrowdNav [18]. Our results show that it is possible to exploit the knowledge of the target system to inform the self-optimization process, assuming that the system resides in an unknown situation and that we only have a small total budget for experimentation.

The rest of the paper is organized as follows. Section 2 provides an overview of the different phases of the approach, while Section 3 describes the running example and illustrates the challenges of optimizing it. In Section 4, the proposed approach is described in full detail, while in Section 5 a validation of the main claim is provided. Section 6 discusses important assumptions, Section 7 compares our approach to related ones, and Section 8 concludes with a summary of contributions.

2 OVERVIEW OF THE APPROACH

A self-optimization approach for systems with noisy outputs and high cost of bad adaptation decisions needs to be (i) efficient in finding an optimal configuration in the least amount of time and (ii) safe in not incurring too high cost. To achieve these goals, we propose to use prior knowledge of the system (the K in the MAPE-K loop for self-adaptive systems) in order to guide the exploration of promising configurations. We also propose to measure the cost of the individual stages in the self-optimization and stop the evaluation of bad configurations. We connect this with statistical guarantees, which represents an approach beyond the state of the art.

Formally, the self-optimization problem we are considering consists of finding the minimum of a response or output function $f: \mathbb{X} \rightarrow \mathbb{R}$, which takes n input parameters X_1, X_2, ..., X_n, which range in domains $Dom(X_1)$, $Dom(X_2)$, ..., $Dom(X_n)$ respectively. $\mathbb{X}$ is the *configuration space* and corresponds to the Cartesian product of all the domains of the parameters $Dom(X_1) \times Dom(X_2) \times ... \times Dom(X_n)$. A *configuration C* assigns a value to each of the input parameters.

Based on the above definitions, our approach for self-optimization consists of the following three phases (Figure 1):

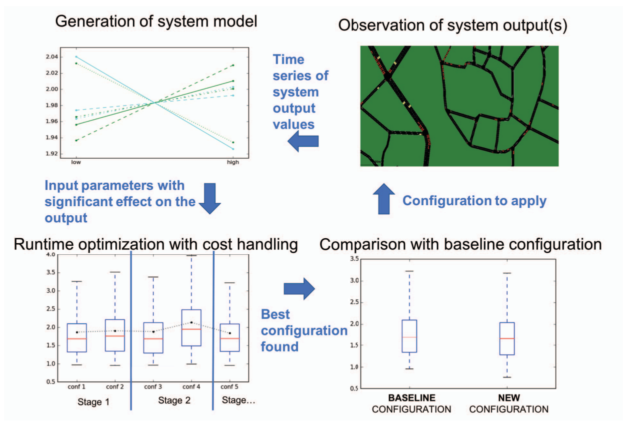

Generation of system model
Observation of system output(s)
Time series of system output values
Input parameters with significant effect on the output
Configuration to apply
Runtime optimization with cost handling
Comparison with baseline configuration
Best configuration found
conf 1
conf 2
conf 3
conf 4
conf 5
Stage 1
Stage 2
Stage...
BASELINE CONFIGURATION
NEW CONFIGURATION

Figure 1: Overview of the approach.

Generation of system model (Section 4.1), deals with building and maintaining the knowledge needed for self-optimization. Here we use factorial analysis of variance to process incoming raw data and create a statistically relevant model that is used in the subsequent phases. This model describes the effect of changing a single input parameter on the output, while ignoring the effect of any other parameters. It also describes the effects of changing multiple input parameters *together* on the output. This phase is run both prior to deploying the system using a simulator (to bootstrap the knowledge) and while the system is deployed in production using runtime monitoring (to gradually collect more accurate knowledge of the system in the real settings). The output of this phase is a list of input parameters or combinations of input parameters, ordered by decreasing effects (and corresponding significance levels) on the output.

Runtime optimization with cost handling (Section 4.2), selects values for the system input parameters to find a configuration in which the system performs the best, i.e. the output function is minimized. Configurations experimented with in this phase are generated by an optimizer that operates in several stages. At each stage, only a subset of the configuration space is available to the optimizer for generating configurations—we start from the subset that corresponds to the input parameters or combination of input parameters with the highest effect on the output, as reported by the first phase. We assume that configurations are rolled out incrementally in the system. If there is evidence that a configuration incurs high cost, its application stops and the optimizer moves on to evaluate the next configuration. To give statistical significance in determining if a configuration is not worth exploring anymore because of cost overstepping a given threshold, we use binomial testing. The outcome of this phase is the best configuration found by the optimizer.

Comparison with baseline configuration (Section 4.3), makes sure that a *new configuration* determined in the second phase is rolled out only when it is statistically significantly better than the existing configuration (*baseline configuration*). In order for the new configuration to replace the baseline, it has to be checked that (i) it indeed brings a benefit to the system (at a certain statistical significance level), (ii) the benefit is enough to justify any disruption that may result out of applying the new configuration to the system. The last point recognizes the presence of primacy effects, which pertain to inefficiencies caused by a new configuration to the users, as reported in controlled experiments on the web [11].

3 MOTIVATING SCENARIO

We illustrate our approach on the CrowdNav self-adaptation exemplar [18], whose goal is to optimize the duration of car trips in a city by adapting the parameters of the routing algorithm used for navigating the cars. CrowdNav is part of our previous work [18] and released as an open-source project at https://github.com/Starofall/CrowdNav.

In the scenario we considered, a number of cars is deployed in the city of Eichstädt with approx. 450 streets and 1200 intersections. Each car navigates from an initial (randomly allocated) position to a randomly chosen destination in the city. When a car reaches its destination, it picks another one at random and navigates to it. This process is repeated forever.

To navigate from point A to point B, a car needs to ask a router for a route (series of streets). There are two routers in CrowdNav: (i) the built-in router provided by SUMO (the simulation backend of CrowdNav) and (ii) a custom-built parametric router developed in our previous work. A certain number of cars ("regular cars") use the built-in router; the rest use the parametric router—we call these "smart cars".

The parametric router can be configured at runtime; it provides the seven configuration parameters depicted in Table 1. Each parameter is an interval-scaled variable that takes real values within a range of admissible values, provided by the designers of the system. Intuitively, certain configurations of the router's parameters yield better overall system performance.

To measure the overall performance of the system, CrowdNav relies on the metric of *trip overhead*. A trip overhead is a ratio-scaled variable whose values are calculated by dividing the observed duration of a trip versus the theoretical duration of the trip, i.e. the hypothetical duration of the trip if there were no other cars, the smart car travelled in maximum speed and did not stop in intersections or traffic lights. Only smart cars report their trip overheads at the end of their trips. Since some trips will have larger overhead than others no matter what the router configuration is, the data set of trip overheads exhibits high variance (Figure 2)—it can be thus considered a *noisy output*.

Together with the trip overhead, each smart car reports at the end of each trip a complaint value, i.e. a Boolean value indicating whether the driver is annoyed. The complaint value is generated based on the trip overhead and a random chance, as depicted in the simple logic of Listing 1. To measure the cost of a bad configuration in CrowdNav, the metric of *complaint rate* is used: the ratio of issued complaints to total number of observed *(trip overhead, complaint)* tuples.

Finally, CrowdNav resides in different situations depending on two environment parameters that can be observed, but not controlled: the number of regular (non-smart) cars and the number of smart cars. At each situation, a different configuration might be optimal. The task of self-optimizion in CrowdNav then becomes one of quickly finding the optimal configuration for the situation the system resides in and applying it.

[2] An explorer in CrowdNav is a car that is instructed to take longer route in order to provide information on traffic levels of under-explored streets to the router [18].

Table 1: Configurable parameters in CrowdNav's parametric router.

Id	Name	Range	Description
1	route randomization	[0-0.3]	Controls the random noise introduced to avoid giving the same routes
2	exploration percentage	[0-0.3]	Controls the ratio of smart cars used as explorers[2]
3	static info weight	[1-2.5]	Controls the importance of static information (i.e. max speed, street length) on routing
4	dynamic info weight	[1-2.5]	Controls the importance of dynamic information (i.e. observed traffic) on routing
5	exploration weight	[5-20]	Controls the degree of exploration of the explorers
6	data freshness threshold	[100-700]	Threshold for considering traffic-related data as stale and disregard them
7	re-routing frequency	[10-70]	Controls how often the router should be invoked to re-route a smart car

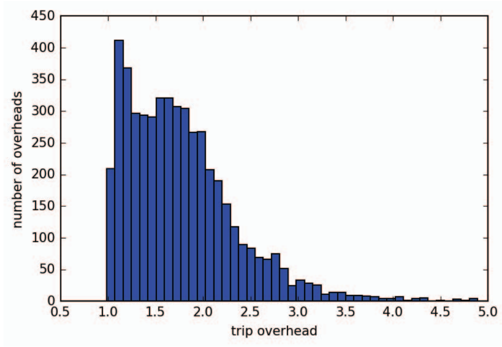


Figure 2: Histogram showing the distribution of trip overheads in an exemplary data set with 5000 data points.

```
def generate_complaint(trip_overhead):
    if trip_overhead > 2.5 and random.random() > 0.5:
        return True
    return False
```

Listing 1: Generation of complaints in CrowdNav. The threshold for overhead values that may warrant a complaint is empirically set to 2.5.

In this context, quickly finding a configuration of parameters that minimizes the trip overhead in a situation, while keeping the number of complaints in check, entails understanding the effect a parameter change has on the trip overhead and evaluating the effect a configuration has on the complaint rate.

Generalizing from this scenario, the problem to solve is: "Given a set of input system parameters *X*, an output system parameter *O* with values exhibiting high variance, an environment situation *S*, and a cost parameter *C*, find the values of each parameter in *X* that optimize *O* in *S* without exceeding *C*, in the least number of tries."

Having been in a number of similar situations the system should ideally learn from them in order for the optimization process to become faster over time. In the simplest case, returning to a known situation, for which an optimal configuration has been found in the past, should not trigger the optimization process, but just apply the optimal configuration.

In the next section, we detail on our approach for cost-aware self-optimization with statistical guarantees.

4 APPROACH

4.1 Generation of system model

Generating a system model to be used in self-adaptation involves observing the (i) situation the system resides in, (ii) its configuration—values of input parameters, (iii) the output of the system. For each situation, a model is built that ranks input parameters according to their effect on the output, from highest to lowest. To build such a model, we use a statistical procedure called factorial ANOVA, which we describe below.

Factorial ANOVA. The between-subjects factorial analysis of variance (ANOVA) is a parametric test employed with interval- or ratio-scaled data [20]. It is used for evaluating a factorial design, i.e. an experimental design employed to simultaneously evaluate the effect of two or more independent variables on a dependent variable. An independent variable is called a *factor*; a factor takes two or more values called *levels*. Factorial design and the associated factorial ANOVA method offer the opportunity to evaluate the effect of several factors on a dependent variable. Concretely, for each factor, factorial ANOVA evaluates the hypothesis of whether at least two of the factor's levels represent populations of the dependent variable with different mean values (i.e. the populations are significantly different at a significance level *a*). This hypothesis evaluates the presence or absence of so-called *main effects* of the factors on the dependent variable. In addition to analyzing main effects, factorial ANOVA also evaluates the hypothesis of whether a significant *interaction* (again, at a significance level *a*) exists between any combination of factors. An interaction is present for a combination of factors when the values of the dependent variable corresponding to the levels of one factor are not consistent across the levels of another factor. The result of factorial ANOVA contains one *F* value—the test statistic—for each of the independent variables and their possible combinations. The *F* values are compared against tabled critical *F* values for different significance levels (e.g. 0.05, 0.01); if the obtained *F* value is larger than the critical one, there is a significant effect. The main effects are linear regression models and thus can be graphically depicted in linear plots (Figure 3).

Example 1: Consider an experiment where two pills, P1 and P2, are evaluated on the treatment effect they have on a disease D. Each pill is a factor which can take two levels, low and high dosage; the independent variable is the treatment effect. In this setting, three different hypotheses can be evaluated with factorial ANOVA: (i) the effect of P1 in curing D, (ii) the effect of P2 in curing D, (ii) the combined effect or interaction of P1 and P2. Assuming the resulting F-values are $F_{P1}=48$, $F_{P2}=27$, *and* $F_{P1P2}=3$, *the corresponding probabilities of accepting the null hypotheses (taking into account the degrees of freedom per case) are 0.000121, 0.000826, and 0.121503, respectively. Thus, we conclude that, at significance level 0.05, there is significant difference in means for each of the factors (since 0.05>0.000121 and 0.05>0.000826) and no significance interaction between the factors (since 0.05<0.121503). Graphically, the main effects are depicted in Figure 3. The stronger effect of P1 is illustrated by the higher slope of the dashed line.*

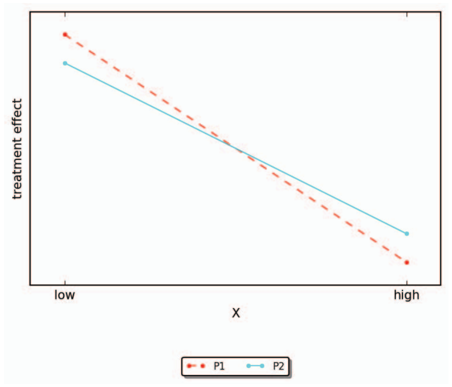


Figure 3: Main effects in Example 1.

Using factorial ANOVA. Viewing the system that we want to optimize as a black box, factorial ANOVA is employed to determine the input parameters that have the largest effect on the system output. Input parameters, in this setting, correspond to the independent variables or factors of the factorial design, while the system output corresponds to the dependent variable. This way, we treat different values of input parameters, resulting in different configurations, as "treatments" to the system we want to optimize.

We also define environment variables that work as blocking factors in our approach. A combination of values of environment variables prescribes a situation the system resides in. For each such situation, we build a system model using factorial ANOVA. The model is used in two places: (i) to set the default configuration for a situation, (ii) to inform the self-optimizer (Section 4.2) with the input parameters or combinations of input parameters that have the largest effect on the system output and thus are the most promising to explore first.

Importantly, the generation of system models in our approach works in an incremental way: the more situations are encountered and configurations are tried out, the more knowledge the system builds up that can be used in self-optimization. We assume that this knowledge is built both before deployment (e.g. in simulations), by performing a systematic application of factorial design on the configuration space, and while the system is in production.

Finally, note that knowledge can be transferred from one situation to another (a concept known as transfer learning [17]). This is important in case the system resides in a situation for which not enough data have been gathered. If the new situation is similar (according to certain similarity metrics) with one or more situations that are already encountered, a model for it can be built by applying factorial ANOVA on the raw data from *all* the similar situations. We come back to this point in Section 5.

Illustration on CrowdNav. In CrowdNav, the input parameters are the seven parameters depicted in Table 1. These become the factors of factorial ANOVA. This means that when applying a full factorial design with two levels per factor (minimum and maximum value for each factor) in CrowdNav, 2^7=128 different configurations need to be evaluated. The output of CrowdNav is the trip overhead function, which becomes the dependent variable of factorial ANOVA. Environment variables are the number of regular cars and the number of smart cars.

For illustration, Figure 4 depicts the main effects retrieved from applying factorial ANOVA to three different situations in CrowdNav, while Table 2 depicts the three most significant effects for each situation at significance level 0.0005. Key takeaways from inspecting both are the following: (i) at each situation, *different* factors have the most significant effects; (ii) the level of statistical guarantees that can be provided depends on the situation. For example, the values of the probabilities of accepting the null hypotheses (last column in the table), are far lower in the last situation than in the first one and thus provide greater guarantees in the last situation than in the first one. Finally, (iii) apart from main effects, interactions between two or even three parameters are also significant (see e.g. line 3 on Table 2 for an example of a significant three-way interaction).

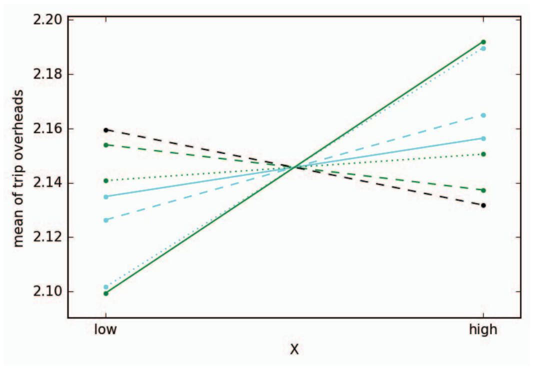
Default values for the parameters of a CrowdNav situation can be determined by taking into account the positive or negative slope of each line in the main effects plots. For example, for the situation in Figure 4(a), default values for exploration weight and re-routing frequency should be the minimum ones, owing to the positive slope of the respective lines.

Table 2: Results of applying factorial ANOVA in three situations of CrowdNav. Results are sorted by significance.

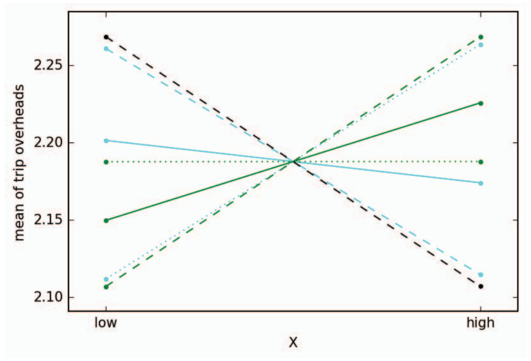
Situation		Parameter(s)	Prob (>F)
0 regular cars, 750 smart cars	1	exploration percentage, data freshness threshold	0.000002
	2	re-routing frequency	0.000004
	3	exploration percentage, static info weight, data freshness threshold	0.000007
400 regular cars, 350 smart cars	1	exploration percentage	0.000029
	2	static info weight	0.000029
	3	exploration weight	0.000085
0 regular cars, 350 smart cars	1	data freshness threshold	$\mathbf{4.21 \times 10^{-25}}$
	2	exploration percentage	$\mathbf{1.22 \times 10^{-14}}$
	3	data freshness threshold, static info weight	$\mathbf{2.25 \times 10^{-10}}$

4.2 Runtime optimization

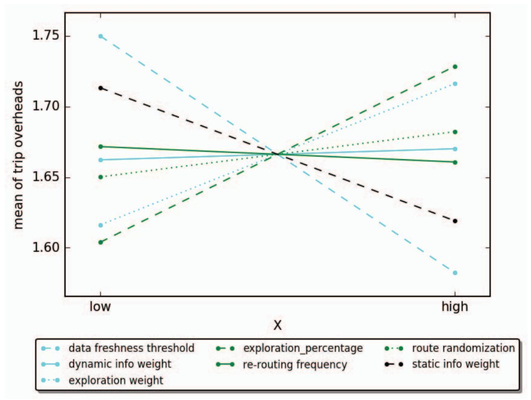
When viewing the general problem of optimizing a system output as the mathematical problem of minimizing a function f (with or without constraints), a number of different algorithms



(a) Situation: 0 regular cars, 750 smart cars



(b) Situation: 400 regular cars, 350 smart cars



(c) Situation: 0 regular cars, 350 smart cars

Figure 4: Main effects resulting from applying factorial ANOVA in three situations of CrowdNav.

and methods can be readily employed, such as variants of the Simplex algorithm [13] or the BFGS algorithm [16]. Each algorithm or method works under certain assumptions. In our approach, reflecting the realistic applicability, we assume the following:

1. *f* is a black-box for which no closed form is available, nor its derivatives;
2. evaluations of *f* are noisy;
3. evaluations of *f* are expensive (in terms of time or other application-specific cost metric).

The above assumptions are also the minimal assumptions of the *Bayesian optimization* framework, a sequential model-based approach to global optimization we have incorporated in our approach. We provide an overview of this framework and its Gaussian-Process-based specialization next.

Bayesian optimization with Gaussian Processes (BOGP). Optimization in the BOGP framework works in the following way [19]. At each execution step, BOGP builds a probabilistic model of the uni- or multi-variate function-to-be-minimized (called *objective function*) *f*, based on the so-far observed valuations of *f*. The model represents an approximation of the real function using a Gaussian process (called *posterior*). Equipped with this model, BOGP induces *acquisition functions* that leverage the uncertainty in the model to guide exploration. In particular, an acquisition function is maximized in order to find the next point for sampling. In our experiments, we used the "expected improvement" acquisition function for BOGP [19], which measures the expected amount of improvement of a configuration w.r.t. the so-far best observed configuration. After determining the next point (configuration) *p* for sampling and evaluating *f(p)*, the algorithm moves to the next execution step. Every BOGP works under a *budget*, which is the number of steps it is allowed to perform before it reports the so-far best value of *f*, together with the parameter values that induce it.

Using Bayesian optimization with Gaussian Processes. BOGP is employed in our approach in order to traverse a configuration space in an effective way when trying to optimize (minimize) the main system output. Instead of providing the BOGP the whole configuration space, we split the configuration space into subspaces that correspond to the parameter combinations outputted in the previous phase (Section 4.1). The optimization phase then executes BOGP in several stages, each of which takes as input a parameter combination. We define as *total budget* of the optimization phase the total number of BOGP steps across all BOGP executions. The total budget is allocated uniformly across all BOGP executions. For instance, assuming a total budget of 12 and 4 BOGP executions, each execution will be invoked with a budget of 3. By splitting the executions of BOGP and optimizing only along orthogonal (i.e. non-interacting subspaces), which were identified by ANOVA to have the largest effect on the system's output, we are able to find a meaningful minimum with even extremely small total budgets. Our results validating this claim are reported in Section 5.

Illustration on CrowdNav. When optimizing CrowdNav, the optimization phase takes the list of the input parameters or combinations of input parameters generated by ANOVA. This list is ordered by having the most significant factors (i.e. configuration parameters) first. For example, for CrowdNav in the situation of (0 regular cars, 750 smart cars), the two most significant sets of parameters are [exploration percentage, data freshness threshold], and re-routing frequency (Table 2). If CrowdNav resides in a situation (e.g. (0 regular cars, 500 smart cars)) for which there are not enough data, we reuse data from similar situations or we reuse data regardless of a situation. The latter strategy naturally lowers the guaranteed significance of input parameters generated by ANOVA, but still provides a best guess equipped with statistical significance based on the behavior of the system in other situations.

Assuming a total budget of 12 split into 4 stages, the first optimization stage uses BOGP to optimize the parameters [exploration percentage, data freshness threshold] in 3 iterations. In the second optimization stage, the best parameters from the previous stage are fixed and BOGP is used to optimize the second set of parameters (re-routing frequency) in 3 more BOGP iterations; the same process applies for the remaining two stages. In all stages, the objective function of BOGP is the average value of a specific number of observed trip overheads. In our experiments, we routinely used 5000 for such a number.

4.3 Cost handling in runtime optimization

At each optimization step, BOGP selects a configuration to try out, applies it on the system, and collects values of system outputs. We assume that the application of an experimental configuration is rolled out incrementally. This means that, at any point in time, only a fraction of the users experience the system under the experimental configuration, whereas the rest view the system in a default configuration. In CrowdNav, this means that only a fraction of the cars (e.g. 10%) use the router that is configured with the experimental configuration, and trip overheads are analyzed only from this fraction of cars. As time passes, the percentage of users exposed to experimental configuration increases (e.g. from 10% to 20%, then 30%, and so on). This incremental rollout allows to control the number of users that are exposed to a potentially bad configuration, i.e. a configuration that incurs a high cost to the system.

In order to identify such a bad configuration, we propose to observe the cost metric of the system under experimentation and interrupt the evaluation of a configuration (*experiment*) if the cost oversteps a certain level that is user-defined. To make sure that there is enough evidence that the high cost occurs consistently, we propose to use statistical testing to test the alternative hypothesis that the cost is larger than the threshold at a given statistical significance level. The choice of a particular statistical test to ensure statistically grounded decision heavily depends on how the cost is measured. We illustrate the usage of binomial test in the case of CrowdNav, where cost is a proportion of values of a binomial distribution, however, other tests can also be used in case of differently scaled cost metric. For instance, multinomial test can be used in case of multinomial distributions or t-test in case of interval- or ratio-scaled costs.

Illustration on CrowdNav. Cost in CrowdNav is measured by the ratio of issued complaints on the total number of observed *(trip overhead, complaint)* tuples. The threshold for the complaint ratio is empirically set to 12%. This means that, if there is sufficient evidence that the complaint ratio will overstep 12%, we abort a running experiment. In this case, the optimization will continue to the next step. To provide such evidence with statistical means, we perform a binomial test.

A **binomial test** is a statistical procedure that tests whether, in a single sample representing an underlying population of two categories, the proportion of observations in one of the two categories is equal to a specific value [20]. In our case, we use a one-tail binomial test employing the directional alternative hypothesis that predicts the underlying population proportion of issued complaints (complaints with value True) is above a specific value—our complaints threshold. We perform this test every ten *(trip overhead, complaint)* tuples that we receive. If we predict that, in the next checkpoint, the complaint ratio is expected to be larger than 12% (assuming that the complaint ratio at the next checkpoint remains the same as the current one) at a significance level α (we used 0.05), we abort the running experiment. This logic is illustrated in Listing 2. This test provides the necessary statistical backing to ensure that any overstepping of the threshold did not happen by accident (which should not be reason enough to abort an experiment).

```
complaints_ratio = issued_complaints / observed_tuples
next_observed_tuples = observed_tuples + step   # step=10
estimated_complaints = next_observed_tuples * complaints_ratio

p_val = binom_test(estimated_complaints, next_observed_tuples,
                   p = 0.12, alternative="greater")

if p_val < binom_test_alpha:   # binom_test_alpha = 0.05
    raise StopIteration("Too costly to continue this experiment")
```

Listing 2: Binomial test for cost handling in CrowdNav (Python snippet).

4.4 Comparison with baseline configuration

Once the total budget of the optimization phase is exhausted, the so-far best configuration is outputted. The third phase of our approach compares this configuration to the baseline configuration of the system. The reason for this is twofold: (i) to ensure that the so-far best configuration is indeed better than the baseline one at a specific statistical significance level *a*, (ii) to ensure that the effect size of the improvement is enough to justify the cost of applying the new configuration to the system. The cost of applying the configuration refers to any disruption that may result from changing the system. Such disruptions may even appear when the new configuration is actually better, since e.g. users are more used to the old configuration and need some time to adjust to the new one [11].

This phase takes thus as input two application-specific parameters, a statistical significance level *a* and a minimum effect size *e*, and performs a statistical test to determine whether the so-far best configuration induces system outputs that are better than the ones induced by the baseline configuration. If a (positive) statistical significant difference is observed at the provided level *a*, the effect size of the difference is calculated and compared to *e*. If the observed effect is larger than e, then the so-far best configuration is applied to the system, meaning that it is rolled out to all the users of the system.

In the following, we describe the use of t-test for this phase, however, other parametric or non-parametric tests for two independent samples can also be used (e.g. Mann-Whitney U), depending on the scale of the main system output and the assumption on its distribution (normal or not).

Illustration on CrowdNav. System output in CrowdNav is measured via the trip overhead metric, a ratio-scaled variable. To measure whether the output is better (smaller) with statistical means we perform t-test.

A **t-test** for two independent samples is a statistical test evaluating whether the samples represent two populations with different mean values [20]. The result of a t-test is a *p-value*, which is compared to a (pre-selected) statistical significance level *a*; if the *p-value* is less or equal to *a*, the test concludes that there is statistical significance difference in the means. Effect size in t-tests is typically computed by the *Cohen's d* metric, which is defined as the difference of the means of the two samples divided by their pooled standard deviation. A value of 1 for *Cohen's d* indicates that the two means differ by one standard deviation, a value of 0.5 by half standard deviation, and so on.

In our case, we are using a one-tailed test evaluating whether the mean of trip overheads derived from applying the so-far best configuration is smaller than the mean of trip overheads derived from applying the baseline configuration. T-test assumes that the distribution of data in the underlying populations is normal, which does not hold for the original data set of trip overheads (Figure 2). We therefore perform a logarithmic transformation on the trip overhead data that results in a distribution that is closer to normal, and thus allows us to perform t-test without compromising its reliability. Also, apart from the standardized Cohen's d effect size, we also provide an unstandardized measure of effect size by calculating the difference in the means of the samples (which is a more natural way to think about the actual difference at the application level).

To illustrate the result of this phase, consider comparing two CrowdNav configurations *C1* and *C2*, where *C1* is the so-far best and *C2* is the baseline configuration. The results are reported as:

p value:	0.000048
Cohen's d:	0.078039
difference in means:	0.024358

Assuming that we have requested, prior to performing the test, a significance level of 0.05 and a difference in means of at least 0.01, these results show both statistical significant difference (p value < 0.05) and significant effect (difference in means > 0.01). This provides statistical evidence for our approach to apply the so-far best configuration to CrowdNav (which then becomes the new baseline for the situation).

5 EVALUATION

To evaluate our approach, we have implemented the three different phases in Python and used it to optimize the CrowdNav exemplar. Our implementation is part of the RTX tool for self-optimization based on Big Data analytics [18]. In particular, we use RTX for observing the data produced by CrowdNav (our testbed) and for applying different configurations to it. Both interactions are mediated via Kafka[3], a distributed messaging system. To persist the raw data and enable also a posteriori analysis we have used Elasticsearch[4], a distributed document database tailored for time series data. For all the statistical procedures and the Bayesian optimization method, we used existing Python libraries[5]. We have open-sourced all the code and the data; more information is provided in the online Appendix at https://github.com/Starofall/RTX/tree/seams18.

After implementing the different phases of the approach and performing several curiosity-driven experiments to acquire domain expertise with CrowdNav that would allow us to set meaningful values to parameters such as significance levels for the various statistical procedures, we ventured into validating one of our major claims, namely:

"Can we exploit the knowledge of our generated system models in informing the self-optimization process, when assuming that the system resides in an unknown situation and that we only have a small total budget for experimentation?"

To answer this question, we performed a comparative experimental study (controlled experiment). The goal of the study was to compare the end result of our approach (the best configuration found) to a baseline scenario in which the state-of-the-art BOGP is used in its vanilla form. In both cases, we apply self-optimization in the CrowdNav situation of (300 regular cars, 300 smart cars), for which we have no prior knowledge. In both cases, the total budget for experimentation is 5 steps of BOGP, and the objective function of BOGP is the average value of each configuration evaluated on 5000 samples of the trip overhead function. We first explain the baseline scenario and show its results, then describe how our approach led to different results and finally compare the two.

Baseline scenario. In this scenario, we let the BOGP optimizer run *once* with the *full* budget (all 5 steps) and with the whole configuration space (all 7 CrowdNav parameters and their ranges as depicted in Table 1). This corresponds to a standard usage of the BOGP optimizer, which is one of the most powerful tools in performing optimization of noisy functions. BOGP terminates after 5 steps, plotted in Figure 5, by reporting the best configuration found, $C_{baseline}$ (Table 3).

Our approach. When applying our approach, we first need to build a system model for the CrowdNav situation. Since we do not have any data (it is a new situation), we use the data from all the other situations we have observed so far, in particular from

[3] https://kafka.apache.org/
[4] https://www.elastic.co
[5] scipy.stats, statsmodels, skopt

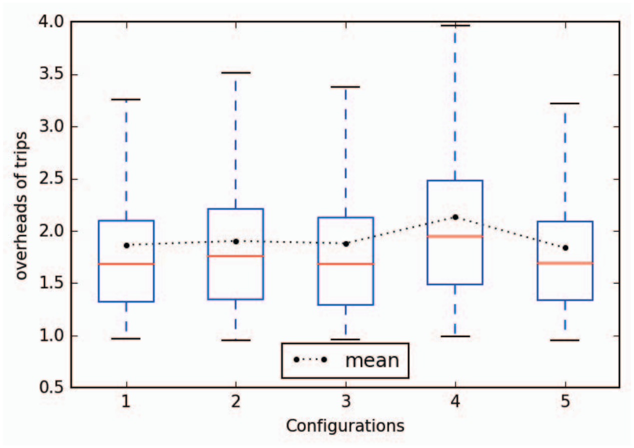


Figure 5: Box plots of BOGP results in the baseline scenario.

Table 3: Best values of CrowdNav parameters.

	$C_{baseline}$	$C_{approach}$
route randomization	0.11615	0
exploration percentage	0.19508	0
static info weight	2.36432	1.22184
dynamic info weight	1.20810	1
exploration weight	15	5
data freshness threshold	359	700
re-routing frequency	63	10

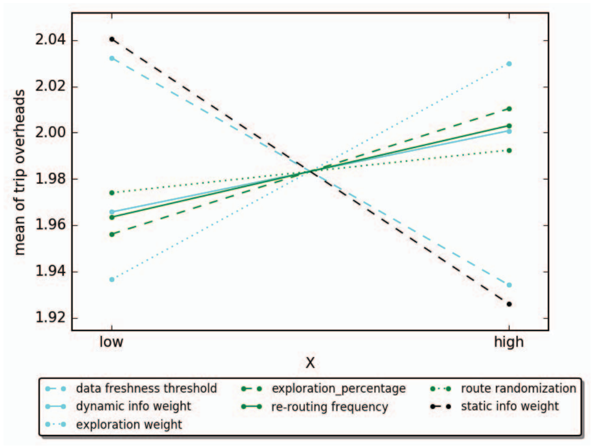


Figure 6: Main effects when applying factorial ANOVA in data from all three situations of CrowdNav.

Table 4: Results (sorted by significance) of applying ANOVA in data from all three situations of CrowdNav.

	Parameter(s)	**Prob (>F)**
1	static info weight	$\mathbf{3.11 \times 10^{-10}}$
2	data freshness threshold	$\mathbf{6.81 \times 10^{-08}}$
3	exploration weight	$\mathbf{2.70 \times 10^{-07}}$

the three situations depicted in Figure 4. The results are depicted in Table 4 (3 most significant effects); the main effects are also graphically depicted in Figure 6. Based on these results, the approach sets the baseline configuration (not to be confused with the baseline scenario) according to the slopes of the lines in Figure 6 and proceeds to perform BOGP in several stages. Since the overall budget is 5, and a BOGP execution needs *at least* 2 steps, only two stages are prescribed: the first stage takes a budget of 3 steps and explores the subspace of static info weight (indicated as the parameter with the highest effect—Table 4); the second stage takes a budget of 2 steps and explores the subspace of data freshness threshold. The best observed configuration of the first stage (static info weight: 1.22184) is applied in all subsequent experiments of the second stage. The result of the second stage (data freshness threshold: 700), together with the result of the first one and the default values for the rest of the parameters form the best configuration returned by our approach, $C_{approach}$ (Table 3). The results of 5 steps (divided into 2 stages) of BOGP in our approach are plotted in Figure 7.

Results and Interpretation. To compare the end result of our approach against that of the baseline, we performed a t-test evaluating whether the mean overhead of $C_{approach}$ is smaller, at a statistically significance level 0.05, than the mean overhead of $C_{baseline.}$ (In particular, we applied the test after applying logarithmic transformation to trip overhead values to normalize their distribution.) The result from the t-test indicated that such a difference indeed exists (p value: 0.013, power: 72%).

Comparing the runs of the BOGP optimizer for each setting side by side (Figure 8), we observe that the most plausible reason why our approach outperformed the baseline is that the BOGP in the baseline case did not have enough budget (5 steps) w.r.t. the number of parameters it needed to experiment with (7 parameters). In comparison, BOGP in our approach, even though it operated in an even more constrained budget (3 and 2 steps), it also operated always with only a single parameter, which made overshooting less probable and traversing the configuration space feasible even under such "starvation" settings.

Limitations. It has to be mentioned that we did not observe the same superiority of our approach in experiments where the overall budget is relatively large. For instance, we experimented with an overall budget of 20 steps, which we divided in our approach into 5 stages of 4 steps each. Under this setting, the baseline performed slightly better than our approach (albeit not statistically significant better). The reason for this is that the BOGP in the baseline had enough budget to effectively traverse the configuration space consisting of all the seven parameters. The 20 steps budget is however more than we usually assume in our approach, where we specifically target systems where experiments are costly or lengthy and the budget is tight (e.g. 5 steps in CrowdNav).

6 DISCUSSION

Scalability of model generation phase. When using a full-factorial design, the number of configurations to evaluate grows exponentially due to combinatorial explosion. In the case of CrowdNav, 7 parameters with 2 values per parameter resulted in 128 configurations to evaluate. To reach a significance level *a* of 0.0005 for three effects (our stopping criterion), we had to collect less than 100 output (trip overhead) values per configuration. Increasing *a* reduces the number of observations needed, but also reduces the obtained statistical guarantees. This nevertheless allows us to collect only as many observations as we are realistically able to at this phase. Another way to speed up this phase is to reduce the number of configurations that need to be evaluated by employing a fractional factorial design (e.g. Plackett-Burman design [20]). This can reduce the number of configurations to half or quarter of the full factorial case; it typically allows, however, to examine for main effects and only for some of the interactions. Finally, a third way to speed up the model generation phase, is to examine multiple configurations in parallel (given the necessary infrastructure), as already done in web experimentation [11].

Assumptions on statistical methods. In our approach we use several statistical methods: analysis of variance (ANOVA), binomial test, and t-test. These methods have several

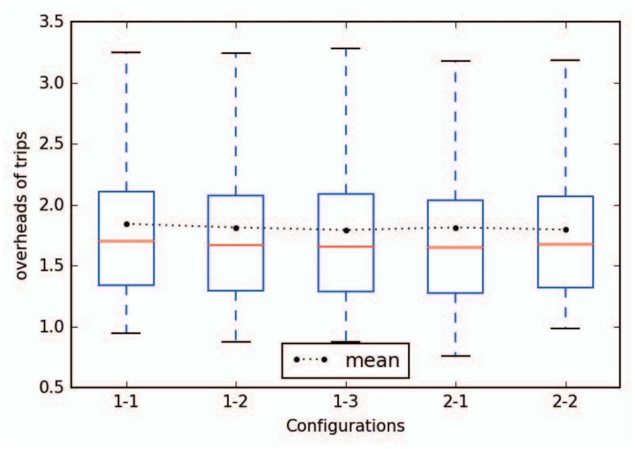


Figure 7: Box plots of BOGP results in our approach: The first 3 steps belong to the first execution stage, the last 2 to the second.

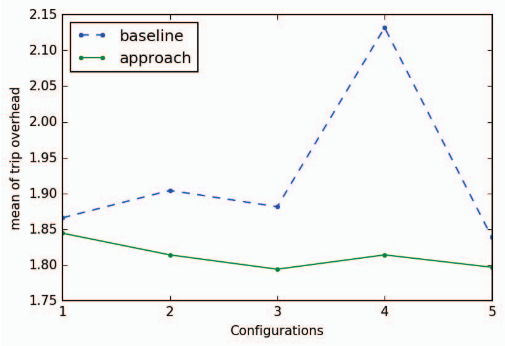


Figure 8: Mean values of trip overhead during the optimization in the baseline scenario and our approach.

assumptions. In particular, they assume that observations are independent. Furthermore, the ANOVA and t-test assume normality and homogeneity of variances across samples. ANOVA and t-test (due to the central limit theorem) are quite robust against violating the normality assumption, which makes them usable for data coming from real systems where the normality assumption does not fully hold. The assumption on independence is more important because its violation makes the tests more confident than they should be, given the collected observations. If there is a reason to assume that data are dependent, it is necessary to adapt the statistical methods to work with dependent observations. This typically requires assumption on bounded length n of the dependency (formally, absolute summability of the autocovariance function is required [12]). With this assumption, it is possible to obtain independent data by taking every n-th item from observations. Alternatively, a statistical method specifically aiming at dependent data can be used instead—e.g. block bootstrap—to determine test quantiles.

7 RELATED WORK

Self-adaptation and self-optimization has been a topic of active research the past years. Most of the proposed approaches rely on some form of *white-box* analytical model of the system-to-be-adapted, i.e. a model of the internal structure of the system that relates the impact of adaptation decisions of the system's goals [2]. Such model can be an architectural model [1, 5], a timed automaton [8], a Markov process [6], or a mathematical model used in feedback-based control [4]. In our approach, we assume that such models are difficult and error-prone to build and maintain for the class of large complex software-intensive systems that we target. We thus rely on *black-box models* as system abstractions and try to perform self-optimization in this setting. We review here other self-adaptation approaches that do the same and compare against them.

One of the first approaches of black-box adaptation was proposed with the FUSION framework [2, 3]. FUSION employs feature modeling to model dependencies between the architectural elements in a system and online learning to induce the impact of selecting a feature configuration on the system's goals. When a situation that warrants adaptation is detected, the system switches to the configuration that best satisfies the system's goals. FUSION employs a learning cycle in which a configuration is applied and values of goal metrics are observed. For each goal metric, a linear regression model is learned (via the M5 model tree algorithm) that associates tunable parameters (Boolean feature selection variables) and their interactions with the goal metric. Learning stops when a learning accuracy threshold is reached. Our approach also generates linear regression models via factorial ANOVA to capture the dependencies between tunable parameters and system output (similar to the work in [22]) and relies on statistical thresholds for controlling the amount of learning uncertainty that can be tolerated. However, we assume tunable parameters to be scalar variables and we use the result of the learning phase to inform a subsequent cost-aware black-box optimization phase.

Black-box optimization is the task of optimizing an objective function with a limited budget for evaluations [7]. A powerful type of black-box optimization because of its ability to tolerate noisy objective functions is Bayesian optimization with Gaussian Processes (BOGP) [21]. We are not the first to employ BOGP in self-optimization. BOGP has been recently successfully used in configuration optimization of Big Data stream processing systems [10]. Also, Gaussian Processes have been recently proposed for building performance models in the context of transfer learning for self-adaptation [9].

Finally, our work is related to recent attempts towards an architectural framework for automated experimentation, which focus on identifying the principle software architecture qualities and design decisions underlying such framework [14, 15]. Our approach, and the corresponding implementation on top of the RTX engine, can be seen as an instantiation of such framework which also allows to (i) deal with noisy outputs, (ii) provide statistical guarantees, (iii) deal with experimentation cost.

8 CONCLUSION

In this paper, we have focused on a class of systems that is both hard and risky to self-optimize. The reason is that complex software-intensive systems are difficult to model a priori, have considerable variation in their outputs (noisy outputs) and incur high cost of bad adaptation decisions. We have detailed on an instance of such systems which pertains to traffic navigation in a city. We have described our framework for self-optimization which consists of three phases: generation of system model, runtime optimization with cost handling, and comparison with baseline configuration. At each phase, we have employed statistical procedures to back up our analysis with statistical guarantees. Our prototype implementation and results prove that it is possible to exploit the knowledge of the target system to inform the self-optimization process, assuming that the system resides in an unknown situation and that we only have a small total budget for experimentation. We believe these results can be generalized to other systems of the same class.

In the future, we intend to investigate the application of transfer learning (similar to [9]) to reuse knowledge from similar situations. We also intend to investigate how to make the Bayesian optimization method aware of application-level risk so that configuration selection is based not only on maximizing the expected improvement but also on minimizing the expected risk.

ACKNOWLEDGMENTS

This work has been partly funded by the Bayerisches Staatsministerium für Wirtschaft und Medien, Energie und Technologie as part of the TUM Living Lab Connected Mobility Project and partly sponsored by the German Ministry of Education and Research (BMBF) under grant n^o 01Is16043A. The work has been partially supported by project no. LTE117003 (ESTABLISH) from the INTER-EUREKA LTE117 programme by the Ministry of Education, Youth and Sports of the Czech Republic.

REFERENCES

[1] Cheng, S.-W. et al. 2012. Stitch: A language for architecture-based self-adaptation. *Journal of Systems and Software*. 85, 12 (2012), 1–38. DOI:https://doi.org/10.1016/j.jss.2012.02.060.

[2] Elkhodary, A. et al. 2010. FUSION: A Framework for Engineering Self-tuning Self-adaptive Software Systems. *Proc. of FSE '10* (2010), 7–16.

[3] Esfahani, N. et al. 2013. A Learning-Based Framework for Engineering Feature-Oriented Self-Adaptive Software Systems. *IEEE Transactions on Software Engineering*. 39, 11 (Nov. 2013), 1467–1493.

[4] Filieri, A. et al. 2014. Automated Design of Self-adaptive Software with Control-theoretical Formal Guarantees. *Proceedings of the 36th International Conference on Software Engineering* (New York, NY, USA, 2014), 299–310.

[5] Garlan, D. et al. 2004. Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*. 37, 10 (2004), 46–54. DOI:https://doi.org/10.1109/MC.2004.175.

[6] Ghezzi, C. et al. 2013. Managing Non-functional Uncertainty via Model-driven Adaptivity. *Proc. of ICSE'13* (2013), 33–42.

[7] Golovin, D. et al. 2017. Google Vizier: A Service for Black-Box Optimization. (2017), 1487–1495.

[8] Iftikhar, M.U. and Weyns, D. 2014. ActivFORMS: Active Formal Models for Self-adaptation. *Proc. SEAMS '14* (2014), 125–134.

[9] Jamshidi, P. et al. 2017. Transfer Learning for Improving Model Predictions in Highly Configurable Software. *arXiv:1704.00234 [cs]* (Apr. 2017).

[10] Jamshidi, P. and Casale, G. 2016. An uncertainty-aware approach to optimal configuration of stream processing systems. *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2016 IEEE 24th International Symposium on* (2016), 39–48.

[11] Kohavi, R. et al. 2009. Controlled experiments on the web: survey and practical guide. *Data Mining and Knowledge Discovery*. 18, 1 (Feb. 2009), 140–181. DOI:https://doi.org/10.1007/s10618-008-0114-1.

[12] Kreiss, J.-P. and Paparoditis, E. 2011. Bootstrap methods for dependent data: A review. *Journal of the Korean Statistical Society*. 40, 4 (Dec. 2011), 357–378. DOI:https://doi.org/10.1016/j.jkss.2011.08.009.

[13] Lewis, R.M. et al. 2000. Direct search methods: then and now. *Journal of Computational and Applied Mathematics*. 124, 1 (Dec. 2000), 191–207. DOI:https://doi.org/10.1016/S0377-0427(00)00423-4.

[14] Mattos, D.I. et al. 2017. More for Less: Automated Experimentation in Software-Intensive Systems. *Product-Focused Software Process Improvement* (Nov. 2017), 146–161.

[15] Mattos, D.I. et al. 2017. Your System Gets Better Every Day You Use It: Towards Automated Continuous Experimentation. *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (Aug. 2017), 256–265.

[16] Nocedal, J. and Wright, S.J. 2006. *Numerical Optimization*. Springer.

[17] Pan, S.J. and Yang, Q. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering*. 22, 10 (Oct. 2010), 1345–1359. DOI:https://doi.org/10.1109/TKDE.2009.191.

[18] Schmid, S. et al. 2017. Self-Adaptation Based on Big Data Analytics: A Model Problem and Tool. *Proc. of SEAMS 2017* (Buenos Aires, Argentina, May 2017), 102–108.

[19] Shahriari, B. et al. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*. 104, 1 (Jan. 2016), 148–175. DOI:https://doi.org/10.1109/JPROC.2015.2494218.

[20] Sheskin, D.J. 2007. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC.

[21] Snoek, J. et al. 2012. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems* (2012), 2951–2959.

[22] Ustinova, T. and Jamshidi, P. 2015. Modelling multi-tier enterprise applications behaviour with design of experiments technique. *Proceedings of the 1st International Workshop on Quality-Aware DevOps* (2015), 13–18.