Search-Based Scheduling of Experiments in Continuous Deployment

Gerald Schermann
University of Zurich
Zurich, Switzerland
schermann@ifi.uzh.ch

Philipp Leitner
Chalmers | University of Gothenburg
Gothenburg, Sweden
philipp.leitner@chalmers.se

***Abstract*—Continuous experimentation involves practices for testing new functionality on a small fraction of the user base in production environments. Running multiple experiments in parallel requires handling user assignments (i.e., which users are part of which experiments) carefully as experiments might overlap and influence each other. Furthermore, experiments are prone to change, get canceled, or are adjusted and restarted, and new ones are added regularly. We formulate this as an optimization problem, fostering the parallel execution of experiments and making sure that enough data is collected for every experiment avoiding overlapping experiments. We propose a genetic algorithm that is capable of (re-)scheduling experiments and compare with other search-based approaches (random sampling, local search, and simulated annealing). Our evaluation shows that our genetic implementation outperforms the other approaches by up to 19% regarding the fitness of the solutions identified and up to a factor three in execution time in our evaluation scenarios.**

I. Introduction

A high degree of automation (e.g., building, testing, and deploying software artifacts) enables companies, and especially Web-based companies, to release new functionality more frequently and faster. Companies such as Microsoft [1], Facebook [2], Google [3], or Netflix [4] are characterized by having hundreds of deployments a day throughout their software ecosystem. Sophisticated monitoring and telemetry solutions keep track of releases and the captured live production data has become the basis for data-driven decision making [5]. Instead of shipping new features or functionality to all users, continuous experimentation practices such as A/B testing [6] or canary releases [7] enable companies to test new features on a small fraction of the user base first. Fast insights from live data are paired with manageable risks, if things go wrong "just" a fraction of the users is affected.

However, setting up such an experimentation infrastructure is not a straightforward task, as demonstrated by experience reports of, e.g., Kevic et al. [1], Xu et al. [8], and Fabijan et al. [9]. An essential requirement for successful experimentation is to collect enough data to draw valid statistical conclusions (cf. Kohavi et al. [10]). Having multiple experiments running at the same time requires careful user assignments (i.e., which users are part of which experiments) as experiments might overlap and influence each other. To avoid these situations, some experiments might require to await the termination of previous experiments, or to run in parallel but on different fractions of the user base. The former is simple, yet not feasible as development work continues and delayed releases should be avoided. The latter requires scheduling a potentially scarce resource (i.e., users interacting with the application) in a domain prone to change. Experiments fail frequently, captured feedback gets integrated, and experiments are reiterated, i.e., re-executed after these adjustments. For example, at Microsoft Bing [1], 33.4% of experiments are ultimately deployed to all users, and experiments are iterated 1.8 times on average. Hence, scheduling experiments is not a self-contained task. Experiment restarts, reschedulings (e.g., different user group, or day), and cancellations (i.e., pre-scheduled resources become available for other experiments) need to be dealt with. Furthermore, the search space of how users can be assigned to experiments is massive.

In this paper, we define the problem of experiment scheduling as an optimization problem. Experiments should start as soon as the coding part of a feature is done, avoiding delays of multiple days or even weeks. Moreover, enough data has to be collected throughout the experiments, but at the same time we need to guarantee that the scheduled resources are distributed fairly to foster parallel experiment execution. For this, we propose a genetic algorithm that is capable of (re-)scheduling (running) experiments. We envision our approach to become an active part of a release or deployment pipeline [7], periodically (e.g., daily or even multiple times a day) updating the experiment schedule, accounting for experiment cancellations or restarts. The resulting schedule is then used to instrument the system for experiment execution, i.e., administering the states of dynamic feature toggles [11] or traffic routing mechanisms [12]. We compare the capability of our genetic algorithm with the capability of other search-based approaches (random sampling, local search, and simulated annealing). To summarize, our main contributions are:

- The definition of experiment scheduling as an optimization problem.
- Implementations of a genetic algorithm (including custom definitions of crossover and mutation), random sampling, local search, and simulated annealing for this domain.
- An extensive evaluation showing that the genetic algorithm outperforms the other search-based approaches by up to 19% regarding the fitness score of the solutions identified and up to a factor of three in execution time.

Our tooling, source code, and evaluation data (i.e., example experiments and scripts) are available online [13] fostering experiment replication and extension.

II. Background

In the following, we provide background information on the different types of experimentation, typical considerations before launching an experiment, and what problem arise in the context of scheduling continuous experiments.

A. Types of Experimentation

Experimentation practices such as A/B testing [6], canary releases [7], or dark launches [2] give companies fast insights into how new features perform while keeping the risks manageable at the same time. Using the terminology of Schermann et al. [14], [15], these experiments are categorized into two flavors: *regression-driven* and *business-driven* experiments.

Regression-driven experiments are used to mitigate technical problems (e.g., performance regressions) when testing new features, to conduct (system) health checks, and to test the scalability of the application's landscape on production workload. These experiments typically run from minutes to multiple days and mainly involve the practices of canary releases and dark launches.

Business-driven experiments guide different implementation decisions or variants of features from a business perspective (e.g., do customers appreciate this feature). These experiments typically involve A/B testing and run for multiple weeks or even months.

A common practice for both types of experimentation are gradual rollouts [7] in which the number of users assigned to an experiment is increased in a stepwise manner (e.g., increase from 2% of the Canadian users to 5%, then 10% and so on).

B. Ingredients of Experimentation

The core ingredients for all these types of experiments are the same. Once the responsible developer or analyst decides to launch an experiment, they need to have an understanding of (1) what to measure, i.e., the overall evaluation criterion (OEC) [16], [9], (2) how many data points to collect for being able to statistically reason about the OEC and thus the experiment's outcome (i.e., sample size), (3) which users to conduct the experiment on (e.g., different user groups, regions), and (4) when to run the experiment.

An OEC is highly domain dependent (e.g., units sold, number of users streaming videos) and represents a quantitative measure of the experiment's objective [16]. An essential decision is which users or user groups to consider for an experiment, thus who to assign to control and treatment groups. Users in treatment groups test new functionality, while control group users continue using the previous (stable) version and serve as reference points for (statistically) evaluating the experiment's outcome. Further factors involve that user groups might interact differently with a system (e.g., usage behavior of users paying for the service vs. users using it for free), that these groups are of different sizes, and that the time when an experiment runs may matter (e.g., time of the day, day of the week). Consequently, the duration of an experiment highly depends on these factors, plus all the other experiments that run at the same time on the same or overlapping user groups, i.e., there might be less traffic available for a single experiment in such a setting requiring longer experiment duration to collect a sufficient number of data points.

C. Uncertainty of Experimentation

Continuous experiments test new ideas. However, it is quite natural that not every idea ends up being successful. Experiments fail, failures are analyzed, and if there is a way to improve, experiments are repeated. The consequence is that scheduling experiments is not a one-time task. Resources budgeted for canceled experiments can be reused by other experiments, potentially reducing their overall execution time as the required sample size is reached faster (though still keeping in mind that there often exists a minimum duration to measure a certain effect).

Furthermore, dealing with traffic profiles in the context of experimentation is trying to estimate how the future might be while looking into the past. However, we cannot foresee how a new feature changes the users interactions' with a system. For instance, a particularly well-received new feature may cause traffic to explode. Consequently, experiment schedules need to be periodically re-evaluated, and experiments may need to be extended, shortened, or postponed.

In our research, we take this "uncertainty" into account and provide an approach that is able to deal with rescheduling of experiments, frequently adjusting to changed (user) behavior, and results in experimentation schedules that support a "valid" (i.e., avoid overlapping experiments) execution of multiple experiments at the same time.

III. Related Work

We distinguish between related work on continuous experimentation in general, involving experience reports on how companies conduct experiments, and how search-based software engineering techniques have been used in similar domains and in the context of scheduling.

Continuous Experimentation. Research on continuous experimentation has gained traction recently. There have been experience reports specifically investigating the process, challenges, and characteristics of conducting experiments in an enterprise company setting. These reports involve for example work by Kevic et al. [1] and Fabijan et al. [9] looking at the process of Microsoft. Moreover, the work of Xu et al. [8] and Tang et al. [3] specifically covers how LinkedIn and Google approach handling multiple experiments in parallel. Different to our own and Google's approach, at LinkedIn experiments are fully overlapping by default as in most cases their tests are restricted to the UI level and run on different (sub-)parts of the system. Google uses a system of layers and domains to divide up the user space in order to avoid overlapping and conflicting experiments. However, the underlying assumption

is that there is always enough user interaction available to collect enough data across all layers and subdomains, which may be true for Google, but less so for other companies with different patterns of user interaction and a smaller user base. Beside research focusing on single companies, there exist also multiple empirical studies, e.g., Lindgren and Münch [17] and Schermann et al. [14]. Still within the context of continuous experimentation, but more from a data science angle is the work of Kohavi et al. [16], [6], [10] and Crook et al. [18].

Search-based Software Engineering. Over the last years search techniques, and especially genetic algorithms, have become popular to tackle problems in multiple areas of software engineering, ranging from test data generation [19], software patches and bug fixes [20], refactoring [21], effort estimation [22], defect prediction [23], [24], to cloud deployment [25]. In the context of continuous experimentation, Tamburrelli and Margara [26] formulate automated A/B testing as an optimization problem. They propose a framework that supports the generation of different software variants using aspect-oriented programming, the runtime evaluation of these variants, and the continuous evolution of the system by mapping A/B testing to a search-based SE problem. While they focus on the generation of the variants, our focus is on the actual execution of continuous experiments taking into account various constraints such as a limited number of users and avoiding overlapping experiments. Heuristics including genetic algorithms and simulated annealing have also been used in the context of scheduling resources. Wall [27] provides a general overview of resource-constrained scheduling and describes scheduling as a dynamic problem, i.e., scheduling algorithms must be capable of reacting to changing requirements (e.g., when the availability of resources changes, or interruptions occur).

To the best of our knowledge, the problem of scheduling continuous experiments (i.e., a domain that is prone to change) has not yet been tackled.

IV. Problem Representation

We formulate the problem of finding valid experimentation schedules as an optimization problem. We use weighted-sum as the parametric scalarizing approach [28] to convert multiple objectives into a single-objective optimization problem. Given a set of n experiments $E = \{E_1, E_2, \ldots, E_n\}$, the goal is to identify a valid schedule S with maximal fitness, thus, maximizing the values of the problem's objectives. In this paper, we focus on scheduling experiments targeting a single service. First, we discuss how experiments and schedules are represented. Next, we discuss how the fitness of a schedule is defined and what defines a valid schedule.

A. Experiments

Scheduling experiments requires some basic information for every experiment E_i (summarized in Table I). Besides a unique *identifier*, this involves the *type* of the experiment (i.e., business- or regression-driven experiment), the minimum experiment *duration* that is needed for measuring a certain effect, the *sample size*, thus how many data points to collect for an experiment, the *priority* of an experiment, and the *preferred user group* to test with.

TABLE I
Input data for experiments

Property	Type	Description	Example
ID	Integer	Unique experiment ID	1
Type	Enum	Business- or regression-driven experiment?	Regression
Min Duration	Integer	Minimum experiment duration in hours	240
Sample Size	Integer	Minimum required exp. sample size (RESS)	10,000,000
Priority	Integer	Experiment priority (≥ 0)	6
Preferred UGs	[String]	List of preferred user groups to test with	$[3, 4]$
Gradual	Boolean	Stepwise increase of assigned users?	True
Start Traffic	Integer	In case of gradual, # of users to start with	10,000

When it comes to how experiments accumulate the required sample size in the course of their execution we distinguish between *gradual* and *constant* traffic consumption.

Constant consumption: Throughout its execution an experiment "consumes" a constant amount of traffic, i.e., the number of data points to collect at every hour x is defined as $c_x = \frac{samplesize}{duration}$.

Gradual consumption: Starting with a sample size t, the number of data points accumulated at every hour throughout an experiment with duration d increases in a stepwise manner. In our case, we rely on a simple linear function with a positive slope, i.e., the consumption c_x at hour x corresponds to $c_x = kx + t$. The factor k for the increase is obtained from the integral $\int_1^d kx + t\,dx = samplesize$, i.e., the total consumption throughout the experiment (i.e., the area of the linear function for the interval 1 to d) has to sum up to the minimum sample size.

B. Schedules

A schedule S for n experiments $E = \{E_1, \ldots, E_n\}$ consists of a set $S = \{S_1, S_2, \ldots S_n\}$, in which every schedule S_i corresponds to an experiment E_i. Every schedule S_i is a tuple $\langle \tau, A \rangle$, consisting of a start slot τ (i.e., the hour to launch the experiment) and a tuple of assignments $A = \langle A_1, A_2, \ldots, A_d \rangle$, where d corresponds to the duration of the experiment in hours. Every individual assignment A_i specifies how many users (in percent) of which user groups are part of the experiment at hour i. This is defined as a matrix $\{group : consumption\}$, e.g., $A_4 = \{group1 : 0.05, group2 : 0.0, group3 : 0.01\}$. In this example, 5% of the users of user group 1 and 1% of the users of user group 3 are part of the experiment at hour 4 of its execution.

C. Fitness

The fitness function applied on a schedule S represents a trade-off between three conflicting objectives: experiment duration, start, and user group. All three are assigned separate scores.

Duration score: An experiment E_i should not take longer than required, i.e., the length of its schedule S_i should – in the best case – equal the experiment's minimum duration. For example, results for an experiment on the system's scaling

capabilities should be available after 3 days instead of 2 weeks. The duration score ds_i of experiment E_i corresponds to $ds_i = \frac{minDuration}{d}$, where d denotes the duration of the schedule S_i. Thus, the maximum score equals 1.0 if and only if the experiment does not take longer than its specified minimum duration.

Start score: Experiments should start as soon as possible. The start score ss_i of experiment E_i with schedule S_i corresponds to $ss_i = \frac{1}{\tau}$. Thus, the maximum score equals 1.0 if and only if the experiment starts at hour $\tau = 1$.

User group score: An experiment E_i should (mainly) involve its *preferred user groups* during its execution. The user group score us_i of experiment E_i corresponds to $us_i = \frac{\sum_1^d coverage(A_i)}{d}$, in which $coverage(A_i)$ is a function returning 1.0 if and only if at least one of the experiment's preferred user groups captures the majority of the sample data at hour i, otherwise 0. The sum is then divided by the experiment's duration d, resulting in the average user group coverage. Consequently, if the coverage criterion is fulfilled for every hour of the experiment's execution, the maximum score equals 1.0.

Combined fitness score: The combined fitness score f of schedule S consisting of n experiments $E = \{E_1, \ldots, E_n\}$ is obtained by summing up the individual scores for every experiment E_i, taking into account the different experiment priorities $\langle p_1, \ldots, p_n \rangle$ and weighting of the scores $\langle w_{ds}, w_{ss}, w_{us} \rangle$. Therefore, we transform our three objectives into a single scalar objective using the weighted-sum strategy, thus leading to a single-objective optimization problem. The weights sum up to 1, thus, the fitness score of a schedule S is in the range 0 to 1. Prioritization allows favoring some experiments over others.

$$f = w_{ds} * \frac{\sum_1^n ds_i * p_i}{\sum_1^n p_i} + *w_{ss} * \frac{\sum_1^n ss_i * p_i}{\sum_1^n p_i} + w_{us} * \sum_1^n us_i * p_i$$

D. Constraints

For a schedule S to be considered valid, four constraints have to be fulfilled. We distinguish between experiment constraints and overarching constraints. The former checks on experiment level (i.e., checking S_i of E_i), the latter requires checking the entire schedule S. Experiment constraints involve checking for valid business-driven experiments, checking that experiments collect sufficient data, and that they are not interrupted, the overarching constraint ensures that we schedule not more resources than available.

Valid business experiments: A schedule S_i of a business-driven experiment E_i (i.e., experiment type $=$ `business`) is valid if and only if it involves the same user groups during all hours of its execution, i.e., the user groups with $consumption \geq 0$ in every $A_j \in A$ of the schedule S_i's assignment A are the same. The reason for this constraint is that business experiments measure a certain effect for particular user groups, often for a long period, therefore switching user groups during the execution would skew results. However, for a regression-driven experiment testing, for example, the scaling capability of a new service, it generally does not matter whether user groups change within the experiment.

Sufficient data points: This constraint validates that $consumedTraffic(A_x) \geq c_x$ for every hour $x = \langle 1, \ldots, d \rangle$ of the experiment E_i with schedule S_i, duration d, and assignments $A = \langle A_1, \ldots, A_d \rangle$. The function $consumedTraffic(A_x)$ returns the total number of users that are assigned to the experiment at hour x taking into account the traffic that is expected according to the underlying traffic profile (e.g., see an example profile for a user group in Figure 3). Thus, it is checked whether the minimum required sample size c_x (either constant or gradual) is met for every hour.

Non-interrupted experiments: This constraint ensures that experiments continuously collect data throughout their execution. Thus, there does not exist an assignment A_i consuming zero traffic and there has to be an assignment $A_i \in A$ for every $i = \langle 1, \ldots, d \rangle$.

Sufficient traffic available: This overarching constraint ensures that at a time slot x there is no user group across all schedules $S = \{S_1, \ldots, S_n\}$ such that the total traffic consumption within a user group is more than 100%. For every schedule S_i we consider the user groups of assignment A_j with $\tau + j - 1 = x$.

V. Approach

In this section we look at approaches to generate valid experiment schedules as solutions for the presented optimization problem. First, we start with a genetic algorithm, followed by random sampling, local search, and we conclude with simulated annealing as a slight modification of local search.

A. Genetic Algorithm

Genetic algorithms (GA) [29] group candidates, typically called individuals, in populations. The basic idea of genetic algorithms is to mimic an evolutionary process in which the best-suited candidates in each generation are selected and used as basis for the next generation of solutions. Starting with an initial population of multiple individuals (i.e., different valid solutions of the optimization problem), these selected candidates evolve through multiple generations in which mutation and crossover operations are applied and after several generations of reproduction those individuals that inherited superior properties become dominant. The reproduction within each generation consists of the following basic steps for genetic algorithms as presented, for example, by Harman [30]. In our case an additional *repair* step is added (see Section V-A5).

1) Parent selection
2) Crossover
3) Offspring mutation
4) Repair
5) Fitness and validity evaluation
6) Next generation selection

The initial population is created using random sampling (see Section V-B for details). Individuals are represented as chromosomes (see Section V-A1) and a fitness function serves as basis for their assessment (see Section IV-C). Individuals

for the next generation are primarily created using crossover and mutation operations, but a small set of the best individuals (*ELITISM_SIZE* parameter) of the previous generation is also passed on to the next generation unchanged (step 6). The GA stops after a specified number of generations, or if one individual solution reaches the desired level of fitness.

All of the presented approaches (i.e., genetic algorithm, local search, simulated annealing) use random sampling as starting point and the chromosome structure described in the following to represent solutions for the optimization problem.

1) Chromosome Representation: A chromosome, i.e., a solution of the optimization problem, is represented as shown in Figure 1. We rely on value encoding and a chromosome corresponds to a schedule S defined in Section V, i.e., a chromosome consists of multiple schedule genes $S_i \in S$ corresponding to their experiments $E_i \in E$ (top layer in Figure 1). Every schedule gene S_i further contains assignment genes $A_j \in A$ and a gene encoding the start hour of the schedule S_i (middle layer in Figure 1).

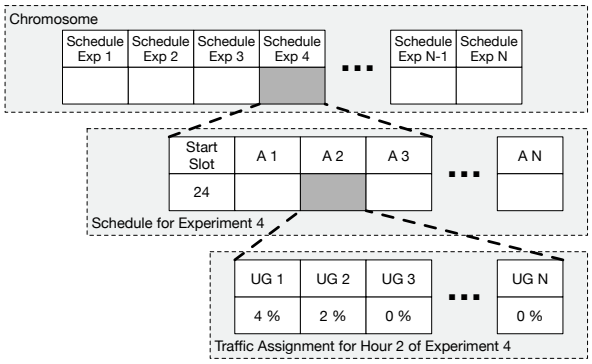

Fig. 1. Chromosome representation using value encoding

Every assignment gene further contains a gene for every single user group (UG). This is used to encode how many users of a certain user group are assigned to the experiment E_i at the respective hour. In Figure 1, bottom layer, at hour 25 (start time slot (τ) $24 + 2 - 1$), 4% of the users of user group 1 are assigned to experiment 4.

2) Parent Selection: The selection of individuals (i.e., parents) for reproduction within each generation plays an important role. Selecting only the highest-score individuals could result in reduced genetic diversity and lead to premature convergence, thus there is the chance that the entire population gets "stuck" at a lower quality solution. Hence, we use fitness proportionate selection [31], as it is simple to implement and has been proven to produce acceptable solutions [32]. The fitness of every individual is obtained, those individuals with higher fitness have a higher probability to get selected for reproduction (i.e., crossover and mutation steps).

3) Crossover: Crossover is the process of creating an offspring of two selected parent individuals by applying a crossover operation with a certain probability P_c. In our implementation, when performing a crossover of two individuals *A* and *B*, we compare the fitness on experiment level. Thus, for each experiment $E_i \in E$, we compare the fitness of *A*'s schedule S_{iA} with the fitness of *B*'s schedule S_{iB}. The single schedule with the higher fitness is added to the offspring as visualized in Figure 2. Consequently, in our approach a single offspring is created during crossover and moved to the subsequent mutation step. In case that no crossover happens (regulated by P_c), both parents are passed on unchanged to the mutation step.

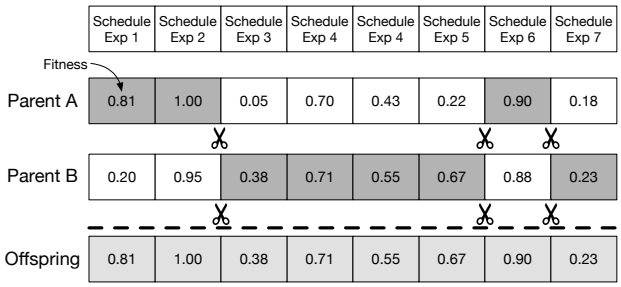

Fig. 2. Crossover example

4) Mutation: Once the offspring is created by applying the crossover operation, the offspring is mutated by performing *NUM_OPS* mutation operations on randomly selected experiments E_i and their respective schedule S_i with a mutation probability P_m. Mutation is important to ensure genetic diversity within the evolving population and helps to avoid convergence to a local optimum. In our implementation, six mutation operations exist that are described in the following.

Move schedule: Pre- or postpones the execution of a certain experiment E_i by *MOVE* hours by mutating the start slot τ in the respective schedule S_i. In case of a reevaluation of the entire experiment schedule, i.e., experiments are already running, some get canceled, and new ones are added to the schedule, those experiments that are already running are omitted for move operations as this would violate the non-interruption constraint.

Shorten/Extend schedule: Shortens (extends) the selected schedule S_i of experiment E_i by *SHORTEN* (*EXT*) hours. In case of shortening, *SHORTEN* assignment genes are removed from S_i's assignment. In case of extension, *EXT* assignment genes are added to S_i's assignment by duplicating the last assignment gene.

Flip user group: Changes which user groups are assigned to the selected experiment E_i, either for the entire schedule S_i (i.e., all $A_i \in A$), or for a certain range (u, v) within the schedule (i.e., $A_k \in A, 1 \leq u \leq k \leq v \leq d$). Takes randomly a used user group ug_x of the experiment's schedule S_i (or from within the range) and replaces it with another randomly retrieved user group ug_y. If the new user group ug_y was already used within S_i, the traffic of ug_x is added to ug_y's traffic. In case of schedule reevaluation, already running business-driven experiments are excluded from the flip operation. In addition, the flip range operation is not applied on business experiments as this would lead to a constraint violation.

Add/Remove user group: Similar to *flip user group*, instead of replacing a user group for the entire schedule S_i (or for a certain range within the schedule), an unused user group is added, or a used user group is removed as long as there

is at least one user group left. The same conditions apply for schedule reevaluation involving running business experiments and range operations on business experiments.

5) Repair: Mutating a schedule S_i of an experiment E_i has a direct effect on the number of data points collected during its execution. In order to ensure that enough data points are collected to fulfill the validity constraints (e.g., after a user group is removed from the experiment), a *repair action* is executed after the mutation step. The repair action adjusts for every mutated schedule S_i every single assignment $A_j \in A$ in such a way that $consumedTraffic(A_j) \geq c_j$ is fulfilled. The repair action distributes the required data points for every c_j across the user groups used by A_j. This is achieved by considering the estimated traffic within the user groups at a specific time slot x from the underlying traffic profile and the required sample size for this specific hour c_x which itself depends on the experiment's sample size and its schedule's duration. To deal with the uncertainty regarding the estimated traffic profile, we introduce a buffer (*BUFFER* parameter) such that slightly more (e.g., 0.5%) traffic is consumed than required.

B. Random Sampling

Random sampling (RS) [33] tries to find valid solutions by creating individuals purely by chance. Our RS approach makes use of the fitness function for assessing the individuals and the constraints for checking their validity as presented in Section IV. A set of *POP_SIZE* individuals for n experiments $E = \{E_1, \ldots, E_n\}$ to be scheduled is created as follows. For an individual solution, randomly take an experiment E_i to be scheduled. Create a schedule S_i with a random start time, a random selection of one or two user groups out of the pool of existing user groups, and a random duration d that is larger or equal than the experiment's minimum duration. Then create d assignment genes such that the minimum required sample size of E_i is reached during the course of the experiment on the selected user groups on the estimated traffic profile. If the created schedule S_i is valid, then it is added to the overall schedule S and the next experiment E_j is picked for scheduling. The random schedule creation is repeated until a valid schedule for every picked experiment is found. After *POP_SIZE* valid individuals are created, the one individual with the highest fitness score is chosen as the result of RS. Strictly speaking, as we pick one experiment after the other and only proceed when a valid schedule is found, our approach is not "pure" random sampling but rather categorized as systematic random sampling [33].

C. Local Search

The local search (LS) algorithm starts with the best individual generated by random sampling and iteratively tries to optimize it by applying the same mutation operations as with the genetic algorithm, followed by the same repair step after each iteration. Again *NUM_OPS* mutation operations are performed on randomly selected experiments E_i and their respective schedule S_i. If the newly generated neighbor resulting from the mutation is an invalid solution, the mutation is reset and the process is repeated until a valid solution is found. After this step, the resulting valid neighbor is compared to the current solution, if the neighbor's fitness score is higher, then the neighbor becomes the current solution. This process is then repeated *NUM_ITERATIONS* times and the final "current" solution returned.

D. Simulated Annealing

The main issue of local search algorithms is that — by design — they get stuck in a local optimum from which no further improvements are possible. Simulated annealing (SA) [33] as a variant of local search algorithms tries to overcome this issue by applying a technique simulating the physical process of annealing in metallurgy. Transferred to our optimization problem and in contrast to our local search implementation, neighbor solutions with worse fitness than the current solution have a certain probability to get accepted, thus the likelihood to run into a local optimum is reduced. The likelihood to accept worse solutions is tied to the current temperature. Initially the temperature is high, thus the algorithm is more likely to accept neighbor solutions with a lower fitness score than the current solution. After every iteration the temperature slowly decreases by a cooling factor, thus the acceptance of worse solutions is less likely. The process of finding valid neighbors and optimizing them using the mutation operators is exactly the same as in our local search implementation, just with the slight addition that the acceptance criterion is added.

VI. Evaluation

For the evaluation of the capabilities of the previously discussed approaches, we implemented them in Java and we assessed them in three aspects: (1) maximum fitness scored for a specific set of experiments, (2) comparison when running an increasing amount of experiments at the same time, and finally, (3) dealing with the reevaluation of an existing schedule. Before we dive into the evaluation, we briefly describe the setup we used as basis for the evaluation.

A. Setup

The setup involves the description of the used traffic profile, the experiments created (in different sizes), how the various algorithms were calibrated, and finally, on which hardware we executed the evaluation runs.

1) Traffic Profile: For our evaluation we mimicked a real traffic profile. We used GitLab's public monitoring tooling[1] and extracted the hourly interaction of users based on the number of returned HTTP status codes for the months January and February 2018. This total traffic per hour served as our baseline and we reserved 10% traffic to serve as the control group being not involved in any experiment. For our evaluation scenario, we divided the remaining traffic into five user groups: group 1 (40% traffic, simulating logged off users), group 2 (10%, paying single license users), user group 3 (20%, free single users), user group 4 (15%, paying company license

[1] https://monitor.gitlab.net/dashboard/db/fleet-overview

users), and group 5 (15%, free company users). To simulate longer running experiments we replicated the two month period to get a twelve month profile.

2) Experiments: We created a baseline of 10 experiments. This involved six regression-driven experiments (two with gradual, four with constant consumption) and four business-driven experiments (constant consumption). Their minimum duration ranged from a single day up to 18 days. As basis for our experiments we used the durations reported by Kevic et al. [1] for Microsoft Bing. To evaluate the algorithms under different scenarios, we created three variations of our baseline: with low, medium, and high required experiment sample sizes (*RESS*), i.e., how many data points does an experiment need to collect to reason about a certain effect. The baseline with low *RESS* requires 15 million data points in total (i.e., the sum of the *RESS* of the 10 experiments), with medium *RESS* 30 million, and with high *RESS* 55 million.

To evaluate the algorithms on different numbers of experiments running in parallel, we used the baseline of 10 experiments and duplicated them with a step size of 5 experiments to create sets with up to 70 experiments. This resulted in sets of $10, 15, 20, 25, \ldots, 70$ experiments, each set in 3 variations with low, medium, and high *RESS*. For example, 70 experiments with high *RESS* require to collect $55 * 7 = 385$ million data points in total.

Figure 3 demonstrates the effect of the different *RESS* variants when scheduling 30 experiments. The high number of required data points leads to a longer schedule in case of the high *RESS* variant. There is simply not enough traffic available within user group 3 to host all experiments in parallel at the same time. Further, Figure 3 depicts the effect of the gradual experiments and how the numbers of users assigned to these experiments increase during their execution.

3) Calibration: To calibrate the algorithms we followed an iterative exploratory parameter optimization procedure with 25 experiments with low *RESS*. We increased the population size and the number of generations for the GA starting from 10 in steps of 10 until we reached 100. The number of iterations for LS and SA was evaluated for 1,000 to 10,000 iterations (step 1,000). Crossover probability was increased from 70% to 100% (step 5%), and mutation probability from 10% to 100% (step 10%). The number of the executed mutation operations *NUM_OPS* is defined as being dependent on the number of experiments to schedule (e.g., 10 experiments and *NUM_OPS* of 20% leads to 2 mutation operations). We tested *NUM_OPS* from 5% to 30% (step 5%).

Based on this procedure, we decided for a population size of 40, 90 generations, a crossover probability of 90%, a mutation probability of 50%, *NUM_OPS* of 15%, *ELITISM_SIZE* of 5, and 3000 iterations (LS and SA). For the sample with 25 experiments, SA achieved the best results with a starting temperature of 0.007 and temperature decrease of 1% per iteration. We further set *MOVE* to 48 hours, *SHORTEN* and *EXT* to 6 hours. For obtaining a scalar fitness value we used the weightings $\langle w_{ds} = 0.4, w_{ss} = 0.4, w_{us} = 0.2 \rangle$.

4) Hardware: We conducted the evaluation on the Google Compute Engine public cloud service. We used custom Intel Skylake instances with 4 vCPUs and 4.75 GB memory running Debian 9 and OpenJDK 8.

B. Maximum Fitness

Given a set of 15 experiments with medium *RESS*, the goal of this aspect of evaluation is to identify the maximum fitness score we can obtain for any of the discussed algorithms. Further, we want to identify how stable the results are, i.e., we repeat the execution of each algorithm 20 times. In contrast to the other aspects of the evaluation (i.e., stepwise increase and reevaluation) that exactly use the findings of the calibration, we use 150 generations for the genetic algorithm (GA) and 5000 iterations for local search (LS) and simulated annealing (SA). The reason is to give the algorithms more time to optimize their results, while the calibration results (90 generations, 3000 iterations) were a trade-off between fitness score and execution time, especially taking effect for the stepwise evaluation.

To have a fair comparison of the algorithms, for every of the 20 repetitions, we create an initial population using RS that is then also used for the respective runs of the GA, LS, and SA. We measure the execution time for every run, including the time it takes to generate the initial population.

TABLE II
STATISTICS FOR SCHEDULING 15 EXPERIMENTS WITH MEDIUM *RESS*

Statistic	RS	GA	LS	SA
Mean exec. time (min)	0.96	19.88	26.64	26.04
Mean fitness	0.41	0.88	0.86	0.86
Max fitness	0.45	0.96	0.92	0.93
SD fitness	0.02	0.04	0.04	0.05
Mean ds	0.67	0.97	0.84	0.85
Mean ss	0.02	0.74	0.82	0.81
Mean us	0.70	1.00	0.99	0.99

Figure 4 visualizes the resulting fitness scores in form of violin plots. Table II provides additional statistics. The GA, LS, and SA implementations optimize the fitness score of RS by a factor ≥ 2. GA achieves slightly better fitness scores than LS and SA in less execution time (20 vs. 26 minutes). The achieved fitness scores are relatively stable with a SD of about 4%. Breaking down the combined fitness score into the individual scores (duration score ds, start score ss, and user group score us) reveals that the GA reaches almost the absolute minimum duration (ds 97%), while LS and SA are better when it comes to letting the experiments start as early as possible (ss $\sim$80%). When analyzing how the scores for the best run (i.e., max fitness) evolve, we notice that in case of GA, no further optimizations are performed after 140 generations, in case of LS, the scores are stable after 4800 iterations, and only in case of SA, the scores keep changing even at iteration 5000.

C. Dealing with Multiple Experiments

The goal of this aspect of our evaluation is to identify how the algorithms deal with an increasing amount of experiments

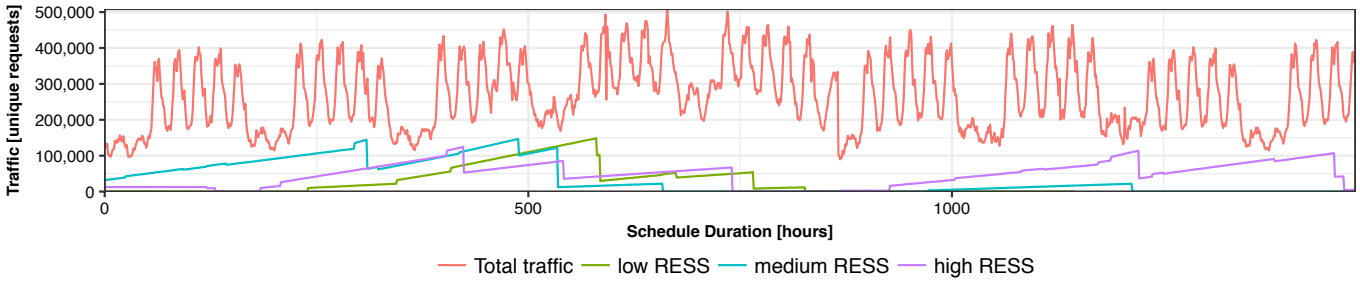

Fig. 3. Traffic profile for user group 3 and traffic consumption of three example schedules (30 experiments each) with low, medium, and high *RESS*.

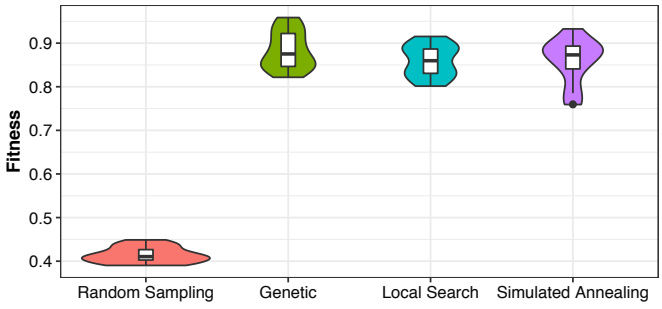

Fig. 4. Comparison of fitness scores for 15 experiments to schedule (GA: 150 generations, LS & SA: 5000 iterations), 20 repetitions in total

to schedule. We conduct evaluation runs in a stepwise manner. Starting with 10 experiments, we increase the number of experiments to schedule by 5 experiments per step, until we reach 70 experiments. Similar to the previous aspect, we are primarily interested in the fitness scores achieved, the overall execution time, and how the single objectives evolve. For every step, we conduct 5 runs with low, medium, and high *RESS* each, i.e., 15 runs per algorithm per step. Again, the GA, local search (LS), and SA implementations use the initial population generated by RS for the respective runs. We use the parameters determined by the calibration runs.

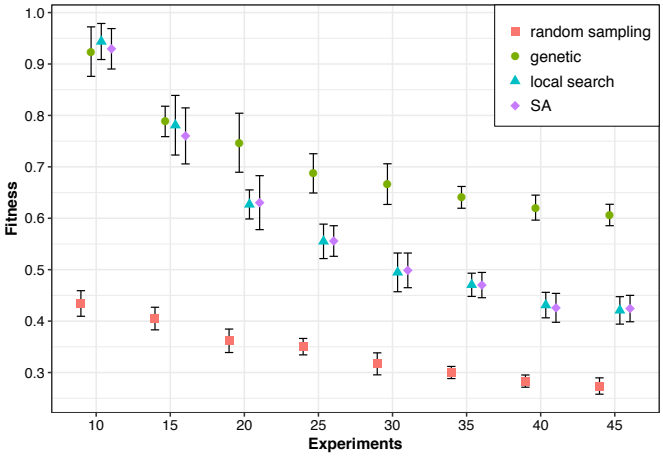

Fig. 5. Fitness scores obtained for different algorithms when number of experiments to schedule is increased. Error bars represent ± one standard deviation.

Figure 5 visualizes the fitness scores achieved and the error bars (± one standard deviation) combining the results of the runs with low, medium, and high *RESS*. In addition, Table III

TABLE III
COMPARISON OF EXEC. TIMES IN MINUTES FOR INCREASING NUMBER OF EXPERIMENTS TO SCHEDULE WITH LOW AND HIGH *RESS*

		Number of Experiments							
	Stat.	10	15	20	25	30	35	40	45
RS low	Mean	0.1	0.7	1.6	3.7	6.6	16.4	18.1	42.5
low	SD	0.0	0.0	0.1	0.4	0.6	3.4	0.9	14.4
high	Mean	0.2	1.1	2.3	5.0	9.4	14.5	25.7	43.9
high	SD	0.0	0.1	0.1	0.3	0.4	1.0	3.4	14.3
GA low	Mean	2.9	9.5	14.2	26.4	36.9	69.7	74.4	129.1
low	SD	0.2	1.2	0.5	1.4	2.3	10.6	5.0	23.9
high	Mean	5.5	14.5	24.3	45.4	60.6	86.1	110.5	178.5
high	SD	0.7	0.9	1.6	1.6	4.8	6.6	6.8	21.0
LS low	Mean	3.9	14.9	32.0	54.9	93.9	168.4	204.3	517.2
low	SD	0.7	1.7	3.4	5.0	6.7	18.0	13.1	321.6
high	Mean	6.6	20.9	47.6	103.2	153.0	194.3	280.2	416.5
high	SD	1.3	3.4	8.2	10.6	28.2	28.2	38.3	46.7
SA low	Mean	3.9	13.8	32.4	57.6	92.6	169.9	204.7	586.2
low	SD	0.6	1.4	1.2	2.8	4.3	7.9	22.4	355.6
high	Mean	7.7	21.1	49.9	104.2	159.2	200.0	273.7	453.8
high	SD	2.5	3.0	9.7	16.2	34.0	31.3	27.1	126.3

outlines execution behavior, i.e., how long it took to generate the schedules. For space reasons, we omit the results on medium *RESS*. As a single run for LS and SA took up to 10 hours for 45 experiments (with high standard deviations), we decided to cut the evaluation at this point.

The GA outperforms the other approaches for 20 and more experiments to schedule. This does not only apply for the achieved fitness scores (e.g., 40 experiments with high *RESS*: GA reaches 62%, SA 42%, and LS 43%), but also when it comes to execution behavior. While it takes the GA on average 110 minutes to schedule 40 experiments with high *RESS*, LS and SA take almost three times as long on average (280 and 274 minutes). The GA implementation was able to finish scheduling 70 experiments (with low *RESS*) within 8 hours. Similar to the previous evaluation, the achieved fitness scores are quite stable. The error bars in Figure 5 are mainly driven by the slightly different results (± 5 – 6%) of the runs with low, medium, and high *RESS*. Runs with low *RESS* achieve better results. Inspecting only runs within a certain stepsize and within the same *RESS*, the standard deviation of the achieved fitness scores is rarely larger than 3%.

Breaking down the fitness score into individual scores, we notice that the user group score (i.e., scheduling on preferred user groups) is in almost all cases ≥ 98% (except random sampling in which no optimization happens). The GA is strong

when it comes to keeping the experiment's execution duration short, the duration score ds is on a high level throughout the various step sizes and decreases only from 98% when scheduling 10 experiments to 83% when scheduling 45 experiments with high *RESS*. In contrast, SA and LS reach a ds of 82% when scheduling 15 experiments and the score drops below 50% when scheduling 40 and more experiments (for all *RESS* variants). Similar to our previous observation, LS and SA begin with higher start scores (i.e., running more experiments right at the schedule's launch), but with an increasing number of experiments to schedule the scores drop below 25% with 25 experiments or more. The start scores of the GA are worse in the beginning (e.g., 10 and 15 experiments), but the decline throughout the various stepsizes is smaller.

D. Reevaluating an Existing Schedule

One essential requirement for our approach is that the implementations are able to deal with the reevaluation of a schedule, i.e., taking into account experiments that (1) finished within the already executed period, (2) got canceled, and (3) are added to be scheduled as well. Further, reevaluation means that the required sample sizes of running experiments (i.e., *RESS*) are adjusted according to the actual data points that were captured until the moment of the reevaluation. Thus, depending on the real traffic situation, an experiment's duration might need to be adapted.

To test this behavior of our implementations, we select the best resulting schedule of the GA from the previous evaluation step with 30 experiments and medium *RESS*. The schedule's fitness value is 74% (duration score 88%, user group score 100%, start score 47%). The reevaluation is conducted after 72 hours. Out of the initial 30 experiments, 3 are canceled, 3 finished within the 72 hours, and 5 new experiments with medium *RESS* are added.

We conduct 10 evaluation runs in total. Again, the resulting population of every run of RS is used by the respective GA, LS, and SA runs. Random sampling in the context of a reevaluation is special as one individual within the population is based on the existing schedule, taking into account already existing optimizations. For the newly added experiments within this individual the usual sampling process applies.

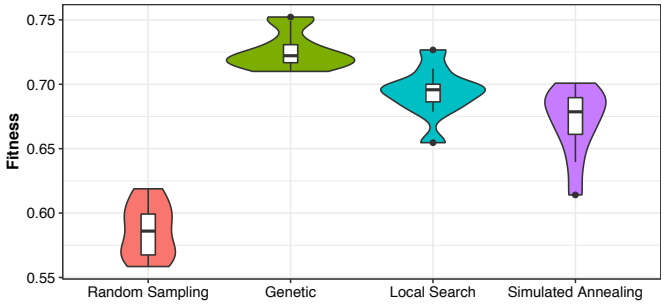

Fig. 6. Fitness scores after reevaluation (10 runs): (re-)scheduling 29 experiments in total (3 discarded, 3 finished, and 5 new experiments)

Figure 6 again shows the achieved fitness scores using violin plots. The fitness scores of the resulting schedules are slightly below (1% in case of GA) the values of the original schedule, but still in a similar range as the evaluation runs of the previous aspect. The gap between the individual approaches (i.e., GA 73% fitness on average, LS 69%, and SA 67%) is much smaller than in the previous evaluation runs with a similar number of experiments to schedule (e.g., 30 experiments, medium *RESS*: GA 68% on average, LS 50%, SA 51%). The reason is that both LS and SA benefit from an already optimized schedule with an especially high duration score ds. The execution time is on a similar level than for scheduling 30 experiments: 29 minutes on average for the GA, and 52 minutes on average for both LS and SA.

VII. Discussion

We now briefly reflect on and discuss the implications of our results.

Scheduling as Part of a Release Pipeline. As observed during our evaluation, for a smaller number of experiments (i.e., 10 and 15) the achieved fitness scores of the GA, LS, and SA implementations are on a similar level. However, when it comes to a larger number of experiments, the GA implementation not only outperforms the other approaches in the fitness scores, but also drastically in execution time. This is especially of interest when we envision scheduling and (re-)scheduling of already running experiments (which achieves stable results as demonstrated in the final aspect of our evaluation) to become an active part in a release pipeline, e.g., the scheduling is triggered as soon as source code changes pass the quality assurance phases. Clearly, the maximum acceptable execution time for scheduling to become part of a release pipeline depends on the release frequency of a company, but for example an execution time of 40 minutes to schedule a set of 30 experiments on cheap public cloud instances is a promising result. Further, due to the nature of the genetic algorithm (i.e., offspring for the next generation is created independently) a higher level of parallelization is possible compared to the LS and SA implementations. Thus, we expect that stronger computing machinery could even decrease the time needed to find suitable solutions.

Importance of Calibration. It is not a straightforward task to tune multiple parameters to achieve acceptable results in various execution scenarios. This can be especially observed for our results of SA. Even though there are "only" two parameters to tune, in most of our evaluation runs SA performed slightly worse than its counterpart local search. The reason is that the starting temperature and the cooling factor were calibrated for a set of 25 experiments. Consequently, we would need to fine-tune these parameters for different numbers of experiments to achieve better results. Another factor, not only influencing the results of SA, is the weighting of the three fitness scores (i.e., objectives). We have observed that the user group score achieves very high values (rarely below 98%) across our evaluation runs. This could be an indicator that the weighting could be decreased to better optimize for the other two objectives. In cases for which scheduling the preferred

user group is of absolute importance this could be ensured by choosing a higher experiment priority.

Crossover and the Destruction of Valid Schedules. One of the important steps that help genetic algorithms avoiding the traps of a local minimum or maximum is its crossover operation. In our implementation, crossover returns a single offspring and this offspring is created in a "greedy" way. An potential effect of this setup on the GA's results is that the duration scores are consistently higher compared to LS and SA implementations. The downside of this approach is that during the process of reproduction the validity of the schedule and thus the overarching constraints are not taken into account. Consequently, many created children are invalid and thus thrown away. We experimented with multiple different strategies, conservative ones such as coin flips on whether to include a schedule for an experiment from the first or second parent, and "smarter" ones such as trying to preserve how user groups are distributed. However, all of these alternative strategies were outperformed by the "greedy" variant. However, we still believe that there is space to improve how offspring is created during the crossover process by better taking validity constraints into account.

VIII. Threats to Validity

We now the discuss issues that form a threat to the validity of our results.

Construct Validity. The main threat regarding construct validity is that our definition of scheduling continuous experiments as an optimization problem is not an adequate representation of the domain. This is especially the case for the definition of constraints (i.e., the validity of a schedule) and how we determine the fitness of a schedule (e.g., fitness function and the used weighted-sum approach). We mitigated this threat by strongly relying on reported empirical work (e.g., Kevic et al. [1], Schermann et al. [14], and Lindgren and Münch [17]). Another threat regarding the representation is that we limited our approach to scheduling experiments for a single service. This is acceptable as long as experiments do not involve or target more than one component or service. Otherwise, this would require additional overarching constraints that we plan to address in future work. Further, the choice and the implementation of the algorithms to identify solutions for the presented optimization problems have influence on the results. There could exist other heuristics that provide better results than the implemented algorithms. Further, there might be better ways to tailor local search and simulated annealing implementations rather than reusing the genetic algorithm's mutation operations.

Internal Validity. Threats to internal validity involve potentially missed confounding factors during result interpretation (e.g., when breaking down achieved fitness scores into the individual scores) and that the calibration of the algorithms affects the results. We mitigated this threat by performing various calibration runs with different parameter settings on 5 user groups and 25 experiments to schedule. However, as discussed earlier, calibration was a trade-off between fitness scores and execution time. Different and more fine-tuned parameters on different numbers of experiments to schedule or user groups might result in better results for the various algorithm implementations. In addition, prior work (e.g., Arcuri and Fraser [34]) raise concerns that (hyper-)parameters of search-based techniques have strong impact on results and conclusions of studies.

External Validity. Threats to the external validity concern the generalization of our findings. Even though we used a real world traffic profile, we mimicked the distribution of users into multiple user groups which could influence our results. Further, using traffic profiles with different user interaction patterns could also lead to different results among the various algorithms. Our evaluation only relied on self-generated experiments, even though we created them based on knowledge (e.g., duration of experiments) gathered from various reports in literature (e.g., Kevic et al. [1], or Fabijan [9]). To mitigate this threat we created multiple scenarios (i.e., experiments with low, medium, and high required sample sizes) and evaluated the implemented algorithms on different numbers of experiments to schedule. We conducted our evaluation in a virtualized environment, i.e., Google Compute Engine. It is possible that the performance variations inherent to public clouds [35] have influenced the results. To mitigate this risk, we repeated every evaluation run at least five times.

IX. Conclusion

We formulated the problem of scheduling continuous experiments (i.e., which users participate in which experiments and when to run experiments) as an optimization problem involving three objectives. (1) experiments should not take longer than necessary to collect the required data points; (2) experiments should start as soon as possible to avoid any delay of ongoing development work; and (3), experiments should be executed on the preferred user groups to measure a certain effect. Using a weighted-sum approach we transformed these objectives into a single-objective optimization problem and we implemented a genetic algorithm, random sampling, local search, and simulated annealing to generate solutions.

Our evaluation on multiple aspects has shown that starting from 15 or more experiments to schedule, the genetic algorithm not only outperforms the other approaches when it comes to the fitness scores of the identified solutions (e.g., up to 19% for 40 experiments with high required experiment sample sizes), but also in terms of the execution time needed to find these solutions (e.g., almost a factor three for 40 experiments with high required experiment sample sizes). Currently, our approach is limited to experiments targeting a single service or component and the crossover operation does not take the validity constraints into account during reproduction. We plan to address both aspects in future work.

Acknowledgment

The research leading to these results has received funding from the Swiss National Science Foundation (SNF) under project Whiteboard (no. 149450).

References

[1] K. Kevic, B. Murphy, L. Williams, and J. Beckmann, "Characterizing experimentation in continuous deployment: A case study on bing," in *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, ser. ICSE-SEIP '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 123–132. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2017.19

[2] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and Deployment at Facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, 2013.

[3] D. Tang, A. Agarwal, D. O'Brien, and M. Meyer, "Overlapping experiment infrastructure: More, better, faster experimentation," in *Proceedings 16th Conference on Knowledge Discovery and Data Mining*, Washington, DC, 2010, pp. 17–26.

[4] C. A. Gomez-Uribe and N. Hunt, "The netflix recommender system: Algorithms, business value, and innovation," *ACM Transactions on Management Information Systems (TMIS)*, vol. 6, no. 4, p. 13, 2016.

[5] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," *Software, IEEE*, vol. 32, no. 2, pp. 50–54, Mar 2015.

[6] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann, "Online Controlled Experiments at Large Scale," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*. New York, NY, USA: ACM, 2013, pp. 1168–1176. [Online]. Available: http://doi.acm.org/10.1145/2487575.2488217

[7] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010.

[8] Y. Xu, N. Chen, A. Fernandez, O. Sinno, and A. Bhasin, "From infrastructure to culture: A/b testing challenges in large scale social networks," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '15. New York, NY, USA: ACM, 2015, pp. 2227–2236. [Online]. Available: http://doi.acm.org/10.1145/2783258.2788602

[9] A. Fabijan, P. Dmitriev, H. H. Olsson, and J. Bosch, "The evolution of continuous experimentation in software product development: From data to a data-driven organization at scale," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, May 2017, pp. 770–780.

[10] R. Kohavi, A. Deng, R. Longbotham, and Y. Xu, "Seven rules of thumb for web site experimenters," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14. ACM, 2014.

[11] M. T. Rahman, L.-P. Querel, P. C. Rigby, and B. Adams, "Feature Toggles: Practitioner Practices and a Case Study," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 201–211. [Online]. Available: http://doi.acm.org/10.1145/2901739.2901745

[12] G. Schermann, D. Schöni, P. Leitner, and H. C. Gall, "Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies," in *Proceedings of the 17th International Middleware Conference*, ser. Middleware '16. New York, NY, USA: ACM, 2016, pp. 12:1–12:14. [Online]. Available: http://doi.acm.org/10.1145/2988336.2988348

[13] G. Schermann and P. Leitner. (2018) Paper Online Appendix. https://github.com/sealuzh/icsme18-fenrir.

[14] G. Schermann, J. Cito, P. Leitner, U. Zdun, and H. C. Gall, "We're doing it live: A multi-method empirical study on continuous experimentation," *Information and Software Technology*, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584917302136

[15] G. Schermann, J. Cito, and P. Leitner, "Continuous experimentation: Challenges, implementation techniques, and current research," *IEEE Software*, vol. 35, no. 2, pp. 26–31, March 2018.

[16] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, "Controlled experiments on the web: survey and practical guide," *Data Mining and Knowledge Discovery*, vol. 18, no. 1, pp. 140–181, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10618-008-0114-1

[17] E. Lindgren and J. Münch, "Raising the Odds of Success: the Current State of Experimentation in Product Development," *Information and Software Technology*, vol. 77, pp. 80 – 91, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300647

[18] T. Crook, B. Frasca, R. Kohavi, and R. Longbotham, "Seven pitfalls to avoid when running controlled experiments on the web," in *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '09. New York, NY, USA: ACM, 2009, pp. 1105–1114. [Online]. Available: http://doi.acm.org/10.1145/1557019.1557139

[19] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," in *Proceedings of the 5th International Conference on Software Engineering and Applications*, 1992, pp. 625–636.

[20] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070536

[21] D. Romano, S. Raemaekers, and M. Pinzger, "Refactoring fat interfaces using a genetic algorithm," in *2014 IEEE International Conference on Software Maintenance and Evolution*, Sept 2014, pp. 351–360.

[22] F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective software effort estimation," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 619–630. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884830

[23] W. Fu, T. Menzies, and X. Shen, "Tuning for software analytics: Is it really necessary?" *Information and Software Technology*, vol. 76, pp. 135 – 146, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584916300738

[24] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto, "The impact of automated parameter optimization on defect prediction models," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[25] S. Frey, F. Fittkau, and W. Hasselbring, "Search-based genetic optimization for deployment and reconfiguration of software in the cloud," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 512–521. [Online]. Available: http://dl.acm.org/citation.cfm?id=2486788.2486856

[26] G. Tamburrelli and A. Margara, "Towards Automated A/B Testing," in *Proceedings of the 6th International Symposium on Search-Based Software Engineering (SSBSE)*, ser. Lecture Notes in Computer Science, vol. 8636. Springer, 2014, pp. 184–198.

[27] M. B. Wall, "A genetic algorithm for resource-constrained scheduling," Ph.D. dissertation, Massachusetts Institute of Technology, 1996.

[28] K. Deb, "Multi-objective optimization using evolutionary algorithms: an introduction," *Multi-objective evolutionary optimisation for product design and manufacturing*, pp. 1–24, 2011.

[29] C. M. Fonseca, P. J. Fleming *et al.*, "Genetic algorithms for multiobjective optimization: Formulation, discussion and generalization," in *Icga*, vol. 93, no. July. Citeseer, 1993, pp. 416–423.

[30] M. Harman, "Software engineering meets evolutionary computation," *Computer*, vol. 44, no. 10, pp. 31–39, 2011.

[31] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," ser. Foundations of Genetic Algorithms, G. J. RAWLINS, Ed. Elsevier, 1991, vol. 1, pp. 69 – 93. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780080506845500082

[32] T. Bäck, D. B. Fogel, and Z. Michalewicz, *Evolutionary computation 1: Basic algorithms and operators*. CRC press, 2000, vol. 1.

[33] F. Schoen, "Stochastic techniques for global optimization: A survey of recent advances," *Journal of Global Optimization*, vol. 1, no. 3, pp. 207–228, 1991.

[34] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, Jun 2013. [Online]. Available: https://doi.org/10.1007/s10664-013-9249-9

[35] P. Leitner and J. Cito, "Patterns in the chaos - a study of performance variation and predictability in public iaas clouds," *ACM Trans. Internet Technol.*, vol. 16, no. 3, pp. 15:1–15:23, Apr. 2016. [Online]. Available: http://doi.acm.org/10.1145/2885497