Retrofitting Admission Control in an Internet-Scale Application

Tanmay Chaudhry[1], Christoph Doblander[2], Anatol Dammer[1],
Cristian Klein[3]*, Hans-Arno Jacobsen[2]
[1]*SimScale GmbH, Germany* [2]*Technische Universität München, Germany* [3]*Umeå University, Sweden*

Submission Type: Experience

Abstract

In this paper we propose a methodology to retrofit admission control in an Internet-scale, production application. Admission control requires less effort to improve the availability of an application, in particular when making it scalable is costly. This can occur due to the integration of 3rd-party legacy code or handling large amounts of data, and is further motivated by lean thinking, which argues for building a minimum viable product to discover customer requirements.

Our main contribution consists in a method to generate an amplified workload, that is realistic enough to test all kinds of what-if scenarios, but does not require an exhaustive transition matrix. This workload generator can then be used to iteratively stress-test the application, identify the next bottleneck and add admission control.

To illustrate the usefulness of the approach, we report on our experience with adding admission control within SimScale, a Software-as-a-Service start-up for engineering simulations, that already features 50,000 users.

1 Introduction

Internet-scale applications are expected to be always available. To achieve this, the application needs to automatically scale as required to serve incoming load in a responsive manner [24]. However, with "lean thinking" new features are constantly developed, whose customer uptake is uncertain. Hence, it might not be economically efficient to design new features in a scalable manner from start. The new feature may only serve for discovering user requirements or validate a business hypothesis, hence, scalability may be seen as gold plating or over-engineering. The cost of making it scalable may further increase if legacy code or 3rd-party components are used that were not specifically designed to run in a cloud environment.

Admission control may be employed, to quickly ship a new feature, while minimizing the risk of compromising its business value and avoiding a costly scalability implementation. Admission control[1] consists in "reducing the amount of work, the server accepts when it is faced with overload", for example, by rejecting requests or degrading certain features of the application in a controlled manner [7, 12]. This can be either the newly introduced or existing ones, depending on the importance. For example, a chat application may relax delivering messages in real-time and add a small latency to cope with the overload. Admission control is cheaper to employ, from a resource consumption perspective, than over-provisioning. Also, it has lower risk of introducing bias in business-related metrics, as all users are exposed to the new feature.

Before designing admission control, several questions need to be answered:

- **Actuators:** What features to disable or degrade, and in what order?
- **Sensors:** What conditions should trigger admission control?
- **Coordination:** How to ensure that features are disabled in a controlled order without provoking oscillations?

While admission control is not new, few papers report on deploying it in practice on an Internet-scale, production application with a large code-base, integrating a large amount of legacy code.

In this paper, we share our experience in designing and deploying admission control inside SimScale, a Software-as-a-Service platform for engineering simulations. The application offers pre-processing, numerical simulation and post-processing capabilities, integrating a large number of open-source and commercial software libraries, that were not specifically designed for a cloud environment, which significantly increases the cost of scalability. Furthermore, in spirit of "lean thinking" non-functional properties are often delayed until the user requirements, i.e., the functional properties, become clearer (Section 2).

*Work done while working at SimScale GmbH, Germany

[1]As supported by cited work, we use the most general definition of admission control, that shares similarities with service differentiation and service degradation. We prefer using "admission control" since, fundamentally, our approach admits or reject the execution of certain code.

The contributions of this paper are two-fold:

- We present a methodology to retro-fit admission control into a large code base. At the core of our contribution lies a workload generator that produced tunable user behaviours based on log files. Our approach helps to reduce the size of the transition matrix, while keeping the generated workload realistic. We employ techniques, such as classification, logical states, operations and pre-defined workflows (Section 3).
- We show the benefits of our method when applied to SimScale: Our method discovered two actuators, an internal one, which reduces update rate but otherwise allows users to continue work undisturbed, and an external one, which blocks new users from logging in. The benefits, implications and coordination of the two actuators are discussed in Section 4.

2 Background

In this section, we introduce the necessary background to our contribution, which includes lean thinking and the SimScale Platform, the latter being also used as a running example throughout our contribution. The SimScale platform allows the user to simulate computationally-intensive physics simulations required for product design, like fluid analysis, stress analysis and thermal analysis, all through a web browser.

2.1 Lean Thinking

Start-ups and small companies undergoing rapid growth are operating under constant uncertainty. They have to either develop a product (or set of features) that allow them to acquire new customers, or find the customers that are willing to pay for the current product. To reduce risks and costs associated with this discovery process, the "Lean Startup" book [21] advocates explicitly formulating a business hypothesis – e.g., this feature will increase sales – and producing a Minimum Viable Product (MVP) that validates or rejects the business hypothesis. In practice, the MVP has to be "sufficiently complete" to get some useful feedback from potential customers. Hence, to reduce costs, a company will generally decide to skip implementing non-functional requirements, such as performance, scalability and resilience – they are better delayed to future when the requirements of the new feature are better understood and it becomes clearer how to design the feature taking non-functional requirements into account.

Such is the case at SimScale, where implementing non-functional requirements is further complicated by our unique context. The product integrates a large variety of 3rd-party, legacy components developed over decades, including mesh generators, numerical simulators, linear solvers and post-processors. Large amounts of data need to be transferred between these components and they can only start processing when the whole data is available. This makes it challenging to design a system that features both low-latency and is distributed. Hence, MVPs are developed under the assumption that most components are running on the same large machine, so as to take advantage of disk caches to reduce latency.

Nevertheless, it is desirable to avoid the product becoming "a victim of its own success" and overload, thus compromising both business insight gained through the new feature as well as existing customers. Therefore, some form of non-costly resilience, that does not have to be explicitly designed for, is desirable. Admission control techniques are suitable to reach this goal, as they can be easily retro-fitted without incurring technical debt, and have extensively been studied both in industry [3] and academia [8].

2.2 Running Example: The SimScale Platform

As a running example for our approach and to better understand our challenges, we provide a short introduction into the SimScale platform.

After having created an account, a typical user starts her journey on the platform on the *login* view. After authentication she is presented with the *workspace*, which gives an overview of all the projects. A *project* is a way for the user to group related simulation artifacts, such as geometries, meshes and simulation results.

From the workspace a user may either open an existing project or create a new one, in which case she is presented with the project view. From here, three choices are possible: enter the pre-processing view, the simulation view or post-processing view. In the *pre-processing* view, she can upload a new geometry, set meshing parameters and start a meshing job. Such jobs are executed asynchronously in the background, while the user is kept up-to-date about the status through the *status panel*.

In the *simulation* view, the user can work on a new simulation or edit an existing simulation. A simulation contains a mesh and a set of simulation parameters, such as initial conditions and boundary conditions. The "validate simulation" feature ensures that the simulation is physically feasible and correctly configured. Once the simulation is validated, the user may run a simulation job, whose status is reported alongside meshing jobs.

Once the simulation job finished, the user can inspect the results in the *post-processing* view. This view allows the user to choose what result to visualize, what field of those results (e.g., speed, pressure), what filters to apply

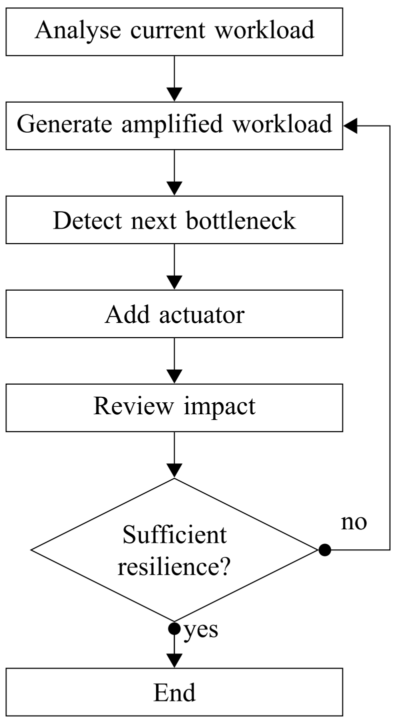

Figure 1: Overview of our approach

(e.g., to display airflow as stream lines) and take screenshots.

3 Approach

An overview of our approach to add admission control is illustrated in Fig. 1. We start by analysing the existing workload of the platform and configuring a realistic workload generator that can arbitrarily amplify the workload to test various what-if scenarios. Then, we detect the next bottleneck in the platform, and decide what actuator to add and in what conditions it should trigger. These steps are done in several iterations until it is decided that sufficient resilience is present in the platform. Most of the software artifacts produced in these steps only incur a one-time cost or can be reused from other development activities. This approach can be applied regularly, depending on how much the user behaviour or the platform have changed.

Below we go into more details within each step, highlight issues specific to SimScale and gathered learnings.

3.1 Realistic User Behavior Modelling

SimScale is used by over 50,000 users spanning a large number of countries and markets. The huge variety of users results in a wide range of usage behavior. This represents a challenge to create a realistic work load. To model the user, we extract significant boundary events from logs of the platform. From these events we fitted a probabilistic Markov model [23]. However, due to the heterogeneity of the users, the state transition matrix became too large and would only lead to a too random workload. In a first approach we tried to merge states but this resulted in a unrealistic workload. Therefore, we used a layered Markov chain, consisting of user classes, user logical states and user operations.

3.1.1 User Classes

Upon initial survey of the logs, we discovered that the workflows of the users are highly heterogeneous. While this was quite expected, it also meant that a simple probabilistic model based on the entire set would result in a user behaviour that is too random to be useful.

To overcome this problem, we decided to **cluster users into classes**, with each class having its own model, so as to minimize the overall variance of each user model and allow a broader exploration of what-if scenarios. For example, one what-if scenario that we wanted to explore is how to restrict the impact of admission control to non-paying users, so as not to affect paying users. As highlighted by the example, this step requires some business insight to predict the kind of what-if scenarios that are of interest.

At SimScale, we chose to classify users as follows:

Customers: Users who have subscription at SimScale. These are mainly characterized by multiple visits to the platform and performing meaningful, goal-driven actions, such as running simulation jobs.

Prospects: Users who behave mostly like a paying user, returning to the platform on multiple occasions and performing meaningful, goal-driven actions. The users however do not possess a subscription.

Players: Users who do not fall into either of the above categories. They show highly random behavior, which mostly does not amount to meaningful, goal-driven actions.

3.1.2 User Logical States

With the users classified, the next step consists in building transition matrices for each user class. However, the SimScale platform exposes a large number of user operations, ranging from simple action like logging in to complex action like setting boundary conditions on simulation variables. Without any form of aggregation, the extracted transition matrix would be so large, that it would provide little in terms of creating a realistic user workflow, as the probabilities on each transition arc would be very low.

To resolve this issue, we decided to aggregate user operations into **logical states**. This represents a coarser view on the kind of task a user is performing, rather than the operation itself. Based on existing business knowledge, we defined the following logical states:

UnAuthorized: The user is about to log in or has logged out.

Workspace: The user is looking at the list of projects, without having selected a particular project to work on.

Project: The project logical state is activated as soon as the user either selects an existing project, or creates a new one. All pre-processing operations on geometries and meshes are included in this state.

Simulation: This state includes configuring a particular simulation, for example, setting boundary conditions and physical contacts.

Job: This state is reached once the user finalizes the simulation and starts a simulation job. The only operation inside this state is the starting of the job itself.

PostProcessor: The user is visualizing simulation results.

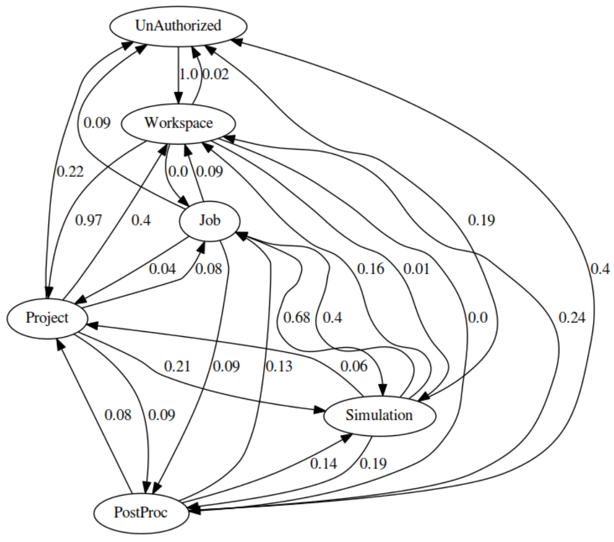

Figure 2: Transition matrix obtained for prospects. The numbers on the arcs represent the probability of the user transitioning from one state to another. The sum of the probabilities on all outgoing arcs from a given state is 1.

3.1.3 User State Transitions

Before computing (logical) state transition matrices for each user class, one must define how the user transitions from one state to another. The transitions refer to chosen user actions. As an example, the transition "Create Project" refers to a state change from "Workspace" to "Project".

Having defined user classes, logical states and trigger operations, one can obtain a transition matrix for each user class by parsing the platform logs, which contains information about all boundary events, such as API calls. Thanks to the previously performed steps, the transition matrices are meaningful and does not generate invalid user workflows (Fig. 2).

Note that, in case of the SimScale platform, using the transition matrix for workload generation as such, may lead the emulated user performing invalid operations, such as trying to post-process results, without having any project in the workspace. To counter-act this, all emulated users are initialized with a set of projects already present in their workspace, similarly to human users on the production platform, whose workspace contains either a few tutorial projects (for new users), or projects previously worked on.

3.1.4 User Operations

User operations can be viewed as a second-level of user behaviour modeling that determine the exact operation (i.e., API call or UI interaction) that the emulated user should perform next, based on the current logical state. To generated relevant user operations in each state, we propose two methods, depending on the class of user.

For the "player" class, a list of operations is generated for each state, each operation being associated with the probability of performing that operation and the think time (i.e., user idle time) after the operation was performed. Although, this simplification does not always generated a valid workflow to interact with the platform, it proved sufficient for generating realistic load for this class of users, as they never launch any kind of jobs.

For the classes prospects and customers, *pre-defined workflows* are required to ensure that the emulated users are capable of running jobs, without being blocked by the platform's validation rules. These validation rules essentially prevent the user from running a simulation that does not make physical sense and is most likely due to human error. Therefore, if an emulated user reaches the Job state, as given by a realisation of the transition matrix, then a pre-defined workflow from one of the imported projects is triggered. This provides a realistic mix of behavior between, e.g., users who entered the platform to run a job and users who entered the platform to inspect past simulation results.

However, the type of job that is run can greatly influence the load on the platform. Fortunately, by analysing the production logs a realistic distribution of job types can be determined. At SimScale, as at many other Software-as-a-Service company, the privacy of the users are of uttermost importance, hence their data cannot be readily used for load testing. Therefore, based on job type, we select one of our sample project that contain the same type of job. We call the list of sample projects with associated probabilities the *project type distribution*.

3.1.5 Integrating the Components

Let us now see how the above-presented user behavior model components are integrated to obtain an emulated user.

1. The emulated user is assigned a class.
2. Using the transition matrix of that class, a complete workflow is generated, starting from the UnAuthorized logical state, up to the next UnAuthorized logical state. This essentially models a whole session for the emulated user, from entering the platform until exiting.
3. If the workflow contains a transition to the Job state, a pre-defined workflow is triggered. The workflow is decided by randomly choosing a sample project according to the the project type distribution.
4. If the workflow of the emulated user does not reach the Job state, operations from the the list of operations associated to each logical state are randomly selected.

To sum up, by employing the techniques of user classes, logical states, user operations and pre-defined workflows, we obtained a user model that is useful for stress-testing. Writing the program which generates such a model based on production logs only incurs a one-time cost, hence, updating the model to reflect latest changes in production is cheap.

3.2 Amplified Workload Generation

Given the realistic user behavior, we can implement a workload generator. The user operations can be cost-efficiently implemented by reusing code produced as part of integration or automated UI testing. Indeed, Quality Assurance (QA) teams generally implement automated UI testing using the *Page Object Design Pattern*, essentially coding one object per page/view, that abstracts information presented on that page as well as actions that can be performed through that page.

At SimScale, we used the Selenium[2] framework, which is useful to programatically drive browser actions, such as filling text boxes or clicking web page buttons. This also showed to be the natural abstraction level for stress-testing the platform, given its Software-as-a-Service nature. Selenium Grid can be used to coordinated a set of Selenium worker machines, hence obtaining a scalable workload generator. Workloads can be amplified in two ways, either by increasing the number of emulated users, or by reducing think times.

Thanks to the user behavior model, that contains enough details to be realistic, but without overfitting on the production logs, we can test various what-if scenarios, such as increased number of users, increased number of a particular user class, increased usage of a feature (user operation) or modify the kind of simulations that users run.

3.3 Bottleneck Detection (Sensor)

The next step consists in defining when the system is overloaded and admission control needs to be triggered. From the user's perspective, the application is overloaded when it feels slow. Studies show that the tolerable waiting time is around 4 seconds [16].

We define two metrics: *Combined Throughput* measures the number of user operations the server successfully completed for all the emulated users, and *Individual User Throughput* measures the average number of operations one emulated user performs per unit time, within the given tolerable waiting time.

The combined throughput is primarily useful for analysing the peak capacity of the server. For an ideally resilient server, the combined throughput should not drop once it reaches the peak, no matter how high the number of incoming requests is. Therefore, this metric helps us to determine the peak capacity of a server, as well as the performance drop it experiences as load is increasing beyond what it can handle at peak capacity.

Individual user throughput focuses on a single user rather than the server as a whole. It measures the drop in quality of experience of a user due to the server's lack of responsiveness, which is overloaded with serving too many users simultaneously. The individual user throughput is complementary to the first metric because a server must not only serve a high number of operations but also make sure that they are completed within the tolerable waiting time.

While the two above metrics are useful to guide admission control and to evaluate its effectiveness, they are unsuitable to trigger admission control, due to the difficulty of measuring them server-side. Indeed, a user operation does not map one-to-one to API calls, hence, an additional server-side module would be necessary to track user operations. Since admission control is supposed to be a last-resort to improve the availability of the platform, we felt uncomfortable implementing such a complex solution.

Therefore, we chose to translate user experience degradation, due to individual user throughput reduction, into a bottleneck resource. This could be either CPU utilization, CPU load, memory utilization or I/O bandwidth utilization. Translating user-centric metrics, as individual user throughput, to system-centric metrics is in general chal-

[2]`http://www.seleniumhq.org/`

lenging [15, 18]. Hence, to reduce complexity, we opted to profile our system offline. This allows determining a system-centric overload condition, such as "when CPU load is above 8 then user experience is degraded", which serves to trigger admission control.

3.4 Circuit Breaking (Actuator)

Once the system-centric overload condition is determined, the next step is to add actuators, also called *circuit breakers*, which is code that disables some resource-hungry code.

If overload is mainly caused by CPU saturation, a recently developed technique called *CPU flame graphs* [6] can help quickly find hotspot codes. In essence, a sampling profiler takes a snapshot of the callstack that the CPU was executing at regular time intervals and summarizes this information in a visually useful way. Various tools exist to generated CPU flame graphs both at process-level, allowing to determine which process is using most CPU, and inside a process, allowing to pin-point the most CPU hungry code (see Fig. 3).

The exact choice of code that is to be adapted for admission control is highly dependent on business objectives. Potential candidates include:

Code used for notifying the user: The delivery of notification or update events, such as job completion, can be delayed without significantly impacting user experience.

Code returning information: Partial information, e.g., the first 10 items of a list, may be returned to reduce CPU demand without significantly impacting user experience.

Optional content: Some content is not required for allowing a user to reach her goals [14], e.g., computing disc quota usage. Such code can be disabled to avoid overload.

As a last resort, if no such code is available, the number of users accessing the platform can be limited, by disabling the ability to log in. Although this leads to some users being unhappy, at least the users who are already on the platform can finish their work.

In the end, the actuator is essentially a control variable between 0 and 1, that probabilistically enables or disables the resource-hungry code. A value of 0 means that the code is not run at all, 1 means that the code is always executed and 0.5 means that the code is executed 50% of the time.

Once the sensor and the actuator are identified, one can apply the methodology presented in [5] to obtain a self-adaptive software system with control-theoretical guarantees. In brief, the relationship between the level of actuation and the effect of the actuation is modeled as a linear relationship with a parameter measured at run-time and a controller can be designed that drives the system to the desired state, without inducing oscillations. In the present case, the desired state is having the actuator close to the maximum value that does not cause overload. It has been proven that these guarantees are valid, even if the actuator is not linear.

3.5 Re-iterate

Once a first actuator is implemented and tested, one needs to re-iterate over the last two steps, to identify the next overload condition and add a new actuator.

Coordinating multiple feedback loops need to be done carefully [25]. For simplicity, we propose to assign each actuator a different range. For example, the first actuator has an effect in the range 0 to 1, whereas the second actuator has an effect from 0.5 to 1. This gives us the ability to control the strictness of admission control, not just in amount but also in type. Hence, allowing concepts such as "last resort" actuators to be used without the danger of triggering them unnecessarily.

If the product or the user base changes significantly, one should regenerate the user behaviour model and implement the required user operations. As highlighted before, these steps incur a high one-time cost, that is quickly amortised in subsequent iterations, as a lot of knowledge and code can be reused. For implementing user operations, artifacts produced by the QA team is readily available.

4 Evaluation

In this section, we evaluate our proposed approach for adding admission control by applying it to the SimScale platform.

4.1 Experimental Setup

As recommended in [9], each developer is assigned a dedicated (virtual) server that contains a scale-down version of the platform, to allow them to work independently and avoid coordination overhead. The experiments presented in this section were performed on a Paravirtualized Xen [2] virtual machine with two virtual CPU cores on a Intel® Core™i7-4770 CPU at 3.40 GHz and 8 GiB of memory.

Each experiment begins with a specific number of *starting users* which simultaneously access the platform. The number of users is increased step by step. When a user exits the platform, a new user is added to the platform.

For amplifying the workload, the think time which users spends between each high level operation is set to zero. Although, reducing the think times between operations

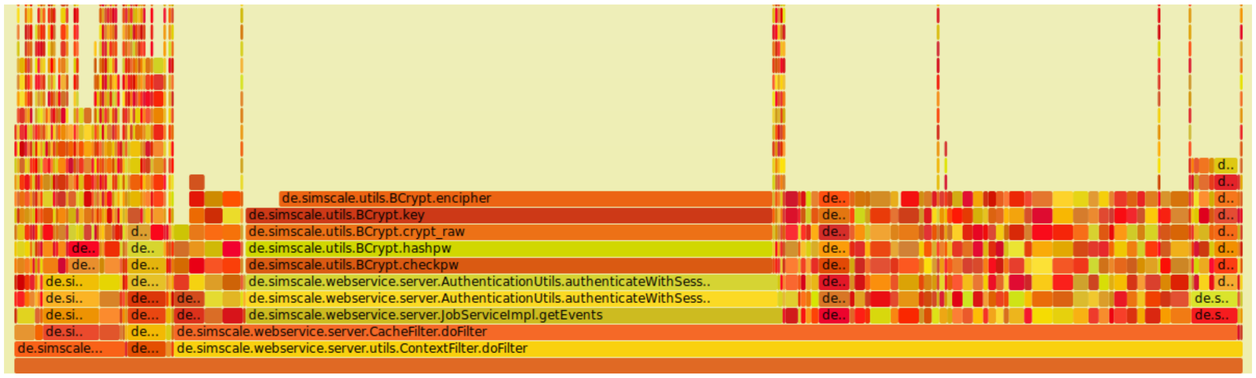

Figure 3: Example of a CPU frame graph obtained during stress-testing on the SimScale platform with an amplified workload. The x-axis represents time, whereas the y-axis represents the call stack. One can quickly identify the Java method that consumes most CPU resources (2nd quarter of the figure, from left to right) and the call stack that lead to it, which helps decide where to place admission control.

Starting users	8 users
User increment	8 users / step
Number of steps	5 steps
Duration of each step	500 seconds
Think times	0 seconds

Table 1: Parameters used for workload generation

takes away a part of the realism of the users, for the purpose of load testing this makes sense, as our primary focus is on load generated by users actually interacting with the platform. This amplifies the number of operations at any given point of time, but does not disturb the ratio of the different types of operations.

The parameters we used are listed in Table 1. We choose those parameters so that no circuit breaking is necessary right at the beginning of the experiment. When the amount of users are increased step by step they react if necessary. We test both the ability of the circuit breakers to deal with overload, as well as their ability to do "no harm" when they are not required.

4.2 Discovered Admission Control

We applied our approach to the SimScale platform and discovered an *internal* circuit breaker, i.e., one that does not limit the number of users on the platform. Furthermore, we added an *external* circuit breaker, that triggers in extreme cases, to limit the number of users on the platform.

Internal breaker The internal circuit breaker is placed around the `getEvents` public API, which allows the browser to pull events from the server, such as job completion, project changes, support messages received, etc. Reducing the time between the occurrence of an event and the notification of the user is desirable for good user experience, but not critical to allow users to interact with the platform. Therefore, we placed a circuit breaker that returns "no events" with a given probability: If the circuit breaker is completely disengaged, events are always returned, whereas if the circuit breaker is fully engaged, 60% of requests for events immediately return an empty event list. This probability is adjusted incrementally based on measured CPU load. Using a model inspired by [20], we perform incremental updates to gradually engage or disengage our breaker. The gradient of the adjustment is determined by the difference between current CPU load and the experimentally found optimum value.

On an individual request level, the circuit breaker reduces server load both by avoiding business logic (e.g., authentication, input validation) and persistence logic (e.g., retrieving events from a database). On a global level, the circuit breaker essentially switches the system from low-latency to high-throughput, by coalescing event retrieval, similarly to hardware interrupt coalescing [1].

Many modern Internet applications use similar mechanism to deliver events to the user through the browser. Therefore, such an internal breaker is applicable to a wide range of applications.

External breaker The external circuit breaker is placed around the login view. The reason for that is that the user cannot call the expensive `getEvents` public API unless she is logged in. If fully disengaged, this breaker allows all users to log in, whereas if fully engaged, it blocks all users. Between these two configurations, it probabilistically allows users to log in, with probability between 0 and 1, depending on resource availability. This breaker triggers similarly to its internal counterpart but with one important variation. Being an actuator with a higher negative impact on the user experience, this breaker starts its action at a CPU load that is 140% higher than the internal one. This way we ensure that this breaker only triggers if the internal breaker fails to sufficiently address the overload.

4.3 Evaluation of Admission Control

In this section, we evaluate the effectiveness of the discovered circuit breakers (actuators) and the way they are engaged (sensors). For reminder, we evaluate them based on two metrics, the *individual user throughput* and *combined throughput*, that were defined in Section 3.3. Complementing the insight given by the combined throughput, we also measure the number of *active users*, which is the number of users that were admitted by the external circuit breaker.

We evaluate the platform in four cases: no breakers, only internal breaker, only external breaker and both breakers.

The results, presented in Figure 4, show that the internal breaker alone is unable to keep the platform usable at high loads and essentially behaves as if no circuit breaker was in place. Upon seeing these results, we specifically investigated whether we might have made an implementation mistake. However, CPU flame graphs collected at high load showed that the circuit breaker was working correctly, and that the CPU time spent on the CPU-hungry method (see Fig. 3) was indeed reduced.

We then realised that this is due to the closed-loop nature of the workload: The `getEvents` API, that the internal breaker is actuating on, is called by the browser regularly, with a fixed *delay* – as opposed to a fixed interval – between calls. This means that, above a certain number of users, the quicker the API returns, the more often it is called by the browser, thus leading to a constant load, despite the circuit breaker. For a more detailed discussion about open vs. closed workloads, we refer the reader to [22]. Our approach to evaluating circuit breakers has helped uncover this phenomenon.

In contrast, the external breaker alone did manage to keep the platform usable at higher loads, however, this came at the expense of the number of active users. In other words, the external breaker alone needs to blocking users from accessing the platform already at lower loads. This can also be observed in the sudden drop in combined throughput as the number of users increases.

By combining the two breakers, one gets the best of both worlds: The platform is usable at higher loads, while the number of active users is maintainer higher, even at higher loads.

5 Related Work

Our work is related to workload generation, bottleneck detection and admission control.

Workload Generation Van Hoorn et al. [23] describe how workflows of existing users can be transformed into Markov models and used in load testing schemes. We build upon their work to create a workload generator that is both realistic, but without overfitting, which could happen if all the user behavior is captured in a transition matrix. This allows a better study of various "what-if" scenarios, which is particularly important when designing admission control.

If used for evaluating elasticity (auto-scaling), an important aspect of workload generation is modeling its intensity [11]. Since admission control is complementary to elasticity and to allow us to better understand how the system perform in extreme scenarios, we chose to use a static workload intensity in our experiments.

Bottleneck Detection Automated bottleneck detection [10] has been investigated with the aim of producing a system that automatically resolves them, for example, by tuning the level of workload consolidation to address tail latency requirements [19]. In contrast, our approach resorts to a more manual approach, as the choice of which bottlenecks to address and how to address them is highly dependent on business objectives.

Our work is closely related to online server performance bug inference [4]. The main difference is that our approach is performed during development using a realistic workload generator. That way it is possible to explore various "what-if" scenarios.

Admission Control Admission control is a popular way to maintain availability and performance of Internet applications [7]. It can be deployed either at the entry-point of an application or between application components [9, 17], the latter being useful to isolate failure in large-scale distributed systems. Based on the granularity, admission control can be either request-based – blocking individual requests in isolation – or session-based – blocking whole session in an attempt to ensure users can finish started work. The circuit breakers added to the SimScale platform were deployed at the entry-point. The internal one is request-based, while the external one behaves like a session-based admission controller.

Brownout [14] is a software engineering technique with control-theoretical guarantees to maintain application responsiveness, despite capacity shortage. It works by automatically disabling optional computations, as required to reduce resource usage and restore application responsiveness.

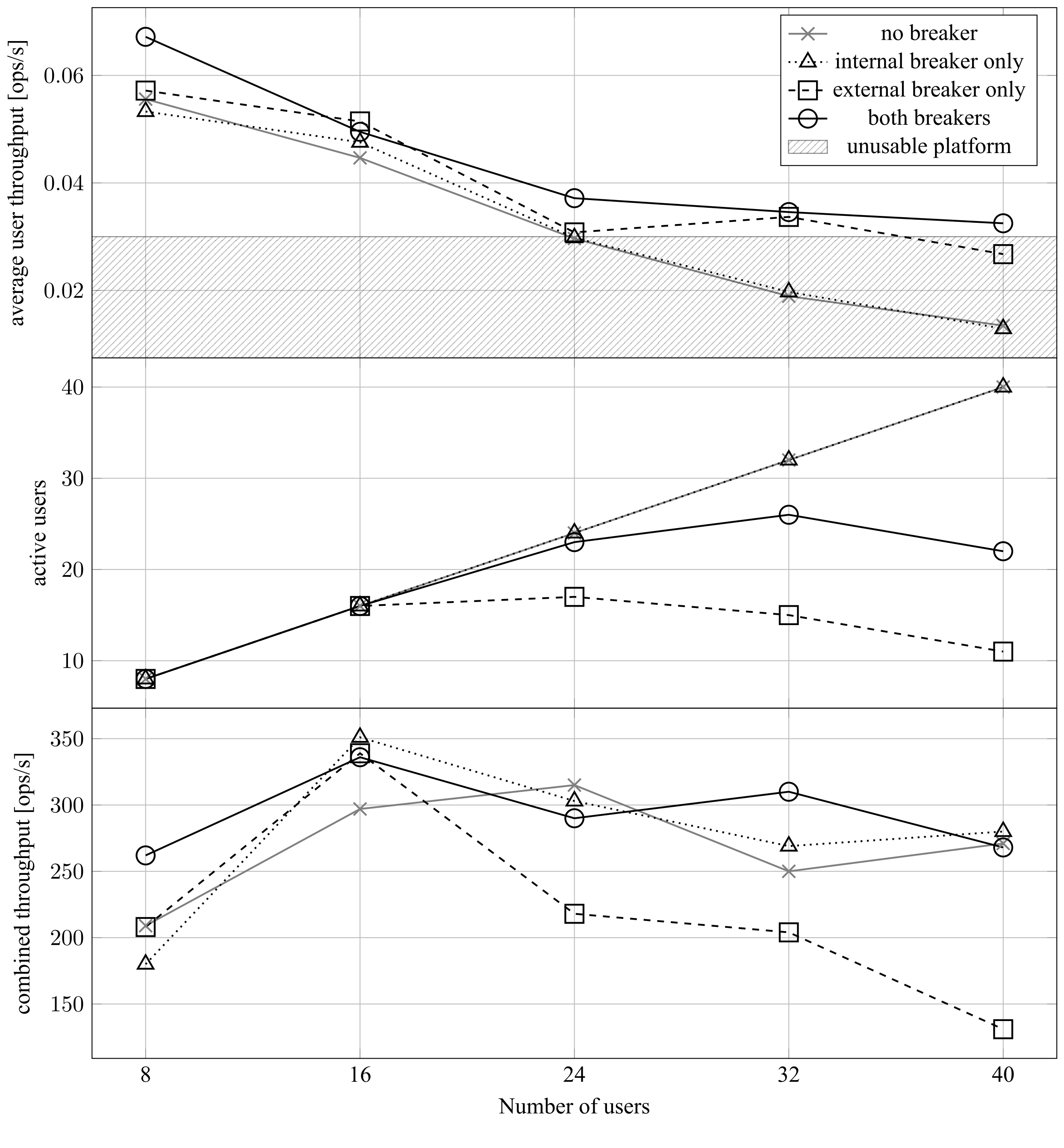

Figure 4: Experimental results showing from top to bottom: the average user throughput, the number of active users and the combined throughput. The region in which the platform is unusable, due to a high negative impact on the productivity of the emulated users, is highlighted in the topmost plot. The load is increased by increasing the number of users from 8 to 40 with an increment of 8. As the load increases, the internal breaker alone is unable to keep the platform usable, due to the closed-system nature of the workload, but does not reduce the number of active users. The external breaker alone is too quick at limiting the number of active users, but keeps the platform usable at higher loads. When both breakers are deployed, one gets the best of both worlds: high number of active users and unsable platform at high load.

Partial execution [13] was proposed to ensure search engines return results within a given deadline, so as to maintain user experience.

To sum up, while existing research has provided valuable building blocks, none of them shows how to retro-fit admission control in an existing production application. Amongst others, this requires producing a realistic workload generator based on production logs, that is flexible enough to stress-test the application in various potential future scenarios.

6 Conclusion

Admission control is a complementary technique to elasticity, to ensure application performance and availability. Admission control may be preferred if elasticity is costly to implement and the spirit of MVP non-functional properties are delayed. We present our experience in implementing admission control in a production Internet-scale application. A prerequisite is generating a realistic workload that, amongst others, passes the input validation rules of the application. We propose extracting a user behavior model from production logs, separating users into classes, using logical states and pre-defined workflows.

We also evaluated our approach against the SimScale platform, a Software-as-a-Service engineering simulation application. Our approach discovered two circuit breakers, each having its drawbacks when used in isolation, but providing the required level of resilience when combined. Further circuit breakers may be discovered iteratively using our approach.

We hope that by sharing our experience, we help start-ups and small companies, that struggle with both small teams and uncertain user requirements, can systematically raise the availability of their applications with little cost.

References

[1] I. Ahmad, A. Gulati, and A. J. Mashtizadeh. vic: Interrupt coalescing for virtual machine storage device io. In *USENIX*, 2011.

[2] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. *SIGOPS Oper. Syst. Rev.*, 37(5):164–177, Oct. 2003.

[3] L. Cherkasova and P. Phaal. Hybrid and predictive admission control strategies for a server, Mar. 19 2002. US Patent 6,360,270.

[4] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang. Perfscope: Practical online server performance bug inference in production cloud computing infrastructures. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, pages 8:1–8:13, New York, NY, USA, 2014. ACM.

[5] A. Filieri, H. Hoffmann, and M. Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 299–310, New York, NY, USA, 2014. ACM.

[6] B. Gregg. The flame graph. *Queue*, 14(2):10:91–10:110, Mar. 2016.

[7] J. Guitart, J. Torres, and E. Ayguadé. A survey on performance management for internet applications. *Concurr. Comput. : Pract. Exper.*, 22(1):68–106, Jan. 2010.

[8] V. Gupta and M. Harchol-Balter. Self-adaptive admission control policies for resource-sharing systems. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):311–322, 2009.

[9] J. Hamilton. On designing and deploying internet-scale services. In *Proceedings of the 21st Conference on Large Installation System Administration Conference*, LISA'07, pages 18:1–18:12, Berkeley, CA, USA, 2007. USENIX Association.

[10] O. Ibidunmoye, F. Hernández-Rodriguez, and E. Elmroth. Performance anomaly detection and bottleneck identification. *ACM Comput. Surv.*, 48(1):4:1–4:35, July 2015.

[11] S. Islam, S. Venugopal, and A. Liu. Evaluating the impact of fine-scale burstiness on cloud elasticity. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 250–261, New York, NY, USA, 2015. ACM.

[12] M. Kihl, A. Robertsson, M. Andersson, and B. Wittenmark. Control-theoretic analysis of admission control mechanisms for web server systems. *World Wide Web*, 11(1):93–116, 2008.

[13] J. Kim, S. Elnikety, Y. He, S.-w. Hwang, and S. Ren. Qaco: Exploiting partial execution in web servers. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, CAC '13, pages 12:1–12:10, New York, NY, USA, 2013. ACM.

[14] C. Klein, M. Maggio, K.-E. Årzén, and F. Hernández-Rodriguez. Brownout: Building more robust cloud applications. In *36th International Conference on Software Engineering*, pages 700–711. ACM, 2014.

[15] E. B. Lakew, C. Klein, F. Hernandez-Rodriguez, and E. Elmroth. Towards faster response time models for vertical elasticity. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 560–565, Washington, DC, USA, 2014. IEEE Computer Society.

[16] F. F.-H. Nah. A study on tolerable waiting time: how long are web users willing to wait? *Behaviour and Information Technology*, 23(3), 2004.

[17] NetFlix. Hystrix: Latency and fault tolerance for distributed systems. Available online: `https://github.com/Netflix/Hystrix`.

[18] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing, ICAC'13, San Jose, CA, USA, June 26-28, 2013*, pages 69–82, 2013.

[19] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion. Energy proportionality and workload consolidation for latency-critical applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, SoCC '15, pages 342–355, New York, NY, USA, 2015. ACM.

[20] D. Raumer, L. Schwaighofer, and G. Carle. Monsamp: A distributed sdn application for qos monitoring. In *Computer Science and Information Systems (FedCSIS), 2014 Federated Conference on*, pages 961–968, Sept 2014.

[21] E. Ries. *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*. The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses. Crown Business, 2011.

[22] B. Schroeder, A. Wierman, and M. Harchol-Balter. Open versus closed: A cautionary tale. In *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 18–18, Berkeley, CA, USA, 2006. USENIX Association.

[23] A. van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In *Proceedings of the SPEC International Workshop on Performance Evaluation: Metrics, Models and Benchmarks*, SIPEW '08, pages 124–143, Berlin, Heidelberg, 2008. Springer-Verlag.

[24] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.

[25] T. Vogel and H. Giese. Model-driven engineering of self-adaptive software with eurema. *ACM Trans. Auton. Adapt. Syst.*, 8(4):18:1–18:33, Jan. 2014.