Non-functional Regression Testing on Serverless Applications Using Controlled Online Experiments

Simon Frey
B.Sc. Computer Science

Supervisor: Prof. Dr.-Ing. Stefan Tai
Secondary Supervisor: Prof. Dr. habil. Odej Kao
Advisor: Jörn Kuhlenkamp

Technische Universität Berlin

November 23, 2020

I hereby declare that the thesis submitted is my own, unaided work, completed without any unpermitted external help. Only the sources and resources listed were used.

The independent and unaided completion of the thesis is affirmed by affidavit:

Simon Frey
November 23, 2020

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Simon Frey
November 23, 2020

1 Abstract

With the increasing popularity of serverless applications, the need for controlled experimentation in these systems arises. A key objective is an experimentation based on live traffic in order to ensure a high quality experiment traffic (so-called online experiments). In comparison to offline experiments, no external load generator is used and the experiment traffic is directly rerouted from the live traffic. The thesis covers the topic if workloads with special traffic shapes can be used in an online experiment on serverless applications by proposing requirements for a system enabling these experiments. In addition, a system design is proposed and evaluated with a prototypical implementation against the aforementioned requirements. The evaluation is done by experiments on a case study application. It is shown that at the moment of writing the elasticity issues of the platform require to use of provisioned concurrency in order for the proposed system to be a viable solution for online experiments. As this feature has a high cost overhead using specialized workloads in online experiments on serverless applications is possible, but expensive.

Mit der steigenden Popularität von serverless Applikationen entsteht der Wunsch nach kontrollierten Experimenten auf diesen Systemen. Um eine hohe Workload Qualität zu gewährleisten werden sogenannte online Experimente eingesetzt, welche auf dem live Traffic der Applikation basieren. Im Vergleich zu offline Experimenten wird kein externer Load Generator genutzt, sondern der Traffic ist direkt vom live Traffic umgeleitet. Diese Bachelorarbeit beschäftigt sich mit dem Thema, ob Workloads mit frei definierten Traffic Formen für online Experimente in Serverless Applikationen benutzt werden können. Hierfür wird ein Systemdesign und eine Implementation dessen vorgeschlagen. Das Design wird mittels eines experimentellen Ansatzes getestet und es wird dargelegt, dass zum jetzigen Zeitpunkt die fehlende Elastizität der Plattform *Provisioned Concurrency* unumgänglich machen. Diese Funktion ist mit hohen Kosten verbunden, aber ermöglicht die gewünschten kontrollierten online Experimente mit frei definierten Traffic Formen.

Contents

2 Introduction

2.1 Motivation

The serverless computing [24] or function as a service (FaaS) execution model gains more traction[1] as it "[...] offers a new alternative to develop cloud-based applications. [...] Instead of using the cloud infrastructure directly, developers provide short running code in the form of functions to be executed by a FaaS platform provider. This simplified programming model lets developers create applications without the burden of server-related operational tasks such as provisioning, managing, or scaling of resources."[19, p. 1] This growth is further facilitated by two differences compared to the infrastructure as a service (IaaS) model, the first being the Fully-managed elastic scalability where "[...] the [cloud] provider dynamically adapts resource allocations according to the customer's demand [workload] and the customer pays for the actually consumed resources [...]"[33, p.1]. This allows to focus more resources on business logic than DevOps[22]. Additionally FaaS offers a potential cost benefit[6] empowered by the fine-granular pay-per-execution cost model[2]. Because of this model FaaS does not have cost at rest. Thus, volatile workloads are a scenario where FaaS is most beneficial.

Both characteristics are most beneficial if the application code has a good quality. In order to ensure a high code quality, the software needs to be tested on a regular bases. One test schema is regression testing which "[...] is to ensure that changes made to software, such as adding new features or modifying existing features, have not adversely affected features of the software that should not change. [...]"[35, p.1]. Regression testing is of interest for web applications as they "[...] must undergo rapid adjustments, since the businesses they support are frequently changing [...]"[25, p.1]. The e-commerce website presented in the evaluation section is such a web application profiting off regression testing. Regression testing consists of functional and non-functional test cases. Functional cases test the actual behavior against the specification and non-functional cases test "[...] the way a system operates, rather than specific behaviors of that system [...]"[3].Non-functional regression testing is an important mechanism to ensure good application behavior. Manual regression testing is resource-intensive as time-consuming processes are often repeated. Automatic regression testing solves this resource problem by automating the test execution and evaluation. These tests are also referred to as Experiments. Automatic regression testing can be implemented in two different ways: Offline or Online[14]. Offline experiments run within a dedicated experimentation infrastructure isolated from the live application. This brings two problems: The Infrastructure overhead increases with the complexity of the application as all resources need to be copied and continuously updated to stay in sync with the live application. The other problem is the difficulty to ensure a good workload quality as the workload is artificial and can differ a lot from the actual live application. Online experiments try to tackle these problems, by being executed in the live application. Thereby the workload quality is high because it is the live workload. The infrastructure overhead is smaller as no isolated environment needs to be created and kept up to date. These benefits come with the downside of user-facing side effects, which depend on the chosen deployment method. With the usage of the live workload, it is difficult to run certain experiments relying on certain traffic patterns. The different deployment methods and the problems with traffic depended on experiments are presented in the next subsection.

With the rise of FaaS and its usage in more complex applications, non-functional automatic regression testing is required to ensure the same application quality for FaaS as for conventional execution models. The distributed nature of serverless applications increases the difficulty to keep an offline testing environment up to date as to why online experiments are beneficial. Online experiments on serverless applications are not supported out of the box by the cloud providers and research does not yet cover this topic.

In conclusion, non-functional regression testing on serverless applications using controlled online experiments is relevant but yet unexplored.

[1]https://cacm.acm.org/magazines/2019/12/241054-the-rise-of-serverless-computing/fulltext
[2]https://aws.amazon.com/lambda/pricing/
[3]https://en.wikipedia.org/wiki/Non-functional_testing

2.2 Problem

Online experiments require the integration of an experiment setup into the production environment. There are three different deployment methods known from continuous deployment[27] (CD) which are also utilized for this integration. The first is *Percentage based routing* a group combining different strategies found in the literature(Canary deployment[13], A/B deployment[16] & Gradual deployment[13]). All are based on the same technique: Routing a certain percentage of the workload through the experiment. Sometimes the terms are used interchangeable as there is no clear-cut. The second one also depending on traffic routing is *Shadow/Dark deployment*[27]. It works with duplicating a certain amount of traffic to the experiment. Compared to the percentage based deployment a dedicated second deployment of the function is utilized which results are not returned into the live application. The last one, not depending on traffic routing, is called *Feature toggles*[4]. Utilizing special code in the live codebase the activation is done within the application logic.

Depending on which deployment method is used the amount of user-facing side effects differ. All deployment methods work on the live workload resulting in a realistic workload, but the ability to use specific workloads is low, as only workload pattern occurring during the time of the experiment can be tested. This is a problem for capacity experiments[13, p. 76]. This kind of experiments test the application behavior regarding special workload scenarios. E.g. The e-commerce website used for evaluation may have a lot more traffic during the Christmas time than the rest of the year. In order to be able to ensure the application can handle that load without impairing the user experience capacity experiments are required but difficult to implement in an online experiment setup. Generating a dedicated traffic demand on a serverless application is yet unexplored.

2.3 Research question

The aforementioned problem results in the following research question:

How can experimenters use specialized workloads in online experiments on serverless applications on demand?

Special workloads are the key shortcoming of the currently available online experiment systems, which enable traffic routing to different serverless functions, but no additional traffic. If and how this additional traffic injection can be archived in a serverless application without the need for additional offline traffic (e.g. a traffic generator) is a key part of the thesis. The other important aspect of the research question is the "on demand" requirement resulting in an operational overhead demanded to be as low as possible. The experimenter is the human actor using ExSys in order to execute and evaluate experiments on the system.
When it is possible to use specialized workloads in online experiments, non-functional regression testing on serverless is enabled by executing an experiment with the same configuration for every new version that should be tested.

2.4 Approach

To tackle the research question the thesis *proposes a system to shape the experiment traffic (ExSys)* in order to run online experiments and *evaluates the system quality* within the e-commerce case study application.
The proposed system enables percentages based routing and shadow deployments by injecting additional functions into the live system and logging their results. The data evaluated in the thesis is gathered from this system and crosschecked with client side data in order to guarantee its validity.
Feature flags are not enabled within ExSys as the nature of their implementation is contrary to the design principles in serverless functions. If there are different code paths to be evaluated, the nature of serverless functions enables that without requiring the different code paths in the function itself. Deploying several functions and routing the traffic to them as shown in the thesis enables the same experiments without requiring explicit feature flags in the code.
In order to verify the validity of the system the experiments are run on a well-defined case-study application. The application mimics an e-commerce store and represents a typical use-case for serverless applications as used in the industry. Research ([34],[12]) also experiments with using serverless applications for big data processing, but this use case is out of scope for this thesis.

2.5 Contributions

The thesis provides requirements for an online experimentation system which is able to help experimenters to use specialized workloads in online experiments. Additionally, a design and a prototypical implementation of an online experimentation system is introduced. All three elements, the requirements, the design and the prototypical implementation are evaluated by experiments.

2.6 Organization

After presenting Related work in the field of serverless computing and online experiments various terms are defined and are explained in the Background section. By defining hard and soft criteria for the system evaluation in the Requirements section it is ensured that all design decisions in the design are beneficial to the goal of answering the research question. The Design section lays out the design of the experiment system showcasing the control and data flow, as well as potential problems that are inherent with the chosen design. In the Implementation section the design of the previous section is implemented in a Node.js prototype targeting the Amazon AWS ecosystem. The Evaluation lays out the experiment setup and discusses the prototype by comparing the results against the requirements. In the Conclusion the thesis lays out why ExSys in its current state is no possible solution for the online experiments on serverless applications and where further research could start.

3 Related work

In their Bifrost paper Schermann et al.[29] lay out the architecture of a tool enabling percentage based routing & shadow deployment as deployment methods for experiments in container based application. Their insights are used to explore comparison criteria and base experiment designs on. Their case study application is used in order to evaluate ExSys.

Ernst et al. propose the usage of ephemeral proxy in their canary paper[7]. It is to be evaluated if this approach is superior to the permanent proxy used within bifrost for percentage based routing & shadow deployment.

Apart from Bifrost Scherman et al. have published other relevant papers regarding the thesis covering research on continuous experimentation[30],[28].

Within their paper Kuhlenkamp et al. [19] propose an experiment design for testing the elasticity of FaaS platforms. ExSys tries to enable this experiment type as online experiment.

Online experiments rely on zero downtime deployment and the research in this field[13][9][7] helps as base for the system design.

With AWS Codeploy[4] there is already a percentage based release strategy implemented within AWS. In the Evaluation this implementation is discussed and evaluated if it is usable for controlled online experiments.

In comparison to [23] exploring the possibility of passive benchmarks by observating jobs, all experiments being conducted with ExSys do require operational overhead as the results are actively measured within the ExSys system. Passive benchmarking are not a goal of the system.

4 Background

4.1 Serverless Applications

As mentioned in the Motivation FaaS is getting more popular over the course of the last years as can be seen in the following chart displaying the results for the search term *serverless computing* on Google Scholar[5] evaluated by year:

[4]https://aws.amazon.com/de/blogs/compute/implementing-safe-aws-lambda-deployments-with-aws-codedeploy/

[5]https://scholar.google.com/

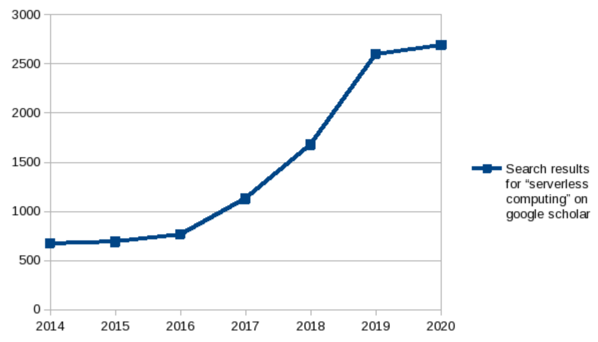


Figure 1: Results for *serverless computing* on Google Scholar by year

The definition of serverless applications differs depending on the paper used as basis. This thesis uses the characteristics layed out in [32]:

- **Hostless**: The application engineer does not need to handle the server operations as this task is taken over by the platform. "One advantage this brings is the significantly less operational overhead — there are no servers to upgrade and no security patches to apply. But it also means that different kinds of metrics need to be monitored in an application, and thus there is a need for relearning how to tune the architecture."[32]
- **Stateless**: "When state is not stored in applications, horizontal scaling is very easy — you just spin up more instances. Being stateless also means that room for error is greatly reduced. But being stateless also means that techniques that require state can't be used in application development; for example, it's not possible to use HTTP sessions."[32]
- **Elastic**: "This means there is no need for the manual management of resources, and that many challenges in resource allocation disappear. Commonly, it also means paying only for the resources that are actually used."[32]
- **Distributed**: Serverless applications are distributed by default. "Persistence is done in Backend as a Service (BaaS), code is run in multiple functions, other services are used for authentication and queues, etc. Being distributed also brings high availability to the architecture. If the current availability zone is facing problems, the architecture can utilize another zone that is available."[32]
- **Event-driven**: Serverless applications are invoked via events on external services. Most platforms offer a wide array of invocations events rendering serverless applications especially useful for connecting different services.

additionally the **Usage based pricing** is an advantage over the classical computing models, as only resources that are actually used have to be payed resulting in less cost overhead for functions at rest.

Apart from the classical use case of building user facing web applications research begins to utilize serverless functions also for big data processing [34],[15],[18],[20]. This thesis focuses on the classic use case, evaluating the experimentation system in a serverless web application.

4.1.1 Elasticity promise

"FaaS platforms promise the core capability to execute code without requiring a FaaS platform client to manage resource allocations. Thus, a cloud automatically provisions resources to execute application code with incoming events. In an application architectural context, this capability implies that a FaaS platform provides an option for throughput-based application objectives. We refer to this option as elasticity."[19] In the paper [19] Kuhlenkamp et al.state that the platforms at the moment do not fully deliver on this elasticity promise. In the experiments Platform elasticity issues are a problem for the validity of the experiments. This problem is tackled in ExSys by using Provisioned concurrency which enables higher loads on the system by pre-provisioning a selected amount of parallel functions. Contrarian to the elasticity being one of the main selling points the platforms still have to improve to keep up with this promise.

4.1.2 The use of VMs in serverless applications

”There are a lot of misconceptions surrounding serverless starting with the name. Servers are still needed, but developers need not concern themselves with managing those servers. Decisions such as the number of servers and their capacity are taken care of by the serverless platform, with server capacity automatically provisioned as needed by the workload. This provides an abstraction where computation (in the form of a stateless function) is disconnected from where it is going to run.”[3, p. 4]. The serverless functions itself still runs in a virtual machine (VM). In comparison to a server full model these VMs are managed by the cloud provider and not by the developer themselves. This reduces overhead for the user of a serverless solution, but also reduces the amount of possible individualization.

4.1.3 Connection of microservice architecture & serverless applications

”In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.” [10] Serverless and microservice are no distinct but interconnected architectures. A microservice architecture can be build with serverless functions. On the other hand, only because serverless functions are used the application is not automatically a microservice architecture. The Application scenario does not follow the microservice architecture and in order to reduce infrastructural overhead.

4.2 Regression Testing

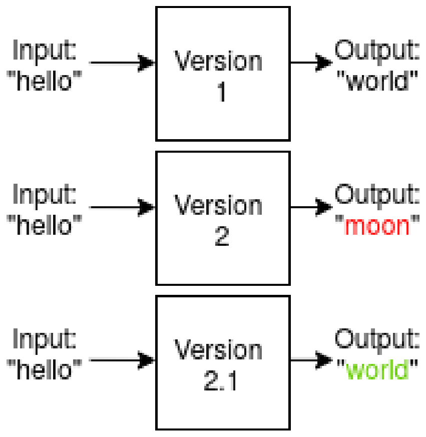


Figure 2: Functional regression testing

”[...] Regression testing is a testing process which is applied after a program is modified. It involves testing the modified program with some test cases in order to re-establish our confidence that the program will perform according to the (possibly modified) specification. In the development phase, regression testing may begin after the detection and correction of errors in a tested program. [...] Regression testing is a major component in the maintenance phase where the software system may be corrected, adapted to new environment, or enhanced to improve its performance. Modifying a program involves creating new logic to correct an error or to implement a change and incorporating that logic into an existing program. The new logic may involve minor modifications such as adding, deleting, rewriting a few lines of code, or major modifications such as adding, deleting or replacing one or more modules or subsystems. Regression testing aims to check the correctness of the new logic, to ensure the continuous working of the unmodified portions of a program, and to validate that the modified program as a whole functions correctly. [...]”[21, p. 1]

Regression tests are always testing a delta against a base (in the example Functional regression testing this is *Version 1*). The definition of regression as stated above is covering the so-called functional regression testing, in which the correctness of the systems is ensured by testing new code paths against previously test cases which are assumed to be valid. (In the example *Version 2* is producing a wrong result resulting in the regression test failing) The functional regression testing is not the focus of ExSys, even if it would be possible with the tool. The thesis focuses on non-functional regression testing.

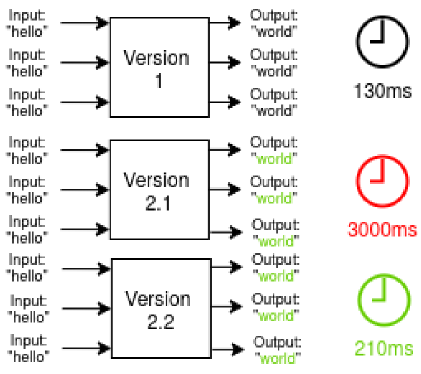


Figure 3: Non-functional regression testing on the metric *elasiticity*

The differentiation between functional and non-functional regression tests depends upon the requirements it evaluates. Functional requirements are typically covering required operations or input-output data formats while non-functional requirements are covering topics such as security, performance, elasticity, maintainability, etc. (See [11]).
As can be seen in the example in Non-functional regression testing on the metric *elasiticity*, the *Version 2.1* produces a correct output, but requires a lot more time for serving all three requests. This regression test would fail, as it is too slow. On the other hand *Version 2.2* is fast enough, while still being slower than the basis (*Version 1*). This shows that non-functional regression tests are not evaluated in a boolean fashion, but test against an accepted result range.

In the experiments the thesis focuses on the non-functional regressions' execution speed and elasticity which are the two most important metrics for a serverless application.

4.3 Online experiments

Online experiments are one possible way to do regression tests. For every new version an online experiment (with the same configuration as for the base) is conducted and the results are compared against the base. For elasiticity this comparison focuses on the possibility of the new version scaling up according to the defined quantity.

The terms online and offline experiments are used with various meanings throughout research. For this thesis the differentiation between online and offline experiments is the infrastructure and workload used for executing an experiment. Explicitly the terms do not refer to internet access, both offline and online experiments are usually executed on a server and require internet access.

The infrastructure used in offline experiments is a copy of the live system. An application (e.g. the web shop used as case-study application in the Evaluation) is copied into a second environment completely separated from the live system. Exclusively in this copy the experiments are run. In comparison, online experiments do not require a separate environment and the experiments are run in the live system.

The isolated nature of offline experiments results in the workload used for the experiment being artificial. The online experiments use the live workloads generated by the users of the live system.

The Motivation lays out that online experiments have several benefits over offline experiments:

- Low infrastructure overhead: Not the entire infrastructure has to be replicated
- High workload quality: Accurate test traffic generated by real use of the system

This benefits come at the cost of potential user facing side effects and a low workload expressiveness.
Offline experiments are easy to conduct with serverless applications and do not differ a lot from offline experiments in classic execution models. Online experiments on the other hand are not yet explored in the serverless execution model and bring several challenges which are explored in the thesis.

4.4 Specialized workloads

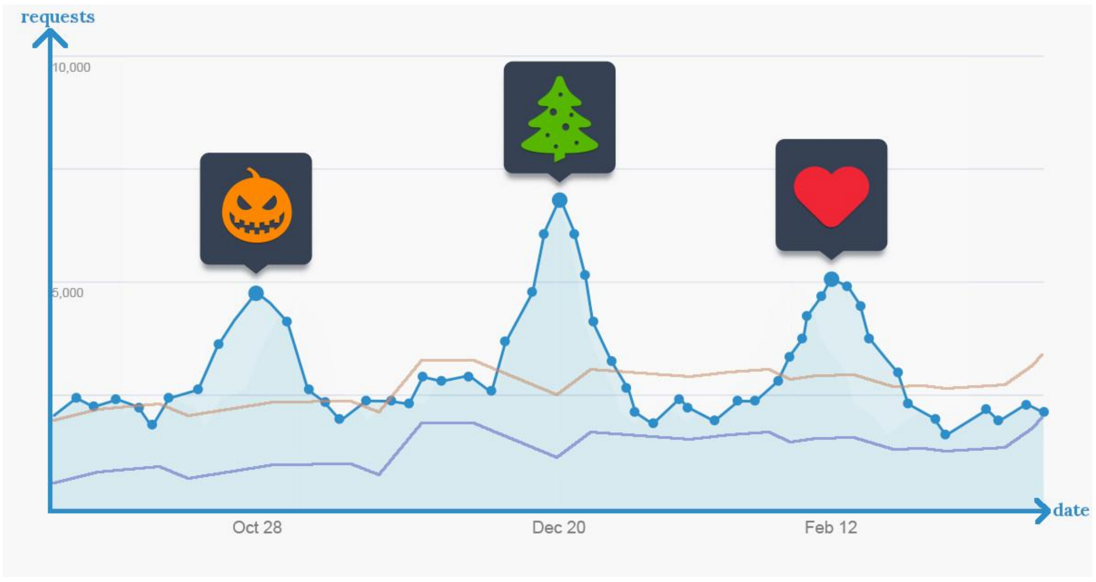


Figure 4: Traffc example on an online shop. (Violet line: Live traffic) based on[6]

The requirement for specialized workloads comes from the current experiment frameworks lacking the possibility to define shadow traffic by a *request/second* ratio. As seen in the example above, there are certain events throughout the year when the online shop has increased traffic. In order to verify upfront if the application can handle this, it would be beneficial to test this traffic scenarios.

Simple systems for experimentation in serverless applications do only allow to define target traffic by a multiple. (In the graphic above, this multiple is represented by the orange line) A multiple allows testing if the application can handle more traffic overall, but is not expressive enough to test the elasticity of the system. It is limited to the traffic shapes that are coming into the system via the live traffic. With the experiments only running a few minutes, this live traffic is mostly stable.

The desire to test more specialized traffic shapes, for example the steep increase of traffic for Christmas a more granular control is required. A method allowing building experiments with the desired traffic shapes, is defining the traffic as a *request/second* ratio.

ExSys supporting the traffic control via this *request/second* ratio is the key benefit over other systems and enables a high workload expressiveness.

5 Requirements

In order for an experimentation system to be considered as fulfilling the research question, the following requirements must be met.

5.1 Functional requirements

The most important requirement are low *Client-visible Side-effects (CvSEs)* which should be minimized during the execution of the experiments to keep the live application functional. These CvSEs are measured by comparing the request response time (in ms) and the error rate (HTTP status code) between the basis system and the system with a running experiment. If the CvSEs are too high a rollback of the experiment might be required to keep the application available.

Another requirement is a high *Workload accuracy (WA)*, which describes how accurate ExSys can generate the required workload. It is measured by averaging out the percentage the experiment workload is resembling the intended workload. The live nature of the traffic makes it highly unlikely that the workload accuracy will reach 100%. If that accuracy is too low the experiments will generate wrong data which results in wrong decisions.

In order to enable continuous experimentation a low *Time to experiment start (TtES)* (in s) is required. This characteristic involves the time it takes the system to inject and start a new experiment. This time increases with more complex applications involved. The time from recognizing a problem during the experiment execution until the completed removal of the experiment is called *Mean time to repair (MttR)* (in seconds). The MttR should be as small as possible and is mainly influenced by the deployment and removal process of the experiment system.

5.2 Non-functional requirements

There is the non-functional requirement for a good *Workload Expressiveness (WE)*, which is the ability to express different workloads. If this is low, only very few possible workload scenarios can be tested. WE is a soft requirement as it is not directly measurable, though evaluated by inspecting all possible workload scenarios enabled with an experimentation system.
Another requirement is a low *Infrastructure cost overhead (ICO)*. It depends on the amount of additional infrastructure required. The cost is evaluated as a percentage based on required additional function calls and external services. Too high costs prevent continuous execution of experiments as of economic reasons.
The softest requirement is a good *Usability (U)* of the experimentation system. Not influencing the actual experiment execution a too difficult system might not be used widely enough to bring value. It is not measurable in numbers and is evaluated by inspecting the interface an experimentation system provides.

6 Design

Design challenge	**ExSys approach**
Workload expressiveness	Allow experiment definition based on a *request/second* ratio (see Experiment config)
Low client visible side effects	Asynchronous execution of shadow functions (see Experiment config)
Usability	REST User API for experiment control
Infrastructure cost overhead	Ephemeral Proxy function only being deployed for the duration of the experiment
Custom traffic routing	Intermediate Proxy function function controlling experiment state
Measure incoming traffic	Distributed counter via database stream
Elasticity issues	Provisioned concurrency

The experimentation system (ExSys) proposed in this section aims to answer the research question by fulfilling all requirements. It offers the possibility to run multiple traffic based experiments within a serverless application and automatically integrates with an existing serverless deployment. By allowing percentage based routing and shadow deployments a wide range of experiments are enabled with ExSys. The quality of ExSys is evaluated on the example of two different experiments. A prototypical implementation of ExSys on AWS (ExSysProto) is further discussed in the Implementation section.
In the following the terms *Original function* & *Experiment function* are used. The original function is the serverless function that is originally deployed in the serverless application. It is the start and desired end function being deployed i the system. The experiment function is the function which is experimented on. ExSys helps to deploy this function in a canary or shadow traffic strategy. It can be the same as the original function.
A stage within an experiment is a single part with the same configuration. An experiment can theoretical consist of $[1; \infty[$ stages. When the experiment configuration changes e.g. the amount of canary traffic changes from 5% to 10% the experiment stages is changed. Even stages with the same configuration are considered to be distinct.

6.1 System overview

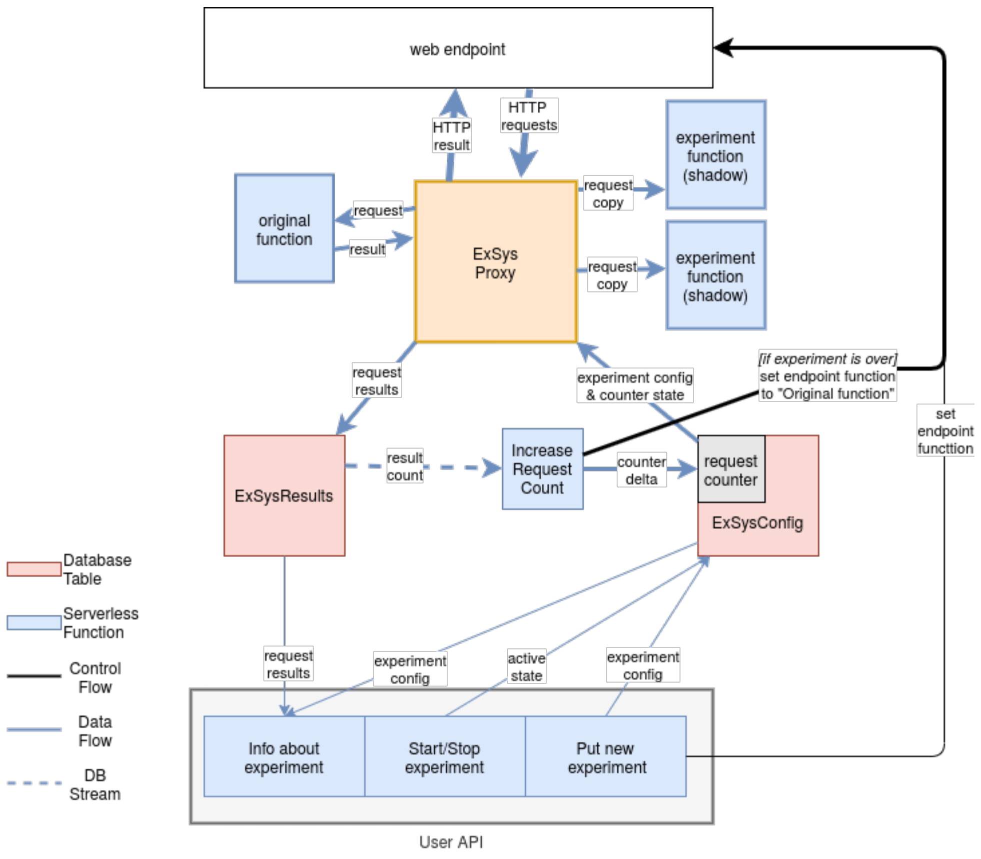


Figure 5: ExSys system overview

ExSys consist of three parts:

- Increase request counter function (IRCF): The IRCF controls increases the request counter throughout the ongoing experiment. It checks the incoming experiment results and when the experiment is over it automatically stops the rerouting of the requests and removes the proxy function. The IRCF is executed by so-called database streams.These streams call the configured serverless function whenever a table is modified. Based on the information in the modification statement the functions do their work.
- User API: This REST API allows the experimenter to start & stop experiments and gather the experiment results. The API endpoints are described in User API
- Proxy function: The proxy function is deployed in between the original serverless function and the web endpoint. It is shaping the traffic, e.g. duplicating events to simulate a higher traffic scenario. Side effects have to be prevented within the experiment function and are thereby out of scope for ExSys. The proxy function is configured through an entry in a distributed database table *ExSysConfig* which it loads on function startup. It persists its results into an additional database table *ExSysResults*. Troughout the experiment it loads the current request counter and calculates the current $request/second$ (r/s) ratio.

In the following subsections the Business Process Model and Notation (BPMN)[7] is used for visualizing the inner workings of the components.

[7]https://en.wikipedia.org/wiki/Business_Process_Model_and_Notation Accessed: 2020-10-09

6.2 Increase request count function

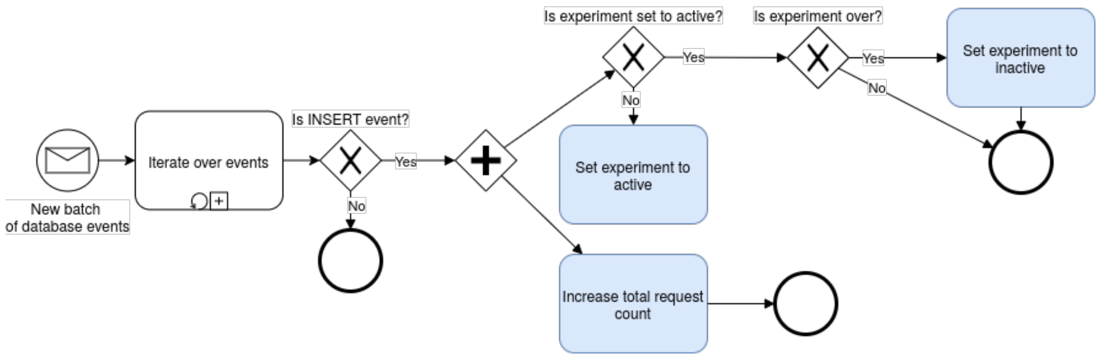

New batch of database events
Iterate over events
Is INSERT event?
Yes
No
Is experiment set to active?
Yes
No
Set experiment to active
Is experiment over?
Yes
No
Set experiment to inactive
Increase total request count

Figure 6: Increase request count function BPMN

The Increase request count function (IRCF) is neither executed directly by the experimenter nor via the proxy. It is executed via a database stream on the ExSysResults table. The ExSys proxy writes a batch of ten entries into this table, than the IRCF is called. It has two functions: It increases the total requests counter (see Distributed counter) and stops the experiment when it is over. An experiment is defined as being over when all stages have been executed. At that moment the IRCF removes the experiment proxy, reverting the serverless function which was experimented on to it's pre-experiment state and sets the *active* field in the experiment configuration to *false*.

As layed out the IRCF relies on a database stream, which is compared to a log based approach (parsing of application logs) a more expensive resource. This increased cost come with the advantage of a faster speed. Logs have a delay until they would be available to ExSys. This delay renders them useless for the experiment control, as the delayed data results in the experiment stages and the whole experiment running for a longer time than intended. Database streams are a lot faster and thereby the higher price is justified in order to run more accurate experiments.

The database streams have a configured batch size for which they call the attached serverless function. This batch size is set to ten. This results in not every single modification of the database causing an execution of the attached function. This is a bit cheaper as the functions are executed less often. The value ten was chosen by evaluating different settings. This setting was the best in order of accuracy for the experiment control. For systems with higher traffic this batch size can/should be increased in order to gain additional performance and reduce cost.

6.3 Proxy function

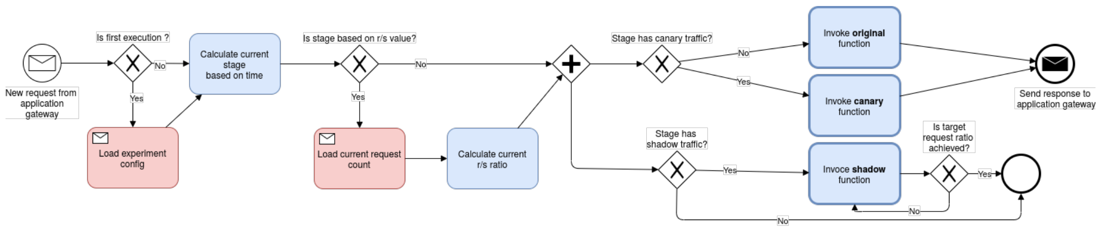

New request from application gateway
Is first execution ?
No
Yes
Load experiment config
Calculate current stage based on time
Is stage based on r/s value?
No
Yes
Load current request count
Calculate current r/s ratio
Stage has canary traffic?
No
Yes
Invoke original function
Invoke canary function
Send response to application gateway
Stage has shadow traffic?
Yes
No
Invoce shadow function
Is target request ratio achieved?
Yes
No

Figure 7: Proxy function BPMN

The proxy function does the heavy lifting of the ExSys functionality. It is an application level traffic router. In order to enable an experiment the proxy function replaces the original function in the serverless application. When the proxy function is called it executes the original function or the experiment function (as canary traffic). Additionally, it copies the request input and executes as many shadow functions with this payload as configured for the stage. It calculates the current stage based on the start timestamp of the configuration. For every request that is executed via the proxy it inserts an entry into the ExSysResults table. (see Get results of experiment).

In order to allow ExSys to be used for a wide range of functions and attached systems, ExSys assumes that side effects are handled within the functions themselves. The proxy is forwarding the HTTP events to the functions and acting as an application level router. Neither the functions, nor the HTTP body itself are changed. In order to allow the functions to prevent side effects, ExSys injects an additional HTTP header. The existence of this header signals the called functions that they are utilized via ExSys and enables them to prevent side effects. More on this is explained in the *Additional headers* paragraph.

Bifrost uses a permanent proxy for enabling the experiments. This is a decision that is suitable for the microservice architecture in bifrost. The microservices need a router anyway and thereby this internal routing can be done via the permanent bifrost proxy. The bifrost proxy replaces another routing component and thereby the overhead of having it deployed permanently is low to non-existent. In the serverless architecture on the other hand this routing is done by an external service. Caused by the pricing model the proxy function causes a high cost-overhead as at least two functions are executed: The original lambda function and additional the proxy function. The proxy function runs until the original function is finished. This double execution has a big cost overhead. In order to optimize the overall cost of an experiment it is beneficial to have the proxy function deployed as short as possible. Similar to [7] ExSys uses an ephemeral proxy that is only deployed for the time of the experiment. The managed structure of the serverless model helps to keep the operational overhead for deploying and removing the proxy functions low and it is automatically deployed on experiment put. The ephemeral proxy outperforms the permanent proxy in costs and general system performance. The overhead of the ephemeral proxy are evaluated in the second experiment.

The experiment stages are controlled by time. Every stage runs X seconds. This is the predominantly used way of controlling experiments in the research community [29] [17] [7]. With ExSys this might have the following downside: If no traffic at all arrives at the function under test, ExSys will not be able to start/stop the experiment at the given times. The stage control depends on steady incoming traffic. The traffic is not required to be uniform. This design choice removes overhead by not depending on additional background workers taking care about the experiment state. Any system under mediocre load has no problems with this choice as of why the benefits out-weight the downsides.

6.3.1 Additional headers

The ExSys proxy injects an additional header with a boolean value:

- *false*: The traffic is either original or canary traffic. The results of the function are returned to the client. The function can cause side effects.
- *true*: The traffic is artificial (shadow) traffic. The results are **not** returned to the client. The function **should not** cause any side effects like data changes in the persistence layer.

If the header is not present the function is not called via the ExSys proxy.

In order to allow the function to return additional information for experiment evaluation the ExSys proxy checks the *ExSysResult* header. This header can contain an arbitrary stringified JSON object. This header must only be return if the *ExSysArtificialTraffic* header is present on the input event. The returned additional results are saved into the *ExSysResults* table and the header is removed before returning the result to the client. The content of the header has to be manually evaluated by the experimenter and is only passed through by ExSys. An example use case is the identification of different function VMs, as it can be seen in the Evaluation.

6.3.2 Distributed counter

In order to control the r/s ratio for shadow traffic, ExSys needs a way to count the amount of current requests. With this current request count a multiplier for the shadow traffic is easy to calculate. E.g. if it should be ensured that 80 r/s are injected into the system and know the current ratio is 20 r/s, ExSys has to inject $80/20 - 1 = 3$ additional requests as shadow traffic. This calculation is trivial if both ratios current and desired r/s are known. The desired one can easily be retrieved from the configuration as it is static per stage.

In a distributed system, the counting of the current requests per second is a non-trivial task. There is no central point where this metric can be accessed in real time and it has to be collected within ExSys. Most cloud providers offer a distributed database can be utilized for this task. These products are closed source as of why the exact inner workings are not known.
Quite likely conflict-free replicated data types are used in most implementations. "A conflict-free replicated data type (CRDT) is an abstract data type, with a well de-fined interface, designed to be replicated at multiple processes and exhibiting the following properties: (i) any replica can be modified without coordinating with an-other replicas; (ii) when any two replicas have received the same set of updates,they reach the same state, deterministically, by adopting mathematically soundrules to guarantee state convergence."[31, p. 1]
As these datatypes used for the counters naturally converge to the same stage on all replicas, the overall correctness of the counter can be ensured. As the counter still needs to be synchronized a delay is inherent in the system and the counter might not represent the current traffic state at all time but might be a bit off. This causes problems discussed later on.

The distributed counter works in an absolute fashion. It increases throughout the whole time of the experiment execution, representing all non-artificial request in an experiment. The counter averages out the r/s ratio over the whole experiment time, resulting in single traffic spikes having less direct impact on the experiment control with an overall traffic change still being reflected. This design increases ExSys resilience against traffic changes. To still enable the counter to react on recent traffic changes, it uses a weighted approach. Instead of increasing by one per request count it increases by $count_{request} * ss$ with ss being the seconds since the experiment start. Thereby more recent request have a higher multiple, resulting in being weighted more in the result. The current r/s ratio is calculated within the proxy function like following:

$$ratio_{r/s} = \frac{weigthed\_total\_request\_count}{\sum_{i=1}^{ss} i}$$

6.4 User API

For controlling the system ExSys offers a HTTP API containing the following endpoints:

- **/example** GET : *Get a example experiment configuration which can be edited and then PUT to /experiments*
- **/experiments** PUT : *Create a new experiment*
- **/experiments/{id}** GET : *Get the experiment configuration & results*
- **/experiments/{id}/start** GET : *Start an experiment*
- **/experiments/{id}/stop** GET : *Stop an experiment*

All endpoints return the experiment configuration (see: Experiment config) and if available the results (see: Get results of experiment).

6.4.1 Get results of experiment

You can received the current experiment results at any time with a GET to */experiments/{id}*. If the experiment is not finished the results will differ each time. Additionally to the Experiment config the API returns the request results. This is an array of stages containing an array of single results with the following structure:

```
{
 "startTime": 0,
 "proxyID":"sda21sd",
 "strategy": "original",
 "stage": 0,
 "trafficArtificial": false,
 "res": {},
 "responseBody":{},
 "executionTime": 0
}
```

Apart from *res & responseBody* the parameters are self-explanatory. *res* is the information that is returned in the *ExSysResult* header (See Additional headers). If *saveResponseBody* is set to *true* for that certain stage (See Experiment config) *responseBody* contains the full HTTP response returned from the function. The *proxyID* is a unique id generated on the first start of the function VM, which helps to identify the VM reuse of the platform.

6.4.2 Execute experiment

There are two API actions required for executing an experiment:

1. PUT a new Experiment config to */experiments*. The experiment is now configured in ExSys and the proxy function is deployed, but not started yet. If the syntax is wrong or there is already an experiment running on that certain function an error is returned. The experiments did show that no user facing side effects are noticeable when the proxy is deployed.
2. POST to */experiments/{id}/start* with the experiment id received in response of step 1 in order to start the experiment. If the id is wrong, an error is returned.

The experiment is then run according to the configuration until all stages are over. It then is automatically stopped and the proxy removed to restore the original state of the serverless application. The current state of the experiment can be accessed via a GET to */experiments/{id}* as further explained in the next paragraph.

6.5 Experiment config

The experiment configuration used to configure an experiment in ExSys looks like the following:

```
{
  [FaaS function identifiers]],
  "preProvision": 70,
  "stages": [
    {
      "time_s": 0,
      "canary": {
        "percent": 0
      },
      "shadow": {
        "totalRequestsPerSecond": 70,
OR      "multiplier":2
      },
      "saveResponseBody": false
    }
  ]
}
```

It contains three parts:

- *FaaS function identifiers* block: How to access the serverless functions where the experiment proxy should be injected. This block varies depending on the platform it is implemented for.
- *preProvision*: Pre-provision the given number of functions in order to prevent elasticity issues
- *stages* block: Contains $1 \leq n < \infty$ stage configurations. These configurations define how the traffic should be routed to the experiment function.
  - *time_s* configures for how many seconds this stage is active.
  - *saveResponseBody* defines if the complete response payload of the functions should be saved into the database. Be aware that this could be rather long and may clutter results.

  - *canary* percentage defines how many percent of the traffic should be routed to the experiment as canary traffic. Canary traffic is allowed to cause side effects and substitutes traffic to the original function.
  - *shadow* strategy can be configured in two different ways as following (only one can be used at a time):
    - *multiplier*: For every request that arrives at the system $multiplier - 1$ additional requests are injected into the system. When the multiplier is < 2 no additional requests are injected.
    - *totalRequestsPerSecond*: Defines how many r/s are desired to be total in the system. If the current request load is lower than this, ExSys takes care about injecting additional shadow traffic.If the current r/s ratio is higher than the desired *totalRequestsPerSecond* ExSys is not doing anything. The experiment results are off in this case, as an incorrect ratio was used. The SLAs prevent ExSys from canceling the request in order to archive the required ratio. The case where more original requests are received by the system than desired and results being off is a known limitation.issues. This parameter is only evaluated for shadow stages with *totalRequestsPerSecond*.

  Shadow traffic is artificial traffic which results are not returned to the client.

6.5.1 Provisioned concurrency

Provisioned Concurrency aims to prevent elasticity issues in serverless applications by diverging from the original serverless approach of provisioning functions only when they are required by the traffic. The former resulting in a big cold start performance penalty. Cold start describes the mechanism of having a serverless VM being started. The platform receives a request for the execution of a serverless function and if not enough VMs are ready to serve the current request load, a new VM will be created. This creation process takes a lot more time than having the request handled by an already created (warm) VM.
Provisioned concurrency aims to solve this problem by pre-warming (starting) a configured amount of functions in order to circumvent the cold start problem. This is diverging from pure serverless, as the number of provisioned concurrency must be configured by the developer resulting in the problem of over- and under fitting as known from technologies before serverless.

In order to evaluate the capabilities of provisioned concurrency circumventing the cold start problem ExSys offers the option to provision concurrency for an experiment. The concurrency has to be set by the experimenter, but in most cases it should be set equal to the highest r/s over all stages. ExSys than provisions the configured concurrency on the PUT /experiments call. The experiment can only be started when the provisioned concurrency is ready and which can take several minutes. ExSys provisions the given concurrency for the proxy function, the original and the experiment function resulting in $3 * preProvision$ (if original and experiment function are equal only $2 * preProvision$) provisioned concurrency. This results in a high cost overhead, as the platforms costs add up over the time of the execution. If this cost overhead is justified needs to be decided by the experimenter.

6.6 Limitations

ExSys allows to have an unlimited amount of experiments running in parallel, but every single experiment can only cover one single function. If the results of various experiments should be combined, this has to be done manually by the experimenter. Synchronizing the start of the different experiments can be difficult and could be supported by adding a timestamp when the experiment should start.
ExSys relies on steady traffic as it utilizes no background worker. It can only do its work when the functions are called as result to incoming requests.
Very volatile traffic can be a problem for ExSys as it might not be able to adapt fast enough, as can be seen in the Get results of experiment. This is less of a problem with longer *time_s* per stage.
If too many original requests are injected the experiment results are off, as no requests can be dropped. There is no possibility for ExSys to generate less traffic than is arriving.

7 Implementation

This section describes a prototype implementation of ExSys on Amazon Web Services (AWS) called ExSysProto. The thesis focus on AWS as it is the biggest (by market share[8]) provider of cloud infrastructure.
The overhead of supporting multiple cloud providers is out of scope and does not influence the significance of the experiment results.
In alignment to the ExSys spec only HTTP request are used as triggers. This narrows the scope to only use AWS API Gateway[9] events as triggers. By its usage of the HTTP protocol it allows using JMeter[10] for reproducible evaluation traffic generation.
The serverless function product in AWS is called AWS Lambda[11] and is used with the Node.js runtime. The code of ExSysProto can be found in the online appendix[12].

7.1 Used technology

The general system design of ExSys is applied in ExSysProto to a concrete implementation. For this implementation an array of products is used:

- Serverless functions: AWS Lambda[13] with a Node.js (version 10) runtime.
- NoSQL Database: AWS DynamoDB[14] with one configured database stream on the ExSysResults table.
- API routing: Amazon API Gateway[15] serving the ExSys User API for deploying and controlling ongoing experiments.

7.2 FaaS function identifiers

In the description of the Experiment config, the *FaaS function identifiers block* is a placeholder for prototype specific parameters as this block depends on how the used platform identifies its functions. In ExSysProto this block looks like the following:

```
"apiGateway": {
"region": "[AWS REGION]",
"restApiId": "[REST API ID]",
"stage": "[STAGE]",
"endpoint": {
"resourceId": "[RESOURCE ID]",
"httpMethod": "[METHOD OF THE ENDPOINT]"
}
"arns": {
"experiment": "[EXPERIMENT FUNCTION ARN]",
"original": "[ORIGINAL FUNCTION ARN]"
},
},
```

All these parameters need to be provided on the creation of a new experiment for ExSysProto to be able to identify which function it should work on. A different parameter in any of these will either make the proxy deployment fail or the experiment will produce an unexpected outcome. The later can not be prevented by ExSysProto as it can only work on the parameters it was provided with. This requires the experimenter to ensure the correctness of the parameters.

Explanation of the different parameters:

[8]https://www.statista.com/chart/18819/worldwide-market-share-of-leading-cloud-infrastructure-service-providers/ Accessed: 2020-05-31
[9]https://aws.amazon.com/api-gateway/
[10]https://jmeter.apache.org/
[11]https://aws.amazon.com/lambda/ Accessed: 2020-07-31
[12]https://github.com/simonfrey/exsys/exsysproto
[13]https://aws.amazon.com/lambda/
[14]https://aws.amazon.com/dynamodb/
[15]https://aws.amazon.com/api-gateway/

- *region*: AWS offers data center in different regions. ExSys must know in what region the system operates, in order to control the functions.
- *restApiId*: Every API of API Gateway gets a unique ID for identification. This ID is used to select the appropriate API.
- *stage*: API Gateway can be deployed with multiple stages. These stages are used to test a different version of the API Gateway. To operate on the intended stage, ExSysProto needs to know the stage name.
- *resourceId*: The resource ID is a unique ID for a HTTP resource inside the API Gateway. A resource is a path e.g. "/products".
- *httpMethod*: For each HTTP method[16] a different serverless function is connected. In order to work on the right one ExSysProto needs to know on which method it should operate.
- *experiment* & *original*: This ARNs specify which functions the experiment runs with. The functions are uniquely identified across the whole Amazon Ecosystem with their Amazon Resource Names. "Amazon Resource Names (ARNs) uniquely identify AWS resources. We require an ARN when you need to specify a resource unambiguously across all of AWS, such as in IAM policies, Amazon Relational Database Service (Amazon RDS) tags, and API calls."[17]. The original function needs to be the function currently defined in API Gateway. The experiment function can be any function which has to be already deployed into AWS. The experiment function is also allowed to be the same as the original function. (As done in the experiments)

7.3 AWS Lambda function aliases

For the deployment of the ExSys Proxy functions ExSysProto uses AWS Lambda function aliases[18]. "A Lambda alias is like a pointer to a specific function version."[19] This is helpful as ExSysProto can point the aliases all to the same version, but encode the experiment id and the region in the alias name. When the proxy function is spawned and run the very first time, it decodes the information from its own alias name and retrieves the experiment configuration from the database. With this mechanism it is possible to reuse the same proxy function for various experiments. The proxy function is only required to be deployed once. This speeds up the setup of an experiment a lot, as the lambda deploy mechanism takes a lot more time than the creation of the alias.
Downside of this mechanism is that the configuration needs to be loaded on the first execution. With a custom build function per experiment the configuration could be integrated directly into the code and safe time on startup. This effect could outweigh the advantage of the re-usability in a high traffic scenario where a lot of proxy functions are spawned, but as still the current experiment state needs to be loaded from the database in the ExSys use-case this effect if marginal and can be neglected.

In the Related work section was set out that AWS offers a percentage based deployment via AWS Codedeploy. This method also works on function aliases[20]. Though the same underlying technology is used for percentage based routing and deployment Codedeploy and ExSys serve two different needs: Codedeploy can be used to gradually deploy a new version of a function and rollback the deployment if the function fails. ExSys on the other hand has no interest in deploying a function into a live state, but run ephemeral experiments. ExSys & Codedeploy can coexist as after a successful experiment on the system the tested function could be rolled out with Codedeploy. Codedeploy lacks the possibility for a shadow deployment and the possibility to inject additional headers into the function. The functionality that could be replaced with Codedeploy would still be required for the shadow deployment and thereby using Codedeploy would not simplify ExSys but rather make it more difficult. The mentioned points and the lack of a granular traffic control during the different stages prevent Codedeploy of being a sensible option for ExSysProto.

[16]https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods Accessed: 2020-7-31
[17]https://docs.aws.amazon.com/general/latest/gr/aws-arns-and-namespaces.html Accessed: 2020-7-31
[18]https://docs.aws.amazon.com/lambda/latest/dg/configuration-aliases.html Accessed: 2020-09-14
[19]https://docs.aws.amazon.com/lambda/latest/dg/configuration-aliases.html Accessed: 2020-09-14
[20]https://docs.aws.amazon.com/lambda/latest/dg/configuration-aliases.html

7.4 Distributed counter

Amazon DynamoDB is a fully managed proprietary NoSQL database service that supports key-value and document data structures[21]
DynamoDB derives from [5] which inspired several other NoSQL databases "such as Apache Cassandra[22], Project Voldemort[23] and Riak[24]."[25].
Most important for the request counters is DynamoDBs atomic counters[26]. This feature offers the possibility to increment a counter without reading its current value. Assuming the consistency model of DynamoDB is working, ExSys can rely on DynamoDB as a single source of truth.

DynamoDB is a proprietary software as of why it is not known exactly how this counter is implemented. [5, p. 6-8] describes DynamoDBs write process. Writes are processed in the order they arrive onto a master node. From there the data is replicated into other nodes. As in-/decreasing of a number are cumulative operations, the order in which the in-/decrease operations arrive to the master are of no difference. As they are not idempotent the network has to make sure that each operation arrives exactly once. This is solved in the transport layer by using TCP.

As the counter is based on a distributed database, the CAP theorem applies to it and it will be further discussed how this influences the results. It is widely used in the research community to discuss limitations of distributed database implementations. The following introduction by Simon Salome describes the assumptions of the CAP theorem:
"A distributed database has three very desirable properties:

1. Tolerance towards Network Partition
2. Consistency
3. Availability

The CAP theorem states: You can have at most two of these properties for any shared-data system. Theoretically there are three options:

1. Forfeit Partition Tolerance
   The system does not have a defined behavior in case of a network partition. Brewer names 2-Phase-Commit as a trait of this option, although 2PC supports temporarily partitions (node crashes, lost messages) by waiting until all messages are received.
2. Forfeit Consistency
   In case of partition data can still be used, but since the nodes cannot communicate with each other there is no guarantee that the data is consistent. It implies optimistic locking and inconsistency resolving protocols.
3. Forfeit Availability
   Data can only be used if its consistency is guaranteed. This implies pessimistic locking, since we need to lock any updated object until the update has been propagated to all nodes. In case of a network partition it might take quite long until the database is in a consistent state again, thus we cannot guarantee high availability anymore.

" [26, p. 2]

DynamoDB offers Eventually Consistent Reads & Strongly Consistent Reads with the default being Eventually Consistent Reads: "When you read data from a DynamoDB table, the response might not reflect the results of a recently completed write operation. The response might include some stale data. If you repeat your read request after a short time, the response should return the latest data."[27]. The counter in DynamoDB has been "choosing to go with AP (availability and partition-tolerance) and give

[21] https://aws.amazon.com/dynamodb/faqs/ Accessed: 2020-07-27
[22] https://cassandra.apache.org/
[23] https://github.com/voldemort/voldemort
[24] https://riak.com/
[25] https://en.wikipedia.org/wiki/Dynamo_(storage) Accessed: 2020-07-27
[26] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/WorkingWithItems.html#WorkingWithItems.AtomicCounters
[27] https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.ReadConsistency.html Accessed: 2020-07-27

up strong consistency and general distributed transactions"[1, p. 2] in the therms of the CAP theorem. This yields in having only a loose guarantee on the accuracy of the data. As speed is more important for ExSys than 100% correct counter values this is a good trade off.

7.5 Limitations

ExSysProto can only be deployed into the us-east-1 region. This does not affect the validity of the experiment results presented in the thesis, but may be a limitation for a wider use.

7.6 Deployment

ExSysProto is written in TypeScript[28]. After compiling it to JavaScript it is deployed with the major Node.js version 10, the memory size for the single functions is 1024 MB and a timeout of 30 seconds is selected. The timeout should be adapted to the function under test in order to prevent timeouts in ExSysProto, that would happen e.g. if the original function runs for 50 seconds, but the proxy times out after 30. The deployment is done with the Serverless framework[29].

After deploying ExSysProto into AWS no further deployment steps are required as of the used AWS Lambda function aliases. All further operations are done from this point on via User API.

[28]https://www.typescriptlang.org/
[29]https://serverless.com

8 Evaluation

8.1 Application scenario

The case study application from Bifrost[30] is used as base application as ”Unfortunately, few suitable open source microservice-based applications exist [...] [and that] application simulates a generic e-commerce website selling consumer electronics. It was kept simple in order to provide a testbed for the performance evaluation” [29, p.7]. As pointed out in [2] the majority of users use serverless applications for APIs and web applications benefit from non-functional regression testing because of their rapid changing nature which makes the application a valid use-case for online experiments. The application is adapted to fit into the AWS Lambda infrastructure. The code of lambdafrost can be found in the online appendix[31]

8.1.1 Structure of the application

The lambdafrost application offers an API for accessing a rudimentary online shop. The shop sells laptops and it offers the functionality to browse different items, search in the product description and purchase an item with the currently signed-in user. There is no inventory management, so a single item can be purchased an infinite amount of times and never runs out of stock. The original bifrost case study application offered a frontend as well. As a fronted is of no use to the experiments in the thesis it was not adapted for lambdafrost. The API has the following endpoints:

- **/seed** PUT: *Set the database to the initial data state*
- **/customers/login?email=[USER EMAIL]&password=[USER PASSWORD]** GET : *Login user with the given credentials*
  Response body:

```
{
  "email": "...",
  "jsonwebtoken": "..."
}
```

- **/products** GET* : *Get all available products*
  Response body:

```
[
  {
    "category": "...",
    "description": "...",
    "id": "...",
    "name": "...",
    "image": "..."
  },
  ...
]
```

- **/products/{id}** GET* : *Get information about single product*
  Response body:

```
{
  "category": "...",
  "description": "...",
  "id": "...",
  "name": "...",
  "image": "...",
  "buyers":0
}
```

[30]https://github.com/sealuzh/bifrost-microservices-sample-application
[31]https://github.com/simonfrey/exsys/lambdafrost

- **/products/{id}/buy** POST* : *Buy the product with the current logged in user*
  Response body:

```json
{
  "category": "...",
  "description": "...",
  "id": "...",
  "name": "...",
  "image": "...",
  "buyers":0
}
```

- **/products/{id}/buyers** GET* : *Show all buyers of the product*
  Response body:

```json
[
  {
    "product": "...",
    "id": "..."
  },
  ...
]
```

All endpoints with a * attached to their method need to be authorized. This authorization is done via the JSON web token received from the */customers/login* endpoint. This web token must be provided in the authorization header[32] to the endpoints (*Authorization: Bearer [JSON webtoken]*)

8.1.2 Modifications

As stated in the structure paragraph the frontend of the case study application is not required for the thesis. The overhead of porting it to a recent tech stack is not justified and thereby the frontend is removed.

Bifrost works on a microservice architecture, which required the case study application to have certain infrastructural overhead. The login and search have both been separate services and the product service internally accessed these two services via their API. This is a good idea in a microservice architecture in order to have separated concerns and clean data separation. This overhead was removed as in the serverless architecture they both access the same database anyways and thereby separation of concerns is inherently broken and the overhead is of no advantage. The functions in lambdafrost directly access the database in order to enable a search and to check if the authorization header is valid. Because of these changes the two endpoints */search* and */customers/checkLogin* are no longer required and removed.

In comparison to the deployment mechanism in bifrost the seeding of the database can not be done at the moment of deployment. To still enable the experiment to start from a clean and reproducible start point the */seed* endpoint is added. When calling it the database is set to a deterministic start state.

As the metrics for lambdafrost can be gathered from within the AWS infrastructure or ExSys all */metrics* endpoints are removed and the Prometheus instances are not deployed.

The case study application in bifrost uses MongoDB[33] as persistence layer. MongoDB is no native Cloud Storage to AWS and requires specialized deployment. There is an MongoDB compatible AWS service: DocumentDB[34]. As stated by MongoDB the AWS DocumentDB is only 37% compatible and "Any existing MongoDB apps relying on this functionality would need to be re-engineered if they are to be migrated to DocumentDB"[35]. This results in two possible scenarios for using MongoDB with lambdafrost:

[32]https://developer.mozilla.org/en-US/docs/Web/HTTP/Authentication Accessed: 2020-09-14
[33]https://www.mongodb.com/
[34]https://aws.amazon.com/documentdb/
[35]https://www.mongodb.com/atlas-vs-amazon-documentdb/compatibility Accessed: 2020-05-22

- Use DocumentDB with incompatibilities and the requirement to test if the application works as intended
- Host MongoDB as own service on AWS EC2 as described in the AWS quickstart[36]

Both solutions come with a deployment overhead. The most widely used database for serverless applications on AWS is DynamoDB[37]. It does not need to be deployed and can be used directly via the AWS SDK. In comparison to MongoDB it offers a HTTP API which does not need an open database connection, this is beneficial for the cold start times of the VMs, as no connection needs to be opened up and maintained. Additional this circumvents MongoDB problem that only a (high but) limited amount of open connections to the MongoDB are possible. DynamoDB is widely used for AWS lambda applications. As of the aforementioned points lambdafrost uses DynamoDB instead of MongoDB as persistence layer. This comes with a performance hit for the */products/search* endpoint, as DynamoDB does not offer a full text search on columns. The lambdafrost search fetches all product entries and does the full text search within code. As only 100 products are in the case study application, the overhead of doing so is of no major impact to the results. This has be verified with the following client side benchmark:

1. GET request against */products/{id}* return a single product (direct database access)
2. GET request against */products/search?query=mother* returning 16 products with the aforementioned in-code full-text search
3. Repeat step 1 & 2 for 10 times

Run	Endpoint	Response time (ms)
0 (cold start)	/products/{id}	1404
1	/products/{id}	363
2	/products/{id}	475
3	/products/{id}	435
4	/products/{id}	516
5	/products/{id}	312
6	/products/{id}	469
7	/products/{id}	417
8	/products/{id}	516
9	/products/{id}	511
10	/products/{id}	415
0 (cold start)	/products/search?query=mother	1405
1	/products/search?query=mother	397
2	/products/search?query=mother	441
3	/products/search?query=mother	222
4	/products/search?query=mother	403
5	/products/search?query=mother	493
6	/products/search?query=mother	444
7	/products/search?query=mother	417
8	/products/search?query=mother	452
9	/products/search?query=mother	300
10	/products/search?query=mother	492

As can be seen from the result table, averaging the runs 1-10 (cold start is not taken into consideration as it is expected to be higher and would change the average in a bad way. The cold start of both functions only differs by 1 ms) the response time for */products/{id}* is 442ms and for */products/search?query=mother* it is 406ms. In the experiment the average response time of the search was even faster than the more performant direct access on a single product. This proves the hypothesis that the overhead of the full text search in code can be neglected for the 100 products.

Lambdafrost makes use of the additional header injected by ExSys (see Additional headers) to prevent side effects for artificial traffic. This side effect prevention is only important for the */products/{id}/buy*

[36]https://docs.aws.amazon.com/quickstart/latest/mongodb/overview.html
[37]https://aws.amazon.com/dynamodb/

endpoint as it is modifying the database. (This endpoint is not used in the experiments)
If the ExSys header is present, the handler returns a result object containing the following additional information:

- *timeout* & *memory* are derived from the context passed into the handler. They represent the configuration of the lambda function
- *version* is a hard coded value that needs to be changed by the developer when the function is updated.
- *vm_id* is an id generated for every unique VM. With it the experiment tracks on what VM a certain request did run and how many new VMs are created.
- *boot_time* is a timestamp generated on booting the VM the first time.
- *new_vm* is a boolean signaling if this is the first call to this certain VM.

These values are similar to the ones used within [19] in order to ensure a comparability with its elasticity experiment. In comparison to [19] the *new_vm* parameter is added and used for signaling the new creation of a VM. As ExSys is running under the assumption of a live workload it can not use the status code for this information(how it is done in [19]) as if it would do that, the experiment would produce user facing side effects which have to be prevented.

8.1.3 Deployment

Lambdafrost is written in Node.js and deployed with the version 10, the memory size for the single functions is 1024 MB and a timeout of 30 seconds is selected. These parameters are similar to the ones in [19] in order to allow comparability of the results. The deployment is done with the Serverless framework[38].
Because of the 30-second timeout the standard timeout (see Deployment) in ExSysProto does not need to be modified.

8.2 Traffic generator

In order to generate test traffic for the experiment Apache JMeter[39] is used. This tool is also used in the Bifrost paper. The basic test suite[40] from Bifrost is used for traffic generation. It is adapted to fit to the lambdafrost API. The following three modifications were done:

- Adapt the paths to lambdafrost
- Call */seed* endpoint before the experiment start to ensure a clean data basis
- Call gets all products (GET */products*) endpoint, as the proxy function is injected there and all the other endpoints are not measured anyways. This saves cost for the experiment and allows to have a more accurate request count on the measured function. Before changing to this strategy only about 1/3 of the test traffic arrived at the function which reduced the data quality.

The test suite simulates X_u users logging into the online store and randomly informing themselves about a product. The simulated users do this over and over again. This simulates a stead access on lambdafrost and is a valid traffic pattern. The traffic scenario has no fluctuation simulating an optimal test scenario, which is good for the thesis. Fluctuation in the traffic and how ExSys could adapt to this could fuel a followup research and is out of scope for this thesis.
The responses of the API are stored. With this data user facing side effects are measured. Important values for the measurement of user facing side effects are the status code and the response time. If the status code is not a 200 (HTTP OK) this would indicate that the experiments break the system and if the response time increases a lot that could render ExSys useless, as the SLAs for the applications could not be kept.
The traffic generator is executed on a personal computer with a 50mbit internet connection. The experiments are conducted in close time after each other. A small natural network delay is inherent to all requests and can be neglected.

[38]https://serverless.com
[39]https://jmeter.apache.org/
[40]https://github.com/sealuzh/bifrost-microservices-sample-application/blob/master/jmeter/jmeter.jmx

8.2.1 Result format

JMeter writes its results with following values per request:

- timeStamp: Request start timestamp in ms
- elapsed: Time the request took in ms
- responseCode: Returned http status code
- threadName: Name of the thread (format: [thread group]-[thread])
- bytes: number of bytes received
- grpThreads: number of threads in thread group
- allThreads: number of all running threads
- URL: Url of the request
- SampleCount: Amount of requests in this call
- ErrorCount: Amount of errors in this call
- IdleTime: Time the request was idle in ms
- Connect: Time it took to connect to the HTTP endpoint in ms

When referred to client results these values are the base.

8.3 Experiment setup

8.3.1 Steady "live" traffic

The same steady workload as described in Traffic generator is used. All amplification and traffic shaping in order to test the system is done within ExSysProto and does not require a change in the client generated traffic. The incoming traffic for both experiments is equivalent but not the same. For both experiments three traffic scenarios are tested: $X_u = 10$, $X_u = 35$, $X_u = 90$.
In order to allow a clean beginning state, the system under test is removed and re-deployed in between the different traffic scenarios. Without this cleanup there could be still additional VMs in the system which would distort the experiment results as scaling would be better than expected.

Only a single endpoint of lambdafrost: *GET /products* is experimented on and only a single experiment is injected into lambdafrost in order to prevent cross-influences between experiments. The endpoint is chosen as it is highly frequented (see Traffic generator) and does not produce any side effects as of why the experiment function can be exactly the same as the original function. For the validity of the experiments conducted the endpoint is of no certain influence, as ExSysProto and the platform are the entities being tested and not the single handler function.

The ExSys proxy is deployed via *PUT /experiments* after JMeter did run for 15 seconds. After 10 more seconds the experiment is started via *GET /experiments/{id}/start*, with the id received from the previous PUT request.

8.3.2 Experiment 1: Elasticity experiment

This experiment tries to replicate part of the elasticity experiment implemented by Kuhlenkamp et al. in [19] and thereby show that ExSys enables online elasticity experiments.

The paper questions the following premise: "FaaS platforms promise the core capability to execute code without requiring a FaaS platform client to manage resource allocations. Thus, a cloud automatically provisions resources to execute application code with incoming events. In an application architectural context, this capability implies that a FaaS platform provides an option for throughput-based application objectives. We refer to this option as elasticity."[19, p.2]

Only a part of the experiment is replicated, by using just a single workload scenario (WL3 [19, p.4]) instead of all five introduced by Kuhlenkamp et al. as the goal is to test if ExSys enables elasticity experiments and not to completely ver-/falsify all experiments done in the paper.

The "workload model uses three sequential phases (P0-P2): warm up phase (P0), scaling phase (P1), and cool down phase (P2)"[19, p.4].
The three phases have a defined duration in the Kuhlenkamp et al. paper: "[...] the warmup phase P0 [...] [and] the scaling phase P1 last[...] for 60 seconds. During the scaling phase P1, we increase trps for each interval by a constant factor of 0.5, 1, or 2. Thus, we increase trps for 60seconds linearly.[...] the duration of the cooldown phase (P2) is 180 seconds."[19, p.4]
The chosen workload WL3 has following target r/s for the different phases:

1. Phase 1 (warm up): 60 r/s
2. Phase 2 (scaling): $60 + [0.5 * t]$ r/s with t being the elapsed time since the phase start in seconds.
3. Phase 3 (cooldown): 75 r/s

This results in an ExSys configuration with one stage for phase 1, 60 stages for phase 2 and one stage for phase 3. All additional traffic produces is shadow traffic. The complete config executed can be found in the online appendix[41].
The minimum requests per second count in the experiment being 60 renders the $X_u = 90$ traffic scenario particularly interesting as there are already more request in the system than desired. It is evaluated if the experiment results can deliver any value to the experimenter or in that case are nonsense.

8.3.2.1 Differences to the base experiment

In comparison to [19] the experiments run in a live scenario as of why it can not be assumed that there will ever be 0 r/s. Only additional traffic is possible and not drop any. This results in only the workload scenarios WL3 & WL4[19, p.4]) being feasible to be evaluated. They work on a minimum of 60 r/s. By only evaluating workload scenario WL3 the shortcoming of requiring 0 r/s are circumvented.

In addition to the three phases in the base experiment a fourth one is added *Phase 0 (ExSys warm up)*: 20 sec without any additional traffic. This phase is used to allow ExSys to get accurate data about the current load on the system and to allow the platform to adapt to the base workload. The early experiments conducted did not have this phase which resulted in ExSys over-&underfitting the traffic in the beginning of an experiment. More significant was the problem the platform had. As stated in [19] it is hard for the platforms to stick to their promise of instant scalability. The platform was not able to adapt fast enough to the base traffic and with the additional traffic injected by ExSys this got even worse. With this additional phase the platforms has 20 seconds time to adapt the VMs to the basic workload. In a real system there would be already steady base traffic and the platform would already have been adapted. As of this reason this additional stage does not influence the experiment results negatively. In the evaluation this phase is skipped and not visualized in the graphs. It can be found in the data in the online appendix[42] if it is of interest.

8.3.3 Experiment 2: ExSys overhead experiment

This experiment tries to evaluate the impact that ExSys has on the system and if it can fulfill its performance requirements. It is based on the experiment in the bifrost paper [29, p.8] which consists of four phases:

1. Canary launch (60s): Redirects 5% of the as canary traffic to the test function
2. Dark launch (60s): Redirect 100% of the traffic with a multiplier of 1 to the test function
3. A/B Test (60s): Redirect 50% as canary traffic to the test functional

[41] **https://github.com/simonfrey/exsys/experiment_configs/elasiticity.json**
[42] **https://github.com/simonfrey/exsys/experiment_results/elasticity**

4. Gradual rollout (200s): Gradually increase the canary traffic to the test function from 5% to 100%. Increase by 5% every 10 second.

This results in an ExSys configuration with one stage for phase 1, 1 stages for phase 2 and 1 stage for phase 3 and 20 stages for phase 4. The complete configuration executed can be found in the online appendix[43].

"Note that, in order to compress the total duration of the experiment to 380 seconds, we chose extremely short execution times for each phase. Obviously, in practice, developers would typically choose longer durations for each phase. We initiated the execution of the live testing strategy after a ramp up period of 30 seconds to slowly increase the load and after an additional 60 seconds for health checking the deployed services. After the ramp up, a steady traffic of 35 requests per second was simulated using a JMeter test suite."[29, p.8]

8.3.3.1 Differences to the base experiment

In differentiation to the experiment in Bifrost not different code as experiment function is deployed but the same as the original function (by setting the *arns.experiment* equal to the *arns.original*). The main objective of this experiment is to measure potential overhead of ExSys and adding the extra variable of a different codebase could distort the result times. This would influence the experiment results and the validity of the experiment negatively.

In addition to the 35 r/s tested within the bifrost paper the experiment is also executed for 10 and 90 r/s.

Same as for the elasticity experiment an additional fourth phase is added: *Phase 0 (ExSys warm up)*: 20 sec without any additional traffic. It is added as of the same reasons as discussed for experiment 1.

8.3.4 Result data

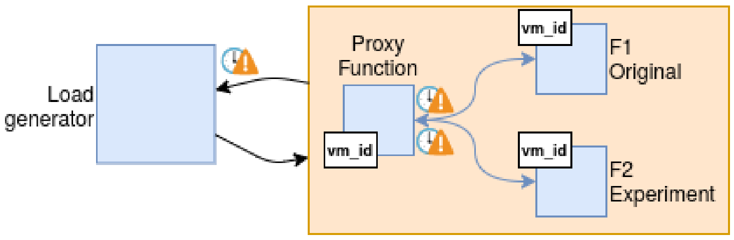


Figure 8: Result data collection points

Result data is collected at three different points for both experiments:

- Client side data collected by JMeter including the response code and request time. For in-depth info on the collected data see:Result format
- Data collected by the ExSys proxy including the errors and the time for the function execution. Additionally, the vm_id is saved of the proxy. For in-depth info on the collected data see: Get results of experiment
- Function information collected by the function itself and returned in the ExSys result header. Most important for the Discussion is the vm_id. For in-depth info on the collected data see: Modifications.

The insights of these results are combined for the evaluation in Discussion

8.3.5 Weaknesses and strengths of the chosen methods

[43]**@github.com/simonfrey/exsys/experiment_configs/overhead.json**

8.3.5.1 Strengths

The client side measurement with JMeter is a stable solution and as the configuration from bifrost is used the traffic scenario was already used in a well cited paper. With sticking with that method there is a high confidence in the client side results generated.

By testing the same traffic scenario with three different user counts a miss correlation between the results and the ingoing traffic amount is prevented.

8.3.5.2 Weaknesses

A problem for the validity of the experiments could be the uniform traffic which might not simulate a real world traffic scenario good enough. It is important to have this fact in mind when interpreting the experiment outcomes.
Despite traffic fluctuation not being configured, the personal computer introduces it as of performance and network issues. The usage of a personal computer instead of a server could be a thread to validity as the traffic generator is not performant enough (see Traffic generator issues)
By relying on live traffic the experiments could be significantly influenced by cold start times in case of a small amount of requests as the platforms scale down the VMs in between. This effect should not play a big role in the experiments as the steady traffic generated and the short test period (< 5 min) prevents the platform from doing so.

8.4 Results

In the following section the results of the conducted experiments are presented. All data and the scripts to generate the graphs are provided in the online appendix[44].

The figures show five or six graphs, correlating in different areas.

- **Top**: Displays the results measured from within ExSys. The lines represent the different traffic methods:
  - Base: The traffic generated via JMeter from the outside
  - Expected total: Is the total traffic expected by the experiment plan. In the case of canary traffic & shadow traffic defined by a multiplier it is irrelevant as the expected always equals the total traffic.
  - Shadow: The traffic generated via the shadow traffic strategy
  - Total: The real total traffic in the system. The aim is to have total = expected total
- **Middle**: The second graph visualizes the client visible side effects by displaying the r/s measured by the JMeter client. The different lines represent different HTTP status codes[45]. If the r/s change a lot or the error rate increases ExSys influence on client visible side effects should be considered high.
- *Middle 2 (only for 10 r/s)*: The total requests in system, visualizing the total request expected in the system and the actual request. This displays the inaccuracy of ExSys.

- **Bottom left**: Active VMs: Distinct VMs being active per second. The different VMs are identified by the *vm_id* in the *res* object injected by lambdafrost. This is a good metric for checking the platform's elasticity behavior. The yellow line is the VM count of the proxy functions (determined by the proxyID).
- **Bottom middle**: Average response time: The average response time measured by JMeter on the client side. This is another indicator for client visible side effects. If the response time increases this can break SLAs
- **Bottom right**: Average lambda execution time: The average time a function needs for execution. This correlates with the average response time and is a indicator for the platform having issues during scaling up the VMs.

[44]**https://github.com/simonfrey/exsys**
[45]https://ietf.org/assignments/http-status-codes/http-status-codes.xml Accessed: 2020-08-31

Converse to the expectations there is no additional error count recognizable at the client side at seconds 15 & 25 when the ExSys proxy was deployed and started.
All three experiments show that ExSys introduces no client visible side effects for the shadow deployment strategy as neither the average response nor the error count is increasing.

The graphs below are Gaussian smoothed[46] with the following normalized kernel:

0.039999999999999994
0.07999999999999999
0.12
0.15999999999999998
0.19999999999999996
0.15999999999999998
0.12
0.07999999999999999
0.039999999999999994

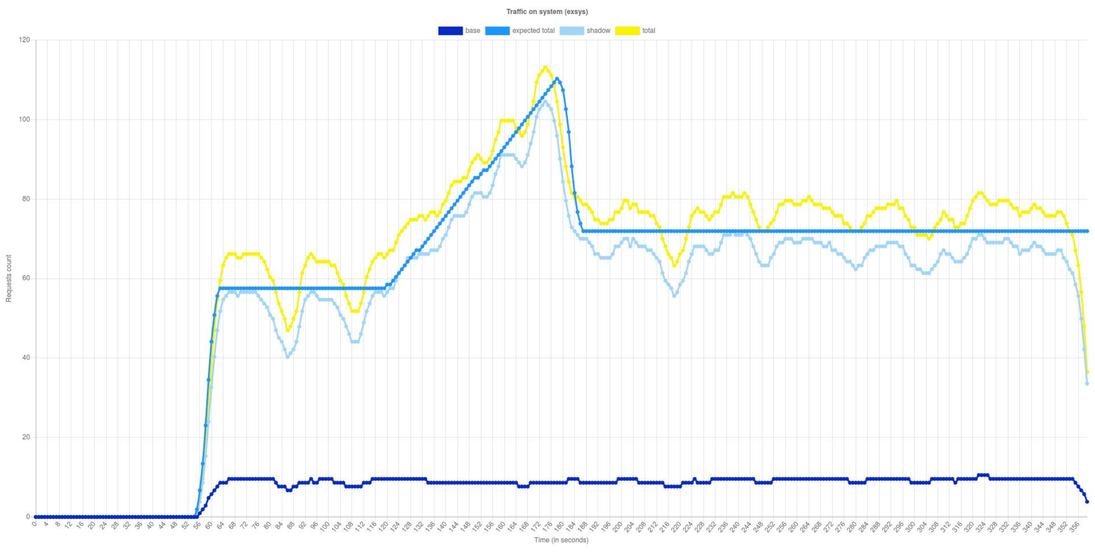
This smoothing makes the data better readable as mainly the overall trend is displayed. The unsmoothened data can be found in the online appendix[47].

8.4.1 Results of Elasticity experiment

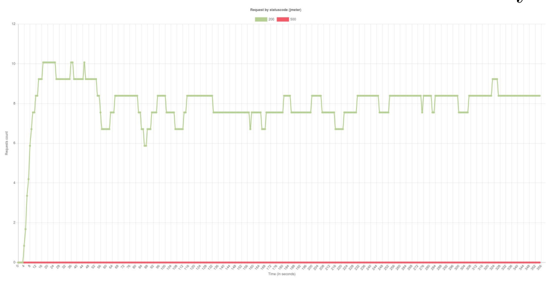
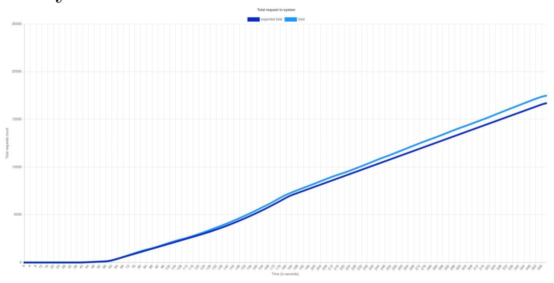
This experiment makes heavy use of shadow traffic which causes the following significant result: There is more traffic in the system than expected. This happens as the distributed counter, the injection of shadow traffic relies on, is not fully accurate all the time. The in-accuracy of the counter and the resulting traffic can be a problem for the experimenter and is further discussed in Distributed counter issues.

[46]https://en.wikipedia.org/wiki/Gaussian_blur Accessed: 2020-09-02
[47]**https://github.com/simonfrey/exsys**

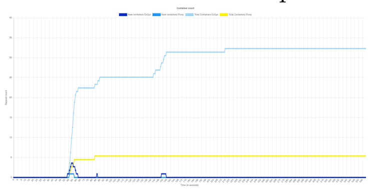
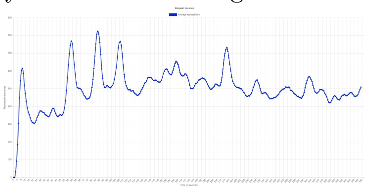
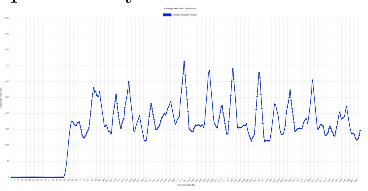
10 r/s



Traffic on system by method

Left: Responses by status code — Right: Total requests in system

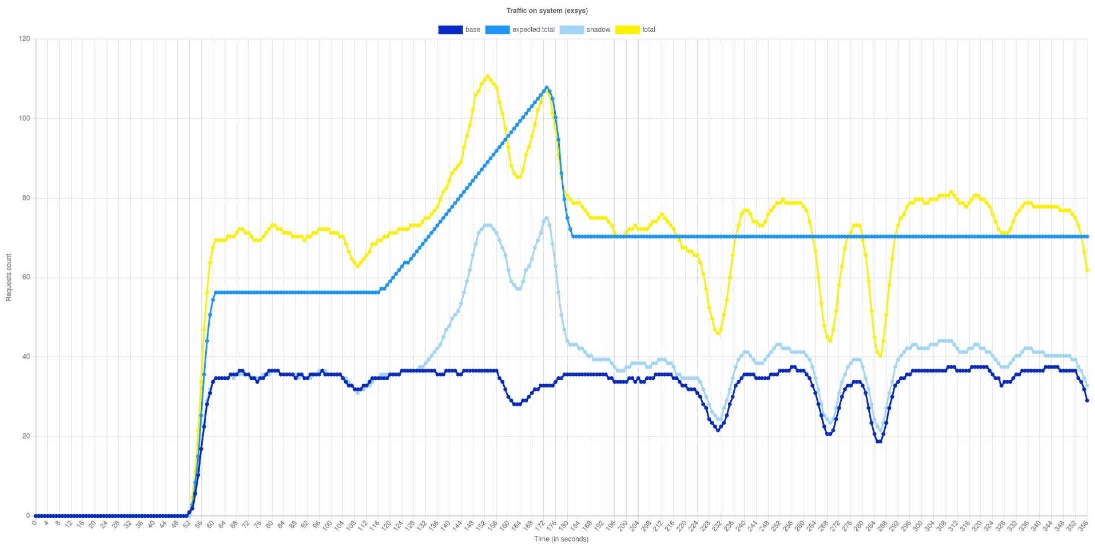




Left: Active VMs — Middle: Average response time — Right: Average lambda execution time

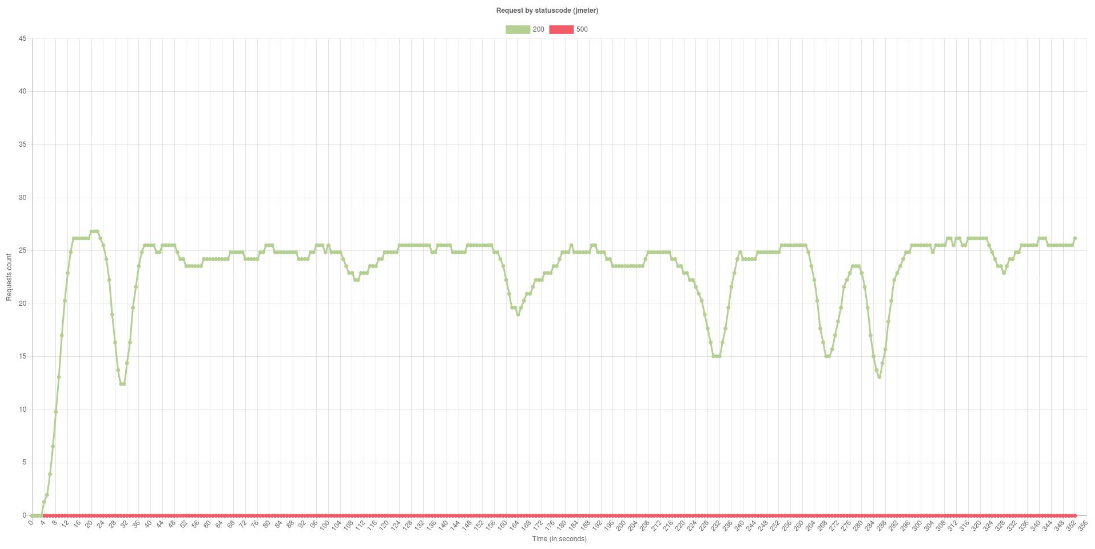
Figure 9: 10 r/s elasticity experiment results

Analyzing the traffic graphs in 10 r/s elasticity experiment results it is recognizable that the target traffic shape is met. As expected and described above the traffic is not exactly accurate, but can be considered good enough. The traffic generated too much is the result of the aforementioned inaccuracy of the distributed counter.
From the standpoint of this experiment ExSys would be a viable solution for live traffic experiments on serverless infrastructure. A complete different picture is painted in 35 r/s elasticity experiment results & 90 r/s elasticity experiment results as discussed later on.

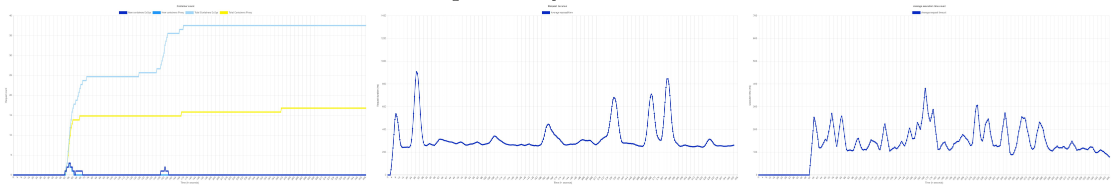
35 r/s



Traffic on system by method



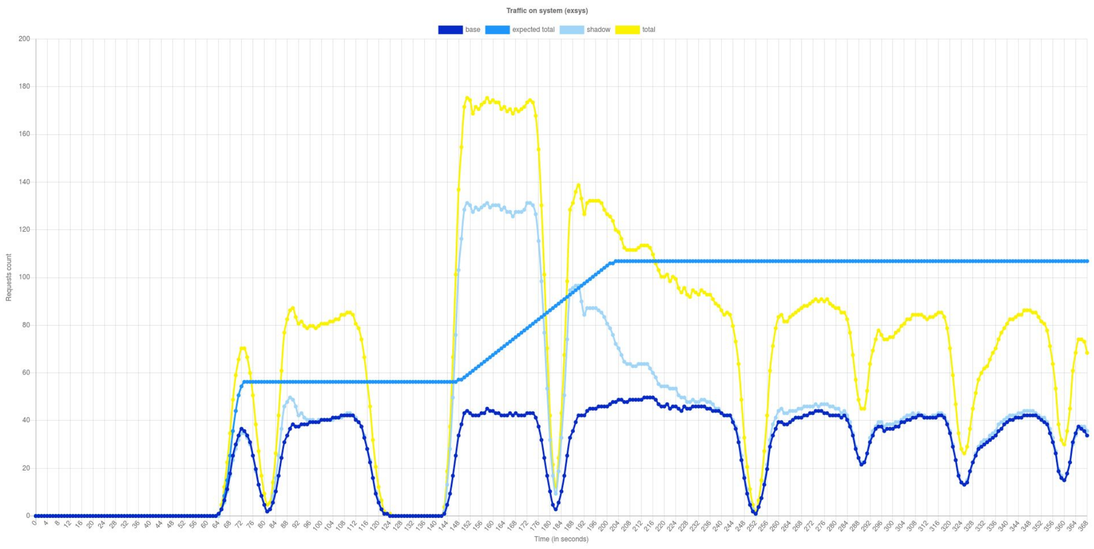
Responses by status code



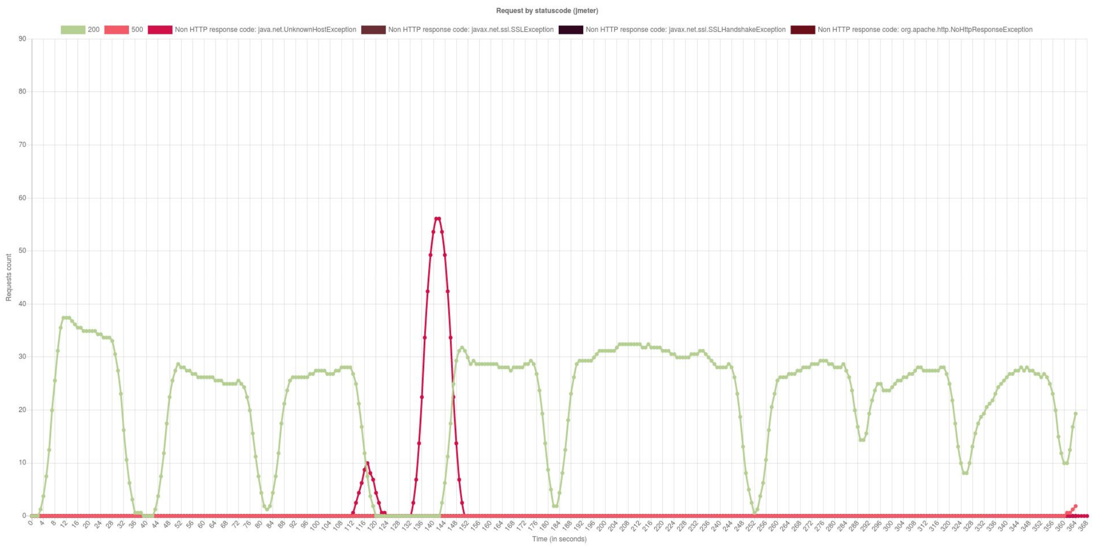
Left: Active VMs — Middle: Average response time — Right: Average lambda execution time

Figure 10: 35 r/s elasticity experiment results

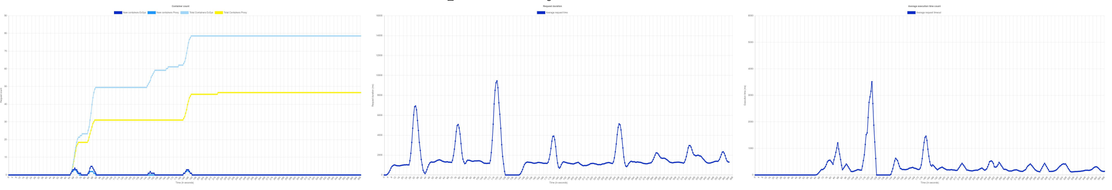
90 r/s



Traffic on system by method



Responses by status code



Left: Active VMs — Middle: Average response time — Right: Average lambda execution time

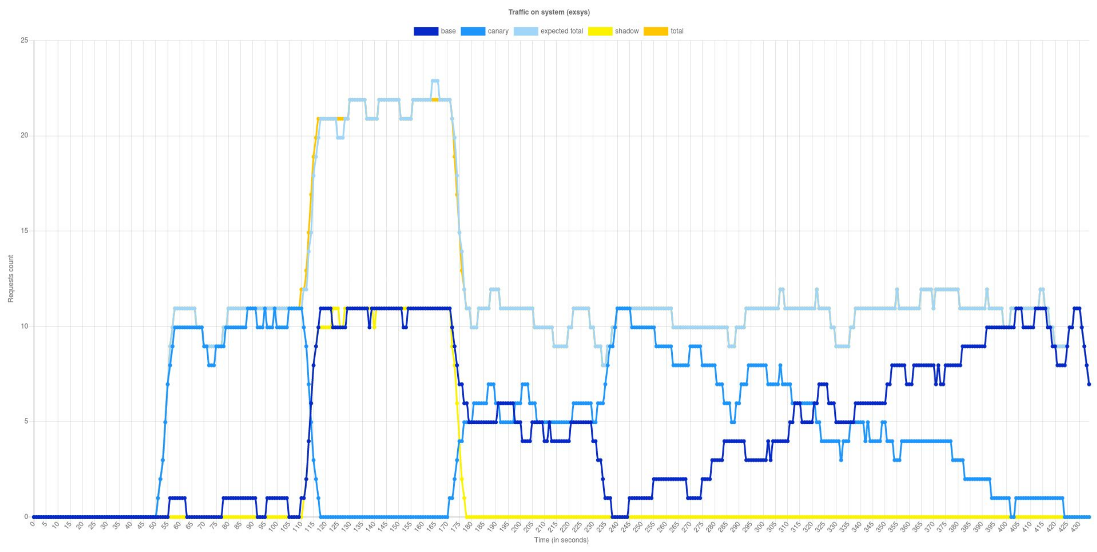
Figure 11: 90 r/s elasticity experiment results

Inspecting 35 r/s elasticity experiment results & 90 r/s elasticity experiment results it is recognizable that ExSys is not able to meet the expected traffic requirements at all. A general trend is recognizable that ExSys still somewhat adapts to the traffic requirement, but the drift has a too big impact.

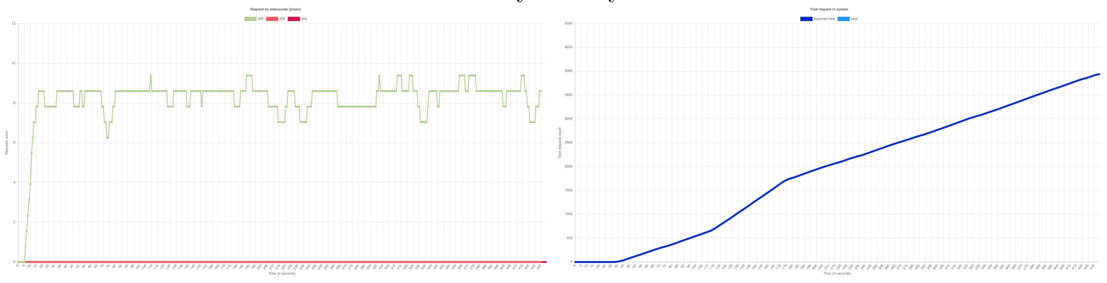
Additionally in the 90 r/s elasticity experiment results it can be seen, that the traffic drops to zero several times. This is due to the fact of the traffic generator on the personal computer not being able to keep up with the traffic load. Also there is no moment, when 90 r/s are generated by the load generator. For this experiment we can state, that the personal computer is not powerful enough and further research should run its traffic generator on a more powerful machine to test higher workloads. As the 90 r/s experiment was not meant to produce a valid experiment output anyways (as the base traffic is already higher than the intended 60 r/s) this does not invalidate the experiment results gathered in the 10 r/s and 35 r/s experiments.

8.4.2 Results of ExSys overhead experiment

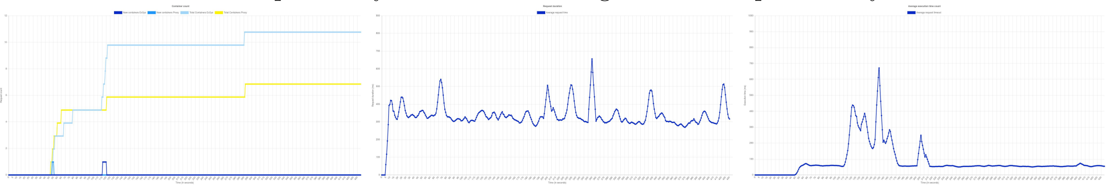
10 r/s



Traffic on system by method



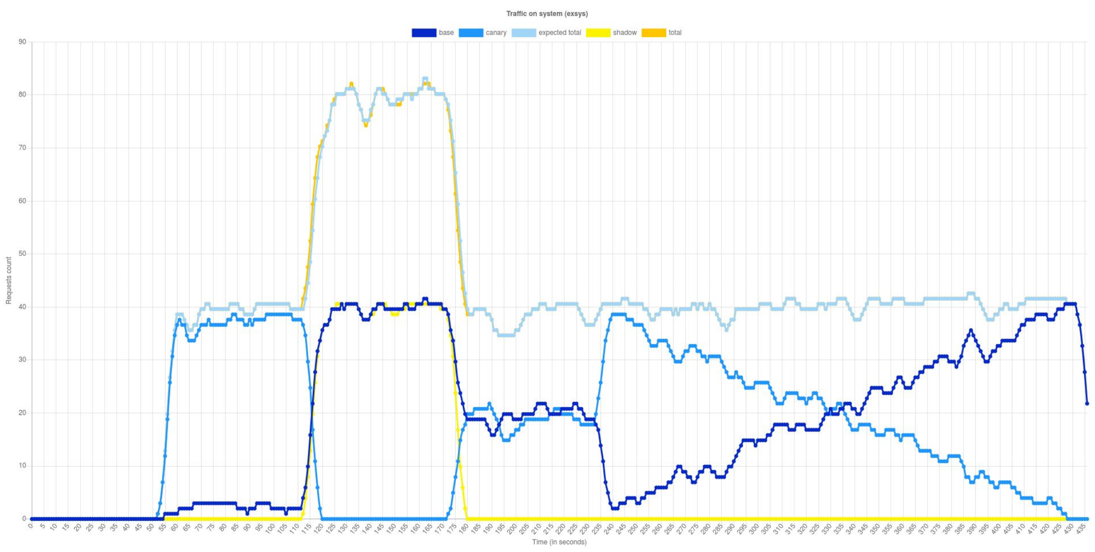
Left: Responses by status code — Right: Total requests in system



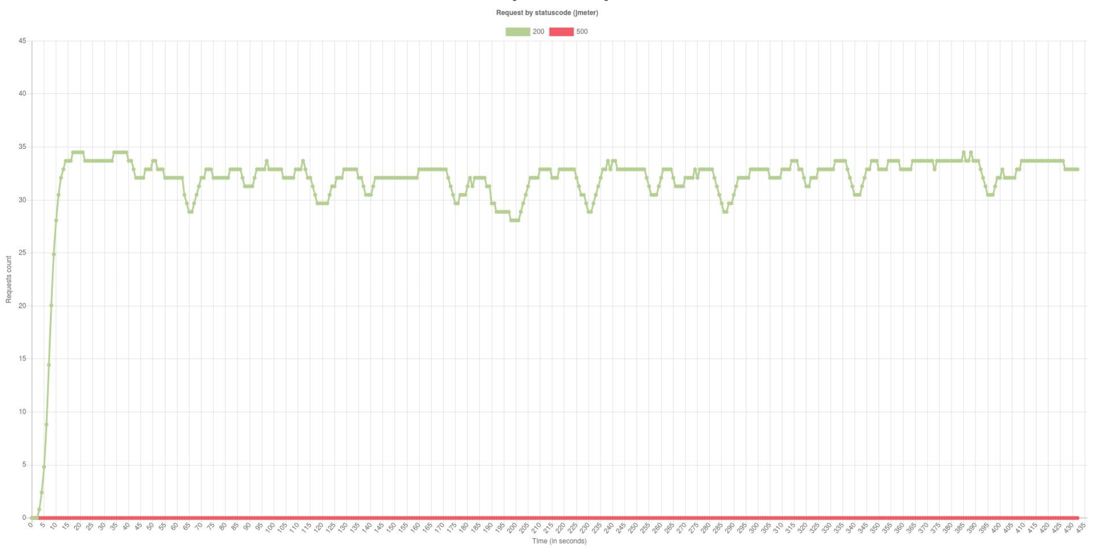
Left: Active VMs — Middle: Average response time — Right: Average lambda execution time

Figure 12: 10 r/s overhead experiment results

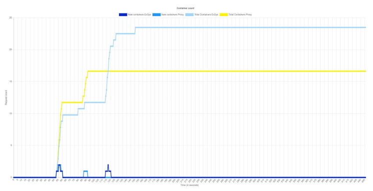
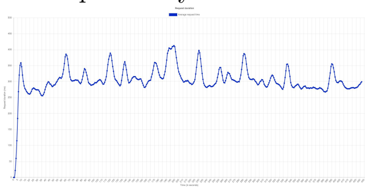
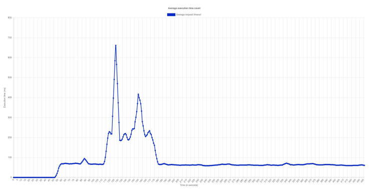
35 r/s



Traffic on system by method



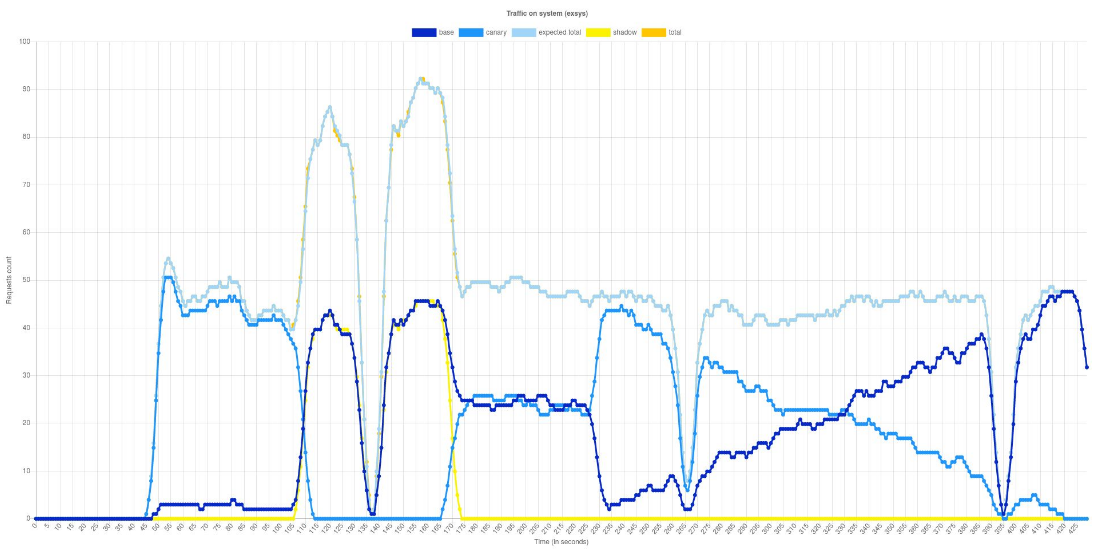
Responses by status code

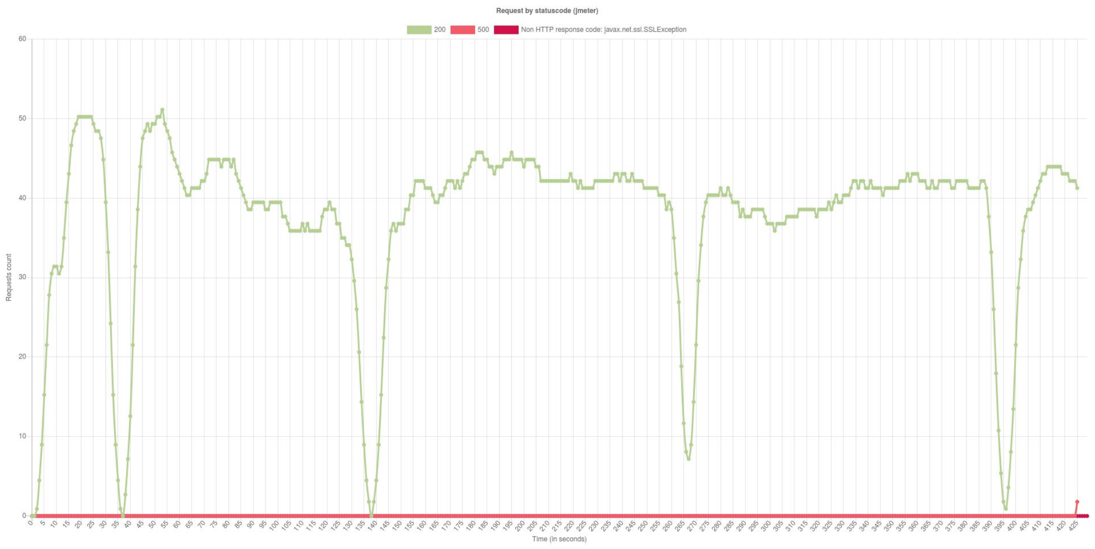
Left: Active VMs — Middle: Average response time — Right: Average lambda execution time

Figure 13: 35 r/s overhead experiment results

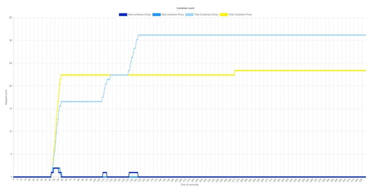
90 r/s

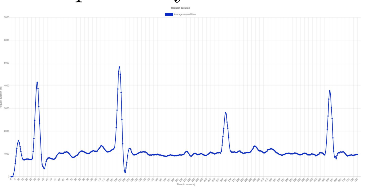


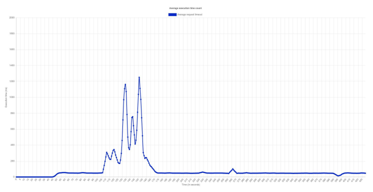
Traffic on system by method



Responses by status code







Left: Active VMs — Middle: Average response time — Right: Average lambda execution time

Figure 14: 90 r/s overhead experiment results

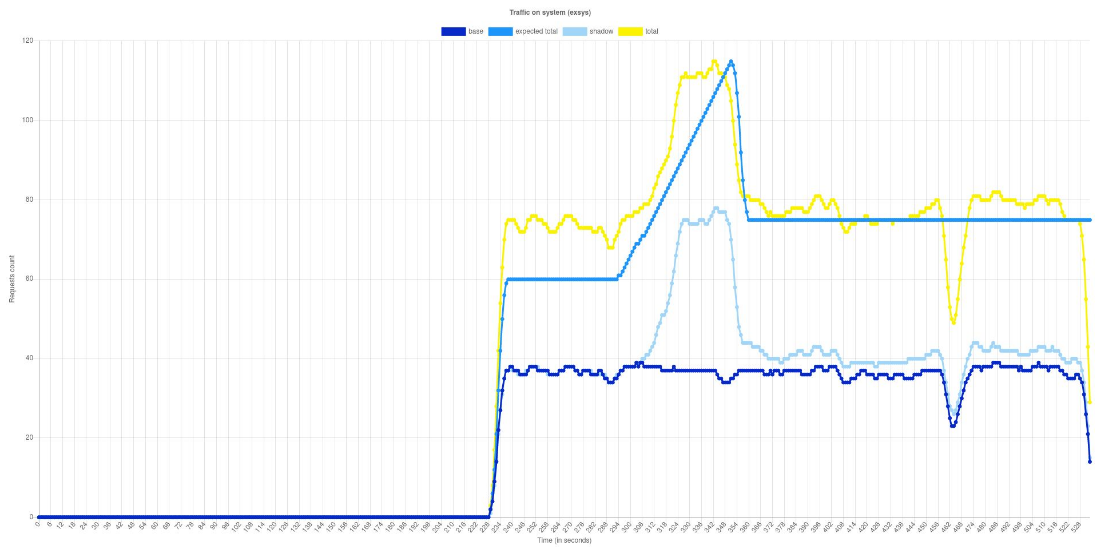
This results of the overhead experiment are a bit confusing as the expected total and the total are most of the time exactly the same as of why the total (yellow curve) is not displayed. Only in the stages where shadow traffic is used, the total drifts a little. In 10 r/s overhead experiment results in the later part of the diagram the shift from base to shadow traffic is clearly visible and ExSys performs quite good on this task. A tendency of this change is also visible in 35 r/s overhead experiment results but a lot less clear. In 90 r/s overhead experiment results the elasticity problems (see in the following section) is too big and the results are completely off. In comparison to the Results of Elasticity experiment the targeted traffic is met, which was expected as the experiment definition only uses relative values as of why whatever traffic patterns occur, by experiment definition they would be correct. This reduces the main insight of the experiment to the proof that canary traffic is possible and the performance overhead of ExSys is minimal as discussed in Overhead.

Similar to experiment 1 the 90 r/s experiment fails to produce the required load and the traffic drops several times to zero as of the aforementioned personal computer inabilities.

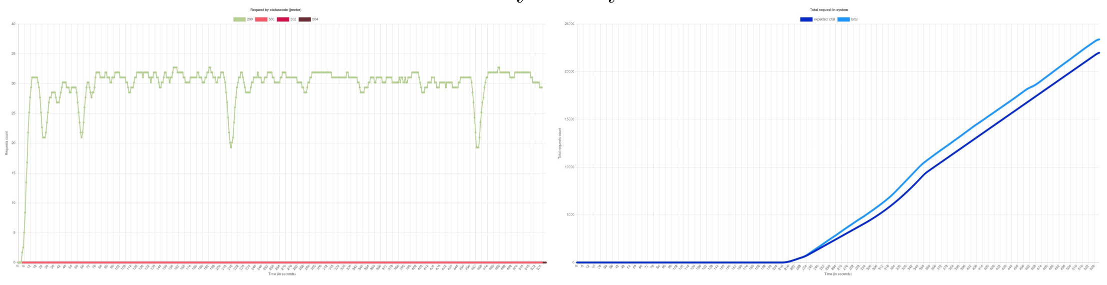
8.4.3 Results of provisioned concurrency experiment

In order to check if the Platform elasticity issues can be solved with the Provisioned concurrency another run of experiment 1 was conducted. The elasticity experiment was run with 119 provisioned concurrencies.

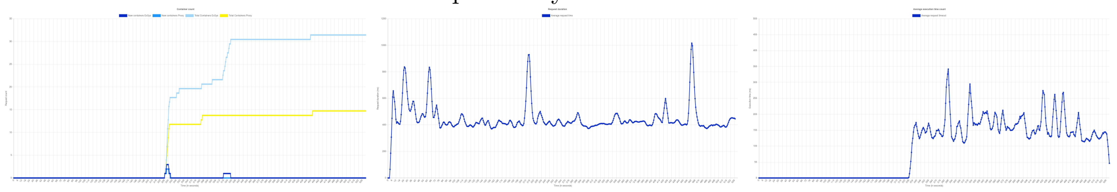
35 r/s



Traffic on system by method



Left: Responses by status code — Right: Total requests in system
Responses by status code



Left: Active VMs — Middle: Average response time — Right: Average lambda execution time

Figure 15: 35 r/s elasticity experiment results with 119 provisioned concurrency

Comparing the graph to its counterpart in the non-provisioned experiment it is noticeable that they look quite similar. The elasticity experiment fails to exactly meet the required test traffic curve.

8.5 Discussion

This section discusses the results of the both experiments and highlights special findings. All insights are applicable to both experiments if not stated otherwise.

8.5.1 Traffic generator issues

As can be seen in both 90 r/s experiment runs, the base traffic stagnates around 50 r/s. This is caused by JMeter not being able to inject the requested traffic. This can have several causes:

1. JMeter generally not able to generate that amount of request. As JMeter is an industry wide used tool, this cause is highly unlikely.
2. Personal computer resources are not enough. As the experiments have been conducted from a personal computer, the possible r/s are effected by the internet connection and the resources of the machine.
3. AWS not able to scale up the required VMs fast enough. Looking at **??** we see the problem still persists with the provisioned concurrency. As of this the platform can not be the problem

Evaluating the three causes it can be concluded that the personal computer resources are the bottleneck for the injection of 90 r/s.

8.5.2 Distributed counter issues

The effect of the distributed counter inconsistency comes to play when the platform scales up and down the VMs a lot. The goal is to have 60 r/s in the system. In the first time iterate there are 10 VMs in the system and thereby a baseline of 10 r/s is measured. ExSys instructs all proxies based on this 10 r/s distributed counter to inject an additional 5 request. If in that time period 5 more VMs are introduced by the platform also this new VMs get the instruction to inject additional 5 request. This results in a total of 90 r/s in the system. After a short time ExSys recognizes this too high request count in the system and reduces the additional requests to 3 ($ceil(100/15) = 7$). This results in the traffic spikes when the platform introduces more VMs.

The same mechanism comes into play for base traffic changes. Here it is not the platform that changes the VMs, but the client sending a different request amount. The effect is the very same, when fewer VMs than expected by ExSys are executed and thereby the counter is inaccurate for some time and ExSys injects less traffic than expected.

This effect is by design as if the counter needs to be fully accurate it would be required to wait for all transactions to be finished. This would slow down the system as a parallel execution (which is a reason to choose serverless computing) would not be possible anymore and the system would be a distributed monolith[8].

A result of the distributed counter not being accurate enough the total requests in system are off by an average of 30%. Depending on the experiment goal, this accuracy might be good enough. But as this consideration is imposed on the experimenter the usability of ExSys is affected negative and if a better accuracy is required, ExSys is no solution at all.

8.5.2.1 Design issues

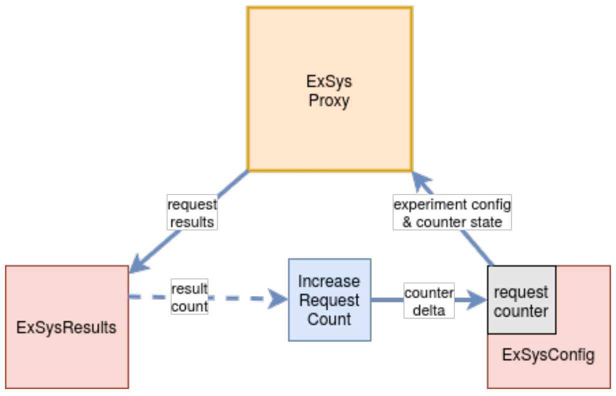


Figure 16: Distributed counter design

There is a high likelihood that a different distributed counter implementation would perform a lot better than the chosen one. Inspecting the Distributed counter design three shortcomings of the counter design are found:

1. Four steps and two databases are involved until the increased counter information is propagated back to ExSys proxy. This design is time-consuming
2. The ExSys proxy needs to load the request counter from the database on every request
3. The databases are not in-memory, resulting in further delays

One improvement would be to increase the counter directly in the proxy and not with the delay of the four steps. Another one would be a central counter service based on an in-memory database.
Both options could fuel further research, but are out of scope for this thesis. There is no guarantee that these options improve the counter without further investigation. At the moment of writing the chosen counter implementation is considered suboptimal with options to be improved.

8.5.3 Platform elasticity issues

A higher traffic experiments 35 r/s elasticity experiment results, 90 r/s elasticity experiment results,35 r/s overhead experiment results & 90 r/s overhead experiment results show that the moments the traffic drops the average request time spikes. The average lambda execution time graph shows a correlation between the response time spikes and the lambda execution. This fuels the assumption that the platform is not able to scale up the environment for additional functions fast enough, a circumstance Kuhlenkamp et al. discovered in [19] as well.
The effects of the platform not being able to scale up the environment for additional functions is worsened by ExSys as of following mechanism:

1. The platform is not able to scale up fast enough to serve the base traffic
2. ExSys registers less base traffic and instructs the proxy functions to inject even more additional traffic
3. This even higher load on the platform can not be served and the circle continues with *1.*.

For example: There should be 60 r/s in the system. There is 35 r/s base traffic which results in ExSys injecting an additional request for every request arriving. The platform is not able to satisfy the 35 r/s and only scales up to 20 r/s. ExSys then injects additional 2 request for every request even worsening the scaling problem.

8.5.4 Provisioned concurrency experiments

As expected the provisioned concurrency solved the elasticity problem as can be seen in 35 r/s elasticity experiment results with 119 provisioned concurrency. With configuring the concurrency upfront the elasticity issues are circumvented. Still there is no mayor impact for the overhead experiment as of the experiment definition the results have also been fine without the provisioned concurrency.

In addition to this insight the provisioned concurrency experiments show, that the accuracy of the distributed counter has the biggest impact for the experiment results. Even in the provisioned concurrency experiments the traffic fluctuates and base traffic drops are heavily influencing the goal of achieving the target traffic.

8.5.5 Overhead

Only 10 r/s elasticity experiment results can be taken into account for the overhead evaluation, as the other experiments suffer from the aforementioned issues. In the client visible side effects, there is no increase in errors or the average response time measurable. Neither injecting the proxy, nor starting the experiment did influence the results and thereby the ExSys overhead can be considered minimal.

8.5.6 Threads to validity of the results

The biggest thread to validity of the experiment results is the point of the traffic generation. As pointed out in the Traffic generator issues JMeter is not able to inject the requests amount of traffic for the high traffic (90 r/s) experiment. This might be caused by the traffic generator running on a personal computer with a private internet connection. The problems discussed might be an effect of an under powered machine and an insufficient internet connection. Re-doing the experiments in a data center would help to figure out the impact of this local execution.

As pointed out in the elasticity experiments Differences to the base experiment only one workload from Kuhlenkamp et al. [19] is evaluated. As only workloads are tested that ExSys is capable of handling there is already a bias regarding the *workload expresiveness.* This effect is considered in the Evaluation of ExSys

The ExSys overhead experiment results are quite good because of the experiment definition (see Results of ExSys overhead experiment). This might be a thread to validity as however ExSys performs the results show a great outcome. The experiment only gives little valuable insights

The response time of the lambda functions is measured in the ExSys proxy. This might cause the results measurements being of as the systems is measured within the system as of why a malfunction of the proxy function could distort the data of the (properly working) experiment function.

8.5.7 Evaluation of ExSys

Based on the results presented in Discussion only the 10 r/s experiments can be evaluated at all regarding the Requirements. All other experiments are so heavily influenced by the Platform elasticity issues that they are of no use for the evaluation of ExSys and are discussed later on.

- *Client-visible Side-effects (CvSEs)*: The client visible side effects are minimal but present, which can be evaluated in the second (middle) and the fourth (bottom middle) graph of the experiment results: Neither the error rate (second) or the average response time increase at the time of injection or the time of the experiment start when the proxy gets active. Throughout the experiment a minimal increase in average response time can be seen, but this could also be caused by network delays.
- *Workload accuracy (WA)*: The Distributed counter issues make a 100% WA impossible. Only a small percentage overhead of additional requests is injected. As can be seen in 10 r/s elasticity experiment results & 10 r/s overhead experiment results the Total request in the system do differ by an average of 30%. The workload can be considered not accurate enough as the effects of the distributed counter are influencing the outcome too much and thereby imposing the decision if ExSys is a valid tool for every experiment again on the experimenter. This inherent decision process reduces the usability of the system and might cause errors in the experiment if the decision of the experimenter is wrong.
- *Time to experiment start (TtES)*: The time to proxy deployment can be client measured and is 1.5 seconds on average. This is a fast deployment as only the alias needs to be configured in the API gateway and no new functions needs to be deployed. The start command takes on average 1 second (measured from the client). This is also a quite fast start, due to the alias implementation.

- *Mean time to repair (MttR)*: The stop call takes about 1 seconds (measured from the client). This can be considered a low time to repair
- *Infrastructure cost overhead (ICO)*: The cost overhead in measured in the experiments is as expected. There is close to 100% cost overhead for the base traffic, as additional to the actually called function the proxy also needs to be called and executed. Depending on the configuration of the original and experiment function the cost overhead is less than 100% as the proxy function needs a lower memory config than the actual function. As in the experiments the proxy and the actual function have the same config the cost overhead is close to 100%. For additionally called functions (especially in the Experiment 1: Elasticity experiment) every additional called function is another 100% cost overhead. So if 5 additional functions are called, ExSys has a cost overhead of round about 600%. The cost overhead is that high as every function executed in a serverless environment produces this monetary overhead. If the experimenter chooses to opt for provisioned concurrency, the cost overhead increases again. In the case of AWS lambda for an experiment lasting 15 minutes this would again be 800% additional cost overhead, totaling in 1400% overhead.
- *Workload Expressiveness (WE)* The workload expressiveness is high. All required workload cases can be configured. Even more complex examples as in 10 r/s overhead experiment results are possible as of the flexibility of the ExSys stage configuration. One downside of ExSys is the lack of dropping requests. If there are already more request incoming than the experiment desires, the results are off.
- *Usability (U)* With ExSys offering an easy-to-use HTTP API the Usability is high as only two HTTP request need to be sent to start the experiment. After the end of the experiment ExSys removes the proxy automatically and the results can be retrieved via the HTTP API. Even a GUI tool would be imaginable as the HTTP request are clearly structured and documented. The results of ExSys can be visualized with the given diagram tooling in the online appendix[48].

Even though the evaluation criteria are fulfilled for the 10 r/s experiments, the Platform elasticity issues render ExSys not a viable solution to enable live workload experiments as the experiments did show that ExSys is only working in low traffic environments. Additional ExSys has the inherent problem off not being able to produce less traffic than already incoming into the application. This is a shortcoming within online experiments overall and narrows down the potential situations where ExSys can be beneficial.

ExSys itself brings the Distributed counter issues which are impacting the results, but are not as significant as can be seen in the 10 r/s experiments. The platform issues on the other hand are not solvable by ExSys and do prevent an operation of ExSys in a production environment with shadow traffic. As seen in Results of ExSys overhead experiment canary traffic works with ExSys as it is not influenced by the scalability and distributed counter issues. Canary traffic is working, but there is still the 100% cost overhead with ExSys. If only the canary traffic is required AWS Codedeploy, with slight additional utilization, would be a viable and more cost-efficient solution. As long as the platforms do not solve their scalability issues, ExSys will only be able to work with provisioned concurrency.

[48]**https://github.com/simonfrey/exsys**

9 Conclusion

Thinking back to the research question *How can developers use specialized workloads in online experiments on serverless applications on demand?* the thesis offers a two folded answer:

Developers can use specialized workloads with a shadow deployment tooling. ExSys supports the developer by getting their systems tested on demand with every workload they imagine.

On the other hand the thesis found the elasticity issues of the platform (in the experiment case AWS) rendering an on-demand workload control useless, as neither the base system nor the ExSys proxy gets enough resources by the platform in a reasonable time. This is a problem with the serverless platforms overall as described in [19]. Also, offline experiments are experiencing the elasticity problem. Equally, to offline experiments ExSys relies on the platform scaling up the VMs fast enough. This is at the moment of writing not the case.

Using the provisioned concurrency feature of AWS this elasticity problem is circumvented, but a high cost overhead is introduced for each experiment with this feature and a core principle of serverless architecture is removed. The experimenter needs to decide what amount of functions should be provisioned and can not rely on the platform for this question rendering the usability of ExSys low in this case.

A reevaluation of the experiments should be considered in the future to check if the platforms solved their elasticity issues and if online experiments with shadow traffic are a viable option for serverless functions. Until then the provisioned concurrency can be used for shadow traffic and canary experiments can be run with ExSys or platform native tooling.

The distributed counter issues might be solvable with a better design (see Distributed counter issues). When that problem is solved the usability of ExSys would further increase and the system fitting for a broader array of experiment cases.

ExSys might be used in any form of serverless application as the experiment overhead is as low as possible (with still being significant). As long as no experiment is running, ExSys imposes no costs or side effects on the serverless application. It has to be evaluated by the experimenter if the cost overhead is bearable.

The research question can be considered partially fulfilled. On the one hand it was proven that *specialized workloads in online experiments on serverless applications* are possible. On the other hand the *on demand* requirement is not fulfilled as the different problems do not enable arbitrary experiments with a standard configuration, but require the experimenter to investigate if and how the intended experiment can be run.

With specialized workloads being possible, also non-functional regression testing for serverless applications can be considered available. As pointed out in Online experiments we can conduct regression testing on serverless applications by executing experiments for the different versions and comparing the results of the experiment.

10 Bibliography

References

[1] Paulo Sérgio Almeida and Carlos Baquero. “Scalable Eventually Consistent Counters over Unreliable Networks”. In: (2013). arXiv: `1307.3207 [cs.DC]`.

[2] Passwater Andrea. *Serverless community survey: Huge growth in serverless usage.* Accessed 2020-5-18. 2018. URL: `https://serverless.com/blog/2018-serverless-community-survey-huge-growth-usage/`.

[3] Ioana Baldini et al. “Serverless Computing: Current Trends and Open Problems”. In: *Research Advances in Cloud Computing.* Ed. by Sanjay Chaudhary, Gaurav Somani, and Rajkumar Buyya. Singapore: Springer Singapore, 2017, pp. 1–20. ISBN: 978-981-10-5026-8. DOI: `10.1007/978-981-10-5026-8_1`. URL: `https://doi.org/10.1007/978-981-10-5026-8_1`.

[4] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect’s Perspective.* SEI Series in Software Engineering. New York: Addison-Wesley, 2015. ISBN: 978-0-13-404984-7. URL: `http://my.safaribooksonline.com/9780134049847`.

[5] Giuseppe DeCandia et al. “Dynamo: Amazon’s Highly Available Key-Value Store”. In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980. DOI: `10.1145/1323293.1294281`. URL: `https://doi.org/10.1145/1323293.1294281`.

[6] Adam Eivy. “Be Wary of the Economics of ”Serverless” Cloud Computing”. In: *IEEE Cloud Computing* 4.2 (Mar. 2017), pp. 6–12. ISSN: 2372-2568. DOI: `10.1109/MCC.2017.32`.

[7] Dominik Ernst, Alexander Becker, and Stefan Tai. “Rapid Canary Assessment Through Proxying and Two-Stage Load Balancing”. In: *2019 IEEE International Conference on Software Architecture Companion (ICSA-C).* Mar. 2019, pp. 116–122. DOI: `10.1109/ICSA-C.2019.00028`.

[8] Justin Etheredge. *You’re Not Actually Building Microservices.* Accessed 2020-8-231. 2018. URL: `https://www.simplethread.com/youre-not-actually-building-microservices/`.

[9] Dror Feitelson, Eitan Frachtenberg, and Kent Beck. “Development and Deployment at Facebook”. In: *IEEE Internet Computing* 17.4 (July 2013), pp. 8–17. ISSN: 1941-0131. DOI: `10.1109/MIC.2013.25`.

[10] Martin Fowler and James Lewis. *Microservices a definition of this new architectural term.* Accessed 2020-5-22. 2014. URL: `https://martinfowler.com/articles/microservices.html`.

[11] Martin Glinz. “Rethinking the notion of non-functional requirements”. In: *Proc. Third World Congress for Software Quality.* Vol. 2. 2005, pp. 55–64.

[12] Joseph M. Hellerstein et al. “Serverless Computing: One Step Forward, Two Steps Back”. In: *CoRR* abs/1812.03651 (Jan. 2018). URL: `http://arxiv.org/abs/1812.03651`.

[13] Jez Humble and David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation.* 1st. Addison-Wesley Signature Series (Fowler). Addison-Wesley Professional, 2010. ISBN: 9780321670229. URL: `https://books.google.de/books?id=6ADDuzere-YC`.

[14] Richard Karp. “On-Line Algorithms Versus Off-Line Algorithms: How Much is it Worth to Know the Future?” In: July 1992. URL: `http://www.icsi.berkeley.edu/pubs/techreports/TR-92-044.pdf`.

[15] Youngbin Kim and Jimmy Lin. “Serverless Data Analytics with Flint”. In: *2018 IEEE 11th International Conference on Cloud Computing (CLOUD).* 2018, pp. 451–455. DOI: `10.1109/CLOUD.2018.00063`.

[16] Ron Kohavi, Randal M. Henne, and Dan Sommerfield. “Practical Guide to Controlled Experiments on the Web: Listen to Your Customers Not to the Hippo”. In: *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD ’07. San Jose, California, USA: Association for Computing Machinery, 2007, pp. 959–967. ISBN: 9781595936097. DOI: `10.1145/1281192.1281295`. URL: `https://doi.org/10.1145/1281192.1281295`.

[17] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. “Benchmarking Scalability and Elasticity of Distributed Database Systems”. In: *Proceedings of the VLDB Endowment* 7.12 (2014), pp. 1219–1230.

[18] Jörn Kuhlenkamp et al. "An Evaluation of FaaS Platforms as a Foundation for Serverless Big Data Processing". In: *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. UCC'19. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 1–9. ISBN: 9781450368940. DOI: `10.1145/3344341.3368796`. URL: `https://doi.org/10.1145/3344341.3368796`.

[19] Jörn Kuhlenkamp et al. "Benchmarking Elasticity of FaaS Platforms as a Foundation for Objective-driven Design of Serverless Applications". In: *35th ACM/SIGAPP Symposiumon Applied Computing (SAC '20)*. Apr. 2020. DOI: `10.1145/3341105.3373948`.

[20] Dimitar Kumanov et al. "Serverless computing provides on-demand high performance computing for biomedical research". In: (2018). arXiv: `1807.11659 [q-bio.QM]`.

[21] Hareton Leung and Lee White. "Insights into regression testing (software testing)". In: *Proceedings. Conference on Software Maintenance - 1989*. 1989, pp. 60–69. DOI: `10.1109/ICSM.1989.65194`.

[22] Zhu Liming, Bass Len, and Champlin-Scharff Georg. "DevOps and Its Practices". In: *IEEE Software* 33.3 (May 2016), pp. 32–34. ISSN: 1937-4194. DOI: `10.1109/MS.2016.81`.

[23] Christian Nieke and Wolf-Tilo Balke. "Monitoring Performance in Large Scale Computing Clouds with Passive Benchmarking". In: *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*. 2017, pp. 188–195. DOI: `10.1109/CLOUD.2017.32`.

[24] Mike Roberts. *Serverless Architectures*. `https://martinfowler.com/articles/serverless.html`. [Online; accessed 27.02.2019]. May 2017.

[25] Michael Ruth and Shengru Tu. "Towards Automatic Regression Test Selection for Web Services". In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. Vol. 2. July 2007, pp. 729–736. DOI: `10.1109/COMPSAC.2007.219`.

[26] Simon Salomé. "Brewer's cap theorem". In: *CS341 Distributed Information Systems, University of Basel (HS2012)* (2000).

[27] Tony Savor et al. "Continuous Deployment at Facebook and OANDA". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. May 2016, pp. 21–30.

[28] Gerald Schermann, Jürgen Cito, and Philipp Leitner. "Continuous Experimentation: Challenges, Implementation Techniques, and Current Research". In: *IEEE Software* 35.2 (Mar. 2018), pp. 26–31. ISSN: 1937-4194. DOI: `10.1109/MS.2018.111094748`.

[29] Gerald Schermann et al. "Bifrost: Supporting Continuous Deployment with Automated Enactment of Multi-Phase Live Testing Strategies". In: *Proceedings of the 17th International Middleware Conference*. Middleware '16. Trento, Italy: Association for Computing Machinery, 2016. ISBN: 9781450343008. DOI: `10.1145/2988336.2988348`. URL: `https://doi.org/10.1145/2988336.2988348`.

[30] Gerald Schermann et al. "We're doing it live: A multi-method empirical study on continuous experimentation". In: *Information and Software Technology* 99 (2018), pp. 41–57. ISSN: 0950-5849. DOI: `https://doi.org/10.1016/j.infsof.2018.02.010`. URL: `http://www.sciencedirect.com/science/article/pii/S0950584917302136`.

[31] Marc Shapiro et al. "Conflict-Free Replicated Data Types". In: (2011). Ed. by Xavier Défago, Franck Petit, and Vincent Villain, pp. 386–400.

[32] Jan Stenberg. *Characteristics of Serverless Architecture*. Accessed 2020-11-07. 2019. URL: `https://www.infoq.com/news/2019/08/traits-serverless-architecture/` (visited on 08/13/2019).

[33] Andreas Weber et al. "Towards a Resource Elasticity Benchmark for Cloud Environments". In: *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability*. HotTopiCS '14. Dublin, Ireland: Association for Computing Machinery, 2014. ISBN: 9781450330596. DOI: `10.1145/2649563.2649571`. URL: `https://doi.org/10.1145/2649563.2649571`.

[34] Sebastian Werner et al. "Serverless Big Data Processing Using Matrix Multiplication as Example". en. In: *Proceedings of the IEEE International Conference on Big Data*. Big Data'18. IEEE, Dec. 2018, pp. 358–365. ISBN: 978-1-5386-5035-6. DOI: `10.1109/BigData.2018.8622362`.

[35] W. Eric Wong et al. "A study of effective regression testing in practice". In: *Proceedings The Eighth International Symposium on Software Reliability Engineering*. Nov. 1997, pp. 264–274. DOI: `10.1109/ISSRE.1997.630875`.