Performance testing

Tools and Techniques for Software Testing - Barbara Russo

SwSE - Software and Systems Engineering group

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Modern (online) systems may underperform as they are overloaded

Performance testing

- Performance testing is the **process** of determining the **speed, responsiveness and stability** of a computer, network, **software program** or device **under a workload**
- Performance testing can involve quantitative tests done in a lab, or occur in the production environment in limited scenarios

wikipedia

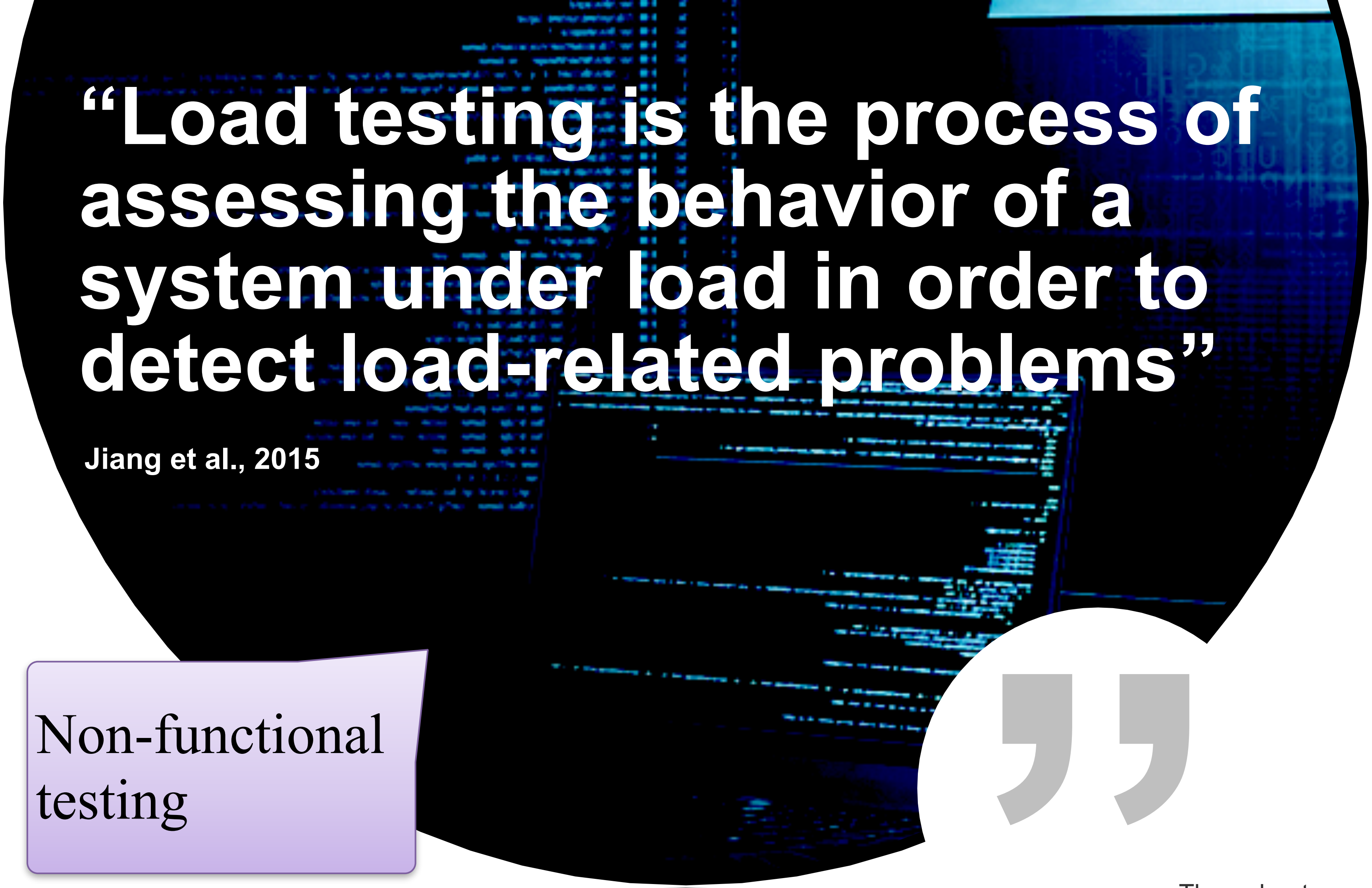
Open question: how to test in in-production environments?

Example of performance testing: Load Testing

Tools and Techniques for Software Testing - Barbara Russo

SwSE - Software and Systems Engineering group

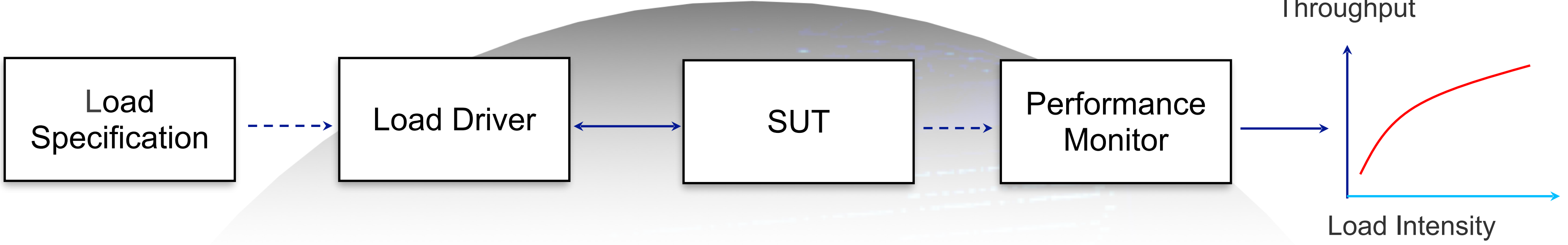
unibz Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan



“Load testing is the process of assessing the behavior of a system under load in order to detect load-related problems”

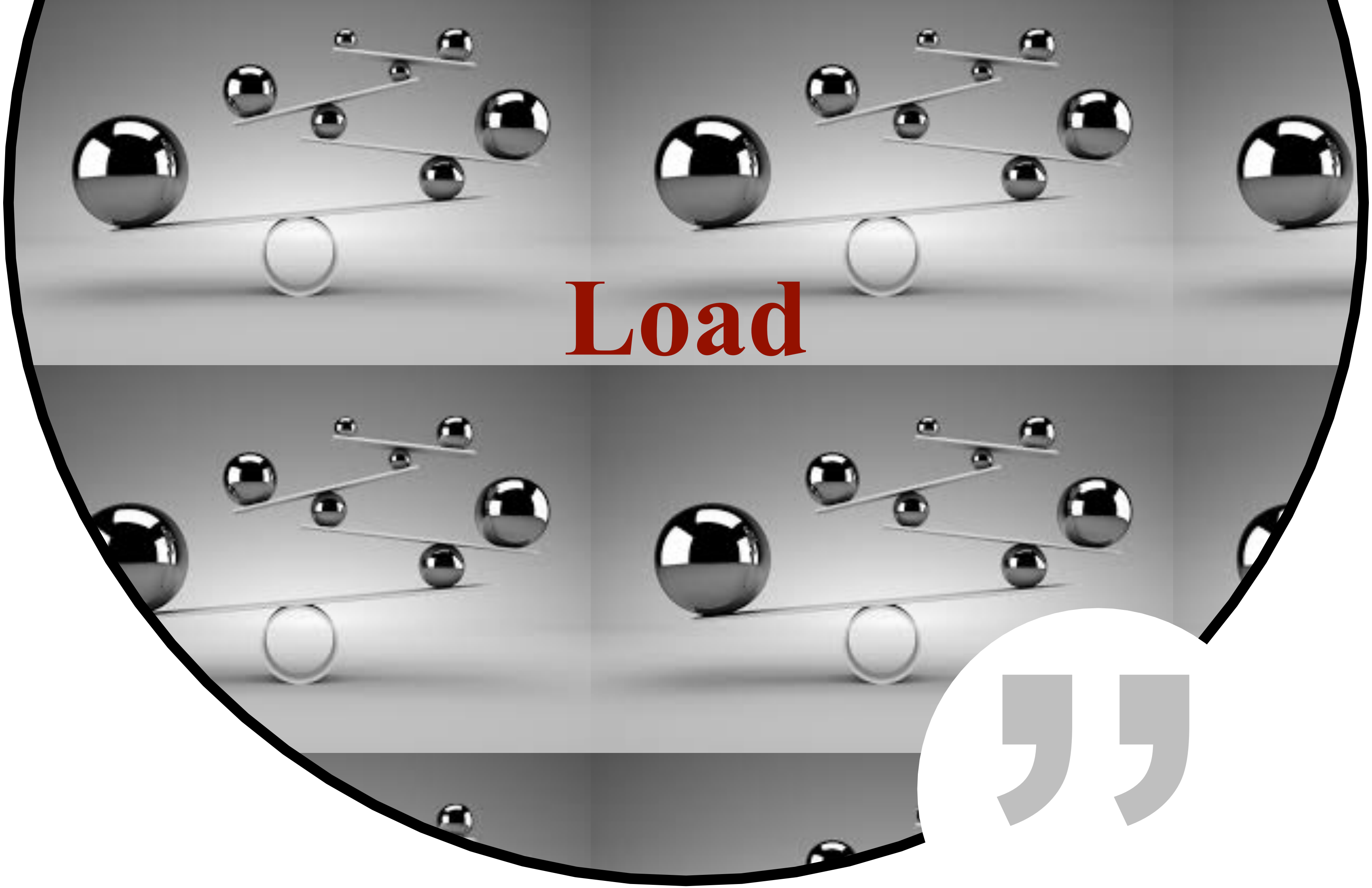
Jiang et al., 2015

Non-functional testing

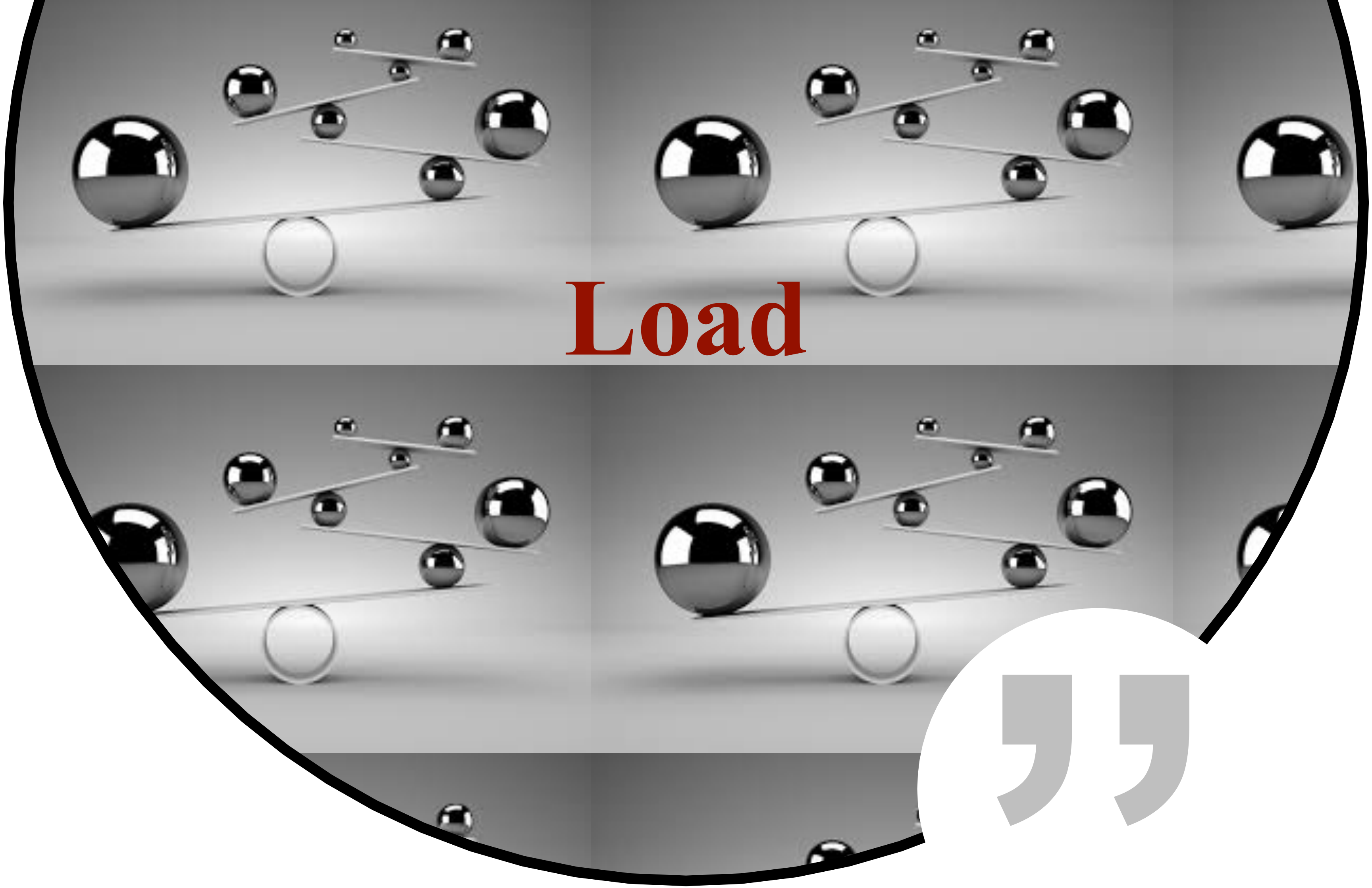


What is load?

Load



Amount of computational work being performed by a software system

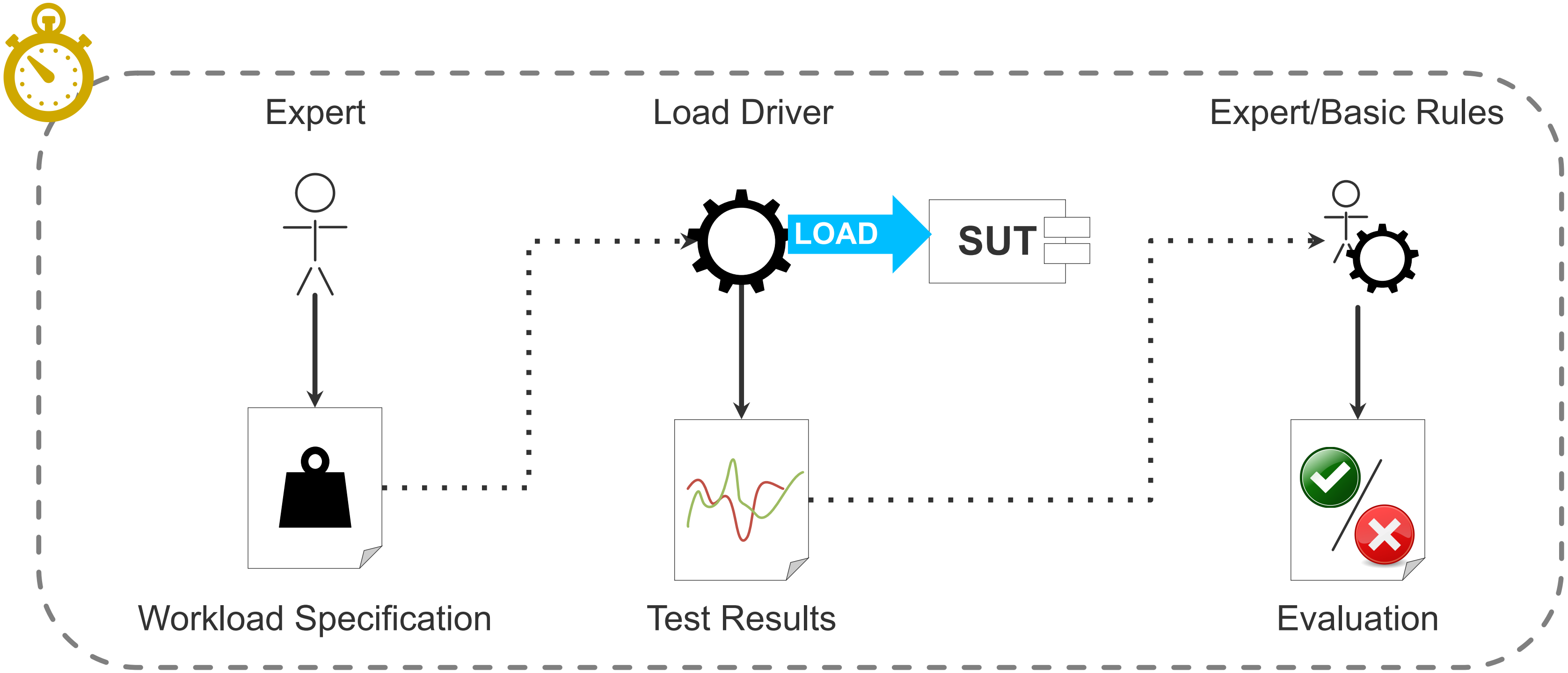


Load

Amount of concurrent users

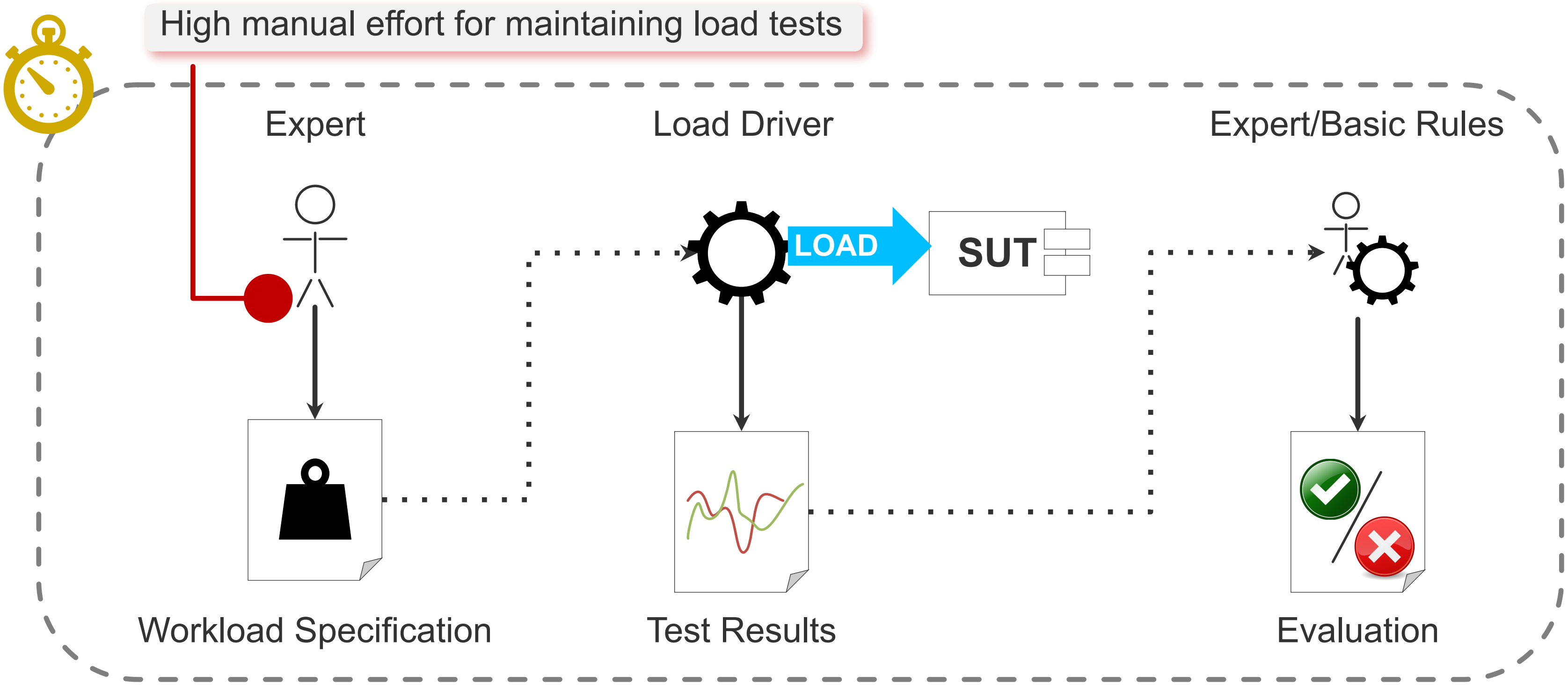
The Classic Load Testing Approach

… and Classic Problems



The Classic Load Testing Approach

… and Classic Problems

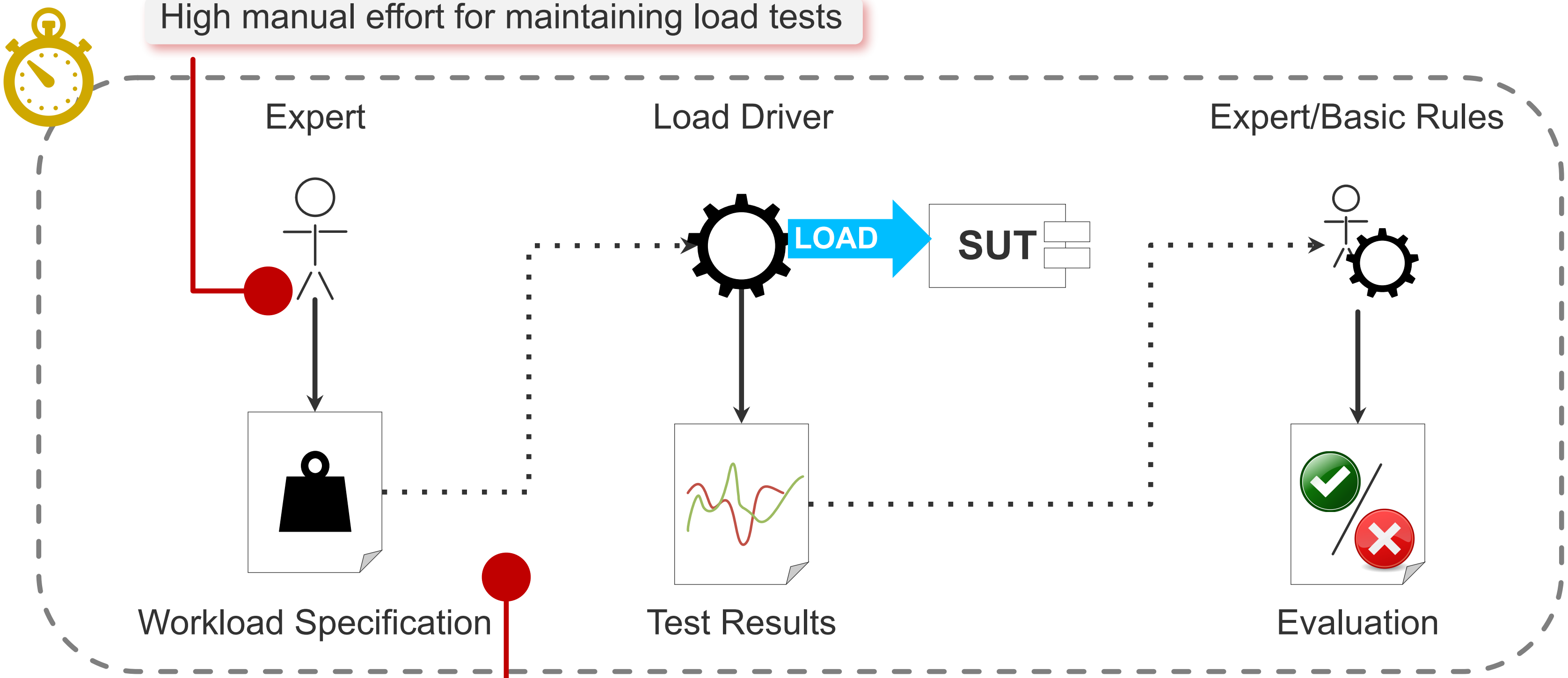


High manual effort for maintaining load tests

The Classic Load Testing Approach

… and Classic Problems

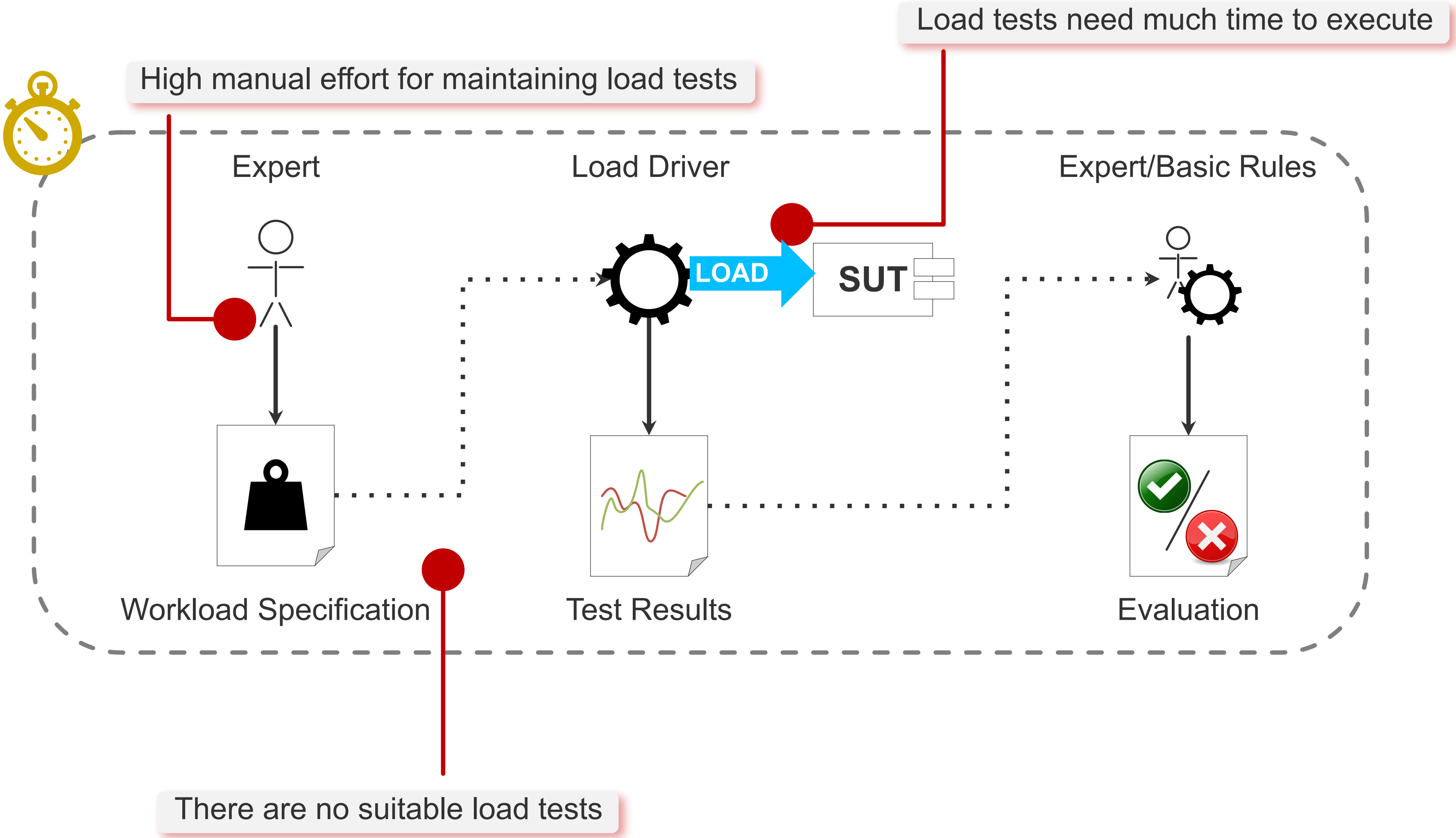
High manual effort for maintaining load tests



There are no suitable load tests

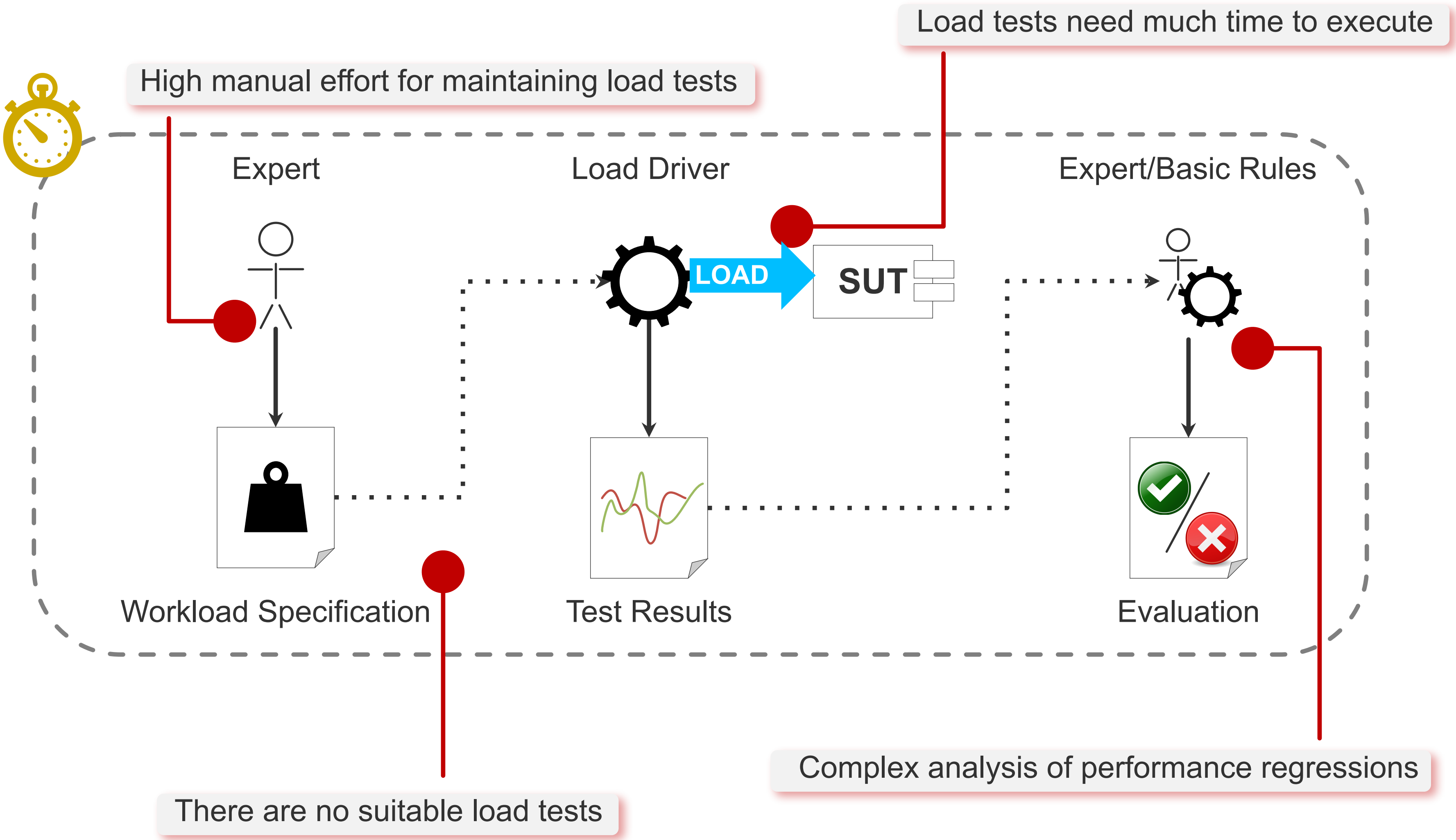
The Classic Load Testing Approach

… and Classic Problems


Load tests need much time to execute
High manual effort for maintaining load tests
Expert
Load Driver
Expert/Basic Rules
LOAD
SUT
Workload Specification
Test Results
Evaluation
There are no suitable load tests

The Classic Load Testing Approach

… and Classic Problems



High manual effort for maintaining load tests

Load tests need much time to execute

Complex analysis of performance regressions

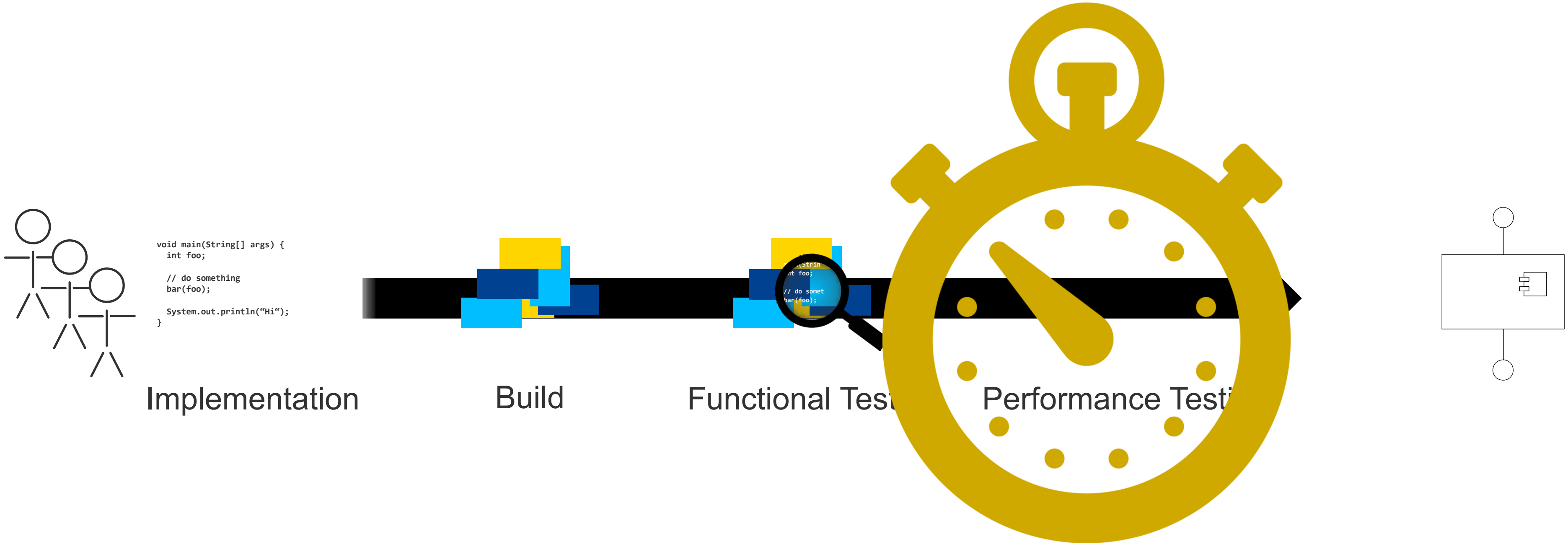
There are no suitable load tests

Load Testing in Continuous Delivery Pipelines

… How Problems Get Worse

Load tests need much time to execute

High manual effort for maintaining load tests



```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println(“Hi”);
}
```

Complex analysis of performance regressions

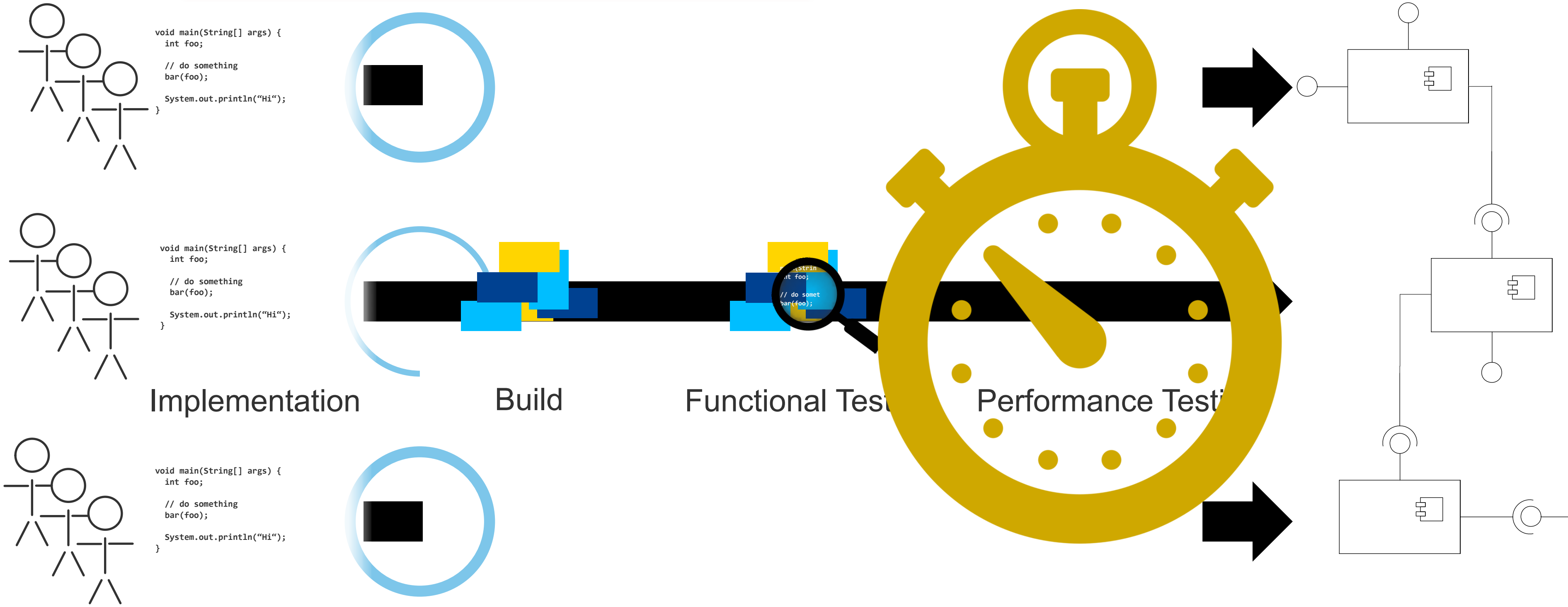
There are no suitable load tests

Load Testing in Continuous Delivery Pipelines

… How Problems Get Worse

Load tests need much time to execute

High manual effort for maintaining load tests



```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println(“Hi“);
}
```

```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println(“Hi“);
}
```

```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println(“Hi“);
}
```

Complex analysis of performance regressions

There are no suitable load tests

Load Testing in Continuous Delivery Pipelines

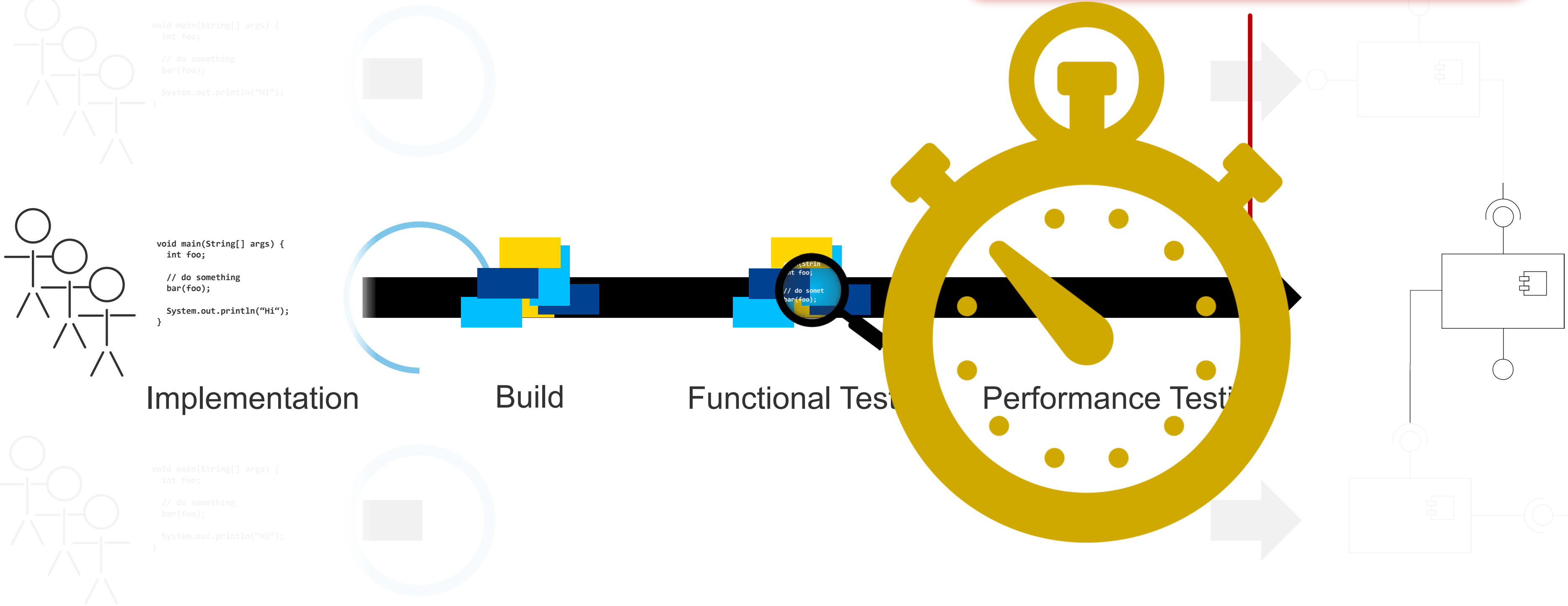
… How Problems Get Worse

High manual effort for maintaining load tests

Load tests need much time to execute

vs.

Fast & frequent releases



```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println("Hi");
}
```

Complex analysis of performance regressions

There are no suitable load tests

Load Testing in Continuous Delivery Pipelines

… How Problems Get Worse

High manual effort for maintaining load tests

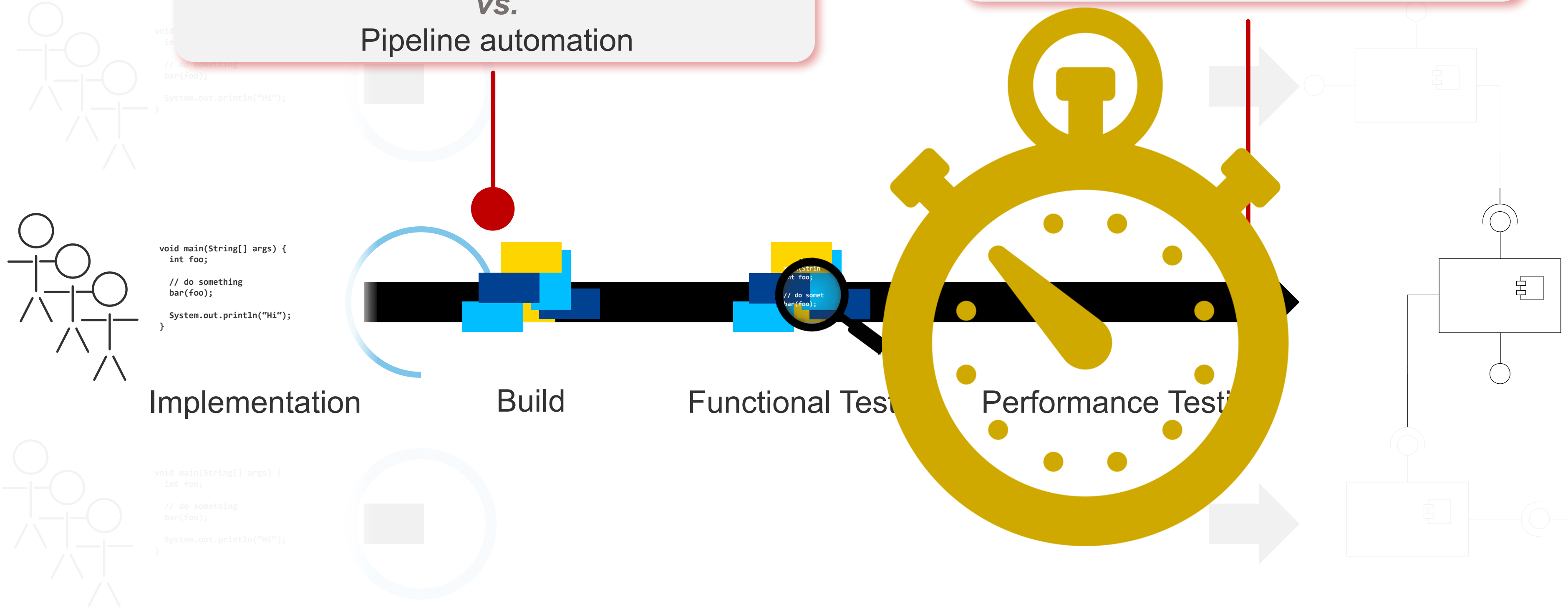
vs.

Pipeline automation

Load tests need much time to execute

vs.

Fast & frequent releases



```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println(“Hi”);
}
```

Implementation

Build

Functional Test

Performance Testi

Complex analysis of performance regressions

There are no suitable load tests

Load Testing in Continuous Delivery Pipelines

… How Problems Get Worse

High manual effort for maintaining load tests

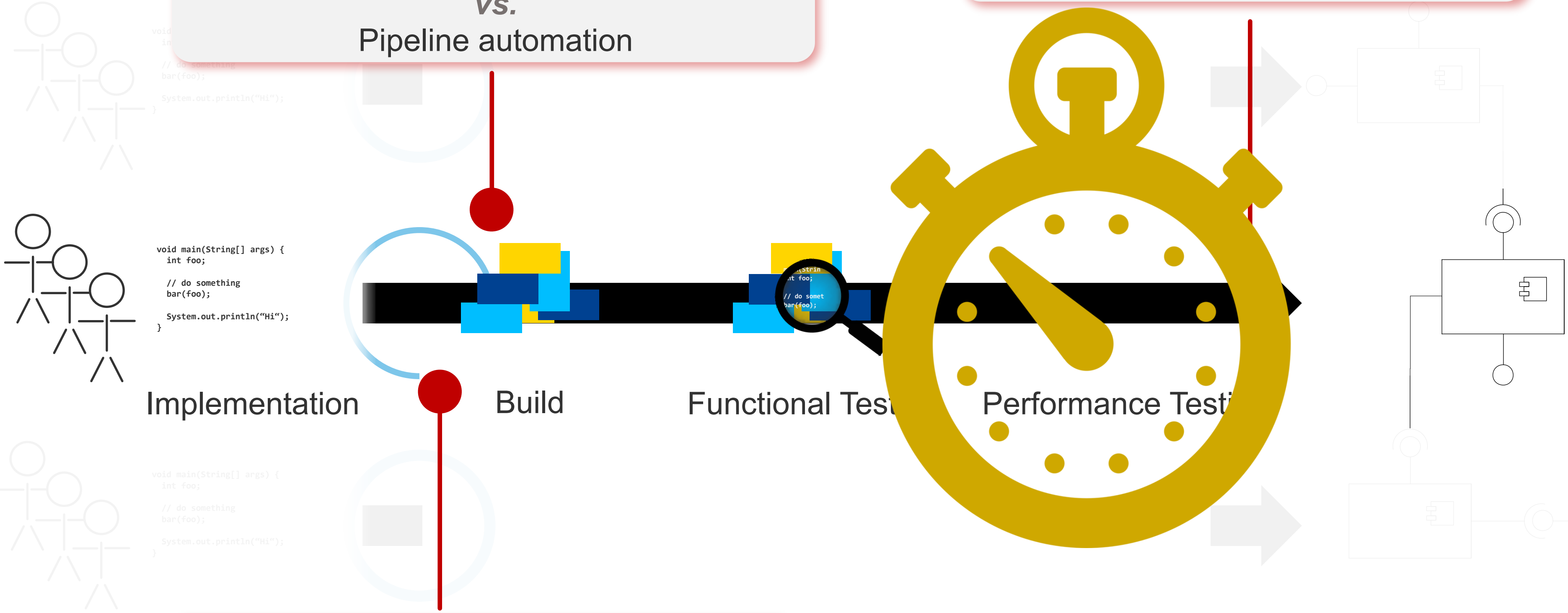
vs.

Pipeline automation

Load tests need much time to execute

vs.

Fast & frequent releases



```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println(“Hi”);
}
```

Service-focus requires multiple tests

vs.

There are no suitable load tests

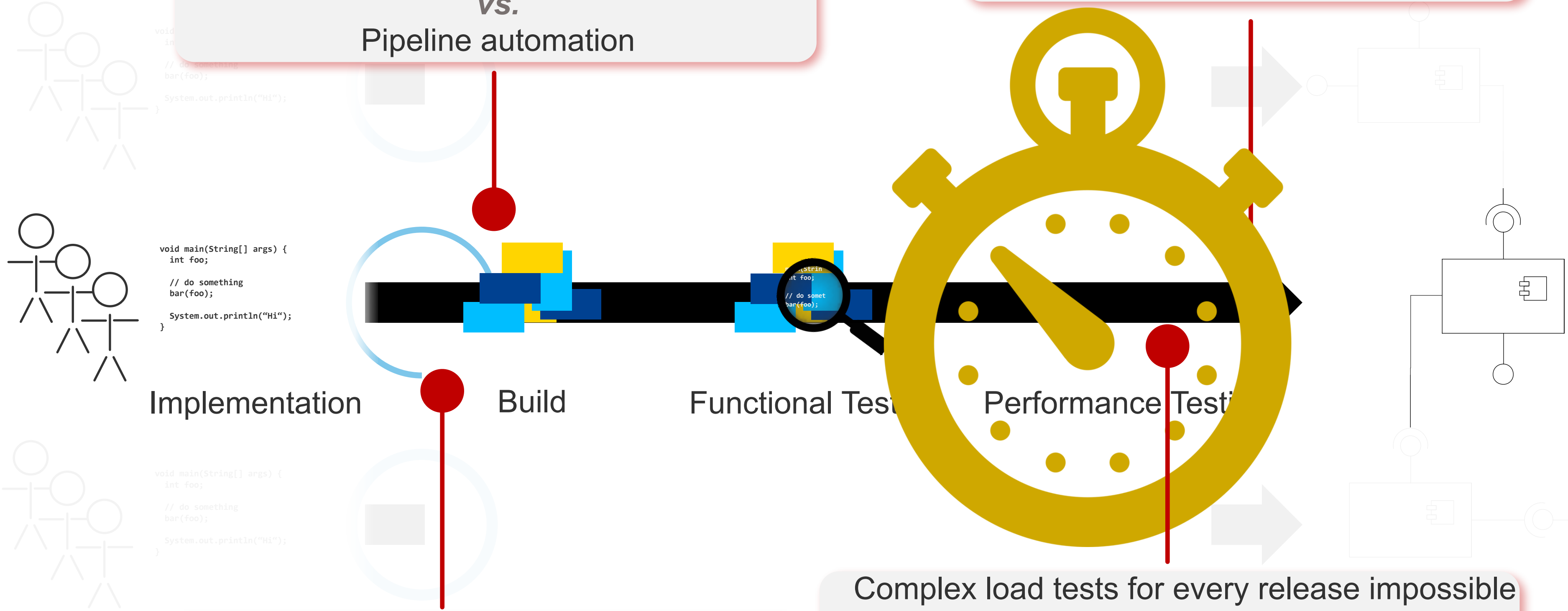
Complex analysis of performance regressions

Load Testing in Continuous Delivery Pipelines

… How Problems Get Worse

High manual effort for maintaining load tests
vs.
Pipeline automation

Load tests need much time to execute
vs.
Fast & frequent releases



```
void main(String[] args) {
  int foo;

  // do something
  bar(foo);

  System.out.println(“Hi”);
}
```

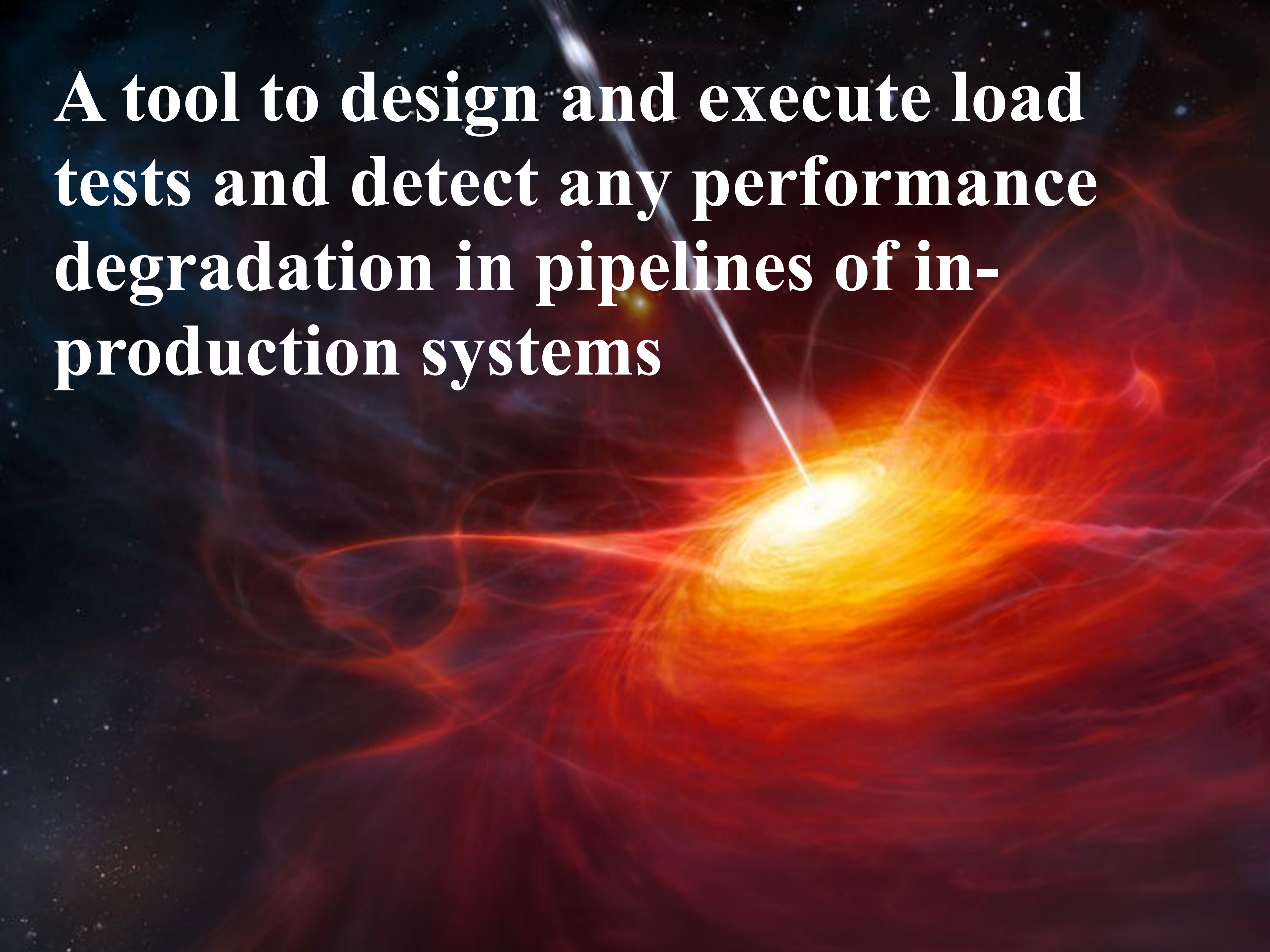
Service-focus requires multiple tests
vs.
There are no suitable load tests

Complex load tests for every release impossible
vs.
Complex analysis of performance regressions

Workload specification

- The workload specification Model consists of:
- An *Application Model*, specifying **allowed sequences of service invocations** and SUT-specific details for generating valid requests
- A set of *Behavior Models*, each providing a probabilistic representation of user sessions in terms of **invoked services and think times** between subsequent invocations as **Markov Chains**
- A *Behavior Mix*, specified as **probabilities (frequencies) for the individual Behavior Models to occur** during workload generation
- A *Workload Intensity* that includes a function which specifies the (possibly varying) **number of concurrent users** during the workload generation execution

A tool to design and execute load tests and detect any performance degradation in pipelines of in-production systems



Monitoring data

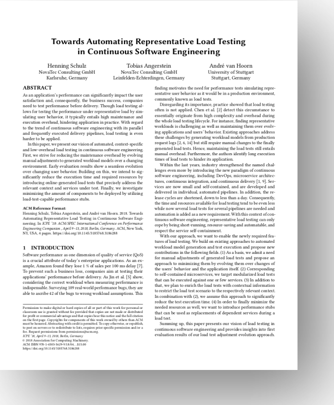
- **Loads frequencies**, **Request logs, traces, and response times** of the service interfaces
- The data is enriched by various **contextual information**, e.g., marketing campaigns, public holidays, or sports events

ContinuITy

Load Test Generation & Execution

ContinuITy

Load Test Generation & Execution



Towards Automating Representative Load Testing in Continuous Software Engineering

Henning Schulz
NovaTec Consulting GmbH
Karlsruhe, Germany

Tobias Angerstein
NovaTec Consulting GmbH
Leinfelden-Echterdingen, Germany

André van Hoorn
University of Stuttgart
Stuttgart, Germany

ABSTRACT

As an application's performance can significantly impact the user satisfaction and, consequently, the business success, companies need to test performance before delivery. Though load testing allows for testing the performance under representative load by simulating user behavior, it typically entails high maintenance and execution overhead, hindering application in practice. With regard to the trend of continuous software engineering with its parallel and frequently executed delivery pipelines, load testing is even harder to be applied.

In this paper, we present our vision of automated, context-specific and low-overhead load testing in continuous software engineering. First, we strive for reducing the maintenance overhead by evolving manual adjustments to generated workload models over a changing environment. Early evaluation results show a seamless evolution over changing user behavior. Building on this, we intend to significantly reduce the execution time and required resources by introducing online-generated load tests that precisely address the relevant context and services under test. Finally, we investigate minimizing the amount of components to be deployed by utilizing load-test-capable performance stubs.

ACM Reference Format:
Henning Schulz, Tobias Angerstein, and André van Hoorn. 2018. Towards Automating Representative Load Testing in Continuous Software Engineering. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186288

1 INTRODUCTION

Software performance as one dimension of quality of service (QoS) is a crucial attribute of today's enterprise applications. As an example, Amazon found they lose 1 % of sales per 100 ms delay [7]. To prevent such a business loss, companies aim at testing their applications' performance before delivery. As Jin et al. [5] show, considering the correct workload when measuring performance is indispensable. Surveying 109 real-world performance bugs, they are able to ascribe 42 of the bugs to wrong workload assumptions. This finding motivates the need for performance tests simulating representative user behavior as it would be in a production environment, commonly known as load tests.

Disregarding its importance, practice showed that load testing often is not applied. Chen et al. [2] detect this circumstance to essentially originate from high complexity and overhead during the whole load testing lifecycle. For instance, finding representative workloads is challenging as well as maintaining them over evolving applications and users' behavior. Existing approaches address these challenges by generating workload models from production request logs [2, 6, 14] but still require manual changes to the finally generated load tests. Hence, maintaining the load tests still entails manual overhead. Furthermore, the authors identify long execution times of load tests to hinder its application.

Within the last years, industry strengthened the named challenges even more by introducing the new paradigm of continuous software engineering, including DevOps, microservice architectures, continuous integration, and continuous delivery [1, 9]. Services are now small and self-contained, and are developed and delivered in individual, automated pipelines. In addition, the release cycles are shortened, down to less than a day. Consequently, the time and resources available for load testing tend to be even less while now several load tests for several pipelines are needed and automation is added as a new requirement. With this context of continuous software engineering, representative load testing can only cope by being short-running, resource-saving and automatable, and respect the service self-containment.

With our approach, we want to enable the newly required features of load testing. We build on existing approaches to automated workload model generation and test execution and propose new contributions in the following fields. (1) As a basis, we admit a need for manual adjustments of generated load tests and propose an approach to minimizing them by evolving them over changes of the users' behavior and the application itself. (2) Corresponding to self-contained microservices, we target modularized load tests that can be executed against one or few services. (3) In addition to that, we plan to enrich the load tests with contextual information to restrict the load test scenario to the respectively relevant context. In combination with (2), we assume this approach to significantly reduce the test execution time. (4) In order to finally minimize the needed resources as well, we want to introduce performance stubs that can be used as replacements of dependent services during a load test.

Summing up, this paper presents our vision of load testing in continuous software engineering and provides insights into first evaluation results of our load test adjustment evolution approach.

ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186288

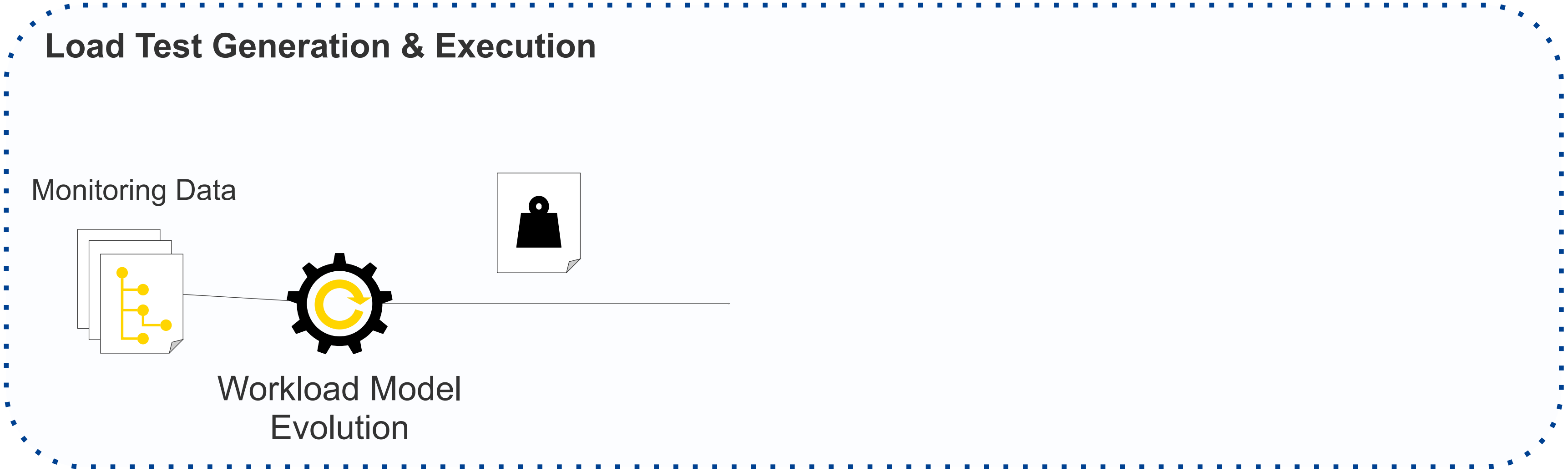
Henning Schulz, Tobias Angerstein, and André van Hoorn:
Towards automating representative load testing in continuous software engineering.

In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE 2018) Companion (7th International Workshop on Load Testing and Benchmarking of Software Systems, LTB 2018), pages 123–126. ACM, 2018

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

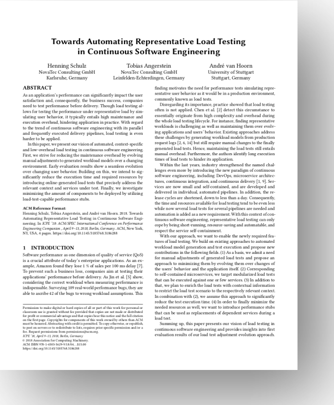
ContinuITy

Load Test Generation & Execution



Monitoring Data

Workload Model Evolution



Towards Automating Representative Load Testing in Continuous Software Engineering

Henning Schulz
NovaTec Consulting GmbH
Karlsruhe, Germany

Tobias Angerstein
NovaTec Consulting GmbH
Leinfelden-Echterdingen, Germany

André van Hoorn
University of Stuttgart
Stuttgart, Germany

ABSTRACT

As an application's performance can significantly impact the user satisfaction and, consequently, the business success, companies need to test performance before delivery. Though load testing allows for testing the performance under representative load by simulating user behavior, it typically entails high maintenance and execution overhead, hindering application in practice. With regard to the trend of continuous software engineering with its parallel and frequently executed delivery pipelines, load testing is even harder to be applied.

In this paper, we present our vision of automated, context-specific and low-overhead load testing in continuous software engineering. First, we strive for reducing the maintenance overhead by evolving manual adjustments to generated workload models over a changing environment. Early evaluation results show a seamless evolution over changing user behavior. Building on this, we intend to significantly reduce the execution time and required resources by introducing online-generated load tests that precisely address the relevant context and services under test. Finally, we investigate minimizing the amount of components to be deployed by utilizing load-test-capable performance stubs.

ACM Reference Format:
Henning Schulz, Tobias Angerstein, and André van Hoorn. 2018. Towards Automating Representative Load Testing in Continuous Software Engineering. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186288

1 INTRODUCTION

Software performance as one dimension of quality of service (QoS) is a crucial attribute of today's enterprise applications. As an example, Amazon found they lose 1 % of sales per 100 ms delay [7]. To prevent such a business loss, companies aim at testing their applications' performance before delivery. As Jin et al. [5] show, considering the correct workload when measuring performance is indispensable. Surveying 109 real-world perfomance bugs, they are able to ascribe 42 of the bugs to wrong workload assumptions. This finding motivates the need for performance tests simulating representative user behavior as it would be in a production environment, commonly known as load tests.

Disregarding its importance, practice showed that load testing often is not applied. Chen et al. [2] detect this circumstance to essentially originate from high complexity and overhead during the whole load testing lifecycle. For instance, finding representative workloads is challenging as well as maintaining them over evolving applications and users' behavior. Existing approaches address these challenges by generating workload models from production request logs [2, 6, 14] but still require manual changes to the finally generated load tests. Hence, maintaining the load tests still entails manual overhead. Furthermore, the authors identify long execution times of load tests to hinder its application.

Within the last years, industry strengthened the named challenges even more by introducing the new paradigm of continuous software engineering, including DevOps, microservice architectures, continuous integration, and continuous delivery [1, 9]. Services are now small and self-contained, and are developed and delivered in individual, automated pipelines. In addition, the release cycles are shortened, down to less than a day. Consequently, the time and resources available for load testing tend to be even less while now several load tests for several pipelines are needed and automation is added as a new requirement. With this context of continuous software engineering, representative load testing can only cope by being short-running, resource-saving and automatable, and respect the service self-containment.

With our approach, we want to enable the newly required features of load testing. We build on existing approaches to automated workload model generation and test execution and propose new contributions in the following fields. (1) As a basis, we admit a need for manual adjustments of generated load tests and propose an approach to minimizing them by evolving them over changes of the users' behavior and the application itself. (2) Corresponding to self-contained microservices, we target modularized load tests that can be executed against one or few services. (3) In addition to that, we plan to enrich the load tests with contextual information to restrict the load test scenario to the respectively relevant context. In combination with (2), we assume this approach to significantly reduce the test execution time. (4) In order to finally minimize the needed resources as well, we want to introduce performance stubs that can be used as replacements of dependent services during a load test.

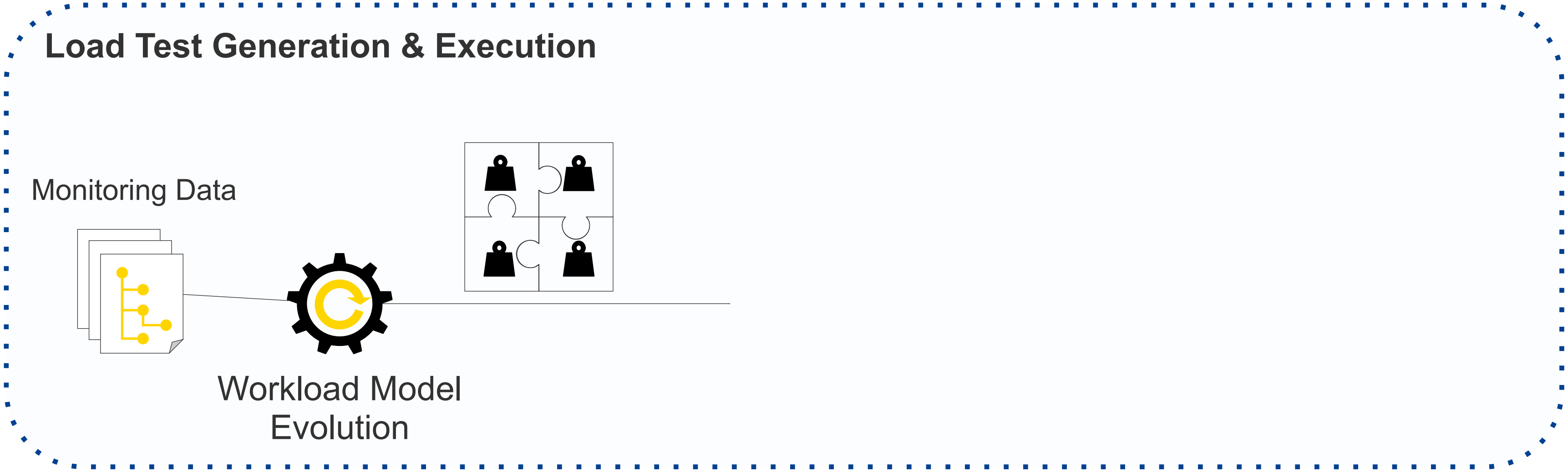
Summing up, this paper presents our vision of load testing in continuous software engineering and provides insights into first evaluation results of our load test adjustment evolution approach.

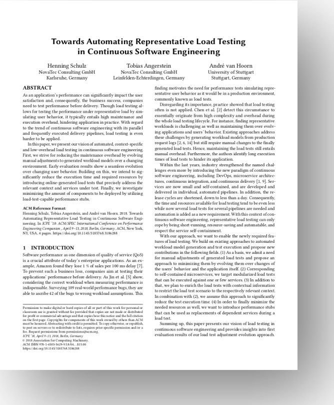
ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5629-9/18/04...$15.00
https://doi.org/10.1145/3185768.3186288

Henning Schulz, Tobias Angerstein, and André van Hoorn:
Towards automating representative load testing in continuous software engineering.

In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE 2018) Companion (7th International Workshop on Load Testing and Benchmarking of Software Systems, LTB 2018), pages 123–126. ACM, 2018

ContinuITy





Towards Automating Representative Load Testing in Continuous Software Engineering

Henning Schulz
NovaTec Consulting GmbH
Karlsruhe, Germany

Tobias Angerstein
NovaTec Consulting GmbH
Leinfelden-Echterdingen, Germany

André van Hoorn
University of Stuttgart
Stuttgart, Germany

ABSTRACT

As an application's performance can significantly impact the user satisfaction and, consequently, the business success, companies need to test performance before delivery. Though load testing allows for testing the performance under representative load by simulating user behavior, it typically entails high maintenance and execution overhead, hindering application in practice. With regard to the trend of continuous software engineering with its parallel and frequently executed delivery pipelines, load testing is even harder to be applied.

In this paper, we present our vision of automated, context-specific and low-overhead load testing in continuous software engineering. First, we strive for reducing the maintenance overhead by evolving manual adjustments to generated workload models over a changing environment. Early evaluation results show a seamless evolution over changing user behavior. Building on this, we intend to significantly reduce the execution time and required resources by introducing online-generated load tests that precisely address the relevant context and services under test. Finally, we investigate minimizing the amount of components to be deployed by utilizing load-test-capable performance stubs.

ACM Reference Format:
Henning Schulz, Tobias Angerstein, and André van Hoorn. 2018. Towards Automating Representative Load Testing in Continuous Software Engineering. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186288

1 INTRODUCTION

Software performance as one dimension of quality of service (QoS) is a crucial attribute of today's enterprise applications. As an example, Amazon found they lose 1 % of sales per 100 ms delay [7]. To prevent such a business loss, companies aim at testing their applications' performance before delivery. As Jin et al. [5] show, considering the correct workload when measuring performance is indispensable. Surveying 109 real-world performance bugs, they are able to ascribe 42 of the bugs to wrong workload assumptions. This finding motivates the need for performance tests simulating representative user behavior as it would be in a production environment, commonly known as load tests.

Disregarding its importance, practice showed that load testing often is not applied. Chen et al. [2] detect this circumstance to essentially originate from high complexity and overhead during the whole load testing lifecycle. For instance, finding representative workloads is challenging as well as maintaining them over evolving applications and users' behavior. Existing approaches address these challenges by generating workload models from production request logs [2, 6, 14] but still require manual changes to the finally generated load tests. Hence, maintaining the load tests still entails manual overhead. Furthermore, the authors identify long execution times of load tests to hinder its application.

Within the last years, industry strengthened the named challenges even more by introducing the new paradigm of continuous software engineering, including DevOps, microservice architectures, continuous integration, and continuous delivery [1, 9]. Services are now small and self-contained, and are developed and delivered in individual, automated pipelines. In addition, the release cycles are shortened, down to less than a day. Consequently, the time and resources available for load testing tend to be even less while now several load tests for several pipelines are needed and automation is added as a new requirement. With this context of continuous software engineering, representative load testing can only cope by being short-running, resource-saving and automatable, and respect the service self-containment.

With our approach, we want to enable the newly required features of load testing. We build on existing approaches to automated workload model generation and test execution and propose new contributions in the following fields. (1) As a basis, we admit a need for manual adjustments of generated load tests and propose an approach to minimizing them by evolving them over changes of the users' behavior and the application itself. (2) Corresponding to self-contained microservices, we target modularized load tests that can be executed against one or few services. (3) In addition to that, we plan to enrich the load tests with contextual information to restrict the load test scenario to the respectively relevant context. In combination with (2), we assume this approach to significantly reduce the test execution time. (4) In order to finally minimize the needed resources as well, we want to introduce performance stubs that can be used as replacements of dependent services during a load test.

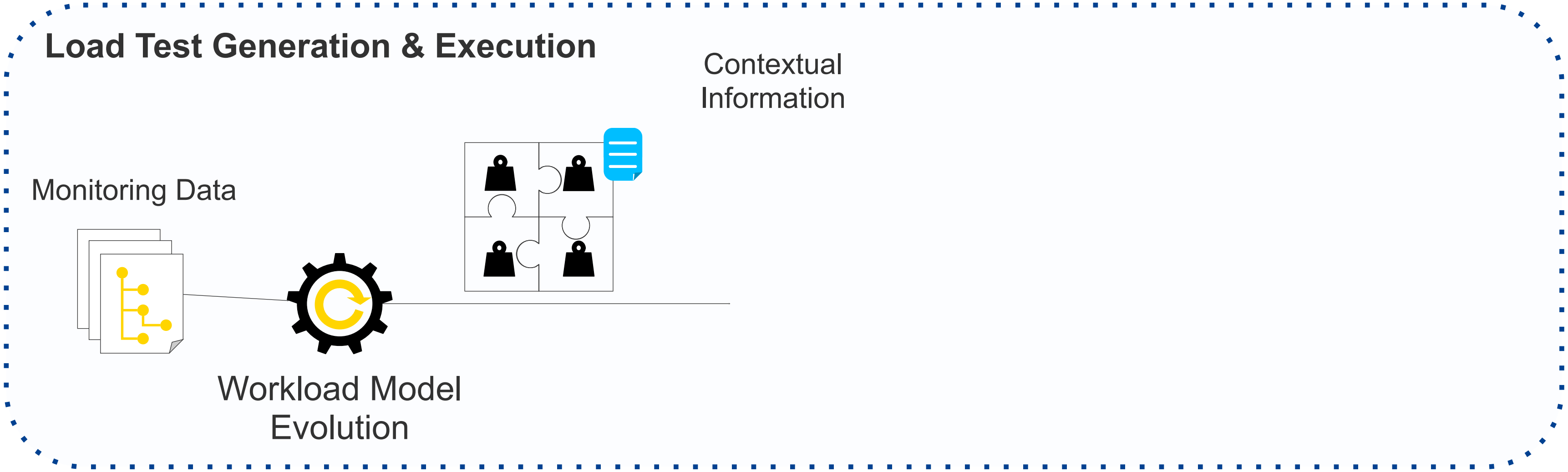
Summing up, this paper presents our vision of load testing in continuous software engineering and provides insights into first evaluation results of our load test adjustment evolution approach.

ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5629-9/18/04...$15.00
https://doi.org/10.1145/3185768.3186288

Henning Schulz, Tobias Angerstein, and André van Hoorn:
Towards automating representative load testing in continuous software engineering.

In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE 2018) Companion (7th International Workshop on Load Testing and Benchmarking of Software Systems, LTB 2018), pages 123–126. ACM, 2018

ContinuITy

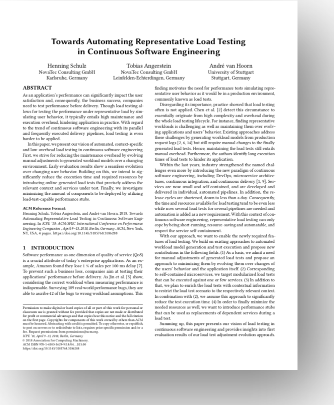


Load Test Generation & Execution

Monitoring Data

Workload Model Evolution

Contextual Information



Towards Automating Representative Load Testing in Continuous Software Engineering

Henning Schulz
NovaTec Consulting GmbH
Karlsruhe, Germany

Tobias Angerstein
NovaTec Consulting GmbH
Leinfelden-Echterdingen, Germany

André van Hoorn
University of Stuttgart
Stuttgart, Germany

ABSTRACT

As an application's performance can significantly impact the user satisfaction and, consequently, the business success, companies need to test performance before delivery. Though load testing allows for testing the performance under representative load by simulating user behavior, it typically entails high maintenance and execution overhead, hindering application in practice. With regard to the trend of continuous software engineering with its parallel and frequently executed delivery pipelines, load testing is even harder to be applied.

In this paper, we present our vision of automated, context-specific and low-overhead load testing in continuous software engineering. First, we strive for reducing the maintenance overhead by evolving manual adjustments to generated workload models over a changing environment. Early evaluation results show a seamless evolution over changing user behavior. Building on this, we intend to significantly reduce the execution time and required resources by introducing online-generated load tests that precisely address the relevant context and services under test. Finally, we investigate minimizing the amount of components to be deployed by utilizing load-test-capable performance stubs.

ACM Reference Format:
Henning Schulz, Tobias Angerstein, and André van Hoorn. 2018. Towards Automating Representative Load Testing in Continuous Software Engineering. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186288

1 INTRODUCTION

Software performance as one dimension of quality of service (QoS) is a crucial attribute of today's enterprise applications. As an example, Amazon found they lose 1 % of sales per 100 ms delay [7]. To prevent such a business loss, companies aim at testing their applications' performance before delivery. As Jin et al. [5] show, considering the correct workload when measuring performance is indispensable. Surveying 109 real-world performance bugs, they are able to ascribe 42 of the bugs to wrong workload assumptions. This finding motivates the need for performance tests simulating representative user behavior as it would be in a production environment, commonly known as load tests.

Disregarding its importance, practice showed that load testing often is not applied. Chen et al. [2] detect this circumstance to essentially originate from high complexity and overhead during the whole load testing lifecycle. For instance, finding representative workloads is challenging as well as maintaining them over evolving applications and users' behavior. Existing approaches address these challenges by generating workload models from production request logs [2, 6, 14] but still require manual changes to the finally generated load tests. Hence, maintaining the load tests still entails manual overhead. Furthermore, the authors identify long execution times of load tests to hinder its application.

Within the last years, industry strengthened the named challenges even more by introducing the new paradigm of continuous software engineering, including DevOps, microservice architectures, continuous integration, and continuous delivery [1, 9]. Services are now small and self-contained, and are developed and delivered in individual, automated pipelines. In addition, the release cycles are shortened, down to less than a day. Consequently, the time and resources available for load testing tend to be even less while now several load tests for several pipelines are needed and automation is added as a new requirement. With this context of continuous software engineering, representative load testing can only cope by being short-running, resource-saving and automatable, and respect the service self-containment.

With our approach, we want to enable the newly required features of load testing. We build on existing approaches to automated workload model generation and test execution and propose new contributions in the following fields. (1) As a basis, we admit a need for manual adjustments of generated load tests and propose an approach to minimizing them by evolving them over changes of the users' behavior and the application itself. (2) Corresponding to self-contained microservices, we target modularized load tests that can be executed against one or few services. (3) In addition to that, we plan to enrich the load tests with contextual information to restrict the load test scenario to the respectively relevant context. In combination with (2), we assume this approach to significantly reduce the test execution time. (4) In order to finally minimize the needed resources as well, we want to introduce performance stubs that can be used as replacements of dependent services during a load test.

Summing up, this paper presents our vision of load testing in continuous software engineering and provides insights into first evaluation results of our load test adjustment evolution approach.

ICPE '18, April 9–13, 2018, Berlin, Germany

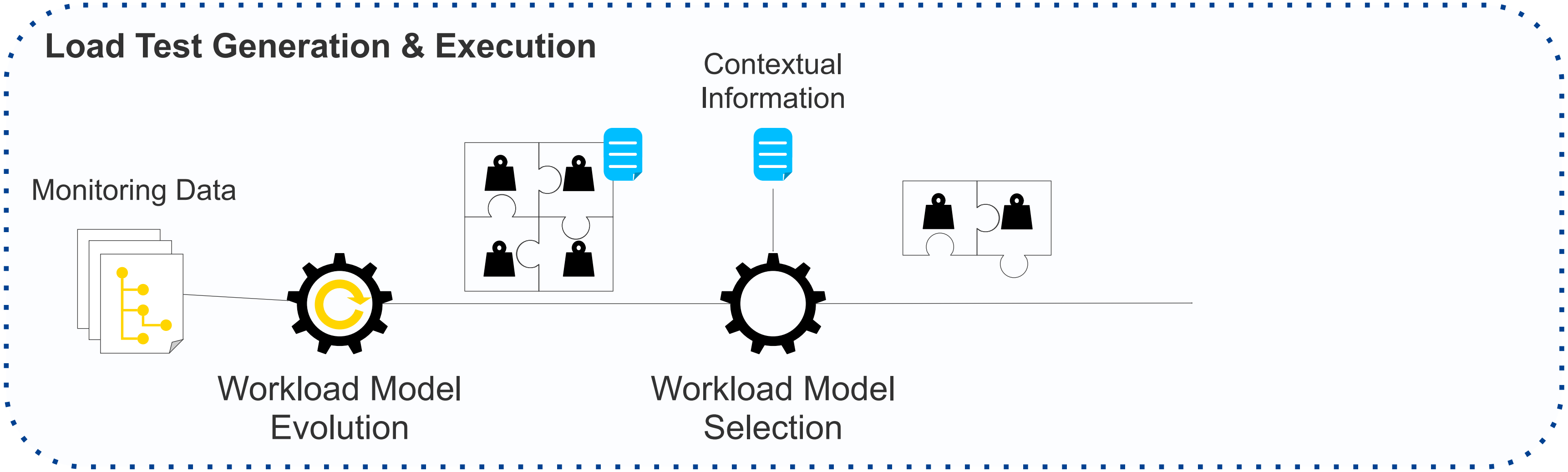
ACM ISBN 978-1-4503-5629-9/18/04...$15.00
https://doi.org/10.1145/3185768.3186288

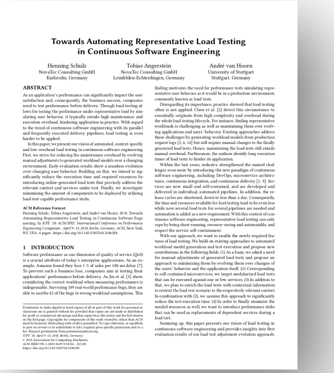
Henning Schulz, Tobias Angerstein, and André van Hoorn:
Towards automating representative load testing in continuous software engineering.

In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE 2018) Companion (7th International Workshop on Load Testing and Benchmarking of Software Systems, LTB 2018), pages 123–126. ACM, 2018

unibz
Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

ContinuITy





Towards Automating Representative Load Testing in Continuous Software Engineering

Henning Schulz
NovaTec Consulting GmbH
Karlsruhe, Germany

Tobias Angerstein
NovaTec Consulting GmbH
Leinfelden-Echterdingen, Germany

André van Hoorn
University of Stuttgart
Stuttgart, Germany

ABSTRACT

As an application's performance can significantly impact the user satisfaction and, consequently, the business success, companies need to test performance before delivery. Though load testing allows for testing the performance under representative load by simulating user behavior, it typically entails high maintenance and execution overhead, hindering application in practice. With regard to the trend of continuous software engineering with its parallel and frequently executed delivery pipelines, load testing is even harder to be applied.

In this paper, we present our vision of automated, context-specific and low-overhead load testing in continuous software engineering. First, we strive for reducing the maintenance overhead by evolving manual adjustments to generated workload models over a changing environment. Early evaluation results show a seamless evolution over changing user behavior. Building on this, we intend to significantly reduce the execution time and required resources by introducing online-generated load tests that precisely address the relevant context and services under test. Finally, we investigate minimizing the amount of components to be deployed by utilizing load-test-capable performance stubs.

ACM Reference Format:
Henning Schulz, Tobias Angerstein, and André van Hoorn. 2018. Towards Automating Representative Load Testing in Continuous Software Engineering. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186288

1 INTRODUCTION

Software performance as one dimension of quality of service (QoS) is a crucial attribute of today's enterprise applications. As an example, Amazon found they lose 1 % of sales per 100 ms delay [7]. To prevent such a business loss, companies aim at testing their applications' performance before delivery. As Jin et al. [5] show, considering the correct workload when measuring performance is indispensable. Surveying 109 real-world performance bugs, they are able to ascribe 42 of the bugs to wrong workload assumptions. This finding motivates the need for performance tests simulating representative user behavior as it would be in a production environment, commonly known as load tests.

Disregarding its importance, practice showed that load testing often is not applied. Chen et al. [2] detect this circumstance to essentially originate from high complexity and overhead during the whole load testing lifecycle. For instance, finding representative workloads is challenging as well as maintaining them over evolving applications and users' behavior. Existing approaches address these challenges by generating workload models from production request logs [2, 6, 14] but still require manual changes to the finally generated load tests. Hence, maintaining the load tests still entails manual overhead. Furthermore, the authors identify long execution times of load tests to hinder its application.

Within the last years, industry strengthened the named challenges even more by introducing the new paradigm of continuous software engineering, including DevOps, microservice architectures, continuous integration, and continuous delivery [1, 9]. Services are now small and self-contained, and are developed and delivered in individual, automated pipelines. In addition, the release cycles are shortened, down to less than a day. Consequently, the time and resources available for load testing tend to be even less while now several load tests for several pipelines are needed and automation is added as a new requirement. With this context of continuous software engineering, representative load testing can only cope by being short-running, resource-saving and automatable, and respect the service self-containment.

With our approach, we want to enable the newly required features of load testing. We build on existing approaches to automated workload model generation and test execution and propose new contributions in the following fields. (1) As a basis, we admit a need for manual adjustments of generated load tests and propose an approach to minimizing them by evolving them over changes of the users' behavior and the application itself. (2) Corresponding to self-contained microservices, we target modularized load tests that can be executed against one or few services. (3) In addition to that, we plan to enrich the load tests with contextual information to restrict the load test scenario to the respectively relevant context. In combination with (2), we assume this approach to significantly reduce the test execution time. (4) In order to finally minimize the needed resources as well, we want to introduce performance stubs that can be used as replacements of dependent services during a load test.

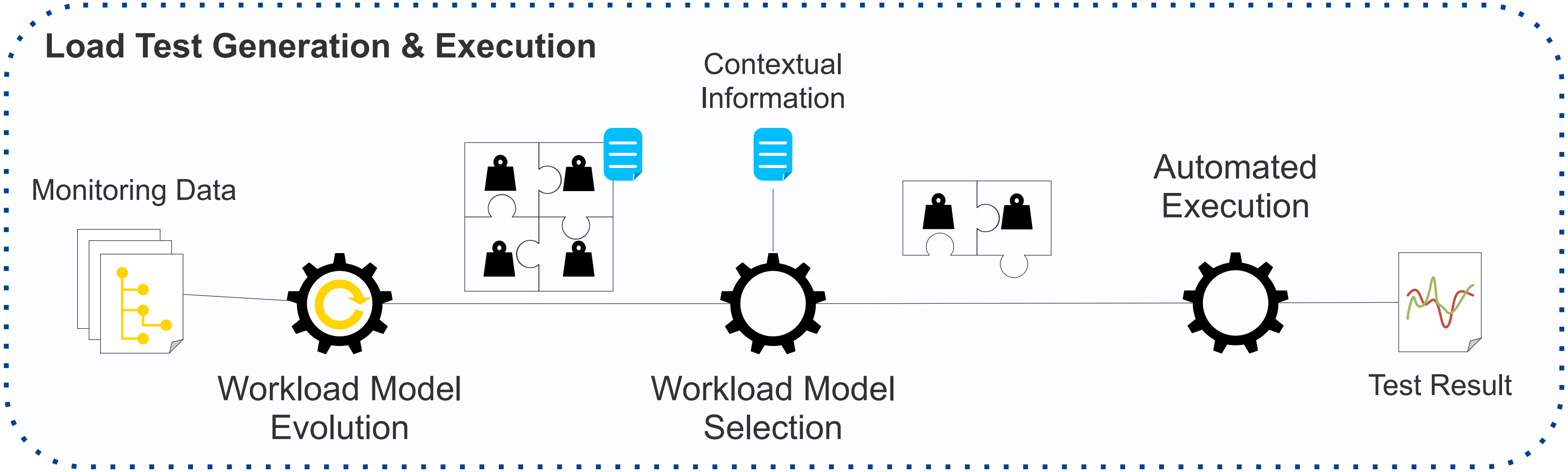
Summing up, this paper presents our vision of load testing in continuous software engineering and provides insights into first evaluation results of our load test adjustment evolution approach.

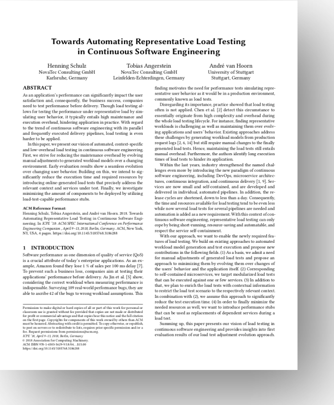
ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5629-9/18/04…$15.00
https://doi.org/10.1145/3185768.3186288

Henning Schulz, Tobias Angerstein, and André van Hoorn: *Towards automating representative load testing in continuous software engineering*.

In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE 2018) Companion (7th International Workshop on Load Testing and Benchmarking of Software Systems, LTB 2018), pages 123–126. ACM, 2018

ContinuITy





Towards Automating Representative Load Testing in Continuous Software Engineering

Henning Schulz
NovaTec Consulting GmbH
Karlsruhe, Germany

Tobias Angerstein
NovaTec Consulting GmbH
Leinfelden-Echterdingen, Germany

André van Hoorn
University of Stuttgart
Stuttgart, Germany

ABSTRACT

As an application's performance can significantly impact the user satisfaction and, consequently, the business success, companies need to test performance before delivery. Though load testing allows for testing the performance under representative load by simulating user behavior, it typically entails high maintenance and execution overhead, hindering application in practice. With regard to the trend of continuous software engineering with its parallel and frequently executed delivery pipelines, load testing is even harder to be applied.

In this paper, we present our vision of automated, context-specific and low-overhead load testing in continuous software engineering. First, we strive for reducing the maintenance overhead by evolving manual adjustments to generated workload models over a changing environment. Early evaluation results show a seamless evolution over changing user behavior. Building on this, we intend to significantly reduce the execution time and required resources by introducing online-generated load tests that precisely address the relevant context and services under test. Finally, we investigate minimizing the amount of components to be deployed by utilizing load-test-capable performance stubs.

ACM Reference Format:
Henning Schulz, Tobias Angerstein, and André van Hoorn. 2018. Towards Automating Representative Load Testing in Continuous Software Engineering. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering Companion , April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3185768.3186288

1 INTRODUCTION

Software performance as one dimension of quality of service (QoS) is a crucial attribute of today's enterprise applications. As an example, Amazon found they lose 1 % of sales per 100 ms delay [7]. To prevent such a business loss, companies aim at testing their applications' performance before delivery. As Jin et al. [5] show, considering the correct workload when measuring performance is indispensable. Surveying 109 real-world performance bugs, they are able to ascribe 42 of the bugs to wrong workload assumptions. This finding motivates the need for performance tests simulating representative user behavior as it would be in a production environment, commonly known as load tests.

Disregarding its importance, practice showed that load testing often is not applied. Chen et al. [2] detect this circumstance to essentially originate from high complexity and overhead during the whole load testing lifecycle. For instance, finding representative workloads is challenging as well as maintaining them over evolving applications and users' behavior. Existing approaches address these challenges by generating workload models from production request logs [2, 6, 14] but still require manual changes to the finally generated load tests. Hence, maintaining the load tests still entails manual overhead. Furthermore, the authors identify long execution times of load tests to hinder its application.

Within the last years, industry strengthened the named challenges even more by introducing the new paradigm of continuous software engineering, including DevOps, microservice architectures, continuous integration, and continuous delivery [1, 9]. Services are now small and self-contained, and are developed and delivered in individual, automated pipelines. In addition, the release cycles are shortened, down to less than a day. Consequently, the time and resources available for load testing tend to be even less while now several load tests for several pipelines are needed and automation is added as a new requirement. With this context of continuous software engineering, representative load testing can only cope by being short-running, resource-saving and automatable, and respect the service self-containment.

With our approach, we want to enable the newly required features of load testing. We build on existing approaches to automated workload model generation and test execution and propose new contributions in the following fields. (1) As a basis, we admit a need for manual adjustments of generated load tests and propose an approach to minimizing them by evolving them over changes of the users' behavior and the application itself. (2) Corresponding to self-contained microservices, we target modularized load tests that can be executed against one or few services. (3) In addition to that, we plan to enrich the load tests with contextual information to restrict the load test scenario to the respectively relevant context. In combination with (2), we assume this approach to significantly reduce the test execution time. (4) In order to finally minimize the needed resources as well, we want to introduce performance stubs that can be used as replacements of dependent services during a load test.

Summing up, this paper presents our vision of load testing in continuous software engineering and provides insights into first evaluation results of our load test adjustment evolution approach.

ICPE '18, April 9–13, 2018, Berlin, Germany
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5629-9/18/04...$15.00
https://doi.org/10.1145/3185768.3186288

Henning Schulz, Tobias Angerstein, and André van Hoorn: *Towards automating representative load testing in continuous software engineering*.

In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE 2018) Companion (7th International Workshop on Load Testing and Benchmarking of Software Systems, LTB 2018), pages 123–126. ACM, 2018

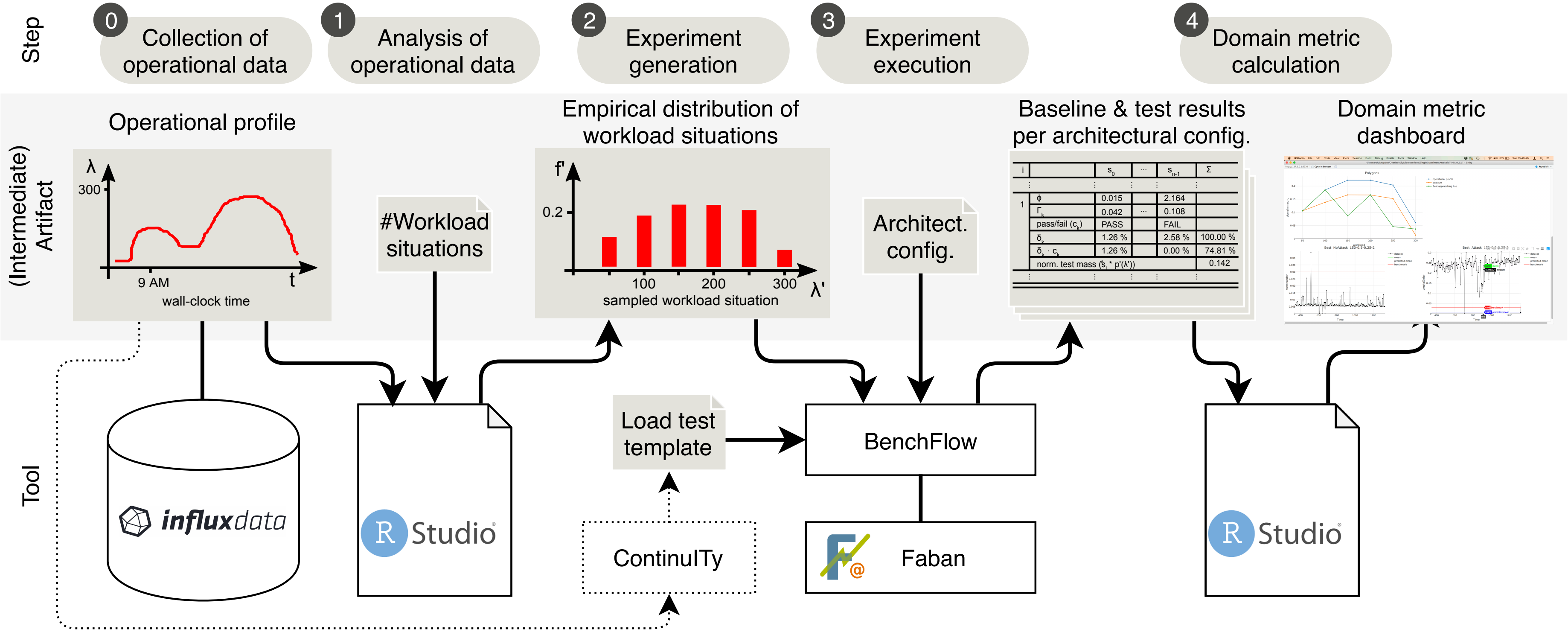
Load test automation

What is it for?

- Identify deployment configuration(s) for which the system performs best for all workloads
- Characterize systems‘ resilience over workload
- Analyse individual service failure or degradation
- Reveal attacks
- Monitor the performance in a transition to microservices

PPTAM

Production and Performance Testing-Based Application Monitoring



Step

0 Collection of operational data

1 Analysis of operational data

2 Experiment generation

3 Experiment execution

4 Domain metric calculation

(Intermediate) Artifact

Operational profile

#Workload situations

Empirical distribution of workload situations

Architect. config.

Baseline & test results per architectural config.

Domain metric dashboard

Tool

influxdata

RStudio

Load test template

ContinuITy

BenchFlow

Faban

RStudio

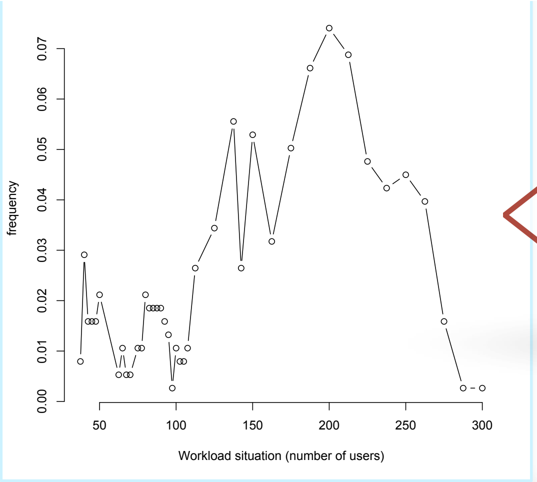
i		s_0	···	s_{n-1}	Σ
⋮	⋮	⋮	⋮	⋮	
1	ϕ	0.015		2.164	
	Γ_k	0.042	···	0.108	
	pass/fail (c_k)	PASS		FAIL	
	δ_k	1.26 %		2.58 %	100.00 %
	$\delta_k \cdot c_k$	1.26 %		0.00 %	74.81 %
	norm. test mass ($\hat{s}_i$ * p'(λ'))				0.142
⋮	⋮	⋮	⋮	⋮	

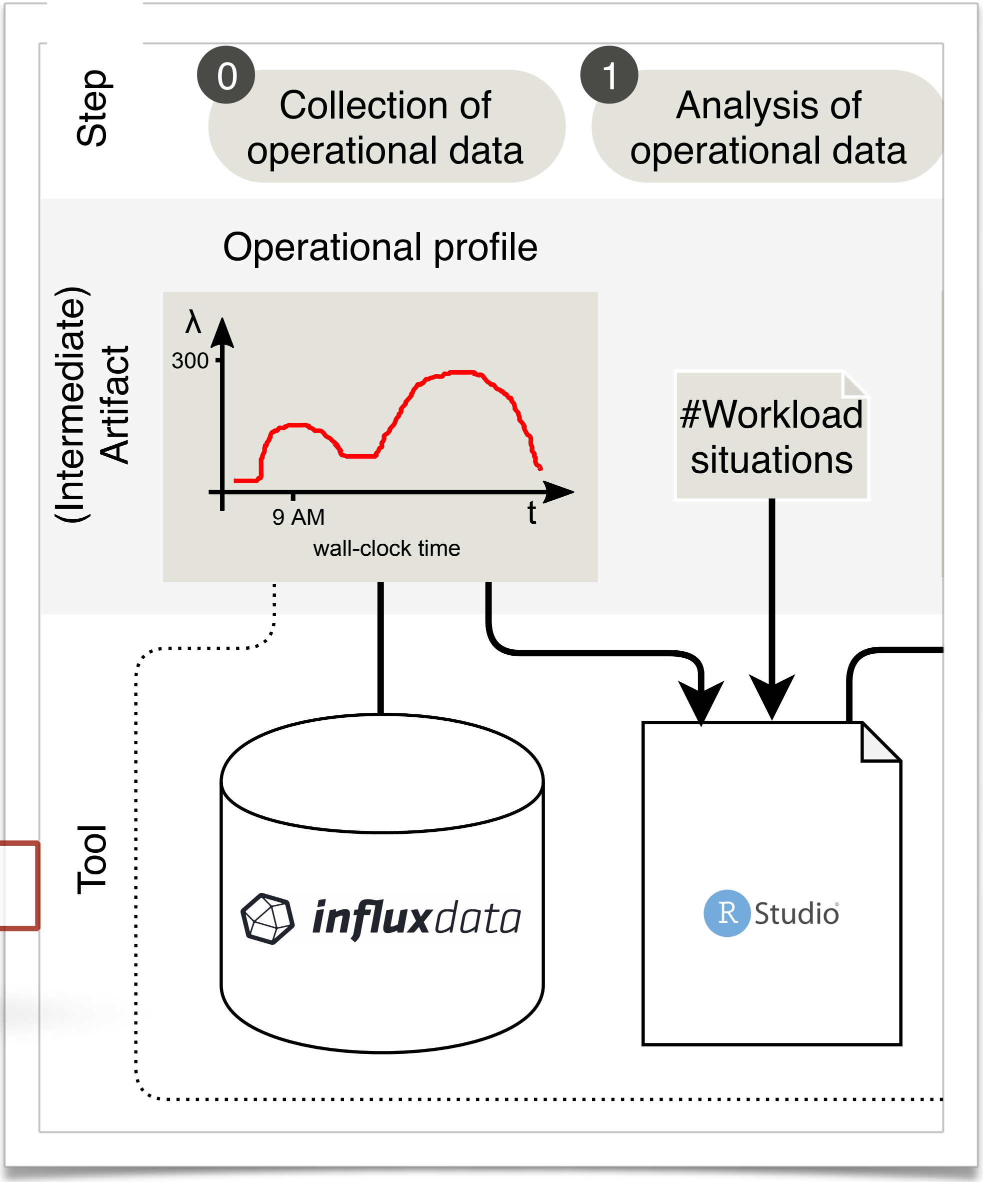
What it does?

- It **collects the operational profile** of a microservice system
- It **automatically runs a series of experiments** with given usage profiles and probability of use (R scripts)
- It **runs experiments** according to templates that control time, number of agents, and single operation max response (Faban)
- It sets the **goals of testing (e.g., resource config)** and **collects data for each experiment** or multi-experiment (Benchflow)
- It **identifies failing services** (baseline threshold R scripts)
- It **computes a total metric of performance** on non-failing services for each system configuration (CPU, memory, replicas) over workloads (via R scripts)
- It visualizes into interactive graphs (R shop, R shiny, and R plottly)

What it does

- Collects the operational data of systems
- Builds the operational profile




Step

0 Collection of operational data

1 Analysis of operational data

(Intermediate) Artifact

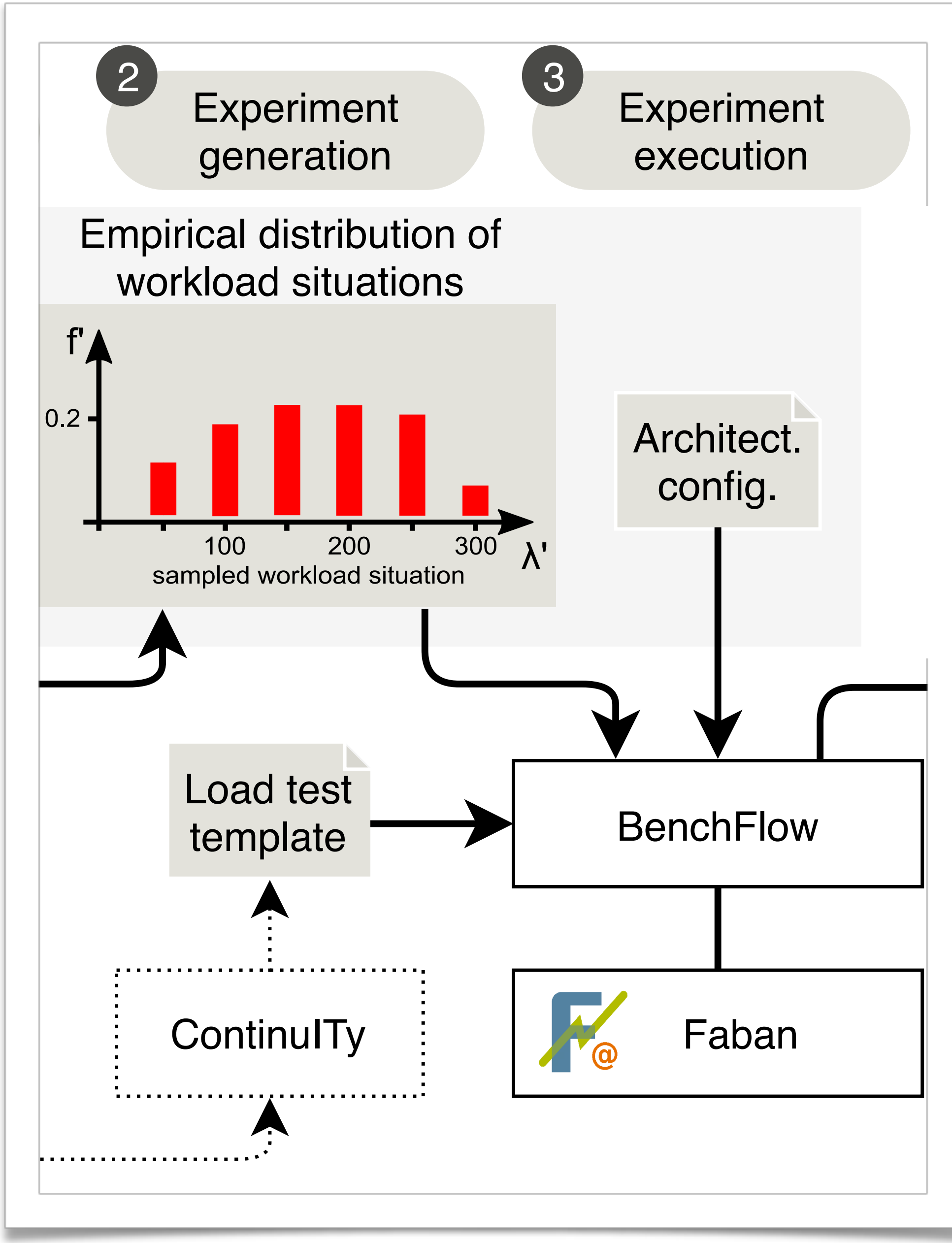
Operational profile

#Workload situations

Tool

What it does

- Automatically runs a series of experiments with given usage profiles and probability of use



BenchFlow Automation Framework

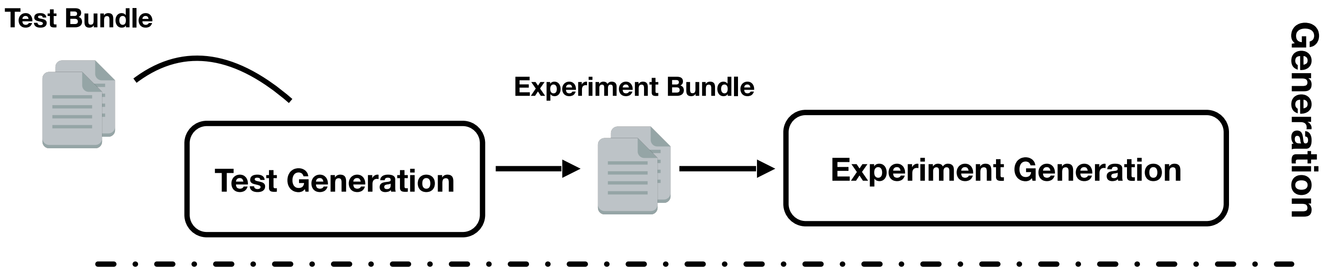
BenchFlow Automation Framework

Test Generation
Generation

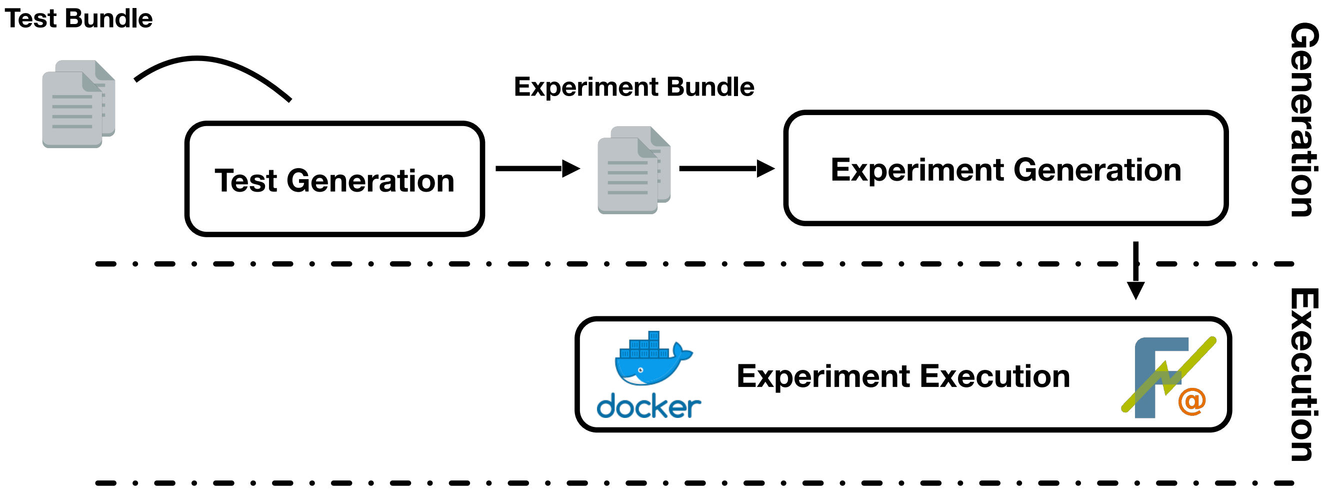
BenchFlow Automation Framework

Test Bundle
Test Generation
Generation

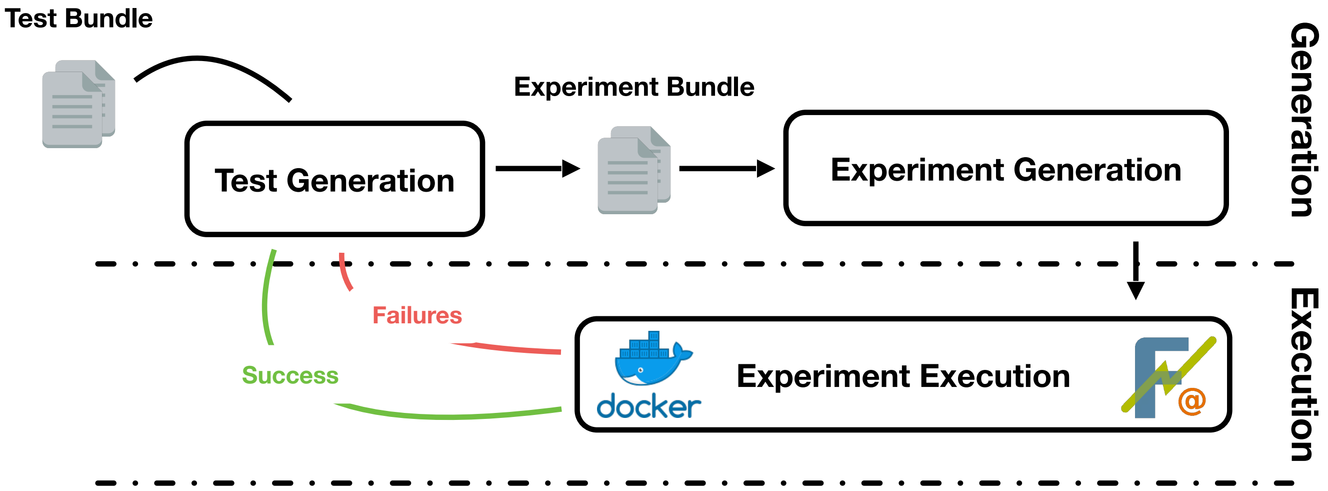
BenchFlow Automation Framework



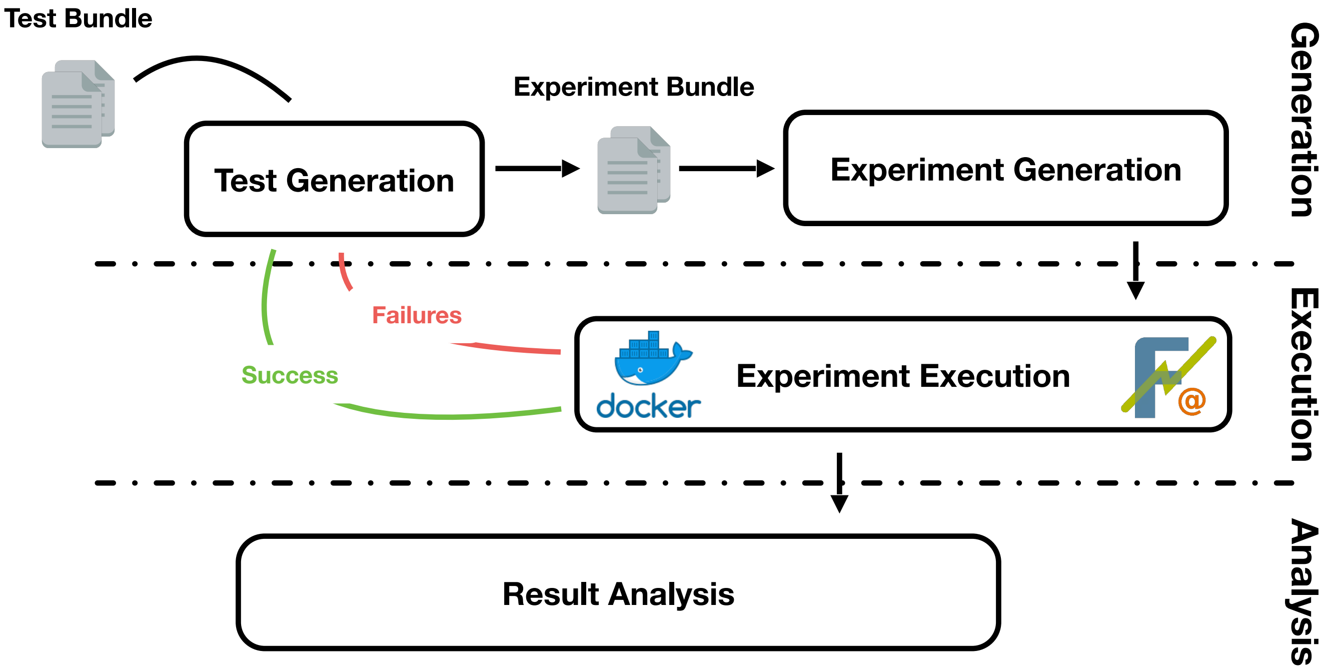
BenchFlow Automation Framework



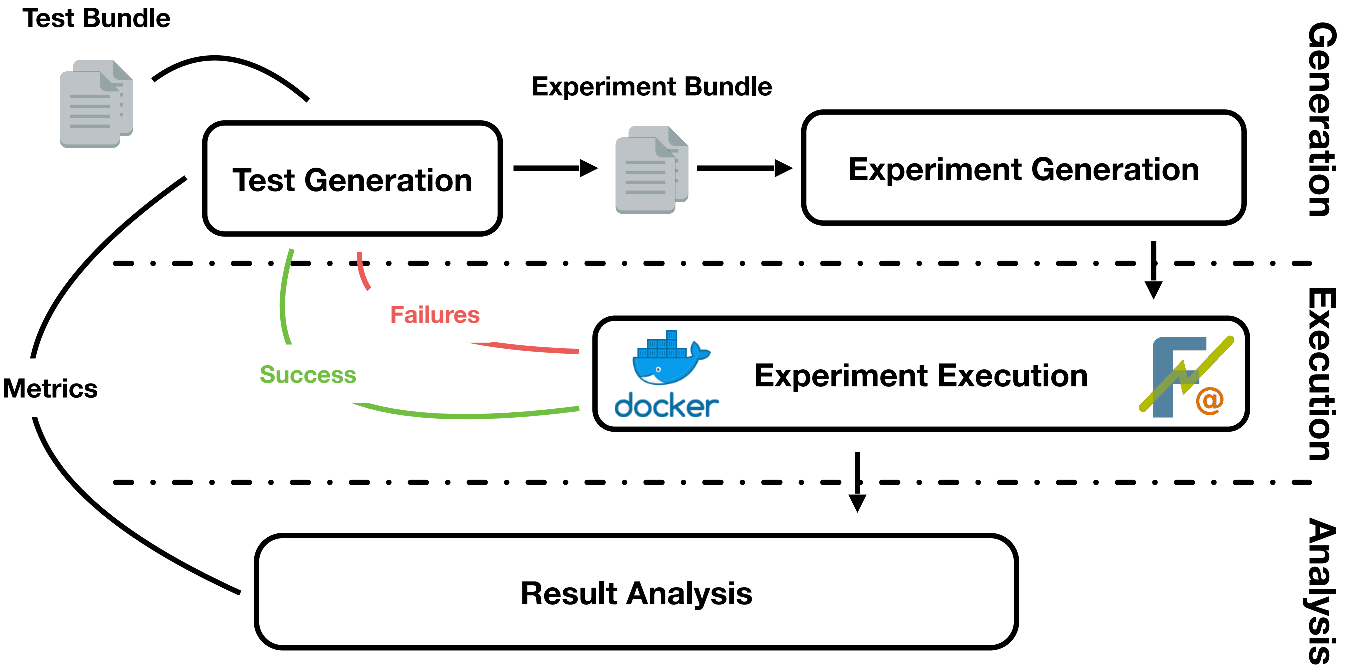
BenchFlow Automation Framework



BenchFlow Automation Framework

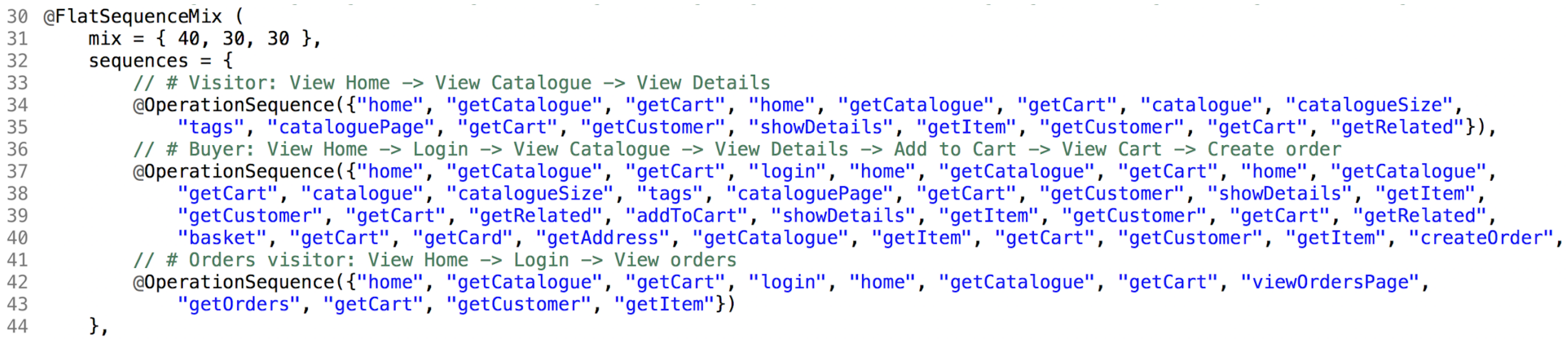


BenchFlow Automation Framework



Experiment execution design (faban)

- Usage profiles (WebDriver.java)



```java
@FlatSequenceMix (
    mix = { 40, 30, 30 },
    sequences = {
        // # Visitor: View Home -> View Catalogue -> View Details
        @OperationSequence({"home", "getCatalogue", "getCart", "home", "getCatalogue", "getCart", "catalogue", "catalogueSize",
            "tags", "cataloguePage", "getCart", "getCustomer", "showDetails", "getItem", "getCustomer", "getCart", "getRelated"}),
        // # Buyer: View Home -> Login -> View Catalogue -> View Details -> Add to Cart -> View Cart -> Create order
        @OperationSequence({"home", "getCatalogue", "getCart", "login", "home", "getCatalogue", "getCart", "home", "getCatalogue",
            "getCart", "catalogue", "catalogueSize", "tags", "cataloguePage", "getCart", "getCustomer", "showDetails", "getItem",
            "getCustomer", "getCart", "getRelated", "addToCart", "showDetails", "getItem", "getCustomer", "getCart", "getRelated",
            "basket", "getCart", "getCard", "getAddress", "getCatalogue", "getItem", "getCart", "getCustomer", "getItem", "createOrder",
        // # Orders visitor: View Home -> Login -> View orders
        @OperationSequence({"home", "getCatalogue", "getCart", "login", "home", "getCatalogue", "getCart", "viewOrdersPage",
            "getOrders", "getCart", "getCustomer", "getItem"})
    },
```

- Configuration file.xml

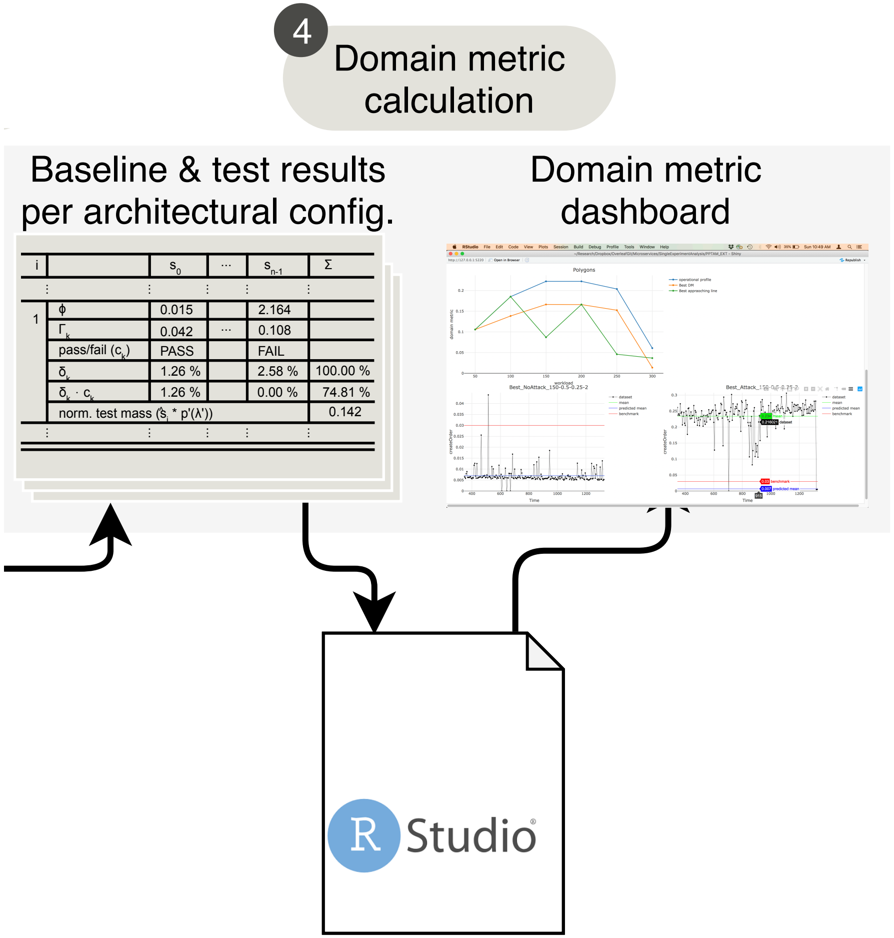
```xml
<!-- The rampup, steadystate, and rampdown of the driver -->
  <fa:runControl unit="time">
   <fa:rampUp>60</fa:rampUp>
   <fa:steadyState>1800</fa:steadyState>
   <fa:rampDown>0</fa:rampDown>
  </fa:runControl>
```

```xml
<!-- The number of agents, or host:agents pairs
        separated by space -->
<agents>10</agents>
```

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

What it does

- Service failures over time
- Total performance of system in use
- Per (micro)service performance time series
- Performance degradation under an attack



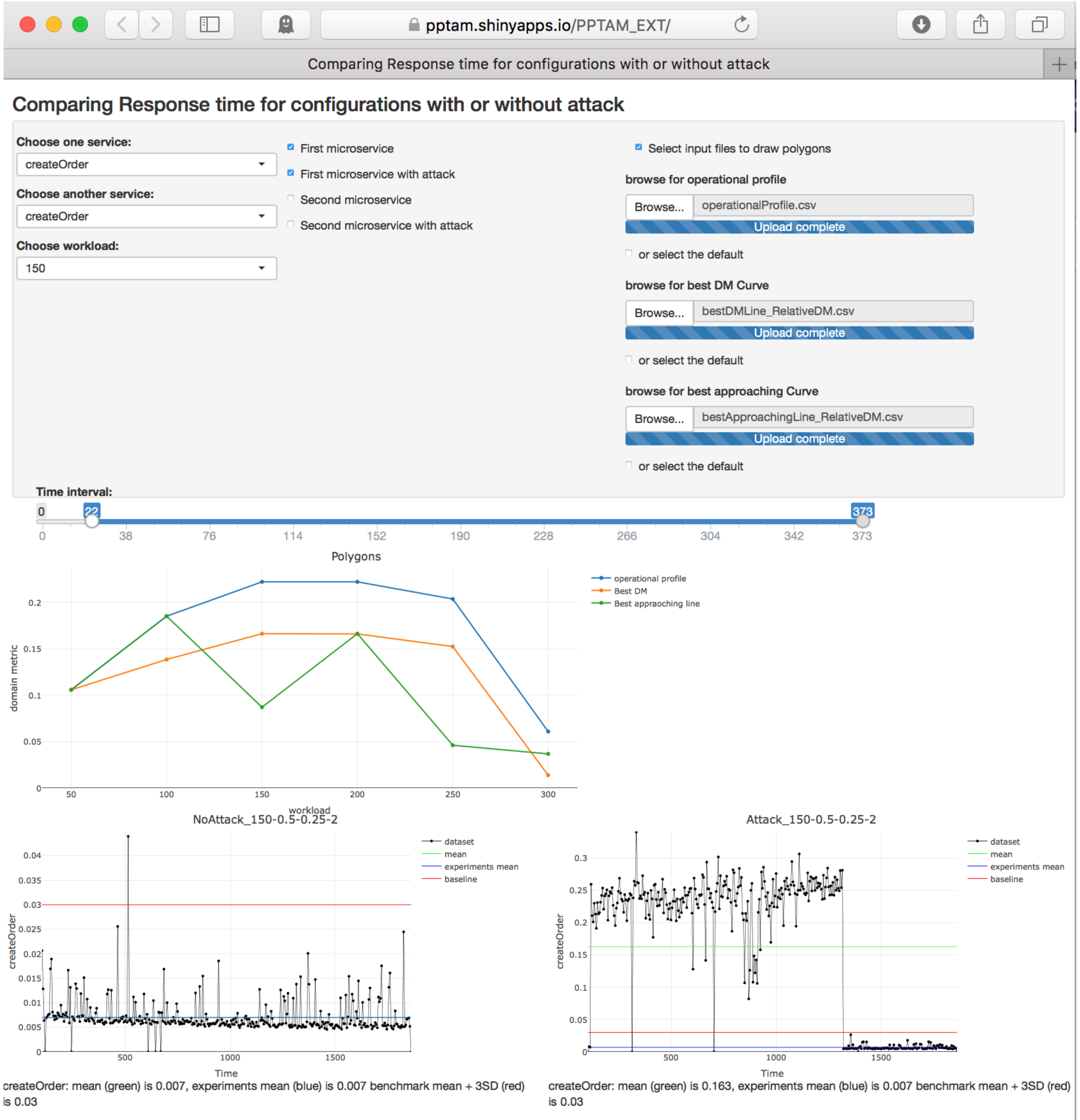
4 Domain metric calculation

Baseline & test results per architectural config.

i		s_0	···	s_{n-1}	Σ
⋮		⋮	⋮	⋮	⋮
1	ϕ	0.015		2.164	
	Γ_k	0.042	···	0.108	
	pass/fail (c_k)	PASS		FAIL	
	δ_k	1.26 %		2.58 %	100.00 %
	$\delta_k \cdot c_k$	1.26 %		0.00 %	74.81 %
	norm. test mass ($\hat{s}_i$ * p'(λ'))				0.142
⋮		⋮	⋮	⋮	⋮

Domain metric dashboard

https://pptam.shinyapps.io/PPTAM_EXT/



pptam.shinyapps.io/PPTAM_EXT/

Comparing Response time for configurations with or without attack

Comparing Response time for configurations with or without attack

Choose one service: createOrder

Choose another service: createOrder

Choose workload: 150

☑ First microservice

☑ First microservice with attack

☐ Second microservice

☐ Second microservice with attack

☑ Select input files to draw polygons

browse for operational profile

Browse... operationalProfile.csv

Upload complete

☐ or select the default

browse for best DM Curve

Browse... bestDMLine_RelativeDM.csv

Upload complete

☐ or select the default

browse for best approaching Curve

Browse... bestApproachingLine_RelativeDM.csv

Upload complete

☐ or select the default

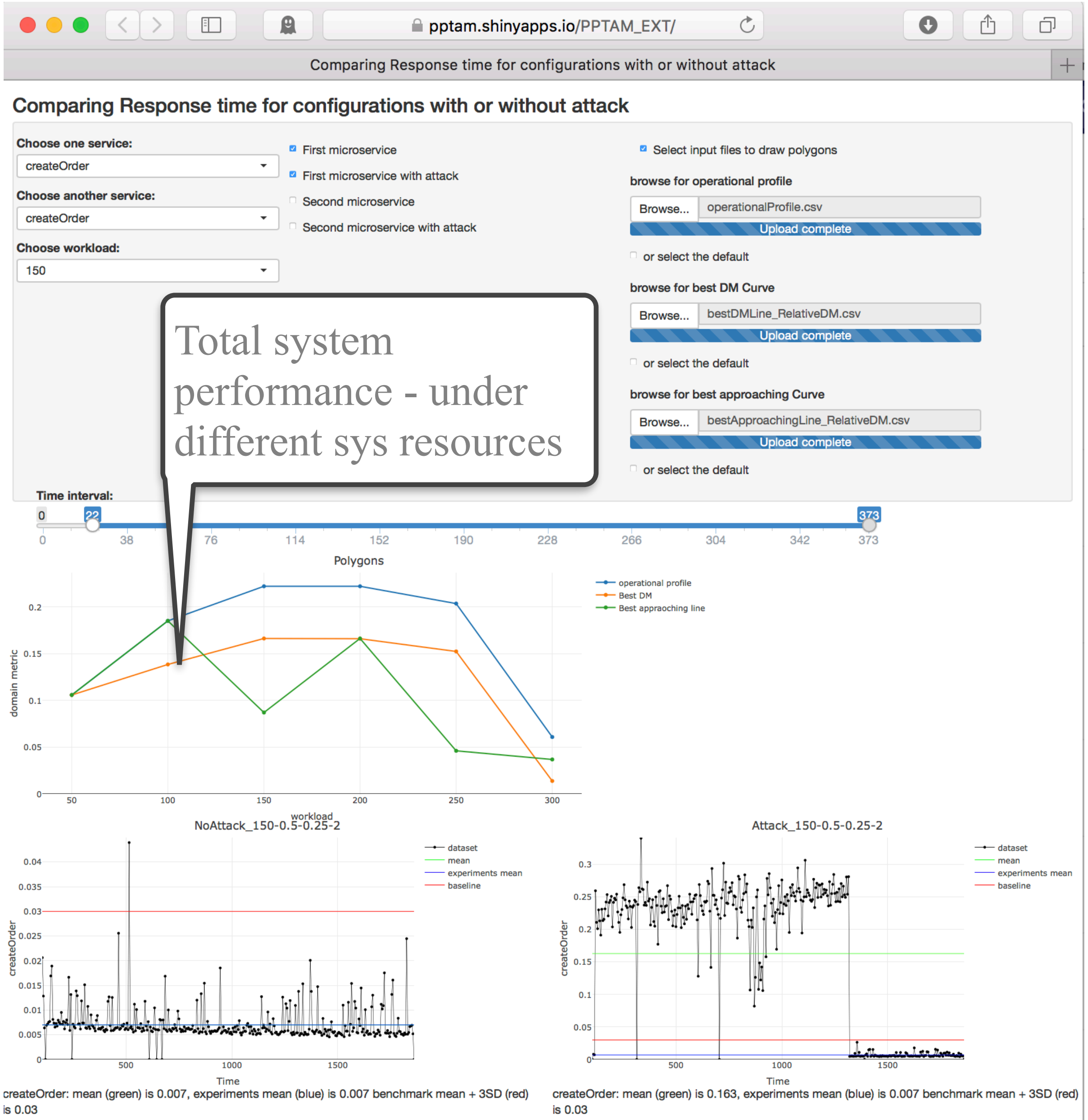
Time interval: 22 – 373

createOrder: mean (green) is 0.007, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

createOrder: mean (green) is 0.163, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

unibz

https://pptam.shinyapps.io/PPTAM_EXT/



pptam.shinyapps.io/PPTAM_EXT/

Comparing Response time for configurations with or without attack

Comparing Response time for configurations with or without attack

Choose one service: createOrder

Choose another service: createOrder

Choose workload: 150

☑ First microservice

☑ First microservice with attack

□ Second microservice

□ Second microservice with attack

☑ Select input files to draw polygons

browse for operational profile

Browse... operationalProfile.csv

Upload complete

□ or select the default

browse for best DM Curve

Browse... bestDMLine_RelativeDM.csv

Upload complete

□ or select the default

browse for best approaching Curve

Browse... bestApproachingLine_RelativeDM.csv

Upload complete

□ or select the default

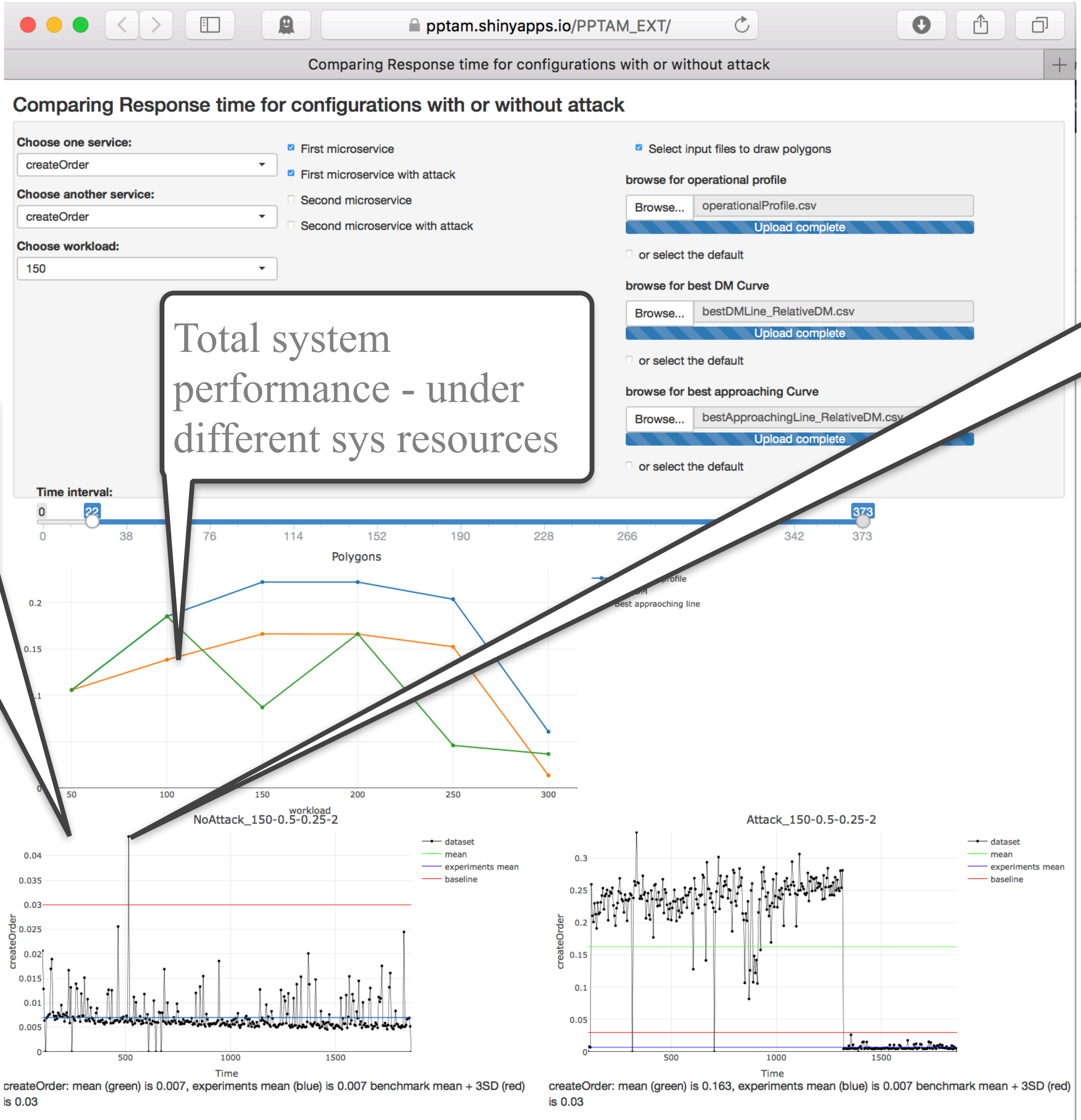
Time interval: 22 – 373

Total system performance - under different sys resources

createOrder: mean (green) is 0.007, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

createOrder: mean (green) is 0.163, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

https://pptam.shinyapps.io/PPTAM_EXT/



Comparing Response time for configurations with or without attack

Comparing Response time for configurations with or without attack

Choose one service: createOrder

Choose another service: createOrder

Choose workload: 150

☑ First microservice

☑ First microservice with attack

☐ Second microservice

☐ Second microservice with attack

☑ Select input files to draw polygons

browse for operational profile

Browse... operationalProfile.csv

Upload complete

☐ or select the default

browse for best DM Curve

Browse... bestDMLine_RelativeDM.csv

Upload complete

☐ or select the default

browse for best approaching Curve

Browse... bestApproachingLine_RelativeDM.csv

Upload complete

☐ or select the default

Time interval: 22 – 373

createOrder: mean (green) is 0.007, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

createOrder: mean (green) is 0.163, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

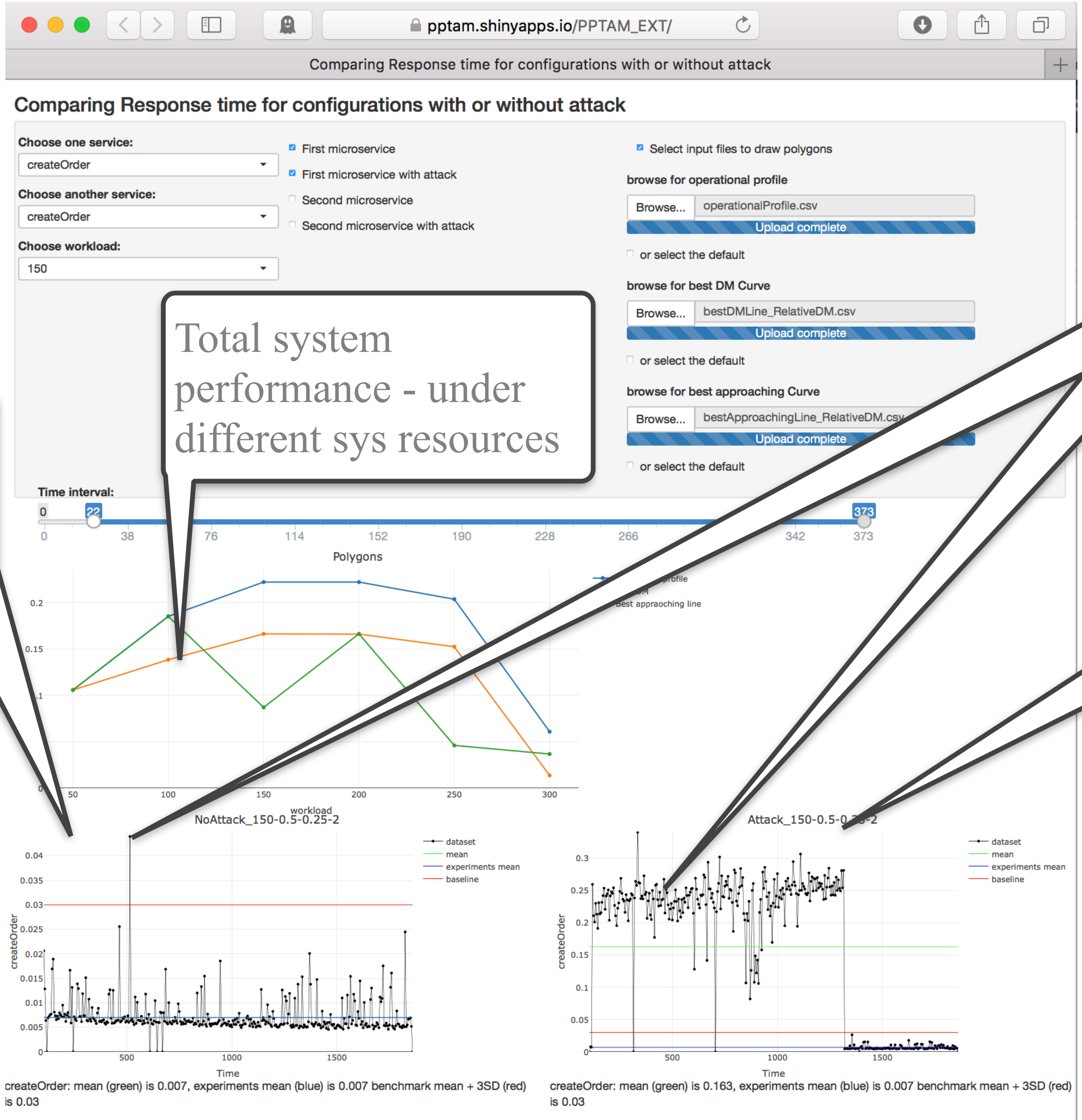
Total system performance - under different sys resources

No attack

Service failure

unibz

https://pptam.shinyapps.io/PPTAM_EXT/



Comparing Response time for configurations with or without attack

Comparing Response time for configurations with or without attack

Choose one service: createOrder

Choose another service: createOrder

Choose workload: 150

☑ First microservice

☑ First microservice with attack

□ Second microservice

□ Second microservice with attack

☑ Select input files to draw polygons

browse for operational profile

Browse... operationalProfile.csv

Upload complete

□ or select the default

browse for best DM Curve

Browse... bestDMLine_RelativeDM.csv

Upload complete

□ or select the default

browse for best approaching Curve

Browse... bestApproachingLine_RelativeDM.csv

Upload complete

□ or select the default

Time interval: 0 – 22 – 373 (0, 38, 76, 114, 152, 190, 228, 266, 342, 373)

createOrder: mean (green) is 0.007, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

createOrder: mean (green) is 0.163, experiments mean (blue) is 0.007 benchmark mean + 3SD (red) is 0.03

Total system performance - under different sys resources

No attack

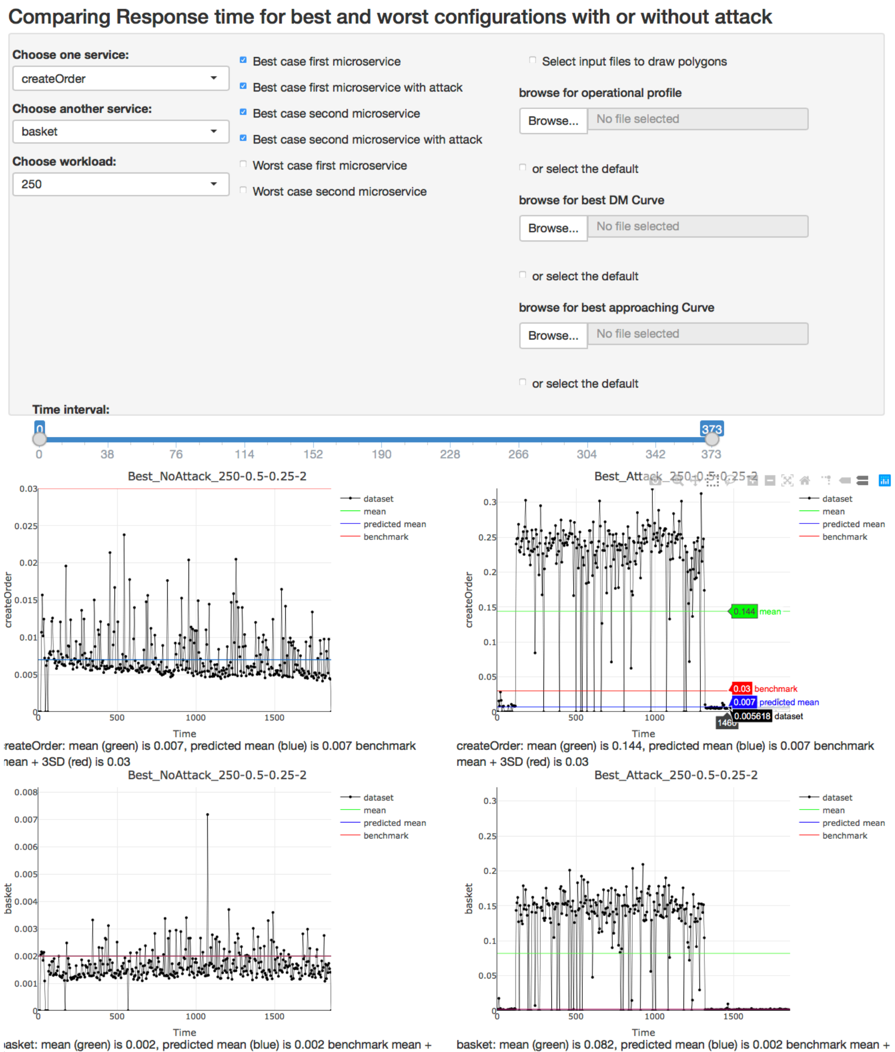
Service failures

Under attack

unibz

Monitoring cockpit

Cockpit



Comparing Response time for best and worst configurations with or without attack

Choose one service:
createOrder

Choose another service:
basket

Choose workload:
250

☑ Best case first microservice
☑ Best case first microservice with attack
☑ Best case second microservice
☑ Best case second microservice with attack
□ Worst case first microservice
□ Worst case second microservice

□ Select input files to draw polygons

browse for operational profile
Browse... No file selected

□ or select the default

browse for best DM Curve
Browse... No file selected

□ or select the default

browse for best approaching Curve
Browse... No file selected

□ or select the default

Time interval:
0 – 373

reateOrder: mean (green) is 0.007, predicted mean (blue) is 0.007 benchmark
nean + 3SD (red) is 0.03

createOrder: mean (green) is 0.144, predicted mean (blue) is 0.007 benchmark
mean + 3SD (red) is 0.03

asket: mean (green) is 0.002, predicted mean (blue) is 0.002 benchmark mean +

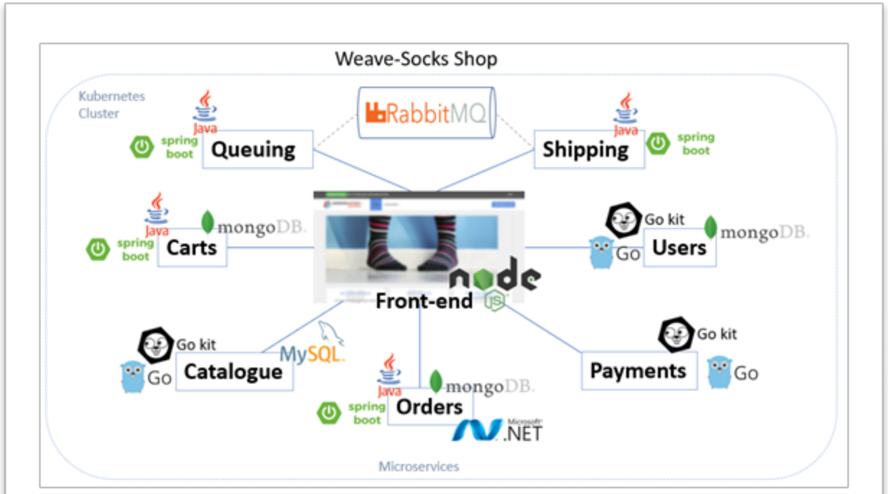
basket: mean (green) is 0.082, predicted mean (blue) is 0.002 benchmark mean +

https://pptam.shinyapps.io/PPTAM_EXT/

Case studies

Microservice architecture

- In a microservice architecture, services are **fine-grained** and the protocols are lightweight rendering each micro service **loosely coupled** with the others
- Microservice architectures often use **containers** to enforce service independence



Before starting a transition …

- There are characteristics that are shared by microservice architectures:
  - Data is organized in a decentralized way: each service manages its own data makes it independently deployable
  - Teams that build systems with microservices extensively use infrastructure automation techniques (like continuous integration or continuous delivery)

Data independently deployable

Automation

Quantitative Assessment of Deployment Alternatives

- Challenge: **assess performance of architectural deployment alternatives** (e.g., number of replicas, CPU/memory allocation, technology stack) under fuzzy requirements

A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing

Alberto Avritzer[1], Vincenzo Ferme[2], Andrea Janes[3], Barbara Russo[3], Henning Schulz[4], and André van Hoorn[2]

[1] EsulabSolutions, Inc., Princeton, NJ, USA
[2] University of Stuttgart, Germany
[3] Free University of Bozen-Bolzano, Italy
[4] NovaTec Consulting GmbH, Leinfelden-Echterdingen, Germany

Abstract. Microservices have emerged as an architectural style for developing distributed applications. Assessing the performance of architectural deployment alternatives is challenging and must be aligned with the system usage in the production environment. In this paper, we introduce an approach for using operational profiles to generate load tests to automatically assess scalability pass/fail criteria of several microservices deployment alternatives, using a baseline computation of performance requirements. We have evaluated our approach with different architecture deployment alternatives using extensive lab studies in a large bare metal host environment and a virtualized environment. We have found that in auto-scaling cloud environments careful performance engineering activities shall be executed before additional resources are added to the architecture deployment configuration.

1 Introduction

The microservices architectural style [9] is an approach for creating software applications as a collection of loosely coupled software components. These components are called microservices, and are supposed to be autonomous, automatically and independently deployable, and cohesive [11]. This architecture lends itself to decentralized deployment, and for continuous integration and deployment by developers. Several large companies (e.g., Amazon and Netflix) are reporting significant success with microservice architectures [7,9].

Currently, several architecture deployment alternatives are being used for microservices deployment, as for example, serverless microservices using lambdas, container-based deployment (e.g., Docker[5]), virtual machines per host, and several hosts. Of course, depending on the microservice granularity, a combination of these deployment mechanism could be used.

Microservices are supposed to be independent of each other. However, the underlying deployment environment might introduce coupling and impact the

[5] https://www.docker.com/

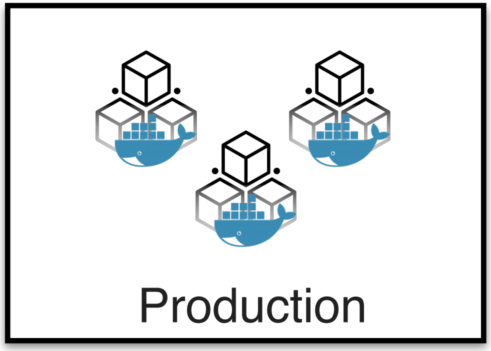
Alberto Avritzer, Vincenzo Ferme, Andrea Janes, Barbara Russo, Henning Schulz, and André van Hoorn: *A Quantitative Approach for the Assessment of Microservice Architecture Deployment Alternatives by Automated Performance Testing*.

In Proceedings of the 12th European Conference on Software Architecture (ECSA). LNCS, Springer, 2018 (Accepted)

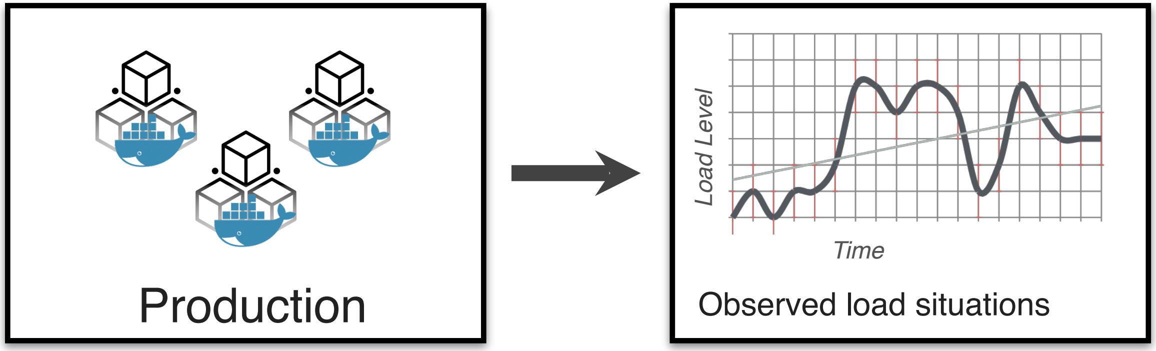
Quantitative Assessment of Deployment Alternatives

- Approach
  - Use operational data to generate and weigh load tests
  - Measure baseline requirements
  - Design a metric that allows quantitative comparison of deployment alternatives

Overview of Approach



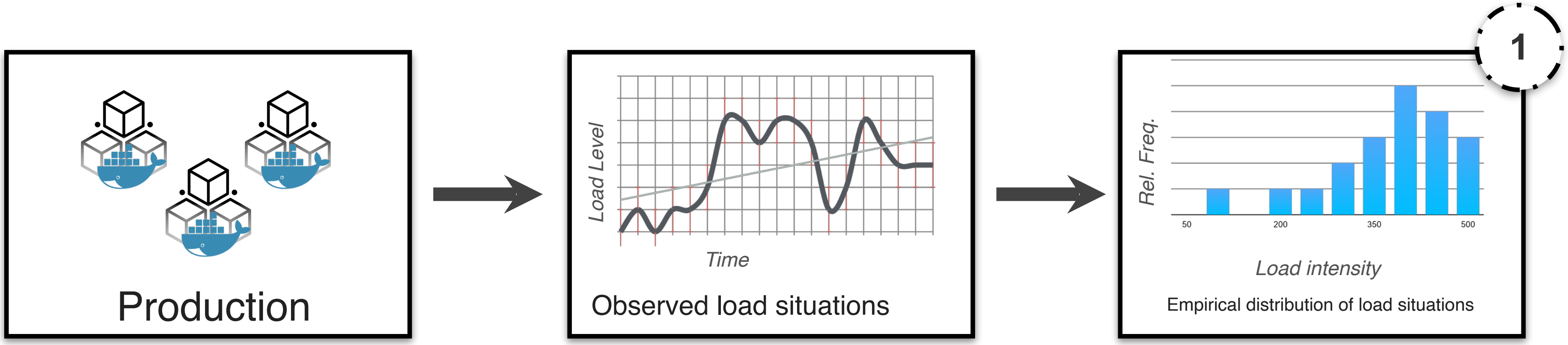
Overview of Approach



Production

Observed load situations

Overview of Approach

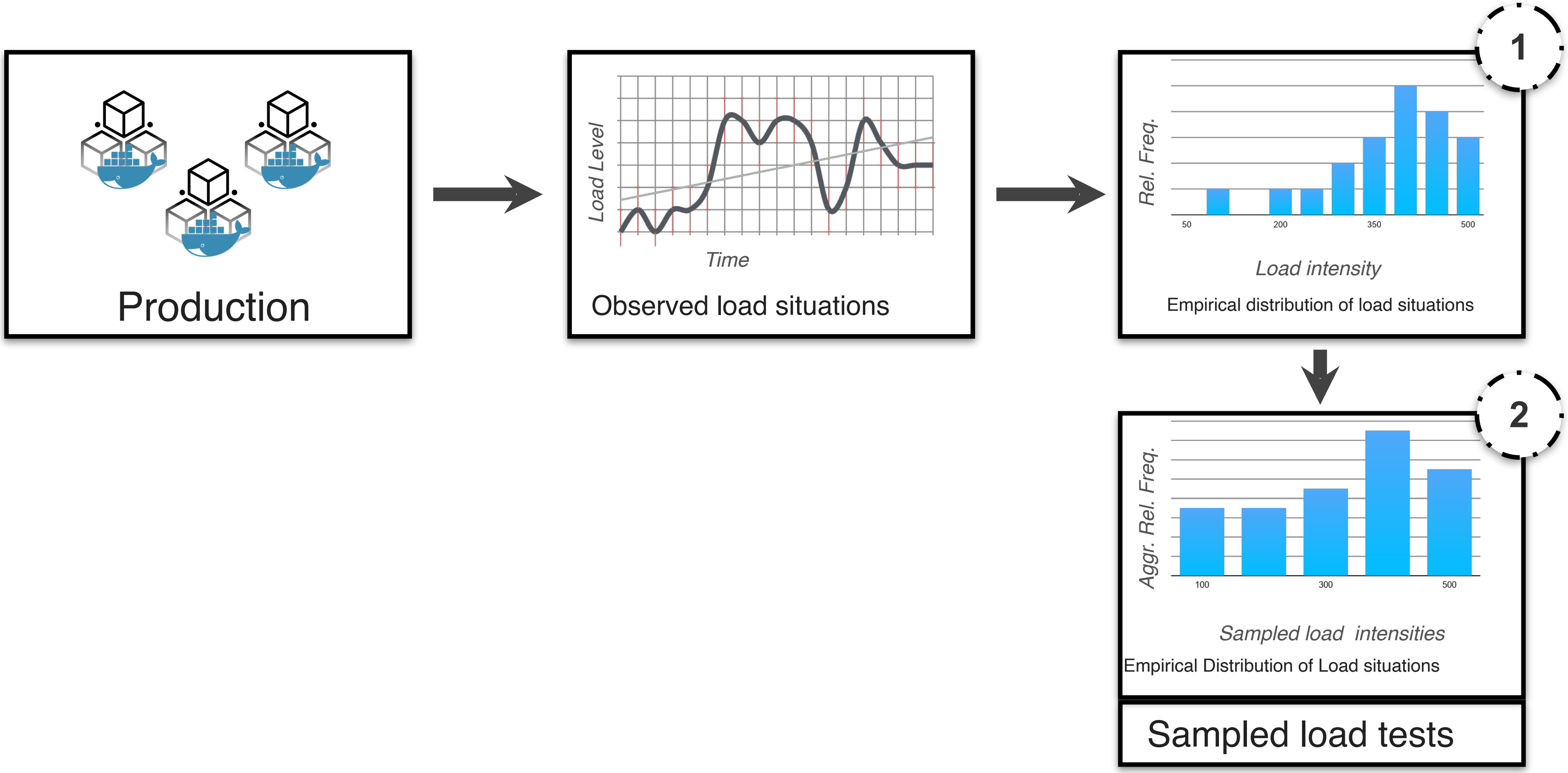


Production

Observed load situations

Empirical distribution of load situations

Overview of Approach



Production

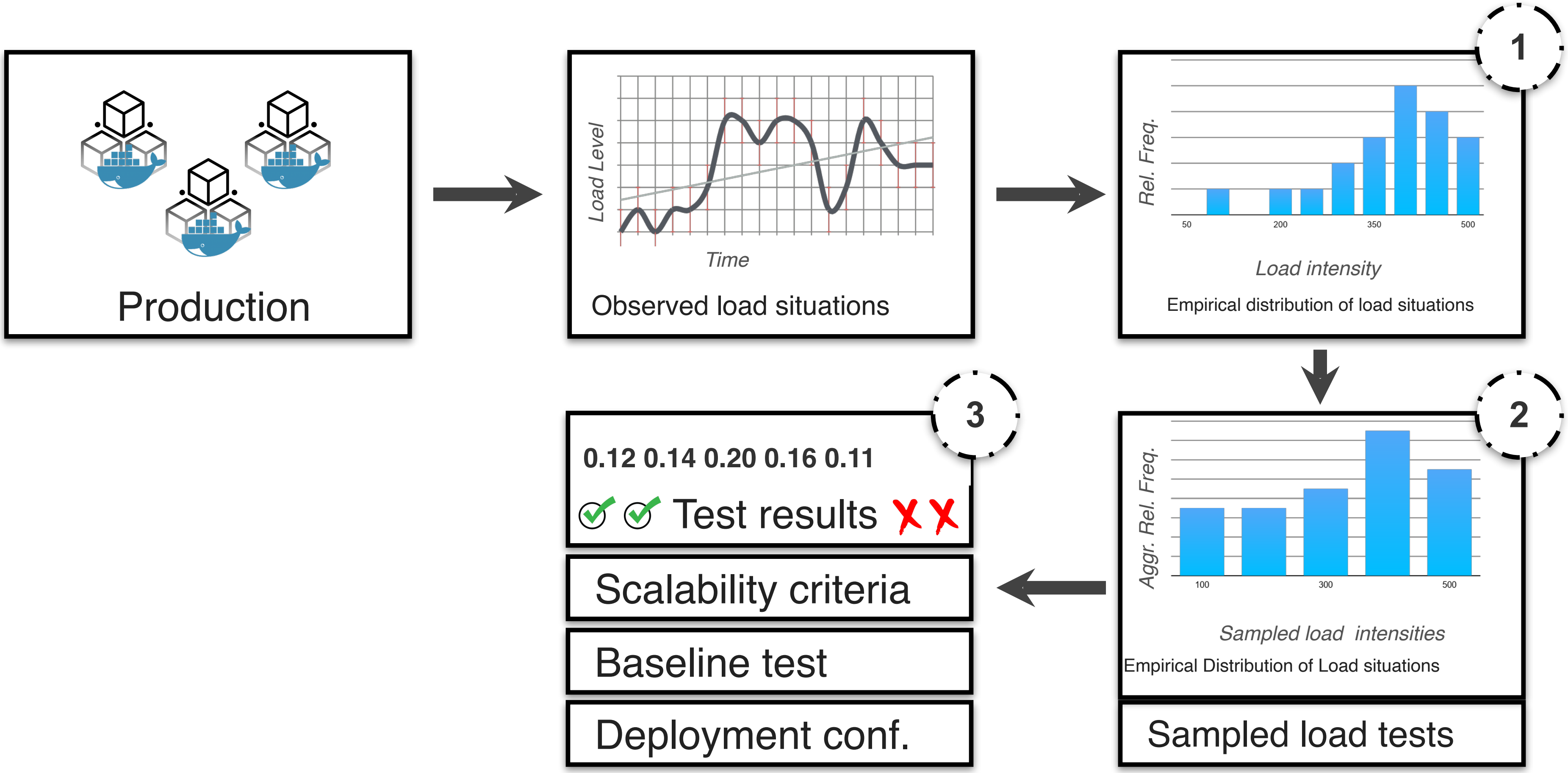
Observed load situations

Empirical distribution of load situations

Empirical Distribution of Load situations

Sampled load tests

Overview of Approach



Production

Observed load situations

Empirical distribution of load situations

Empirical Distribution of Load situations

Sampled load tests

0.12 0.14 0.20 0.16 0.11

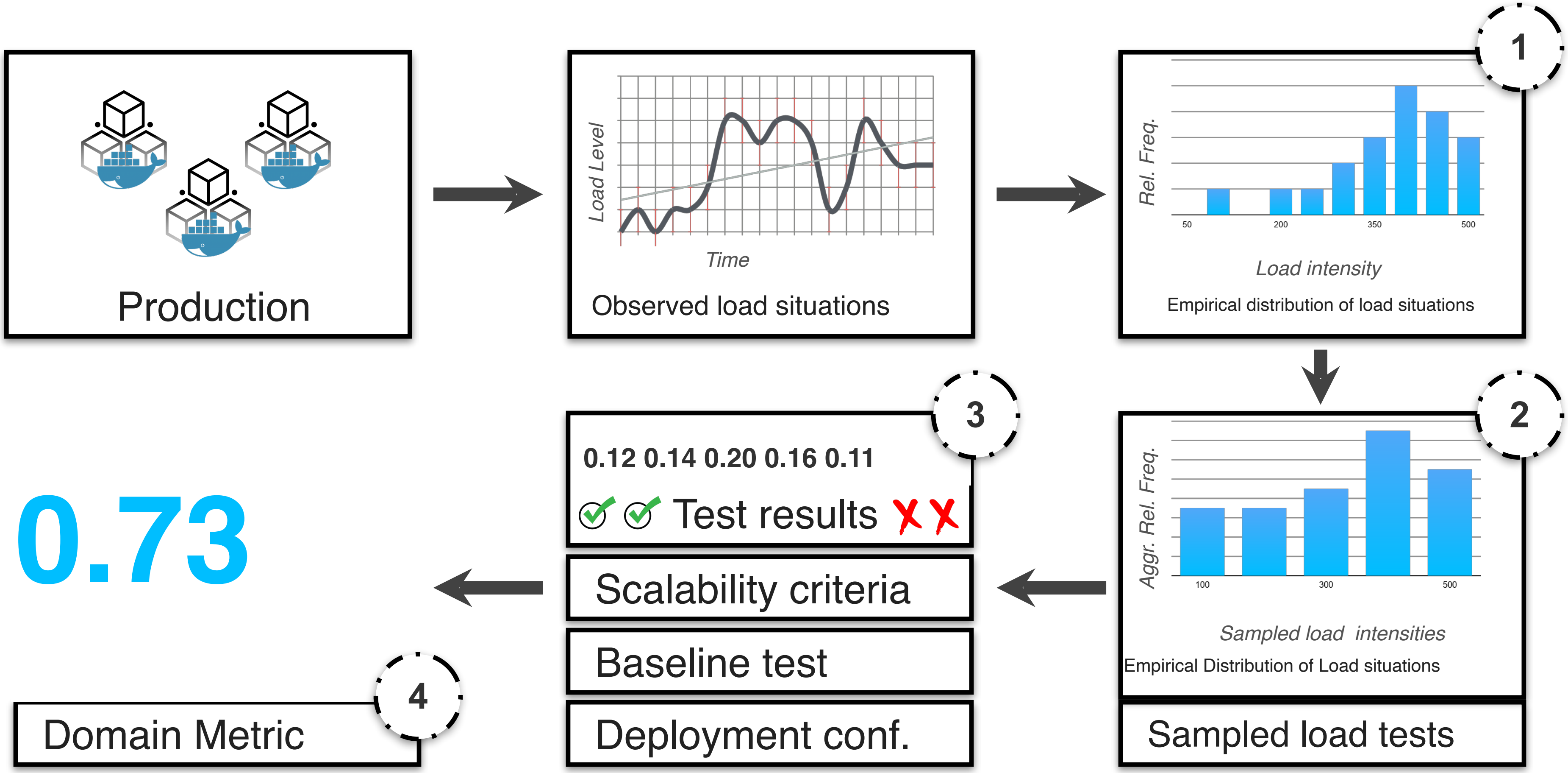
✓ ✓ Test results ✗ ✗

Scalability criteria

Baseline test

Deployment conf.

Overview of Approach



Production

Observed load situations

1

Empirical distribution of load situations

2

Empirical Distribution of Load situations

Sampled load tests

3

0.12 0.14 0.20 0.16 0.11

✓ ✓ Test results ✗ ✗

Scalability criteria

Baseline test

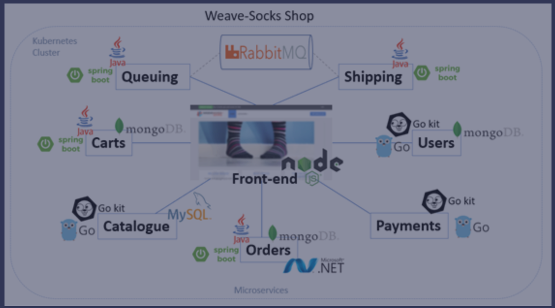
Deployment conf.

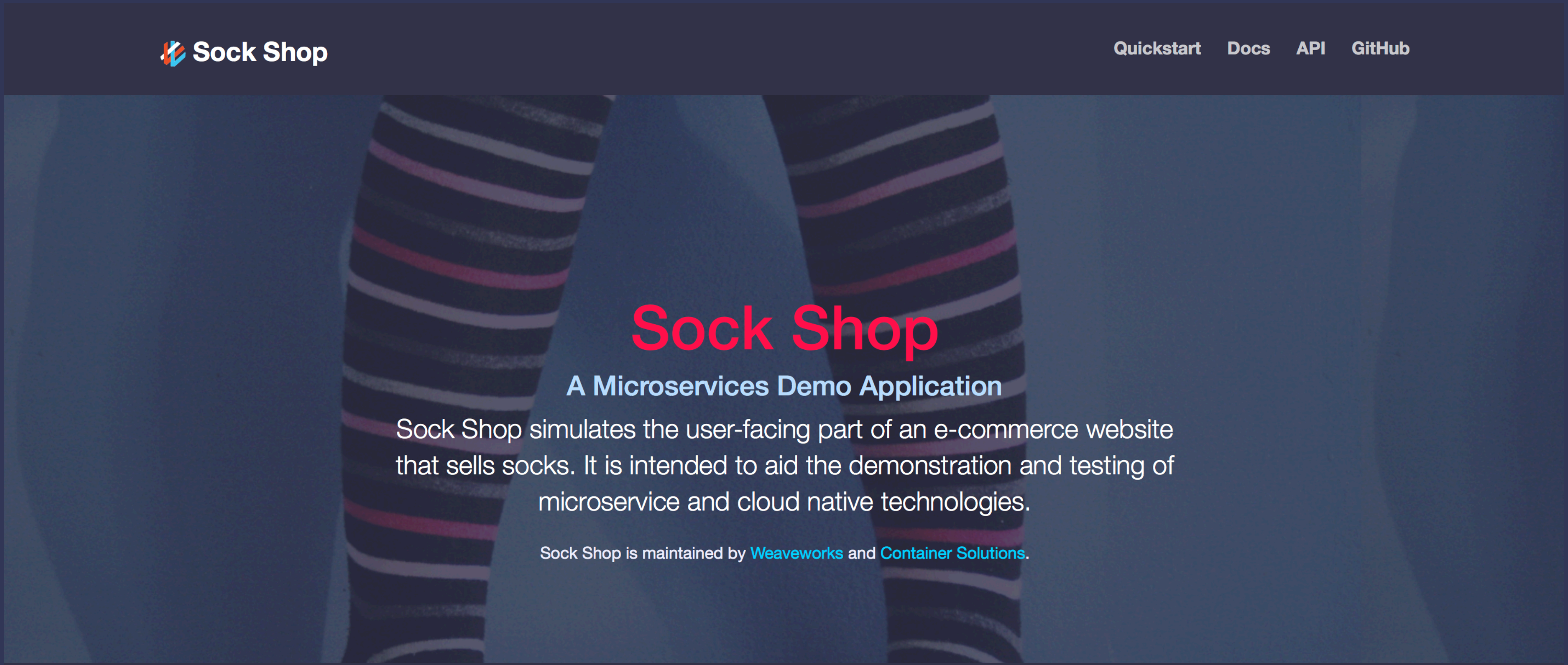
0.73

4

Domain Metric

System Under Test





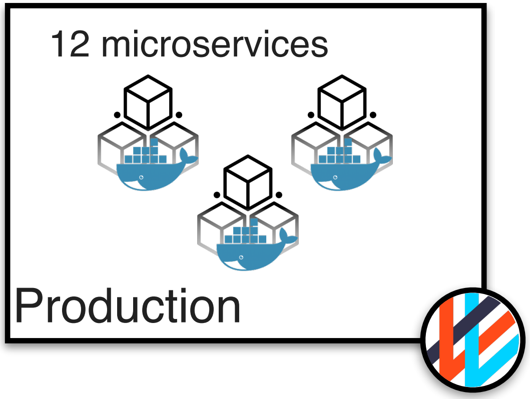
Sock Shop

A Microservices Demo Application

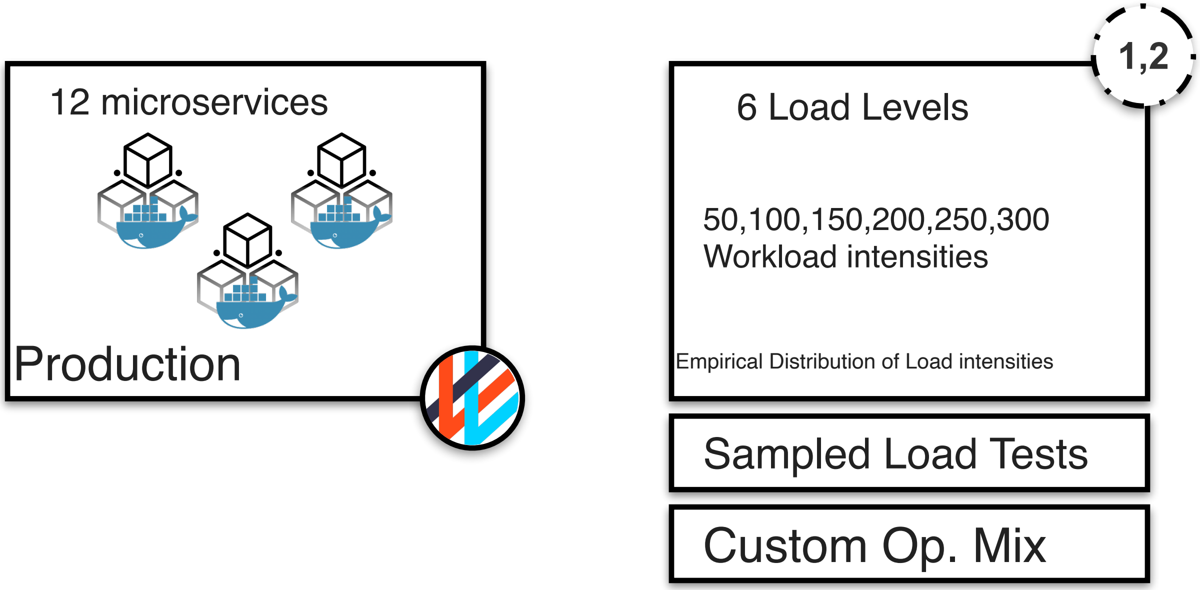
Sock Shop simulates the user-facing part of an e-commerce website that sells socks. It is intended to aid the demonstration and testing of microservice and cloud native technologies.

Sock Shop is maintained by Weaveworks and Container Solutions.

Experiments



Experiments



12 microservices

Production

1,2

6 Load Levels

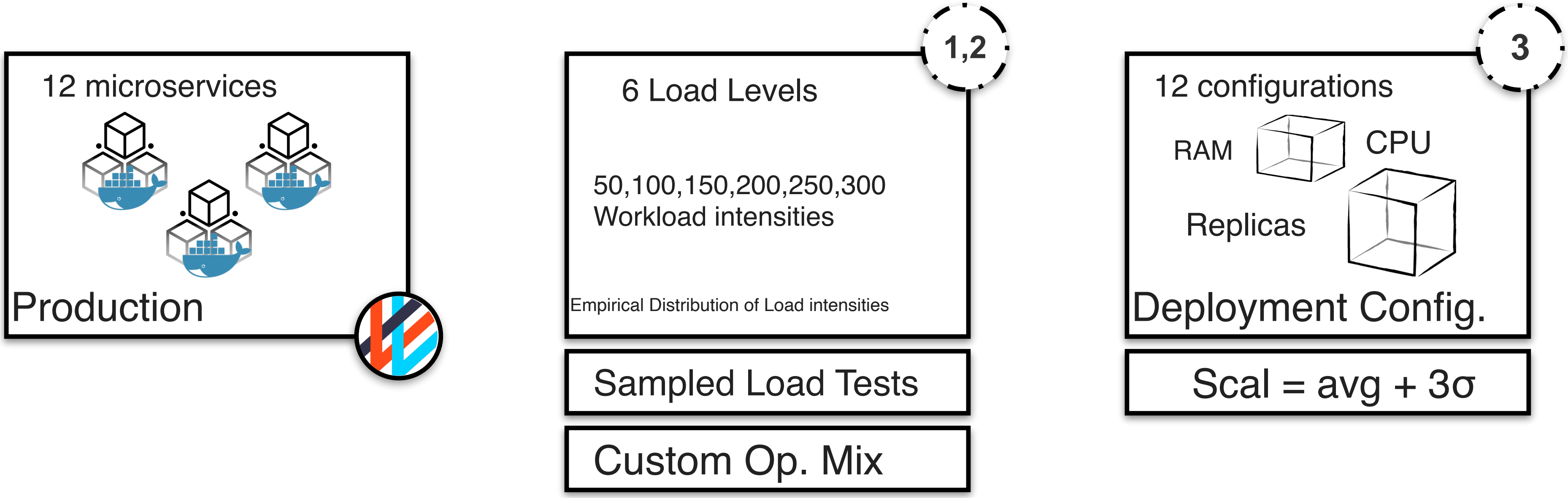
50,100,150,200,250,300
Workload intensities

Empirical Distribution of Load intensities

Sampled Load Tests

Custom Op. Mix

Experiments



12 microservices

Production

1,2

6 Load Levels

50,100,150,200,250,300
Workload intensities

Empirical Distribution of Load intensities

Sampled Load Tests

Custom Op. Mix

3

12 configurations

RAM

CPU

Replicas

Deployment Config.

Scal = avg + 3σ

Experiment Results: Computation of Domain Metric (1/2)

API	Scalability Criteria
GET /	**PASS**
GET /cart	**PASS**
POST /item	**FAIL**

Users	Aggr. Rel. Freq.
50	0.10582
100	0.18519
150	0.22222
200	0.22222
250	0.20370
300	0.06085

Custom Op. Mix

Aggr. Rel. Freq.

Contrib. to Domain Metric

Deployment Configuration: 1 GB RAM, 0.25 CPU, 1 Replica

Experiment Results: Computation of Domain Metric (1/2)

API	Scalability Criteria
GET /	**PASS**
GET /cart	**PASS**
POST /item	**FAIL**

Custom Op. Mix

Users	Aggr. Rel. Freq.
50	0.10582
100	0.18519
150	0.22222
200	0.22222
250	0.20370
300	0.06085

Aggr. Rel. Freq.

Max: 0.20370

Actual: 0.13580

Contrib. to Domain Metric

Deployment Configuration: 1 GB RAM, 0.25 CPU, 1 Replica

Experiment Results: Computation of Domain Metric (2/2)

Users	Contribution
50	0.10582
100	0.18519
150	0.22222
200	0.07999
250	0.13580
300	0.04729

Contrib. to Domain Metric

Deployment Configuration: 1 GB RAM, 0.25 CPU, 1 Replica

Experiment Results: Computation of Domain Metric (2/2)

Users	Contribution
50	0.10582
100	0.18519
150	0.22222
200	0.07999
250	0.13580
300	0.04729

Max: 1

Contrib. to Domain Metric

Domain Metric

4

Deployment Configuration: 1 GB RAM, 0.25 CPU, 1 Replica

Experiment Results: Computation of Domain Metric (2/2)

Users	Contribution
50	0.10582
100	0.18519
150	0.22222
200	0.07999
250	0.13580
300	0.04729

Contrib. to Domain Metric

Max: 1

Actual:

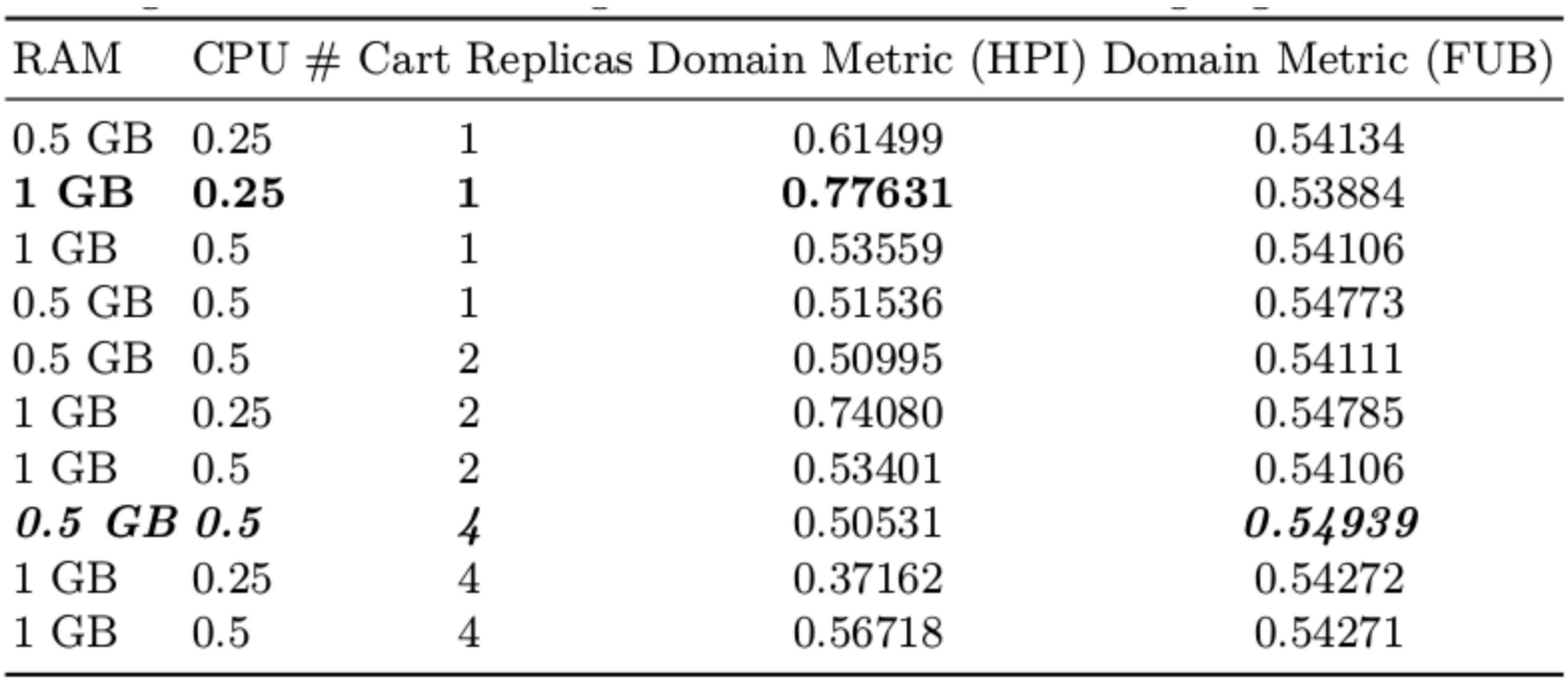
0.77631

4

Domain Metric

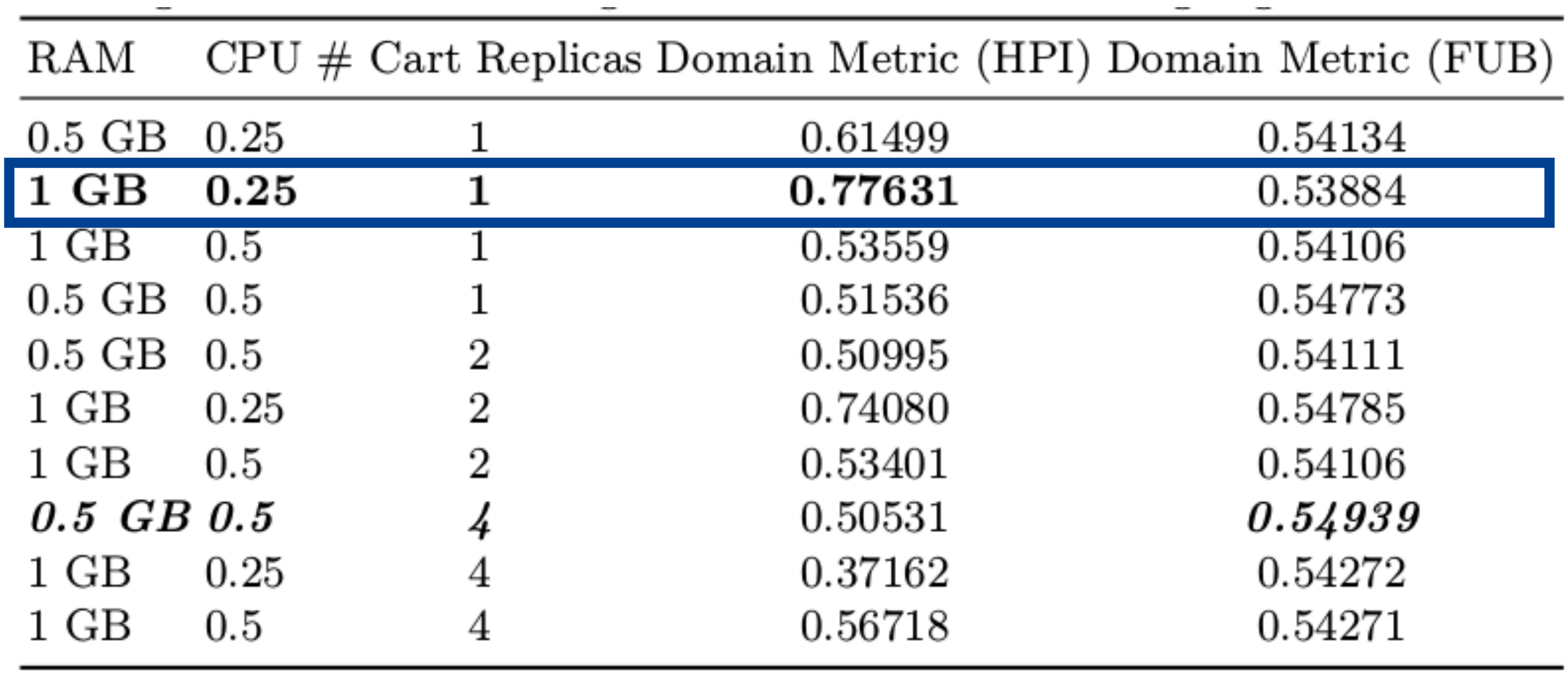
Deployment Configuration: 1 GB RAM, 0.25 CPU, 1 Replica

Experiment Results: Single-Metric Comparison of Alternatives



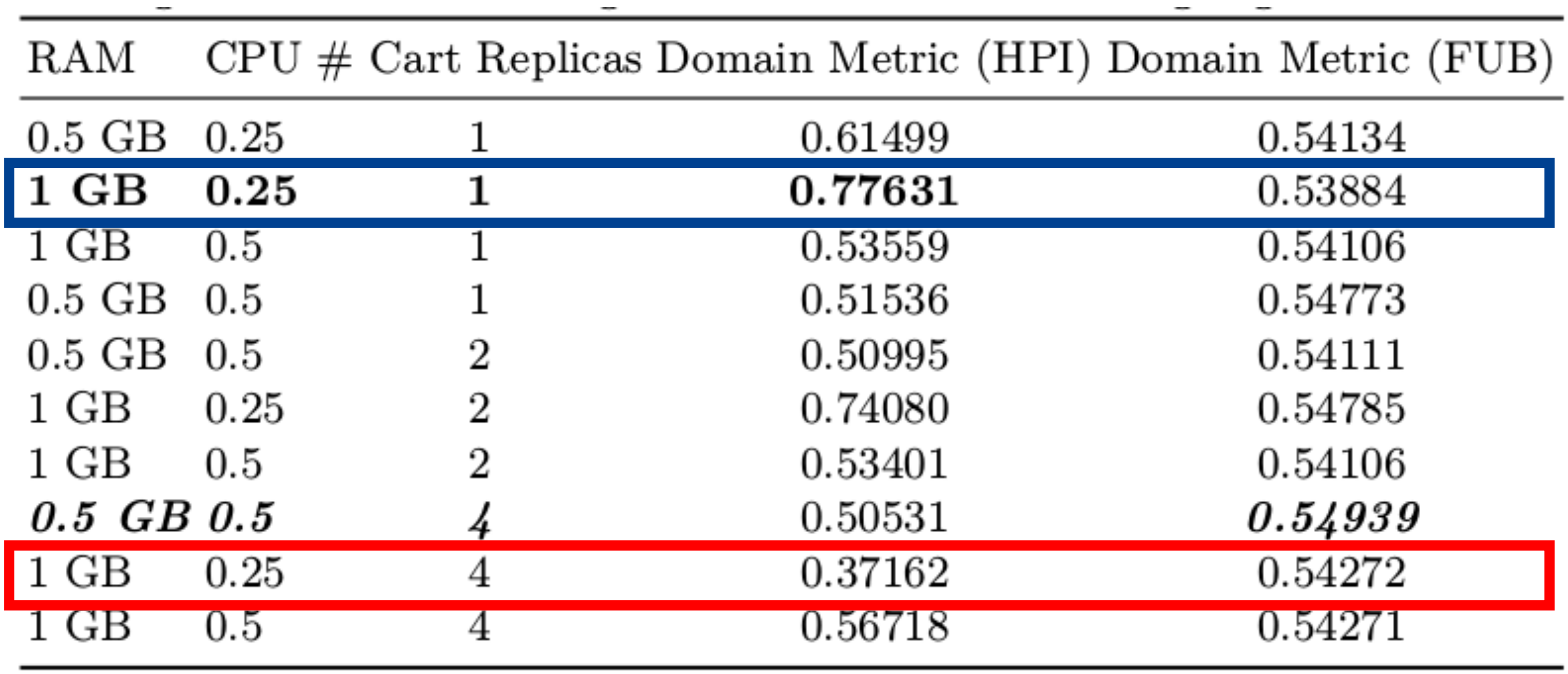
RAM	CPU	# Cart Replicas	Domain Metric (HPI)	Domain Metric (FUB)
0.5 GB	0.25	1	0.61499	0.54134
1 GB	**0.25**	**1**	**0.77631**	0.53884
1 GB	0.5	1	0.53559	0.54106
0.5 GB	0.5	1	0.51536	0.54773
0.5 GB	0.5	2	0.50995	0.54111
1 GB	0.25	2	0.74080	0.54785
1 GB	0.5	2	0.53401	0.54106
0.5 GB	***0.5***	*4*	0.50531	***0.54939***
1 GB	0.25	4	0.37162	0.54272
1 GB	0.5	4	0.56718	0.54271

Experiment Results: Single-Metric Comparison of Alternatives



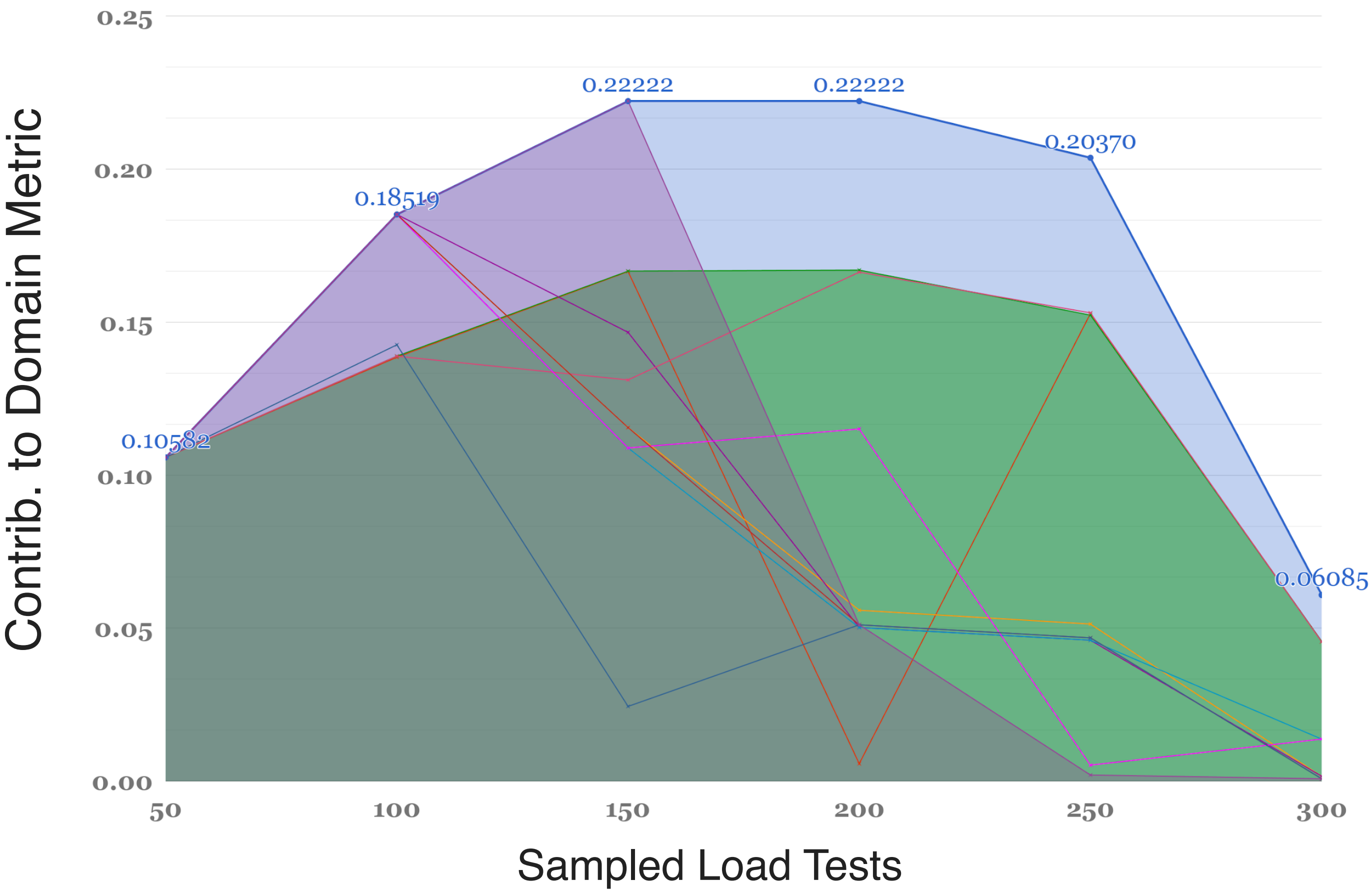
RAM	CPU	# Cart Replicas	Domain Metric (HPI)	Domain Metric (FUB)
0.5 GB	0.25	1	0.61499	0.54134
1 GB	**0.25**	**1**	**0.77631**	0.53884
1 GB	0.5	1	0.53559	0.54106
0.5 GB	0.5	1	0.51536	0.54773
0.5 GB	0.5	2	0.50995	0.54111
1 GB	0.25	2	0.74080	0.54785
1 GB	0.5	2	0.53401	0.54106
0.5 GB	***0.5***	***4***	0.50531	***0.54939***
1 GB	0.25	4	0.37162	0.54272
1 GB	0.5	4	0.56718	0.54271

Experiment Results: Single-Metric Comparison of Alternatives



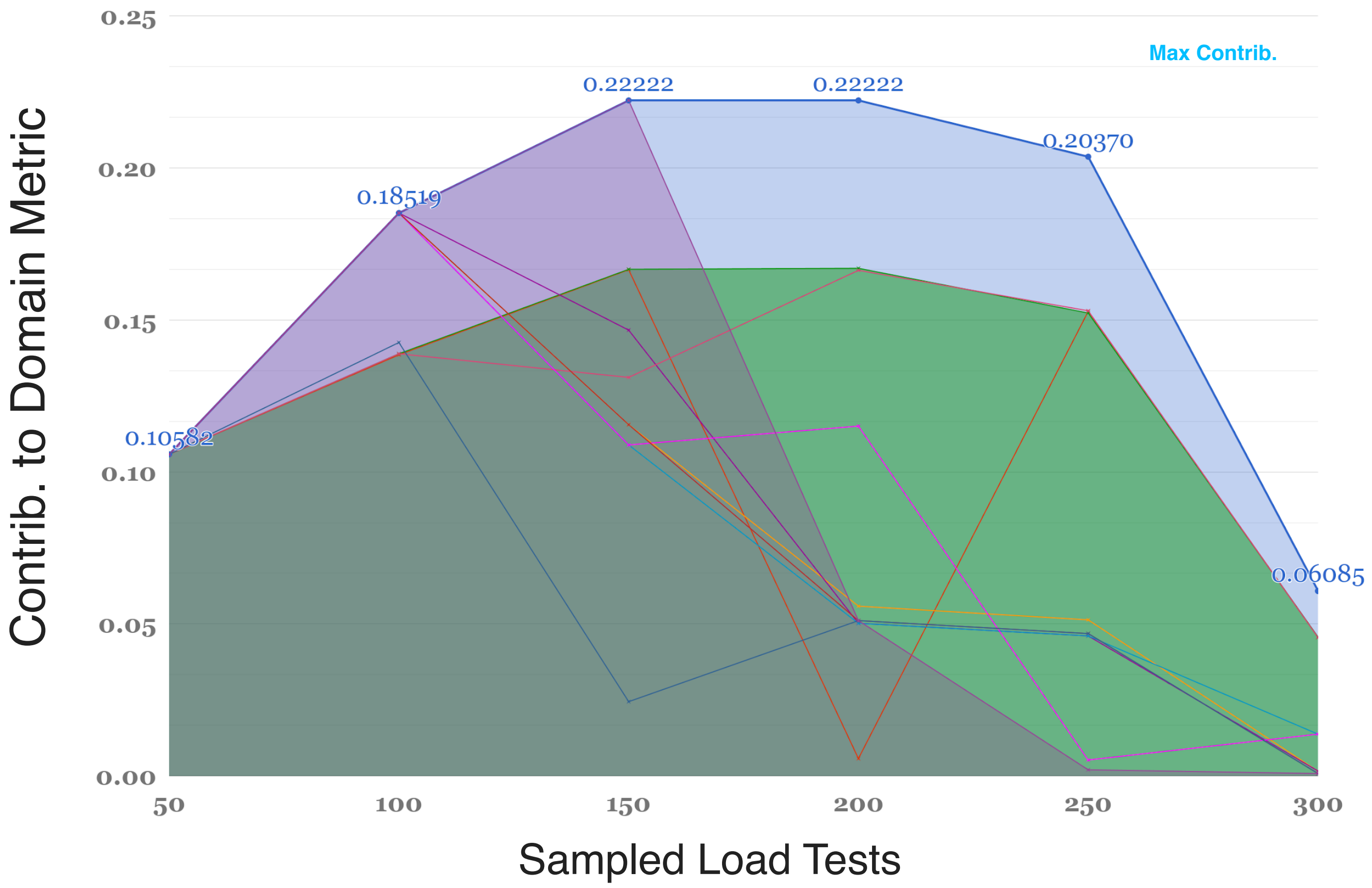
RAM	CPU	# Cart Replicas	Domain Metric (HPI)	Domain Metric (FUB)
0.5 GB	0.25	1	0.61499	0.54134
1 GB	**0.25**	**1**	**0.77631**	0.53884
1 GB	0.5	1	0.53559	0.54106
0.5 GB	0.5	1	0.51536	0.54773
0.5 GB	0.5	2	0.50995	0.54111
1 GB	0.25	2	0.74080	0.54785
1 GB	0.5	2	0.53401	0.54106
0.5 GB	***0.5***	***4***	0.50531	***0.54939***
1 GB	0.25	4	0.37162	0.54272
1 GB	0.5	4	0.56718	0.54271

Experiment Results: Visual Comparison of Alternatives

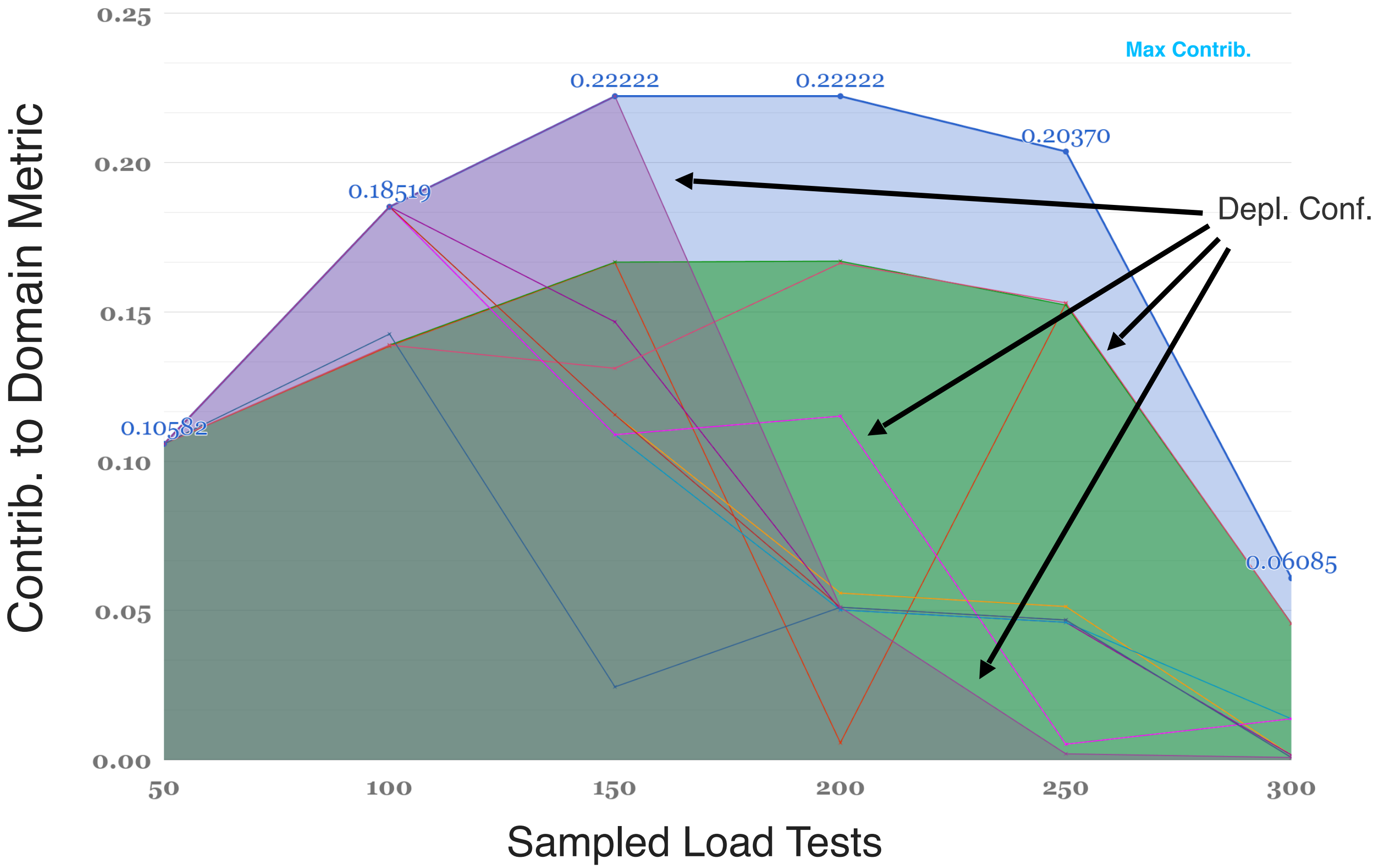


Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Experiment Results: Visual Comparison of Alternatives



Experiment Results: Visual Comparison of Alternatives



Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Extensions/Application

Extensions/Application

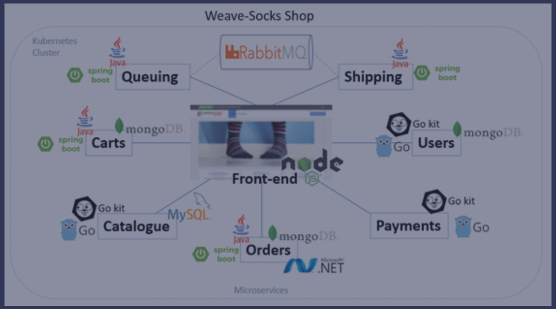
- We have tested it on an online demo-platform
- We have extended it to monitor performce degradation under attacks by incorporating Mirai

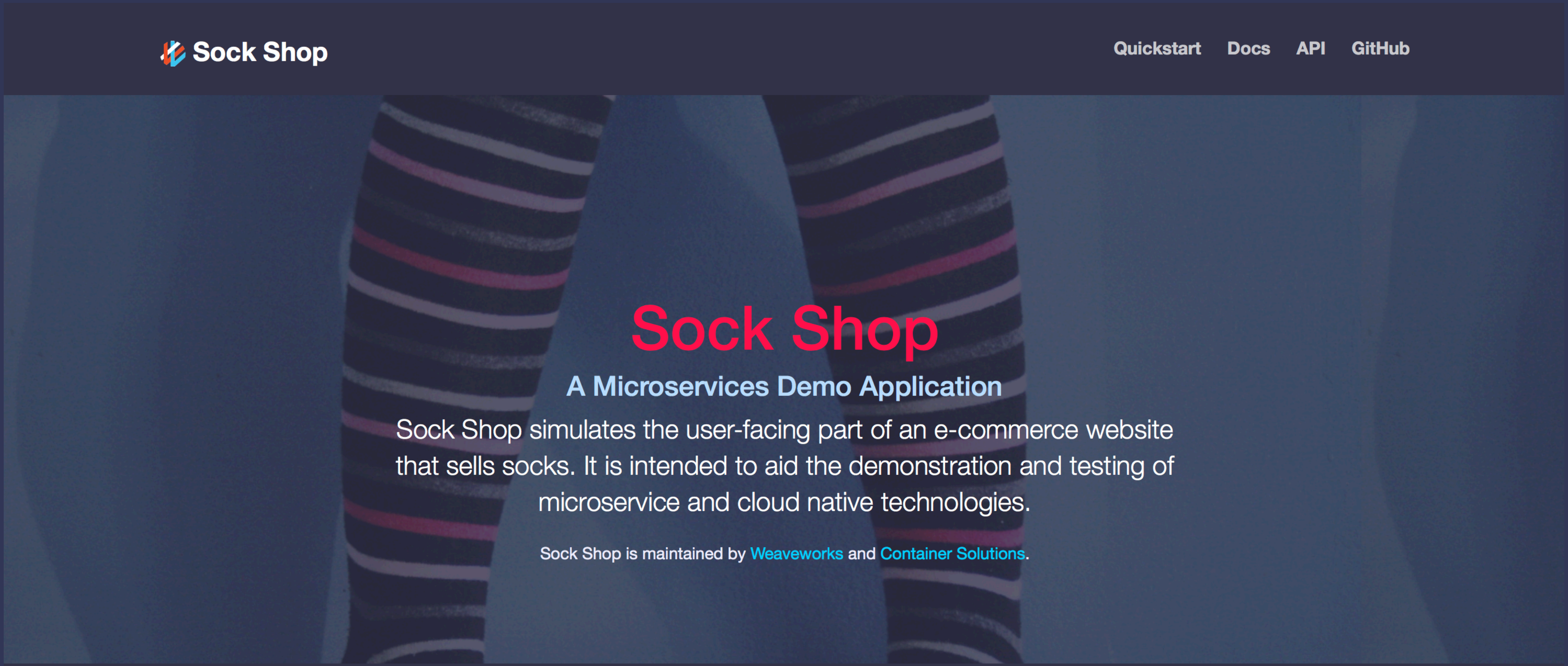
Extensions/Application

- We have tested it on an online demo-platform
- We have extended it to monitor performce degradation under attacks by incorporating Mirai
- We have designed it for monitoring performance degradation during a transition to microservices

Bare-metal versus virtualization environment

System Under Test





Sock Shop

A Microservices Demo Application

Sock Shop simulates the user-facing part of an e-commerce website that sells socks. It is intended to aid the demonstration and testing of microservice and cloud native technologies.

Sock Shop is maintained by Weaveworks and Container Solutions.

Experiment settings

- 2 VM one for SUT and one for Test
- SUT: docker containers each for on micro service, one for DB

Bare-metal

- The containerized bare metal machines:
- Load driver server - **32 GB RAM, 24 cores** (2 threads each) at 2300 MHz and SUT server - **896 GB RAM, 80 cores** (2 threads each) at 2300 MHz
- Both machines use magnetic disks with 15 000 rpm and are connected using a shared 10 Gbit/ s network infrastructure

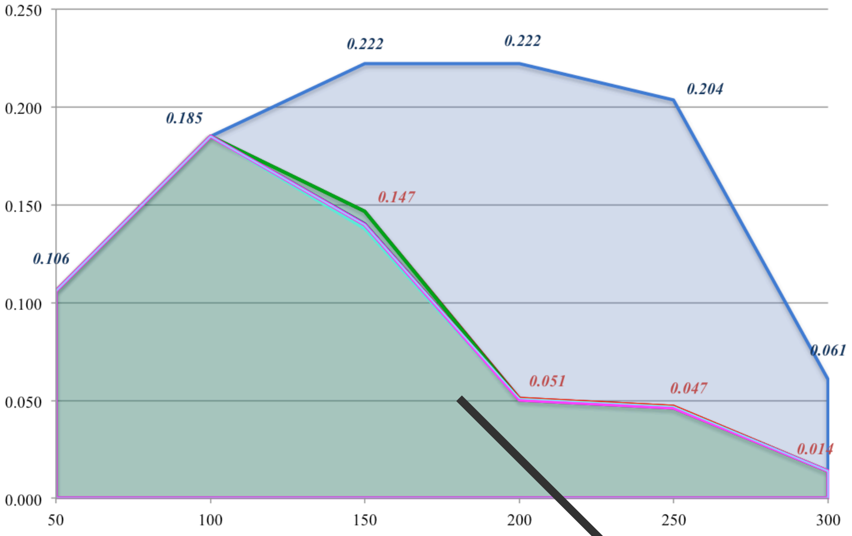
Virtual

- The containerized deployment in virtual machines:
- *Load driver server* - **4 GB RAM, 1 core** at 2600MHz and *SUT server* - **8 GB RAM, 4 cores** at 2600 MHz with SSDs
- Both machines use an EMCVNC 5400 series network attached storage solution12 and are connected using a shared 10 Gbit/ s network infrastructure

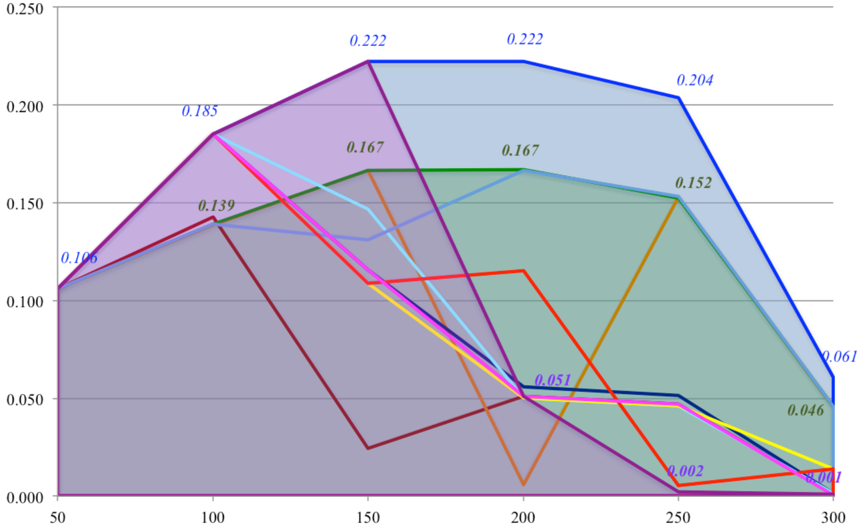
- We replicated with *SUT server* - **16 GB RAM, 8 cores**

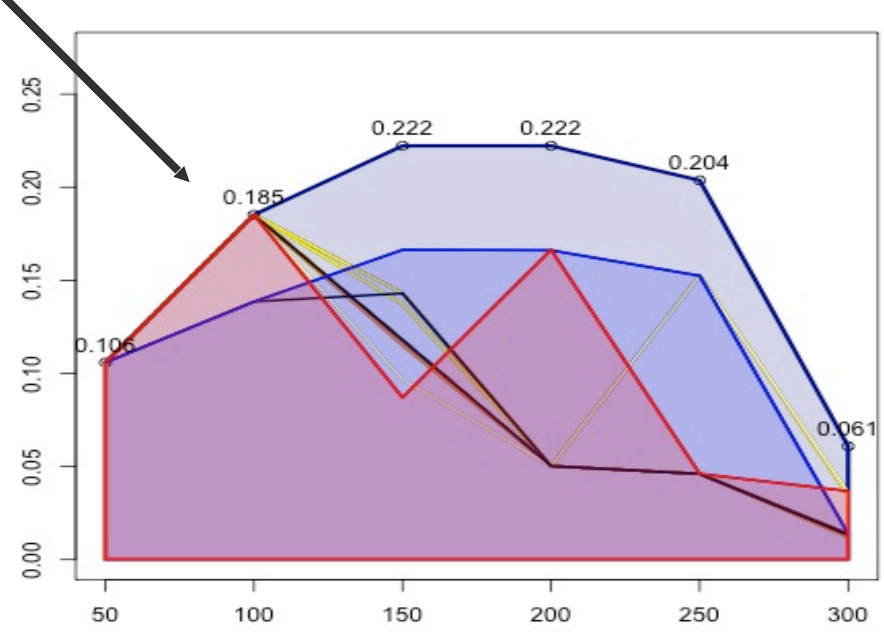
Experiments - results

Virtual



Bare metal

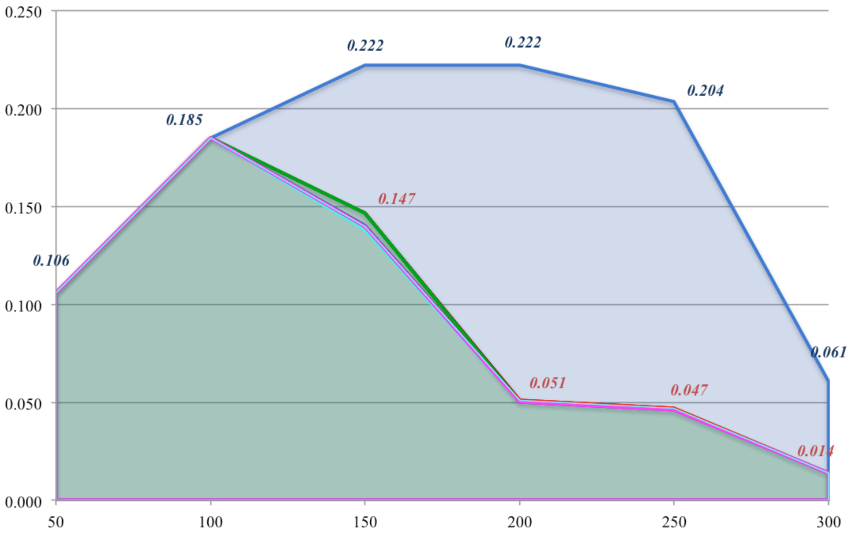




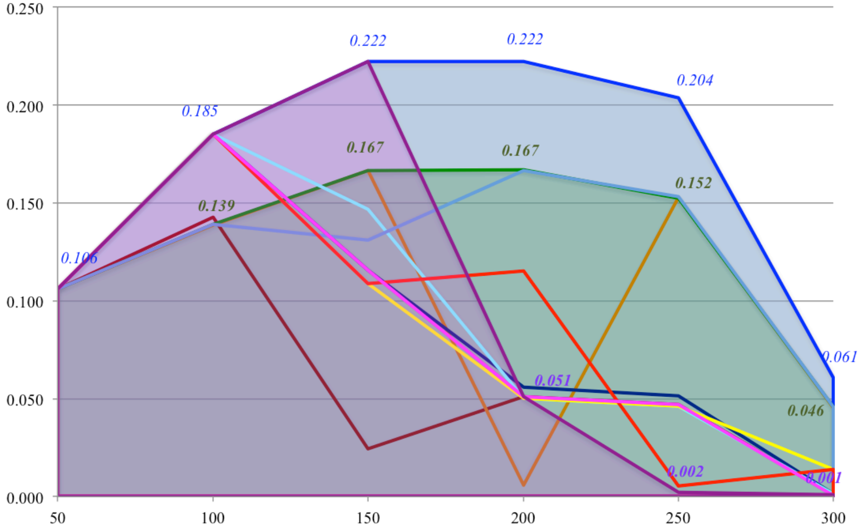
Virtual more resources for SUT

Experiments - results

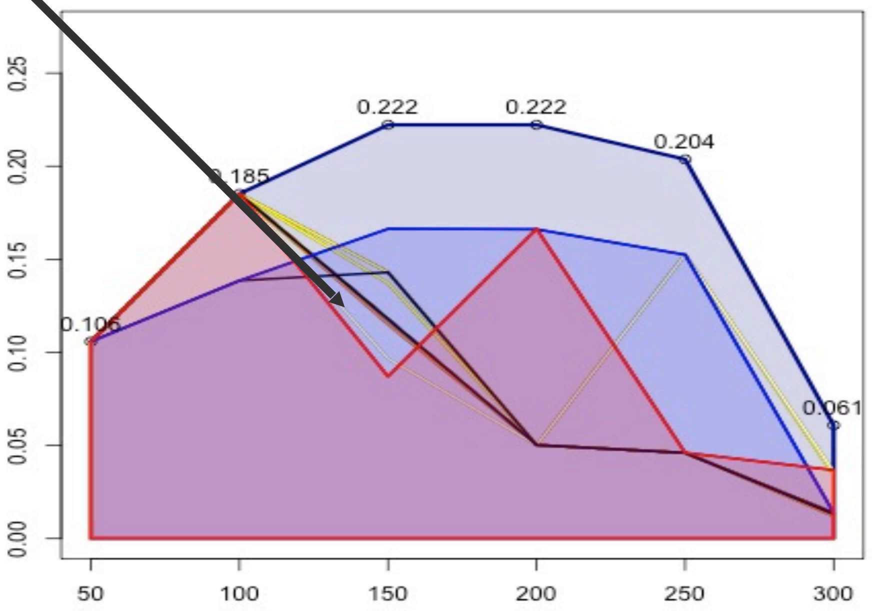
Virtual



Bare metal



Virtual more resources for SUT

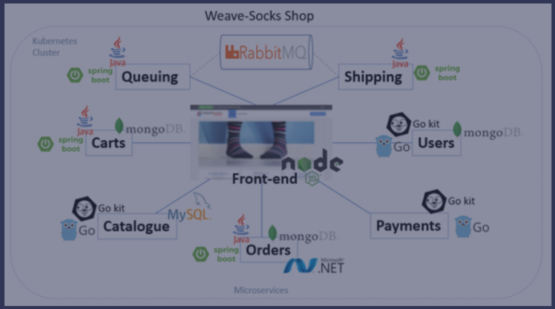


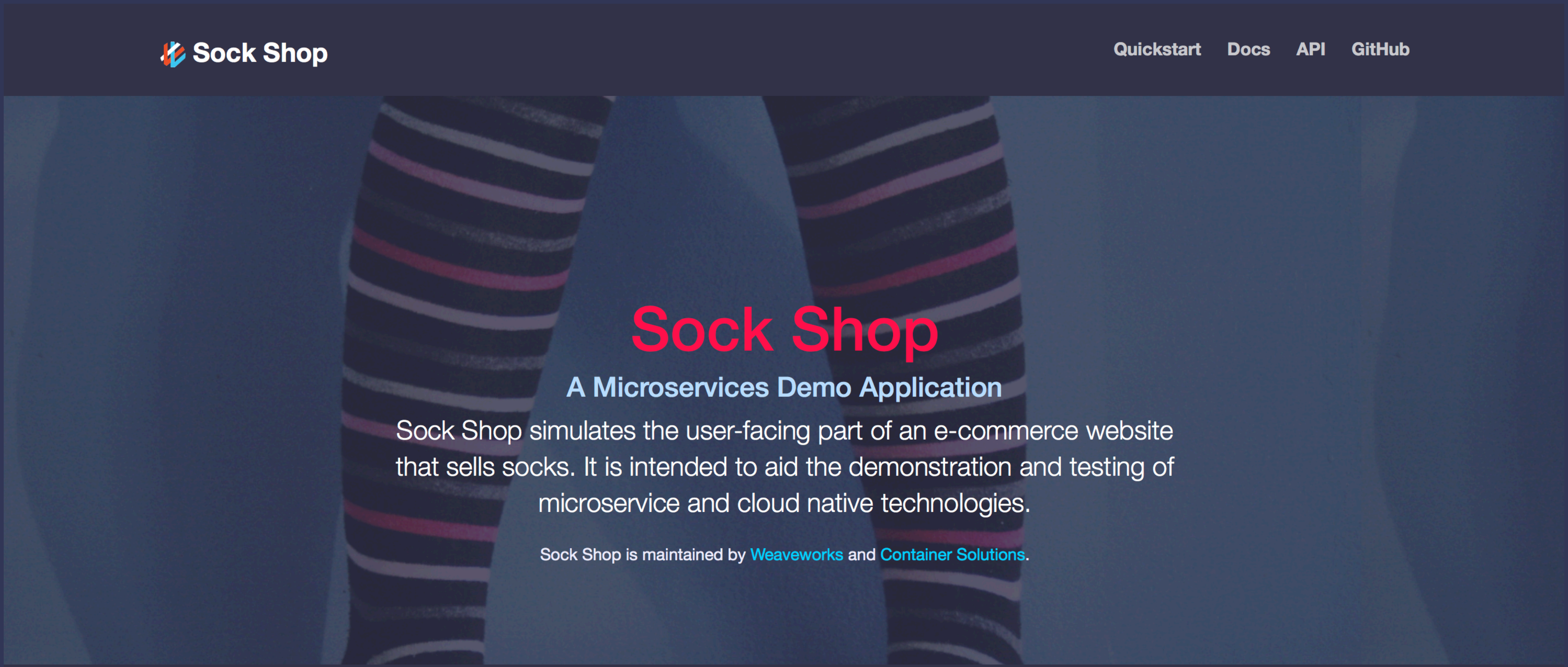
Bare-metal versus virtualization environment

Bare-metal versus virtualization environment

Monitor performance degradation under attacks

System Under Test





Sock Shop

A Microservices Demo Application

Sock Shop simulates the user-facing part of an e-commerce website that sells socks. It is intended to aid the demonstration and testing of microservice and cloud native technologies.

Sock Shop is maintained by Weaveworks and Container Solutions.

Mirai BotNet

- Mirai is a **malware** that has been used to turn networked devices (cameras) running Linux into remotely controlled bots
- We use an academic version of it to attack the system in controlled experiments
- It can perform different types of attack. By now, we have explored http, syn, ack

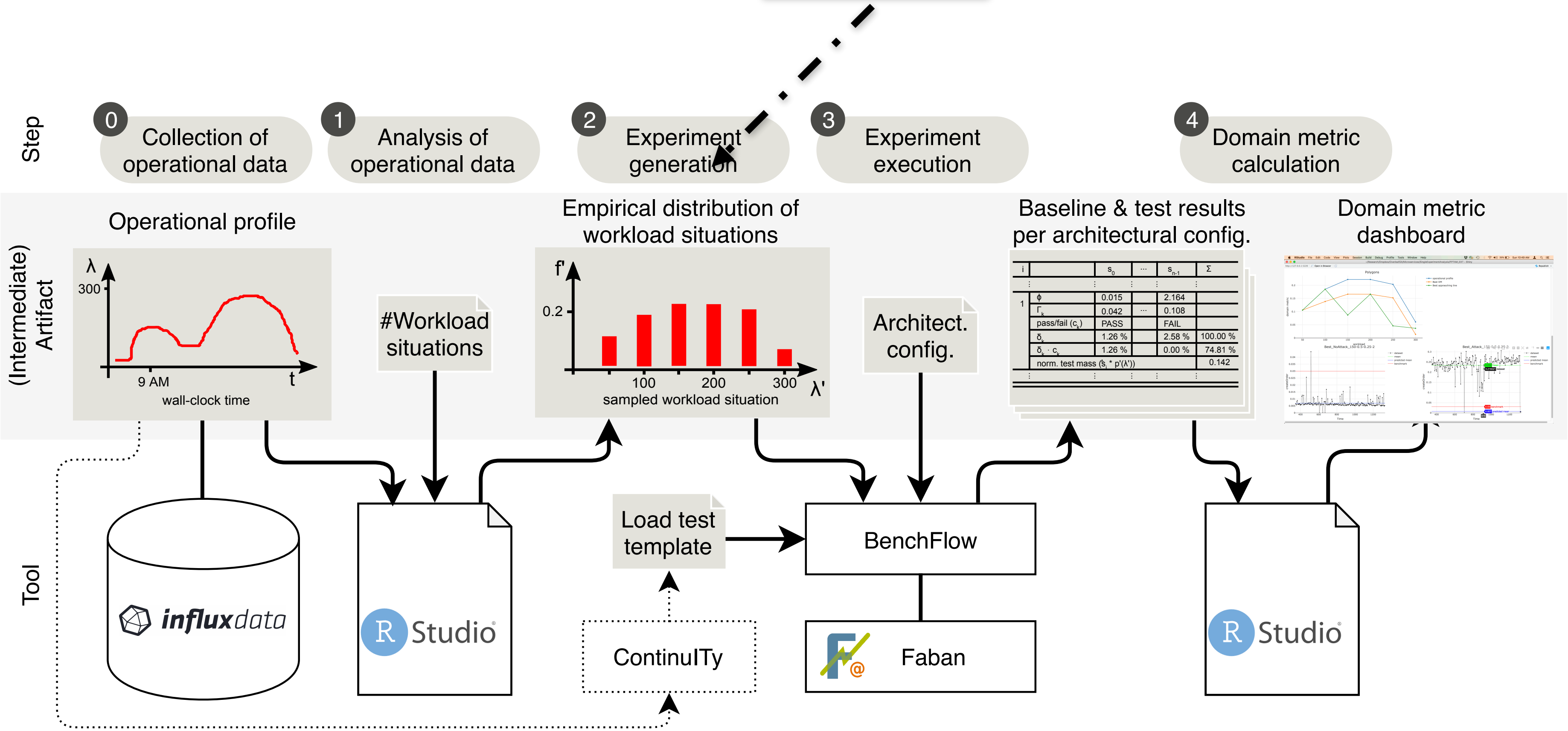
Experiments with Mirai

- Attack with simple http requests (GET and POST to home - increase the load)

- Compute the metric with and without attack to understand:
  - the resilience of a system
  - the early prediction of an attack

PPTAM

Mirai

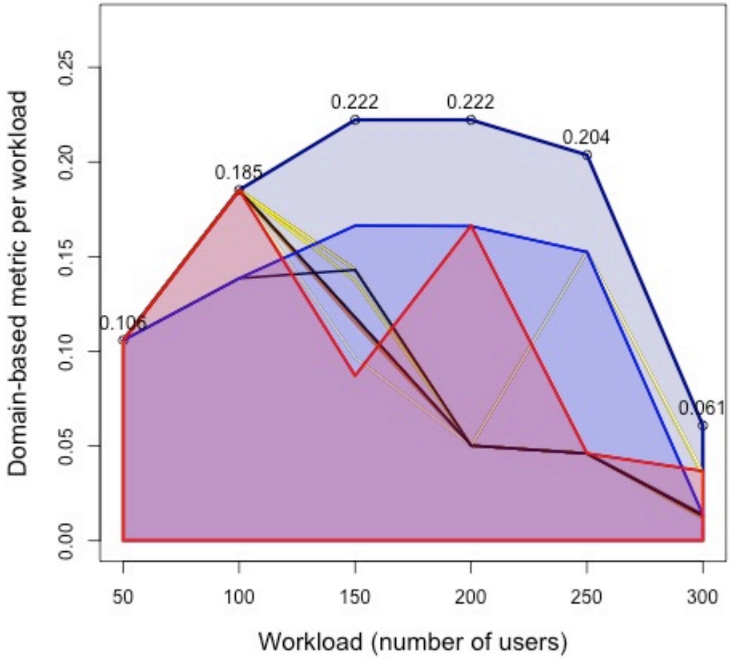


i		s_0	···	s_{n-1}	Σ
⋮	⋮	⋮	⋮	⋮	
1	ϕ	0.015		2.164	
	Γ_k	0.042	···	0.108	
	pass/fail (c_k)	PASS		FAIL	
	δ_k	1.26 %		2.58 %	100.00 %
	$\delta_k \cdot c_k$	1.26 %		0.00 %	74.81 %
	norm. test mass ($\hat{s}_i$ * p'(λ'))				0.142
⋮	⋮	⋮	⋮	⋮	

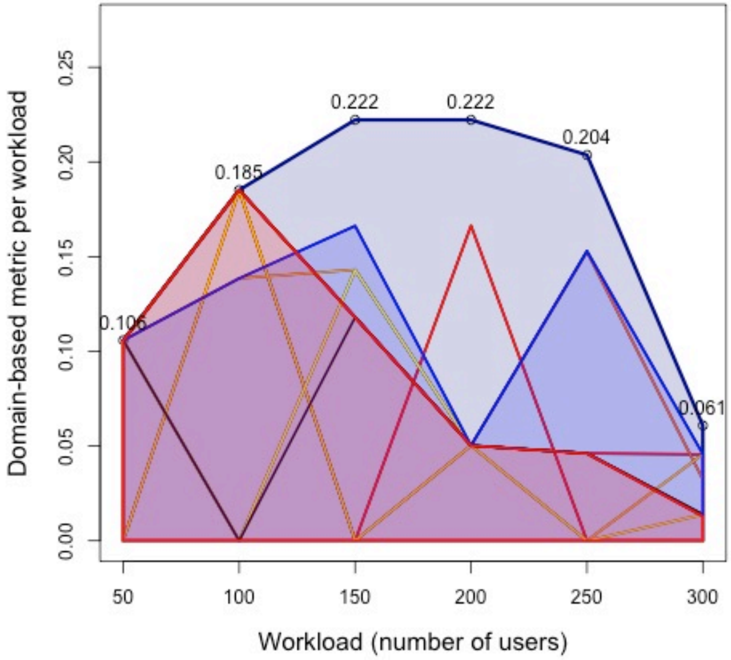
unibz Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Experiments - results

Virtual – no attack



Virtual - attack



Attack design

- After few piloting attacks (5-10-20 mins)

- Duration of attack: 20 minutes (1200 seconds);
- Protocol used: HTTP;
- IP address to attack: the IP address of the SUT, i.e., the
- Machine with Sock Shop installed;
- Number of threads: 256.

Bare-metal versus virtualization environment

Monitor performance degradation under attacks

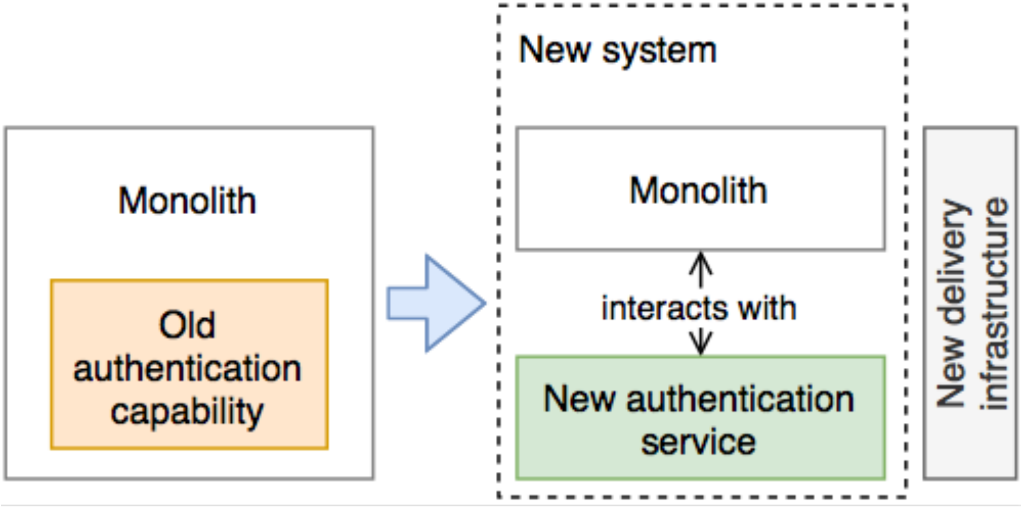
Bare-metal versus virtualization environment

Monitor performance degradation under attacks

Monitoring performance degradation during a transition to microservices

Dehghani's approach to transition

- Identify one capability in the monolith to transform it into microservice(s)


Monolith
Old authentication capability
New system
Monolith
interacts with
New authentication service
New delivery infrastructure

Dehghani’s approach to transition

- Decouple it from the monolith into an external service
- Maintain the old monolith with all its existing functionalities
- Work incrementally: build, test, and deploy

Z. Dehghani, “How to break a Monolith into Microservices,” April 2018, Fowler’s page

Transition uncertainty

- Some aspects are new:
  - one has to decide on a communication infrastructure
- Other aspects that are valid when developing a monolith have to be reconsidered
  - For instance, how to keep communication between services minimal (as communication is costly and might impede scalability)

Transition uncertainty

- It requires the team to acquire new knowledge and to learn how to apply it
- New software design patterns for microservice architectures:
  - API Gateway pattern to organize how clients can access individual services

F. Pacheco, Microservice Patterns and Best Practices: Explore Patterns Like CQRS and Event Sourcing to Create Scalable, Maintainable, and Testable Microservices, Packt Publishing, 2018

Transition uncertainty

- A transition to microservices may or may not end up with the same or better performing system

- It depends on the ability of the developers to design microservices and the capability of the microservices architecture to represent the system

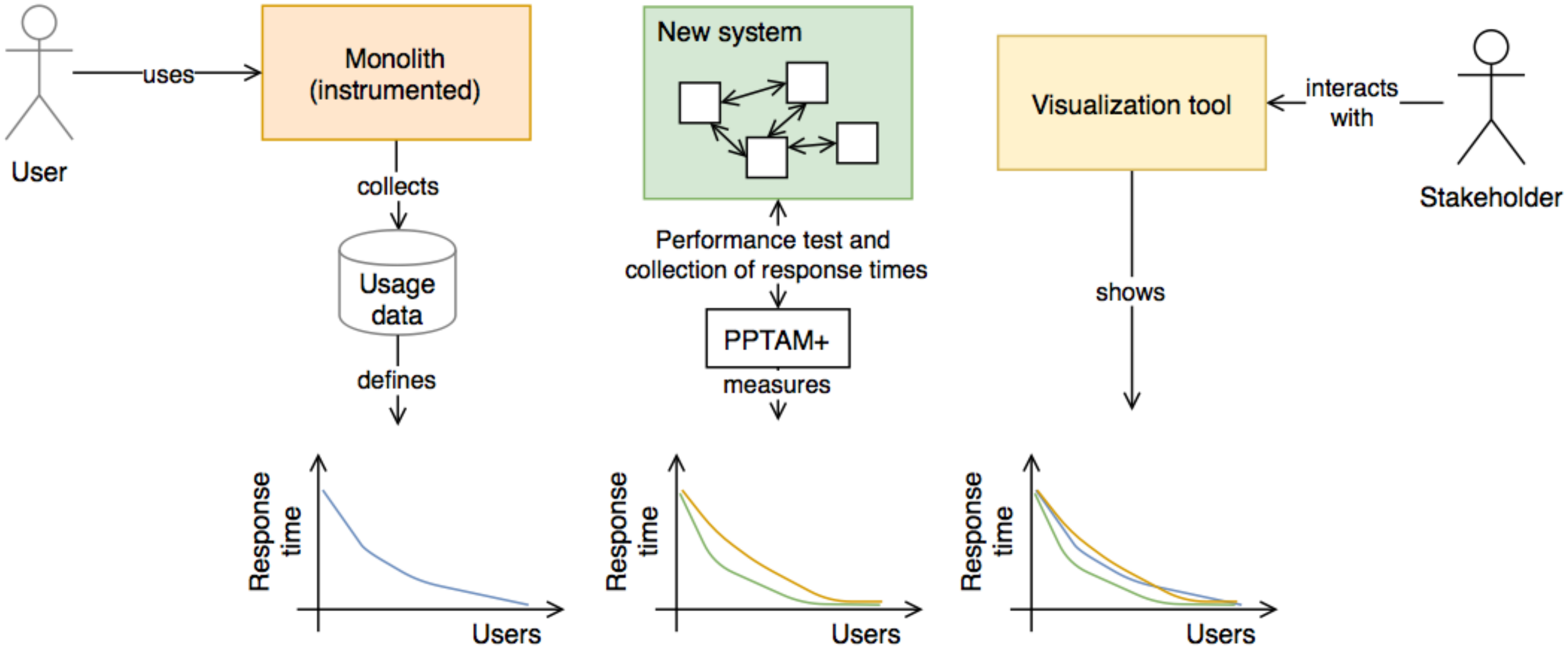
Main steps

Main steps

- Compute the operational profile of a monolith
- Apply PPTAM to collect individual service - individual experiment - individual workload time series
- Monitoring performance degradation over time against baseline and experiments' average performance

Main steps

- Compute the operational profile of a monolith
- Apply PPTAM to collect individual service - individual experiment - individual workload time series
- Monitoring performance degradation over time against baseline and experiments' average performance
- Analytic extension: visualize such analysis (R shiny)


User
uses
Monolith (instrumented)
collects
Usage data
defines
Response time
Users
New system
Performance test and collection of response times
PPTAM+
measures
Response time
Users
Visualization tool
interacts with
Stakeholder
shows
Response time
Users

Application to a transition

- If the new architecture **performs under a given threshold**, developers **stop and rethink** of the **architecture** or rethink the **used patterns** to guarantee that the new system - while having all advantages of a microservice architecture - does not fall short in terms of performance