The Journal of Systems and Software 116 (2016) 162–176

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Systematic scalability assessment for feature oriented multi-tenant services

Davy Preuveneers*, Thomas Heyman, Yolande Berbers, Wouter Joosen

iMinds-DistriNet, Department of Computer Science, KU Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

ARTICLE INFO

Article history:
Received 9 October 2014
Revised 11 December 2015
Accepted 14 December 2015
Available online 22 December 2015

Keywords:
Distributed systems
Scalability
Tool support

ABSTRACT

Recent software engineering paradigms such as software product lines, supporting development techniques like feature modeling, and cloud provisioning models such as platform and infrastructure as a service, allow for great flexibility during both software design and deployment, resulting in potentially large cost savings. However, all this flexibility comes with a catch: as the combinatorial complexity of optional design features and deployment variability increases, the difficulty of assessing system qualities such as scalability and quality of service increases too. And if the software itself is not scalable (for instance, because of a specific set of selected features), deploying additional service instances is a futile endeavor. Clearly there is a need to systematically measure the impact of feature selection on scalability, as the potential cost savings can be completely mitigated by the risk of having a system that is unable to meet service demand.

In this work, we document our results on systematic load testing for automated quality of service and scalability analysis. The major contribution of our work is tool support and a methodology to analyze the scalability of these distributed, feature oriented multi-tenant software systems in a continuous integration process. We discuss our approach to select features for load testing such that a representative set of feature combinations is used to elicit valuable information on the performance impact and feature interactions. Additionally, we highlight how our methodology and framework for performance and scalability prediction differs from state-of-practice solutions. We take the viewpoint of both the tenant of the service and the service provider, and report on our experiences applying the approach to an industrial use case in the domain of electronic payments. We conclude that the integration of systematic scalability tests in a continuous integration process offers strong advantages to software developers and service providers, such as the ability to quantify the impact of new features in existing service compositions, and the early detection of hidden feature interactions that may negatively affect the overall performance of multi-tenant services.

1. Introduction

Feature oriented software development, cloud computing and multi-tenancy are triggering a tremendous shift in the software systems landscape. Software product line (SPL) oriented development methods and feature modeling allow customers to pay only for the features they need, resulting in a potentially large reduction of software costs. Cloud based deployment environments allow customers to pay only for the computational resources they need, resulting in a potentially large reduction of operational costs. This operational cost reduction is pushed even further in multi-tenant software as a service (SaaS) applications, where all tenants share resources by using the same instance of the SaaS application. In order to maintain flexibility and be able to cater to varying tenant requirements, dynamic software product lines are often used, where tenant specific customization of the SaaS application is enforced at runtime. However, all this flexibility comes with a catch: As the combinatorial complexity of feature and deployment variability increases, the difficulty of assessing system qualities such as scalability and quality of service increases too. And if the software itself is not scalable (for instance, because of a specific set of selected features), deploying additional service instances is a futile endeavor.

There are two stakeholders in particular to whom (unanticipated) feature interactions and their impact on scalability and quality of service are important: the service customer (i.e. the entity that acquires a product from a software product line, or the tenant in a multi-tenant system) and the service provider (i.e. the

* Corresponding author. Tel.: +3216327853.
E-mail address: davy.preuveneers@cs.kuleuven.be (D. Preuveneers).

http://dx.doi.org/10.1016/j.jss.2015.12.024
0164-1212/© 2015 Elsevier Inc. All rights reserved.

owner of a software product line, or the manager of a multi-tenant system). Service customers need to know what the cost of selecting a new feature is on overall service scalability, as they want to avoid rendering the system unable to scale up, resulting in an unresponsive system and frustrated end users. Service providers need to know what tenants can be hosted on the same machines without unanticipated interactions in virtualized environments. Deploying a new customer's service (and associated features) in a virtualized environment could interact with, and impair, the scalability of another customer's service, which would equally frustrate that customer.

Clearly there is a need to systematically measure the impact of feature selection on scalability, as the potential cost savings can be completely mitigated by the risk of having a system that is unable to meet service demand. While a large body of work exists that focuses on specific parts of this scalability problem, not much work has been done to study whether it is feasible to quantify the scalability and performance of such a complex distributed system in a practical way. In this work, we document a holistic approach on systematic load testing for automated quality of service and scalability analysis. The major contribution of our work is tool support and a methodology for scalability analysis of these distributed, feature oriented software systems throughout a continuous integration process. We take the viewpoint of both the service customer (i.e. the tenant) and the service provider, and report on our experiences applying the approach to an industrial use case in the domain of electronic payments. We conclude that it is possible to integrate systematic scalability tests in a continuous integration process, which allows early detection of feature interactions that negatively impact performance.

This paper is structured as follows. In Section 2, we make explicit what we mean by scalability and quality of service, and we summarize the current state of the art of scalability and quality of service assessment. In Section 3, we introduce a case study in the domain of e-payment, as well as an implementation, to illustrate the problem of scalability assessment in this context. Additionally, we introduce our system for systematic scalability assessment. In Section 4, we introduce the framework and methodology we use to perform systematic scalability assessment on multi-tenant feature oriented systems. In Section 5, we document our results of applying the framework to the case study. The work is discussed in Section 6. We conclude in Section 7.

2. Background and related work

We give a brief overview of scalability in Section 2.1, and overview dynamic software product lines and feature modeling in Section 2.2. We summarize related work on performance modeling and prediction in Section 2.3.

2.1. Scalability

Assessing system performance and scalability is a practice that cross cuts many levels of abstraction, ranging from low-level benchmarks of execution environments and embedded software, to high-level distributed systems and business process benchmarks. For instance, Guthaus et al. (2001) perform benchmarking on embedded programs and provide a comparison with the industry standard benchmark suite SPEC2000. Ghosh et al. (2005) analyze the performance of WiMax networks. Uskov (2012) provides a comprehensive study of the performance of authentication and encryption algorithms for virtual private networking. Rashwan et al. (2012) study the performance of message authentication codes for mobile networks, for both residence time and power consumption. Dayarathna and Suzumura (2013) document their results of comparing the performance of three complex event processing engines via benchmarking. Carvalho and Pereira (2010) document a method to analyze scalability of running systems from the data center viewpoint, by only measuring CPU utilization.

This work considers the scalability of large, multi-tenant distributed software systems. The scalability of a system is often defined as its ability to handle increasing user load. When faced with a user load of p concurrent users that issue a certain volume of requests per second, a service will be able to successfully handle a specific fraction of p, called its throughput, and denoted by $X(p)$. In the ideal case, a service can handle the requests generated by all users concurrently, or $X(p) = p$. That situation, also referred to as linear scalability, only tends to hold in real systems for low values of p, i.e. low load situations. For increasing loads, however, $X(p) < p$, as a system will not be able to successfully handle the requests generated by all users concurrently, and some requests will have to wait or, in extreme cases, be dropped. To find $X(p)$, we can simulate p concurrent users that issue requests at a fixed rate, and measure $X(p)$. Based on this data, we can calculate the capacity ratio $C(p) = X(p)/X(1)$, which is the ratio of the throughput of the system for a load of p, compared to the baseline of its throughput for a load of 1. The relative capacity curve $C(p)$ is a good indicator for how much the observed behavior diverges from the ideal linear scalability $C_L(p) = p$. The closer $C(p)$ is to $C_L(p)$, the better the scalability of that system configuration.

There are a number of models that attempt to quantify the capacity C of a system. One of these models, the universal scalability law (USL) (Gunther, 1993; Gross et al., 2013), takes into account both the serial nature of the workload of that system (i.e. how much of the workload can be parallelized in theory) and coherency costs (i.e. the costs incurred when waiting for data to become consistent between different collaborating processes). The universal scalability law quantifies capacity in function of user load as:

$$C(p) = \frac{p}{1 + \sigma(p-1) + \kappa p(p-1)}$$

Here, κ denotes the impact of coherency on the system performance, and σ denotes the serial fraction, which is the fraction of the workload that cannot be parallelized. When the coherency factor κ is negligible, the maximum performance of the system is bounded only by the serial fraction, and the model reduces to Amdahl's Law (Amdahl, 1967). When κ is non zero, the performance model of a system will have a specific maximum, achieved for a load $p^* = \lfloor\sqrt{(1+\sigma)/\kappa}\rfloor$. Beyond p^*, the throughput of a system will decrease. An illustration of a scalability curve according to the Universal Scalability Law, and a comparison to Amdahl's Law, is given in Fig. 1. We can find values for σ and κ for a specific service deployment by performing linear regression on measured values for $C(p)$ (Gunther, 2007).

So what happens to a request when the load is sufficiently high that $C(p) \ll p$? When the system remains stable and does not drop requests, the residence time of those requests (i.e. the time between issuing a request and receiving an answer) will start to grow exponentially. While the service capacity considers the business view (i.e. *How many servers do I need to handle this many concurrent users?*), residence time considers the end user's perspective (i.e. *How long do I have to wait before my request is handled?*), and is therefore an equally important aspect of scalability to consider. However, only knowing the average residence time does not suffice. In the case of quality of service policies, which are usually expressed in function of X% of requests that need to be handled within Y (milli-) seconds, we need to know the overall statistical distribution of the residence times.

To measure values for $C(p)$ and obtain residence time distributions, load testing frameworks simulate users by generating actual requests. There are a number of load testing frameworks in

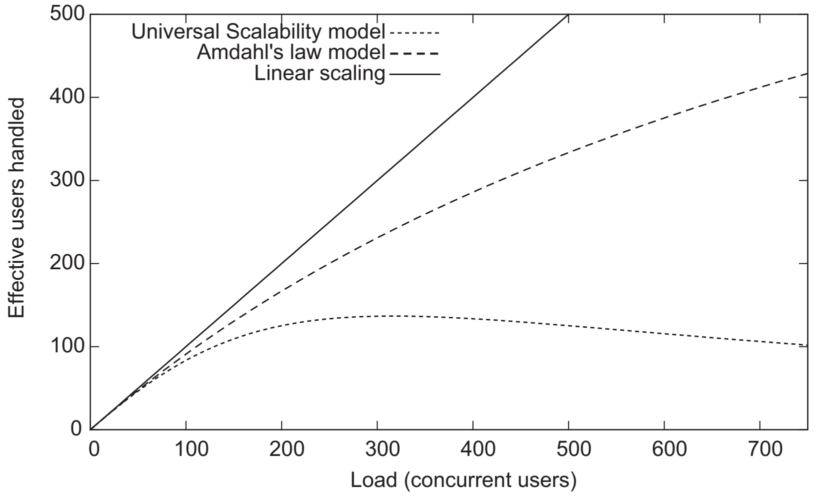

Fig. 1. The Universal Scalability Law, and the impact of σ and κ. In the general case of the Universal Scalability Law, where $\sigma \neq 0$ and $\kappa \neq 0$, the system has a finite maximum capacity, after which a higher load results in an even smaller effective throughput. When $\kappa = 0$, the USL reduces to Amdahl's Law, and capacity is bound only by how much the task can be parallelized. When both $\sigma = 0$ and $\kappa = 0$, i.e. there is no coherency penalty and the task can be parallelized ad infinitum, the system scales linearly.

existence, ranging from load tests embedded in integrated development environments (such as Microsoft Visual Studio) to web testing frameworks with support for distribution (such as Apache JMeter). We briefly overview four representative instances (Selenium, Gatling, JMeter and The Grinder).

Selenium (2014) is a browser automation framework. It comes in two flavors. Selenium IDE is a Firefox add-on that allows recording of interactions between a user and a website that can be played back later. Selenium WebDriver is a collection of language specific bindings that allow controlling a browser programmatically. Selenium is not built specifically for load testing, and does not come with built in scalability and quality of service analysis tools. There is also no out of the box support for communication and synchronization between these nodes. Finally, Selenium is focused on web applications, and it is not suited to automate load testing of arbitrary (i.e. non web) services.

Gatling (2014) is an open source tool that focuses on load and stress testing. Scenarios can be encoded in either a domain specific language, or recorded via a proxy that intercepts browser traffic. It provides reports for load testing analysis which include a breakdown of active sessions, requests per second, and request residence time distribution. Gatling focuses mainly on testing HTTP(S) scenarios. It does not support clustering out of the box, but there is a workaround to aggregate output from multiple independent Gatling instances. However, these can neither synchronize nor communicate with each other.

The Apache Software Foundation (2014) is one of the best known open source load testing tools. It is very flexible, and supports not only web applications, but also SOAP, FTP, various database protocols, SMTP(S), etc. It also supports clustering out of the box, by leveraging RMI. This limits JMeter clusters to the same subnet. There is also no inter machine communication facility, except for passing static data in configuration files. Although JMeter is fully extensible by means of plugins, there is no default support for scalability analysis (e.g., by means of applying the Universal Scalability Law).

The Philip Aston (2014) is a Java load testing framework for HTTP web servers, SOAP and REST web services, and application servers. It offers a simple and minimal runtime, with flexible scripting in Jython. It is fairly straightforward to run distributed tests that leverage many load injector machines. As with JMeter, The Grinder has distributed agents that collate the data and send it back to the coordinator. The Grinder differs from JMeter in architectural design: its scripting based nature is a major difference with the component based structure of JMeter. Similarly to JMeter, however, The Grinder does not offer default built-in support for scalability analysis.

Given the aforementioned shortcomings, systematically applying these tools to analyze the scalability of large, distributed systems that support dynamic feature adaptation and complex work flows, is not straightforward.

2.2. Customization with dynamic software product lines

Software product lines (SPL) (Clements and Northrop, 2001) are rapidly emerging as a viable and important software development paradigm. An SPL is a set of software intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment and that are developed from a common set of core assets in a prescribed way. The Software Product Line Engineering (SPLE) (Krut and Cohen, 2009; Lee and Kotonya, 2010; Galster et al., 2013; Mahdavi-Hezavehi et al., 2013) methodology is a widely adopted approach to ensure that artifacts generated in the development of a product can be reused in a related product. Variation points are typically bound during different stages of development, e.g., at design time before the delivery of the software, or even at run time.

Feature modeling (Czarnecki et al., 2004) is a design technique to formally and graphically model the attributes or features of a family of products. It describes the interdependencies of the product features and permitted variants and configurations of the product family. Clafer (Bak et al., 2011) is such an example of a meta modeling language with first-class support for feature modeling. It couples feature models and meta models via constraints to map feature configurations to component configurations.

Feature models have also been applied in dynamic software product lines (DSPL) to implement variability reconfiguration at runtime. DSPLs (Hallsteinsen et al., 2008; Istoan et al., 2009; Hinchey et al., 2012; Rosenmüller et al., 2011) produce software that is capable of dynamically adapting its behavior to changes in user requirements, resource constraints, adverse conditions, etc. This dynamism means that the configuration of features can vary at runtime several times during the whole life cycle of the product (Shen et al., 2011; Apel et al., 2013; Mietzner et al., 2009).

2.3. Performance modeling and prediction

Accurate performance modeling and prediction has been a topic of interest in the area of High Performance Computing (HPC) for more than a decade. Early work by Snavely et al. (2002) presents a framework for performance modeling and prediction that is faster than cycle-accurate simulation. Their prediction model is based on a single processor and network performance. Pace (Nudd et al., 2000) is a similar performance analysis toolset that provides information about execution time, scalability and resource use in terms of a hierarchical model of the application, its parallelization and hardware components. Similar to our contribution, these frameworks are used to analyze and compare different computational infrastructures, the effect of tuning and the scalability of scientific computation application kernels. Contrary to these works, our Scalar framework treats the application or service under test as a black box and does not aim to differentiate between the computation and communication aspects for the sole reason the application space that we aim to cover is much broader and potentially more complex compared to these high performance computing applications. Additionally, their objective is to predict the completion time of a scientific computation task in terms of a growing size of the mathematical problem, whereas our goal is to analyze the impact of a growing number of concurrent users of a service and the effect of customization.

From a typical software engineering perspective, Balsamo et al. (2004) present a comprehensive overview of various model-based performance prediction techniques, ranging from annotated UML and Queueing Network (QN) models, architectural pattern- and trace-based methodologies, to process-algebra, Petri-Net and simulation-based methods. They argue that software development should not only address software correctness, but also incorporate performance prediction at software development time in order to better bridge the gap between software engineers and quality assurance experts. Our contribution falls in the category of simulation-based approaches, and considers the service under test as a black box. The motivation for this black box approach is that it is very hard and time consuming to get QN models correct and detailed enough to obtain accurate predictions, especially for software services that can be customized per user or tenant. Scalar's performance and scalability prediction capabilities do not rely on any manually crafted application model with a detailed decomposition of its individual components that interact with one another.

In recent years, there has been an increased interest in performance and scalability prediction research for software and systems operating with shared resources, such as cloud applications and services. Abel et al. (2013) reviews the impact of resource sharing on performance prediction for multi-core systems with shared buses. They use resource-stressing benchmarks to explicitly quantify the worst-case impact of the interferences on shared resources. They conclude that the interference upper bounds of these benchmarks are pessimistic and that concurrent applications rarely exercise such a large stress on a shared resource. Zhang et al. (2014) propose OPred, a framework for online performance prediction for service oriented architectures. Their focus is on distributed service deployments and obtaining a personalized web service composition by intelligently selecting among multiple candidates those services to fulfill a particular function in the web service composition with the best quality of service.

Multi-tenant services are frequently being customized towards the needs and preferences of their tenants, and SPL-based methodologies have been adopted by Walraven et al. (2014) to efficiently customize such multi-tenant SaaS applications. Guo et al. (2013) investigated performance prediction of such applications that exhibit a high-degree of variability. They used statistical learning techniques to predict performance based on a small sample of measured variants. Their approach aims to take into account the impact of features within an application that interact with one another in unforeseen ways. Our work further investigates the notion of feature interactions, and aims to quantify the impact of hidden interactions across tenants. We return to this in Section 4.

3. Software and systems

In order to illustrate the scalability assessment framework documented in this paper, we introduce our case study in the domain of e-payment, along with the various features it supports, in Section 3.1. The system for scalability assessment upon which we build our approach in Section 4, is documented in Section 3.2.

3.1. MobiCent, a system for online payment

MobiCent[1] is an industrial case study on developing a multi-tenant software service for payment products. The MobiCent platform supports various payment work flows with different requirements, implemented as payment features, in a single payment platform instance. The payment domain is characterized by a broad variability in functional and non-functional requirements due to the rapid technology evolution and transaction growth. The case study illustrates this complexity, and shows how SaaS providers are faced with the difficulty of managing two goals at the same time, i.e. (1) supporting varying requirements per tenant, and (2) providing horizontal service scalability. Because different payment schemes per tenant have wildly varying requirements, MobiCent supports these requirements as various features, as depicted in Fig. 3. We apply a dynamic software product line methodology in which the variation points, i.e. the various features via which the payment process can be customized, are bound per tenant at runtime. This allows tenants to optimize MobiCent for their own needs, taking into account a variety of performance, availability, scalability and security requirements.

As illustrated in Fig. 2, MobiCent is a multi-tenant software as a service application with RESTful interfaces that allow for the creation and consumption of monetary transactions. A transaction is a collaboration between at least three parties: the merchant, the client, and one or more trusted third parties to verify the identities of the merchant and client, and the validity of the payment transaction. MobiCent allows business owners to create electronic coupons that can later be redeemed by clients, or sellers to create payment requests that can be sent to a buyer for completion at a later time. This decoupling of creating and consuming payment requests allows for more flexibility in applying the payment framework to different use cases with varying security requirements on confidentiality, authentication, integrity, authorization and non repudiation.

For its implementation, MobiCent is built on top of a public key infrastructure, coupled to an off-the-shelf user management and access control system[2]. As one of the goals of MobiCent is to be applicable to widely varying cases, ranging from coupons for small scale promotional campaigns to large scale digital currency deployments, its implementation must be scalable. Therefore, it uses a scalable distributed data store in the backend, which enables clustering of MobiCent servers. Transactions are automatically partitioned and replicated among the different MobiCent instances.

Because MobiCent caters to heterogeneous tenants, ranging from small businesses in the small scale promotional campaign

[1] An instance of the MobiCent service is hosted at https://pong.cs.kuleuven.be/mobicent

[2] For user management and access control, MobiCent leverages OpenAM and the accompanying OpenDJ data store, which can be found at http://forgerock.com/products/open-identity-stack/.

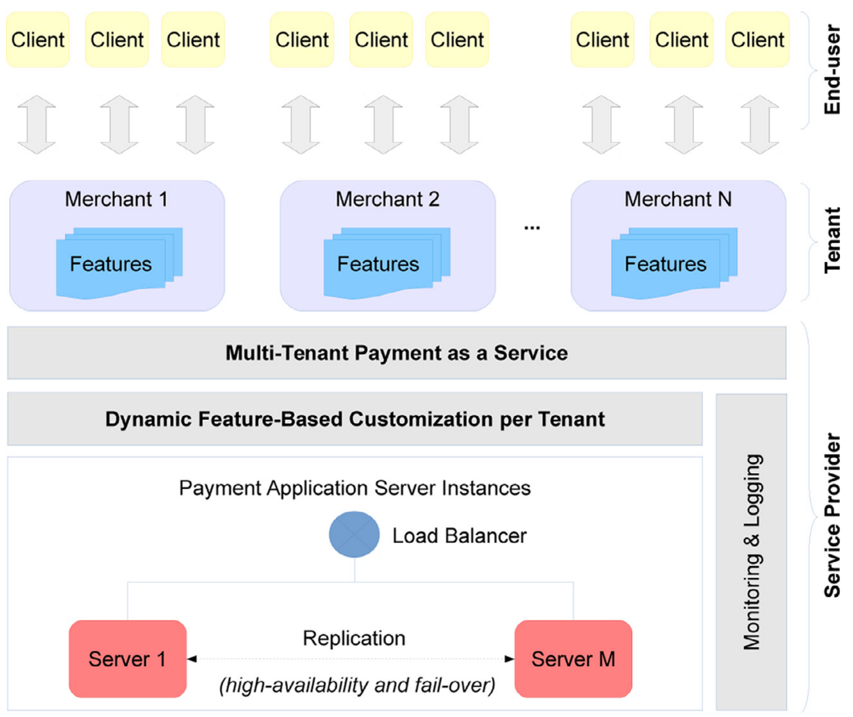

Fig. 2. Conceptual overview of MobiCent, a multi-tenant payment service supporting dynamic feature-based customization per tenant.

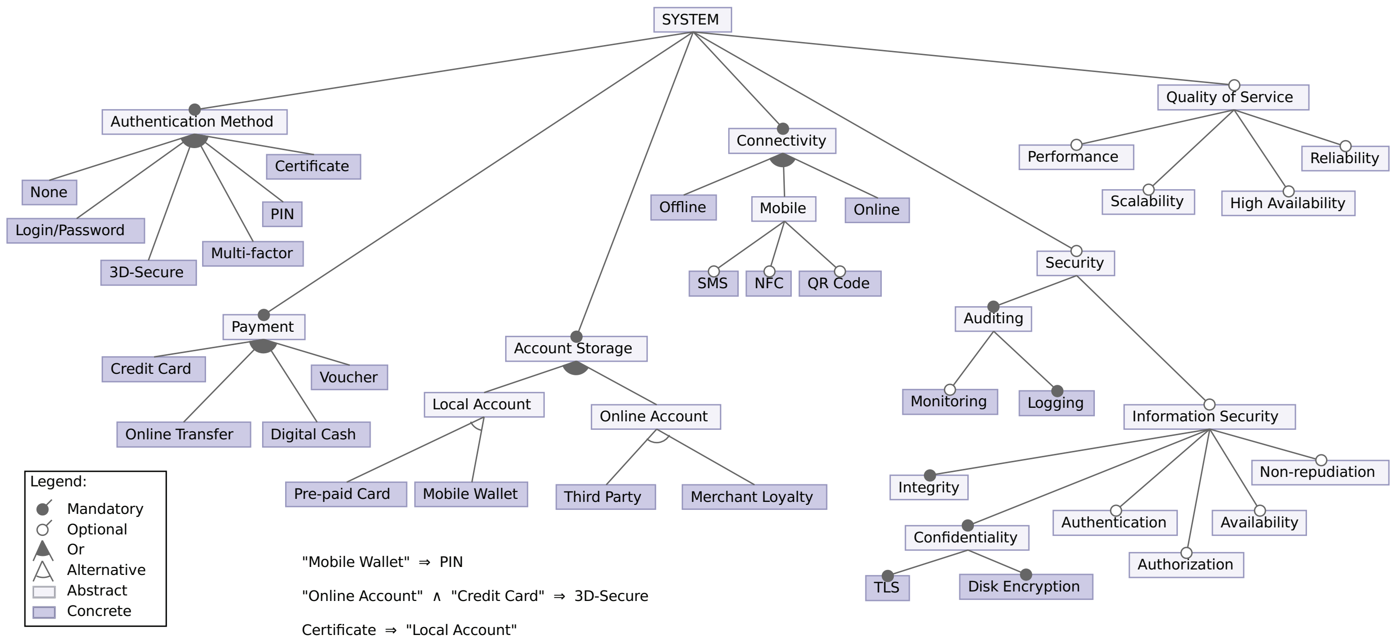

Fig. 3. The feature model of the electronic payment case study.

case, to larger city wide deployments in the case of digital currency used for reward schemes, a lot can be gained by pooling computational resources. That allows the allocation of resources to smaller tenants only when required, and adding resources to larger tenants on demand during peak moments. Therefore, MobiCent is developed as a multi-tenant system: One MobiCent deployment can host many service customers, each of whom is registered as a tenant, and all of whom share computational resources.

3.2. Scalar, a system for systematic scalability assessment

In order to simulate users and generate load to test live service deployments, we leverage Scalar (Heyman et al., 2014a; 2014b). Scalar is a distributed load generation framework, implemented in Java, that allows to easily implement complex work flows that are executed in parallel to generate user load. In order to adapt the generated load to variously sized systems under test, Scalar is designed to be scalable itself. Additionally, Scalar contains self monitoring features that are able to detect and mitigate bottlenecks internal to the load generation process, which makes the load generation process more robust. Furthermore, Scalar offers inter user and inter machine communication and synchronization, and its core functionality can easily be extended by means of plugins—a feature that will be essential to implement our approach as documented in Section 4.

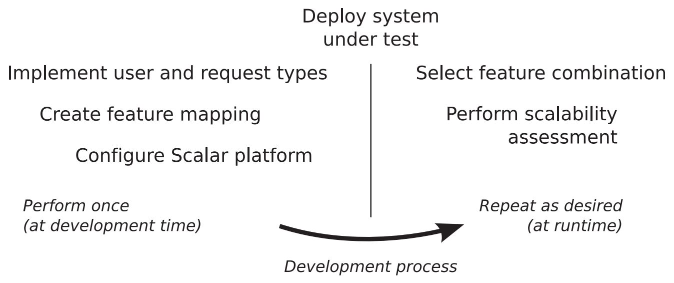
Deploy system under test
Implement user and request types
Create feature mapping
Configure Scalar platform
Select feature combination
Perform scalability assessment
Perform once (at development time)
Repeat as desired (at runtime)
Development process

Fig. 4. High level overview of the scalability assessment process.

The core concepts in Scalar are user types and request types. A user type is a Java class that encapsulates a certain business flow that needs to be followed, and against which the scalability of the system under test is to be evaluated. In the MobiCent case, the business flow consists of an end-user, i.e. an instance of the user type, wishing to check out or pay for goods. The tenant (or seller of the goods) creates a payment transaction, after which that transaction is consumed by the end-user. All scalability results are relative to the encoded user type behavior: A simulated user load of p entails p concurrent user objects that perform the encoded business flow at a configurable fixed rate, e.g. once per second. The corresponding measured capacity $C(p)$ is the number of those flows that were successfully handled by the system in the same time period.

The way in which a user type communicates with the system under test, is via request types. In the MobiCent case, there are two request types: creating a payment transaction and consuming the transaction. Residence times, and corresponding quality of service results, are broken down in function of request types: Of every request, Scalar retains the residence time, as well as the request result (i.e. whether it succeeded or failed). The residence times are subsequently fitted to a gamma distribution, and a breakdown of residence time percentiles is provided.

Based on this load generation and scalability analysis functionality, it is possible to build a framework to automatically study the impact of feature selection on the scalability of multi-tenant systems, which is the topic of the next section.

4. A framework for scalability assessment

In this section, we introduce a framework that allows automated and repeatable scalability assessment of the MobiCent case (as introduced in Section 3.1), built on top of Scalar (as introduced in Section 3.2). An overview of the framework is outlined in Section 4.1, and we apply it to the MobiCent case in Section 4.2. We briefly overview the potential overhead of a multi-tenant architecture in Section 4.3, as well as its impact on scalability. We summarize strategies to select feature combinations to test, including an automated feature selection strategy, in Section 4.4. We discuss load generation models in Section 4.5.

4.1. Overview of the framework

A graphical overview of the process required to apply the framework is presented in Fig. 4. As the framework depends on the Scalar platform, we assume that suitable user and request types have been implemented so that the Scalar platform can generate load to assess the system under test. Once the required user and request types have been implemented, we create a feature mapping, which is a configuration file that describes how features are mapped to code, and how they can be programmatically activated and deactivated. In order to perform this mapping, we have to distinguish between features that are internal to the service under test and are transparent to users of that service, and those that have an impact on the interaction between a user and the service (i.e. features that impact the service's public interface). For instance, enabling logging to establish an audit trail is transparent to users of that service, but enabling authentication is not, and will alter the way in which users interact with that service. Automating this feature mapping implies that the service has to expose an interface via which a tenant can enable and disable features, something we will return to in Section 4.2. In the case of features that alter the interaction between a user and the service, new Scalar user types can be added, so that the generated load is adapted to the active feature combination. The final step required at development time, is to configure and deploy the Scalar platform itself.

Once the system under test is deployed, the scalability assessment can be performed. In order to do that, the remaining steps involve selecting a feature combination to be tested, and starting the Scalar system. Given the multitude of features as shown in Fig. 3 and the immense number of potential feature selections, it is clear that testing all possible feature combinations would take prohibitively long, especially for large feature models. Instead, we focus only on a predefined set of relevant feature combinations. Feature combinations could be collected automatically from operational monitoring, or selected manually—we return to this in Section 4.4. To select a consistent feature combination manually (i.e. while ensuring that no invariants are violated, such as *Mobile Wallet* implies *PIN*, as shown in Fig. 3), we leverage FeatureIDE (Thüm et al., 2014). The configuration files generated by FeatureIDE are directly compatible with Scalar, and can be trivially incorporated into the Scalar configuration.

Performing the actual scalability assessment then proceeds as follows. Based on the selected feature combination and the feature mapping, the FeatureModelParser plugin of Scalar leverages helper classes to activate and deactivate features as required, bringing the system under test in a consistent state. Additionally, the FeatureModelParser instructs the Scalar cluster what type of traffic composition to use—that is, what user types to instantiate when generating load. Once the system under test and the Scalar cluster are ready, multiple experimental runs are started automatically, and the results are aggregated. When the required data are collected, the Scalar cluster stops. At this point, a new feature combination can be parsed, and the whole process repeats automatically.

The final link in the automated scalability assessment chain is to provide support for integrating all this in a continuous integration process. Similar to how continuous integration build servers automatically run test suites against new versions of a software product, we have created ant build scripts that facilitate automating the load testing of various feature combinations whenever a new version of the system under test is built. This allows early detection of scalability problems arising from unanticipated feature interactions at development time. We return to this in Section 6.

4.2. Multi-tenant customization in MobiCent

An overview of the required domain specific bindings and artefacts to enable scalability analysis of MobiCent with Scalar, is shown in Fig. 5. The typical requests involved in a business flow include the (1) initialization of a payment request by the merchant, (2) the customer downloading the payment request, (3) the customer confirming the payment, and (4) the merchant receiving the payment confirmation. This functionality is encapsulated in the MobiCentUser, resp. the CreateTransaction and ConsumeTransaction request types. Furthermore, depending on the kind of transaction and the features enabled by the tenant, the overall payment transaction can include certificate based authentication, digital signing and/or encryption of the transaction messages, verification of the credentials and digital signatures, balance verification of the end-user account. In the MobiCent case, we have opted to

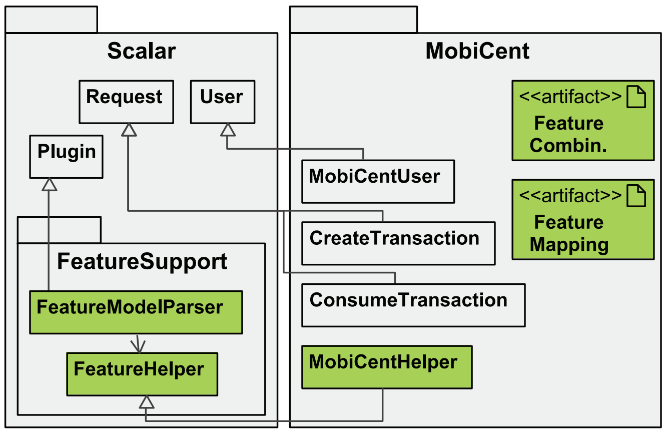

Fig. 5. Instantiating the Scalar framework for MobiCent. The green elements are specific to the feature oriented scalability analysis, the elements in the Scalar package are domain independent and fully reusable, while the elements in the MobiCent package are specific to the MobiCent integration.

integrate the feature specific business logic into the user and request types directly, as opposed to creating multiple distinct user types.

In order to enable automated feature configuration, MobiCent required minimal refactoring to expose the features as run time configurable assets of the software product line. The automated feature configuration is implemented via new RESTful service interfaces for:

- Tenant management and feature configuration.
- Initialization and settlement of payment transactions.
- Authentication, authorization, certificate management.
- Load balancing and replication.
- Logging and performance monitoring.

The MobiCentHelper class encapsulates the details of interacting with these RESTful interfaces into simple helper methods of the form ‘enableX()’ and ‘disableX()’, for every configurable feature X. Given this helper class, the feature mapping configuration takes the following form:

```
1 {
2   "feature": "Logging",
3   "enable": {
4     "helpers": {
5       "MobiCentFeatureConfig": "enableLogging"
6     },
7     "users": {
8       "MobiCentUser": 1.0
9     }
10  },
11  "disable": {
12    "helpers": {
13      "MobiCentFeatureConfig": "disableLogging"
14    }
15  }
16 },
17 ...
```

This fragment of the feature mapping shows how the *“Logging”* feature (line 2) is mapped to the helper implementation. In order to enable the Logging feature, an object of the type *“MobiCentFeatureConfig”* (line 5) should be instantiated, and the method *“enableLogging()”* (line 5) should be called. Additionally, a ratio of 1.0 of the type *“MobiCentUser”* (line 8) should be used to generate representative traffic. Similarly, the *“Logging”* feature should be disabled by calling *“disableLogging()”* (line 13) on the same object.

There are several methods on deriving interesting feature combinations to test. While we return on how to select feature combinations in Section 4.4, some example feature combinations that resulted from that informal selection process, are the following:

```
Online payment:
   Connectivity: Online
   Authentication: Login/Password
   Payment: Online Transfer
   Information Security: Non−Repudiation and Confidentiality

Voucher:
   Connectivity: Online
   Authentication: None
   Payment: Voucher
   Online Account: Merchant Loyalty
   Information Security: Non−Repudiation
```

Feature combinations are specified in a simple line oriented file format listing all the selected features (this format is the default, for instance, for feature combinations made and exported in FeatureIDE). This file is parsed and mapped onto a simple comma separated list of feature names (e.g. *Logging, QR Code*) that can directly be inserted into the Scalar configuration. Some high-level features in Fig. 3 are internally mapped on deployment and configuration parameters of the software assets. For example, the *Non-Repudiation* feature enforces the use public/private cryptographic key pairs and verification of digital signatures in the payment transaction, whereas the *Confidentiality* feature triggers asymmetric encryption of the transaction message using the same key pairs. Even deployment aspects can be configured with feature-based customization. For example, the *High-Availability* feature initializes data replication across all MobiCent nodes to support failover without downtime.

4.3. Assessing the impact of multi-tenancy

When a traditional, single-tenant application is transformed into a multi-tenant system, additional functionality is introduced to allow individual tenants to customize their instance of the system, and ensure that each tenant works in a virtually isolated environment, even though the underlying platform is shared. That may impact the overall system scalability and quality of service, and make it harder to measure and predict in general. The problem is twofold: first, the introduction of a multi-tenant management layer may introduce overhead, and second, feature selections of multiple tenants may interact in unforeseen ways.

First, depending on the way that tenant management layer is implemented, it can introduce a noticeable impact on performance, and negatively impact scalability. Even if there is no significant difference in resource consumption across the tenant configurations or other forms of feature interaction, the customization at service delivery due to late binding of tenant specific feature selections causes a runtime performance overhead that does not exist with tenant dedicated service deployments. Furthermore, in a multi-tenant service deployment the customization at runtime in the tenant management layer may increase as the number of tenants grows.

Second, feature selection (and corresponding resource utilization) of one tenant may negatively impact the quality of service of another tenant (e.g., Siegmund et al., 2012), even though the latter did not change anything to their existing feature set. This can easily occur if the first tenant selected a feature that depends heavily on one specific resource (e.g., disk access). If a new tenant selects features that increase disk utilization, then the first tenant's quality of service will likely decrease, even though that tenant did not change their feature set, or the total number of users of the system remained constant.

We therefore pursue a twofold approach to assess the impact of multiple tenants. First, we measure the overhead of introducing a multi-tenant management layer and a growing number of tenants, and second, we measure the impact of feature interactions by assessing specific feature sets. We want to achieve the first solution step (i.e., measuring the overhead of the management layer) while avoiding modeling pitfalls by treating the system as a black box, as per Section 2.3. This can be done by simulating traffic corresponding to an ever increasing number of tenants per instance, to empirically establish the multi-tenant overhead. In order to get a representative view on the overhead of the multi-tenant management layer, it is imperative to minimize the overhead introduced by other features: The multi-tenant overhead is measured by deploying a simple system (i.e. with a minimum of features enabled), while gradually increasing the number of tenants on that system.

Depending on the outcome of this experiment, two results are possible: the multi-tenant management overhead is negligible, or it is significant. In the first case, we can continue by assessing the system as if the traffic belongs to an arbitrary number of tenants, as this dimension is not relevant to the scalability of the system as a whole. In the second case, however, things are not as straightforward, and the number of tenants needs to be taken into account explicitly as an extra dimension while performing a scalability assessment—for every feature combination of interest, the assessment will have to be repeated for a varying number of tenants.

However, we claim that in reality, it is far more likely to encounter the first situation (where the number of tenants does not impact overall system scalability) than the second. This is because, in practice, the number of tenants is an order of magnitude lower than the number of users of the system. As long as the tenant management system is implemented in a way that tenant specific configurations can be enforced in linear time or better (e.g., when a linked list of tenant specific configurations needs to be searched for every incoming user request), the impact of the number of tenants can safely be ignored when compared to the number of users of the system as a whole.

To address the second solution step, we start from a number of feature combinations, and perform an assessment as presented in Section 4.1. In the case where the multi-tenant management overhead is negligible, we perform one assessment per feature combination for a growing number of users belonging to a fixed number of tenants. In the case where the multi-tenant management overhead is not negligible, we repeat the assessment per feature combination for a growing number of tenants (e.g., ‘few’, ‘normal’ and ‘many’ tenants, where the exact amounts are determined by either operational monitoring of the system (if it has been deployed), or on estimates (if the system is still under development). This leaves the generation of feature combinations to assess, which is the topic of the next section.

4.4. Creating feature combinations for scalability assessment

Interesting feature selections for performance testing can be created either manually, or automatically—the simple interface of Scalar (i.e. providing a comma separated list of features to enable) is deliberately chosen to support both. In this section, we provide some insights into how these feature selection strategies can be implemented.

Manually selecting feature combinations. Despite that even a small number of features easily result in an unrealistically large set of potential feature combinations, manually selecting interesting feature combinations is still feasible, for the following reasons. First, in a multi-tenant system, the service provider can easily see which features are commonly chosen together by customers. That data serves as a rough, but practical, partitioning of the feature space. Given that partitioning, whenever the implementation of a feature changes throughout a continuous integration process, it can easily be established what feature combinations should be assessed again. Furthermore, by basing feature combinations on naturally occurring groupings by service customers, the service provider can be assured that the scalability assessment is optimally relevant for that specific system.

Second, even when no customer data is available, manually selecting feature combinations can still provide useful insights. A rough partitioning of features can be made by the software development team based on the resource that a feature consumes most, i.e. whether it is CPU bound (e.g. features that involve digital signatures), involves disk I/O (e.g. *Online account*, or require additional network resources (e.g. *QR Code* and *Online account*). Additional grouping is performed based on features that depend on each other from a functional perspective, e.g. *Local account* and *Certificate*, or *Monitoring* and *Logging*). Whenever the implementation of a feature changes, feature combinations involving the dependent features, as well as combinations involving features that require the same types of resources consumed by the changed feature, should be assessed.

Automatically selecting feature combinations. When neither customer data nor the underlying causal model (i.e. what features consume the same type of resources, and what features depend on each other functionally) are available, it is possible to leverage automated strategies to predict the scalability of arbitrary feature combinations, based on a set of assessed feature combinations. One such method is the work by Guo et al. (2013), which leverages classification and regression trees to automatically create performance prediction models progressively, based on random samples. That approach can easily be adapted to create a scalability prediction model, based on results from Scalar: The performance measure (i.e. how many seconds it takes to complete a task) can trivially be changed with the relative capacity (i.e. how many concurrent users could effectively be handled by the system in a fixed period of time, for a fixed load).

A model to predict the relative capacity of arbitrary feature combinations can be built as follows. Initially, a starting set of random feature combinations (e.g. 16 random service configurations) is generated and assessed by Scalar. An initial naive model predicts the relative capacity of a new feature combination as the mean of the relative capacities of the observed feature combinations. The prediction error of that naive model is the sum of squared differences between the predicted relative capacity (i.e. the average of the known measurements), and the actual measured relative capacity. That initial model is iteratively refined via a classification and regression tree: For the root of the tree, a feature x is chosen whose state (i.e. enabled or disabled) divides the observed results in two smaller sets, so that the prediction error given by the mean value of each smaller set is minimized. Instead of coarsely predicting the capacity of a new feature as the mean of all values, the new model predicts the capacity as the mean of all values *with the same configuration for x*. New measurements are iteratively added, and the tree is additionally refined, until the prediction error of the partial models are sufficiently small. For more details and an illustration, we refer to Guo et al. (2013).

The advantage of the automatic strategy is that it does not depend on additional insights in the workings of the system under test. The downside is that the generated model is static with respect to the features: If a new feature is added, or an existing feature is changed, the entire model has to be rebuilt from scratch. In the manual process, the potential impact of a new or updated

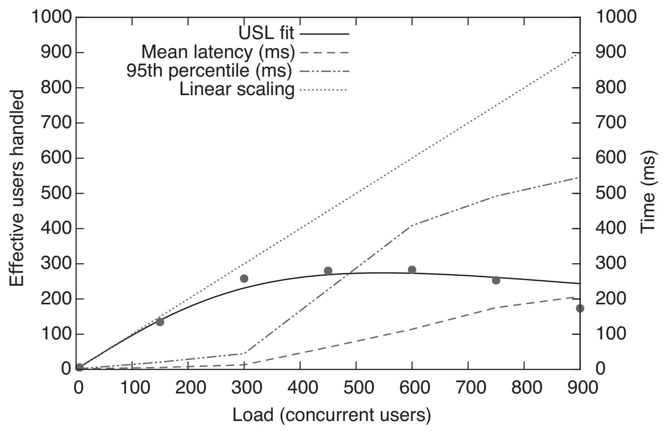

Fig. 6. Basic scenario with 1 MobiCent node and 10 tenants.

feature can be scoped better, and the set of assessments that need to be rerun can be minimized.

Note that this process leaves room for improvement: In the automatic approach, the initial feature selection is random and does not take developer know-how and insights into the overall design of the system into account, even if this would be available. If additional system insights are available (e.g., as per the manual feature selection), then it is possible to partition the feature space into segments of features that compete for the same shared resources (and are therefore likely to impact each other). A separate automated assessment can then be performed per feature segment. In addition, by replacing the initial random feature selection by the most popular features, the relevance of the regression results to the actual system deployment is increased. However, additional work is required to assess such a hybrid feature selection process for very large feature spaces.

Assessing and comparing feature combinations. Once a set of feature combinations is assessed with Scalar (or a sufficiently detailed prediction model is constructed, based on results from Scalar), they can be compared. Each assessed feature combination results in one breakdown of service scalability and request latencies as per Fig. 6, in which the capacity of a feature combination is interpreted according to the Universal Scalability Law. The USL also enables us to calculate the optimal load point p^* (as per Section 2.1), i.e. the number of concurrent users for which a maximum effective capacity is reached. By comparing the optimal load points for a set of feature combinations, they can easily be compared and ranked according to their impact on scalability. Similarly, given a quality of service policy such as what fraction of requests is handled in a specific time, the set of feature combinations can easily be ordered based on their impact on quality of service. Those results offers service providers insights into the maximum capacity of their deployment, the corresponding number of concurrent users, and when it is time to scale out.

4.5. Models for synthetic load generation and replay

Orthogonal to the problem of selecting interesting feature combinations to test, is the problem of how to generate synthetic load. That problem can be broken down into two problems: modeling user behavior and scheduling user requests. The former problem is handled by implementing user types. While manually implementing a user type potentially introduces bias to the load generation process (i.e. there is no guarantee that the encoded user behavior corresponds to reality), we argue that it is advantageous for two reasons. First, it allows benchmarking green field systems that not have been deployed yet, and for which no recorded network traces are available. Second, as the load is generated by explicitly encoded rules, the modeled user behavior is subject to review and can easily be adjusted to take into account new constraints and try out hypothetical situations.

The latter problem can be reduced to calculating think times, or the delay that a user waits before sending out the next request. Scalar supports multiple think time generation models. The default think time generation model in Scalar assumes a constant think time, and starts users off with a random offset in the interval *[0, think time[*. Given a large number of users and a relatively long constant think time, this makes the global inter request arrival period essentially random. Other implemented load generation models include random think times with a specific average, and exponentially distributed think times with a specific average.

Scalar is fully extensible in terms of think time generation model. If mathematical models of the required think time distribution exist, they can easily be dynamically added to the Scalar framework. Furthermore, if existing traffic traces are available, they can be used by Scalar to create a representative load generation model based on that trace. This works as follows: After providing Scalar with a histogram of an observed think time distribution, that histogram is then used as the basis to generate think times for the subsequent load test. However, as we focus in this work on assessing scalability in function of software features, we gloss over different think time distributions, and assume a constant think time distribution throughout the next section.

5. Experimental evaluation

In order to evaluate the proposed approach, we focus on evaluating the following specific points.

Creating a baseline for evaluation: In order to have a baseline to compare results to, we measure the scalability and performance of a simple system deployment and the impact of both a growing number of tenants (e.g. merchants) and a growing number of payment transactions on that simple system.

Multi-tenant invariance: As previously documented in Section 4.3, whether multi-tenancy needs to explicitly taken into account as another feature depends on the overhead introduced by the multi-tenant management layer. We evaluate whether this is the case for MobiCent.

Evaluating individual feature combinations: We evaluate if it is feasible to assess the impact on scalability and performance for individual feature combinations.

Feature selection: We evaluate the feasibility of selection and evaluation strategies of multiple feature combinations.

Clearly, an exhaustive overview of all performed experiments is not possible due to space considerations, and we limit ourselves to the main highlights. We begin in Section 5.1 with establishing a baseline. Section 5.2 looks into multi-tenant invariance and documents the impact of a varying amount of tenants on overall service scalability. Sections 5.4 and 5.5 document the scalability impact of introducing a new feature, resp. a specific feature combination. Finally, Section 5.6 goes into more detail in selecting and evaluating multiple feature combinations.

For our experimental setup, we use 10 desktop machines, each equipped with an Intel Core 2 Duo 3.00 GHz CPU and 4GB of memory. All machines are linked to a 1 Gigabit network. We use an additional 6 machines to simulate users that initiate payment transactions. Each MobiCent instance is deployed on an Apache Tomcat 8.0.11 application server on a 64-bit Ubuntu 14.04 system.

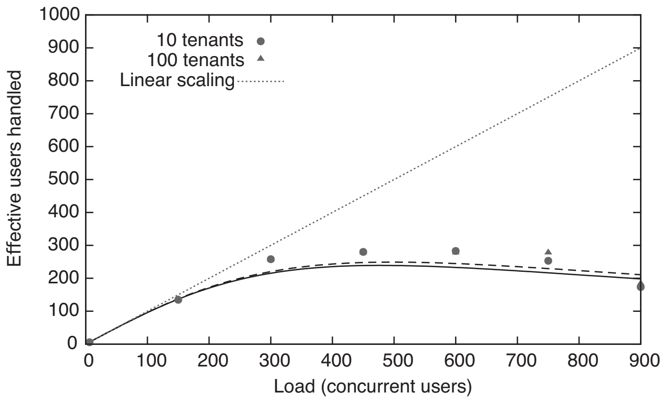

Fig. 7. Impact of 100 versus 10 tenants.

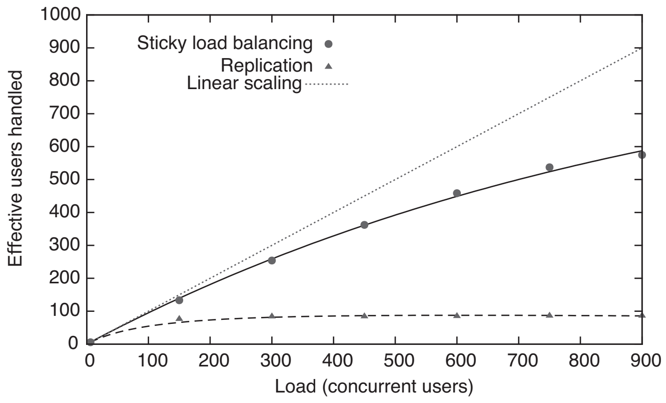

Fig. 8. Impact of sticky load balancing and replication on 3 MobiCent nodes.

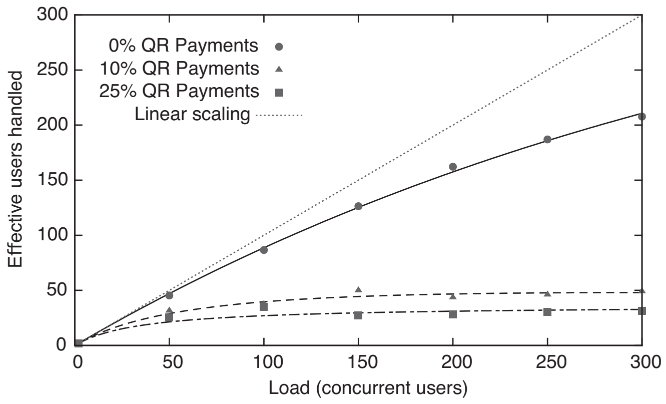

Fig. 9. Impact of QR payments as a new MobiCent service feature.

5.1. Basic scenario and scaling out

In order to establish a baseline for scalability comparisons, we deploy MobiCent on a single application server. The MobiCent service is configured to handle 10 tenants (i.e. 10 merchants). For benchmarking purposes, we let each user (i.e. customer) handle 2 complete payment transactions per second, and evaluate the capacity of the MobiCent service with an increasing load going up to 900 concurrent users (i.e. about 1800 complete payment transactions per second). The results are shown in Fig. 6. The figure shows that up to 100 users, we achieve near linear scalability. Around 400 concurrent users, the MobiCent service has reached its maximum capacity on a single instance deployment. As the number of concurrent users grows, so does the latency to handle the transaction. Fig. 6 shows the mean latency and the 95*th* percentile of the payment request latencies.

Table 1 shows the distribution of the latency times. Note that the distribution of residence times is skewed, as the mean latency is significantly larger than the 50*th* percentile (or median). This can be explained by the impact of outliers, which becomes much larger as the user load grows beyond the maximum capacity of the MobiCent service (as can be confirmed by the overall maximum residence time for processing a payment request).

5.2. Impact of number of tenants on customization overhead

In a second experiment, we reuse the same single node setup of before, while changing the number of tenants (i.e. merchants) from 10 to 100. Perhaps surprisingly, Fig. 7 shows that the number of tenants has little to no runtime customization impact difference on the capacity of the MobiCent service. The experiment shows there is a constant overhead of loading and enforcing a tenant specific configuration at runtime, which implies that overall service scalability is determined only by the selected features, and not by the total number of tenants hosted on one machine.

5.3. Introducing additional MobiCent nodes

In a third experiment, we investigate the impact of the scalability and high availability features by deploying two additional MobiCent instances and analyzing the horizontal scalability of the multi-tenant service. This experiment explores the difference in impact of sticky load balancing and high availability through replication. The results are shown in Fig. 8. The sticky load balancing configuration makes sure that all interactions within the same payment transaction are carried out by the same MobiCent node. In this configuration, there is no replication, meaning that if a node becomes unavailable, all pending transactions assigned to that node will be lost.

The high availability configuration replicates all data across the MobiCent nodes, so that another node can take over ongoing transactions when a given node goes down. To implement this replication, we make use of a distributed hash map built on top of the Hazelcast library. As expected, Fig. 8 shows that replication for high availability has a non negligible impact on the capacity of the payment service.

5.4. Impact of new features

In a fourth experiment, we analyze the impact of introducing a new feature, ‘QR payments’, on the existing setup. This feature generates a QR code of the payment transaction, and sends the resulting PNG image to the user. The QR code encodes the URL of the payment transaction, and corresponds to an additional step in the overall payment flow. Fig. 9 shows the results of 3 scenarios, one without QR payments, one with 10% of the transactions being QR payments, and one with 25% being QR payments. The figure shows that the introduction of the new feature has a significant impact on the maximum capacity of the MobiCent service, reducing it to less 50 concurrent users in the case of 25% QR payments.

Note that users who do not carry out QR payments are also affected. In Table 2 we show the distribution of the latencies of only the non-QR payments (handled concurrently with 10% QR payments). For 150 concurrent users the mean latency goes up from 4.9 to 474 ms, i.e. almost two orders of magnitude higher. Similar observations can be made when analyzing the 50th and 95th percentiles of the processing latencies.

Table 1
Quality of service results for one MobiCent node under increasing user load.

Users	Min (ms)	Max (ms)	Mean (ms)	Std. dev.	50th %ile	95th %ile
6	1	11	1.848	0.913	2	3
150	0	172	4.852	6.534	3	17
300	0	324	14.02	20.34	7	51
450	0	453	67.37	83.58	15	242
600	0	609	126.2	131.6	56	428
750	0	3426	177.6	167.1	117	474
900	0	15460	201.0	732.9	139	527

Table 2
Latency of regular transactions (concurrent with 10% QR payments).

Users	Min (ms)	Max (ms)	Mean (ms)	Std. Dev.	50th %ile	95th %ile
2	1	6	2.197	0.576	12.0	16.0
150	0	1545	474.0	421.2	518.5	1162.9
300	0	3385	911.1	881.9	720.0	2769.3

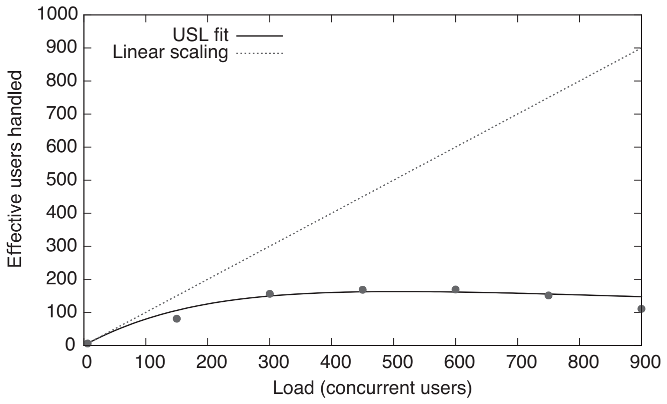

Fig. 10. Scalability assessment of a feature combination.

5.5. Selecting, testing and comparing feature combinations

In a last experiment, we illustrate selecting and assessing feature combinations. This experiment involves a complex scenario with online payments and local electronic wallets. Such offline account storage mechanisms usually eliminate the need to repeatedly enter identifying information into purchase forms, but impose particular security challenges, like *double spending* (Karame et al., 2012), i.e. the ability to spend the same digital currency twice. Without going into details, we implemented a solution that relies on digital signatures and a trusted third party.

In order to systematically assess and compare the scalability impact of features involving that scenario, we proceed as follows. First, we select the features essential to the scenario and enable them; this involves features *Mobile Wallet, Digital Cash, PIN, Online, Integrity* and *Non-Repudiation*. Second, we identify optional features that are commonly (but optionally) enabled in the scope of that scenario, i.e., *Availability* (i.e. with replication across all nodes), *QR Code* (10% of the transactions) and *Confidentiality* (i.e. with asymmetric encryption). Given that there are only 8 remaining feature combinations, all these can easily be assessed automatically by Scalar.

Each of the assessed feature combinations results in a fitted Universal Scalability Model, as illustrated in Fig. 10. Given the automatically calculated values for σ and κ, we can calculate the optimal load point p^*, as outlined in Section 4.4. Table 3 shows the results for a single node MobiCent deployment, ordered by p^*. The results clearly show the scalability impact of the optional features on the overall feature combinations assessed here.

Table 3
Selecting and comparing the results of different feature sets. The p^* value is calculated as per Section 2.1, and denotes the maximum number of concurrent users sustainable with that feature configuration.

Availability	QR Code	Confidentiality	p^* (users)
			1231
		×	986
×			352
×		×	295
	×		164
	×	×	146
×	×		68
×	×	×	52

Note that the constraint on feature combinations regarding online wallets and avoiding double spending significantly reduced the number of remaining feature combinations, up to the point where they can still be exhaustively assessed. Were this not the case, then we can fall back to assessing arbitrary feature combinations that uphold the given constraints, and construct a model to predict p^* for missing feature combinations.

5.6. Predicting the scalability of new feature combinations with automated feature selection

As the number of possible feature combinations can run in the thousands, exhaustively testing all feature combinations of a system can be prohibitively expensive, especially when many feature configurations hardly have any impact on its overall performance or scalability. While in some cases usage data is available on what feature combinations are often selected by tenants (and thereby greatly reducing the number of interesting feature combinations to test), this is not always so; in that case, automated feature selection and regression strategies can be of help. In this section, we discuss how we address the challenge of automatically selecting new feature configurations for scalability testing based on previously obtained performance results of existing feature combination benchmarks. In addition, we elaborate on how we extend our approach when we add new features to an existing service. Finally, we compare our approach with related work by Guo et al. (2013).

We illustrate our approach with a new logical feature set that combines individual feature configurations that were already benchmarked previously. The new configuration involves 5% QR code transactions on a three node MobiCent deployment with sticky load balancing and 25 tenants. The individual features have

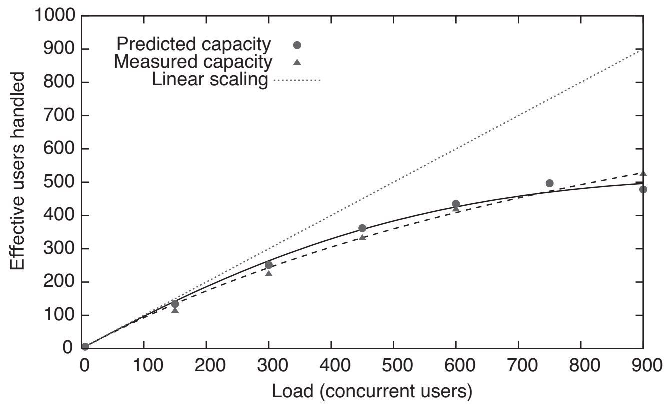

Fig. 11. Predicted versus measured capacity.

been tested in Figs. 7–9, albeit for different feature values for the number of tenants and the ratio of QR code payment transactions.

To estimate the capacity and scalability of a new feature combination upfront, we leverage the linear regression and random forest machine learning algorithms of Weka (Hall et al., 2009) to create a capacity prediction model of MobiCent. We use the performance results of the previous feature configurations as training data. We do not use the individual data points as input, but use the derived Universal Scalability Models for these feature configurations as input. As a result, the amount of data to train on is significantly reduced, and learning the two regression models with Weka takes less than a second. The Weka ARFF training set has less than 500 data points and looks as follows:

```
@attribute nodes numeric
@attribute store {hashmap,hazelcast,h2}
@attribute qr number
@attribute auth {none,payer,receiver,all}
@attribute signature {none,payer,receiver,all}
@attribute load numeric
@attribute capacity numeric

@data
1,hashmap,0,none,none,6,5.6
1,hashmap,0,none,none,150,134.6
1,hashmap,0,none,none,300,257.9
3,hashmap,0,none,none,750,537.2
3,hashmap,0,none,none,900,574.6
...
```

For testing the regression models, we compare the predicted capacity results of these models with the actually measured scalability results for a growing number of concurrent users. The relative error on the predicted capacity of the new feature combination is 22% for linear regression, and 5% for random forest regression. Fig. 11 compares the random forest predicted model against the measured results.

Although the quality of prediction with linear regression is worse, the learning algorithm is still useful in that it offers us a capacity prediction model C (in terms of effective users handled) based on a linear combination of the capacity of different feature configurations f_i and a fixed cost c:

$$C = w_1 * f_1 + w_2 * f_2 + \ldots + w_N * f_N + c \qquad w_i \geq 0$$

Even though the linear regression model has a lower accuracy, it is still useful as the feature configuration weights w_i give us insights into how every feature f_i impacts the overall capacity C of the new service variant. This teaches us that the most contributing features in this experiment are (1) the number of MobiCent nodes and (2) whether replication is used or not. The weights of the number of concurrent users and amount of QR code payments

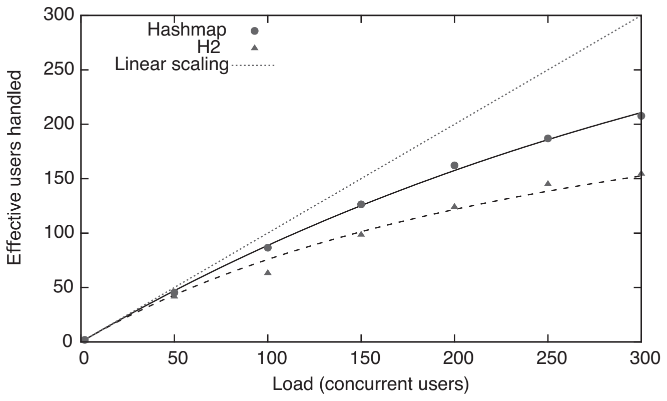

Fig. 12. Capacity of a single MobiCent node with H2 persistent storage.

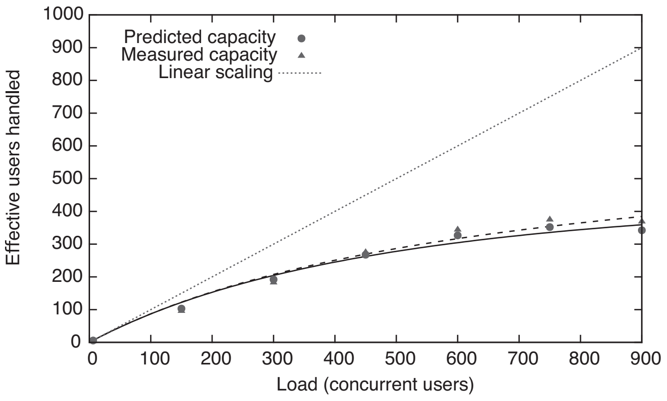

Fig. 13. Predicted versus measured capacity for H2 enabled deployments.

are almost an order of magnitude smaller, and the number of tenants does not have any impact (i.e. has no significant weight w_i), as already established in Section 5.2.

The advantage of this approach is that each new set of capacity measurements can be added to the training data to further improve the accuracy of the performance prediction models. This is particularly important when we add new features to our MobiCent service. To illustrate the approach, we add a new persistent storage feature to log all transactions in a H2 relational database management system. We first benchmark the impact of the new feature on a base configuration of the service: Fig. 12 compares the capacity of a single MobiCent with H2 persistent storage against a similar setup that uses a simple in-memory hash map.

To see the impact of this new feature on more sophisticated feature selections, we predict the performance of a MobiCent deployment on 3 nodes with sticky load balancing, 100 tenants, H2 persistent storage (on all 3 nodes), and 5% QR code transactions. Fig. 13 depicts the capacity results predicted using the random forest regression model versus the measured results. The predicted results are based on the model trained on data from the capacity results of Figs. 11 and 12. This is a fairly good prediction, which can be explained by this new feature being mainly bound by disk input and output performance, while the other features are not disk intensive—the other features were mainly bound by CPU and network performance, and as such their impact on the overall scalability was orthogonal to the H2 relational database management system feature.

When comparing our results with related work by Guo et al. (2013), we find some differences. Guo et al. use CART (classification and regression trees) to create a performance model, which inherently is a linear combination of the performance results of

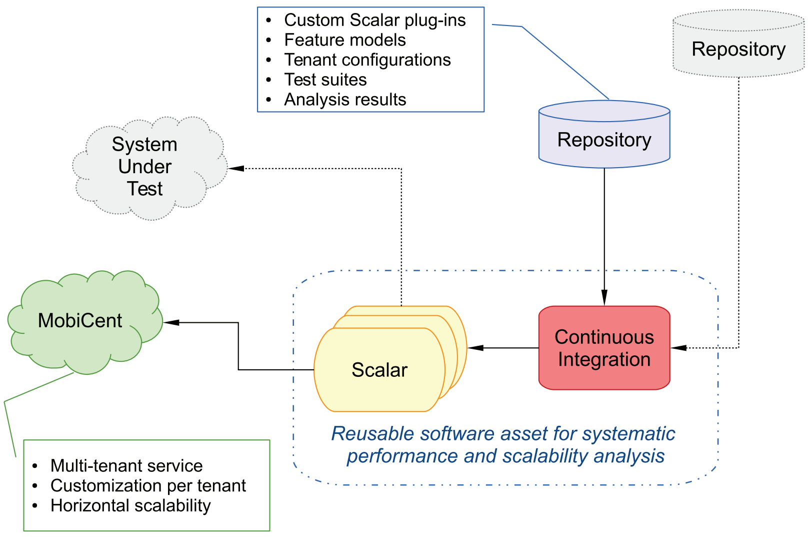

Fig. 14. High level overview of the scalability assessment process. A new project, represented by the grey elements, can be added transparently while reusing the existing scalability analysis infrastructure.

different feature configurations. The authors acknowledge that they experimented with two CART variants, random forests and boosting, but they observed similar prediction improvements compared to a simpler parameter tuning approach. However, in our experiments, there is a clear prediction performance difference (in terms of relative errors on the predicted capacities) between the linear regression and the random forest regression model. One of the reasons for the observed differences is that—contrary to the related work—our performance and capacity results are highly dependent on the number of concurrent users, which is not the case in the work of Guo et al. Our approach builds upon Guo's technique by not combining the performance results of feature configurations for a fixed set of users, but rather by combining scalability prediction functions in terms of concurrent users, i.e., the Universal Scalability Models of different feature combinations.

6. Discussion

The goal of this work is to investigate whether it is possible to systematically assess the scalability of large, complex multi-tenant systems in a practical and holistic way. In the previous section, we have established that it is indeed possible to technically measure and assess the impact of specific feature combinations on the scalability of the system as a whole, by means of novel tool support and constructively leveraging existing research results. In this section, we go into more detail on the practical and usability aspects of the proposed approach.

The experiments show that the approach is able to establish a scalability baseline against which future changes (i.e. new features, as well as bug fixes and updates of existing features) can be compared. Combined with the fact that once initial setup tasks have been performed, the approach is fully automated, this implies that the approach can be for scalability what unit testing is for functionality. By integrating it in a continuous integration process, developers could be warned automatically when a code commit results in a scalability penalty of the system. As a proof of concept, we have integrated the approach in a Jenkins continuous integration server, which automatically executes a set of scalability experiments whenever a subversion repository is updated. Our continuous integration setup is shown in Fig. 14.

Note that deploying Scalar is straightforward. In practice, creating user and request types is trivial when test code (e.g., in the form of unit tests) is available—in that sense, generating traffic by leveraging manually created user code, is actually more straightforward than having to deploy monitoring infrastructure and collect live traffic for later replay. Scalar has already been deployed successfully by multiple independent parties; on average, it takes an hour to port existing test code to Scalar, and get the Scalar infrastructure up and running.

In addition, the scalability analysis infrastructure and continuous integration environment are reusable assets, and can be shared between different projects to minimize operational cost. As shown in Fig. 14, as the continuous integration server and the Scalar cluster are completely domain independent, adding a new project is as easy as configuring the continuous integration server to use an additional repository, while reusing the scalability testing infrastructure.

The approach allows system administrators and maintainers improve system health by studying which tenants are better grouped together on a server. As shown in Section 5.2, in the MobiCent case, the number of tenants has no impact on system performance and scalability. As scalability is determined only by the features selected by the tenants present in the system, that information offers a compelling argument to system administrators for grouping tenants per selected features. Conversely, features that have a disproportionately negative impact on overall scalability can be separated and deployed in isolation. In the case of QR payments as documented in Section 5.4, system maintainers can consider deploying the QR code functionality on a separate machine, so that resources can be tuned appropriately (e.g. allocate more bandwidth for the QR code machine).

In the domain of reliability and fault tolerance, the proposed approach can offer a stepping stone towards better reliability testing through fault injection, as it (1) can provide hints at potentially unreliable feature interactions, and (2) is able to systematically recreate high load situations with specific traffic composition patterns and service configurations, which in turn improves testability.

Finally, the approach helps in future proofing a service, as it is able to give strategic insights into the scalability potential of the production environment, and provides key indicators for monitoring the production environment. On one hand, determining the scalability potential of the production environment allows to answer questions such as “How many additional users can the current environment handle?”, “How much capacity would we gain by deploying an additional instance?, and “How much more can we scale linearly?”. On the other hand, the approach also hints at critical elements to consider to achieve true performance isolation per user.

7. Conclusion

We have presented a novel methodology and tool support that combines elements from existing research to assess the impact of feature combinations on the scalability of multi-tenant feature oriented software services. The framework is reusable, and after a one time setup, assessing the scalability impact of arbitrary feature combinations is fully automated. Additionally, we have applied this methodology to a case study in the domain of electronic payments. Experiments show that this case did indeed contain unanticipated feature interactions on overall scalability, which have subsequently been uncovered. We conclude that it is possible to systematically integrate scalability testing into a continuous integration process which allows early detection of unanticipated and unwanted feature interactions.

Acknowledgments

This research is partially funded by the Research Fund KU Leuven.

References

Abel, A., Benz, F., Doerfert, J., Dörr, B., Hahn, S., Haupenthal, F., Jacobs, M., Moin, A., Reineke, J., Schommer, B., Wilhelm, R., 2013. Impact of resource sharing on performance and performance prediction: A survey. In: D'Argenio, P., Melgratti, H. (Eds.), CONCUR 2013 - Concurrency Theory. In: Lecture Notes in Computer Science, Vol. 8052. Springer Berlin Heidelberg, pp. 25–43. doi:10.1007/978-3-642-40184-8_3.

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities. In: American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Spring Joint Computer Conference, April 18-20, 1967, Atlantic City, New Jersey, USA, pp. 483–485. doi:10.1145/1465482.1465560.

Apel, S., Batory, D., Kstner, C., Saake, G., 2013. Feature-Oriented Software Product Lines: Concepts and Implementation. Springer Publishing Company, Incorporated.

Bak, K., Czarnecki, K., Wasowski, A., 2011. Feature and meta-models in clafer: mixed, specialized, and coupled. In: Proceedings of the Third International Conference on Software Language Engineering. Springer-Verlag, Berlin, Heidelberg, pp. 102–122.

Balsamo, S., Di Marco, A., Inverardi, P., Simeoni, M., 2004. Model-based performance prediction in software development: a survey. IEEE Trans. Softw. Eng. 30 (5), 295–310. doi:10.1109/TSE.2004.9.

Carvalho, N.A., Pereira, J., 2010. Measuring software systems scalability for proactive data center management. In: On the Move to Meaningful Internet Systems, OTM 2010. Springer, pp. 829–842.

Clements, P., Northrop, L., 2001. Software product lines: practices and patterns. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Czarnecki, K., Helsen, S., Eisenecker, U.W., 2004. Staged configuration using feature models. In: Nord, R.L. (Ed.), SPLC. Springer, pp. 266–283.

Dayarathna, M., Suzumura, T., 2013. A performance analysis of System S, S4, and Esper via two level benchmarking. In: Quantitative Evaluation of Systems. Springer, pp. 225–240.

Galster, M., Avgeriou, P., Tofan, D., 2013. Constraints for the design of variability-intensive service-oriented reference architectures - an industrial case study. Inf. Softw. Technol. 55 (2), 428–441. doi:10.1016/j.infsof.2012.09.011.

Gatling, Gatling Stress Tool. http://gatling-tool.org/ (accessed 28.12.15).

Ghosh, A., Wolter, D.R., Andrews, J.G., Chen, R., 2005. Broadband wireless access with wimax/802.16: current performance benchmarks and future potential. Commun. Mag. IEEE 43 (2), 129–136.

Gross, D., Shortle, J.F., Thompson, J.M., Harris, C.M., 2013. Fundamentals of queueing theory. John Wiley & Sons.

Gunther, N.J., 1993. A simple capacity model of massively parallel transaction systems. In: CMG-CONFERENCE-. COMPSCER MEASUREMENT GROUP INC. 1035–1035

Gunther, N.J., 2007. Guerrilla capacity planning - a tactical approach to planning for highly scalable applications and services.. Springer.

Guo, J., Czarnecki, K., Apel, S., Siegmund, N., Wasowski, A., 2013. Variability-aware performance prediction: a statistical learning approach. In: Denney, E., Bultan, T., Zeller, A. (Eds.), Proceedings of 2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013. IEEE, pp. 301–311. doi:10.1109/ASE.2013.6693089.

Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B., 2001. Mibench: a free, commercially representative embedded benchmark suite. In: Proceedings of International Workshop on Workload Characterization, 2001. WWC-4. 2001. IEEE, pp. 3–14.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H., 2009. The weka data mining software: an update. SIGKDD Explor. Newsl. 11 (1), 10–18. doi:10.1145/1656274.1656278.

Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K., 2008. Dynamic software product lines. Computer 41 (4), 93–95. doi:10.1109/MC.2008.123.

Heyman, T., Preuveneers, D., Joosen, W., 2014. Scalar: a distributed scalability analysis framework. In: Norman, G., Sanders, W. (Eds.), Quantitative Evaluation of Systems. In: Lecture Notes in Computer Science, Vol. 8657. Springer International Publishing, pp. 241–244. doi:10.1007/978-3-319-10696-0_19.

Heyman, T., Preuveneers, D., Joosen, W., 2014. Scalar: a distributed scalability analysis framework. In: Norman, G., Sanders, W. (Eds.), Quantitative Evaluation of Systems. In: Lecture Notes in Computer Science, Vol. 8657. Springer International Publishing, pp. 241–244. doi:10.1007/978-3-319-10696-0_19.

Hinchey, M., Park, S., Schmid, K., 2012. Building dynamic software product lines. Computer 45 (10), 22–26. http://doi.ieeecomputersociety.org/10.1109/MC.2012.332.

Istoan, P., Nain, G., Perrouin, G., Jezequel, J.-M., 2009. Dynamic software product lines for service-based systems. In: Proceedings of the 2009 Ninth IEEE International Conference on Computer and Information Technology - Volume 02, pp. 193–198. doi:10.1109/CIT.2009.54. Washington, DC, USA

Karame, G.O., Androulaki, E., Capkun, S., 2012. Double-spending fast payments in bitcoin. In: Proceedings of the 2012 ACM Conference on Computer and Communications Security. ACM, New York, NY, USA, pp. 906–917. doi:10.1145/2382196.2382292.

Krut, R.W., Cohen, S.G., 2009. Service-oriented architectures and software product lines: Enhancing variation. In: Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University, Pittsburgh, PA, USA, pp. 301–302.

Lee, J., Kotonya, G., 2010. Combining service-orientation with product line engineering. IEEE Softw. 27 (3), 35–41. doi:10.1109/MS.2010.30.

Mahdavi-Hezavehi, S., Galster, M., Avgeriou, P., 2013. Variability in quality attributes of service-based software systems: A systematic literature review. Inf. Softw. Technol. 55 (2), 320–343. doi:10.1016/j.infsof.2012.08.010.

Mietzner, R., Metzger, A., Leymann, F., Pohl, K., 2009. Variability modeling to support customization and deployment of multi-tenant-aware software as a service applications. In: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems. IEEE Computer Society, Washington, DC, USA, pp. 18–25. doi:10.1109/PESOS.2009.5068815.

Nudd, G.R., Kerbyson, D.J., Papaefstathiou, E., Perry, S.C., Harper, J.S., Wilcox, D.V., 2000. Pace–a toolset for the performance prediction of parallel and distributed systems. Int. J. High Perform. Comput. Appl. 14 (3), 228–251. doi:10.1177/109434200001400306.

Philip Aston, The Grinder. http://grinder.sourceforge.net/ (accessed 28.12.15).

Rashwan, A., Taha, A., Hassanein, H.S., 2012. Benchmarking message authentication code functions for mobile computing. In: Proceedings of Global Communications Conference (GLOBECOM), 2012 IEEE. IEEE, pp. 2585–2590.

Rosenmüller, M., Siegmund, N., Pukall, M., Apel, S., 2011. Tailoring dynamic software product lines. In: Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering. ACM, New York, NY, USA, pp. 3–12. doi:10.1145/2047862.2047866.

Selenium. Selenium - browser automation. http://docs.seleniumhq.org/ (accessed 28.12.15).

Shen, L., Peng, X., Liu, J., Zhao, W., 2011. Towards feature-oriented variability reconfiguration in dynamic software product lines. In: Proceedings of the 12th International Conference on Top Productivity Through Software Reuse. Springer-Verlag, Berlin, Heidelberg, pp. 52–68.

Siegmund, N., Kolesnikov, S.S., Kästner, C., Apel, S., Batory, D., Rosenmüller, M., Saake, G., 2012. Predicting performance via automated feature-interaction detection. In: Proceedings of the 34th International Conference on Software Engineering. IEEE Press, Piscataway, NJ, USA, pp. 167–177.

Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R., Purkayastha, A., 2002. A framework for performance modeling and prediction. In: Proceedings of Supercomputing, ACM/IEEE 2002 Conference doi:10.1109/SC.2002.10004. 21–21

Thüm, T., Kästner, C., Benduhn, F., Meinicke, J., Saake, G., Leich, T., 2014. Featureide: an extensible framework for feature-oriented software development. Sci. Comput. Program. 79, 70–85. doi:10.1016/j.scico.2012.06.002.

The Apache Software Foundation, . Apache JMeter. http://jmeter.apache.org/ (accessed 28.12.15).

Uskov, A.V., 2012. Information security of ipsec-based mobile vpn: Authentication and encryption algorithms performance. In: Proceedings of the 11th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), 2012. IEEE, pp. 1042–1048.

Walraven, S., Van Landuyt, D., Truyen, E., Handekyn, K., Joosen, W., 2014. Efficient customization of multi-tenant software-as-a-service applications with service lines. J. Syst. Softw. 91, 48–62. doi:10.1016/j.jss.2014.01.021.

Zhang, Y., Zheng, Z., Lyu, M., 2014. An online performance prediction framework for service-oriented systems. IEEE Trans. Syst. Man, and Cybern.: Syst. 44 (9), 1169–1181. doi:10.1109/TSMC.2013.2297401.

Davy Preuveneers is a research expert at the University of Leuven-KU Leuven in the iMinds-DistriNet research group. He obtained a Ph.D. degree from KU Leuven in 2009. His research interests are in the field of scalable distributed

systems and event-based middleware, federated identity and access management, privacy, context-driven services, semantic modeling and reasoning, intelligent environments and mobile applications. He leads a team of researchers and projects on these topics with a particular focus on applications in the area of the Internet of Things, Cyber Physical Systems, media and e-health.

Thomas Heyman is a post-doctoral researcher at the DistriNet research group. His current research activities involve benchmarking and monitoring non-functional qualities of large-scale distributed deployments. His Ph.D. research focused on formally modeling software architectures, and semi-automatically analyzing their security requirements.

Yolande Berbers is a member of the research group DistriNet of the department of computer science of the KU Leuven. She leads a task force of 10 researchers. Her research interests include software engineering for embedded software, ubiquitous computing, context-aware and adaptive systems, service architectures, middleware, component-oriented software development, and distributed systems. She is currently vice-dean of the Faculty of Engineering of the University of Leuven. She has been program director for the degrees of bachelor and master in engineering: computer science for 9 years. She is mainstreaming promotor for the Faculty of Engineering.

Wouter Joosen is full professor in distributed software systems at the Department of Computer Science of KU Leuven, Belgium. He obtained a Ph.D. degree from KU Leuven in 1996. He has also co-founded spin-off companies of KU Leuven: Luciad, a company specializing in software components for Geographical Information Systems, and Ubizen (now part of Verizon Business Solutions), where he has been the CTO from 1996 till 2000, and COO from 2000 till 2002. His current research interests are in distributed systems and cloud computing, focusing on software architecture and adaptive middleware, as well as in security aspects of software.