Feature-based Variability Management for Scalable Enterprise Applications: Experiences with an E-Payment Case

Davy Preuveneers, Thomas Heyman, Yolande Berbers and Wouter Joosen
iMinds-DistriNet-KU Leuven, Belgium
{davy.preuveneers, thomas.heyman, yolande.berbers, wouter.joosen}@cs.kuleuven.be

Abstract

In today's world of electronic payments, which includes credit cards, gift cards, stored value accounts, and micro payments on mobile devices, successful payment services need to be sufficiently scalable and support diverging use cases to be successful. However, the interaction between horizontal scalability, feature variability and stringent performance and security concerns for these heterogeneous payment applications and stakeholders is not always clear, which creates tension (and sometimes interference) in the successful development of these services. In this work, we identify crucial non-functional requirements for payment services and present a simple framework to map these requirements to existing approaches to achieve both horizontal and vertical scalability. We report on experiences with applying a service line engineering (SLE) approach towards payment services that combines cloud computing and SPL to support feature customization in a multi tenant payment service. We conclude with practical experiences and several lessons learned after applying this SPL engineering methodology on our industrial e-payment use case.

Keywords

E-payment, software product lines, security, scalability, feature interaction, customization

1. Introduction

With smart phones gaining relevance as a payment vehicle and as a platform for mobile banking, the world of electronic payments is evolving quickly with mobile payments emerging as an alternative to the credit card/debit card ecosystem (e.g., [1], [2], [3]). Apart from simplified ease of use over the Internet, the main drivers are also the ability to (partially) circumvent the drawback of credit card transactions, where each stakeholder in the processing chain takes a percentage of the payment transaction as interchange fees. Given both ongoing technological advances on the client side and continuously evolving requirements, it is important that this increased complexity is anticipated early in the architectural design phase. Furthermore, to address the increased cost and effort in software development time, software reuse across various payment applications is key. The reusability of software can be optimized by applying methodologies such as software product line and service line engineering.

In order to address multiple, potentially conflicting requirements and increase software reuse, the Software Product Line Engineering (SPLE) methodology is a widely adopted approach to ensure that artifacts generated during product development can be reused in a related product [4], [5], [6], [7]. Variation points are typically bound during different stages of development, i.e., at design time, before the delivery of the software. Apart from design time variability, run time variability binding and reconfiguration for different stakeholders and applications play an important role in developing payment services, as happens for instance in certain service line engineering approaches [8]. By combining SOA [9] and SPLE, we apply and evaluate a feature based approach to create a software architecture for variability intensive service oriented payment applications.

Independent of how software reusability is maximised, the final software product should still uphold various non-functional requirements. Especially in the case of payment services, these software qualities are not trivial, but correctly realising these qualities is key to achieving a successful product:

- **Scalability**: by linking together distributed computing resources, cloud-based software solutions are explored for their capability to handle a continuously increasing number of (parallel) payment transactions.
- **Elasticity**: to handle peak loads without having to over-provision, the pay-per-use model of cloud computing [10] offers an interesting value proposition to dynamically scale resource availability in line with the variability in electronic payment transactions load.
- **Heterogeneity**: the payment ecosystem is characterized by different offline and online solutions for macro- and micropayments, ranging from credit cards, prepaid cards, vouchers, SMS- or barcode-based [11] solutions, mobile applications, etc.
- **Security**: the inherent sensitive nature of payment transactions demands for strict security mechanisms, to defend against fraudulent transactions (i.e. integrity, authorization, authentication, confidentiality, and non-repudiation of transactions).

The key research question that this paper investigates, is how all these methodologies interact during the design and development of a payment service, what value they add to

1530-1605/16 $31.00 © 2016 IEEE
DOI 10.1109/HICSS.2016.717

the final system, and whether their application does indeed result in an acceptable system that fulfills all its non-functional requirements. In order to answer this question, we present an industrial case on developing a multi-tenant domain specific software architecture for payment products supporting tenants with different feature requests on top of a single payment platform instance. This platform is being worked out in collaboration with a company specializing in developing payment software. To support runtime customization per tenant, we apply Dynamic SPL [12] for payment software where variation points are related to dynamic properties bound per tenant at runtime. The goal is to end up with a payment platform that is scalable, elastic, heterogeneous, and offers support for various security mechanisms.

The remainder of this paper is structured as follows. After reviewing related work in section 2, we carry out a domain analysis on payment systems in section 3 and present a framework that helps in systematically integrating all the various development facets into a successful software service in section 4. In section 5, we analyze the feasibility and performance of our approach, and report on lessons learned. We conclude in section 6 summarizing the main insights and identifying possible topics for future work.

2. Related work

Software product lines (SPL) [13] 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 Engineering Institute (SEI) [14] has been a leader in developing a body of knowledge and a set of standard models for software product lines. They offer an extensive framework to put Software Product Line in practice[1]. Many other SPL design methods and methodologies to manage variabilities in software product lines [15] have been proposed over the past two decades with applications in different industrial contexts, including FODA [16], FORM [17], FAST [18], PuLSE [19], COPA [20], KobrA [21] and QADA [22]. A detailed discussion of these methods would be beyond the scope of this document. We refer the interested reader to surveys and comparative studies in this field [23].

Feature modeling [24] is a 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 [25] 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, and has been applied on an e-commerce platform. Feature models have also been applied for variability reconfiguration in Dynamic Software

1. http://www.sei.cmu.edu/productlines/frame_report/

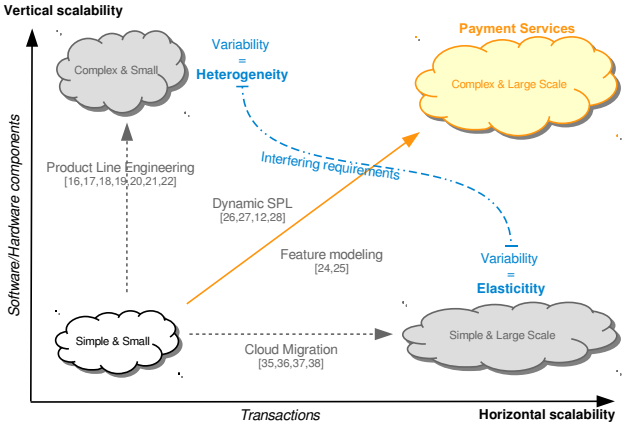

Fig. 1. Combining SPL with cloud computing.

Product Lines (DSPL). DSPLs [26], [27], [12], [28] 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 [29], [30].

While most of the work on DSPLs focuses on reconfiguration based on changing functional requirements, this work aims at adapting enterprise-level applications to different stakeholder or tenant needs, while focusing on satisfying non-functional requirements, such as scalability and security [31]. Cloud computing is known for its *horizontal scalability* and elasticity. However, payment systems are also strongly characterized by a great heterogeneity in the different payment technologies. This requires support for *vertical scalability* or scaling up, where new software components supporting new types of payment have to be added to the same overall payment solution. Fig. 1 depicts the tension created by the interaction between horizontal and vertical scalability to support multiple tenants with different feature requirements and the growing heterogeneity of payment technologies.

In [32] and [33], different case studies were presented on how software product lines have been applied in the e-commerce domain. The focus of those works is on variability management at design time, investigating traceability between the features and the architectural model, the modular structure of the architecture, and minimizing feature scattering. Those works do not address product line driven customization at runtime. In [34], Ruehl et al. have shown how they applied SPL to create customizable software-as-a-service applications. The authors also highlight flexibility of multi-tenant applications as a key challenge. However, compared to our work, they did not offer any experimental evaluation of their solution.

For achieving horizontal scalability, related work exists on how systems have successfully been migrated to a cloud environment. Tran et al. [35] study the migration of an application to a cloud platform, with a specific focus on overall cost. They also provide a taxonomy of migration

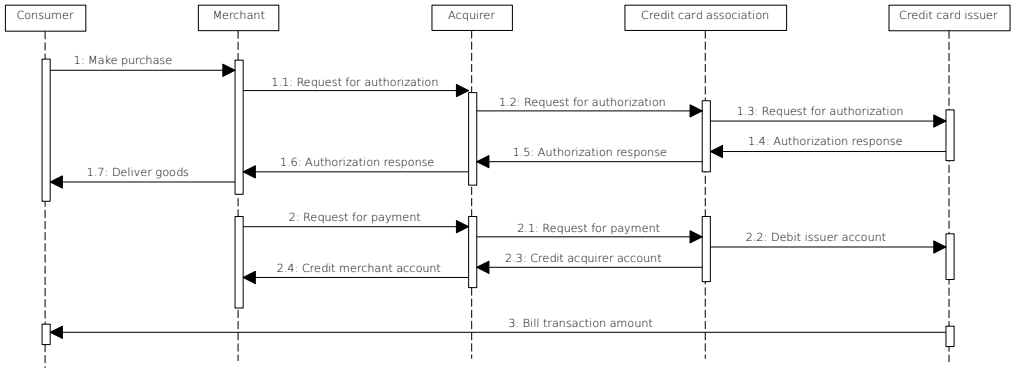

Fig. 2. Different stages between stakeholders in a typical credit card transaction. Other flows not depicted here are cancellations, chargebacks or refunds due to processing errors, insufficient funds, or authorization errors.

tasks. Kabbedijk et al. [36] present three architectural patterns that support variability in a multi-tenant cloud environment and can help during cloud migrations. Kousiouris et al. [37] document the challenges of migrating legacy applications to a cloud environment, and summarize enabling techniques and technologies that help in this respect. Maenhaut et al. [38] also document the migration of a healthcare application to a cloud environment. They conclude that while it can be fairly easy to perform a straightforward cloud migration, the benefits gained are also limited—in order to achieve maximal benefits from a cloud migration, the application should be modified to embrace a multi-tenant architecture.

For payment software solutions, security is a key concern. To protect credit card holders against misuse of their personal information, several security standards have been proposed. The Payment Card Industry Data Security Standard (PCI DSS) [39], [31] is a widely accepted set of international security policies and procedures for credit, debit and cash card transactions. The PCI DSS was developed by Visa and the founding payment brands of the PCI Security Standards Council to help facilitate the broad adoption of consistent data security measures on a global basis. The Payment Application Data Security Standard (PA-DSS) is a set of requirements that are intended to help software vendors develop secure payment applications that support PCI DSS compliance. PA-DSS applies to third-party applications that store, process or transmit payment card holder data. These regulations impose additional complexity as every change or update to the payment software should never give rise to a violation against these security standards. The necessity of always being compliant imposes stringent constraints for the development approach and software life cycle of the payment software architecture.

3. Domain analysis

In this section, we analyze the payment domain to investigate relevant requirements for payment platforms. In general, the payment domain is characterized by a broad variability in functional and non-functional requirements due to the rapid technology evolution and transaction growth:

- **Context of use**: Banking, shopping on the Internet, interact with unmanned point-of-sale (POS) systems, using staffed POS systems.
- **Payment method**: cash, credit or debit card, gift cards, coupon, voucher, stored value card, e-money (virtual and digital currency), carrier billing.
- **Customer payment device**: Credit card and POS terminal, desktop or laptop, smartphone with contactless payment system (SMS, QR code or NFC).
- **Authentication method**: Credit card with Personal Identification Number (PIN), username and password, security token (e.g. Digipass).

This section reviews key characteristic features of widely used and emerging payment systems for online e-commerce transactions and sketch the context in which they are used. We start from the classical credit card scenario, go over more recent trends of digital wallets and mobile payments, and finish with payment-as-a-service.

3.1. Credit cards

Credit card payments are still the most common type of payment. A credit card transaction happens in two stages: the *credit card authorization* and the *credit card clearing and settlement* phases. Different fees are incurred at each stage, and a (partial) failure can result in increased costs and/or credit card sales not being deposited. The typical flow of a credit card transaction only takes about a couple of seconds to be approved and completed, and requires interaction between various stakeholders, as depicted in Fig. 2. The flow resembles the 3-D Secure transaction protocol, a standard for authenticated payment [40].

3.2. Digital wallets and mobile payments

A digital wallet (or e-wallet) is a piece of software that stores credit card and other information to facilitate payments

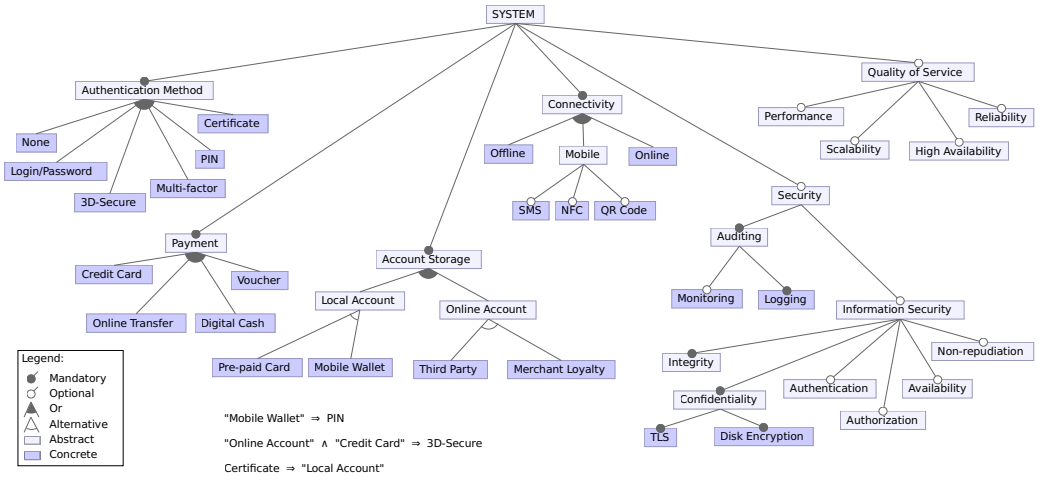

Fig. 3. High-level overview of the payment service feature model.

for goods or services on the web. It offers a secure alternative for the traditional leather wallet, and often takes the form of an application on your smartphone. It is based on encryption software that offers the benefit to consumers of convenient secure monetary transactions, while offering merchants better protection against fraud and the ability to sell products faster. Furthermore, digital wallets can also be used to store gift cards, discounts, coupons, etc.

Digital wallets can be classified into two categories: *client-side* wallets and *server-side* wallets. Client side wallets are maintained by the end-user with a locally installed application that stores pertinent payment and shipping information. When buying goods online, the digital wallet software completes most of the basic information automatically. With server side wallets, such as V.me by Visa[2] or MasterPass by MasterCard[3], all of the wallet data is stored on the company's servers rather than on a local harddrive. When checking out, the end-user has to enter his credentials (username and password) to complete the order. Not having to enter credit card details, the end-user has the benefit that he is protected against any security risks when sharing sensitive information. The benefit for merchants of server side wallets (compared to client side ones) is that the former have a greater degree of standardization. Other examples of web service based digital wallets include PayPal, Yahoo! Wallet, Amazon Payments, Google Wallet, etc.

3.3. Payment-as-a-Service

The POS is the location where the payment transaction between the customer and the merchant is taking place. Traditional or standard POS terminals use phone lines to communicate with the payment processor which requires merchants to either lose their line while the payment transactions are being processed or get a second one. Virtual POS solutions can turn a regular computer into a point-of-sale (POS) terminal by simply adding a USB swipe card reader and using an existing Internet connection – eliminating the need for additional phone lines – to securely transmit sensitive data over a Secure Sockets Layer (SSL) connection. Payment Service Providers (PSP) offer such solutions to credit card accepting merchants to authorize credit card transactions. In fact, the latest PCI DSS v3.1 standard argues that SSL and early versions of the TLS protocol are no longer considered sufficiently secure, and recommends to migrate towards the TLS v1.2 protocol[4].

The rise of cloud computing created the opportunity to have POS systems deployed as software-as-a-service (SaaS), i.e. as a single instance of a service that can serve multiple merchants. This way, POS solutions can be directly accessed over the Internet within a browser environment. An additional advantage is that cloud-based POS systems are independent from the underlying hardware platform and to a lesser extend from operating system limitations. As the POS system is not run locally, there is no on-site installation required. SaaS-based payment solutions are agile, configurable, scalable, and loosely coupled, and can be offered as a single technical gateway for the merchants. These POS services can be customized according to the preferred payment methods or the merchant's needs. Such services rely on metadata to be configured [41]. Offering a centralized SaaS-based solution that supports the different requirements of the merchants through metadata-based configuration is known as *multi-tenancy* [42], [43]. A final advantage of cloud-based POS solutions compared to traditional in-store systems is that they can be used on mobile platforms as well.

4. Feature-based variability

We adopt a *service line engineering* [8] method which combines the benefits of *software product line engineering*

2. https://eu.v.me
3. https://masterpass.com

4. https://www.pcisecuritystandards.org/documents/Migrating_from_SSL_Early_TLS_InformationSupplement_v1.pdf

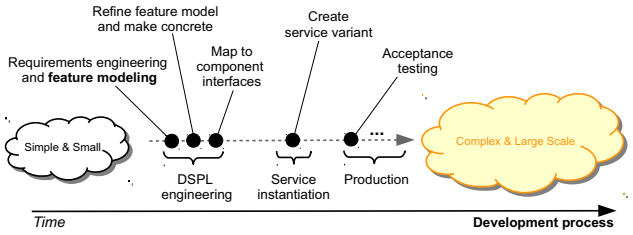

Fig. 4. Creating a complex large scale payment platform.

and those of multi-tenancy so that SaaS applications can be efficiently customized and tailored to tenant specific requirements. The service line engineering methodology starts with a domain analysis (see section 3) and a variability analysis, resulting in a feature-based variability model with the explicit representation of non-functional requirements. The next step is the design and development of the service line architecture. A high level overview of the methodology is shown in Fig. 4, and will be further illustrated in the following sections.

4.1. The payment service feature model

We first identity the scope of the payment product line by defining the mandatory and optional features of products derived from this software product line in a feature model (see Fig. 3, designed with FeatureIDE[5]). The reason for choosing feature modeling is because the simple, hierarchical models that capture the commonality and variability are easy to understand. Feature models provide a generic way to represent variability in the product line that is independent of any specific application domain or software technology.

A payment transaction involves collaboration between 2 parties, the issuing and acquiring party, and one or more trusted third parties to verify the identifies and the validity of the payment transaction. The acquiring party is the stakeholder acquiring money at the end of the transaction (e.g. a merchant), whereas the issuing party sends money (e.g. a customer). The transfer could be an online transfer of a local currency, or someone redeeming a voucher or gift card, etc. Payment transactions can take place between multiple stakeholders. Beyond the regular payment flow, there may also be cancellations, chargebacks or refunds to the customer due to processing errors, insufficient funds, incorrect billing, no or expired authorization, etc. A transfer of one account to another involves multiple changes (debiting and crediting), but is seen as a single transaction. The *Atomicity, Consistency, Isolation, Durability (ACID)* properties guarantee that the transaction is processed reliably. For payment transactions, *idempotence* guarantees that in case of a system or network failure, an attempt to reissue the request (or the reversal after a timeout) does not lead to unpredictable behavior (e.g. duplicate charges).

5. http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

When a transaction is completed successfully, the accounts of both entities are adjusted accordingly. For non-cash payment transactions, some intermediary parties are involved to confirm the identity of the customer and approve the transaction. Offline account storage mechanisms, such as electronic wallets, usually eliminate the need to repeatedly enter identifying information into purchase forms, but encounter other security challenges, like *double spending* [44].

4.2. Non-functional eature variability

Electronic payments involving multiple (wireless) networks present various security challenges:

- **Confidentiality**: Only those stakeholders involved in a payment transaction should know what was purchased and the payment mode used.
- **Authentication**: The merchant or customer must be able to trust claimed identities in the transaction.
- **Integrity**: Unauthorized parties should be prevented from being able to modify the transaction data.
- **Authorization**: Procedures must be in place to verify that the customer can make the requested purchase.
- **Non-repudiation**: Procedures should be in place to prevent against any single party from reneging on an commitment after the fact.

Note that these security features are not mandatory. For example, authentication of the customer may not be required for vouchers, whereas for credit card payments it is.

Security, as a software quality, cannot in general be directly mapped to features, as feature oriented software development typically considers functional aspects. However, while security is often introduced as a non-functional requirement early on in the development process, it needs to be operationalized and translated into specific responsibilities that can be assigned to (and implemented by) individual components (e.g., via the KAOS approach by Van Lamsweerde et al. [45]). Once this is done, the resulting security functionality is not inherently different from that of a functional requirement. As an illustration, consider the abstract confidentiality feature from Fig. 3. As a non-functional requirement, it is not straightforward to simply instantiate 'confidentiality'. However, confidentiality can be refined into concrete features of 'TLS' and 'Disk encryption', which can be instantiated.

Once a feature tree is sufficiently refined so that all branches are concrete, implementable features that can be assigned to individual components, we can modify these software components to expose this functionality via an interface. For instance, consider the case of confidentiality being refined into TLS and disk encryption: Implementing TSL implies that networked components are modified to use a TLS library. Similarly, components that write to and read from disk, can be modified to encrypt, resp. decrypt disk contents. These modifications can be exposed via an interface that allows users of these components to easily enable and disable TLS, and enable and disable disk encryption, given a key.

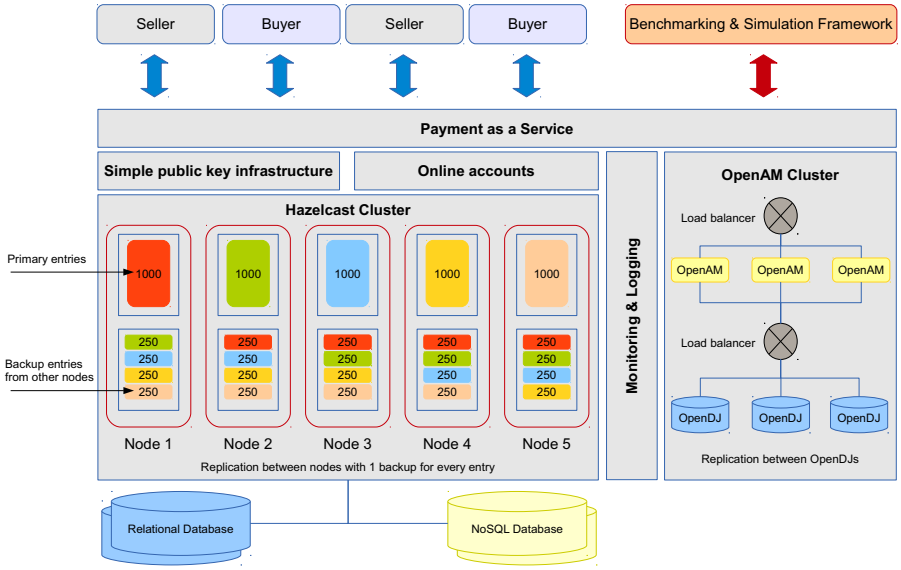

Fig. 5. Main components of the scalable payment service architecture.

4.3. Service variants through feature selection

Each and every feature in the feature model represents a desired characteristic of the system for one of the stakeholders. For the concrete realization of the application, a permissible selection of these features correspond to software building blocks that need to be composed into new product variants. These common reusable software building blocks are often known as core assets.

Once the desired features are specified, the feature model can be customized such that the requested features are included and the non relevant ones are left out. Instantiating a new payment service variant boils down to defining a fully concrete feature model with no points for further customization. This process results into what is known as a configuration. Any configuration can be checked against the feature model to determine whether it is a correct and complete configuration of a software product variant.

4.4. Implementation with reusable core assets

We have implemented an online RESTful payment system that allows the creation and consumption of monetary transactions as a service. It is partially inspired by the RIPPLE open source payment system[6], but serves a different purpose. Our system enables business owners to create electronic coupons that can later be redeemed by customers, 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. In order to achieve this, it is built on top of a public key infrastructure, coupled to an off-the-shelf user management and access control system[7]. To ensure scalability and availability, it uses a scalable distributed data store in the backend, which enables clustering of service instances, and transactions are automatically partitioned and replicated among the different service instances.

A high-level graphical overview of the service is shown in Fig. 5. It depicts a logical view on the payment service, the core assets and how they are linked with one another. The payment-as-a-service application is built on top of two main components, (1) a Hazelcast cluster, i.e. an efficient fault tolerant in memory data grid with support for distributed computing, caching with bindings for monitoring and persistent data stores, and (2) an OpenAM cluster used for identity and access management, including authentication and authorization. All core assets are exposed through RESTful services to simplify the mapping onto the feature model. The service offers a RESTful interface for 1) configuration management to customize the service deployment and the assets it provides, 2) user authentication to identify buyers and sellers using the identity and access management framework, 3) user provisioning for online bank accounts, 4) key pair management for signing and verifying digital monetary transactions, 5) transaction management with support for distributed locking and various replication modes, and 6) performance monitoring.

A seller creates a payment request, adds an authentication token, adds his certificate and digitally signs the request. This request is then uploaded to a payment service which uses the token to verify that the seller is authenticated, and that the payment request is valid. A URL to the payment request is returned, which can then be used by the buyer to consume the payment request and complete the transaction. Completing a transaction mirrors the payment request creation, but instead

6. https://ripple.com/build/rest-tool/

7. http://forgerock.com/products/open-identity-stack/

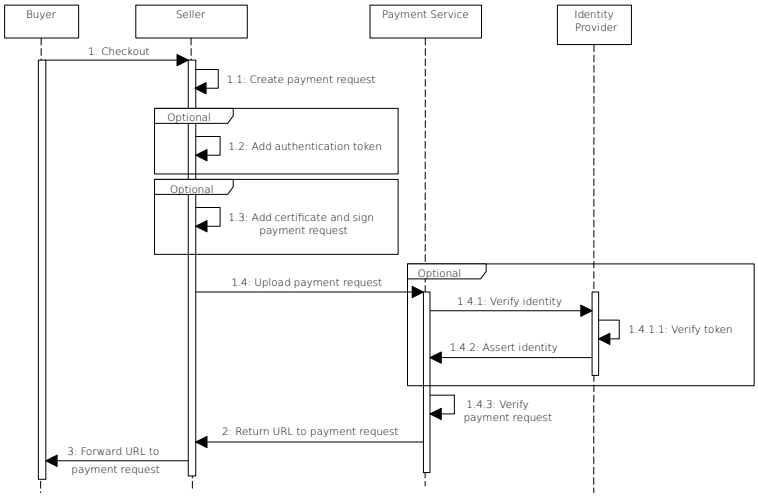

Fig. 6. Dynamic customization of a payment request workflow with optional features.

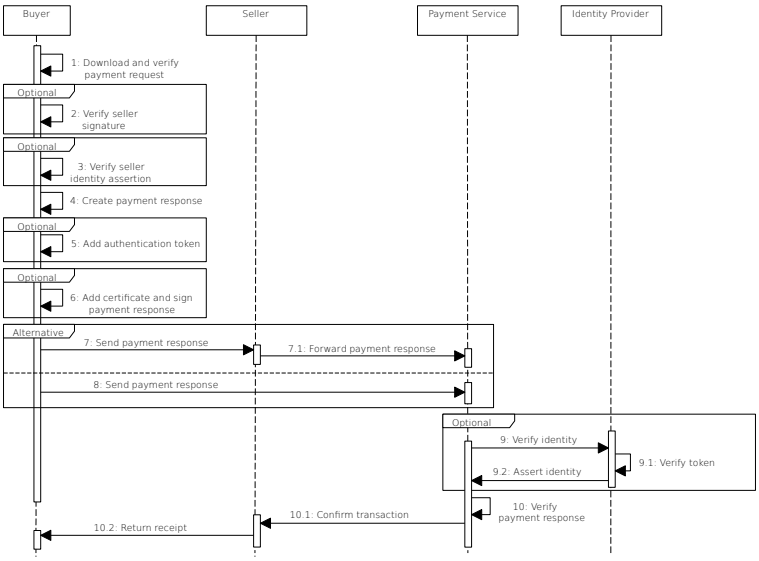

Fig. 7. Dynamic customization of a payment response workflow with optional features.

creates an authenticated response to the original request. A detailed discussion on the operation of this service is out of scope, but a running instance of this service with detailed documentation with sequence diagrams is shown in Fig. 6 and 7.

5. Evaluation

At this point, we have elicited requirements and translated them to a feature model, refined the feature model and mapped it to an implementation, and enabled easily selecting service variants through feature selection. As shown in Fig. 4, we are now in a position where a customer can easily instantiate a customized version of the payment platform. This section evaluates that process in two dimensions: 1) whether the runtime adaptation itself is scalable and does not negatively impact the quality of service of the resulting service, and 2) whether the process itself is sufficiently flexible. These topics are handled in section 5.1 and 5.2, respectively.

5.1. Evaluating non-functional requirements

Our experiments will investigate the impact of using feature models at runtime. Instantiations of these feature models define the configuration of each tenant of the payment service. Depending on the payment type (e.g., credit card, voucher, gift card, mobile payment), different security features are activated (e.g., authentication of the customer). Our exper-

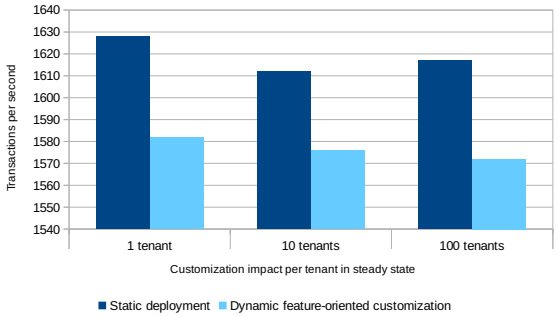

Fig. 8. Feature-oriented customization per tenant

(Note the scale: the Y-axis starts at 1540 transactions/sec)

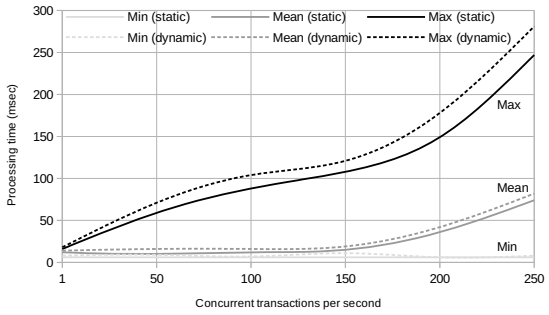

Fig. 9. Performance impact of per transaction

iments investigate the impact of (1) a growing number of tenants (e.g. merchants), and (2) a growing number of payment transactions. Rather than using a production environment for the systematic performance and scalability assessment, we carried out the evaluation in a controlled environment with reproducible workloads to better understand possible feature interactions that influence performance. The experimental setup consists of 10 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 5 machines to simulate event streams of a large user base. Each instance is deployed on an Apache Tomcat 8.0.18 application server on a 64-bit Ubuntu 14.04 system. To benchmark the scalability, the load N (in terms of transactions per second) is incremented on a fixed configuration.

In a first experiment, we show the impact of runtime feature oriented customization on the service. We present results for 1 up to 100 tenants (e.g. merchants) each selecting a random configuration from the feature model. We compare the performance where the configuration for the information security features is fixed at deployment time and tested individually per tenant, versus a multi-tenant scenario where these features are dynamically (re-) configured per tenant at runtime.

The configuration of each tenant is stored in a relational database (see Fig. 5). These configurations are loaded into memory and replicated with Hazelcast[8] to all nodes. At startup, we see a growing performance impact on the network for this initial replication. However, once the system is in steady state, we measure an average performance impact of 2-3% compared to a scenario where the configuration is fixed, and we measure no statistically significant difference between 10 or 100 tenants (see Fig. 8).

In a second experiment, we show the performance impact of runtime customization on the total time to handle a payment transaction. Fig. 9 shows the overall impact on the residence time, i.e., the end-to-end duration, of a transaction. For a static scenario, using dedicated instances with fixed configurations for 10 tenants, it depicts the minimum, mean and maximum time to complete a transaction for an increasing number of concurrent transactions per second. The mean value is compared with the processing times for a dynamic scenario using feature oriented customization per tenant. Here, we see an impact of maximum 10%.

In general, the previous results for the performance overhead of tenant specific runtime customization are considered acceptable for two reasons. First, these overheads are worst case scenarios when each transaction needs to be customized. Additionally, the overhead is predictable and can be anticipated in case of stringent QoS constraints.

5.2. Practical experiences and lessons learned

In this work, the objective was to investigate the interaction between horizontal scalability, feature variability and stringent performance and security concerns for different payment applications and stakeholders. In this section, we will briefly summarize some practical experiences and the main lessons learned.

5.2.1. Dynamic workflow customization. Payment transactions implement protocols with different degrees of complexity for both interoperability (e.g. protocol adapters to support different international payment standards) and different authentication and authorization requirements for micro payments, gift cards, mobile phone payments, etc. Managing the feature variability at this level not only requires proper management of selecting and mapping features to reusable software components, but also an explicit representation of how features can be mapped onto protocols or work flows (see Fig. 2, 6 and 7) that can be customized dynamically at runtime (e.g. by the tenants). A straightforward mapping of a single feature map onto a workflow is not feasible, as many features may interact with the same workflow and consistent behavior in the workflow must be guaranteed upon feature selection. Additionally, while some customizations are technically feasible, they sometimes do not make any sense from a security perspective. Therefore, it is advisable to constrain (or even predefine) the amount of configurations upfront.

5.2.2. Enhanced scalability assessment. We have analyzed the impact of feature based customization per tenant and the

8. http://hazelcast.org

performance overhead per tenant. Our starting assumption was that there was a fairly good isolation between tenants regarding the performance and latency of payments. However, we found that different feature set selections may trigger hidden or unexpected feature interactions causing SLA violations, but only under particular workloads (i.e. above a certain threshold of concurrent transactions).

To systematically compare the performance and scalability results of different configurations under different loads, we need better support for theoretical scalability models, and we intend to further explore the applicability of Gunther's Universal Scalability Law (USL) [46]. The Universal Scalability Law combines (a) the initial linear scalability of a system under increasing load, (b) the cost of sharing resources, (c) the diminishing returns due to contention, and (d) the negative returns from incoherency.

Unfortunately, predicting feature interactions is far from trivial as benchmarking all configurations upfront is not feasible. Further research is necessary to predict the scalability of a system given a scalability model (like the USL) derived from a limited set of feature selections.

5.2.3. Performance impact prediction of individual features on existing configurations. In this work, we focused on the scalability assessment of multi-tenant SaaS applications that can be customized following a service line engineering methodology. However, testing each and every feature combination is time consuming and not feasible due to the sheer amount of feature combinations. We found that it is very hard to automatically select a representative or most valuable set of feature combinations for load testing, such that we can carry out a comprehensive assessment of the overall scalability based on the testing results of a limit set of feature combinations.

This is especially important if we want to anticipate the performance impact when enabling a new or existing feature in an existing service configuration of a particular tenant. Although many features were tested individually, predicting the performance impact on a given feature selection was not straightforward. We are currently exploring to what extend methods such as those proposed by Guo et al. [47] can be applied to investigate performance prediction of 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. However, our case is more sophisticated as these hidden feature interactions can also occur within work flows and across tenants with different feature set combinations.

6. Conclusion

The use of a SPLE is well established for the development of applications that are characterized by a certain functional variability within the same product family. The key research question that we investigated in this paper is how software development methodologies like Dynamic SPL [12] and cloud deployment strategies interact during the design and development of software-as-a-service applications that can be customized at runtime per tenant. To answer this question, we presented an industry case on a multi-tenant domain specific software architecture for payment products. With our methodology, we can validate whether the application indeed results in an acceptable system that fulfills all its non-functional requirements. Experimental results in a multi-tenant scenario show that dynamic tenant based customization for security and scalability with feature models causes a minimal overhead of less than 10% on the actual payment transactions, and hardly any statistically relevant impact for a growing number of tenants. However, a limitation of our evaluation is that the experimentation was carried out in a controlled environment with reproducible workloads to better understand possible feature interactions that influence the performance. Further validation is necessary to ascertain whether we can generalize our conclusions to real life workloads and different tenant configurations.

Future work will evaluate our framework on more complex use cases with extra value added services that are more computationally intensive (e.g. for fraud detection). We will explore trade-offs between scale up (vertical scalability) and scale out (horizontal scalability). Other topics of interest are new strategies to automatically identify a minimum set of feature configurations to produce an accurate scalability characterization without testing all combinations exhaustively.

Acknowledgment

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

References

[1] E. Valcourt, J.-M. Robert, and F. Beaulieu, "Investigating mobile payment: supporting technologies, methods, and use," in *Wireless And Mobile Computing, Networking And Communications, 2005. (WiMob'2005), IEEE International Conference on*, vol. 4, 2005, pp. 29–36 Vol. 4.

[2] T. Dahlberg, N. Mallat, J. Ondrus, and A. Zmijewska, "Past, present and future of mobile payments research: A literature review," *Electronic Commerce Research and Applications*, vol. 7, no. 2, pp. 165 – 181, 2008, special Section: Research Advances for the Mobile Payments Arena.

[3] Y. B. Choi, R. L. Crowgey, J. M. Price, and J. S. VanPelt, "The state-of-the-art of mobile payment architecture and emerging issues," *Int. J. Electron. Financ.*, vol. 1, no. 1, pp. 94–103, Jan. 2006.

[4] R. W. Krut and S. G. Cohen, "Service-oriented architectures and software product lines: Enhancing variation," in *Proceedings of the 13th International Software Product Line Conference*, ser. SPLC '09. Pittsburgh, PA, USA: Carnegie Mellon University, 2009, pp. 301–302.

[5] J. Lee and G. Kotonya, "Combining service-orientation with product line engineering," *IEEE Softw.*, vol. 27, no. 3, pp. 35–41, May 2010.

[6] M. Galster, P. Avgeriou, and D. Tofan, "Constraints for the design of variability-intensive service-oriented reference architectures - an industrial case study," *Inf. Softw. Technol.*, vol. 55, no. 2, pp. 428–441, Feb. 2013.

[7] S. Mahdavi-Hezavehi, M. Galster, and P. Avgeriou, "Variability in quality attributes of service-based software systems: A systematic literature review," *Inf. Softw. Technol.*, vol. 55, no. 2, pp. 320–343, Feb. 2013.

[8] S. Walraven, D. Van Landuyt, E. Truyen, K. Handekyn, and W. Joosen, "Efficient customization of multi-tenant software-as-a-service applications with service lines," *J. Syst. Softw.*, vol. 91, pp. 48–62, May 2014.

[9] G. Lewis and D. Smith, "Service-oriented architecture and its implications for software maintenance and evolution," in *Frontiers of Software Maintenance, 2008. FoSM 2008.*, 2008, pp. 1–10.

[10] T. Dillon, C. Wu, and E. Chang, "Cloud computing: Issues and challenges," in *Advanced Information Networking and Applications (AINA), 2010 24th IEEE International Conference on*, 2010, pp. 27–33.

[11] J. Gao, V. Kulkarni, H. Ranavat, L. Chang, and H. Mei, "A 2d barcode-based mobile payment system," in *Multimedia and Ubiquitous Engineering, 2009. MUE '09. Third International Conference on*, 2009, pp. 320–329.

[12] M. Hinchey, S. Park, and K. Schmid, "Building dynamic software product lines," *Computer*, vol. 45, no. 10, pp. 22–26, 2012.

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

[14] L. M. Northrop, "Sei's software product line tenets," *IEEE Softw.*, vol. 19, no. 4, pp. 32–40, Jul. 2002.

[15] M. A. Babar, L. Chen, and F. Shull, "Managing variability in software product lines," *IEEE Software*, vol. 27, no. 3, pp. 89–91, 94, 2010.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, CMU, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[17] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh, "FORM: A feature-oriented reuse method with domain-specific reference architectures," *Annals of Software Engineering*, vol. 5, pp. 143–168, 1998.

[18] D. M. Weiss and C. T. R. Lai, *Software product-line engineering: a family-based software development process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1999.

[19] J. Bayer, O. Flege, P. Knauber, R. Laqua, D. Muthig, K. Schmid, T. Widen, and J. M. Debaud, "PuLSE: a methodology to develop software product lines," in *SSR '99: Proceedings of the 1999 symposium on Software reusability*. New York, NY, USA: ACM, 1999, pp. 122–131.

[20] P. America, J. H. Obbink, R. C. van Ommering, and F. van der Linden, "Copam: A component-oriented platform architecting method family for product family engineering," in *SPLC*, 2000, pp. 167–180.

[21] C. Atkinson, J. Bayer, and D. Muthig, "Component-based product line development: the kobra approach," in *Proceedings of the first conference on Software product lines : experience and research directions: experience and research directions*. Norwell, MA, USA: Kluwer Academic Publishers, 2000, pp. 289–309.

[22] M. Matinlassi, E. Niemelä, and L. Dobrica, "Quality-driven architecture design and quality analysis method. a revolutionary initiation approach to a product line architecture," Technical Research Center of Finland, Espoo, Finland, VTT Publications 456, 2002.

[23] M. Matinlassi, "Comparison of software product line architecture design methods: Copa, fast, form, kobra and qada," in *Proceedings of the 26th International Conference on Software Engineering*, ser. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 127–136.

[24] K. Czarnecki, S. Helsen, and U. W. Eisenecker, "Staged configuration using feature models," in *SPLC*, 2004, pp. 266–283.

[25] K. Bak, K. Czarnecki, and A. Wasowski, "Feature and meta-models in clafer: Mixed, specialized, and coupled," in *Proceedings of the Third International Conference on Software Language Engineering*, ser. SLE'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 102–122.

[26] S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid, "Dynamic software product lines," *Computer*, vol. 41, no. 4, pp. 93–95, Apr. 2008.

[27] P. Istoan, G. Nain, G. Perrouin, and J.-M. Jezequel, "Dynamic software product lines for service-based systems," in *Proceedings of the 2009 Ninth IEEE International Conference on Computer and Information Technology - Volume 02*, ser. CIT '09, Washington, DC, USA, 2009, pp. 193–198.

[28] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel, "Tailoring dynamic software product lines," in *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, ser. GPCE '11. New York, NY, USA: ACM, 2011, pp. 3–12.

[29] L. Shen, X. Peng, J. Liu, and W. Zhao, "Towards feature-oriented variability reconfiguration in dynamic software product lines," in *Proceedings of the 12th International Conference on Top Productivity Through Software Reuse*, ser. ICSR'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 52–68.

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

[31] M. Nicho, H. Fakhry, and C. Haiber, "An integrated security governance framework for effective pci dss implementation." *IJISP*, vol. 5, no. 3, pp. 50–67, 2011.

[32] R. P. Azzolini, C. M. F. Rubira, L. P. Tizzei, F. N. Gaia, and L. Montecchi, "Evolving a Software Products Line for E-commerce Systems: a Case Study," in *Proceedings of the Workshop on Variability for Qualities in Software Architecture (VAQUITA 2015)*, September 2015.

[33] M. Laguna and C. Hernandez, "A software product line approach for e-commerce systems," in *e-Business Engineering (ICEBE), 2010 IEEE 7th International Conference on*, Nov 2010, pp. 230–235.

[34] S. T. Ruehl and U. Andelfinger, "Applying software product lines to create customizable software-as-a-service applications," in *Proceedings of the 15th International Software Product Line Conference, Volume 2*, ser. SPLC '11. New York, NY, USA: ACM, 2011, pp. 16:1–16:4.

[35] V. Tran, J. Keung, A. Liu, and A. Fekete, "Application migration to cloud: a taxonomy of critical factors," in *Proceedings of the 2nd international workshop on software engineering for cloud computing*. ACM, 2011, pp. 22–28.

[36] J. Kabbedijk and S. Jansen, "Variability in multi-tenant environments: architectural design patterns from industry," in *Advances in conceptual modeling. recent developments and new directions*. Springer, 2011, pp. 151–160.

[37] G. Kousiouris, D. Kyriazis, A. Menychtas, and T. Varvarigou, "Legacy applications on the cloud: Challenges and enablers focusing on application performance analysis and providers characteristics," in *Cloud Computing and Intelligent Systems (CCIS), 2012 IEEE 2nd International Conference on*, vol. 2. IEEE, 2012, pp. 603–608.

[38] P.-J. Maenhaut, H. Moens, M. Verheye, P. Verhoeve, S. Walraven, E. Truyen, W. Joosen, V. Ongenae, and F. De Turck, "Migrating medical communications software to a multi-tenant cloud environment," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE, 2013, pp. 900–903.

[39] PCI Security Standards Council, "PCI SSC Data Security Standards Overview," https://www.pcisecuritystandards.org/security_standards/, 2013.

[40] Q. Chen, C. Zhang, and S. Zhang, *Secure Transaction Protocol Analysis: Models and Applications*. Berlin, Heidelberg: Springer-Verlag, 2008.

[41] R. Mietzner, T. Unger, R. Titze, and F. Leymann, "Combining different multi-tenancy patterns in service-oriented applications," in *Enterprise Distributed Object Computing Conference, 2009. EDOC '09. IEEE International*, 2009, pp. 131–140.

[42] R. Mietzner, F. Leymann, and M. Papazoglou, "Defining composite configurable saas application packages using sca, variability descriptors and multi-tenancy patterns," in *Internet and Web Applications and Services, 2008. ICIW '08. Third International Conference on*, 2008, pp. 156–161.

[43] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. 't Hart, "Enabling multi-tenancy: An industrial experience report," in *Software Maintenance (ICSM), 2010 IEEE International Conference on*, 2010, pp. 1–8.

[44] G. O. Karame, E. Androulaki, and S. Capkun, "Double-spending fast payments in bitcoin," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 906–917.

[45] A. Van Lamsweerde, *Requirements engineering: from system goals to UML models to software specifications*. Wiley, 2009.

[46] N. J. Gunther, "A general theory of computational scalability based on rational functions," *CoRR*, vol. abs/0808.1431, 2008.

[47] J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, 2013, pp. 301–311.