Microservice Architectures for Scalability, Agility and Reliability in E-Commerce

Wilhelm Hasselbring
Software Engineering Group
Kiel University
D-24098 Kiel, Germany
Email: hasselbring@email.uni-kiel.de

Guido Steinacker
Otto GmbH & Co KG
Werner-Otto-Straße 1–7
D-22179 Hamburg, Germany
Email: guido.steinacker@otto.de

***Abstract*—Microservice architectures provide small services that may be deployed and scaled independently of each other, and may employ different middleware stacks for their implementation. Microservice architectures intend to overcome the shortcomings of monolithic architectures where all of the application's logic and data are managed in one deployable unit.**

We present how the properties of microservice architectures facilitate scalability, agility and reliability at otto.de, which is one of the biggest European e-commerce platforms. In particular, we discuss vertical decomposition into self contained systems and appropriate granularity of microservices as well as coupling, integration, scalability and monitoring of microservices at otto.de. While increasing agility to more than 500 live deployments per week, high reliability is achieved by means of automated quality assurance with continuous integration and deployment.

I. Introduction

Traditionally, information system integration aims at achieving high data coherence among heterogeneous information sources [1], [2]. However, a great challenge with integrated databases is the inherently limited horizontal scalability of transactional database management [3]. One of the intentions of microservice architectures is to overcome the limited scalability of such monolithic architectures. A system has a microservice architecture when that system is composed of many collaborating microservices; typically without centralized control [4]. Microservices are built around business capabilities and take a full-stack implementation of software for that business area. The following topics are of eminent relevance.

1) Vertical Decomposition for Microservices: The trade-off between many small microservices and a few more coarse grained services must be considered in microservice architectures. To achieve an appropriate granularity, we propose a vertical decomposition into self contained systems (scs-architecture.org) along business services, as exemplified at otto.de. Besides scalability, an appropriate modular structure supports program comprehension, resilience (inhibiting error propagation) and autonomous teams with good knowledge of their vertical domain.

2) Loose Coupling and Eventual Consistency: Decentralizing responsibility for data across microservices has implications for managing updates. The traditional approach to dealing with updates is to use transactions to guarantee consistency when updating multiple resources. This approach is often used within monoliths. Using transactions this way helps with consistency, but imposes significant coupling, which is problematic across multiple services. Distributed transactions are notoriously difficult to implement and as a consequence microservice architectures emphasize transaction-less coordination between services, with explicit recognition that consistency may only be eventual consistency and problems are dealt with by compensating operations.

3) Microservices for Scalability and Fault Tolerance: Non-functional attributes, such as scalability and fault tolerance for high availability, are addressed by microservice architectures. A consequence of using microservices is that applications need to be designed such that they can tolerate the failure of individual services. Since services can fail at any time, it is important to be able to detect the failures quickly and, if possible, automatically restore services. Microservice applications put a lot of emphasis on real-time monitoring of the application, checking both technical metrics (e.g. how many requests per second is the database getting) and business relevant metrics (such as how many orders per minute are received). Monitoring can provide an early warning system of something going wrong that triggers development teams to follow up. Scalable systems should allow to react to changing workloads automatically via elastic capacity management, as offered by cloud infrastructures. With microservice architectures, you can dynamically replicate those microservices to cloud infrastructures that are under heavy load. It is not necessary to scale the complete system, as it would be required with a monolithic system. Small services are easier to deploy, and since they are autonomous, are less likely to cause system failures when they go wrong.

4) Microservices and DevOps: The DevOps movement intends to improve communication, collaboration, and integration between software developers (Dev) and IT operations professionals (Ops). Automation is key to DevOps success: automated building of systems out of version management repositories; automated execution of unit tests, integration and system tests; automated deployment in test and production environments; including performance benchmarks [5].

5) Scalable Microservice Deployment: Containerization is a new trend that is well suited for microservices [6]. By uti-

Preprint of: Wilhelm Hasselbring, Guido Steinacker: "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce"
In: Proceedings 2017 IEEE International Conference on Software Architecture Workshops.

lizing containers, for instance via Docker (www.docker.com), one can deploy service instances with lower overheads than via operating-system virtualization. The deployment on compute clusters or on the cloud is performed using containers running in cluster-management infrastructures such as Apache Mesos (mesos.apache.org). These cluster-management infrastructures schedule containers onto nodes in a compute cluster and manage load balancing among containers on these clusters.

6) Scalable Microservices Development: The well known Conway's Law states that organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations [7]. If the organizational structure is decomposed vertically and according to the microservices structure into cross-functional feature teams, scaling development capacities according to changing business requirements is enabled. The feature teams should be highly independent, having members of all roles and skills that are required to build and maintain their microservices. Microservices reinforce modular structure, which is particularly important for larger teams. Decoupling teams is as relevant as decoupling software modules.

II. Microservices at otto.de

Having a turnover of more than 2.563 billion Euros in business year 2015/2016 and up to 1 million visitors per day, otto.de is one of the biggest online shops in Europe. In 2011 Otto started a complete re-implementation of their e-commerce software from scratch. The drivers for this decision primarily were non-functional requirements like scalability, performance and fault tolerance. Regarding scalability and agility, they were not only thinking about technical scalability in terms of load or data. Particularly, a solution that was scaling with respect to the number of teams and/or developers working on the software at a given time was needed. In addition to that, it was planned to practice DevOps including continuous deployment, in order to deliver features quickly to the customer.

What was found initially was somewhat unusual, but in the end highly successful: Instead of setting up a single development team to create a new platform for the shop, Otto was actively employing Conway's Law by starting development with initially four separate teams. Consequently, they were not building a single, monolithic application, but a vertically decomposed system consisting of four loosely coupled applications: Product, Order, Promotion, and Search/Navigation. In the following years, Otto founded more teams and systems. Today, there are 18 Teams working on 45 different applications in 12 so-called "verticals" as illustrated in Figure 1.

A vertical is a part of the platform that is responsible for a single bounded context in a business domain [8]. Verticals could be as small as a microservice, but most of the time, they are more coarse grained. Communication between verticals is only allowed by accessing REST APIs in the background using the "Backend Integration Proxy" – see Figure 1. This makes it easier to ensure that slow or unavailable applications cannot tear down other applications or the whole shop with a snowball effect.

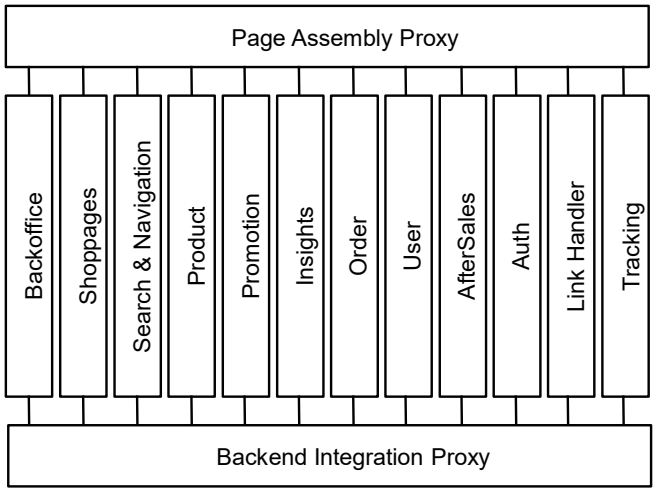

Fig. 1. Current Vertical Decomposition at otto.de

Verticals follow the "Shared Nothing" principle: They do not share state, no infrastructure components beside of the two Proxies, no database or other shared resources. Verticals do not make use of HTTP sessions, shared caches or similar things. Only a very limited amount of client-side state (using cookies or local storage) is shared between different systems, in order to have a common understanding on who is accessing the shop. The big advantages of shared-nothing architectures are excellent horizontal scalability and improved fault-tolerance. The reason for this is apparent: if two components are not sharing anything, they are obviously unable to have a negative impact on each other.

1) Integrating Verticals at otto.de: All pages of the shop contain fragments from different verticals: a preview to the shopping cart, a navigation structure, maybe some products or other parts. In order to integrate these fragments, the following principles for the "Page Assembly Proxy" are used [9]:

- Fragments that are not part of the primary content or fragments that are initially invisible are preferably integrated at the client side using AJAX [10]. The shopping cart preview, for example, is one of these features that is included on almost every page.
- Primary content is integrated at the server side using Edge Side Includes [11] resolved through a Varnish Reverse Proxy (www.varnish-cache.org).

This way, the verticals are integrated at the user interface into a single page website. Users experience the shop as a consistent entity, despite the backend decomposition.

2) Communication among Verticals at otto.de: All verticals have redundant data using pull-based data replication. This ensures that a vertical is able to deliver the content without having to access other verticals during a request. At otto.de, the pull principled via the Apache Kafka high-throughput distributed messaging system (kafka.apache.org) in combination with Atom feeds (tools.ietf.org/html/rfc4287) is implemented.

There exist a few features with push notifications in situations, where guaranteed ordering and delivery is not required.

3) Verticals and Microservices at otto.de: For microservices, as for other software components, it is essential to design for the appropriate granularity [12]. The vertical domains, as illustrated above, could be small enough to be implemented as a microservice – but they may also be too coarse grained. Thus, it is sometimes necessary to further refine those vertical pillars: If possible by extracting independent features from existing code into a new vertical (ideally being a microservice), or by cutting the vertical into a distributed system of microservices.

4) Scaling Delivery Pipelines at otto.de: To deploy frequently and automatically, continuous deployment pipelines [13] are used for every single application. Every commit is first checked out, compiled, packaged, deployed and tested in the continuous-integration stage. After all the tests have passed, the container is deployed to the next stage, called testing. This stage is used to run load and integration tests and is also used to approve stories by the product owners. Because all teams are continuously integrating, this stage contains the latest development versions of all verticals.

The last step before going live is the pre-live stage, where the next deployment of a service is tested against those versions of other services, which are currently deployed and live. Another suite of automatic (and some manual) integration tests is executed, to ensure the compatibility of new and old versions of the software being deployed. The final step is the deployment to the live environment.

Due to the high number of deployment pipelines at otto.de, traditional continuous-integration servers like Jenkins (jenkins.io) reach their limits. If you need to keep dozens of pipelines up to date and in a similar configuration, you have to engineer these pipelines as any other critical software components. Meanwhile, the pipelines are implemented with the internal domain-specific language LambdaCD (www.lambda.cd) to describe and run deployment pipelines. Because LambdaCD pipelines are nothing else but microservices responsible for building, testing and deploying a single application, they are running in the same infrastructure as other microservices. They can be tested, they are running locally on a notebook without any extra continuous-integration or application server, and – interestingly enough – they can be debugged just like any other software system.

5) Agility and Reliability at otto.de: At otto.de, most microservices are deployed fully automatically after every single push to the version control system. Automation is key to DevOps success: automated building of systems out of version management repositories; automated execution of unit tests, integration tests, and system tests; automated deployment in test and production environments.

Since the start of 2015, more and more microservices were introduced within the verticals. Meanwhile, the number of live deployments increased from 40 to currently more than 500 deployments per week. Figure 2, blue line, shows the number of live deployments per week for the last two years.

At the same time, the reliability is retained and even improved: The number of live incidents is not following these numbers, but is staying on the same, very low level; refer to Figure 2, red bars. The incidents are counted since 2015, thus no incident data is available for 2014. However, it indicates that the quality assurance measures implemented for continuous integration and deployment really take effect for reliability.

6) Monitoring Microservices at otto.de: Every feature team has at least two large screens beside their team space: one is used to monitor the deployment pipelines and additional build-related information, the other monitor provides various graphs and metrics for all the microservices and verticals of the team. The increasing number of microservices is already becoming a challenge for some teams. Basic monitoring and alarming is not sufficient anymore. In the future, it is planned to find solutions to automatically detect anomalies [14] in all the available metrics, such that the dashboards can give an overview about only those graphs that are currently of interest.

7) Dynamic Scaling of Microservices at otto.de: Via monitoring the CPU utilization and the number of incoming request, otto.de is able to react to changing workloads automatically via elastic capacity management [15]. With the microservice architecture, those features that are under heavy load are dynamically replicated. Developers are now able to set up, deploy and scale microservices without any support from the operations team. Applications can be scaled dynamically, depending on the current load that a single microservice is facing.

8) Code Sharing and Reuse at otto.de: Some code to implement common cross-functional requirements is required. For example, many microservices regularly run data import jobs, because of polyglot persistence. A common solution to implement such cross-cutting concerns is to extract the required code into a number of libraries that are shared among all services. However, the downside of this approach would be a tight coupling of service implementations. Particularly, third-party dependencies would restrict teams in their freedom to choose the best solution for the particular problem domain.

We argue that code should not be shared among microservices to avoid dependencies; only reuse of framework code as open source software is recommended. In order to keep teams and applications as independent and loosely coupled as possible, no code is shared between verticals or microservices. Beside of the already mentioned frontend assets, there is only one exception to this rule: General-purpose code that is loosely coupled, highly coherent and "good" enough to be open sourced may be published on GitHub (github.com/otto-de). Other teams are free to use these libraries – or to write their own solutions.

Apparently, open sourcing the code instead of sharing common private libraries seems to be almost the same. However, the open-source approach has some psychological effects: Developers show a tendency to apply higher quality standards if they know that the code will be publicly available.

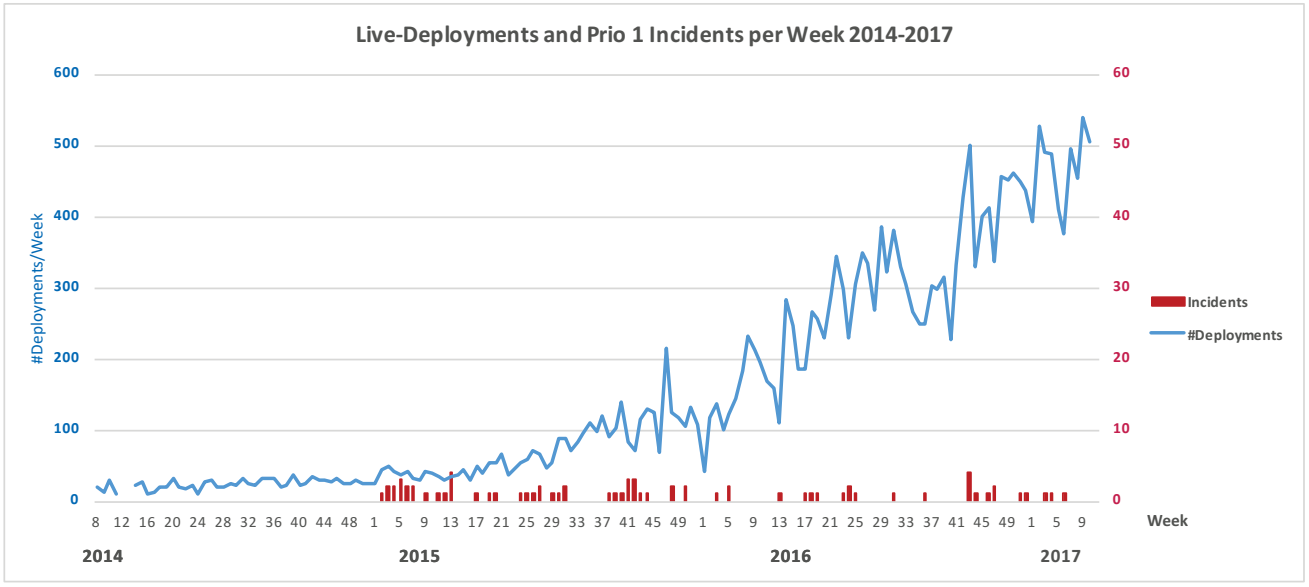

Fig. 2. Number of life deployments per week at otto.de over the last two years. Despite the significant increase of deployments, the number of live incidents remains on a very low level.

III. Conclusions and Take Away

Microservice architectures can be an enabler for scalable, agile and reliable software systems, as demonstrated with the successful re-implementation of otto.de. A vertical decomposition along business services provides the basis for highly scalable and reliable software services. Other e-commerce systems such as Amazon follow similar approaches.

We discussed how coupling, integration, scalability, monitoring and development of microservices are addressed in teams at otto.de. Besides focusing on the scalability of a microservice-based system itself, we emphasize scalability of deployment pipelines for continuous delivery.

Team organization is vital for success. Microservice architectures allow to assign the responsibility for all concerns of certain business capabilities – from requirements to operations – to individual teams. The responsibility, combined with open-source development, yields team's empathy for "their" microservices. Both, the architecture and the organizational structure are vertically decomposed. This enables Otto to scale development capacities according to new requirements.

Additional concerns at otto.de, not discussed in the present paper, are behavior-driven design, test-driven development, customer-driven contracts, feature toggles, polyglot programming, and embedding application servers.

Full automation of quality assurance and software deployment allows for early fault and error detection, thus reducing repair times both during development and during operations.

Microservice architectures enable scalability [16]. As a take away of this paper, we presented how both agility and reliability may be achieved; in addition to scalability.

However, be aware that microservice architectures also come with costs. Maintaining consistency, monitoring, alarming and fault tolerance are difficult for a distributed system, which means that you have to operate a much more complex system than in monolithic architectures. You need a mature operations team to manage lots of services, which are being redeployed frequently.

References

[1] W. Hasselbring, "Information system integration," *Communications of the ACM*, vol. 43, no. 6, pp. 32–36, 2000.

[2] ——, "Web Data Integration for E-Commerce Applications," *IEEE Multimedia*, vol. 9, no. 1, pp. 16–25, 2002.

[3] M. Abbott and M. Fisher, *The Art of Scalability*, 2nd ed. Addison-Wesley, 2015.

[4] S. Newman, *Building Microservices*. O'Reilly, 2015.

[5] J. Waller, N. C. Ehmke, and W. Hasselbring, "Including performance benchmarks into continuous integration to enable DevOps," *ACM SIGSOFT Softw. Eng. Notes*, vol. 40, no. 2, pp. 1–4, Mar. 2015.

[6] V. Marmol, R. Jnagal, and T. Hockin, "Networking in containers and container clusters," in *Proceedings NetDev 0.1*, 2015.

[7] M. E. Conway, "How do committees invent?" *Datamation*, vol. 14, no. 4, pp. 28–31, 1968.

[8] E. Evans, *Domain-driven design*. Addison-Wesley, 2004.

[9] G. Steinacker, "On monoliths and microservices," 2015, http://dev.otto.de/2015/09/30/on-monoliths-and-microservices/.

[10] E. Woychowsky, *AJAX: Creating Web Pages with Asynchronous JavaScript and XML*. Prentice Hall, 2006.

[11] M. Tsimelzon *et al.*, "ESI language specification," 2001, w3C Note 04 August 2001, https://www.w3.org/TR/esi-lang.

[12] W. Hasselbring, "Component-based software engineering," in *Handbook of Software Engineering and Knowledge Engineering*. World Scientific Publishing, 2002, pp. 289–305.

[13] J. Humble and D. Farley, *Continuous Delivery*. Pearson, 2010.

[14] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring, "Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation," in *Proc. CSMR 2009*, 2009.

[15] R. von Massow, A. van Hoorn, and W. Hasselbring, "Performance simulation of runtime reconfigurable component-based software architectures," in *Proceedings ECSA 2011*, 2011, pp. 43–58.

[16] W. Hasselbring, "Microservices for scalability: Keynote talk abstract," in *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE 2016)*. ACM, 2016, pp. 133–134.