

An Overview of Microservice Architecture Impact in Terms of Scalability and Reliability in E-Commerce: A Case Study on Uber and Otto.De

Janita J. Phatak
Department of Informational Technology
S. S. and L. S. Patkar Varde College, University of Mumbai, Mumbai, Maharashtra, India
phatakjaita@gmail.com

I. INTRODUCTION

A Microservices architecture consists of a collection of services that are small and autonomous. Each service is self-contained and should imply a single business capability. Thus, these applications built on microservices architecture are loosely coupled. The basic idea of implementing an microservice architecture is that when an application is broken down into small, composable pieces or components that work together are easy to deploy, create, and maintain. Each service has a separate codebase structure, thus can be managed by a small group of development team. This is the opposite to Traditional monolithic architectural style where the final product of application is developed all in single and indivisible unit. In monolithic architecture, all features of the application are written as separate modules which are then packaged in a single main application. We should use microservice architecture when there is a large application and need high release speeds, high scalability, high resilience, easier debugging and maintenance.

From an e-commerce perspective, today's competitive world of e-commerce requires various strategies to increase revenue and customer satisfaction. E-commerce companies are required to share various services to achieve flexibility. They should have reusability services, automated service deployments, and fast service scalability. Today, monolithic applications in e-commerce, are becoming a barrier to innovation as they are deployed at once and need to be checked and tested at the end, it comes with complexity and flexibility. Microservices are the ultimate risk-free way to build e-commerce platforms for traffic peaks, as well as implement and test new trends, such as new payment methods, voice assistants or progressive web apps. Micro-service can also be used to set up complex omnichannel systems. In order to meet your customers' expectations in omnichannel systems, you need to gather all the information about products, shipments, stocks and orders and keep them up to date. Microservices enable companies to use API gateways that integrate POS, ERP, or WMS solutions that are in the best range and synchronize them with existing processes.

The concept of microservices was coined by the evangelists of microservices, with Martin Fowler at the forefront and global companies facing the wall in terms of business scalability, agility and speed of implementation of changes [1]. Amazon, a provider of major online marketplaces, was one of the first. After this, Other leading e-commerce companies have transformed their infrastructure into micro-services. eBay, Coca Cola, Netflix, Spotify, Uber, Etsy, Gilt and Zalando, just to name a few, used microservices to create a flexible, global system and a whole new work culture, easy to access and inspiring for developers.



Figure 1. E-Commerce platforms using Microservice Architecture [2]
DOI: 10.48175/568

微服务架构对电子商务可扩展性与可靠性影响的综述：以Uber和Otto.de为例的案例研究

贾尼塔·J·法塔克
信息技术系
S.S.与L.S.帕特卡尔瓦尔福德学院, 孟买大学, 印度马哈拉施特拉邦孟买
phatakjaita@gmail.com

一、引言

微服务架构由一组小型、自治的服务组成。每个服务均自成一体, 且应体现单一的业务能力。因此, 基于微服务架构构建的应用程序具有松散耦合的特点。实现微服务架构的基本思想是: 将应用程序拆分为若干小型、可组合的组件或模块, 这些组件协同工作, 从而更易于部署、开发与维护。每个服务拥有独立的代码库结构, 因而可由一个规模较小的开发团队负责管理。这与传统的单体架构风格截然相反——在单体架构中, 应用程序最终以一个不可分割的整体进行开发; 所有功能模块虽各自编写, 但最终被打包进同一个主应用程序中。当面临大型应用且对发布速度、可扩展性、弹性容错能力、调试与维护便捷性等方面有较高要求时, 应采用微服务架构。

从电子商务视角来看, 在当今竞争激烈的电商环境中, 企业需采取多种策略以提升营收与客户满意度。电商公司必须共享各类服务, 以实现更高的灵活性; 其服务应具备可复用性、支持自动化部署, 并能快速扩展。当前, 电商领域中的单体应用正日益成为创新的障碍: 此类应用需整体部署, 且所有功能须在最终阶段统一测试与验证, 由此带来复杂性并制约灵活性。微服务则是构建电商系统以应对流量高峰的最稳妥方案, 同时也便于实施和测试新兴趋势(例如新型支付方式、语音助手或渐进式网页应用)。此外, 微服务还可用于构建复杂的全渠道系统。为满足客户对全渠道系统的期望, 企业需全面采集并实时更新有关商品、物流、库存及订单等全部信息。微服务使企业能够借助API网关, 集成最适合自身需求的销售点系统(POS)、企业资源计划系统(ERP)或仓储管理系统(WMS), 并将其与现有业务流程无缝同步。

微服务概念由微服务布道者提出, 马丁·福勒(Martin Fowler)是其中的领军人物; 与此同时, 全球各大企业也正面临业务可扩展性、敏捷性以及变更实施速度等方面的瓶颈[1]。作为大型在线电商平台的提供商, 亚马逊(Amazon)便是最早采用该架构的企业之一。此后, 其他领先的电子商务公司也纷纷将其基础设施转型为微服务架构。eBay、可口可乐(Coca Cola)、Netflix、Spotify、Uber、Etsy、Gilt和Zalando等公司——仅举几例——均借助微服务构建起灵活、全球化的系统, 并催生了一种全新的工作文化: 这种文化对开发者而言易于融入, 且极具激励性。



图 1. 采用微服务架构的电子商务平台 [2]
DOI: 10.48175/568

2.1 The Advantages of Microservices

A. Independent Usage

Each service will have a codebase. You can also separate the database layer. One service can be tested independently without worrying about the entire product. You can also choose different techniques to use in each service. Security can be different for everyone.

B. Improved defect isolation

Large applications can be remained unaffected by the failure of a single module.

C. Fast Deployment

Microservices architecture allows rapid development and deployment as it simplifies design. Each module is easily created, tested and deployed independently, thus giving your business increased agility and faster time to market.

D. Scalability

Microservices offers ease of scalability according to the architectural design because the modules work independently and the context of each is limited. The design is much simpler allowing new features to be added faster. Each module is tested separately, relying on DevOps methods and test automation, and the entire system is automatically tested in a matter of minutes, thus reducing the time to create, test, and release new features [3].

At the infrastructure level, each micro-service is measured and managed separately. You do not need to scale the entire system in terms of peak time and load. Only micro services are counted under load. This is done automatically without any human intervention with self-scaling capabilities and self-healing.

E. Reliability

A successful microservice architecture prevents any system failure. Its repeatable automation, and designing pattern features keep the system running.

2.2 Guidelines for Implementing Microservices Architecture

As a developer, when you decide to build an application, separate the domain and be clear about the functionality.

- Each micro service you design will focus on just one service of the application.
- Make sure you design the application in such a way that each service is individually deployable.
- Make sure that the communication between the microservices is done through a stateless server.
- Each service can be redeveloped into smaller services with their own microservices.

2.3 Microservices Architecture in Ecommerce

Electronic commerce, also known as e-commerce, is a way of buying or selling products on web or online services. Technologies such as m-commerce (mobile commerce), electronic fund transfer, online transaction processing, internet marketing, electronic data interchange (EDI), there are supply chain management, inventory management systems and automated data collection systems included in e-commerce.

Microservices architecture is used as a separate component or assembly connected by microservices, REST API, to create ecommerce applications. Multiple user interfaces can be created using the same backend microservices. While the features of this architectural approach are of major interest to development teams, its benefits to businesses are numerous. Small development teams can work simultaneously to create different services for faster application implementation and market entry. Migration into modular ecommerce architecture is an investment in instalments. You can rebuild and modernize your eCommerce solution step-by-step by replacing each business function with a micro-service. You can start with areas where custom workflows or designs can influence the customer experience and therefore lead to the highest sales. Scaling a microservices-based ecommerce application is easier and less expensive because each service lives through its own lifecycle - it is created, modified, tested, and (if necessary) removed separately from other services. This is beneficial for growth-focused companies that are planning to invest and gradually develop their e-commerce presence. Ecommerce applications

2.1 微服务的优势

A. 独立使用

每项服务均拥有独立的代码库。数据库层亦可实现分离。某项服务可独立进行测试, 无须担忧整个产品的其他部分。您还可为各项服务分别选用不同的技术方案。此外, 每项服务的安全策略也可各不相同。

B. 缺陷隔离能力提升

大型应用程序可免受单个模块故障的影响。

C. 快速部署

微服务架构通过简化设计, 支持快速开发与部署。每个模块均可独立创建、测试和部署, 从而提升企业敏捷性, 并显著缩短产品上市时间。

D. 可扩展性

微服务架构在设计上天然具备良好的可扩展性, 因为各模块相互独立, 且各自上下文范围有限。整体设计更为简洁, 使得新功能能够更快添加。每个模块均单独进行测试, 依托 DevOps 方法与测试自动化; 整个系统可在数分钟内完成自动测试, 从而大幅缩短新功能的开发、测试与发布周期[3]。

在基础设施层面, 每个微服务均被单独度量与管理。您无需为应对峰值时段与负载而对整个系统进行扩容, 仅需按实际负载情况对相关微服务进行扩容。该过程通过自扩展能力与自愈机制自动完成, 无需人工干预。

E. 可靠性

成功的微服务架构可防止任何系统故障。其可重复的自动化机制以及设计模式特性, 可确保系统持续稳定运行。

2.2 微服务架构实施指南

作为开发人员, 在决定构建应用程序时, 应将领域进行合理划分, 并明确各项功能职责。

- 您所设计的每个微服务均应专注于应用程序的单一业务功能。
- 请确保以各服务均可独立部署的方式设计应用程序。
- 确保微服务之间的通信通过无状态服务器进行。
- 每个服务均可重构为更小的独立服务, 各自拥有其自身的微服务。

2.3 电子商务中的微服务架构

电子商务(又称电子商贸)是一种通过网页或在线服务购买或销售商品的方式。电子商务所涵盖的技术包括移动商务(m-commerce)、电子资金转账、在线事务处理、网络营销、电子数据交换(EDI), 以及供应链管理、库存管理系统和自动数据采集系统等。

微服务架构作为一种独立的组件或模块, 通过微服务与 REST API 相互连接, 用于构建电子商务应用。可基于同一套后端微服务开发多个用户界面。尽管该架构模式对开发团队具有显著吸引力, 但其为企业带来的益处同样十分丰富: 小型开发团队可并行协作, 分别开发不同服务, 从而加速应用交付与市场投放。向模块化电商架构迁移是一项分阶段投入的战略性投资: 您可逐步重构并现代化您的电商解决方案, 即逐一以微服务替代各项业务功能。建议优先从那些可通过定制化 workflow 或界面设计显著提升客户体验、进而带来最高销售转化率的业务领域入手。基于微服务的电商应用更易于扩展, 且扩展成本更低——因为每项服务均拥有独立的生命周期: 其创建、修改、测试乃至(如确有必要)移除, 均可与其他服务完全解耦。这一特性对于专注于增长、计划持续投入并渐进式拓展其电商布局的企业尤为有利。电子商务应用

(and therefore online sales) are more flexible because malformations within a single microservice do not interfere with the entire application. The cost of infrastructure can be optimized because microservices are cloud-native and each service can be hosted on a different cloud instance based on its bandwidth requirements.

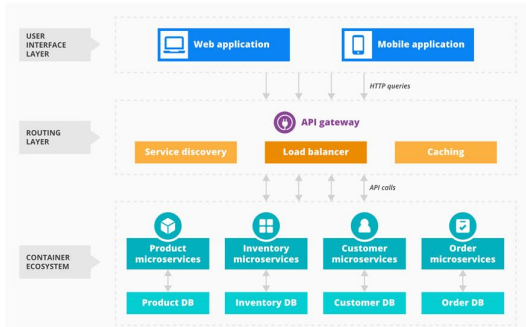


Figure 2: Microservice Architecture in ecommerce [4]

- The user interface layer is used to create multiple digital customer touchpoints using the same microservices on the back-end.
- The routing layer connects HTTPS queries to the corresponding microservices.
- API Gateway is used to build any amount of API.
- Service Discovery is used to find dynamically assigned microservices instances network locations.
- Load balancers are used to deliver API calls in micro services.
- Caching is used to store and return static data (e.g. text files) for faster uploading of web pages and good customer experience.
- The container ecosystem stores units of micro services. Microservices are built around specific business contexts: data types, responsibilities, functions.

2.4 Common Microservices Used in Ecommerce

There are various types of micro-services available in e-commerce. Some examples of microservices include:

A. Order Micro-Service

Order Micro-Services enables direct access to all orders on the channel in which your business operates, With web stores, mobile apps, IoT devices. There are micro-service orders Functionality that allows your business to maintain and route, cancel, refund, satisfy, or duplicate orders

B. Pricing Micro-Service

Micro-service ("pricing") allows the pricing manager to create and manage multiple price lists. Each price list has its own currency and its own individual alternative price types.

C. Customer Micro-Service

Customer Micro-Services enables your business to manage all customer information, such as registration. Allows you to store past orders, addresses and privacy and consent preferences and everything about your customers in one place.

D. Merchandising Micro-Service

Merchandising is a micro-service that gives merchants and managers the ability to create and manage product presentations on the storefront. Merchants can create and product catalog pages using static and dynamic (rule-based) categories, like product detail pages, category landing pages, site navigation and other digital experiences.

(因而在在线销售额) 具备更高灵活性: 单个微服务内部出现异常时, 不会影响整个应用的正常运行。基础设施成本亦可得到优化, 因为微服务原生支持云环境, 且各服务可根据其带宽需求, 分别部署于不同的云实例之上。

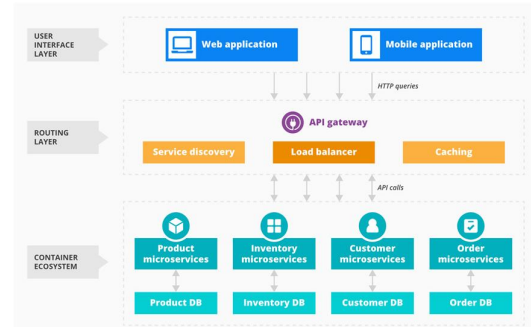


图2：电子商务[4]中的微服务架构

- 用户界面层利用后端相同的微服务, 构建多个数字化客户触点。
- 路由层将 HTTPS 请求转发至对应的微服务。
- API 网关用于构建任意数量的 API。
- 服务发现用于动态定位微服务实例的网络地址。
- 负载均衡器用于在微服务架构中分发 API 调用。
- 缓存用于存储并返回静态数据(例如文本文件), 以加快网页加载速度, 提升客户体验。
- 容器生态系统用于托管微服务单元。微服务围绕特定的业务场景构建, 涵盖数据类型、职责与功能。

2.4 电子商务中常用微服务

电子商务领域存在多种类型的微服务。以下是一些微服务示例:

A. 订单微服务

订单微服务使您的业务能够直接访问您所运营渠道中的全部订单, 包括网站商店、移动应用及物联网设备。该微服务提供订单管理功能, 支持您对订单进行维护、路由、取消、退款、履约或复制操作。

B. 定价微服务

定价微服务(“pricing”)允许定价管理员创建并管理多个价目表。每个价目表均拥有独立的货币及多种可选的价格类型。

C. 客户微服务

客户微服务使您的业务能够统一管理所有客户信息(例如注册信息), 支持存储客户的过往订单、地址、隐私与授权偏好等全部相关信息。

D. 商品营销微服务

商品陈列(Merchandising)是一项微服务, 可为商家和管理人员提供在 storefront(线上店铺)上创建和管理商品展示内容的能力。商家可通过静态分类及动态(基于规则的)分类来创建商品目录页面, 例如商品详情页、分类着陆页、网站导航及其他数字体验页面。

3.1 Microservices at Uber

A. Uber's Previous Architecture

Like most startups, Uber embarked on a journey through a single city with a monolithic architecture designed for a single offer. It had one codebase at that time, and it was sufficient to solve Uber's core business problems. However, as Uber began to expand worldwide, it faced various challenges in terms of scalability and continuous integration.



Figure 3: Uber's Monolithic architecture [5]

The above diagram shows Uber's previous architecture.

- The passenger and driver connect with the REST API.
- They use three different adapters with APIs, such as billing, payment, email / messaging, which we see when we book a cab.
- They use a MySQL database for storing all their data.

Therefore, if you notice all the features like passenger management, billing, notification features, payment, trip management and driver management have been created in one framework.

B. Problem Statement

This type of framework posed various challenges as Uber began to expand worldwide. The following are some of the major challenges

- All features needed to be recreated, deployed, and re-tested to update a single feature.
- Fixing a bug in a repository was extremely difficult because developers had to change the code over and over again.
- With the introduction of new features from worldwide, scaling all features simultaneously becomes very difficult to handle at a single place.
- Availability Risks. A single regression within a monolithic code base can bring the whole system (in this case, all of Uber) down.
- Risky, expensive deployments. These were painful and time consuming to perform with the frequent need for rollbacks.
- Poor separation of concerns. It was difficult to maintain good separations of concerns with a huge code base. In an exponential growth environment, expediency sometimes led to poor boundaries between logic and components.
- Inefficient execution. These issues combined made it difficult for teams to execute autonomously or independently.

3.1 Uber 的微服务架构

A. Uber 此前的架构

与大多数初创公司一样, Uber 最初仅在一个城市开展业务, 其采用的是面向单一业务场景设计的单体架构。当时, Uber 仅维护一个代码库, 该代码库足以解决其核心业务问题。然而, 随着 Uber 开始向全球扩张, 其在可扩展性与持续集成方面面临了诸多挑战。

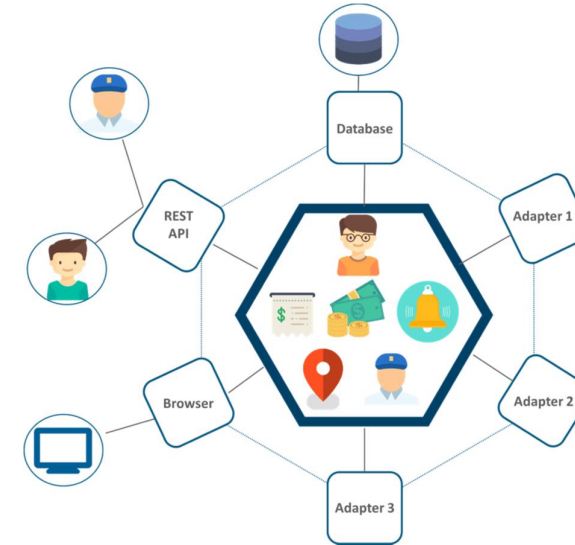


图 3: Uber 的单体架构 [5]

上图展示了优步 (Uber) 此前的架构。

- 乘客与司机均通过 REST API 进行连接。
- 他们使用了三种不同的适配器来对接各类 API, 例如计费、支付以及电子邮件/消息服务等, 这些我们在预订出租车时即可看到。
- 他们采用 MySQL 数据库存储所有数据。

因此, 若您留意所有功能模块——如乘客管理、计费、通知、支付、行程管理及司机管理——便会发现它们均集成于同一框架之中。

B. 问题陈述

随着优步 (Uber) 开始向全球扩张, 此类框架带来了诸多挑战。以下是一些主要挑战:

- 每次仅更新一项功能, 都需重新开发、部署并再次测试全部功能。
- 在代码仓库中修复一个缺陷极为困难, 因为开发人员不得不反复修改代码。
- 随着全球各地不断引入新功能, 在单一位置同步扩展所有功能变得极难应对。
- 可用性风险: 单体式代码库中一旦出现任何回归问题, 便可能导致整个系统 (本例中即优步全部服务) 宕机。
- 高风险、高成本的部署。由于频繁需要回滚, 此类部署过程痛苦且耗时。
- 关注点分离不充分。面对庞大的代码库, 难以维持良好的关注点分离。在指数级增长的环境中, 追求效率有时会导致逻辑与组件之间的边界模糊不清。
- 执行效率低下。这些问题叠加在一起, 使得团队难以自主或独立地开展工作。

C. Solution

To avoid such problems, Uber decided to change its design and follow Amazon, Netflix, Twitter and many other hyper-growth companies. Thus, Uber decided to break its monolithic architecture into multiple codebases to create microservice architectures.

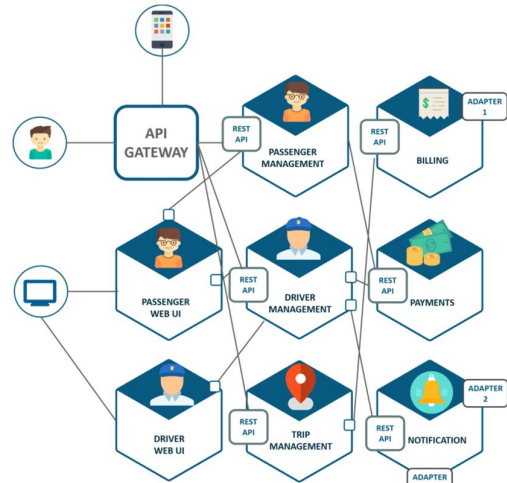


Figure 4: Uber's Microservice architecture [6]

The above diagram depicts Uber's Microservice architecture.

- The major change we see here is the introduction of the API gateway through which all drivers and passengers are connected. From the API gateway, all internal points like passenger management, driver management, trip management and others are connected.
- Units are separate deployable units that perform separate functions.
- For example: if you want to change anything in Billing Microservices, you only need to deploy Billing Microservices and no other deployments.
- All attributes are now measured individually, meaning that the interdependence in each attribute is removed.

For example, we all know that the number of people looking for a cab is actually higher than the number of people actually booking and paying for a cab. From this we conclude that the number of processes working on the travel management micro service is more than the number of processes working on payment.

D. Results

Uber deployed so many microservices till now. Following picture shows its growth:

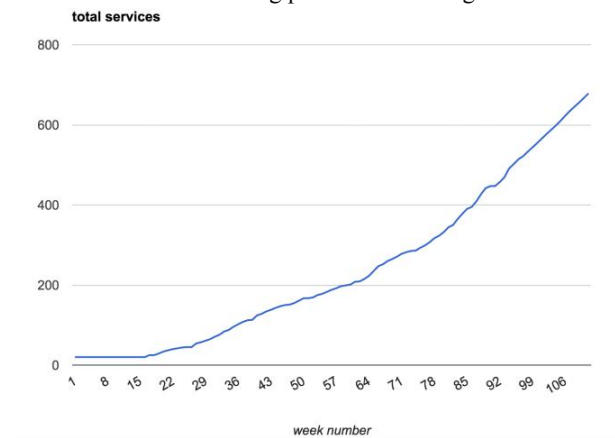


Figure 5: Uber's Microservices deployment growth [6]

C. 解决方案

为避免此类问题，Uber 决定调整其设计思路，效仿 Amazon、Netflix、Twitter 以及众多其他高速成长型企业。因此，Uber 决定将原有的单体架构拆分为多个代码库，构建微服务架构。

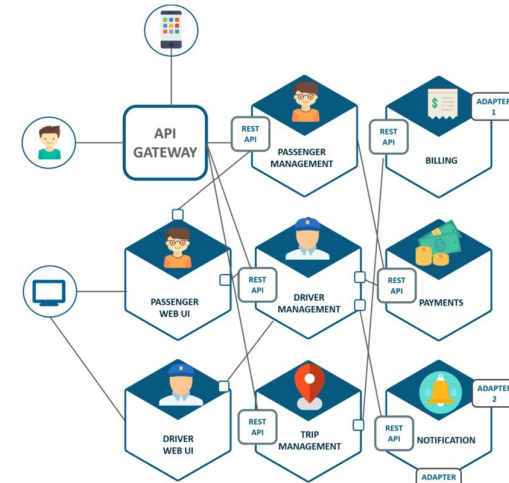


图4: Uber 的微服务架构 [6]

上图展示了 Uber 的微服务架构。

- 此处最显著的变化是引入了 API 网关，所有司机和乘客均通过该网关进行连接。从 API 网关出发，可连接所有内部模块，例如乘客管理、司机管理、行程管理等。
- 单元是彼此独立、可单独部署的模块，各自执行不同的功能。
- 例如：若您需要修改计费微服务中的任何内容，则只需重新部署计费微服务，无需部署其他服务。

- 所有属性如今均被单独度量，这意味着各属性之间的相互依赖关系已被消除。

例如，众所周知，寻找出租车的人数实际上高于实际预订并支付车费的人数。由此我们推断，处理行程管理微服务的进程数量多于处理支付服务的进程数量。

D. 结果

优步（Uber）迄今已部署了大量微服务。下图展示了其发展历程：

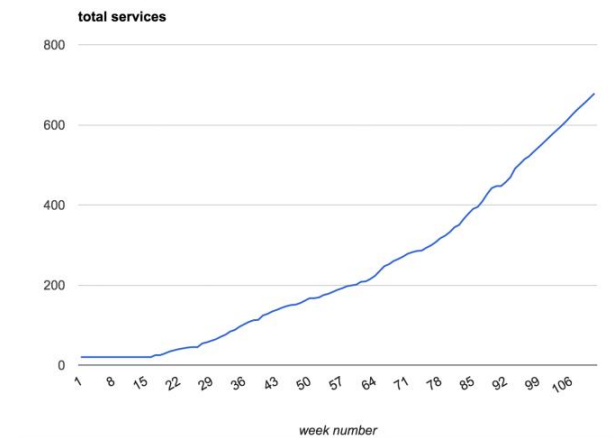


图5: Uber 微服务部署规模增长 [6]

Matt Ranney is the Chief Systems Architect at Uber and was previously a founder and CTO of Voxer. At QCon San Francisco, he gave a talk called Scaling Uber [6].

Scaling from monolithic to microservices: Horizontal scaling means scaling by adding more machines to your pool of resources (also described as “scaling out”), whereas Vertical scaling refers to scaling by adding more power (e.g. CPU, RAM) to an existing machine (also described as “scaling up”).

Uber used Horizontal scaling. Currently Uber supports Java, Python, Node.js , Go.

3.2 Microservices at Otto.De

OTTO has been one of the most successful e-commerce companies for over 60 years and the largest online retailer of fashion and lifestyle products for end customers in Germany.

With a turnover of over 2.563 billion euros and 1 million daily visitors in the business year 2015/2016, otto.de is one of the largest online stores in Europe. In 2011, Otto relaunched its e-commerce software from scratch. The drivers for this decision were primarily non-functional requirements such as scalability, performance and fault tolerance. In terms of scalability and agility, they weren't just thinking about technical scalability in terms of load or data. In particular, a measurable solution was needed in relation to the number of teams and / or developers working on the software at a given time. In addition, it was planned to practice DevOps with continuous deployment, in order to deliver features quickly to customers.

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 6.

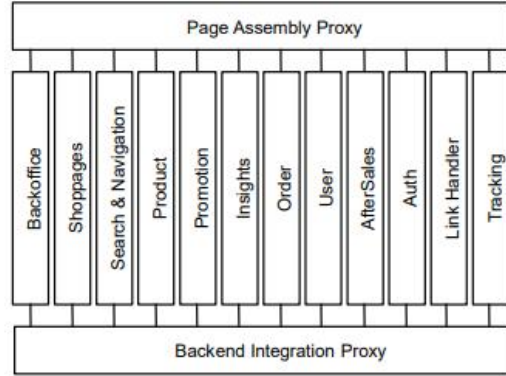


Figure 6: Current Vertical Decomposition at otto.de [7]

Vertical is part of the platform responsible for single bound context in the business domain [8]. Verticals can be as small as microservices, but most of the time, they are more rugged. Vertical communication is allowed only by accessing the REST API using the "Backend Integration Proxy" in the background - see Figure 1. This makes it easy to make sure that slow or unavailable apps can't destroy other apps or the entire store with a snowball effect.

The vertical adheres to the "nothing to share" principle: they do not share states, there are no infrastructure on either side of the two proxies, no databases or other shared resources. Does not use vertical HTTP sessions, shared cache or similar. Only a very limited number of client-side states (using cookies or local storage) are shared across different systems to gain a general understanding of who is entering the store. The major advantages of a shared-nothing architecture are excellent horizontal scalability and improved defect-tolerance. The reason for this is obvious: if two elements do not share anything, they cannot have a negative effect on each other.

马特·兰尼 (Matt Ranney) 现任优步 (Uber) 首席系统架构师, 此前曾是Voxer公司的联合创始人兼首席技术官。在QCon旧金山大会上, 弗朗西斯科 (Francisco) 曾发表题为《扩展优步 [6]》的主题演讲。

从单体架构向微服务架构演进: 水平扩展指通过向资源池中增加更多机器来实现扩展 (也称为“向外扩展”), 而垂直扩展则指通过为现有机器增加更多计算能力 (例如CPU、内存) 来实现扩展 (也称为“向上扩展”)。

优步采用的是水平扩展方式。目前优步支持Java、Python、Node.js和Go语言。

3.2 Otto.de的微服务架构

OTTO 是一家拥有逾 60 年历史的最成功电子商务公司之一, 同时也是德国面向终端消费者的时尚与生活方式类产品领域规模最大的在线零售商。

在 2015/2016 财年, otto.de 实现了超过 25.63 亿欧元的营业额, 日均访问量达 100 万人次, 是欧洲规模最大的在线商店之一。2011 年, Otto 从零开始全面重构其电子商务软件系统。推动这一决策的主要因素是非功能性需求, 例如可扩展性、性能及容错能力。在可扩展性与敏捷性方面, Otto 所考虑的不仅限于负载或数据层面的技术可扩展性; 尤其重要的是, 还需提供一种可量化评估的解决方案, 以应对同时参与软件开发的团队数量和/或开发人员数量的增长。此外, 公司还计划践行 DevOps 理念并实施持续部署, 从而快速向客户交付新功能。

最初采用的方案看似颇为独特, 但最终却取得了巨大成功: Otto 并未组建单一开发团队来构建全新的电商平台, 而是主动运用康威定律 (Conway’s Law), 自始即由四个彼此独立的团队同步启动开发工作。因此, 他们并未构建一个单一的单体式应用, 而是打造了一个垂直分解的系统, 该系统由四个松耦合的应用组成, 分别负责商品、订单、促销以及搜索/导航功能。在随后几年中, Otto 又陆续成立了更多团队与系统。如今, 如图 6 所示, 共有 18 个团队在 12 个所谓“垂直领域”内协作开发 45 个不同的应用。

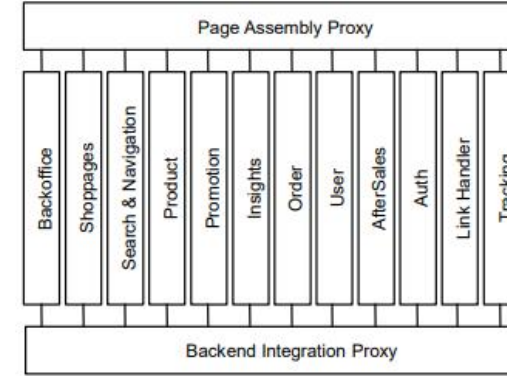


图6: otto.de 当前的垂直化拆分结构 [7]

垂直模块是平台的一部分, 负责业务领域中单一的边界上下文 [8]。垂直模块可以小至微服务, 但大多数情况下其粒度更粗、更稳健。垂直模块之间的通信仅允许通过后台的“后端集成代理”调用 REST API 实现——参见图1。这种机制可轻松确保响应缓慢或不可用的应用程序不会因雪球效应而影响其他应用程序或整个商城。

垂直架构遵循“无需共享”原则: 不共享任何状态, 两个代理之间不存在任何基础设施 (如数据库或其他共享资源)。不使用垂直 HTTP 会话、共享缓存或类似机制。仅通过客户端状态 (例如 Cookie 或本地存储) 在不同系统间共享极少量信息, 以大致了解访问店铺的用户身份。无共享架构的主要优势在于卓越的水平扩展能力以及更强的容错性。原因显而易见: 若两个组件之间完全不共享任何内容, 它们便无法相互产生负面影响。

A. Integrating Verticals on otto.de

All shop pages have different vertical pieces: preview of shopping cart, navigation structure, maybe some products or other parts. To assemble these pieces, the following principles are used for "Page Assembly Proxy" [9]:

- Pieces that are not part of the primary content or pieces that are initially invisible are assembled on the client side using AJAX [10]]. The shopping cart preview, for example, is one of those features that is included on almost every page
- The primary content is aggregated on the server side using Edge Side Includes [11] resolved by Varnish Reverse Proxy (www.varnish-cache.org).

In this way, verticals are integrated into a one-page website on the user interface. Despite the backend, users experience the store as a consistent entity.

B. Communication Among Verticals at otto.de

All verticals contain redundant data using pull-based data replication. This ensures that the content can be delivered without the other vertical access during the vertical request. On otto.de, the pull principle is applied in conjunction with Atom feeds (tools.ietf.org/html/rfc4287) via the Apache Kafka High-throughput Distributed Messaging System (kafka.apache.org).

C. Vertical and Microservices on otto.de

Microservices, like other software components, must be designed for proper granularity [12]. Vertical domains, as described above, may be small enough to be executed as a microservice - but they can also be large. Thus, sometimes those verticals need to be further refined: if possible, by extracting independent new features from the existing code into a new vertical (ideally as a microservice) or by cutting verticals into a distributed system of micro-services.

D. Scaling Delivery Pipeline at otto.de:

For frequent and automatic deployment, continuous deployment pipelines [13] are used for each application. Each vertical commit is first checked, compiled, packaged, deployed, and tested in a continuous-integration phase. After all the tests are passed, the container is deployed to the next stage, called testing. This phase is used to run load and integration tests. A second set of automated (and some manual) integration tests is implemented to ensure compatibility with newer and older versions of deployed software. The final step is to deploy application directly into the atmosphere. Due to the large number of deployment pipelines on otto.de, pipeline are implemented, described and operated with the internal domain-specific language LambdaCD (www.lambda.cd). Because the LambdaCD pipeline is nothing more than a microservices responsible for creating, testing, and deploying a single application, they operate in the same infrastructure as other microservices. They can be tested, run locally on notebooks without any additional continuous-integration or application server, and - in particular - they can be debugged just like any other software system.

E. Agility and Reliability on otto.de:

On otto.de, most microservices are deployed fully automatically after each push on the version control system. Automation is the key to DevOps success: automatic building of systems out of version management repositories; Automated implementation of unit tests, integration tests, and system tests; Automated deployment in testing and production environments. Since the beginning of 2015, more and more micro services have been launched in Vertical. Meanwhile, the number of direct deployments has increased from 40 to more than 500 deployments per week, which says that scalability has increased. Figure 7, the blue line, shows the number of live deployments per week for the years 2015-2017. At the same time, reliability is maintained and improved: the number of live events remains the same, very low levels; See Figure 7, red stripes. Incidents have been calculated since 2015, thus no incident data is available for 2014. However, this suggests that the quality assurance measures implemented for continuous integration and deployment are indeed effective for reliability.

A. 在 otto.de 上整合垂直业务模块

所有店铺页面均由不同的垂直模块组成，例如购物车预览、导航结构，以及部分商品或其他组件。“页面组装代理” [9] 在整合这些模块时，遵循以下原则：

- 不属于主要内容的组件，或初始状态下不可见的组件，均通过 AJAX 在客户端进行组装[10]]。例如，购物车预览便是此类功能之一，几乎在每个页面中都会被嵌入。
- 主要内容则通过边缘端包含（Edge Side Includes，ESI）技术在服务端进行聚合[11]，并由 Varnish 反向代理服务器（www.varnish-cache.org）解析。

通过这种方式，垂直业务模块被整合至用户界面的单页网站中。尽管后端架构复杂，用户所感知到的却是一个统一、连贯的在线商店。

B. otto.de 网站各垂直业务域之间的通信

所有垂直业务域均采用基于拉取（pull-based）的数据复制方式，其中包含冗余数据。这确保了在处理某一垂直业务域的请求时，无需访问其他垂直业务域即可交付内容。在 otto.de 网站上，拉取机制通过 Apache Kafka 高吞吐量分布式消息系统（kafka.apache.org）结合 Atom 订阅源（tools.ietf.org/html/rfc4287）实现。

C. otto.de 网站上的垂直业务域与微服务

微服务与其他软件组件一样，必须针对适当的粒度进行设计 [12]。如前所述，垂直业务域可能足够小，可直接作为微服务运行；但也可规模较大。因此，有时需要对这些垂直业务域作进一步细化：若可行，可从现有代码中提取独立的新功能，构建为新的垂直业务域（理想情况下以微服务形式实现）；或可将垂直业务域拆分为由多个微服务组成的分布式系统。

D. otto.de 网站交付流水线的规模化扩展：

为实现高频次、自动化的部署，每个应用程序均采用持续部署流水线[13]。每次针对垂直业务域的代码提交，首先在持续集成阶段完成检查、编译、打包、部署及测试。待全部测试通过后，容器将被部署至下一阶段——测试阶段。该阶段主要用于执行负载测试和集成测试。此外，还实施了第二套自动化（以及部分人工）集成测试，以确保新旧版本已部署软件之间的兼容性。最终步骤是将应用程序直接部署至生产环境。鉴于 otto.de 网站上存在大量部署流水线，这些流水线均使用公司内部的领域专用语言 LambdaCD（www.lambda.cd）进行实现、描述与运维。由于 LambdaCD 流水线本质上即为一种微服务——其职责是创建、测试并部署单个应用程序——因此它们与其他微服务共用同一套基础设施。这些流水线可像其他软件系统一样接受测试，在笔记本电脑上本地运行（无需额外的持续集成服务器或应用服务器），尤其重要的是，也可像其他软件系统一样进行调试。

E. 在 otto.de 上的敏捷性与可靠性：

在 otto.de，大多数微服务在版本控制系统中每次提交后均实现全自动部署。自动化是 DevOps 成功的关键：系统自动从版本管理仓库构建；单元测试、集成测试及系统测试自动执行；测试环境与生产环境的部署也完全自动化。自 2015 年初起，越来越多的微服务在垂直业务线（Vertical）中陆续上线。与此同时，直接部署次数已从每周 40 次增至每周逾 500 次，表明系统的可扩展性显著提升。图 7 中的蓝色曲线展示了 2015 至 2017 年间每周实时部署数量的变化趋势。与此同时，系统可靠性不仅得以维持，而且持续改善：线上事件数量始终保持在极低且稳定的水平；参见图 7 中的红色条形图。自 2015 年起，公司开始统计线上事故（incidents）数据，因此 2014 年无相关事故记录。但这一趋势表明，为持续集成与持续部署所实施的质量保障措施确实在提升系统可靠性方面行之有效。

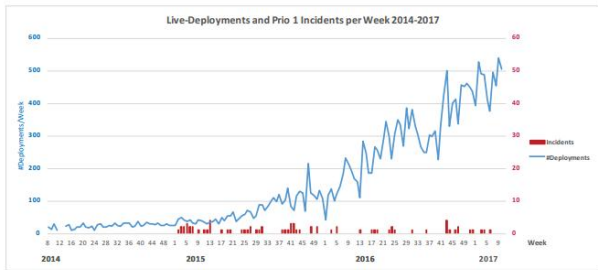


Fig. 2. Number of live deployments per week at otto.de over the last two years. Despite the significant increase of deployments, the number of live incidents remains on a very low level.

Figure 7: otto.de's Microservices deployment growth [7]

IV. CONCLUSION

In recent years, micro-services are becoming a large standard. As they have many benefits, they are super helpful within the era of e-commerce. We discussed the benefits of microservices than monolithic architecture. We have an overview of the Microservice Architecture used in e-commerce. Microservice Architectures can enable scalable, agile and reliable software system, such as otto.de's successful re-execution is mentioned above. Other E-Commerce or M-Commerce System, such as Amazon, Uber also adopted microservice architecture. We at otto.de discussed how coupling, integration, scalability of entire application as well as scalability for deployed pipelines for continuous delivery, and microservices are developed in teams. Micro-Services use genus API and Communication protocols to move along with each other, however they do not believe each other. Micro-Services Design means every micro-service separately developed, maintained and deployed by each individual the group. This type of single-responsibility result is also an alternative edge. The Applications made of micro-services are higher scales, because when you are required, you can scale them one-one. Besides, there is more failure tolerance for services.

V. ACKNOWLEDGMENT

I would like to express my deep and sincere gratitude to my research supervisor Mrs. Manali Patil, M.Sc. Professor, Information Technology, Patkar College for giving me the opportunity to do research and providing in valuable guidance throughout this research. I would also like to thank other professors, my parents who helped me a lot in finishing this research within the limited time frame.

REFERENCES

- [1]. MACH eCommerce Blog by Divante <https://www.divante.com/blog/what-are-microservices-introduction-to-microservice-architecture-for-ecommerce-2>
- [2]. Microservices Use Cases Blog by Alpacked.io, AUG 10, 2020, <https://alpacked.io/blog/microservices-use-cases/>
- [3]. Benefits of Microservices for Your Business Agility Blog by sumerge, November 2, 2020 <https://www.sumerge.com/benefits-of-microservices/>
- [4]. Microservices-Based Architecture in Ecommerce | Modular Solutions Blog by Science Soft <https://www.scnsoft.com/ecommerce/microservices>
- [5]. Blog on Microservices Zone,by Sahiti Kappagantula,Jul. 03, 18.<https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a>
- [6]. “Uber Rush and Rebuilding Uber’s Dispatching Platform” <https://qconnewyork.com/ny2015/system/files/presentation-slides/uberRUSH.pdf>
- [7]. "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce" , IEEE ICSAW Conference:2017
- [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/>.

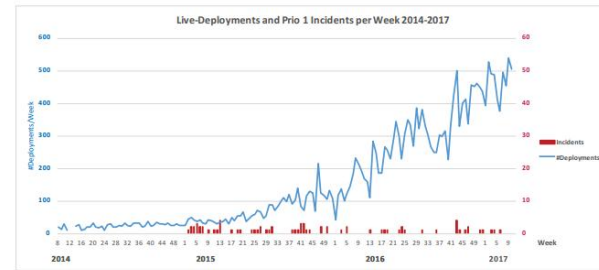


Fig. 2. Number of live deployments per week at otto.de over the last two years. Despite the significant increase of deployments, the number of live incidents remains on a very low level.

图7: otto.de 微服务部署规模增长 [7]

四、结论

近年来, 微服务正日益成为业界主流标准。由于其具备诸多优势, 在电子商务时代尤为实用。我们已探讨了微服务相较于单体架构的优势, 并概述了电子商务领域所采用的微服务架构。微服务架构能够支撑构建具备可扩展性、敏捷性与高可靠性的软件系统——例如上文提到的 otto.de 成功重构案例。其他电子商务 (E-Commerce) 或移动商务 (M-Commerce) 系统, 如亚马逊 (Amazon)、优步 (Uber), 同样采用了微服务架构。在 otto.de, 我们深入讨论了整体应用的耦合度、集成方式、可扩展性, 以及面向持续交付的部署流水线与微服务本身的可扩展性, 并强调微服务由不同团队分别开发。微服务通过通用 API 和通信协议彼此协作, 但彼此之间并不相互信任。微服务设计意味着每个微服务均由独立团队分别开发、维护与部署, 这种单一职责原则本身即构成一项关键优势。基于微服务构建的应用具备更高的可伸缩性: 当实际需要时, 可针对各服务单独进行弹性伸缩; 此外, 系统整体对服务故障亦具备更强的容错能力。

五、致谢

在此, 我谨向我的研究指导教师——帕特卡尔学院信息技术系硕士、教授玛娜莉·帕蒂尔女士, 致以最深切、最诚挚的感谢。感谢她为我提供了开展此项研究的机会, 并在整个研究过程中给予我宝贵而悉心的指导。同时, 我也衷心感谢其他各位教授以及我的父母, 他们在我完成本项研究的过程中给予了大力支持, 使我得以在有限的时间内顺利完成研究工作。

参考文献

- [1]. MACH eCommerce Blog by Divante <https://www.divante.com/blog/what-are-microservices-introduction-to-microservice-architecture-for-ecommerce-2>
- [2]. Microservices Use Cases Blog by Alpacked.io, AUG 10, 2020, <https://alpacked.io/blog/microservices-use-cases/>
- [3]. Benefits of Microservices for Your Business Agility Blog by sumerge, November 2, 2020 <https://www.sumerge.com/benefits-of-microservices/>
- [4]. Microservices-Based Architecture in Ecommerce | Modular Solutions Blog by Science Soft <https://www.scnsoft.com/ecommerce/microservices>
- [5]. Blog on Microservices Zone,by Sahiti Kappagantula,Jul. 03, 18.<https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a>
- [6]. “Uber Rush and Rebuilding Uber’s Dispatching Platform” <https://qconnewyork.com/ny2015/system/files/presentation-slides/uberRUSH.pdf>
- [7]. "Microservice Architectures for Scalability, Agility and Reliability in E-Commerce" , IEEE ICSAW Conference:2017
- [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.

[10]。 E. 沃伊乔夫斯基: 《AJAX: 使用异步JavaScript和XML构建网页》, Prentice Hall, 2006年。

[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.