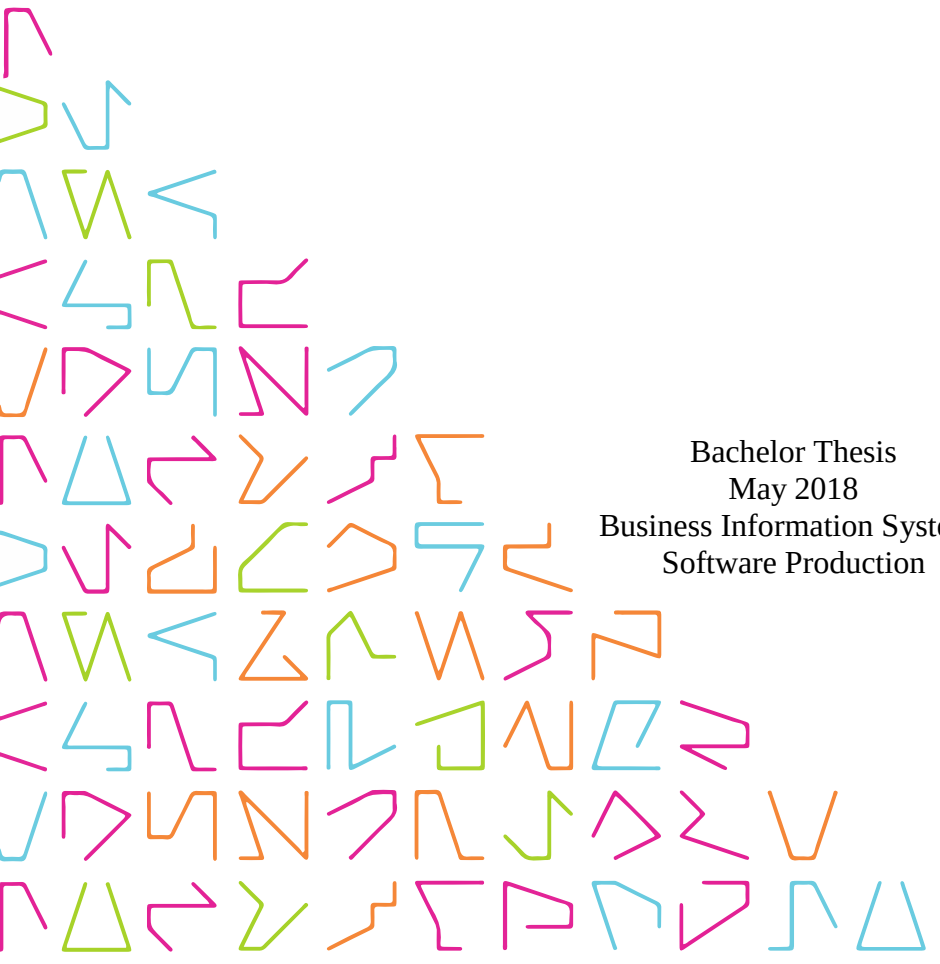
TAMPEREEN
AMMATTIKORKEAKOULU

MICROSERVICE ARCHITECTURE SUITABILITY FOR CONTEMPORARY SOFTWARE DEVELOPMENT

Tomy Salminen



Bachelor Thesis
May 2018
Business Information Systems
Software Production

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Business Information Systems
Software Production

SALMINEN, TOMY:
Microservice Architecture Viability for Contemporary Software Development

Bachelor's thesis 31 pages
May 2018

The objective of the thesis was to assess how well microservice architecture suits contemporary software development. Assessment is done through both technical and organizational lens'. The purpose was to implement a microservice for a customer of Futurice. The customer required a solution which would enable customer's in-house service to integrate to a 3rd party service.

The bachelor thesis is primarily a case study. Conclusions are based on observations during the project as well as literature sources.

Microservice architecture provides solutions issues commonly identified in large projects. Incrementally adopting new technologies, maintaining, scaling and quick development cycle is easier to achieve with microservice architecture, than with monolithic architecture. Microservice architecture makes testing more complicated and introduces complexity to system operations. To make full use of microservice architecture, organizations must be able to answer the challenges it presents.

Despite added complexity, microservice architecture overall provides relief to issues identified in web application development. It should be a default choice for medium to large projects. For smaller projects, it is be advised to assess the benefits and trade-offs of microservice case by case.

Key words: microservice, software architecture

TIIVISTELMÄ

Tampereen ammattikorkeakoulu
Tietojenkäsittely
Ohjelmistotuotanto

SALMINEN, TOMY:
Microservice Architecture Suitability for Contemporary Software Development
Opinnäytetyö 31 sivua
Toukokuu 2018

Opinnäytetyön tavoite oli selvittää mikropalveluarkkitehtuurin soveltuvuus nykyaikaiseen sovellusten kehitykseen kehittäjäkokemus ja organisaatio huomioiden. Tarkoitus oli toteuttaa Futurice Oy:n asiakkaalle mikropalvelu helpottamaan asiakkaan sisäisen ja kolmannen osapuolen palvelun välistä integraatiota. Opinnäytetyö on ensisijaisesti tapaustutkimus. Päätelmät nojautuvat asiakastyössä tehtyihin havaintoihin sekä kirjallisuuslähteisiin.

Mikropalveluarkkitehtuuri helpottaa erityisesti suurten projektien tyypillisiä ongelmia. Uusien teknologioiden asteittainen käyttöönotto, ohjelmiston ylläpidettävyys, skaalautuvuus ja nopea kehityssykli on helpompi saavuttaa mikropalveluarkkitehtuurilla, kuin monoliittisella arkkitehtuurilla. Mikropalveluarkkitehtuuri hankaloittaa testausta ja luo monimutkaisuutta. Saadakseen kaiken hyödyn irti mikropalveluista organisaatioiden on pystyttävä vastaamaan sen tuomiin haasteisiin.

Tuomistaan haasteista huolimatta mikropalveluarkkitehtuuri helpottaa verkkopalveluiden kehittämisessä ilmeneviä ongelmia. Se on lähtökohtaisesti suositeltava arkkitehtuuri keskikokoisille ja suurille verkkopalveluille. Pienempien palveluiden kohdalla on hyvä arvioida tapauskohtaisesti, onko projekti tarpeeksi laaja, jotta se hyötyy mikropalveluarkkitehtuurin tuomista hyödyistä.

Asiasanat: mikropalvelu, ohjelmistoarkkitehtuuri

ABBREVIATIONS AND TERMS

Anti-pattern	Commonly recurring counterproductive solution.
API	Abbreviation for Application Programming Interface. Defines how programs or parts of it intercommunicate.
CRM	Customer relationship management.
DevOps	Development and operations. A philosophy which advocates high automation and monitoring to accelerate development and operations.
End-to-end testing	Testing of application flow from start to finish.
HTTP	Hypertext transfer protocol. Widely used protocol for transferring data over networks.
Input/Output (I/O)	Communication between systems capable of processing information.
Integration testing	Testing interactions between modules.
Lean	Principle for managing a project, which aims to minimize waste while maintaining productivity.
Library	A collection of implementations.
Mental overhead	A phenomenon in which an individual must use an excessive amount of working memory.
Modularity	The extent which an application is partitioned into modules.
Module	An independent component, which performs one or more tasks.
MVP	Minimum viable product. The smallest set of features that make the product usable.
PaaS	Platform as a service. Service model in which platform is rented from a service provider.
Process	An instance of a computer program that is being executed.
Pull request	Git feature. Is a request for the destination to pull a set of changes into their tree.
REST	Architecture which defines how an API over HTTP should be like.

S3	A service provided by Amazon Web Services. Used for storing arbitrary data
SaaS	Software as a service. A service model in which software is rented from a service provider.
Scope	A feature set which must be met before the product can be considered complete.
Scope creep	Uncontrolled growth of project or product scope.
System operations	Act of running server(s).
Unit testing	Testing of individual parts of the code.

Table of contents

1 INTRODUCTION TO MICROSERVICE ARCHITECTURE

Bachelor thesis was ordered by Futurice. The implemented project was for an anonymous client of Futurice, henceforth simply referred to as the customer.

The objective of the thesis is to examine and compare microservice architecture to monolithic architecture, both from a technical point of view, as well as from an organization's point of view, and assess how well it suits modern software development needs. Technical point of view includes topics such as developer experience, scalability, testing and deployment. Organizational point of view focuses on assessing the kind of restrictions and value propositions each architecture imposes on organizations.

The purpose was to implement a microservice for a customer. The customer needed a way to transfer data between services, with the possibility to enrich the data by extrapolating values from incomplete data.

Bachelor thesis is structured so that monolithic architecture pattern is examined first. Answers to what it is, and what sort of issues are identified with it are provided. Then microservice architecture pattern is introduced and how it attempts to solve afore mentioned issues is discussed. Some of microservice architecture's issues are also highlighted. Thirdly, a case study is presented. Lastly, based on previous chapters some conclusions are drawn on when and how to choose a suitable architecture pattern depending on the project.

2 MONOLITHIC ARCHITECTURE

An application may be considered monolithic, if it is a complete, self-contained module run as a single unit (Rouse & Wigmore 2016, Amazon Web Services n.d.). The definition often also includes the notion that the application is designed without modularity (Richardson 2016), also sometimes referred to as single tiered application (Microsoft n.d.). For this thesis, it does not matter. So long as the application is deployable as a single unit, it will be considered a monolith. FIGURE 1 depicts how a monolithic web store might look like.

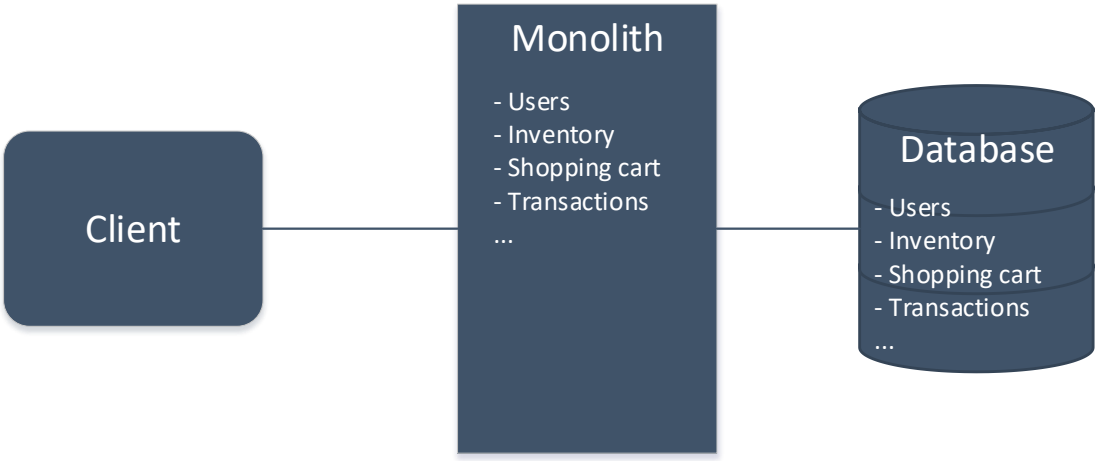


FIGURE 1. A simplified monolithic web store. Notice how it is a single instance, containing all the necessary components.

While still a viable pattern, projects that utilize monolithic architecture have several issues, which tend become increasingly pronounced as the project grows (Richardson 2016).

2.1 Technology lock-in

Monolithic architecture pattern forces a set technology stack to be used throughout the entire application (Richardson C n.d.). Different technologies can be viewed as tools, which require a certain type of task to be the most effective. (Mulesoft n.d.).

After locking in a technology stack, the project may end up using suboptimal tools, if the scope or requirements change, or the initial choice was a mistake. In case the chosen technology is to be changed, the whole application may need to be rewritten because not all technologies are compatible. (Richardson n.d.). The undertaking may be so laborious and cost ineffective, that it is no longer justifiable to undergo.

2.2 Maintainability

Maintainability may be challenging in case any part of the project needs to be refactored. As different components of the application are tightly coupled, changes to one part of the application may require cascading changes to other parts as well. (Rouse 2016). This may be somewhat controlled using a modular design within the monolith.

As a project grows, so does its complexity. A project may grow out to be so complex, that nobody is able to understand it in its entirety. This anti-pattern is referred to as “big ball of mud” (Mulesoft n.d.). In a scenario like this, new developers joining the project later in the process have a particularly steep learning curve before becoming productive members of the team.

A big ball of mud may lend itself to software entropy. In thermodynamics, entropy is a measure of disorder, which can only stay unchanged or increase (Jones A. 2017). A similar concept seems applicable to software. Unless a conscious effort is made, modifications to a software leads to greater disorder (Jacobson 1998).

If developers working on a big ball of mud cannot fully comprehend it, it is unlikely that code entropy can be controlled effectively. The result is accelerated accumulation of code entropy.

2.3 Scaling

Scaling means adjusting application's resources to meet current demands. Scaling an application can happen in one of two ways, vertically or horizontally, also known as scaling up or -out respectively. (Inviqa 2014).

Scaling is usually desirable since it allows the service to quickly respond to changed resource demands, without having to make changes to the software itself. It is not to be conflated with performance. Performance depicts how much work application can perform with given resources. Scaling means the degree which application can utilize resources of varying levels. (Inviqa 2014).

2.3.1 Vertical scaling

Vertical scaling means upgrading the hardware the application is run on (Inviqa 2014). By doing this, a single instance becomes more performant, as seen in FIGURE 2. For example, vertically scaling a REST API may lead it to be able to serve responses more quickly, thus it can deal with a larger number of clients before becoming overloaded.

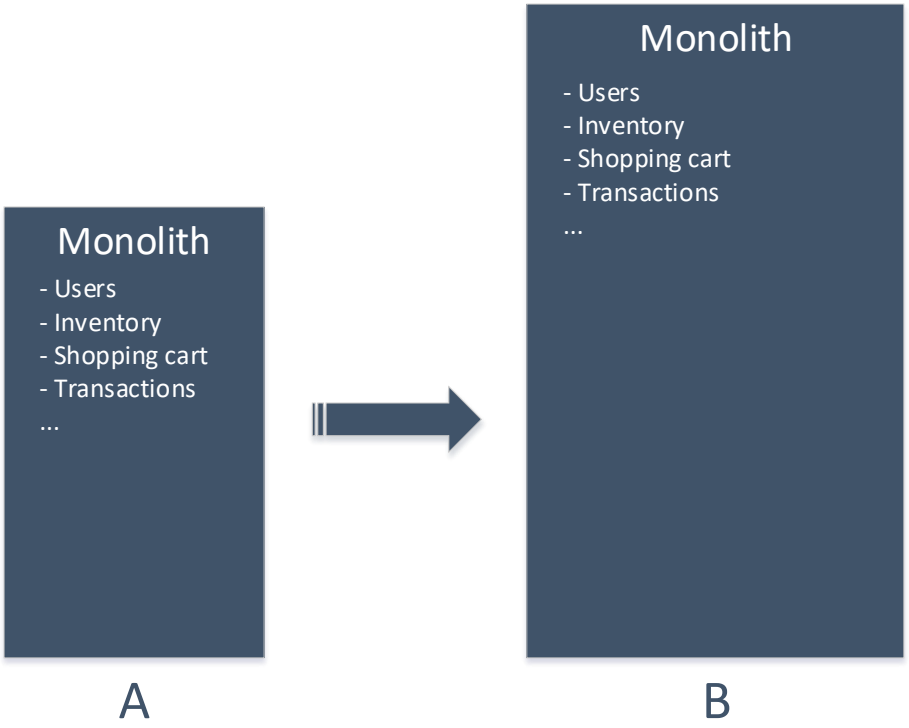


FIGURE 2 Example of vertical scaling. Larger size indicates higher performance. A is before-, B is after scaling up.

Vertical scaling is limited by the available hardware, effectively placing fixed limits on how much performance gain is possible to achieve this way. Performance ceiling is raised only as more powerful components become available. (Rouse 2014).

Another thing to consider is cost efficiency. Vertical scaling compared to horizontal scaling, which is discussed in the next chapter, may not be as cost effective. (DNS Made Easy 2013). Businesses will have to consider case-by-case if the cost of vertical scaling can be justified.

It is also worth keeping in mind that the machine is a single point of failure. Whenever its hardware configuration is being changed, any application it was previously running will be unavailable. Effectively this means that each time vertical scaling occurs, it will result in application downtime.

2.3.2 Horizontal scaling

Horizontal scaling is increasing the number of application instances that are run in parallel, as seen in FIGURE 3 (Inviqa 2014). Compared to vertical scaling, this is the more sustainable way to scale an application.

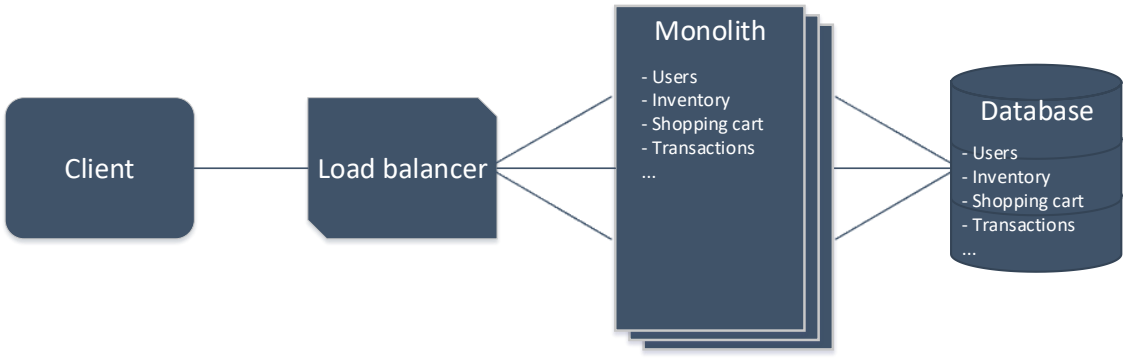


FIGURE 3. Example of horizontal scaling. Notice how there are multiple instances of monolith running concurrently.

Each instance can be run on a separate machine. This increases fault tolerance significantly by introducing redundancy. The machine the application is run on is no longer a single point of failure. In case it is taken down for maintenance, breaks, or is offline for any other reason, the application is still available. All incoming requests for the offline machine are redirected to any of the still operational machines (Rouse 2014). Distributing incoming calls to available machines is done by a load balancer (Citrix n.d.).

Applications that scale horizontally the best are stateless applications. Each incoming request can be handled only based on given input. History of previous requests and actions do not impact the handling of the current request. This way it doesn't matter which instance receives the request, or which instances may have handled any previous requests. There is no need for a shared resource to track client state, which could become a bottleneck as the number of instances grow. (Bartels 2009).

Typically, applications need to read and or write to a database, and often databases cannot be instance specific. Database connection may become a bottleneck for scaling an application, as each instance needs to perform I/O operations to the same database. (Bartels 2009). For the monolithic application, this is particularly problematic, as it always needs to have access to all data, as seen in FIGURE 3 on page 15.

While the monolithic application can potentially scale both horizontally and vertically, it is always a blanket upgrade. Scaling cannot be targeted to specific portions of the application. If a part of the application is identified as a bottleneck, it would have to be programmatically solved. Bottlenecks throughput cannot be increased in proportion to the rest of the application by scaling.

3 MICROSERVICE ARCHITECTURE

The basic premise of microservice architecture is to break down a monolithic application into smaller applications, each with a narrower scope, as seen in FIGURE 4. Microservice architecture doesn't have a universally agreed upon definition. However, for this thesis, definition coined in Microservice Architecture (Nadareishvili, Mitra, McLarty & Amundsen 2016) will be used. "A microservice is an independently deployable component of bounded scope that supports interoperability through message-based communication. Microservice architecture is a style of engineering highly automated, evolvable software systems made up of capability-aligned microservices." (Nadareishvili, Mitra, McLarty & Amundsen 2016).

The prefix "micro" in microservice does not imply complexity, number of lines of written code, or anything relating to the size of the service. A microservice can be a large application, but it only handles a set of features serving a well scoped business goal. (Nadareishvili, Mitra, McLarty & Amundsen 2016).

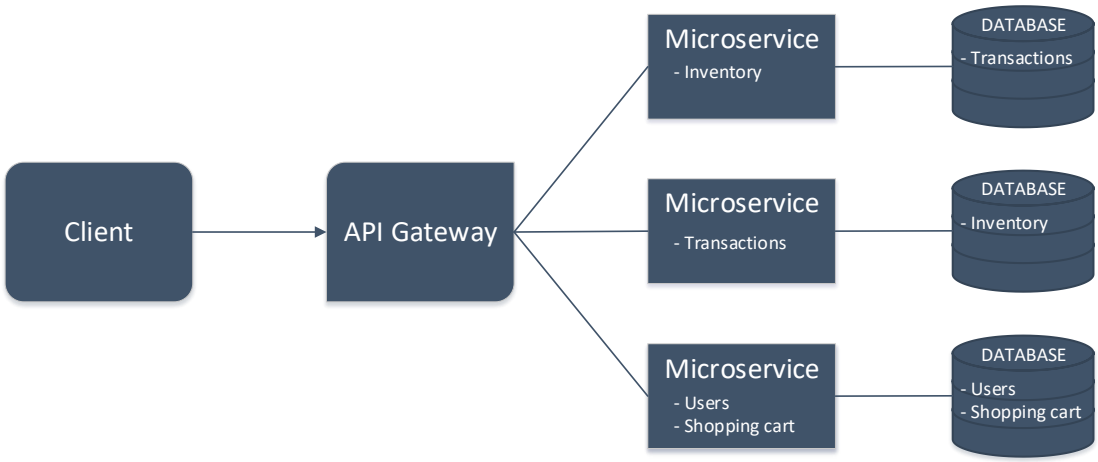


FIGURE 4. Example of how a monolith may be broken down into microservices.

A microservice architecture may leverage API gateway design pattern. Its purpose is to be the point of contact for the client (Feitosa 2018). The benefit is that the client will not have to know about how microservices are partitioned, at the cost of complexity.

3.1 Advantages

Microservice architecture attempts to provide solutions to problems identified in monolithic architecture. However, these advantages do not benefit all project equally. It should be evaluated if the pros outweigh cons for desired use case before choosing to proceed with microservice architecture.

3.1.1 Maintainability

Each microservice should have well-defined interfaces, which form an information boundary. A developer can work on a microservice without much knowledge beyond its boundaries (Nadareishvili, Mitra, McLarty & Amundsen 2016). This is especially helpful for developers joining the project later in the process by giving them a smaller scope to learn, enabling them to become productive in a much shorter time span compared to a monolithic project.

Strong interfaces also have the benefit of limiting cascading changes. Assuming the interface is not to be changed, cascading changes stop at the interface the latest. This also means is that each microservice can be rewritten or replaced entirely, without imposing changes to other microservices. (Mulesoft. n.d.).

Smaller scope means less mental overhead, and thus developers have an easier time creating software entropy neutral solutions. This, in turn, means that microservices are less likely to devolve into a big ball of mud. (Merson 2015).

3.1.2 Technology lock-in

One of the benefits of microservices, is that they intercommunicate via technology agnostic means, such as HTTP. If desired, this enables each microservice to be written in completely different technology, enabling every microservice to utilize the best tools for the job. (Nadareishvili, Mitra, McLarty & Amundsen 2016).

Due to modularity, as discussed in the previous chapter, it is possible to adopt new technologies incrementally. If the application was a monolith, the entire software would likely to be replaced in one go. However, with microservices changes can be rolled out a service at a time. This makes it easier to migrate between different technologies, if desired.

3.1.3 Scaling

While each microservice faces the same obstacles for scaling as a monolithic application, microservice architecture has some advantages. Each microservice can be scaled independently, as seen in FIGURE 5. If a microservice is bottlenecking the rest of the system, its throughput can be scaled in proportion to the rest of the services. (Merson 2015).

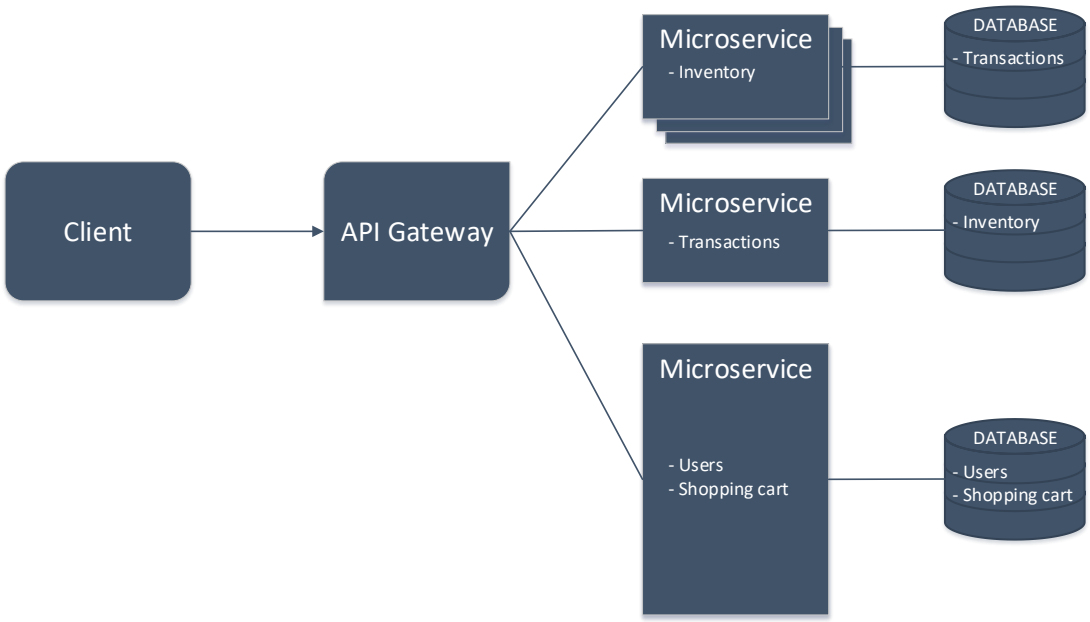


FIGURE 5. Example of how each microservice can be scaled individually.

Microservices are independent modules, with each their own databases. While a monolith requires access to all the data always, microservice architecture naturally breaks a single database into multiple, smaller ones. The benefit of this is that database I/O operations are distributed across multiple databases. (Feitosa 2018).

3.1.4 Development and deployment

Multiple microservices can be developed in parallel due to loose coupling. Work can be distributed across multiple teams more easily and each team may develop, build, test and deploy their work independently. This allows teams to release incremental changes in accelerated cycle, compared to if they were working on a monolith instead. (Kharenko 2015).

Microservices also offer the benefit of reduced build times. Since each microservice is its own module, only the module to which changes occur needs to be rebuilt. This reduces the need to batch changes into a larger set before building. (Merson 2015).

3.2 Criticism

While microservice architecture pattern has gained popularity since its inception, it has also garnered some criticism. Some are concerns are valid, while some stem from misguided choices regarding when and how to use microservices.

3.2.1 Nanoservice

When designing a microservice, it is important not to reduce its scope too much. If a microservice performs a very niche task, overhead from developing and upkeeping it may outweigh its utility. Arnon Rotem-Gal-Oz (n.d.) referred to such anti-pattern as nanoservice.

Nanoservice anti-pattern can be resolved by simply expanding microservice's scope. Either by assigning it more functionalities or by combining it with similar services. Alternatively, service may be repackaged into a library.

3.2.2 Complexity

Well-scoped microservices are simple, perform a clear task and usually are better optimized than their monolithic counterparts. However, microservices are criticized for introducing complexity due to inter-service calls happening over networks. Developers will have to think about e.g. added asynchronicity, latency and exception handling (Wootton 2014).

Fred Brooks (1986) argues that software has innate complexity, that cannot be removed without also removing functionality. Irakli Nadareishvili (2016) further elaborates that what microservices do, is shifting complexity from code implementation to system operations. The benefit being that system operations can be automated, while code implementation cannot. The result is neutral net complexity while gaining increased automation. (Nadareishvili 2016).

3.2.3 Latency

Decentralized applications typically suffer from higher response times compared to single process applications. Microservices need to intercommunicate over networks, instead of being able to rely on in-process calls, unlike monoliths. (Nolle n.d.). FIGURE 6 provides an example of how a monolith web application can handle a client request.

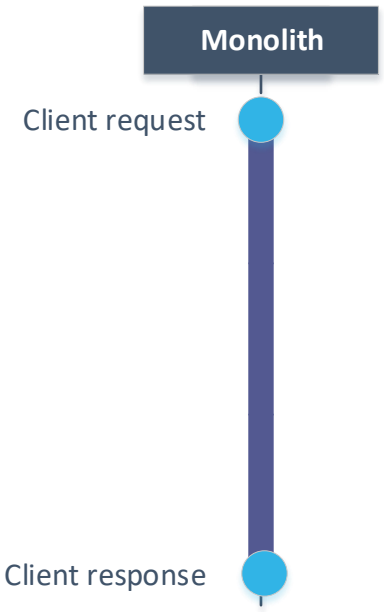


FIGURE 6. A monolith can process incoming the request without making additional network requests.

Distributed applications inevitably have measurably longer response times than monolithic applications. In FIGURE 7, each arrow from API Gateway to a microservice represents a network request. Each request introduces some amount of latency, which is higher than if the same request was made using an in-process call.

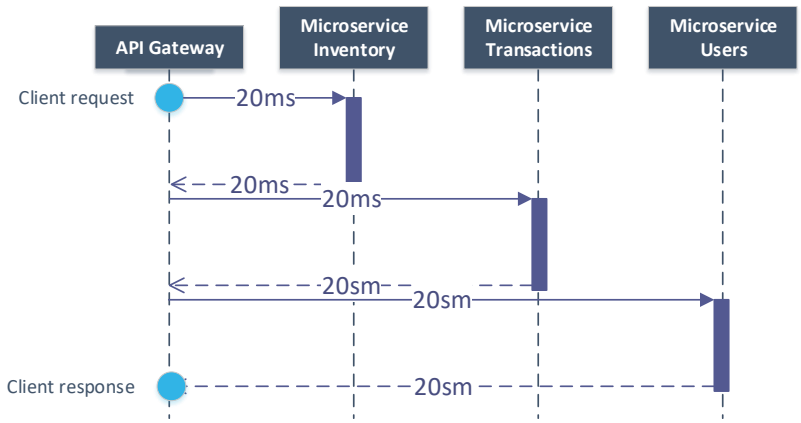


FIGURE 7. A gateway receives a client request but will have to make requests over networks to microservices which hold the relevant data for the response.

However, it can be argued that if latency is an issue to a point where user experience is affected, the problem is in how microservices were partitioned. Microservices may be too fine grained and numerous, resulting in an excessive amount of network calls. See chapter 3.2.1 for how a situation like this may be remedied.

3.2.4 Testing

Testing microservices can be more difficult than testing monolithic applications. While unit testing is largely unaffected, integration and end-to-end test suites can become more complex. (Wootton 2014).

If microservices are built using varying technologies, the test suite may need to support multiple runtime environments simultaneously (Merson 2015). Testing is not an insurmountable obstacle for microservices, but it will require work from DevOps engineers (Wootton 2014).

4 CUSTOMER PROJECT CASE STUDY

Futurice was tasked to deliver a microservice to be used for mapping data from a format to another. The customer needed to feed data from their system into a SaaS based CRM system. Next chapters asses the project circumstances and success.

4.1 Project justification, expectations and restrictions

The customer wanted to utilize a CRM system to enhance their customer retention. However, source data needed to be heavily modified so that it would be usable by the CRM system. The customer wanted to order a custom solution from a 3rd party provider, instead of building one in-house. Futurice was chosen because of previous successful projects and pre-existing domain knowledge.

The customer expected to have a fully automated solution, which would take input from multiple sources and convert it into a format supported by the CRM system. The solution would sometimes have to extrapolate values to populate fields to amend errors in source data.

The biggest restriction was source data quality. Our primary data source was a large data repository, which collected and standardized data from numerous and different subsystems. Fragmented subsystems, and the fact that the central data repository was still partly under construction, meant that the resulting data had, at times, significant gaps.

4.2 Implementation

This chapter is an overview of the tools and designs selected for the project, covering rationale for why they were chosen. Lastly, commentary on project's success is provided.

4.2.1 Architecture

A microservice architecture was chosen because customer's pre-existing applications already follow this design pattern. It also allowed Futurice project team to develop the application with independence. Architecture diagram can be seen in FIGURE 8.

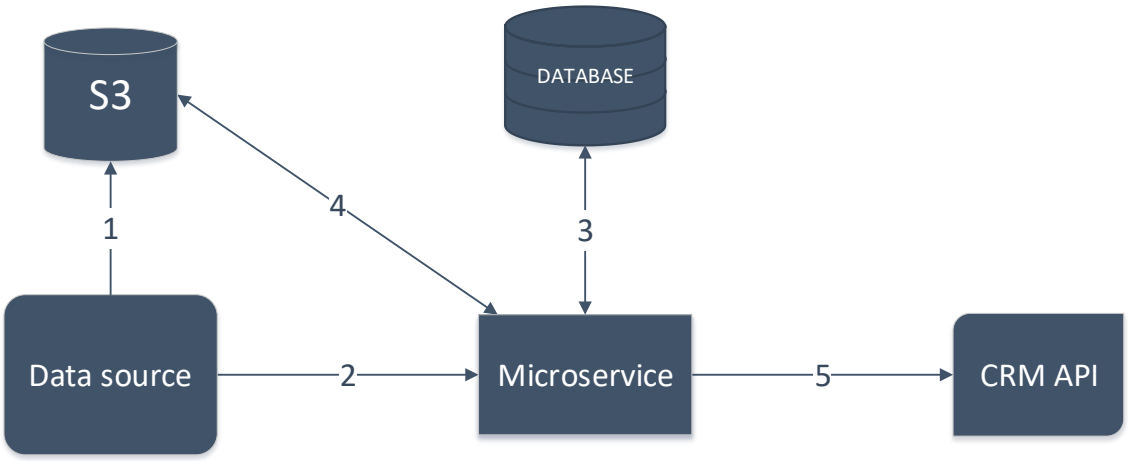


FIGURE 8. Simplified architecture depicting microservice dependencies to other services.

Each batch of data is handled in the following manner:

1. Data source exports a batch of data into S3.
2. Microservice is notified that new data is available.
3. Microservice checks previous notifications and determines if it safe to begin processing the newest batch.
4. If safe, data is fetched and processed.
5. Processed data is passed to CRM system API.

While it may seem so, the project is not actually a monolith. The goal was not to create a complete application consisting of multiple microservices but implement a single node in a web of microservices. Data source especially is a hub to which multiple other services integrate to.

4.2.2 Technologies

Microservice was written in JavaScript with Node.js as runtime environment. Node.js makes it possible to have JavaScript-based server-side applications (Node.js. n.d.). Among the main reasons for why it was chosen is Node Package Manager, or NPM for short. It is a tool for easily managing project dependencies (Node.js n.d.). Also, the project team and customer have had previous experience working with Node.js and JavaScript.

Express.js was used as the main framework. It is a widely used framework for Node.js. Express is a lightweight library which streamlines developing apps for Node.js by cutting down on boilerplate code and adding some frequently used features absent in Node.js. (Hahn 2016).

Microservice made use of some services provided by Amazon Web Services, or AWS for short. S3 was used as a temporary data storage. It is a service provided by AWS for storing arbitrary data. See FIGURE 8 how S3 is positioned in relation to Microservice.

Simple Queue Service, or SQS, was also used. SQS is a message queue, which handled messages sent from Data source to Microservice. While not explicitly drawn into the architecture diagram, SQS is responsible for handling messages generated in step 2 of FIGURE 8.

4.2.3 Infrastructure

Heroku is a PaaS built on top of AWS. The main strength of Heroku, and why it was selected as the hosting platform, is that it allows running applications with minimal focus on maintaining infrastructure.

Each application instance is run on a container, called dynos. Scaling an application happens by adjusting either resources reserved for each dyno or the number of dynos running in parallel. Both options result in minimal downtime, but since the application is non-critical, such down times are acceptable.

4.2.4 Continuous integration and testing

Travis CI is a continuous integration service and was used to run test suites on pushes and pull requests. The customer already had an existing subscription, which was a major factor for why it was used. No continuous integration tool was not used to automate deployment, which was done manually instead.

While 100% coverage was not achieved, unit- and integration tests were written to ensure that the system operated as designed. However end-to-end tests were lacking. Mostly because CRM system was not able to provide other environments apart from production.

4.3 Evaluating success and project retrospective

Ultimately, the success of the product cannot be evaluated at the time of writing, as the microservice has not yet been moved to production. However, as stated by the customer, the product met their expectations.

4.3.1 Developer experience

The project setting was challenging. Stakeholders consisted of multiple nationalities spread across at least three different sites. Difficulties in communication lead to misunderstanding, and this meant that the technologies and architecture needed to be agile. Requirements changed often after more clarifications were requested.

Microservice architecture complemented the setting well. Making changes and deployments could be made in a very short notice, without much coordination with other project teams. Being able to work independently allowed the Futurice team to concentrate on actual implementation with very little downtime from having to wait on other teams.

Continuous integration was convenient enough to setup and did not suffer from excessive complexity associated with microservices. Having a test suite automatically run often gave early warnings if breaking changes were being made.

4.3.2 Performance and scaling

The intention was that multiple instances of the microservice could temporarily be launched, in case of data queue starting to fill up. Each instance would capable of handling a data export independently. While this goal was achieved, it was realized that the CRM system API would not be able to handle data which was not in chronological order.

However, scaling is not a concern. Source system exports a data batch daily. Thus, the microservice needs to be able to process it in under 24 hours to keep the queue from building up. Load testing showed that it the microservice can handle a typical daily batch in a few minutes. Since queue max depth is only four days, a single instance is performant enough to clear accumulated data in well within the required time frame.

5 THOUGHTS ON CHOOSING AN ARCHITECTURE

Factors which contribute to choosing an architecture can be divided into categories. Following chapters provide insight regarding which architecture is preferable under which circumstances.

5.1 Time constraint

For-profit organizations require revenue to sustain themselves. The sooner the product is deployed to production, the sooner it can generate revenue.

Microservice architecture compared to monolithic architecture requires more planning. Defining APIs and partitioning software is a necessary step for a successful implementation. While monolith also requires planning, the extent is not quite as large. A team may start working on actual implementation sooner, meaning an MVP may be production ready sooner.

If the expected time delta is critical for the business' viability, it probably is best to start with a monolithic architecture and go in to production earlier. Doing this buys time and gives an opportunity to reassess architecture choices later. Some of the most successful online services, such as Amazon, have started their life as a monolith, but were later converted to use microservice architecture.

5.2 Expected scaling needs

It can sometimes be difficult to estimate the exact required performance needs, and so it is good to have the option to be able to seamlessly scale performance to meet the demand. Making scalable software requires conscious effort, draining often limited resources such as money and time. Teams will have to decide how much resources are spent on making the software scalable.

Theoretically, the more scaling the better. For that, microservices are the better choice, as is established in chapter 2.3. However, practically it does not make sense to create e.g. a web store capable of serving two billion customers a day, if realistically customer volume is in the thousands. Exercising healthy realism is advised.

Microservice architecture should be the default choice, if scaling is a concern. However, if scaling is a nonissue, the monolith is still a viable option. It would still require some other justification as well to be the more sensible choice.

5.3 Resource overhead

Organizations will have to asses, if they can deal with the kind of overhead developing microservices will impose. Developing and maintaining all the services may require more personnel, compared to if the project was implemented using a monolithic architecture. Depending on the size of the project, some specialty skills may be required, such as dedicated DevOps team.

Microservices are the recommended choice, given that organization is confident that resources at their disposal are sufficient. If there are doubts, the same advice holds true for when microservice ramp up time is too slow; start with a monolith and reassess the situation later.

5.4 Project size

For smaller projects, opting to go with a monolith is advised. A small project may be difficult to break down into microservices in a meaningful way. Microservice architecture benefits larger projects more than it does small ones. Likewise, monolith begins to exhibit negative aspects when the project is larger.

5.5 Summary

For making the most scalable software with up to date methods, microservice architecture is the go-to option. However, there are valid cases for when a monolith is the better alternative. It is best to assess the situation, before committing to a solution. Keeping the focus on the actual problem, rather than trying to solve things technology first, should be a priority for any organization.

The answer to the question, is microservice architecture a viable pattern for solving contemporary software development needs, is yes. Surveys done on this subject seem to support this conclusion. In 2017, 80% of the surveyed companies used, or intended to use, microservice architecture pattern (LeanIX 2017). This figure has gone up from 68% since 2015 (NGINX n.d.). While maybe not a perfect fit for every situation, microservice architecture does seem to provide the kind of solutions to problems encountered by organizations.

SOURCES

Amazon Web Services, n.d. What are Microservices? Documentation. Read 2.5.2018. https://aws.amazon.com/microservices/

Bartels A., 17.7.2009. Coding in the Cloud – Rule 3 – Use a "Stateless" design whenever possible. https://blog.rackspace.com/coding-in-the-cloud-rule-3-use-a-stateless-design-whenever-possible

Brooks F., 1986. No Silver Bullet—Essence and Accident in Software Engineering

Citrix, n.d. What is load balancing? Read 4.5.2018. https://www.citrix.fi/glossary/load-balancing.html

DNS Made Easy, 2013. Vertical And Horizontal Scaling. Read 8.5.2018. https://medium.com/@DNSMadeEasyBlog/vertical-and-horizontal-scaling-fdb9df55d51

Feitosa V, 2018. Microservice Patterns and Best Practices. Packt Publishing.

Hahn E., 2016. Express in Action - Writing, building, and testing Node.js applications.

Inviqa. 11.9.2014 Horizontally scalable web applications. Read 2.5.2018. https://inviqa.com/blog/horizontally-scalable-web-applications

Jacobson I. 1998. Object-oriented software engineering: a use case driven approach. Harlow: Addison-Wesley.

Jones A., ThoughtCo. 10.7.2017. How to Calculate Entropy. Read 2.5.2018 https://www.thoughtco.com/entropy-definition-calculation-and-misconceptions-2698977

Kharenko A., 9.10.2015. Monolithic vs. Microservices Architecture. Read 7.5.2018. https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

LeanIX, 2017. BEYOND AGILE: IS IT TIME TO ADOPT MICROSERVICES? Read 4.5.2018 https://cdn2.hubspot.net/hubfs/2570476/Sales%20info/leanIX_Microservices-Study.pdf

Merson P., SATURN Blog 5.11.2015. Microservices Beyond the Hype: What You Gain and What You Lose. Read 2.5.2018. https://insights.sei.cmu.edu/saturn/2015/11/microservices-beyond-the-hype-what-you-gain-and-what-you-lose.html

Microsoft, n.d. Three-tier Application Model. Read 2.5.2018. https://msdn.microsoft.com/en-us/library/aa480455.aspx

Mulesoft. n.d. Microservices vs Monolithic Architecture. Read 2.5.2018. https://www.mulesoft.com/resources/api/microservices-vs-monolithic

Nadareishvili I., 2016. Microservices shift complexity to where it belongs. Read 7.5.2018. https://www.oreilly.com/ideas/microservices-shift-complexity-to-where-it-belongs

Nadareishvili I., Mitra R., McLarty M. & Amundsen M., 2016. Microservice Architecture. O'Reilly Media, Inc.

NGINX, n.d. The Future of Application Development and Delivery Is Now. Read 4.5.2018. https://www.nginx.com/resources/library/app-dev-survey/

Node.js, n.d. About Node.js. Read 7.5.2018. https://nodejs.org/en/about/

Nolle T., n.d- Microservices challenges include latency, but it can be beat. Read 7.5.2018. https://searchmicroservices.techtarget.com/tip/Microservices-challenges-include-latency-but-it-can-be-beat

Richardson C, Microservices.io, n.d. Pattern: Monolithic Architecture. Read 2.5.2018. http://microservices.io/patterns/monolithic.html

Rouse M., TechTarget 2014. horizontal scalability (scaling out). Read 2.5.2018. https://searchcio.techtarget.com/definition/horizontal-scalability

Rouse M., TechTarget 2016. monolithic architecture. Read 8.5.2018. https://whatis.techtarget.com/definition/monolithic-architecture

Rouse M. & Wigmore I., TechTarget. 2016. monolithic architecture. Read 2.5.2018. https://whatis.techtarget.com/definition/monolithic-architecture

Rotem-Gal-Oz A. n.d. Services, Microservices, Nanoservices – oh my! Read 7.5.2018. http://arnon.me/2014/03/services-microservices-nanoservices/

Wootton B., 8.4.2014. Microservices - Not a free lunch! Read. 7.5.2018. http://highscalability.com/blog/2014/4/8/microservices-not-a-free-lunch.html