Aleksi Pekkala

Migrating a web application to serverless architecture

Master's Thesis in Information Technology

February 20, 2018

University of Jyväskylä

Department of Mathematical Information Technology

Author: Aleksi Pekkala

Contact information: alvianpe@student.jyu.fi

Supervisor: Oleksiy Khriyenko

Title: Migrating a web application to serverless architecture

Työn nimi: Web-sovelluksen siirtäminen serverless-arkkitehtuuriin

Project: Master's Thesis

Study line: Master's Thesis in Information Technology

Page count: 31+0

Abstract: This document is a sample gradu3 thesis document class document. It also functions as a user manual and supplies guidelines for structuring a thesis document.

The abstact is typically short and discusses the background, the aims, the research methods, the obtained results, the interpretation of the results and the conculsions of the thesis. It should be so short that it, the Finnish translation, and all other meta information fit on the same page.

The Finnish tiivistelmä of a thesis should usually say exactly the same things as the abstract.

Keywords: serverless, FaaS, architecture, cloud computing, web applications

Suomenkielinen tiivistelmä: Tämä kirjoitelma on esimerkki siitä, kuinka gradu3-tutkielmapohjaa käytetään. Se sisältää myös käyttöohjeet ja tutkielman rakennetta koskevia ohjeita.

Tutkielman tiivistelmä on tyypillisesti lyhyt esitys, jossa kerrotaan tutkielman taustoista, tavoitteesta, tutkimusmenetelmistä, saavutetuista tuloksista, tulosten tulkinnasta ja johtopäätöksistä. Tiivistelmän tulee olla niin lyhyt, että se, englanninkielinen abstrakti ja muut metatiedot mahtuvat kaikki samalle sivulle.

Sen tulee kertoa täsmälleen samat asiat kuin englannikielinen abstrakti.

Avainsanat: serverless, FaaS, arkkitehtuuri, pilvilaskenta, web-sovellukset

List of Figures

Contents

1 Introduction

Cloud computing has in the past decade emerged as a veritable backbone of modern economy, driving innovation both in industry and academia as well as enabling scalable global enterprise applications. Just as the adoption of cloud computing continues to increase, the technologies in which the paradigm is based have continued to progress. Recently the development of novel virtualization techniques has lead to the introduction of *serverless computing*, an architectural pattern based on ephemeral cloud resources that scale up and down automatically and are billed for actual usage at a millisecond granularity. The main drivers behind serverless computing are to both reduce operational costs by more efficient cloud resource utilization and to improve developer productivity by shifting provisioning, load balancing and other infrastructure concerns to the platform. (**buyya2017manifesto**)

As an appealing economic proposition, serverless computing has attracted significant interest in the industry. This is illustrated for example by its appearance in the 2017 Gartner Hype Technologies Report (**walker17gartnerHype**). By now most of the prominent cloud service providers have introduced their own serverless platforms, promising capabilities that make writing scalable web services easier and cheaper (**awslambda0218**; **google18cloudFunctions**; **ibm18cloudFunctions**; **microsoft18azureFunctions**). A number of high-profile use cases have also been presented in the literature (**cncf18serverlessWG**). **baldini17currentTrends** however note a lack of corresponding degree of interest in academia despite a wide variety of technologically challenging and intellectually deep problems in the space.

One of the open problems identified in literature concerns the discovery of serverless design patterns: how do we compose the granular building blocks of serverless into larger systems? (**baldini17currentTrends**) **varghese18next** contend that one challenge hindering the widespread adoption of serverless will be the radical shift in the properties that a programmer will need to focus on, from latency, scalability and elasticity to those relating to the modularity of an application. Considering this and the paradigm's unique characteristics and limitations, it's unclear to what extent our current patterns apply and what kind of new patterns are best suited to optimize for the features of serverless computing. The object of this thesis is to fill the gap by re-evaluating existing design patterns in the serverless context

and proposing new ones through an exploratory migration process.

1.1 Research problem

The research problem addressed by this thesis distills down to 4 different questions:

1. Why should a web application be migrated to serverless?
2. What kind of patterns are there for building serverless web application backends?
3. Do the existing patterns have gaps or missing parts, and if so, can we come up with improvements or alternative solutions?
4. How does migrating a web application to serverless affect its quality?

The first two questions are addressed in the theoretical part of the thesis. Question 1 concerns the motivation behind the thesis and introduces serverless migration as an important and relevant business problem. Question 2 is answered by surveying existing literature for serverless patterns as well as other, more general patterns thought suitable for the target class of applications.

The latter questions form the constructive part of the thesis. Question 3 concerns the application and evaluation of surveyed patterns. The surveyed design patterns are used to implement a subset of an existing traditional web application in the serverless architecture. In case the patterns prove unsuitable for any given problem, alternative solutions or extensions are proposed. The last question consists of comparing the migrated portions of the app to the original version and evaluating whether the posited benefits of serverless architecture are in fact realized.

1.2 Outline

The thesis is structured as follows: the second chapter serves as an introduction to the concept of serverless computing. The chapter describes the main benefits and drawbacks of the platform, as well as touching upon its internal mechanisms and briefly comparing the main service providers. Extra emphasis is placed on how the platform's limitations should be taken into account when designing web application backends.

The third chapter consists of a survey into existing serverless design patterns and recommendations. Applicability of other cloud computing, distributed computing and enterprise integration patterns is also evaluated.

The fourth chapter describes the process of migrating an existing web application to serverless architecture. The patterns discovered in the previous chapter are utilized to implemented various typical web application features on a serverless platform. In cases where existing patterns prove insufficient or unsuitable as per the target application's characteristics, modifications or new patterns are proposed.

The outcome of the migration process is evaluated in the fifth chapter. The potential benefits and drawbacks of the serverless platform outlined in chapter 2 are used to reflect on the final artifact. The chapter includes approximations on measurable attributes such as hosting costs and performance as well as discussion on the more subjective attributes like maintainability and testability. The overall ease of development – or developer experience – is also addressed since it is one of the commonly reported pain points of serverless computing (**van2017spec**).

The final chapter of the thesis aims to draw conclusions on the migration process and the resulting artifacts. The chapter contains a summary of the research outcomes and ends with recommendations for further research topics.

2 Serverless computing

This chapter serves as an introduction to serverless computing. Defining serverless computing succinctly can be difficult because of the relative immaturity of the field. The NIST definitions of cloud computing have yet to catch up with the technology (**nist11definitions**), and an effort to formalize and standardize serverless computing by the industry-headed Cloud Native Computing Foundation is still underway (**cncf18serverlessWG**). As a result the boundaries between serverless and other cloud computing terms are still somewhat blurred, and the terms seem to carry slightly different meanings depending on the author or context. To complicate matters further, serverless computing has come to appear in two different but overlapping forms. A multilayered approach is therefore in order.

We approach the formidable task of defining serverless by first taking a brief look at the history and motivations behind utility computing. After that we'll introduce the basic tenets of serverless computing, distinguish between its two main approaches and see how it positions itself relative to other cloud service models. This is followed by a more technical look at the most recent serverless model, as well as its major providers, use cases, security issues and economic implications. The chapter closes with notes on the drawbacks and limitations of serverless, particularly from the point of view of web application backends. This thesis' definition of serverless leans heavily on the CNCF Serverless Working Group's whitepaper (**cncf18serverlessWG**), the seminal introduction to the topic by **robert2016serverlessarchitectures** as well as a number of recent survey articles (**baldini17currentTrends**; **van2017spec**; **fox17**)

As a sidenote, although the earliest uses of term 'serverless' can be traced back to peer-to-peer and client-only solutions (**fox17**), we're dismissing these references since the name has evolved into a completely different meaning in the current cloud computing context. As per **robert2016serverlessarchitectures**, first usages of the term referring to elastic cloud computing seem to have appeared in around 2012.

2.1 Background

Utility computing refers to a model where computing resources are commoditized and delivered in a manner similar to traditional utilities such as water, electricity and telephony. Utilities are readily available to consumers at any time whenever required and billed per actual usage. In computing, this has come to mean on-demand access to highly scalable subscribtion-based IT resources. The availability of computing as an utility has enabled organizations to avoid investing heavily on building and maintaining complex IT infrastructure. (**buyya09cloud**)

This vision of utility computing can be traced all the way back to 1969 and ARPANET. Leonard Kleinrock, one of the chieft scientists in the project, is quoted as saying, "as of now, computer networks are still in their infancy, but as they grow up and become sophisticated, we will probably see the spread of 'computer utilities' which, like present electric and telephone utilities, will service individual homes and offices across the country" (**kleinrock03internet**). The creation of the Internet first facilitated weaving computer resources together into large-scale distributed systems. Onset by these discoveries, multiple computing paradigms have been proposed and adopted to take on the role of a ubiquitous computing utility, including cluster, grid, peer-to-peer (P2P) and services computing (**buyya09cloud**). Building on the various preceding paradigms, cloud computing has in the past decade revolutionized the computer science horizon and truly enabled the emergence of computing as an utility (**buyya2017manifesto**).

Cloud computing is by now a well-established paradigm that enables organizations to flexibly deploy their software systems over a pool of computing resources. Both major IT companies and startups see migrating on-premise legacy systems to the cloud as an opportunistic business strategy for gaining competetive advantage. Cost saving, scalability, and efficient utilization of resources as well as flexibility are identified as key drivers for migrating applications to the cloud. (**jamshidi13cloudmigrationreview**)

The first cloud providers were born out of huge corporations offering their surplus computing resources as a service in order to offset expenses and improve utilization rates. To address consumers' concerns about outages and other risks, cloud providers guarantee a certain level

of service delivery through so-called Service Level Agreements (SLA) that are negotiated between providers and consumers. (**youseff08cloudOntology**)

The key technology that enables cloud providers to transparently handle the consumers' requests without affecting its own usage of the resources is *virtualization*. Virtualization is one of the main components behind cloud computing, and the thing that sets it apart from grid computing. **sareen13cloudTypes** defines virtualization as using computer resources to imitate other computer resources or whole computers. This enables the abstraction of the underlying physical resources as a set of multiple logical virtual machines. Virtualization has three characteristics that make it ideal for cloud computing: partitioning supports running many applications and operating systems in a single physical system; isolation ensures boundaries between the host physical system and virtual containers; encapsulation enables packaging virtual machines as complete entities to prevent applicationss from interfering with each other.

recent advances in VM, containers, lead to... **bernstein14containers** on how the progression to increasingly lightweight and efficient virtualized and abstracted systems (i.e. virtual machines to containers to unikernels) is indicative of the rapid innovation taking place in the public cloud market.

Serverless platforms promise new capabilities that make writing scalable microservices easier and cost effective, positioning themselves as the next step in the evolution of cloud computing architectures. (**baldini17currentTrends**)

Another particularly important driver for is... Particular importance in the current energy-constrained environment. Data centers use on the order of 1-2% of global energy consumption, but server utilization rates are really low. (**horner16powerusage**)

2.2 Defining serverless

Fundamentally serverless computing is about building and running back-end code that does not require server management or server applications. The term itself can seem a bit disingenuous, since despite the name serverless computing obviously still involves servers. The

name – coined by industry – instead carries the meaning that the resources used by the application are managed by the cloud service provider. As tasks such as provisioning, maintenance and capacity planning are outsourced to the serverless platform, developers are left to focus on application logic. For the cloud customer this provides an abstraction where computation is disconnected from the infrastructure it is going to run on. (**robert2016serverlessarchitectures**; **cncf18serverlessWG**)

van2017spec further define serverless computing by three key characteristics:

1. Granular billing: the user of a serverless model is charged only when the application is actually executing
2. (Almost) no operational logic: operational logic, such as resource management and autoscaling, is delegated to the infrastructure, making those concerns of the infrastructure operator
3. Event-driven: interactions with serverless applications are designed to be short-lived, allowing the infrastructure to deploy serverless applications to respond to events, so only when needed

TODO: application architecture view on serverless (SOA -> microservices -> cloud functions)

Serverless computing has in effect come to encompass two distinct cloud computing models: Backend-as-a-Service (BaaS) as well as Function-as-a-Service (FaaS). The two serverless models, while different in operation as explained below, are nonetheless grouped under the same serverless umbrella since they both deliver the same main benefits: zero server maintenance overhead and the elimination of idle costs. (**cncf18serverlessWG**)

Backend-as-a-Service refers to an architecture where an application's server-side logic is replaced with external cloud services that carry out various tasks like authentication or database access (**buyya2017manifesto**). The model – also known as Mobile-Backend-as-a-Service (MBaaS) (**sareen13cloudTypes**) – is typically utilized in the mobile space to avoid having to manually set up and maintain server resources for the more narrow back-end requirements of a mobile application. The application's core business logic is implemented client-side and integrated tightly with third party remote application services. Since these API-based BaaS

services are managed transparently by the cloud service provider, the model appears to the developer to be serverless.

Function-as-a-Service is a more recent development: the first commercial FaaS platform, AWS Lambda, was introduced in November 2014 (**awslambda0218**). In the FaaS architecture an application's business logic is still located server-side. The crucial difference is that instead of self-managed server resources, developers upload small units of code to a FaaS platform that executes the code in short-lived, stateless compute containers in response to events (**robert2016serverlessarchitectures**). The model appears serverless in the sense that the developer has no control over the resources on which the back-end code runs. **albuquerque17faaspaas** note that the BaaS model of locating business logic on the client side carries with it some complications, namely difficulties in updating and deploying new features as well as reverse engineering risks. FaaS circumvents these problems by retaining business logic server-side.

Out of the two serverless models FaaS is the one with significant differences to traditional web application architecture (**robert2016serverlessarchitectures**). These differences and their implications are further illustrated in section 2.4. As the more novel architecture, FaaS is especially relevant to the research questions in hand and is thus paid more attention in the remainder of this thesis.

Another perspective on the difference between the two serverless models is to view BaaS as a more tailored, vendor-specific approach to FaaS (**van2017spec**). Whereas BaaS-type services function as built-in components for many common use cases such as user management and data storage, a FaaS platform allows developers to implement more customized functionality. BaaS plays an important role in serverless architectures as it will often be the supporting infrastructure (e.g. in form of data storage) to the stateless FaaS functions (**cncf18serverlessWG**). Conversely, in case of otherwise BaaS-based applications there's likely still a need for custom server-side functionality; FaaS functions may be a good solution for this (**robert2016serverlessarchitectures**). Serverless applications can utilize both models simultaneously, with BaaS platforms generating events that trigger FaaS functions, and FaaS functions acting as a 'glue component' between various third party BaaS components. **robert2016serverlessarchitectures** also notes convergence in the space, giving

the example of the user managemement provider Auth0 starting initially with a BaaS-style offering but later entering the FaaS space with a 'Auth0 Webtask' service.

It's worth noting that not all authors follow this taxonomy of FaaS and BaaS as the two subcategories of a more abstract serverless model. **baldini17currentTrends** explicitly raise the question on whether serverless is limited to FaaS or broader in scope, identifying the boundaries of serverless as an open question. Some sources (**hendrickson16openlambda**; **mcgrath17implement**; **varghese18next**) seem to strictly equate serverless with FaaS, using the terms synonymously. Considering however that the term 'serverless' predates the first FaaS platforms by a couple of years (**robert2016serverlessarchitectures**), it seems sensible to at least make a distinction between serverless and FaaS. In this thesis we'll stick to the **cncf18serverlessWG** definition as outlined above.

2.3 Comparison to other cloud computing models

Another approach to defining serverless is to compare it with other cloud service models. The commonly used NIST definition divides cloud offerings into three categories: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS) and Software-as-a-Service (SaaS), in order of increasing degree of abstraction of cloud infrastructure (**nist11definitions**). On this spectrum serverless computing positions itself in the space between PaaS and SaaS, as illustrated in figure 1 (**baldini17currentTrends**). Figure 2 illustrates how the two serverless models relate, with the cloud provider taking over a larger share of operational logic in BaaS. **van2017spec** note that there's some overlap and give examples of non-serverless products in both the PaaS and SaaS worlds that nonetheless exhibit the main characteristics of serverless defined in section 2.2.

Since the gap between PaaS and FaaS can be quite subtle it warrants further consideration. Indeed some sources, including **adzic2017serverless**, refer to FaaS as a new generation of PaaS offerings. Both models provide a high-level and elastic computing platform on which to implement custom business logic. There are however a number of substantial differences between the two models, which ultimately boil down to PaaS being an instance-based model (multiple server processes running on always-on server instances) as opposed to the on-

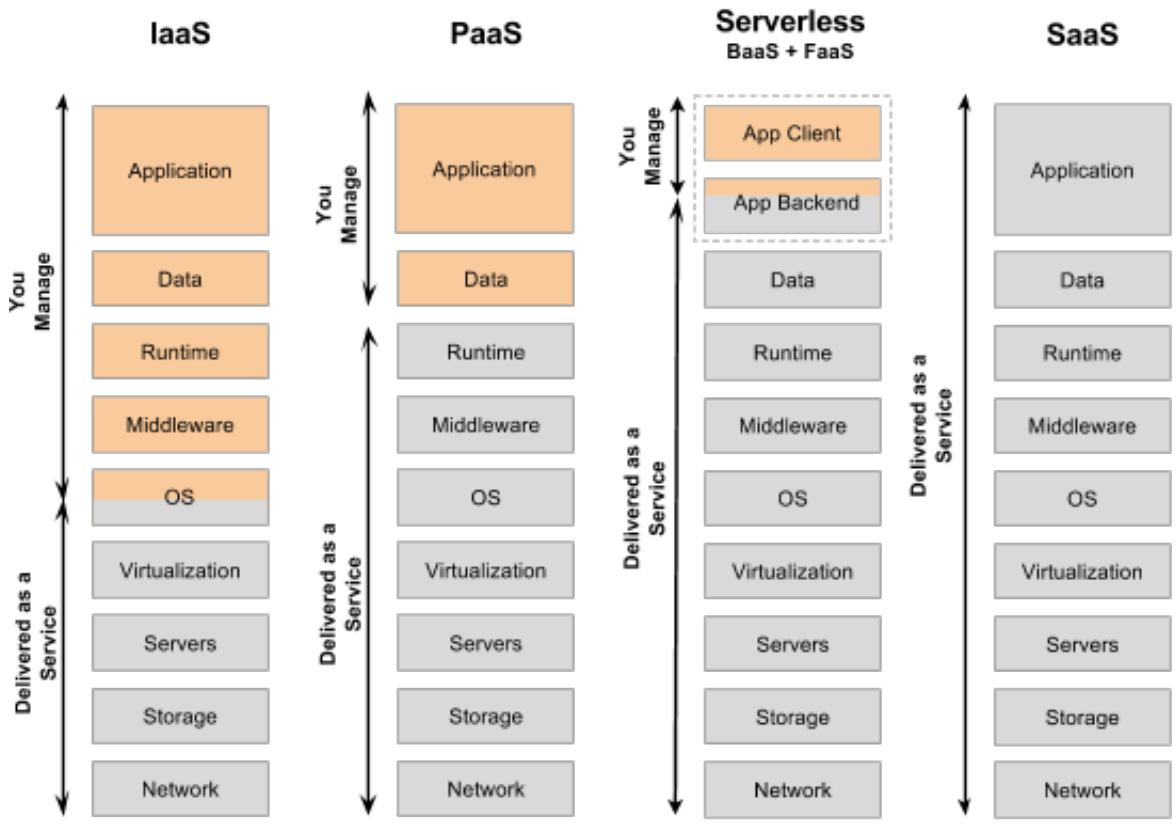

Figure 1. Degree of automation when using serverless (**wolf16serverless**)

demand resource allocation of FaaS – or as **robert2016serverlessarchitectures** puts it, most PaaS applications are not geared towards bringing entire applications up and down for every request, whereas FaaS platforms do exactly this.

albuquerque17faaspaas derive a number of specific differences between PaaS and FaaS in their comparative analysis. First of all the units of deployment vary: in PaaS applications are deployed as services, compared to the more granular function-based deployment of FaaS. Second, PaaS instances are always running whereas serverless workloads are executed on-demand. Third, PaaS platforms, although supporting auto-scaling to some extent, require the developer to explicitly manage the scaling workflow and number of minimum instances. FaaS on the other hand scales transparently and on-demand without any need for resource pre-allocation. Perhaps most important distinction lies in billing: PaaS is billed by instantiated resources whether they're used or not, whereas FaaS is billed per-event on a millisecond granularity. The analysis concludes that PaaS is well suited for predictable or constant work-

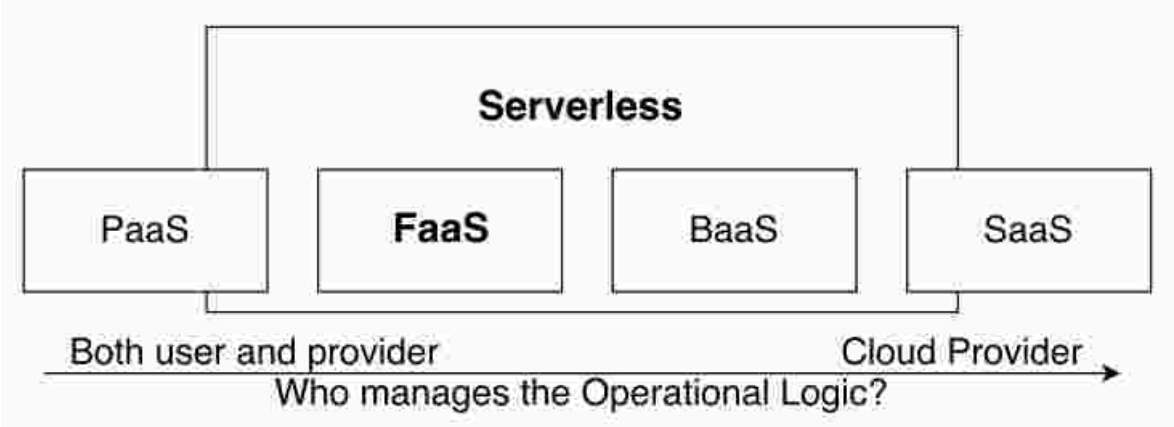

Figure 2. Serverless and FaaS vs. PaaS and SaaS (**van2017spec**)

loads with long or variable per-request execution times; FaaS in turn provides better cost benefit for unpredictable or seasonal workloads with short per-request execution times. It's also to be noted that PaaS doesn't suffer from limits on execution duration and many other restrictions of FaaS as described in section 2.9.

Another recent cloud-native technology is Container-as-a-Service (CaaS) compare to CaaS (**cncf18serverlessWG**). Buyya17 describes containers. **robert2016serverlessarchitectures** also has a few words on the topic.

2.4 Function-as-a-Service

Briefly describe the inner workings of a FaaS runtime. Describe the two supported execution models, synchronous and asynchronous, and how they relate to application design. The former is used to build a typical request-response flow, e.g. a REST API endpoint, whereas the latter relates to pub-sub and other event-driven flows. Give examples on the kind of triggers supported by serverless platforms (HTTP calls, messaging, database events, ...).

fox17 present a useful short definition of serverless as a cloud-native platform for short-running, stateless computation and event-driven applications which scales up and down instantly and automatically and charges for actual usage at a millisecond granularity.

spillner17snafu presents the design and implementation of a research-friendly FaaS run-

time. **mcgrath17implement** present the design of a novel performance-oriented serverless computing platform, discussing implementation challenges such as function scaling and container discovery, lifecycle, and reuse. **hendrickson16openlambda** present OpenLambda, an open-source platform for serverless computation, describing the key aspects of serverless computation and presenting numerous research challenges that must be addressed in the design and implementation of such systems.

2.5 Use cases

Introduce the main/intended use cases for serverless, as well as the more esoteric applications in literature.

mcgrath16cloudEventParadigms present two real-world applications utilizing cloud event architectures.

malawski17executescientific find that while serverless infrastructures are designed mainly for processing background tasks of Web and IoT applications, the simple mode of operation makes this approach easy to use and promising in scientific workflows too. **jonas17occupy** argue that a serverless execution model with stateless functions can enable radically-simpler, fundamentally elastic, and more user-friendly distributed data processing systems. **spillner18faaster** also find that in many domains of scientific and high-performance computing, solutions can be engineered based on simple functions which are executed on commercially offered or self-hosted FaaS platforms.

Introduce edge computing as a particularly strong driver for serverless. **glikson17devicelessedge** propose the novel paradigm of Deviceless Edge Computing that extends the serverless paradigm to the edge of the network, enabling IoT and Edge devices to be seamlessly integrated as application execution infrastructure. **nastic17analyticsedge** present a novel approach implementing cloud-supported, real-time data analytics in serverless edge-computing applications. **baresi17edgecomputing** propose a serverless architecture at the edge, bringing a highly scalable, intelligent and cost-effective use of edge infrastructure's resources with minimal configuration and operation efforts.

fouladi2017encoding present a serverless video-processing framework. **yan16chatbot** present the architecture and prototype of a chatbot using a serverless platform, where developers compose stateless functions together to perform useful actions. **ishakian17neural** evaluate the suitability of a serverless computing environment for the inferencing of large neural network models. **ast17webcomponent** describe an approach of how to utilize serverless computing to enable self-contained web components by deploying Web Component business logic as cloud-hosted functions.

2.6 Service providers

lynn2017preliminary provide an overview and multi-level feature analysis of seven enterprise serverless computing platforms. **baldini17currentTrends** also has a narrower survey of serverless platforms.

malawski18benchmark benchmark different cloud function providers.

2.7 Security

Address the security implications of serverless. Overall the consensus seems to be that compared to services maintaining their own servers and resources, serverless approach reduces the attack surface. However, research needs to understand the new security issues introduced by FaaS. For example, because the infrastructure can share resources among cloud-functions, what is the ideal security vs. performance/cost trade-off? (**van2017spec**)

2.8 Economics of serverless

eivy2017wary and **villamizar2016infrastructure** both focus on the economic aspects of serverless. **adzic2017serverless** explain how novel design patterns are used to significantly optimize costs – just running traditional web apps inside Lambda containers doesn't necessarily equate to savings. **adzic2017serverless** also report savings between 66 and 95% in two case studies, and present a handly table comparing hosting prices for intermittent service tasks. **spillner17exploiting** exploits the control plane of AWS Lambda to implement

services practically for free. **leitner16modelcost** present an approach to model deployment costs of AWS Lambda applications in real-time. **kuhlenkamp17costradamus** present another cost-tracing system that enables per-request cost-tracing for cloud-based software services, noting that cost testing should not only rely on isolated tests of single services but consider comprehensive end-to-end cost traces. **albuquerque17faaspaas** have a detailed price comparison running the same app in FaaS and PaaS.

2.9 Drawbacks and limitations

What to take into consideration when migrating to serverless?

lloydserverless analyze serverless performance and elasticity, identifying the cold start phenomenon. Differences in runtimes/languages, and larger library dependencies lead to slower starts. **oakes17pipsqueak** address the problem by caching package dependencies on platform-level.

baldini17trilemma identify three competing constraints in serverless function composition: functions should be considered as black boxes; function composition should obey a substitution principle with respect to synchronous invocation; and invocations should not be double-billed.

robert2016serverlessarchitectures, **adzic2017serverless** and **baldini17currentTrends** each list a number of limitations, including lack of strong SLA, vendor lock-in, short life-span, immature local development tools, statelessness and many others.

kuhlenkamp17costradamus discover two serverless cost tradeoffs: the retry cost effect and the cost ripple effect.

malawski18benchmark talk about interoperability challenges running a heterogeneous benchmark, as well as discussion on RAM allocation.

cncf18serverlessWG on technical immaturity, lack of interoperability, standardization, tools, documentation, best practices.

The need for circuit breakers (risk of DDoSing yourself) when interacting with non-serverless

components like a database. Mention novel cloud-native database services like Google's Cloud Spanner and AWS Aurora. Figure out a source for this – **hohpe2004enterprise** might have a relevant pattern.

Address maintainability: debugging serverless functions, following the flow of control can be tough.

Composing serverless functions is not like composing regular functions. All the difficulties of distributed computing – message loss, timeouts and others – apply and have to be handled. Possible solutions include retry policies, dead-letter queues and idempotent functions.

A full-fledged general-purpose serverless computing model is still a vision that needs to be achieved. (**buyya2017manifesto**)

Unlike the abstract concept of cloud functions, FaaS cannot completely abstract away all operation logic from the user. The FaaS user can still change parameters and configurations, such as the suggested memory size or number of CPUs of the underlying function host, which influence the operation of the deployed cloufunction. (**van2017spec**)

3 Serverless design patterns

Survey of serverless design patterns. **baldini17currentTrends** put the question as follows:

> Will there be patterns for building serverless solutions? How do we combine low granularity basic building blocks of serverless into bigger solutions? How are we going to decompose apps into functions so that they optimize resource usage? For example how do we identify CPU-bound parts of applications built to run in serverless services? Can we use well-defined patterns for composing functions and external APIs? What should be done on the server vs. client (e.g., are thicker clients more appropriate here)? Are there lessons learned that can be applied from OOP design patterns, Enterprise Integration Patterns, etc.?

3.1 Serverless patterns

A limited number of purely serverless design patters. **sbarski2017serverless** introduce five patterns – Command, Messaging Priority queue, Fan-out and Pipes and filters – but they seem mostly to be reinterpretations of classic Enterprise Integration Patterns.

Common low-level design patterns such as function chaining and fanout in Github.

mcgrath16cloudEventParadigms demonstrate a fan-out pattern, easily and performantly solving a large-scale image resizing task.

API Gateway and messaging patterns as described in platforms' own documentation. (**awslambda0218**)

adzic2017serverless suggest 3 methods for optimizing resource usage: use distributed authorization, let clients orchestrate workflows and allow clients to directly connect to AWS resources. The authors also discuss how paying only for actual utilization has two additional benefits of 1) removing incentives for bundling and 2) removing barriers to versioning: examples of serverless economics affecting architecture. Presenting these (or applications of these) as more formal patterns could be of some value.

robert2016serverlessarchitectures asks how big can FaaS functions get before they get

unwieldy? Assuming we can atomically deploy a group of FaaS functions what are good ways of creating such groupings - do they map closely to how we'd currently clump logic into microservices or does the difference in architecture push us in a different direction? Extending this further what are good ways of creating hybrid architectures between FaaS and traditional 'always on' persistent server components?

Extending this further what are good ways of creating hybrid architect

ast17webcomponent describe self-contained web components with serverless backends.

3.2 Enterprise Integration Patterns

hohpe2004enterprise present a number of asynchronous messaging architectures in the seminal book on EIP. While predating the whole serverless phenomenon the patterns are still relevant. Hohpe even demonstrated implementing one of his patterns on top of Google's serverless platform in a blog post.

3.3 FaaSification

spillner17transformpython describes an automated approach to transform monolithic Python code into modular FaaS units by partially automated decomposition. Doesn't really seem suitable for the web application migration process covered in this thesis but worth mentioning.

4 Migration process

Implement a subset of the target app in a serverless style, utilizing the surveyed patterns and keeping log of the tricky parts. In case the patterns prove unsuitable for the given problem, try to come up with an alternative solution.

Describe the web application to migrate.

Decide on the parts to migrate. Should demonstrate the features and limitations of serverless as outlined above. Possible features include

- A simple REST API endpoint to showcase API Gateway and synchronous invocation. Shouldn't require any big changes to application code.
- Interaction between multiple services to demonstrate distributed transactions.
- A scheduled (cron) event.
- Interacting with an external SaaS service like Twilio, Auth0. Demonstrate event-driven invocation.
- others?

5 Evaluation

Evaluation the outcome of migration process. Estimate the effects on performance and hosting costs. Weigh in on maintainability, testability, developer experience etc.

6 Conclusion

What can we conclude about the research questions? Mention limitations and further research directions.