IT UNIVERSITY OF COPENHAGEN

Thesis

Perspectives on Architecture, Evolution, and Future of Web Applications

Niels Roesen Abildgaard
<nroe@itu.dk>

Supervised by:
Björn Þór Jónsson
<bjth@itu.dk>

September 3, 2018

Abstract

The common view in web engineering of what constitutes a web application has been static for a long time. This thesis presents a variety of perspectives on web applications. Through a study ($n = 6$) we document the breadth of web application architectures in use; and present a novel method for documenting and evaluating architecture evolution, with an emphasis on common patterns of evolution. We offer arguments that connect web engineering to database and distributed systems research, establishing web applications as inherently weakly consistent systems; and present a taxonomy of data in web applications focusing on the refinement of data. The study is of limited size and poorly geographically distributed leading to poor generalizability of the results, but they offer an indication of both novel perspectives and novel findings that can be further investigated.

We contribute (i) a method for describing architecture evolution and finding common narratives across evolutions; (ii) the establishing of variety in web application architectures; (iii) identification of some common narratives of architecture evolution; (iv) the argument that existing frameworks for architectural change are not universally useful; (v) a connection between distributed systems research and web engineering; (vi) a connection between the concept of view maintenance and web engineering; and (vii) an indication of how data is stored and generated in web applications, and which approaches are unexplored.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The perspectives available to us decide what information we can learn. That is, the tools we use to explore a topic decide the realm of what is possible to uncover, and at the same time steer our view on things in such a way that they may blind us to other perspectives on the same subject matter. To avoid tunnel-vision in the exploration of a subject, employing several perspectives at once may reveal broader insights than would otherwise be possible. For several perspectives to be possible, however, several different tools that enable those perspectives must be available, too.[1]

Web engineering is a field that has previously been described as encompassing research from many other fields, such as "systems analysis and design; software engineering; hypermedia and hypertext engineering; requirements engineering; human-computer interaction; user interface development; information engineering; information indexing and retrieval; testing, modeling, and simulation; project management; and graphic design and presentation" [36]. This thesis focuses on the perspectives software architecture (and especially architecture evolution), distributed systems, and database research, while the subject under study from these perspectives is web applications. It can best be described as a web engineering thesis, but emphasis is on connecting research from other fields, rather than further specializing the web engineering niche.

Recent web engineering research has focused on the effort of constructing web applications, while many assumptions of what a web application is—that is, the variety of forms a web application may take—have been left fairly unchallenged. No research, to our knowledge, has focused on attempting to document what variety and breadth of web application architectures exist, and thereby what the established possibility space of web applications is in practice. Meanwhile, the use of web applications is changing and increasing, and the lack of reevaluation

[1] An example of this principle in practice is architectural views in the field of software architecture, which provide different perspectives on an architecture, in order to help the reader of such views understand different facts. "Modern systems are more than complex enough to make it difficult to grasp them all at once," so an all-encompassing view would not be useful [15].

of core ideas of web applications leaves the web engineering field at risk of working with an inaccurate representation of the practical field it purports to study.

Architecture evolution or architecture change research deals with changes to and maintenance of software, and the activities involved [13, 54, 67]. A wealth of frameworks for architectural work, and several for architectural change alone, exist. However, many of these frameworks take an approach that may be described as prematurely prescriptive [7, 67], based on a well-intentioned desire to provide actionable information, but with insignificant amounts of real-world data to support the theoretical constructions. A side effect of frameworks aiming to draw out a clear path of action for users of the framework is that they are often defined as providing the ultimate perspective on some subject—if a framework was not the ultimate, all-encompassing perspective, how can its prescription for future action be trusted? An alternative approach would see frameworks as complimentary, providing different perspectives that reinforce or inform each other, and thus provide a more nuanced view to the user of those frameworks, leaving decisions to the user.

Distributed computing and database research are both fields that see a lot of work. Distributed computing concerns making systems composed of several nodes work together in a way that provides incredible scalability and availability as well as low latency, while making informed tradeoffs in consistency and how network partitions will be dealt with [28, 29, 66]. Database research has some overlap, as databases may themselves be distributed systems. For this thesis, the work on the semantics and maintenance of structured data from the database field is particularly relevant [39, 43].

While both fields are connected to web engineering in that many web applications rely on storage systems that may be of some complexity [61], the connection is limited. Web applications are often constructed in such a way that they contain no state, and can therefore be trivially scaled, pushing the real concerns of the distributed systems field to the data layer, which relies on commodity systems, leaving the properties of the web application as a whole uninvestigated. Web applications are inherently distributed systems, as they rely on the client-server model of communication [23], and the connection between distributed systems research and web engineering and the connection between database research and web engineering could be made more explicit, to the benefit of both fields of research.

This thesis provides several new perspectives on web applications and web architecture change frameworks, and a way of bridging distributed computing and database research with web engineering. The aim is to answer the following research questions, representing a variety of perspectives on web applications, through interviews with companies that have web-based products with a common starting point:

RQ1 How do layered web applications commonly change over the course of their existence?

RQ2 To what extent do existing architectural change classification frameworks

shed light on the reasons behind and effect of common architectural changes?

RQ3 What forms do web application architectures take, and what variation exists in this regard?

RQ4 Which architecture-centric practices are used in the development of web applications?

RQ5 Which types of data is persisted in web applications, and what triggers its generation?

More specifically, the contributions of the thesis are:

- A method for reconstructing architecture evolution paths based on interviews, as well as finding common narratives—a concept translated from narrative analysis in the field of sociology—in such paths.
- Data showing the breadth and variety of web application architectures.
- Data showing some common narratives of architecture evolution—orders of progression shared by multiple architectures.
- An evaluation of two change evaluation frameworks, showing that their perspective is not universally valuable.
- An argument connecting distributed systems research to web engineering, showing that strongly consistent web applications are infeasible to construct.
- A taxonomy connecting database research on materialized views and view maintenance to web engineering, allowing for methods used in database research to be applied to web applications, and hinting at a larger theoretical framework for web applications.
- An evaluation of the taxonomy (previous bullet) indicating some of the current breadth of data generation triggers and persistence locations in web applications.

The rest of the thesis is structured as follows. In chapter 2 we present the research related to the thesis, establishing the background knowledge necessary for the reading of the thesis. Chapter 3 introduces the study design for a study of six companies located in Denmark, focusing on architecture evolution of their web applications. Chapter 4 evaluates the results of the study with a focus on the breadth of web application architectures and architectural patterns.

Chapter 5 likewise evaluates the results of the study, this time with an emphasis on architectural evolution. We evaluate two frameworks for architectural change for their usefulness in revealing patterns of evolution, and apply a novel method for constructing architecture evolution paths and analyzing them with an aim of finding common progression patterns.

Chapter 6 connects distributed systems and database research to web engineering, arguing that web applications are inherently weakly consistent, and presenting a taxonomy from understanding web applications that allows for the application of database research theory. The taxonomy is used to show a variety in approaches to data refinement and persistence in the companies interviewed for the study presented in chapter 3.

Finally, chapter 7 concludes the thesis, connecting the findings and pointing at potential future directions of web engineering research, most notably hinting at formalizations of web architectures through the exploration of *refinement functions* and *response generation strategies*—terms introduced in this thesis.

Chapter 2

Background

Summary This chapter introduces the research relevant to understanding the thesis. In section 2.1 we present relevant research on web applications, including a history, our working definition of a web application, a look at scalability as a property of web applications, and views on data in applications. In section 2.2 we look at approaches to characterizing changes in software architectures, and introduce architectural patterns that will be relevant in the thesis. Section 2.3 presents relevant research in distributed systems and databases. Finally, section 2.4 presents existing work in applying narrative analysis to the field of software engineering.

2.1 Web applications

The *World Wide Web* (or *W3*, as it has been referred to; or simply *the web* today) from which web applications get their name was introduced in 1992 as an information sharing system. The name is a metaphor for the world-spanning web of information that the technology would allow for. The goal with the web was to create a system in which information was easily accessible, usable by a broad audience, and to which it was also easy to add more information [11].

To support this goal, the web was designed as a client-server architecture—in which clients would request information from servers—and introduced some new technologies. The original paper introduced a hypertext language, HyperText Markup Language (HTML), a language based on the Standard Generalized Markup Langauge (SGML) standard; and it introduced a protocol for transferring hypertext documents between client and server, the HyperText Transfer Protocol, HTTP [10].

The protocol allowed for negotiation of content types (supporting other formats than just HTML), and supported both static (unchanging, and in the original case, presumably hand-written) and dynamic (generated by a pro-

gram) content, with the original paper recommending modifying the sample web server code they provided to function as a more dynamic interface to underlying databases. The sample server code would automatically serve content according to a directory structure, making it easy to share static content. Aggregation of information from other services was described already in the beginning [10].

There are several ways of sharing information on the web. *Web pages* were single pages, usually maintained by an individual and hosted on an organization's server, were popular in the early days of the web. *Web sites*—curated collections of pages—were initially only accessible to organizations with the required funds for running a server, but in time became more accessible. The term used for more interactive types of web sites, in which a focus on interaction and business logic was more pronounced, and the content was to a larger extent dynamic, was *web application* [61].

Web services are web servers that provide interfaces only for other programs to interact with, and no human-oriented interfaces [61]. As we noted above, program-to-program interaction over the internet was already an established practice at the beginning of the web, but web services increased the popularity of the approach. The defining characteristic of web services is the use of web technologies in communicating between applications, on web clients or other web servers. *Service-Oriented Architecture* (SOA) emphasized the usefulness of web services by splitting large applications into several services, all communicating among each other and with clients using a standardized protocol, that allowed for automatic discovery of services [57].

The term *web application* allows for the broadest interpretation of all the terms we have introduced, as it encompasses all systems in which a client and server interact using web technologies. Even in cases where a commodity web server is used to serve static files, the server and clients form a client-server application.

Around the year 2000 the most common approach for any kind of web application was using a standardized web server for serving static or dynamic content. There were various approaches to the latter, but one of the more common approaches was the Common Gateway Interface (CGI) [61].

When determining how to respond to a request, it was a common approach for servers to inspect the requested URL path. For example, when using CGI, the server might be configured to look for URL paths containing `/cgi-bin/` or ending in `.cgi`. In this case, the server would execute a program to which the HTTP request body, headers and other relevant information was passed as input. FastCGI provided a speed-up by not having to start a new program with each request, allowing instead for a single long-running program to respond to several requests, minimizing overhead. Still, the server and application code were usually kept as entirely separate programs. Commonly the program executed would be the one located at the file location described by the URL path, with each valid request path mapping to a file on the server machine. Perl was a common CGI-language, and approaches similar to CGI were used by PHP, Java Servlets, and many other programming environments [61].

Web servers, the servers in web applications, performing certain steps once

they receive a request. They (i) parse the request, (ii) decide on how to respond, (iii) fetch required data, (iv) generate a response, and (v) send the response to the client [61]. There may not be any data required for generating a response, in which case this step is trivial. Generation of a response is used loosely, and may be as simple as reading static content from a file.

Web browsers, the most common clients in web applications, are an execution platform onto themselves. Historically, both technologies such as Java Applets or Flash were used to provide interactivity and execution on the client. Today, the most common approach is Javascript, which is executed natively by the browser (whereas Java Applets and Flash required plugins) [52].

Fielding & Taylor define the architectural style used in "well-behaved" web applications as *Representational State Transfer* (REST). This architectural style describes a refinement of the client-server architectural style, where communication between client and server is stateless (e.g. each request and response is independently understandable without knowing previous requests or responses), with cacheability (such that the communication clearly states which values may be cached), and communicating over a uniform interface [23]. In web applications, this uniform interface would be HTTP. Finally, REST architectures support code-on-demand (CoD) where client functionality is downloaded on demand from the server [23]. In web applications this is achieved through servers responding with markup that is evaluated by browsers as an instruction to execute some code.

The notion of *well-behaved* web applications somewhat limits the generalizability of the REST pattern. It is possible to communicate using HTTP and providing a valuable user experience without adhering to the architectural style described. The architectural style has seen strong adoption in web services, where the transfer of state is the primary concern [60].

Many early web application servers were structured in such a way that a clearly defined presentation layer (often using templating languages) communicated with a business logic layer, which in turn communicated with a data layer [61]. We will refer to this as a *layered web application server*, as this agree with the description of the layered architectural style [9, p. 37]. Some web application frameworks[1] provided a different model for web application organization, in which routes were more explicitly defined, decoupling them from a directory structure [61]. This is the approach we now commonly see in modern Model-View-Controller (MVC) frameworks, such as Ruby on Rails [63].

The MVC pattern was originally used to describe a single input element that a user might interact with. Used in this context, the visual part is described by the View, the underlying data and business logic contained in the Model, and the

[1] Web application frameworks such as Ruby on Rails (and many others that use the label) might more accurately be referred to as web application *server* frameworks, as they provide ways of structuring the sever side of a web application, but rarely significantly impact or provide structure for the client side. This is in contrast to those frameworks that provide structure for the client side of a web application, which may more accurately be dubbed web application *client* frameworks. A web application framework would then encompass both of these concerns.

Controller is responsible for reacting to user input. This is also how the pattern was first attempted introduced to web applications, with an approach where components' models and controllers could be flexibly placed on the server or client depending on where it would be most efficient to perform business-related computations. In other words, the pattern was employed to make it easier to find a balance between a thick or thin client—that is, a client with much logic, or one with very little [47]. MVC frameworks such as Ruby on Rails use MVC as a code organization principle, in which the code in the application is split between models, views, and controllers, but where the granularity is typically an entire request at a time, rather than individual components [63]. The pattern used in MVC web frameworks bears little resemblance to the pattern as originally intended and used, other than in name. Significantly, modern MVC frameworks offer no way of easily moving execution responsibility between client and server. In fact, the MVC pattern in such frameworks refers only to the server-side organization of code, and has no impact on potential client-side execution.

More recent web server frameworks, such as Express (a Node.js framework), take the same approach as MVC frameworks of separating paths allowed on the web server from any underlying file system. Unlike MVC frameworks, however, they do not rely on separate server software in order to parse incoming requests [70]. In fact, several newer languages—such as Node.js and Rust—provide standard library support for constructing HTTP servers [76, 77].

Web browsers have (in accordance with Fielding & Taylor's definition of well-behaved web applications) built-in caching, caching results that the web server indicates are cacheable, in order to provide faster response times next time the same resource is requested [61]. Caching has also been utilized on the server side. Static resources can be trivially cached. In order to work with dynamic content, caches need a way to know when the data has gone stale and needs replacing. For rarely changing content, the common way of ensuring cache invalidation is setting a time-to-live on each document, so it is refreshed once in a while, and for more commonly changing content—or in cases where it is critical that responses are up to date—the web application will need to manually implement cache invalidation [42, 65].

Web 2.0 was a term popularly used to denote a new type of interaction and use of the internet, a move from publishing to interactive systems "with PC-equivalent interactivity," that became common around 2001. In Web 2.0, the network is considered the platform with execution "spanning all connected devices". Thicker clients became more common, as exemplified by Java Applets, Flash, and Javascript applications. Ajax (*Asynchronous Javascript and XML*) became a popular pattern used in code executed on web application clients, allowing for more dynamic user experiences [52].

Ajax fundamentally changed the model of interaction in browsers, moving away from the interaction model where each user action would result in a new document being displayed, and towards a model in which the web page became dynamic and changing, relying on asynchronous (no page reload needed) requests to the server for more data [34], "going beyond the page metaphor of Web 1.0 to deliver rich user experiences" [52]. This had previously only

Term	Meaning
Web page	A single HTML document, often manually maintained by an individual.
Web site	A collection of web pages, presented as a coherent experience, often maintained by an organization.
Web application	Traditionally used as a way to distinguish more complex applications on the web from simple, static web sites. Commonly used to describe all modern applications served via the web.
Web service	A web application server that provides a data-based interface targeted at other applications, rather than users.
Web server	The server part of the client-server architecture that makes up a web application. It receives requests from clients, interprets them, determines the appropriate response, and sends this to the client.
Web client	The client part of the client-server architecture that makes up a web application. It commonly takes the form of a web browser, reacting to user input. It sends requests to the server, and renders and makes interactive that which is sent as a response from the server.
Static content	Content that does not change based on user or other interaction. Most commonly used about the data in the server part of the web application, to denote pages or responses that are always the same.
Dynamic content	The opposite of static content, dynamic content changes based on user or other interaction. From the web server perspective, this is often content that is generated based on a particular request from a client (this is the definition of Iyengar & Challenger [42]); but may also relate to any content that is backed by a database, however it is generated (the definition used by O'Reilly [52]). (The two definitions overlap but are not equivalent.)
Thin client	A client that does very little in terms of execution based on the responses it gets from a web server.
Thick client	A client that functions as a significant execution platform, performing many actions.

Table 2.1: Terms used in relation to web applications

been seen done with plugins, but was not possible as a native experience in the browser. In this new world of distributed execution platforms, and with increasingly dynamic content, database operations became a core competency in web engineering [52].

Many of the ideas of Ajax have lived on in Single-Page Web Applications (SPAs). Technologies such as Flash or Java Applets could be (and were) used to construct Single Page Applications, but the approach was popularized relying on Javascript and Ajax. Much like Ajax, a Single Page Application denotes applications executed in the browser that are detached from the document model that the web originated with, requiring no page reloads to deliver data. SPAs take the approach of Ajax as far as possible, delivering most of the business logic of a web application, as well as all of the templates for all views that may be displayed in an initial response from the server. Once the SPA is loaded, the user's interaction with the web application will never require the reloading of a document in the browser, and the client's experience of the application is no longer affected by long load response generation times on the server, that would normally result in a blank screen during loading of each response. In SPAs, web servers are lean and focus on "services like persistent data storage, data validation, user authentication, and data synchronization" [50].

One downside of single page applications is a large initial load time, as the initial response is parsed and executed on the client. Static site generators use a different approach to avoiding response generation on the web application server, which may take a long time and result in a jagged user experience. Static site generators allow for complex sites of entirely static content to be generated based on data. They are often used for web applications where the primary content of the application rarely changes (e.g. a blog, in which posts are published at most a couple of times per day). Every time new content is release, the static files are rebuilt, resulting in the new state being served by the web server [59].

The JAMstack (Javascript, APIs, Markup) describes an approach to static sites in which the user experience is enhanced by using Javascript to communicate to exernal services [74]. For example, an external service might enable comments on the blog posts of the blog. The comment system will not be available with the first rendering in the browser, as this only renders the static blog post content, but execution of Javascript code will enable the comments system.

JAMstack avoids the large up-front loading of all potential views of an application that are necessary in SPAs, and avoids the majority of the work performed on a web server in order to display a dynamic (but rarely changing) view to a user, as this has now been performed at deployment time. Static files can be geographically distributed and with high availability using Content Distribution Networks (CDNs), resulting in a reduction of latency between clients and servers [74].

Progressive Web Apps (PWAs) are an approach to building web applications that allows for the core content to be preemptively cached on clients and made available even when the clients are offline. Further, they let web applications be perceived as more native on mobile devices, as browsers let users install PWAs

on their home screens [12]. The static content of static sites is no longer just distributed to a CDN, but all the way onto the client devices.

Curiously, many of the advancements in client interaction with servers we have presented above have been touted as performance improvements, despite moving in seemingly opposite directions: Ajax and SPAs were introduced to make the web more responsive, but resulted in larger initial loads; the JAMstack and static site generators were introduced because of the long up-front load times of SPAs—all in the name of performance improvements. All of the technologies have been described as bringing more desktop-like experiences to the web [12, 34, 50, 59]. Over time, it seems, a balance is being established, where the web can provide a dynamic user experience with minimal performance penalty over static content.

New features in browsers allow for clients that take on ever more responsibility, enabling them to act more like nodes in a distributed system, moving beyond the client-server pattern. Web sockets provide a means for more direct communication between a client and some other server, no longer relying on the browser's handling of HTTP requests [75], and there are examples of libraries allowing browsers to execute peer-to-peer file sharing protocols [78], or turning browsers into nodes in a distributed data store [72]. In other words, web application clients seem to be evolving beyond the notion of thick clients—as the possible space of their responsibilities change—and into something more akin to a node in a distributed system.

2.1.1 A definition of web applications

What is a web application? We have established a brief history of web applications, as well as an architectural definition of what a web application is (recall Fielding & Taylor). However, the perspective on what exactly constitutes the application, and what should be included when considering properties of the web application, has not been constant. By examining various perspectives (and keeping in mind the way the web has changed), we will attempt to argue for a present-time conception of what constitutes a web application.

With the very first version of web applications, the client was a static presentation layer, with no execution environment. As such, the only part of the web application that could be affected was the code executed by the web server [10]. Even after dynamic content became common, provided by servers that executed small programs in order to generate documents on-demand, clients were fairly static in their capabilities. It was uncommon to see more than simple enhancements to user experience done on the client side [61], despite business logic execution on the client having been conceived of and experimented with before the web was ten years old [47].

The perspective of what constituted an executing part of a web application started to change with Web 2.0, where clients became thicker and presented more interactive user experiences [52]. Today, web clients function as thick clients, executing much of the business logic involved in interacting with a web application [50], even moving beyond the client-server model of communication.

Conallen, in a 1999 paper on modeling web applications with UML, offers this loose definition of a web system [16]:

> a web system (web server, network, HTTP, browser) where user input (navigation and data input) affects the state of the business

Conallen notes that the definition "attempts to establish that a web application is a software system with business state, and that its 'front end' is in large part delivered via a web system". Unique to a web application (and not present in web sites), Conallen argues, is the business logic and state being manipulated by users [16].

Likewise in 1999, in a paper on object-oriented web development, Gellersen & Gaedke define a web application as "any software application that depends on the Web for its correct execution". They explicitly include in their definition "Web sites or Web-based journals", where emphasis is on content, as well as those things that Conallen would characterize as web applications, namely where business logic is executed [35]. Gellersen & Gaedke do not explicitly mention the client as an important part of a web application, but note that they consider anything that is delivered via the web to be a web application. As such, the client in the client-server system must be understood to be significant.

In the 2002 paper introducing REST, Fielding & Taylor describe a well-behaved web application as one in which "a network of Web pages forms a virtual state machine, allowing a user to progress through the application by selecting a link or submitting a short data-entry form, with each action resulting in a transition to the next state of the application by transferring a representation of that state to the user" [23]. We take web pages here to be understood in the broadest possible sense: responses from the web application, which allows for other clients than simply a browser. Fielding & Taylor go on to describe the types of components of a web application [23]:

> A *user agent* uses a client connector to initiate a request and becomes the ultimate recipient of the response. The most common example is a Web browser [...]
>
> An *origin server* uses a server connector to govern the namespace for a requested resource. It is the definitive source for representations of its resources and must be the ultimate recipient of any request that intends to modify the value of its resources. [...]
>
> Intermediary components act as both a client and a server in order to forward, with possible translation, requests and responses. A *proxy* component is an intermediary selected by a client to provide interface encapsulation of other services, data translation, performance enhancement, or security protection. A *gateway* (a.k.a., reverse proxy) component is an intermediary imposed by the network or origin server to provide an interface encapsulation of other services, for data translation, performance enhancement, or security enforcement. Note that the difference between a proxy and a gateway is that a client determines when it will use a proxy.

This list of components uses slightly different names than we have used thus far: we refer to user agents as clients, and origin servers as web servers. Other than that, the list conforms readily with the other definitions we have encountered, with the explicit addition of intermediary components.

In their 2003 book on web application architecture, Shklar & Rosen talk about web applications as "custom applications that operate within the context of a Web server environment, communicating with other Web applications, servers, and clients", noting that "[i]n the past, an 'application' was defined as a program [...] which executes on a single system," but this changes for client-server applications, where processing is distributed between server and client machines [61]. As a more formal definition, they offer the following [61]:

> A Web application is a client-server application that (generally) uses the Web browser as its client. Browsers send requests to servers, and the servers generate responses and return them to the browsers. They differ from older client-server applications because they make use of a common client program, namely the Web browser.

Shklar & Rosen go on to note that web applications are best thought of as multi-tier architectures, in which a client application communicates with a server application, which in turn communicates with a database or some other service [61]. Seeing web applications as multi-tier is consistent with Fielding & Taylor's inclusion of intermediary components.

Common for all the above definitions is that the client is an integral part of the web application. However, none of them discuss any limitations to the architecture of web servers (although we have illustrated some common approaches). We conclude that a web application server may be any arbitrarily complex system, as long as it performs the role as set out in the definitions above, responding to HTTP requests in the appropriate manner.

Our suggestion for a modern, and encompassing, definition of a *web application*—and the one we will be in the discussion of properties of such applications—takes the following form:

*A **web application** is a client-server application with any number of clients and servers, in which client-server communication happens via HTTP, in which both client and server may be execution environments, and in which the server may be any arbitrarily complex system in itself.*

This definition encompasses everything from static sites as well as arbitrarily complex architectures; from services that transfer only data, to web applications with a complex SPA executed on the client. It includes, also, web applications with several web servers, contacted for various purposes, and does not limit communication in the application, requiring only that some of it happens via HTTP. For example, as is the case with sites adhering to the JAMstack architecture [74], one web server might be responsible for the initial request, serving the code necessary to start executing on the client, and other servers could be responsible for, through APIs alone, providing services such as purchasing items from a shopping cart, leaving comments on an article, or similar.

2.1.2 Scalability in web applications

The scalability of a system, architecture or application describes how efficiency of a system changes as more parallelism is introduced [41]. We take a more abstract and requirements-focused view seeing scalability as the ability for an application to *respond to increased load without meaningful degradation of performance*.

Increased load can take the form of increased system interaction over time (e.g., how many clients connect to a server in a client-server architecture over time); the form of an increased amount of data being processed by the system; or any other form that leads to more processing in the system. Degradation of performance, in the definition above, can be defined from a user-centric perspective as a user's perceived decrease in responsiveness of a system.

The primary concern in ensuring the scalability of a web application is that the servers can keep up with the demands of the clients that connect. Client-side execution happens locally on each client, and only ever deals with the demands of a single user.

In web applications with simple layered server architectures, the commonly recommended approach is to make sure the servers do not contain state that is relevant for responding to requests in the business logic layer, pushing all state to the data layer. This is called a *Shared-Nothing* architecture and allows server applications to be replicated, hidden behind a load balancer, and scale to large amounts of requests, as long as the underlying data layer can ensure the required scalability [63].

2.1.3 Understanding data in web applications

There exist several perspectives on data in web applications. Marz & Warren present a highly scalable architecture, in which they conceive as all information in a system as ultimately derived from core data [49]. This view is similar to how event sourcing conceives of data, where events are the most basic data in a system, and more complex aggregates are derived from these events [25].

Core data's *rawness* is determined by how much other information can be derived from it. All derived state can be seen as computed by a function that takes all core data of the system as input. Marz & Warren use this perspective to propose a highly scalable architecture for applications that deal with large amounts of data, in which core data is persisted, and derived state is continuously rederived [49].

Event sourcing typically uses a more push-based approach, as events are published in streams, and aggregates are described by projection functions that perform changes based on the events they encounter. Each new event results in an update of the state the event affects [69].

Helland (2005) provides a perspective on the difference between data inside applications, and that outside or between applications. For inside data, transactions in data storage systems provide a way of seeing chronology, as each transaction is serialized. Outside data, however, is outside of the scope of trans-

actions, and temporally disconnected. Once it reaches its destination it is no longer known if it is outdated, so outside data is "always from the past" [40].

Helland characterizes *stable data* as data that is both immutable and uniquely identifiable, such that the data cannot change, and the context of the data cannot change in a way that would make the data interpreted in a different way. Based on this categorization of data, Helland argues that outside data is stable, such that a repeated request is unchanged, and a reading of it results in the same interpretation, whereas data inside applications is mutable and may change over time [40]. Events in event sourced systems can be seen as immutable data that describes outside data from the perspective of an application, and derived state as inside data, connecting the two perspectives on data.

Finally, Helland identifies different representations of data: querying languages such as SQL do not provide encapsulation, but allow for arbitrary queries; data inside applications, e.g. represented as objects, provide the opposite: encapsulation at the cost of limited queries. data formats such as XML or JSON would provide neither encapsulation or arbitrary queries, but function as an independent representation of the data, representing it in a self-describing (and stable) manner [40].[2]

2.2 Software architecture

Software architecture is a broad field, having been defined as dealing with recommending practices for software development; the structuring of software systems into components and their relationships; governance of system design and evolution; and working to ensure that nonfuctional requirements for a system are met. A concrete architecture can be seen as the meta-structure, how structures are structured within a system [38].

There are many potential perspectives to consider within software architecture. In this thesis, we will primarily be focusing on the perspectives centered around architecture evolution and change—that is, perspectives on how architectures change over time. This section introduces research in these fields, and several architectural patterns that are relevant to the work found later in the thesis.

2.2.1 Software architecture evolution and change

An *architectural style* describes a "recurring organizational patterns and idioms," and can be seen as describing a family of concrete architectures. Architectures may adhere to several (non-interfering) architectural styles at once, given that the constraints for each architectural style does not conflict with any of the others [33]. The REST architectural style for well-behaved web applications is an example. Architectural styles function as a vocabulary for communication

[2] Certain of Helland's observations are tightly coupled to technologies used in SOA. We have omitted these observations and focused instead on the general observations on data in applications.

about architectures, and enables comparisons, of both concrete architectures and of the architectural styles themselves [33].

There are several approaches to classifying changes in architectures, from simply distinguishing between improvements and corrections; to more granular classifications, distinguishing between performance enhancements and functionality-related enhancements, and introducing a category for preventative changes that aim to avoid future problems [62, 67, 73].

Williams & Carver (2010) introduced the SACCS framework, integrating several existing methods of classification, in which changes are classified by a wide range of properties. "SACCS was designed to capture the effects of changes to archi- tecture and provide a structured approach for impact analysis," considering factors such as a change's motivation (enhancing a system, or responding to a defect), the source of the change request, the importance or criticality of a change, the experience of developers making the change with the system they are changing, the change's effect on the architecture, the category of a change, and more [67].

The change's effect on an architecture can be either on the features offered by the system to users, how it affects the components in the architecture, or both. The category of a change uses the previously established distinction between corrective, perfective, preventative, and adaptive changes [67].

Chapin, Hale, Khan, Ramil, & Tan (2001) argue that models of architectural change suffer from several problems, most significantly that similarly named categories have been used with different meanings in different theoretical frameworks, and that they have been applied with little consistency in practical works. The classification of change types is highly dependent on how the classifier understands the motivations behind a change, and the same change may reasonably classified as several different types of change. Reliance on the motivation behind a change makes classification difficult, as motivations are difficult to accurately assess after the fact. Further, these classifications offer relatively little granularity, and, Chapin et al. argue, are a poor representation of practically occuring types of changes [13].

Chapin et al. present a new classification model for architecture maintenance—overlapping somewhat with Williams & Carver's consideration of changes' effect on the architecture, but providing a much more granular classification—in which classification is based on "objective evidence of maintainers' activities ascertainable from observation of activities and artifacts, and/or a before and after comparison of the software documentation". This model considers a larger set of architecture maintenance activities, including support work and changes to documentation, in addition to actual changes to the architecture. The correct category for a maintenance task is determined by looking at its impact, and determining the most impactful category it fits in, thus avoiding considerations of motivation altogether. Changes to business rules are seen as most impactful category (D), as a change that affects business rules likely will also result in changes to the structure of the application, updates to documentation, and new training tasks. Changes to source code that are not connected to changing business rules form the second most impactful category (C); documentation changes

without architectural changes the group below that (B); and finally supporting activities without changes to documentation are the least impactful (A). Within each category, a further level of granularity is offered [13]. Figure 2.1 illustrates the more granular categories for the two categories that concern architecture evolution (source code or business rule changes).

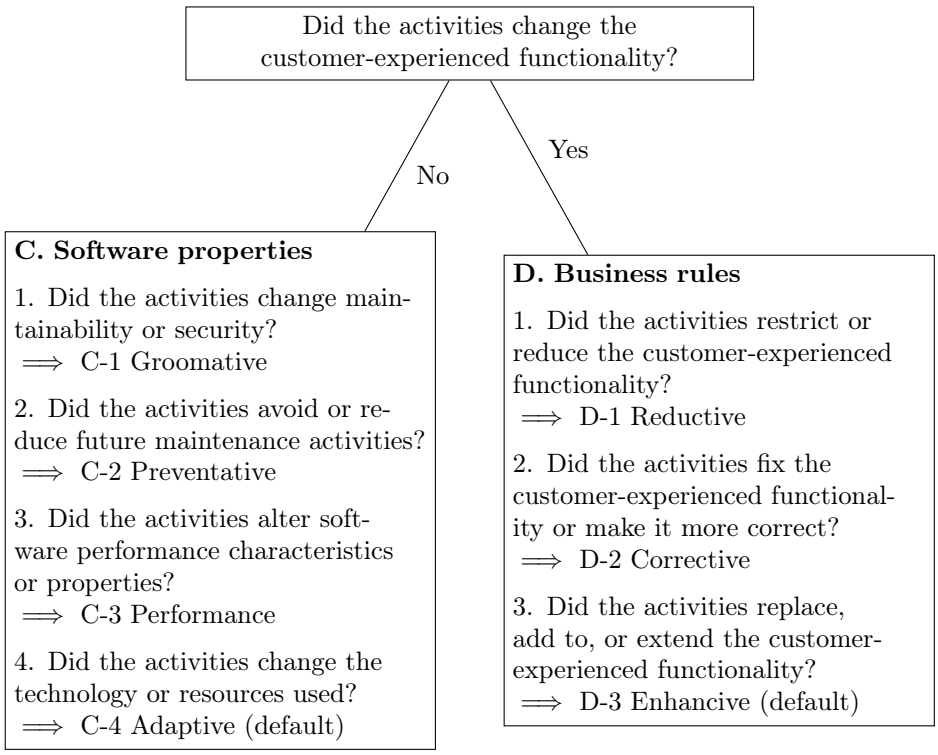


Figure 2.1: Decision tree for classifying architectural changes, adapted from the classification model by Chapin et al. [13].

Architecture evolution paths describe ways in which an architecture may change into another architecture, and provide an alternative perspective on architectural evolution. Rather than attempting to classify the changes based on impact or motivation, the changes are ordered chronologically [7].

Architecture evolution paths are usually used as a predictive tool, in order to investigate how to get from the current architecture (source) to an envisioned, future architecture (target). Evolution paths form a directed acyclic graph connecting source and target architectures, where each node is an intermediate architecture, and each edge is an operator, a change applied to an architecture, a step on the path from source to target architecture. A single operator may contain several *architecture transformations*, each changing the architecture in

some way [7].

We have shown that the study of architectural change has received much attention, but research has largely focused on planning, executing and evaluating such change. Frameworks aim to "assist developers in making decisions about how to address a change request" [67], or evaluate possible actions to decide which one is the better [7]. Case studies provide a view into concrete architectural changes, but are limited in their focus on details of particular cases rather than similarities in change across architectures.

2.2.2 Architectural patterns

As web applications may be arbitrarily complex systems, many architectural patterns may be in use. This subsection provides a brief introduction to those relevant for the work in this thesis.

Events in architecture

When some event occurs in one part of a system (e.g. user submits a comment to an article), it may be published it on a *message bus*, which other components in the system may subscribe to, in order to react when certain events occur (e.g. by notifying the author of the article). In event-based architectures, a message bus functions as a level of indirection for events, allowing any number of components to publish or subscribe to any event, in stark contrast to client-server architecture where each client knows which server they are communicating with [21].

Event sourcing is the approach of using events to determine an application's source of truth. That is, when events occur they are persisted in the system, and the state of the application is derived from these events. When new events occur, they too are persisted, and the appropriate changes are made to application state. The application state can be recreated at any point in time, by rederiving it from the events in the system [25]. This allows for some flexibility over traditional models of persistence that only store the current state of an application (e.g. allowing for a change in how events should be interpreted, and reinterpreting the events to create a new state consistent with the new interpretation).

Event sourcing can be seen as an instance of the Command Query Responsibility Segregation (CQRS) principle, in which the model for writes in an application is separate from the one that is read. That is, writes in the system may take the form of events, while reads may take the form of more complex domain models. This approach is in contrast to the CRUD model of data, in which records are created, read, updated and deleted directly. Commands are the actions that modify state, and queries are actions that read state [26].

Cloud applications, microservices, and serverless functions

Fox et al. (1997) recommended running larger applications on networks of smaller commodity PCs, in order to avoid forklift upgrades (in which a large server is replaced with an even larger server) and lower infrastructure costs [28]. Cloud computing relies on virtualization on top of large data centers full of low-cost commodity machines, making this type of infrastructure commonly accessible, and providing seemingly infinite computational power [6].

The term *Infrastructure as a Service* describes the on-demand virtually provisioned machines that public clouds make available: the infrastructure no longer needs to be physically installed, but can be requested from a service. A *Platform as a Service* is a higher level of abstraction, where a computational platform is made available, abstracting away some of the need for understanding the underlying infrastructure. The distinction between the two terms is not fuzzy, and there may be some overlap in how they are used [6].

Fox et al. (1997) further suggested the decomposition of large applications into smaller components with "well-circumscribed functional responsibilities", and in 1999 Fox & Brewer specified the recommendation, coining the term *orthogonal services*, with independent responsibilities and scalability considerations [28, 29]. This laid the ground work SOA, and for what is today known as *microservices*. The connection between cloud computing and microservices is significant, and explains why the two phenomena became popular simultaneously: smaller, virtual machines made it an advantage to compose systems of smaller, independent services, which microservices are an example of.

A modern definition of a microservice architecture is "a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API", where each service represents a single business capability, and includes "user-interface, persistant storage, and any external collaborations" it needs for this [27]. Using this definition, there are some cases in which architectures are close to being microservice architectures but have one or more deficits that keep them from being *true* microservices.

A *distributed monolith* occurs when a monolithic architecture is distributed across microservices, resulting in binary couplings between services. It can be defined as: a microservice architecture in which services have shared dependencies for common operations, leading to coupling between the services, and in which interaction with the architecture either requires or is made difficult by avoiding these shared dependencies.[3] Another type of "false" microservice is

[3] We know of no definition of distributed monoliths in academic sources. We rely, for our definition, on a 2016 talk titled *Don't Build a Distributed Monolith*, showing that the term is in use in the industry: "It doesn't take long until [sic] you have hundreds of libraries that are required, and the keyword there is required, to run the system. And if you can't actually launch a service and have it interact with your microservice architecture unless you have these hundreds of libraries, and these are the only hundreds of libraries that can work, then we are really losing a lot of the benefits of the microservice architecture. And this is a **distributed monolith**, because we have really just taken a monolithic code base and spread it out across the network" (emphasis added) [14].

one in which the feature set is too large to be considered the representation of a single business capability.

Load balancing provides a way of scaling both microservices and web applications, provided that they are Shared-Nothing. At the same time, replication makes the system fault-tolerant and highly available [28]. Another appraoch to replication would be creating a failsafe, such that a backup service takes over if the primary service fails. Geographic distribution (such as provided in many CDNs) brings the source of data closer to the users that need it, making latency, and thereby perceived response times, lower [56].

Serverless functions build on the principle of smaller business capabilities from microservices, but are "a new generation of platform-as-a-service offerings where the infrastructure provider takes responsibility for receiving client requests and responding to them, capacity planning, task scheduling and operational monitoring," and where developers provide just the code dealing with client requests [3]. Serverless functions abstract even virtual machines away, and can be seen as a type of more managed infrastructure than these.

Strangler applications

To introduce a strangler application is "to gradually create a new system around the edges of the old, letting it grow slowly over several years until the old system is strangled"; a gradual, lengthy replacement of one system component with another [24]. That is, in a strangler application, an old and a new application live side by side, with the new strangler slowly replacing parts of the old system, until it has replaced enough that the old system can be discarded entirely, and the new system takes over all the responsibilities of the old.

Domain-Driven Design

Domain-Driven Design (DDD) is an approach to software development that starts with considering the problem or business domain in which the software must work and deliver value [22, pp. 3-4]. Bounded contexts are a DDD term used to "mark the boundaries and relationships between different models", that is a model is useable within a single bounded context [22, pp. 333-334], and using a model across bounded contexts will involve some translation [22, pp. 337]. Bounded contexts are a virtual abstraction that can be used in software architectures to determine where interfaces need to exist between components, and how components should interact.

2.3 Distributed systems and database research

This section introduces relevant research in distributed systems and database research, focusing especially on the property of consistency within a system, and the concept of views in databases.

2.3.1 Consistency in distributed systems

Fox, Gribble, Chawathe, Brewer & Gauthier introduced in 1997 the concept of *orthogonal architectures* based on "commodity workstations" in order to eliminate "forklift upgrades" in which the only way of upgrading a server was to replace the entire thing [28]. Their approach, relying on many small machines in orchestration, rather than large machines, is identical in theory to how cloud cloud computing is achieved today.

In order to make their suggested system work, they identified "three fundamental requirements for scalable network services: incremental scalability and overflow growth provisioning, 24x7 availability through fault masking, and cost-effectiveness." That is, a highly scalable network service should scale linearly with the amount of hardware in use; be highly available; be cost-effective—or at least relatively cheaper than systems that would require forklift upgrades—to deploy; and making deliberate tradeoffs around consistency [28].

This led them to the concept of BASE semantics (basically available, soft state, eventual consistency) for distributed systems; an alternative to the established ACID semantics (atomicity, consistency, isolation durability). As they note, "it is preferable for an ACID service to be unavailable than to function in a way that relaxes the ACID constraints," which they do not find to be an adequate trade-off. BASE is introduced with the goal of being a more practically usable and less rigid set of semantics [28].

In strongly consistent systems (such as those adhering to ACID), once an update completes, any clients reading from the system will see the updated value, and not a stale one. Many systems have weaker forms of consistency, such as the eventual consistency in BASE, which can be seen as divergence from strong consistency [66].

Fox & Brewer (both also authors of the above paper) introduced the CAP theorem two years later, in a paper titles *Harvest, yield and scalable tolerant systems*. In the paper, they use harvest as a metaphor for the correctness of read requests to a system, and yield as a metaphor for the success of write requests. A highly consistent system which does "not tolerate harvest degredation because any deviation from the single well-defined correct behavior renders the result useless," would rather deny writes (low yield) than have incorrect reads, and thus prioritizes harvest [29]. In a distributed system, harvest concerns confirming writes, and yield concerns showing the correct state on all nodes.

The CAP theorem states that out of the three properties, *consistency*, *high availability*, and *partition resilience*, only two will ever be perfectly attainable in a single system. That is, it is impossible to have a system with strong consistency, high availability, and partition resilience [29].

A slightly more detailed model of consistency in distributed systems indicates that a distributed system can guarantee strong consistency if $W + R > N$, where W is the number of nodes a change must be written to to be considered successful, before it is accepted; R is the number of nodes a value must be read from in order to be considered a successful read (accepting the latest changed value as the correct one); and N is the number of total nodes in the distributed

system [66].

If $W + R > N$, then the written-to set of nodes and read-from set of nodes must necessarily overlap, which would guarantee an always consistent system, where writes are always propagated to all readers (assuming that readers can distinguish newer values from older ones). Such a system, with $W + R > N$ and a large N, can provide high availability and fault tolerance (through being distributed), as well as strong consistency, but is not partition-tolerant: if a partition happens, only the larger partition (with enough nodes to allow both valid reads and writes) will continue to be operational [66]. A partition-tolerant system must necessarily have $W + R < N$, for partitions to continue being operational given a partition.

Abadi (2012) adds latency as a property that is commonly (and should be) considered in tradeoffs with the other properties. He attempts to counter a common misunderstanding of the CAP as a very rigid tradeoff. "In reality, CAP only posits limitations in the face of certain types of failures, and does not constrain any system capabilities during normal operation." As network partitions are rare, he proposes a tradeoff framework that takes into account the common tradeoff between consistency and latency [1].

When replicating data, "there are only three alternatives for implementing data replication: the system sends data updates to all replicas at the same time, to an agreed-upon master node first, or to a single (arbitrary) node first." The first option can be considered as providing low latency but low consistency, whereas the others provide higher consistency within the system, but at the cost of higher latency for update propagation [1].

Abadi proposes a new theorem, PACELC, in which system designers must make the following tradeoff: given a partition, tradeoff between availability and consistency; else tradeoff between latency and consistency. That is, in the rare case of a partition, a decision between availability and consistency must be made (as noted in the CAP theorem); but even in the case of no partition, a valid tradeoff between latency and consistency may likewise be made. In other words, Abadi notes that latency should be considered an important parameter when making tradeoffs in distributed database systems [1].

While strong consistency may seem a desireable property, Fox & Brewer note that "[n]early all systems are probabilistic whether they realize it or not. In particular, any system that is 100% available under single faults is probabilistically available overall (since there is a non-zero probability of multiple failures)," and propose two possible strategies for coping with this reality: either trading off harvest for yield, ensuring high availability at the cost of consistency; or decomposing the system into smaller subsystems with independent storage, so the tradeoff decisions can be made differently for each component, taking into account the specific requirements for that subsystem [29].

Eventual consistency was in the CAP paper defined implicitly as simply the absence of strong consistency [29]. Others have more explicitly used this definition, e.g. Obasi & Asagba referring to systems that are highly available and partition-tolerant as necessarily at-most eventually consistent, and stating that eventual consistency is "a model for database consistency in which updates

to the database will propagate through the system so that all data copies will be consistent eventually" [51].

Both of these definitions are rather informal. To arrive at a more formal definition, we must decide on a perspective. As Vogels states, "[t]here are two ways of looking at consistency. One is from the developer/client point of view: how they observe data updates. The second way is from the server side: how updates flow through the system and what guarantees systems can give with respect to updates" [66], a perspective which has found some support [37].

Vogels defines the lack of strong consistency as *weak consistency*, a type of consistency in which write in a "system does not guarantee that subsequent accesses will return the updated value" and some conditions "need to be met before the value will be returned" [66]. The most recent value written to a system is called the *fresh* value, and once a value has been overwritten, it is considered *stale*. As such, systems that do not return the updated value, will instead return a stale value [37].

Eventual consistency, then, is described as a type of weak consistency, under which certain additional conditions apply. Specifically, "the storage system guarantees that if no new updates are made to the object, eventually all accesses will return the last updated value", and in which the time between an update and the returning of the correct value being returned can be determined based on "factors such as communication delays, the load on the system, and the number of replicas involved in the replication scheme", provided that no failures occur [66].

This, it should be noted, takes the user-perceived perspective on eventual consistency, defining it based on how the user experiences the consistency of a system. A system-oriented definition would define eventual consistency in terms of how long it takes for changes to propagate through the system, not how long it takes for a user to receive the correct value. Golab et al. provide such a definition, stating that eventually consistent systems are those where "[o]nce a write is acknowledged, the new value is propagated to the remaining replicas; thus, all replicas are eventually updated unless a failure occurs" [37].

We have now established weak consistency as a base level for consistency deviating from the property of strong consistency, and eventual consistency as a stronger definition than weak consistency. Within eventual consistency, there are several different types of consistency, each adding more consistency requirements to a system, that can be discussed. Vogels provides an overview of these stronger types of eventual consistency, of which several may be combined in a single system, so long as they are not contradictory [66]:

- Causal consistency, in which "events happen in order" [48], such that a single client will perceive events as happening in an ordered manner. If a client completes a write to a server, then a subsequent request for the same element will result in that same written value, and not a stale value. That is, from the perspective of a single client, chronological ordering of events appears preserved.[4]

[4]Lloyd et al. seem to view eventual consistency as a separate form of consistency, but their

- Read-your-writes consistency, in which a client will never see an older value than their latest write to the system. (This is a special, and more limited, case of causal consistency.) If client commonly communicate with the same server, this type of consistency is easy to provide; less so in systems where clients do not stick to the same server. It is possible to implement this on the client-side of the system, where the client ensures that if the system returns an older value than its latest write, it will simply use the value it wrote.

- Session consistency, in which a client accesses the system in a session, during which read-your-writes consistency applies. After the session terminates, the guarantee disappears.

- Monotonic read consistency, in which the system will never return an older value if a newer one has been seen. Like read-your-writes consistency, this can be implemented in the client.

- Monotonic write consistency, in which case the system guarantees that writes by the same client are applied in-order. Vogels notes that "[s]ystems that do not guarantee this level of consistency are notoriously hard to program" [66].

The *inconsistency window* is the time between a write of a value, and the time where the value has correctly propagated such that it is read correctly throughout the system [66]. Golab et al. refer to this same time, the time in which a stale value may be returned, as the *staleness* of a value [37]. In order to measure the size of said inconsistency window or staleness, one could employ the measure of Δ-atomicity, which measures the amount of time (in some time unit) a system is *at most* behind the most recent value written [37].

More precisely, Δ-atomicity and k-atomicity (where k is the number of versions behind the fresh value a system will at most be), are defined in terms of the golden standard of *linearizability*. In a fully linearizable system, "the storage system behaves as though it executes operations one at a time, in some serial order, despite actually executing some operations in parallel," and no reads of stale values can occur. Eventually consistent systems can then be described by their deviation from linearizability, and Δ- and k-atomicity are measures of this deviation [37].

Finally, staleness may also be measured as $< k, t >$-staleness, which describes the probability that a read t time-units after the latest write returns a value within some k-atomicity bound. This measure combines the two other measures, and provides it result in a probability, rather than a yes/no answer to whether or not a requirement is met, and may thus be seen as more flexible [37].

other observations do not otherwise contradict that causal consistency is simply eventual consistency with additional consistency requirements [48].

2.3.2 Materialized views

In a database, a *view* is a query over the data in the database, providing some new perspective on what is essentially the same data. A *materialized view* is a view for which the tuples resulting of such a query is saved in the database, and does not need to be constructed once the view is requested. A materialized view need not be fully materialized, but can be materialized on demand, and function like a cache for results from previous queries of the view [39, 45].

Much like with values in a distributed database, materialized views are considered stale if they are out-of-date, meaning that they have not yet incorporated all relevant changes from the base relations, the relations on which the view relies [45]. Materialized views add a layer on top of the freshness and staleness considerations we have previously discussed, as they aggregate those values. We now no longer have to just consider freshness of a single field we read and write, but also the freshness of aggregates that field may be a part of.

Instead of recomputing an entire materialized view when a field is updated (or when a query is made, if the materialized view is only computed on request), *incremental view maintenance* can be employed. Incremental view maintenance figures out how base relations affect the view, and which parts of the view need updating based on which base relations have changed. This allows the database to "compute only the changes in the view to update its materialization". For example, it might quickly be determined that an update to the dataset is an irrelevant update, and does not affect the view, leaving no reason for computation [39].

One approach to incremental view maintenance would be one considering the semantics of the queries that the views represent. If several overlapping queries are performed, but part of the query can be answered by a previously answered query where no results have changed, this will be faster than recomputing all these values. This requires figuring out overlap of queries, and which queries can be used as the basis for computation of which other ones, by looking at the semantics of the queries. This is called semantic caching [43].

Recomputing the materialized view as soon as a base relation changes is called *eager maintenance*, and is not always feasible in large databases with high load, or with many views, where blocking writes in order to ensure consistency until the changes has fully propagated would leave the database unresponsive. Two approaches are possible: accepting that the view may return stale data until the changes have time to propagate, or ensuring an updated view once the view is actually requested. Both of these would be termed *lazy maintenance* [45].

In the world of databases, there have been experiments with caching on database clients as well, in order to provide better performance. "Performance is improved, for example, by reducing interaction with servers and by avoiding the costs of obtaining data from remote sites." Additionally, locally caching data that can be used to answer certain queries will lessen the load on the remote database. If the database server keeps track of which clients have what data, it can notify the clients when data has become invalid, making clients fetch the

data again when they need it. This model is more efficient than pushing updated data, as the server does not know whether or not the data will be relevant to the client again [31].

2.4 Narrative analysis in software engineering

Barnes, Garlan & Schmerl present the concept of architecture evolution paths along with a formal constraint language for mapping such paths. In the constraint language, each step on a path is represented by an operator, which is a collection of *transformations* (structural changes to the architecture) and *preconditions* (the conditions that must be present before the operator can be applied to an architecture), as well as some other information to support analysis, marked *analysis* [7]. In the constraint language, each transformation is a "basic structural change to an architectural model [...] things like adding a component, deleting a port, renaming a connector, and modifying a property", but evolution operators are collections of these that are architecturally significant [7].

In the cleaning of our data (c.f. section 3.3), we found architectural steps that were generic and architecturally significant, at an abstraction level similar to operators in Barnes, Garlan & Schmerl's terminology, albeit less formally defined. Our architectural steps do not list all underlying transformations, and we have not formally categorized preconditions; but they are similar in spirit to operators, and similarly support the first-class analysis of architecture evolution paths.

In order to move slightly closer to the (very) generic operator definitions used by Barnes, Garlan & Schmerl (such as "Wrap a legacy component as a web service" [7]), we find similarities between steps, so e.g. steps that introduce at least one microservice are placed in a category of similar steps; as are steps that introduce a secondary web service with new functionality, to run alongside an existing web service.

The main analysis activity in this study concerns finding common paths of evolution among the architectures we investigated. For this, we take an approach inspired by *narrative analysis* from the field of sociology.

Narrative analysis "refers to a family of approaches to diverse kinds of texts, which have in common a storied form", and is a turn away from analysis looking for adherance to "master theories" [58]. Narrative analysis deals with various data collection and organization in order to produce something for further analysis; narratives are not raw data in and off themselves, but must be analyzed in order to be useful [58]. The texts under analysis are analyzed for "a perceived sequence of nonrandomly connected events" [64, p. 6]; a single narrative is a series of sequential and logically connected elements [32].

Narrative analysis has been defined as a method of "recapitulating past experience by matching a verbal sequence of clauses to the sequence of events which (it is inferred) actually occurred" [46]. The underlying actual sequence of events is sometimes referred to as the *story*, underlying the *narrative*, which

is the story as told. A story told is backed up by a series of actually occurring events, but the narrative is not limited to just the events occurring in the story.

There are various different definitions of the term *narrative*, which leads to different types of analysis [58]. For our purposes we will be relying on thematic analysis, in which emphasis is placed on what occurs, rather than how the narrative is told [58]. That is, we have a strong emphasis on reconstruction of the underlying story.

Reconstructing the underlying stories of narratives enables further analysis. Many cases in similar fields follow similar narratives, one example being *developmental narratives*, the stories of how things came to be [2, p. 68]. Analyzing several told narratives for commonalities can lead to the development of a *stage theory*, a theory of how one stage commonly follows another in "a common sequence of unique events". In these theories, some deviations are allowed, but they describe a pattern common enough that it is worth noting [2, pp.73-74]. Common narratives do not describe situations where an event X necessarily predicts another event Y, but rather situations where an event Y is commonly preceded by an event X. They deal with how an event Y commonly comes about, rather than questioning why it had to occur.

Narrative analysis has previously been used in the field of software engineering, although mostly in the "softer" parts of the field. From the 1990s and onwards, the natural science methods commonly used to tackle all problems faced by software engineering have been increasingly challenged, and more cross-disciplinary approaches have been taken. It is in this context that narrative analysis has been introduced to software engineering [4].

Narrative analysis has been used to interpret various facets of different types of failures of information systems, including users' experience of system failures [19], stakeholders' narratives of how failures came about [18], conflicting narratives on a system's success from different groups of stakeholders [8], and failures in outsourcing of information systems [17]. It has been used to further understanding of organizational practices [20], to explore maintainability in systems integration [53], and to better understand requirements elicitation for system design [5].

Common for all of these applications is that they directly translate the methods of narrative analysis to a new context. The analysis itself remains unchanged, but now interacts with established methods from the fields of requirements engineering, software architecture, and others. Many of the approaches focus on the primary analysis of narrative analysis, understanding nuance of the narratives ("reading between the lines" [19]), and establishing an underlying story from disparate narratives. We know of no studies that attempt to establish common narratives.

The concept of narratives as a storytelling of sequences and the identification of common processes in chronologically structured data has not been applied to more formal data in software development. In fact, some researchers argue that "there has been great success in applying [...] natural science methods to the study of information systems development", apart from in the "softer" parts [4], providing reason for abstaining from applying narrative analysis to less "soft"

parts of software engineering research.

An alternative view would be that thinking in narratives, and especially common narratives, can be useful in places dealing with much more formal and structured data; that narrative analysis has a place when dealing with other texts than transcribed speech.

Chapter 3

Study design

Summary This chapter introduces the study design for a study of 6 companies located in Denmark, where an employee from each company was interviewed regarding the architecture and architecture evolution of the company's web based product. The section covers participant selection, interview procedure, and transcription and coding process. Finally, the section covers validity and reliability of the study for the parts relating to the study design itself. The results, related analysis and discussion of the study, and validity considerations for the analysis can be found in chapters 4, 5, and 6.

3.1 Participant selection

The first step in the process of participant selection was identifying eligible companies. Some companies (3) from the authors' networks were contacted. A large number of companies (17) were contacted based on authors' knowledge of their existence, and an inkling that their software architecture might be eligible for the study. Most of the companies contacted (20) were identified through the following process. *The Hub*[1] provides a list of "startups" in Denmark, categorized according to various criteria. We started from the beginning of the list, contacting companies that: (i) were categorized by *The Hub* as being in the final phase of a startup, *Growth and Expansion*; (ii) were not co-working spaces, venture capital firms, or agencies; (iii) had at least 4 employees; (iv) were not reveal to be a webshop hosted as software as a service (SaaS) by a quick inspection of the company website; (v) had an online presence indicating that the primary product of the company was web-based. These criteria were meant to isolate companies that had likely had to deal with scaling their web-based product, and had a software development team employed.

[1] `https://thehub.dk/` (retrieved 2018-08-11)

All the companies we contacted were sent a standard email inviting them to participate in the study, explaining the context of the study, the type of participant we were looking for (that is, a company that started their product as a layered web application), and the expectations for them as a participant (see appendix A). Companies were contacted in Danish if their website and communication was primarily in Danish, and English otherwise.

Out of the companies contacted, 7 responded in the affirmative, and one negative. One contacted company responded with interest, but did not respond to follow-up emails. One of the positive responses arrived only after interviews had concluded, and as such the company was not included in the study. The overall response rate was 22.5%, with a 100% positive response rate from companies in the authors' networks; a 0.0% positive response rate from companies contacted based on authors' knowledge; and a 15.0% positive response rate from companies contacted via *The Hub*'s list. Six companies were selected for the study, with one interviewee from each company, selected based on the companies' judgment of who might best answer questions regarding the company's software architecture.

In order to get an idea of the companies selected, they were looked up in the Danish government's Central Company Registry (CVR),[2] extracting information about the industry categorization of the company and the number of employees reported per the fourth quarter of 2017.[3] The companies had between 2-4 and 50-99 employees per the fourth quarter of 2017, and were primarily placed in the *Computer programming* (3) industry code, with some participants in *Other publishing of software* (1), *Consulting regarding information technology* (1), and *Other assistance services regarding financial intermediation* (1).

During the interviews, interviewees were asked about their role in the company, and the company itself. Most of the interviewees had the role of *CTO* (4), but we also interviewed one *Director of Product* (1), and one *Backend developer* (1). The interviewees had been with the company between 4 months and 10 years (mean 5.2 years, median 5 years). In relative figures, they had been with the company between 7% and 100% (mean 75%, median 85%) of the company's existence. The companies were founded between 2004 and 2013 (mean 2011, median 2013), launching their products between 2004 and 2015 (mean 2012, median 2013). The reported size of company development departments was between 5 and 70 people (mean 20, median 11); and the reported number of users was between 1500 and 500000 (mean 166500, median 48000). Five companies were located in the Greater Copenhagen Area in Denmark, and one was in another location in Denmark.

[2] `https://datacvr.virk.dk/data/` (retrieved 2018-08-11)

[3] Interviews were conducted in the second quarter of 2018, but unfortunately the most recent employee count available was for the fourth quarter of 2017.

3.2 Interview procedure

Before the interviews were conducted, an interview protocol was created. The interviews were semi-structured interviews, with a list of subjects to cover, but with the interviewee taking the lead on interpretation of the subjects and the interviewer asking clarifying questions. Interviewees were also encouraged to illustrate as they were explaining. The interviews were recorded on two devices simultaneously, to provide redundancy, and the interviewees' drawings were captured by a camera.

At the beginning of the interview, the stage was set, and the participants gave active consent to having their voice and drawings recorded, and to the use of the results in the research project, provided the presentation of the data was anonymized. The topics covered during the interview were, in prioritized order (leaving the potential to skip later topics, in case an interview ran out of time):

- The product's profile (when development started, date of first launch, the team then, the team now, number of users, interaction pattern of users)
- The interviewee's profile (their role in the company, their role while working on the product, their interaction with software architecture as a tool, how software architecture is used in the company)
- The product today (the significant components of architecture)
- The initial architecture of the product (the considerations when starting to develop the product, the significant components)
- Changes along the way (gradual changes, major changes, the causes of said changes, the effects of the changes to the architecture, and potential evaluations of the changes)
- The interviewee's approach to scaling applications today (how would they go about responding to changing scalability requirements in their product as it first looked, if they were to do it today)

The interviews were estimated to take around an hour, and took between 36 and 99 minutes (mean 65, median 62). The full interview protocol can be found in appendix B.

After the interviews were completed, they were transcribed. The transcription was loose, emphasizing meaning over form, leaving out sounds such as stammering, coughing and sighing, but precisely transcribing sentence structure, to avoid interpretation during transcription. Only a single source of sound was used for transcription as none of them had failed during the interviews.

3.3 Coding

Coding is the first step in moving from the raw data of transcripts to more analyzable data. We first developed a preliminary coding table while transcribing

the interviews, noting interesting topics that were mentioned by the interviewees. When we started coding of the transcripts, we continued adapting the coding table, adding new items of interest.

There were at least 41 days between an interview being transcribed and the coding of the interview, leaving time for the researcher to distance themself from the experience of the interview, hopefully allowing for a more objective coding.

At the same time as the coding process we employed narrative analysis to reconstruct the stories behind the narratives told to us by the interviewees. In particular, we focused on reconstructing the order of the events that occurred when the interviewees talked about the items in our coding table. This process led to a construction of a timeline of events, each event representing a step in the architecture evolution of that company, which we noted in a separate document. We also constructed a company profile from the interview, noting significant years, as well as user and employee numbers.

This process resulted in a coding summary for each company, listing information about the company and the interviewee, a timeline of events in the company, a timeline of concrete architectural changes over time, and notes on architecture use in the organization. Certain architectural changes were coded, but could not be placed in time, but were mentioned in the interviews, and were thus listed after the timeline of architectural changes.

Figure 3.1 shows an example of an interview summary, with a company profile on the left and a timeline of the software architecture, with relevant codings, on the right. Identifiable information has been blacked out.

3.4 Validity and reliability

When considering the validity and reliability of the study, we rely on Krippendorff's categorization of types of validity and reliability [44].

3.4.1 Reliability

First, we consider reliability, in which we will look at the notion of stability (whether repeated applications of the same method would lead to the same result), and the slightly stronger notion of reproducibility (whether other researchers, applying the same method, would find the same results).

The most significant flaw in the reliability of the study is that the interviewing, transcription, coding and analysis involved was performed by a single researcher. This is likely to lessen reproducibility, or at very least indicates no positive reproducibility in the study. A common technique to avoid this problem is having several researchers perform the same research step based on the same (part of the) data, and comparing he results. If too much of a variation is found between researchers, reproducibility is considered poor. We have performed no such check.

Additionally, much of the approach presented in the study is novel and experimental, which may lower the stability of the method applied, as parts of the

Company	Product
[redacted]	
Ansatte 4.kv 2017: [redacted] **Branchekode**: Computerprogrammering **2012**: **2013**: **2014**: **????**: **2016**: **2018**: **Brugervækst**: lineær. **Interaktion**: Jævn trafik. Især nogle moduler har hele tiden fast load. Peak for alle brugere et par gange om året (omkring moms-afregning), især generel trafik. Hver dag ligger der næsten ingen load før 6 og efter 23, og det stilner af omkring 18, før en sidste peak kl 22. Interviewee info CTO. Ansvar for "product". Overholdelse af mål omkring automatisering, pipeline, tests. With the company sinxce [redacted]. Architecture in the organization. **T.1 AG.2**	1. Tidligere projekt. **AP.11** 2. Investor foreslår rewrite, C#/.NET kodebase Byggede stor MVC applikation, monolith. Fysisk maskine med db og app. **ACM.8 ACR.7 AP.11** 3. Db på separat boks. **ACM.1.d ACR.7** 4. Tilføjet nye services, parallelle layered apps. **ACM.2 ACR.2 AP.6 AP.8** 5. 2015. Azure. **ACR.3 ACM.6 AP.5 ACM.7.a** 6. Introduce jobs. **ACM.2 ACR.5** 7. Azure service bus. **ACM.9.b ACR.7** [This arch was stable for a while.] 8. 2017. Rabbit MQ to replace Azure message bus. **ACM.5 ACR.9 AP.2** 9. Docker + microservices. **AP.4.a AP.5 ACR.5 AP.9 ACR.8 ACM.6 ACM.7.a AP.2** 10. Tilføj flere microservices. **ACM.2 ACR.2 AP.4.d** 11. Fremtid: PaaS der autoskalerer og provisionerer maskiner til microservices. **ACR.5 ACM.6** Kunne ikke placeres i tid: - Domain-Driven-Design/ Introduktion af bounded contexts. **AP.1** - Læselister, build-on-change, CQRS. **AP.14**

Figure 3.1: Example of an interviewee summary.

method was refined during the coding and analysis of the results found during the interviews.

With these two major objections in mind, the steps of the project can individually be defended as relatively stable for the following reasons:

Participant selection Half of the participants for the study were selected using a formalized method. The other half relied on existing contacts, but received the same information as the formally selected participants. No interviewees were direct acquaintances to the researchers.

Interviews The interviews followed a pre-defined protocol, and allowed researchers only to ensure that all topics were covered, as well as asking exploratory questions, when interviewees responded.

Transcription and coding Multiple recordings existed for each interview, and transcription was verbatim (but not phonetic). Coding was performed using a legend of terms to code for. The architecture evolution paths constructed during coding were only included when directly supported by codings. (As a counter-point, the coding legend was refined as the interviews were transcribed, and different researchers may have found different points of interest in the transcripts.) At least 41 days were left between the transcription and coding of any individual interview. This is a common practice in order to have the transcript appear more as new to the researcher, lessening the impact of impressions the researcher may have had during the interview, and leading the researcher to rely more truly on the transcript itself.

The weakest links in the reliability of the study can be summarized as the lack of an established legend to code for, and a lack of a subjective means for generalizing architecture evolution steps. Both of these problems could have been lessened by having another researcher perform reproducibility checks on the data.

3.4.2 Validity

Validity concerns the design of the study and the use of the methods, and whether or not the methods do indeed perform the action they purport to perform. In this section we will only focus on validity concerns arising directly from the study design. Most of the validity concerns of the study relate to the application of method, and these are covered in the following chapters, where analysis and results are presented.

The only major validity concern based on the study design is the generalizability of the results. The sample size of the study (6) is very limited, and unlikely to provide any generalizable results. Further, the participants all resided in Denmark, which makes it likely that national trends have been captured and presented as general. Finally, the study focusing on a limited set of architectures (those evolved from layered web applications) means that the study does not properly represent the breadth of architectures found in the industry.

The generalizability of the study is therefore severely limited, and the results can generally be considered to show that certain things exist and occur in the industry; but not that there are no more common occurrences than them. It should be noted, however, that the study's goal was never generalizability, but exploration.

Chapter 4

Breadth of web application architectures

Summary This chapter presents a set of results from the study described in chapter 3, and the ensuing discussion attempting to answer research questions RQ3 and RQ4, regarding architectural patterns of web applications and architecture-centric practices in the interviewed companies. This includes findings of several known patterns, and a few previously undocumented patterns of web application organization; as well as finding that most of the interviewed companies had long-lived hybrid architectures, somewhere between architectural patterns that are usually considered to be irreconcilable. All in all, this illustrates the existence of a wide variety of web application architectures. Finally, the chapter presents reliability and validity concerns related to the analysis performed in order to answer these questions.

4.1 Concerns of this chapter

Given the study presented in chapter 3, we can attempt to answer some of the posited research questions. This chapter will present the results and analysis of the data relevant to research questions RQ3 and RQ4, as well as the ensuing discussion.

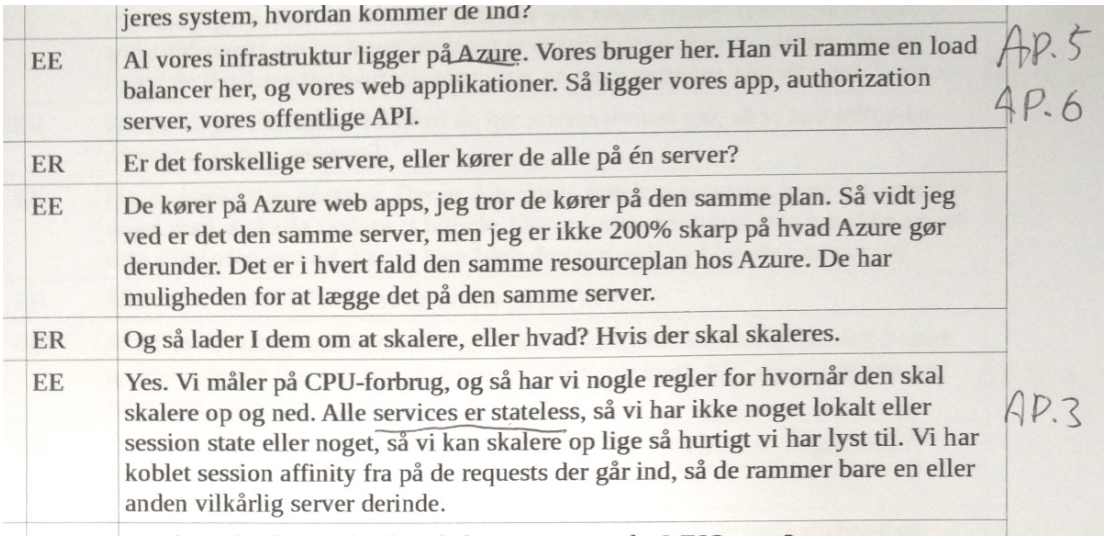
RQ3 What forms do web application architectures take, and what variation exists in this regard?

RQ4 Which architecture-centric practices are used in the development of web applications?

The coding table relevant for this chapter can be found in appendix C.1. The coding concerned itself with noting architectural practices observed (cate-

gorized under the headings *Architecture as a tool in the organization* and *Goals and direction for architecture*); architectural patterns; reasons given for architectural change; and the manner of change (categorized by the type of operation performed on the architecture, such as adding a new component, or removing an existing one).

An example of coding of a section of interview (conducted in Danish) is shown in figure 4.1. The excerpt shows the mention of infrastructure hosting on Azure being coded as AP.5 (Infrastructure or platform as a service); the mention of several web applications accessible by a client coded as AP.6 (Parallel layered web applications); and the mention of all services in the architecture being stateless coded as AP.3 (Shared-Nothing architecture).



	jeres system, hvordan kommer de ind?	
EE	Al vores infrastruktur ligger på Azure. Vores bruger her. Han vil ramme en load balancer her, og vores web applikationer. Så ligger vores app, authorization server, vores offentlige API.	AP.5 AP.6
ER	Er det forskellige servere, eller kører de alle på én server?	
EE	De kører på Azure web apps, jeg tror de kører på den samme plan. Så vidt jeg ved er det den samme server, men jeg er ikke 200% skarp på hvad Azure gør derunder. Det er i hvert fald den samme resourceplan hos Azure. De har muligheden for at lægge det på den samme server.	
ER	Og så lader I dem om at skalere, eller hvad? Hvis der skal skaleres.	
EE	Yes. Vi måler på CPU-forbrug, og så har vi nogle regler for hvornår den skal skalere op og ned. Alle services er stateless, så vi har ikke noget lokalt eller session state eller noget, så vi kan skalere op lige så hurtigt vi har lyst til. Vi har koblet session affinity fra på de requests der går ind, så de rammer bare en eller anden vilkårlig server derinde.	AP.3

Figure 4.1: Example of coding of a section of an interview

4.2 Relevant results

In this section, we present results from the study relevant to answering the research questions posed at the start of the chapter.

Out of the six companies we interviewed, none used formal architecture tools as part of their development process, and none were guided by concrete target architectures. We found that all (6) of the participants had an informal approach to software architecture, with half (3) never using architecture diagrams to discuss architecture or architectural direction, one participant using diagrams solely for external communication, and the remainder (2) participants regularly using diagrams (albeit informally) as part of their architectural design process.

Most of the participants (4) lead their architectural development by architectural principles, one participant had no way of setting architectural direction, and one participant made no indication during the interviews as to how they dealt with architectural direction and goals.

We counted occurrences of architectural patterns in two ways: the number of cases in which an architectural pattern occurred in at any point in time; and the number of current architectures (at the time of interview) an architectural pattern occurred in. The overall occurrence count can be seen in table 4.1.

Architectural pattern	**Cases it occurred in**	**Current architectures it is present in**
Web server application	6	6
• Shared-Nothing, as part of a larger system	5	5
• Shared-Nothing, as only part of the system	3	1
• Stateful, as part of a larger system	1	0
• Stateful, as only part of the system	1	0
Chained web servers	2	2
Parallel web servers	4	4
Replication of components	3	3
• Load balancing	3	3
• Geographic distribution	2	1
• Failsafe	2	0
Bounded contexts	3	3
Strangler pattern	4	2
Microservices	5	5
• True microservices	5	5
• False microservices: large feature set	1	0
• False microservices: not individual storage	2	2
• Machine learning as microservice	3	3
Services in multiple programming languages	3	3
Serverless functions	2	2
Infrastructure or platform as a service	5	5
Shared dependencies across services	3	3
Shared database across services	3	3
Communication via message bus	4	3
CQRS	2	1
Static site/JAMstack	1	1

Table 4.1: Occurrences of architectural patterns

All of the participants (6) started their products with a least one web server application in their architecture, and most (4) had a single web application server with no replication as their entire architecture at the launch of their product. One participant started their architecture in a similar way, but with replication, and the last participant's product's first version had a more complex

architecture.

Throughout the existence of their applications, some participants (2) had stateful (i.e. not Shared-Nothing) web applications, but in current architectures, all of the web applications are Shared-Nothing and therefore scalable by industry recommendations. Half of the cases (3) had done so, by replicating their web applications behind a load balancer. Only a single current architecture remained a single layered web application—all the other architectures evolved beyond this simple pattern, choosing other ways of growth than simply replicating a single Shared-Nothing web application.

In terms of replication, the most common form was load balancing (3), but we also found geographical distribution (2) and replication as failsafes (2), providing backup in case the main system went down. All of the failsafe systems were, however, removed again before the current versions of the architectures; and one of the geographical distributions logged was only planned for the future, and not actually completed yet, however load balancing serve much the same function with the added benefit of lowering load on individual components, and some gave up failsafe replication for this pattern.

We identified two interesting patterns of composition for web servers:

- **Parallel web servers**, in which two parallel web servers use the same data layer, and provide entry points for different users, or differing functionality (see figure 4.2).
- **Chained web servers**, in which one web server uses another as its data layer, providing a further level of abstraction on top (see figure 4.3).

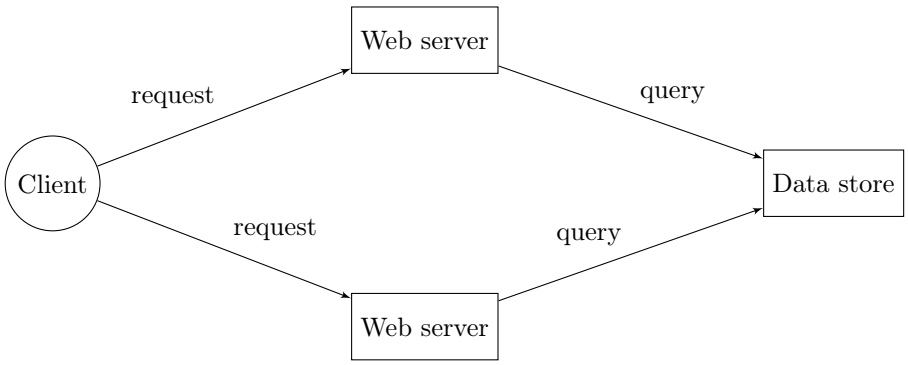


Figure 4.2: The *parallel web servers* architectural pattern

Web applications have long been seen as multi-tier applications [61], and chained web servers may be seen as simply an instance of this pattern. Alternatively, the patterns can be seen as special cases of Service-Oriented Architecture

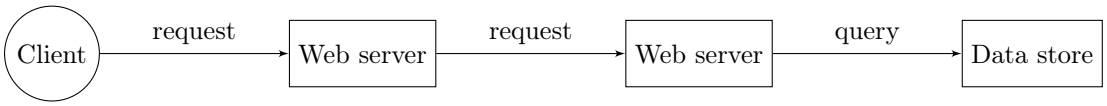


Figure 4.3: The *chained web servers* architectural pattern

(SOA) [57], where web services use standardized protocols to communicate between each other.

The patterns differ from SOA in various aspects, but most significantly that there is no standard way of connecting and communicating between components employed in the architecture; rather, in applications with chained or parallel web applications we do not find a general tendency of web services intercommunicating. The communication is strongly directed (each layer a lower level than the previous one), and have more in common with web services as a general concept than with SOA.

While this type of communication, between web servers, is common, what we observed was a common and novel manner of organization, that is not precisely captured by the labels of multi-tier applications or SOA. Parallel and chained web applications are terms that more precisely communicate the manner in which the components communicate with one another.

We have seen similar patterns in microservices, with communication via HTTP, so it is important to note that neither of these web servers are microservices, either as claimed by interviewees or according to our working definition of microservices.

We found parallel web servers to be a common pattern (4 participants), and chained web servers common enough to be of note (2 participants). Parallel web servers were often started as a way to provide a new feature set with a different type of interface, such as introducing a separate web application to serve as an API service for third-parties to integrate with the product. Chained web servers often took the form of a more specialized outer web application that customers would interact with, and a web application with a broader feature set hidden from the customers. We also saw a combination of the two, where two parallel web servers both used another web server as their data layer.

We found that half of the participants (3) at some point introduced the concept of bounded contexts from domain-driven design, and that this was a significant guiding principle in their architectural design.

Most of the cases (4) at one point introduced a strangler application. Interestingly, while a strangler is ongoing, the architecture is in a sort of hybrid state, with both an old and a new component existing at the same time. We found two cases in which the strangler was completed (and so only two current architectures have an ongoing strangler in them). One took less than 2 years to complete, the other 4 years. The two still ongoing stranglers had lived for 1 and 4 years, respectively, at the time of the interviews.

At the time of interviews, most of the applications were fairly complex, consisting of many different components, including microservices (5), message-based communication (3) or an intention of getting it (4), or different services written in different programming languages (3). Some (2) had introduced serverless functions, a highly managed type of infrastructure, that has been known to significantly lower cost of execution [3]. Using platforms that provide infrastructure as a service is a very common (5) approach among the interviewees, with only a single participant using a more traditional hosting partner, their application tied to a single physical machine.

In trying to determine what was meant by different interviewees when they mentioned microservices, we inquired as to how exactly these services interacted with storage and other components of their architecture. In one case we found (micro)services with a feature set spanning across several business capabilities, and two cases had (micro)services that shared storage between several services. In current architectures, however, the feature sets of the false microservices were made to conform to business capabilities: the existence of a larger, less-conforming microservice was a step on the way to learning the proper way of using microservices in their architecture. One of the remaining cases of microservices with shared storage was justified by the interviewee by noting that the shared storage was inside of a bounded context, showing a different definition of a conforming microservice (i.e. a single business capability, and not shared storage *across bounded contexts*).

None of the interviewed companies had full microservice architectures, architectures made up entirely of small services. All companies with microservices in their architecture (5) used them in coordination with more traditional (and larger feature-set) layered web applications, in a sort of hybrid architecture. An interesting pattern that emerged is that half of the cases (3) experimented with machine learning by adding it as a microservice, separate from the main capabilities of their product, illustrating that microservices were used as a way of experimenting without impacting the existing capabilities. There was no immediate pattern to when companies started introducing microservices in their architectures, the first introduction spanning from 2012 to 2018, with a mean introduction time in 2017.

Expanding the definition of a distributed monolith to simply include shared dependencies across different services (not just within the microservices of the architectures), we found that half (3) of the architectures under study were guilty of this proclaimed anti-pattern. Another indicator of high coupling in the architectures was the shared database access across different services, of which half (3) of the architectures were also guilty. Several interviewees indicated a desire to improve this situation, but that it was accepted as a current state of affairs, as other improvements and additions to the application were prioritized higher. This indicates that what can be presented as an anti-pattern in the industry may be a common pattern in concrete architectures, as it works well enough for the current needs of the architecture.

One interviewee reported implementing some form of Command-Query Responsibility Segregation; another interviewee indicated a desire to implement

something like this in the future. Finally, one interviewee reported having a static website enabled by Javascript and API calls executed in the client, an approach resembling the JAMstack pattern.

4.3 Discussion

During the study we found no of formal use of architecture-centric practices and tools, with a majority of the interviewees steering by way of principles rather than target architectures, and only a few interviewees using any kind of architectural diagrams in their design process.

This lack of formal architectural use supports prior research. For example, Paixao et al., studying developer awareness of architectural impact in an open source project in 2017, found that developers discussed the architectural impact of their changes 38% of time, "suggesting a lack of awareness"; and that architecture is a focus point when it is being improved, rather than when architectural degredations are introduced [55]. Likewise, Ozkaya, Wallin & Axelsson in 2010 found empirical support that "systematical use of architecture-centric practices do not serve as a first class resource" during system evolution in the large scale projects they studied [54].

While the use of architecture-centric tools in the cases we studied was not systematical, architectural thinking *was present*, but managed in a pragmatic way, often using common principles and practices across a team to slowly work towards a (vaguely defined) target. From these results we can expect that participants have an understanding of their architectures, and at least some of them are experienced in illustrating it using diagrams. This gives us some sense of security in the accuracy of the illustrations the interviewees provided.

We found several unorthodox architectures in the cases we studied, including patterns such as parallel and chained web applications, and architectures in a hybrid state between layered web applications and microservices. Infrastructure as a service could be seen as both a necessity and an enabler of the complexity we identified. We found it a fairly common occurrence to have long-lived slow changes as a part of the architecture, and significant complexity in most of the architectures studied.

Long-lived hybrid architectures (in which several architectural patterns that are usually considered in opposition—such as monlithic web applications and microservices—are in use at the same time) were common. This may be taken as an indication that prescriptive architectural tools that focus on efficiently moving from one architecture to another (such as is the declared purpose of architecture evolution paths [7]) may be aiming to perform something that is incompatible with what software developers would naturally do. That is, there may not be value in a complete migration from one architecture to another, if the benefits of two opposing architectures can be achieved at the same time, and some architectural tools may make the choice of a hybrid less obvious, and hence lead architects in a direction that may not be optimal.

As all the interviewed companies has a web-based product as their primary

offering, this all goes to show a breadth and variety of concrete web application architectures in use in the industry.

4.4 Validity and reliability

4.4.1 Reliability

Most of the concerns regarding reliability were covered in section 3.4, and the results presented here are subject to those concerns. The analysis of results in presented in this section consisted of counting occurrences of codings, which is highly reliable (several researchers are likely to come to the same count). In other words, no further reliability concerns were introduced in the analysis.

4.4.2 Validity

We will consider the notions of face validity (does the method readily appear to describe what it claims to describe?); social validity (are the results useful or relevant beyond academic scope?); and content validity (a subset of empirical validity—are all facets of what we claim to study captured by the method?). We consider as a part of empirical validity whether or not the results of the study support existing research. Finally, we consider the external validity (or generalizability) of the study. As we posed several research questions, it will be fitting to consider the validity of each part.

Our findings on architecture-centric tools support prior research, and as such passes face validity. The findings likewise pass content validity, as the full research question was more or less directly asked to interviewees, and their responses were noted and coded for content. The results are not immediately socially valid, as they provide no solution to the problem that such tools are not in common use in industry, but rather provides further encouragement to academics to research and understand *why* the tools established in academia do not make it to the industry.

We identified several established patterns in use, which supports face validity of these findings. We did, however, find them in use in hybrid situations, which is less well supported in the literature. This finding has not been contradicted by literature, and should therefore not be considered to detract from the face validity of the findings.

The results capture existing practices in the industry, and may be considered socially valid in that they can be used by practitioners to determine the kinds of architectural approaches used by their peers.

The findings are by no means exhaustive, both due to the small sample size, and due to the imperfect nature of interviews, in our case manifesting as the interviewee controlling which parts of a system were covered in detail, and therefore which architectural patterns we were likely to discover. The findings can be considered to have content validity if the goal is seen as merely illustrating

the types of nuance that exist in web application architectures, but not if the goal is considered to be exhaustive.

4.5 Summary of findings

We documented a lack of formal architecture-centric tool usage in all the companies we investigated, finding instead that they would steer their architectures with principles, and rarely used diagrams when discussion architecture and architectural changes.

We found a breadth of architectures, many unorthodox and in hybrid states between architectures normally seen as opposing. In particular, we documented two new common architectural patterns, parallel and chained web applications.

Chapter 5

Web application architecture evolution

Summary In this chapter we rely, again, on the data gathered in the study described in chapter 3, this time focusing on answering research questions RQ1 and RQ2. The chapter presents the relevant results from the study, including some data cleaning and analysis, and an ensuing discussion.

In order to evaluate the completeness of two existing change classification frameworks, we reclassify the changes we identified in them, and find that one does not allow us to accurately classify changes, and the other does so only imprecisely. In order to evaluate the usefulness of the same frameworks in our context, we see which patterns they allow us to identify in the architecture evolution paths we constructed, and find that they do not provide any new insights.

We identify common narratives in the architecture evolution paths, and present them. We evaluate the method applied to finding common narratives, indicating future improvements. Finally, we evaluate the reliability and validity of the analysis presented in this chapter.

5.1 Concerns of this chapter

This chapter continues the presentation of results, analysis and discussion of the data gathered by the study presented in chapter 3. This chapter, specifically, focuses on answering research questions RQ1 and RQ2:

RQ1 How do layered web applications commonly change over the course of their existence?

RQ2 To what do existing architectural change classification frameworks shed light on the reasons behind and effect of common architectural changes?

The coding legend relied on in this chapter is the same as the one introduced in section 4.1. This chapter introduces a method for analysis of interview data on architecture evolution in order to establish common narratives. Architecture evolution paths are normally used to assist in deciding on how to get from a source to a target architecture [7], but we use it as a descriptive framework for classification of changes over time.

5.2 Relevant results

The coding of the interviews resulted in concrete timelines of architectural changes in each company's product. The time lines, however, contained much specific information, making it hard to find patterns by comparison of timelines. In order to support such a comparison we interviewed, we constructed more generic versions of the timelines of architectural changes. Each step taken in the architecture was lifted to a more abstract level, reducing the level of detail, and making the steps more akin to the *operators* found in Barnes, Garlan & Schmerl's framework of architecture evolution paths [7]. For example, details on programming languages, frameworks and specific cloud providers were changed to more generic descriptions, changing e.g. *ASP.NET MVC* to the more generic *layered web application*, and *Microservices hosted in Amazon AWS ECS* to *Cloud-hosted microservices*.

During the cleaning we also ensured to construct a fully tagged version of the architecture at each step of the evolution. In the original coding we had only recorded patterns in the steps where they were mentioned by the interviewee, but with this cleaning we ensured that each step of the architecture evolution path mentioned all the applicable patterns at that point in time. This made it possible to determine which types of applications were in existence over the course of the product lifetimes.

Figure 5.1 shows an example of two architectural steps that have been made generic, showing the explicitly described architecture changing, and the coding of change reasons (ACR), manner of change (ACM), and architectural patterns (AP). The full list of cleaned architecture evolution paths identified can be found in appendix D.

During the coding of the interviews we identified and roughly categorized reasons for architectural changes given by interviewees when inquired for these. The questions asked during the interview were left open, and the interviewees' responses were taken at face value.

The result of these questions are shown in figure 5.1, listing both how many of the interviewees (out of 6 cases) mentioned a reason for architectural change, and how many of the total architectural changes (out of 64 registered changes) were explained by the reason. For many architectural changes, multiple reasons were given. Notice, in particular, that reason 7, *Unknown*, lists the number

4. *Add new web application with new functionality*
   ACR.2 ACM.2
   Load-balanced parallel and chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site.
   AP.3.a AP.5 AP.6 AP.7 AP.9 AP.16.a AP.17

5. *Introducing serverless functions*
   ACR.2 ACM.2
   Load-balanced parallel and chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site. Serverless functions.
   AP.3.a AP.5 AP.6 AP.7 AP.9 AP.10 AP.16.a AP.17

Figure 5.1: Example of generic architectural changes.

of changes that no reason was indicated for, either because the interviewer did not inquire about the particular change, or because no reason was given despite inquiry.

Change reason (ACR)	**Cases it occurred in (out of 6)**	**Total occurrences (out of 64)**
1. Migrating when infrastructure automation is ready	1	1 (2%)
2. New feature requirement	6	17 (27%)
3. Poor performance under current conditions	5	18 (28%)
4. Need for future scalability	3	8 (13%)
5. Developer experience improvement	5	12 (19%)
6. Reduce infrastructure costs	2	2 (3%)
7. Unknown	5	13 (20%)
8. Industry popularity	4	5 (8%)
9. Eliminate error sources	4	5 (8%)
10. Escape vendor lock-in	1	2 (3%)
11. Adhere to bounding context	1	1 (2%)

Table 5.1: Reasons for architectural change identified in our study.

A single change was justified as waiting on infrastructure automation to reach a certain level, migrating the infrastructure as soon as a new offering was ready. This particular change related to serverless functions.

A lot of changes (27%) were related to feature requirements, usually improving or adding functionality. It is interesting to note that only feature changes that resulted in architectural changes are noted here.

The biggest category of architectural changes (28%) were the ones justified by poor performance under current conditions, but a large group (13%) were

also justified by a need for future scalability. There was some overlap between the two groups (changes justified by both reasons), but together they indicate thatit is common for architectural changes to happen for performance reasons.

Twelve changes (19%) were driven by a desire to improve developer experience. One case in particular stands out, where the move to microservices (away from a monolithic layered web application) was justified by microservices enabling more developers to work in parallel without disturbing each others' work.

A few changes (3%) were driven by a desire to reduce infrastructure cost; a few (8%) by industry popularity of a technology, or a sense of the solution being "the right way" to do something; a few (8%) by the desire to remove error sources; a few (3%) to escape vendor lock-in; and a single change was made in order to better adhere to the bounding contexts established in the architecture.

Across all 6 cases we identified a total of 64 distinct steps, spread across the six different architecture evolution paths, with lengths ranging from 4 to 15 attached steps per path. There were 7 steps that we were unable to place in order in any of the paths, so these were listed separately as detached steps.

The architecture evolution paths we constructed can be considered attempts at constructing the underlying stories from the narrative conveyed by the interviewees. From these stories we intend to find common narratives of architecture evolution.

First, we established which steps were common between the architecture evolution paths, noting which steps overlapped entirely, and which were similar (see a full table of similarities noted in appendix F), looking for steps that could reasonably be construed as applications of the same operator (in Barnes, Garlan & Schmerl's terminology), so e.g. any step that introduced a microservice to the architecture would be noted as similar to any other step that introduced a microservice to the architecture.

Figure 5.2 shows an overview of the paths, as well as the steps' relationship with each other. Each architecture evolution path is illustrated as steps (circular elements) connected with arrows, showing the progression. Detached steps (that could not be placed in time) are greyed out. Steps that are exactly equal are connected with a thick colored line; steps that are similar are connected with a dashed colored line. Redundant similarity lines have been eliminated, such that a step that is similar to another step may be connected via a step they are also both similar to. Each similarity cluster (a group of steps that are all similar to each other) are connected by lines in a single color, but some clusters have the same connection color.

As noted previously, our steps are not defined exactly as operators, as we did not have the required data for establishing concrete transformations and prerequisites for operators. We rely, instead, on the words of our interviewees, assuming that they mean the same thing when using the same words.

We consider three types of narratives when identifying common narratives in the architecture evolution paths we have constructed. We use $X \rightarrow Y$ to denote a loose sequence, in which some step X occurs before some step Y in an architecture evolution path, but not necessarily directly after each other. A

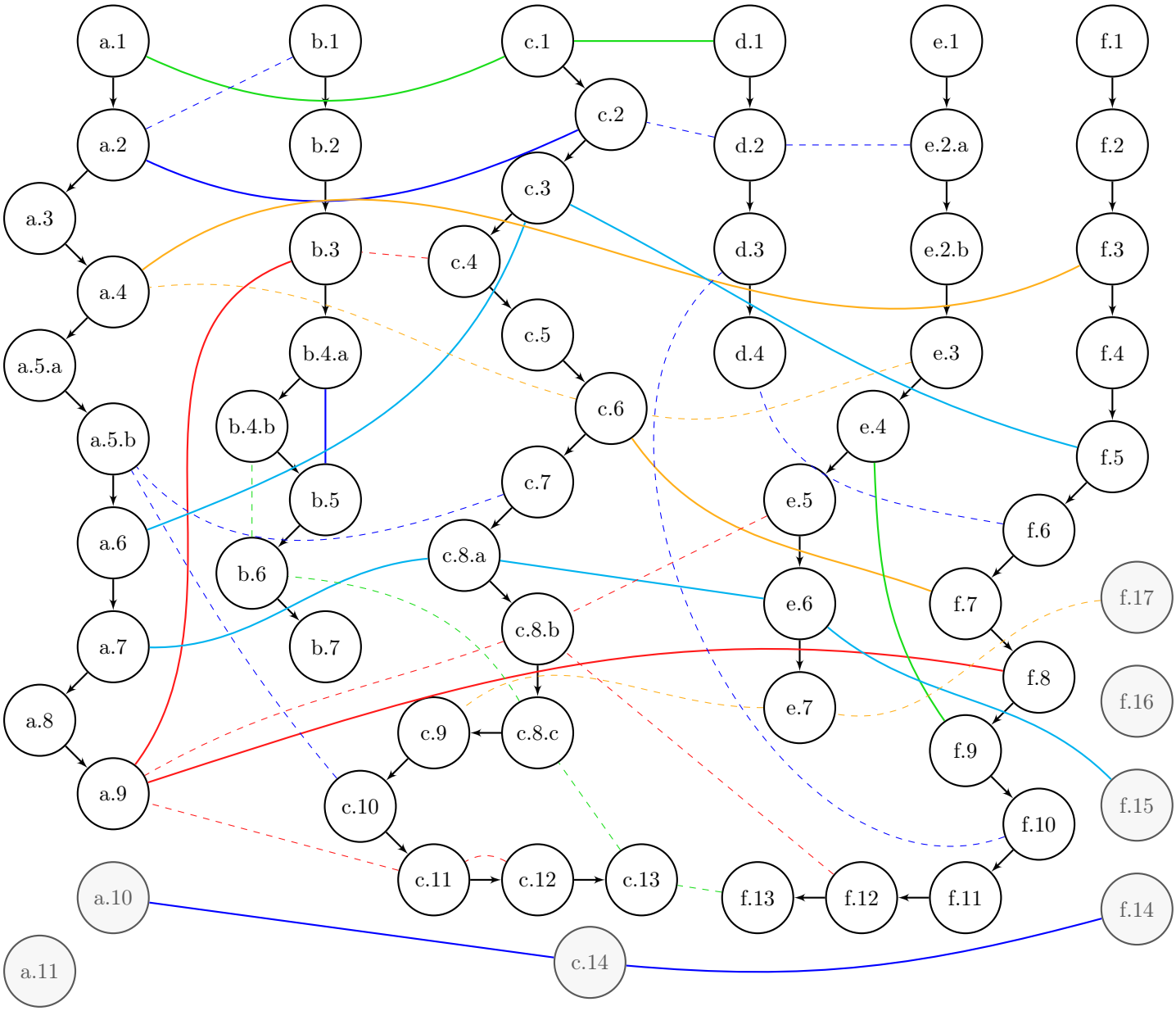


Figure 5.2: Illustration of the similarities between the identified architecture evolution paths.

strict sequence, in which Y directly follows X is denoted $X \xrightarrow{strict} Y$. We denote a group of steps in no particular order in a parenthesis, so the steps X and Y in no particular order is denoted (X, Y).

- **Sequences** are strict sequences, in which one step exactly precedes another step across several cases. E.g. steps $X, Y : X \xrightarrow{strict} Y$ are found in more than one architecture evolution path.
- **Stories** are a weaker notion that sequences (all sequences are stories), and describe common narratives in which steps may be separated by other steps, but occur in the same order in several cases. E.g. steps $X, Y : X \rightarrow Y$ are found in more than one architecture evolution path.
- **Group precedences** are a weaker notion than stories (all stories of length 3 or longer are group precedences), and describe situations where several steps (in no particular order) have been taken before some *resulting step* is taken. E.g. for a group of size 2, steps $X, Y, Z : (X, Y) \rightarrow Z$ are found in more than one architecture evolution path.

In order to limit notation, we later refer to common sequences, stories or group precedences by the notation that describes any occurrence of them.

In order to determine the significance of each common narrative we identify, we establish some measures. We use the term *weight* $w(n)$ of a common narrative n to denote the number of occurrences of the narrative across the architecture evolution paths. This gives us an indication of how seriously a narrative should be taken.

We use the term *step commonality* $c_{\text{step}}(s, n)$ of a step s in a narrative n to denote how many of the occurrences of the step are in architecture evolution paths in which the narrative is also present. That is, it is a measure of how commonly this step is involved in the narrative.

The step commonality leads us to define *mean commonality* $c_{\text{mean}}(n)$ and *median commonality* $c_{\text{median}}(n)$ for a narrative n, found by taking the mean or median, respectively, of the step commonalities $c_{\text{step}}(s, n)$ of all steps $s \in n$. A mean commonality of $c_{\text{mean}}(n) < 0.5$ indicates that, on average, the steps in the narrative appear in more architecture evolution paths where the narrative is not present than in those where it is.

We constructed a small program, *narrative-finder*, which identifies common narratives of the type defined above (see appendix E on the program implementation). We include in our results for further analysis all narratives with a weight of at least 2 (at least 2 out of 6 of the narratives we have constructed present this particular narrative), and where all step commonalities for the steps in the narrative are greater than 0.5 (on average, at least half of the cases in which a step is present is in the identified common narrative).

We eliminated weaker narratives, such that if a common narrative describes a subset of another common narrative *and* has a lower weight and commonality, we discard it.

The result was 1 equal section (in which steps were exact matches of each other, see table 5.2), 1 similar section (in which steps may have just been similar between stories, see table 5.3), 8 equal stories (see table 5.4), 11 similar stories (see table 5.5), 1 equal group precedence (see table 5.6), and 3 similar group precedences (see table 5.7).

#	**Common narrative**	**Weight**	**Mean commonality**	**Median commonality**
	Sections, equal			
1	a.1 First version: monolithic web application a.2 Rewrite of entire architecture	2	0.83	0.83

Table 5.2: Common narratives of the type *section*, looking at exactly overlapping steps, identified in the architecture evolution paths.

#	**Common narrative**	**Weight**	**Mean commonality**	**Median commonality**
	Sections, similar			
1	a.1 First version: monolithic web application a.2 Rewrite of entire architecture	3	0.80	0.80

Table 5.3: Common narratives of the type *section*, treating similar steps as overlapping, identified in the architecture evolution paths.

Despite eliminating some overlap, we still see a lot of the resulting common narratives describe the same thing. For example, equal section 1; similar section 1; equal stories 1, 2, 4, 5, 6, and 7; and similar stories 1 and 2 all describe the entirety or subsections of a common narrative, a.1 → a.2 → a.6 → a.7, but with various subsections having different strengths. This is 10 common narratives potentially describing the same occurrence: that a.1 is commonly followed by a.2, and these are both commonly followed by a.6 and all of these are commonly followed by a.7. The subsections of this chain that remain are the ones with a higher weight or stronger either mean or median commonality than the long chain itself, meaning that they might have a stronger connection than the long chain has. For example, similar story 1, a.1 → a.2, has a weight of 3, which is higher than the long chain, as well as comparable commonalities.

5.3 Discussion

In order to determine our results' support for existing research in change classification, we investigated the completeness and usefulness of two established change classification frameworks. With completeness we understand the ability for the frameworks to adequately fit the changes we observed during our inter-

#	Common narrative	Weight	Mean commonality	Median commonality
Stories, equal				
1	a.2 Rewrite of entire application a.6 Introduce scheduled jobs	2	0.83	0.83
2	a.2 Rewrite of entire application a.7 Introduce message bus	2	0.83	0.83
3	a.4 Introduce a parallel web application a.7 Introduce message bus	2	0.83	0.83
4	a.1 First version: monolithic web application a.2 Rewrite of entire application a.6 Introduce scheduled jobs	2	0.78	1
5	a.1 First version: monolithic web application a.2 Rewrite of entire application a.7 Introduce message bus	2	0.78	1
6	a.2 Rewrite of entire application a.6 Introduce scheduled jobs a.7 Introduce message bus	2	0.78	0.67
7	a.1 First version: monolithic web application a.2 Rewrite of entire application a.6 Introduce scheduled jobs a.7 Introduce message bus	2	0.75	0.83
8	a.6 Introduce scheduled jobs c.6 Add new web application strangling old one	2	0.83	0.83

Table 5.4: Common narratives of the type *story*, looking at exactly overlapping steps, identified in the architecture evolution paths.

#	Common narrative	Weight	Mean commonality	Median commonality
	Stories, similar			
1	a.1 First version: monolithic web application a.2 Rewrite of entire architecture	3	0.80	0.80
2	a.2 Rewrite of entire architecture a.7 Introduce message bus	3	0.80	0.80
3	a.4 Introduce a parallel web application a.7 Introduce message bus	3	0.88	0.88
4	a.2 Rewrite of entire architecture a.4 Introduce a parallel web application a.7 Introduce message bus	3	0.78	0.75
5	a.2 Rewrite of entire architecture a.9 Introduce a microservice	3	0.60	0.60
6	a.9 Introduce a microservice b.4.b Move a component to managed infrastructure	3	0.80	0.80
7	a.4 Introduce a parallel web application a.9 Introduce a microservice	3	0.68	0.68
8	a.7 Introduce message bus c.9 Geographically distribute component	2	0.83	0.83
9	b.4.b Move a component to managed infrastructure b.4.b Move a component to managed infrastructure	2	0.67	0.67
10	a.6 Introduce scheduled jobs b.4.b Move a component to managed infrastructure	2	0.67	0.67

Table 5.5: Common narratives of the type *story*, treating similar steps as overlapping, identified in the architecture evolution paths.

#	Common narrative	Weight	Mean commonality	Median commonality
	Group precedences, equal			
1	a.4 Introduce a parallel web application *and* a.6 Introduce scheduled jobs *Lead to:* a.9 Introduce a microservice	2	0.78	0.67

Table 5.6: Common narratives of the type *group precedence*, looking at exactly overlapping steps, identified in the architecture evolution paths.

#	Common narrative	Weight	Mean commonality	Median commonality
Group precedences, similar				
1	a.2 Rewrite of entire architecture *and* a.4 Introduce a parallel web application *Lead to:* a.7 Introduce message bus	3	0.78	0.75
2	a.2 Rewrite of entire architecture *and* a.4 Introduce a parallel web application *Lead to:* a.9 Introduce a microservice	3	0.65	0.60
3	a.4 Introduce a parallel web application *and* a.6 Introduce scheduled jobs *Lead to:* a.9 Introduce a microservice	3	0.78	0.75

Table 5.7: Common narratives of the type *group precedence*, treating similar steps as overlapping, identified in the architecture evolution paths.

views. With usefulness we understand the ability for the frameworks to shed further light on the research we are undertaking, that is, to enable us to find more patterns and similarities between the architecture evolution paths we have identified.

The two frameworks we investigate are Williams and Carver's SACCS-framework for architectural change [67]; and Chapin et al.'s framework for architectural change [13]. In particular, for the SACCS framework we consider only the classification of changes, as the many other considerations venture into a level of detail where it is unlikely to correctly describe changes based on our interview data. The SACCS framework's consideration of changes' effects on an architecture overlap with Chapin et al.'s framework, but Chapin et al. employ a more nuanced model, which will never result in less granular data than the application of the SACCS equivalent.

The SACCS-framework classifies architectural changes as belonging to one of four categories, based largely on the reasoning behind the change. This overlaps largely with out coding of architectural change reasons, as our data is based on open-ended requests for the interviewees to explain the reasons for the architectural changes they described as happening to their system. We answered the second question by attempting to place the reasons we had noted in the SACCS-frameworks classifications, and then discussing our findings.

For Chapin et al.'s framework, there was little overlap between their classification of architectural changes as based on concrete changes to the architecture. For this reason, we performed a reclassification of each of our generic architectural steps, placing the change in the fitting category in their framework. This process led to a discussion, which answered the third question.

5.3.1 Evaluation of the SACCS-framework's classification

In SACCS the motivation for a change is used for classification, rather than its impact. In the framework, changes are categorized as corrective (respond to defects), perfective (new or changed requirements), preventative (ease future maintenance) or adaptive (moving to a new platform, accommodate new standards) [67].

In order to determine how well the framework classifies the reasons given for architectural changes in our study, we attmpted to map the reasouns we found to the categories of the framework. The result of this mapping can be seen in table 5.8.

We found no satisfying way of classifying our change categories ACR.1, ACR.6, and ACR.8; and we found it difficult to find a satisfying fit for category ACR.3. ACR.1 can be argued to not be a true reason, and rather a reason for not making a change earlier. However, a reduction in infrastructure costs (ACR.6), wanting to follow industry trends (ACR.8), and improving performance of a platform (ACR.3) are all—when taken at face-value—meaningful reasons for making architectural change that are hard to classify using SACCS's classification.

Corrective ACR.9 Eliminate error sources (ACR.3 Poor performance under current conditions)	**Perfective** ACR.2 New feature requirement
Preventative ACR.4 Need for future scalability ACR.5 Developer experience improv.	**Adaptive** ACR.10 Escape vendor lock-in ACR.11 Adhere to bounding context
Not classified ACR.1 Migrating when infrastructure automation is ready ACR.6 Reduce infrastructure costs ACR.8 Industry popularity ACR.7 Unknown	

Table 5.8: The changes we encountered in our study, grouped by the SACCS framework's classifications.

A reduction of infrastructure costs (ACR.6) could, at a stretch, be classified as an adaptive change, but this classification seems to turn the chain of causality on its head: new standards are being followed for a business reason; the reason for the change is not to follow new standards or using a new platform, rather the use of a new platform follows from the need to reduce costs.

While following industry trends (ACR.8) may not be a good or logically defensible reason for architectural change, it was nonetheless encountered several times during our interviews, and had a real impact on the decisions made by the developers interviewed.

We reluctantly categorized improving performance of an application (ACR.3) as a corrective change, as problems are being experienced with the application, causing the change. The problems are, however, not necessarily caused by a defect in the application, but may instead be caused by changing circumstances. Categorizing the change as perfective may be a better solution, as this indicates the changing requirements for the platform, but a concrete change in requirements may not have been the cause for the change, as it may simply be the case that the application was defective or did not live up to already-established requirements. These changes become hard to classify because the interviewed businesses did not utilize formalized architecture requirements documents, and so it is unclear exactly what the requirements for the applications were, when poor performance was observed, and where the cause lies: in a change of requirements, or a defect.

The framework provides no way of classifying changes where no explicit rea-

son was given by the interviewees—at least not without a great deal of guessing based on the concrete changes we observe—so we did not classify these.

Several of the changes observed in the architectures we studied had several additive reasons behind them, for example both improving developer experience (preventative) and escaping vendor lock-in (adaptive). These changes would be hard to decisively classify using SACCS's classification.

Overall, we find it difficult to accurately represent the changes we identified in the SACCS classification framework. Too many of the changes we have observed would be entirely unaccounted for. We suspect that using the framework during interviews might yield a clear classification within the framework, but openly asking interviewees about their reasons for changes did not yield readily classifiable answers.

For this reason, and because many changes would fit into several classifications, we could find no way to use the SACCS classification to reveal patterns in the architecture evolution paths we have constructed. This means our conclusions on both completeness and usefulness for our subject of study are negative for the SACCS-framework.

5.3.2 Evaluation of Chapin et al.'s framework

Chapin et al.'s framework considers the impact of a change to an architecture, rather than the reason or intent behind the change, in order to classify them [13].

In our coding of the interview notes we noted every time a change was mentioned, and the type of step that was taken in the change. For example, we noted if the change extracted functionality from an existing component, added a new component, removed a component, or replaced a component with a new one. A few other types of architectural changes were observed too: moving to a more managed infrastructure, infrastructure as a service; introducing replication of a component; changes in communication patterns; and finally, replacements of the entire architecture at once.

Our classifications fits poorly with Chapin et al.'s framework, so a reclassification is necessary. We performed this reclassification by looking at each step in the architecture evolution paths we had established, and deciding the impact of the step, using the questions highlighted in figure 2.1. Chapin et al.'s framework explicitly deals with potential overlap, by assigning the highest-impact category to any change, making it possible to unambiguously classify a change. The result of our reclassification of changes can be seen in table 5.9.

We did not initially expect any changes to end up outside the major categories C and D, as we assumed all the changes we had talked about at least affected source code or infrastructure. As it turns out, the introduction of Bounded Contexts in the architectures did not fit into those categories, as it was a virtual change, a change to the company's perception of their architecture, rather than a change to any components in the architecture. Therefore we have included the category B-1 Reformative, highlighted in green in the table.

Change classification	Cases it occurred in (out of 6)	Total occurrences (out of 64)
B-1 Reformative	*3*	*3*
C-1 Groomative	2	3
C-2 Preventative	3	7
C-3 Performance	5	13
C-4 Adaptive	5	18
D-1 Reductive	0	0
D-2 Corrective	1	1
D-3 Enhancive	6	15

Table 5.9: Classification of changes identified in our study by Chapin et al.'s framework [13].

In order to determine the correctness of the categorization, we must find a way to evaluate it against the data we originally collected. One way of doing this is evaluating how well items that ended in the default categories, those at the end of each major grouping, fit the question asked for that category. For example, if a change affected functionality, the default category is D-3 Enhancive (*Did the activities replace, add to, or extend the customer-experienced functionality?*); if a change affected only source code, but not functionality, the default category is C-4 Adaptive (*Did the activities change the technology or resources used?*). The following cases presented some difficulty in this regard:

- One change involved rewriting a component in the hope of better living up to industry standards, and was classified as C-4 Adaptive, however neither technology or resources used were changed—the component was rewritten in the same technology using the same resources. We could identify no better classification for the change.
- One step involving extracting a part of one component and moving it to another, in order to better adhere to established bounding contexts, did not readily fit into any category, but a source code change did occur and no changes to perceived functionality occurred. The best fit we could find was categorizing it as changing maintainability, but the lack of a fit would otherwise have left it in the C-4 Adaptive category, where it fits even more poorly.
- Several cases that were originally classified as ACR.4 Need for future scalability were difficult to classify, as changes were made that affected performance (which would at least lead to a classification in C-3), but could also be argued to be a preemptive change to avoid future maintenance activities (which would lead to a C-2 Preventative classification), or they could be argued to change maintainability (resulting in a classification as C-1 Groomative). In the end, we classified them as Preventative, despite

the clear argument that preventing future maintenance by performing the maintenance activity presently is nonsensical.

These examples illustrate that, contrary to Chapin et al.'s claims, the framework does require degree of subjectivity in classification, where changes could be argued to fit in more than one category, depending on interpretation of the term *maintainability*, and how one perceives prevention of activities.

The C-4 Adaptive category was a broad catch-all category (with 18 of the changes we identified being classified as such). This suggests that the category may be advantageously split into smaller categories, offering a more nuanced view of the changes observed. For example, it might be possible to identify different types of technology or resource changes that we could meaningfully distinguish between.

Despite the framework's focus on impact on concrete architectures, some very similar changes (when perceived from the view of a service diagram, or when relying on the wording of interviewees) were categorized very differently. For example, introducing microservices may be classified as D-3 Enhancive if they also provide new functionality, C-3 Performative if they introduce performance improvements (e.g. better scalability than before), or C-4 Adaptive if they do neither of the prior ones. For microservices it is also an open question (and the answers will be very subjective) as to how it changes maintainability—could they reasonably be classified as C-1 Groomative?

This overlap of architectural change categories can be found in other research, too. Di Francesco, Lago & Malavolta found the biggest driver of migration towards microservices to be functionality, introducing new functionality with new functionality [30]. In Chapin et al.'s framework, the similarity in a migration towards microservices would be lost, as these changes would be classified as Enhancive.

When regarding just the number of changes in each category, we see the default categories containing most of the changes: C-4 and D-3 are by far the most common types of change. This could be due to a bias towards these categories, which would indicate that the categories do not accurately represent the nuance present in architectures. On the other hand, it could be argued that Adaptive and Enhancive changes should be the most commonly expected, when asking about architectural evolution: Adaptive changes would contain many architecturally significant changes, and change driven by functionlity would be categorized as Enhancive.

We found no changes to classify as D-1 Reductive (reducing functionality), which can be explained by such a reduction being unlikely to be architecturally significant, unless an entire component is removed at once. Only a single change was classified ass D-2 Corrective, which can be explained by a similar argument.

While Chapin et al. position their framework in opposition to others that require subjectivity in determining the category of a change [13], we found that a degree of subjectivity was indeed required for classification using their framework. There were several changes with sufficient uncertainty that the classification happened on the whim of the researcher, rather than a strong feeling

of security in the correctness of the classification. Many of these changes could likely be objectively classified given enough time and questioning of interviewees.

A big issue remains an uncertainty of terms, in particular the definition of maintainability, and what it means to change the maintainability of an architecture. The framework did not prove useful as a way of expanding our understanding of the architecture evolution paths we had constructed, as too much nuance was lots in the classifications, and too many similar changes (in the type of effect they had on components) were categorized differently.

While completeness could be argued for the framework, we did not find classifications that seemed to accurately represent all the changes we identified. Additionally, too many changes ended up in two categories, making it hard to say anything nuanced about the chronological ordering of changes, and as such the framework did not live up to the criteria of usefulness for our study.

5.3.3 Common architecture evolution paths

We explored some methods for finding common narratives in the architecture evolution paths constructed from the interviews conducted during this study. A common narrative can be seen as a first step towards establishing *evolution path styles*—types of evolution that are formally described in terms of prerequisites and abstract transformation steps, and impact [7]. Many of the resulting common narratives overlapped to some degree, so the large quantity of narratives can be seen as misleading.

In this section, we will briefly cover the unique common narratives we identified, and discuss which of them are likely to have any merit, and which we cannot find an explanation for. We cannot make any definite determinations about common narratives based on the small sample size (6), but aim to explain potential commonalities in web application architecture evolution that could be investigated further in the future. In addition, the discussion of each identified common narrative serves as a face value verification of the method of analysis we employed.

First, the long section $a.1 \rightarrow a.2 \rightarrow a.6 \rightarrow a.7$ likely describes two different common narratives that happened to be present in the same cases. The section $a.1 \rightarrow a.2$ describes an initial prototype of a web application being built, followed by a full rewrite. Note that this section occurred in half (3) of the cases we studied, and is a common occurrence. In fact, four of the cases had an early rewrite, but the last case's rewrite did not start from an undistributed layered web application, but instead a distributed one.

The story, $a.6 \rightarrow a.7$ is slightly less common (occurring in only two of the cases we studied), and describes the introduction of scheduled jobs to an architecture, followed by the introduction of a message bus. One explanation for this story could be the need for actions in the system to be performed irrespective of requests coming into the system. Scheduled jobs are one way of triggering actions without a request, and message based communication, supported by a message bus, is another. We observed a case in which scheduled jobs were explicitly extracted to microservices communicating over message bus, showing

that the same problem could be solved with both technologies, but with the message bus offering greater chronological granularity by responding to events immediately, rather than waiting for a timer.

We assume that the two stories are unconnected as we see no direct correlation between an early rewrite of the entire system and a later introduction of more complex actions in the application. An argument could be made, however, that an early rewrite may indicate greater willingness to make architectural changes.

Parallel web applications as a predictor

We found several common narratives in which something was preceded by the introduction of parallel web applications (a.4). Most (4) of the cases introduced parallel web applications at some point, which means that these predictions may be nothing more than the result of parallel web applications being common. In other words, the common narratives may be false positives.

The strongest common narrative was parallel web applications preceding the introduction of a message bus (weight 3, mean commonality 0.83, median commonality 0.83). We also found narratives in which they preceded the introductions of scheduled jobs and microservices.

All of these cases have in common that they are signs of a fairly complex architecture. We established the connection between scheduled jobs and message buses before this section, and both could be used as coordination mechanisms in more complex architectures with more than a single application. Especially with the introduction of microservices, where the parallel web applications and microservices need an efficient manner of communication.

The main question, however, is *why* parallel web applications commonly precede all these three patterns. Our best theory is that it has to do with taking a first step towards a more complex system architecture. The parallel web application means that more than one component exists in the system, but the components are still fairly traditional, both layered web applications. Once companies have experienced success with several components in their architecture, we posit, they may be more likely to seek even more complex architectures.

Full architecture rewrite as a predictor

Several common narratives had a full architecture rewrite (as covered in the beginning of this section) as a predictor of later changes. With varying degrees of certainty, we found rewrites predicting parallel web applications and then message bus ($a.2 \rightarrow a.4 \rightarrow a.7$); the introduction of microservices ($a.2 \rightarrow a.9$); and a group precedence of a full architecture rewrite and the introduction of parallel web applications predicting the introduction of microservices ($(a.2, a.4) \rightarrow a.9$).

This could be found to support the theory that an early rewrite shows a greater willingness to make architectural changes or build more complex architectures later in the lifespan of an application. On the other hand, rewrites are common (4 out of 6 cases), and the narratives may just be an expression of

common patterns that happen to be preceded by a more common pattern. All of these mentioned common narratives have a weight of 3, however, meaning that they are present in 75% of cases that contain a full architecture rewrite, supporting the theory of a non-random occurrence.

The case $a.2 \rightarrow a.4 \rightarrow a.7$ has 100% step commonality for the step $a.7$, the introduction of message bus, meaning that this is the case for all introductions of a message bus that we have seen, making it unlikely that it is entirely random. It is definitely possible to envision the introduction of a message bus without an early full architecture rewrite, but in our data set no such cases were found.

Microservices

The introduction of microservices was the result of a common narrative identified. We found the introduction of parallel web applications to predict it ($a.4 \rightarrow a.9$, weight 3, mean commonality 0.68, median commonality 0.68) and a group precedence of parallel web application and scheduled jobs introduction to predict it ($(a.4, a.6) \rightarrow a.9$, weight 3, mean commonality 0.78, median commonality 0.75).

The group precedence is strong, and makes sense at face value, too. As previously discussed, parallel web applications can be seen as a first step towards running several services in an architecture; and scheduled jobs indicate a need for more complex action in the system, not dependent on user requests. The microservice pattern can be used as a further step, that addresses some of the same concerns as the scheduled jobs.

What leads to moving to managed infrastructure?

The existence of ever more managed infrastructure offerings may itself be a reason that managed infrastructure becomes common. With managed infrastructure we mean services such as AWS Lambda or Google Cloud Functions offering managed hosting for serverless functions, and AWS ECS, AWS Fargate, or Google Cloud Kubernetes Engine offering managed hosting for containers [68, 71]. A move from a dedicated machine to a VM could also be considered a move towards more managed infrastructure, as could a move from a single VM to a cluster of automatically scaling VMs (such as AWS EC2 or Google Cloud Compute Engine [68, 71]).

While we applied no framework for determining which level of managed infrastructure an architecture moved to in a given step (in fact we know of no such framework), we can try and see if any general patterns emerge as to what leads to a move to more managed infrastructures. This is captured by the step b.4.b in the common narratives we have gathered.

We found the introduction of scheduled jobs ($a.6 \rightarrow b.4.b$, weight 2, mean commonality 0.67, median commonality 0.67) and the introduction of microservices ($a.9 \rightarrow b.4.b$, weight 3, mean commonality 0.80, median commonality 0.80) to predict a move to managed infrastructure. The latter has 100% step commonality for step b.4.b, and a move to more managed infrastructure driven

by the introduction of microservices passes face value validation: with many smaller services, any manual infrastructure maintenance will be forced to scale. Automating parts of maintenance, by using more managed infrastructure, will alleviate this pressure. The introduction of scheduled jobs, as previously noted, may be an indication that a more complex communication pattern is needed in an architecture, which we have previously connected with the introduction of microservices. Additionally, it is possible to move scheduling to managed infrastructure with platforms such as AWS CloudWatch [68].

We also found an interesting narrative showing that moving to more managed infrastructure once is a predictor of doing it again ($b.4.b \rightarrow b.4.b$). This highlights the lack of a framework for determining a level of managed infrastructure, as this would reveal if the narrative describes moves to similarly managed infrastructure, or moves to increasingly managed infrastructure.

Geographic distribution

A single common narrative led to geographic distribution. The step predicting geographic distribution was the introduction of a message bus ($a.7 \rightarrow c.9$, weight 2), and has 100% step commonality for introduction of geographic distribution. We can read this as geographic distribution being more likely to be introduced in fairly complex architectures with complex communication. It is conceivable that geographic distribution could be introduced in architectures with nothing more than a replicated layered web application, but we found no such cases. It may be an indication of which point in an architecture's lifetime geographic distribution becomes a concern.

Evaluation of the applied method

As the method used for constructing architecture evolution paths and analyzing them for common narratives is novel, a discussion of its applicability and usefulness is merited. We did identify interesting common narratives, a novel result, and establish support for them, and the method allowed for a discussion of the interaction of various common narratives, so we can conclude that the method was useful and applicable to interview data. However, the application of the method was experimental and we discovered several manners in which it may be improved before being applied in the future.

Granularity and abstraction level of steps In constructing the architecture evolution paths from interview data, some changes as reported by interviewees contained several concrete architectural changes, on the abstraction level of an operator in the terminology of Barnes, Garlan & Schmerl [7]. This led us to split these steps into several smaller steps (e.g. b.4.a and b.4.b occurred at the same time), in order to be able to more readily map similarities between steps. Unfortunately, these steps were represented as strictly in order (e.g. b.4.b strictly following b.4.a), rather than occurring at the same time, when we were looking for narratives. A manual check found no missed common narratives for

this reason, but in the future it would be useful to be able to represent these operators as occurring at the same time, and allowing them to appear in common narratives in any order.

The first steps in each architecture evolution path were noted as one single step, creating an entire architecture at once. This obscured similarities and differences between the architectures when regarding only the architecture evolution paths, and weakened some of the common narratives identified. For example, one architecture started in somewhat managed architecture, but we had no way of representing this as similar to the other architectures that later moved to managed architectures. An alternative approach would have been representing each initial architecture as a simple layered web application and subsequent operators applied to it, all happening at the same time. Mapping similarities between these operators and those used later might have revealed, strengthened, or weakened common narratives, and thus provided us with data more accurately representing the common progression of layered web application architectures.

We considered two levels of similarity in the paths we analyzed: steps exactly equal to each other, and steps similar to each other. It might be useful to reevaluate this choice, and consider whether a more granular similarity model is needed, e.g. a level of similarity in which adding a microservice is not regarded as being similar to introducing the concept of microservices (which would occur alongside the first microservice's introduction).

We found it difficult, in the interviews, to get a detailed description of when each component in the architecture was introduced. E.g. once the concept of microservices was established in an architecture, further microservices added were not found architecturally significant, and was skipped over by interviewees. We could have embraced this, and changed the abstraction level of our architecture evolution paths, considering only changes in patterns and not individual components. Experimentation on this front is needed in order to find the right approach.

Elimination of uncommon narratives We decided to eliminate from manual analysis any common narrative n containing any step s with a step commonality $c_{\text{step}}(s, n) \leq 0.5$. This decision may have inadvertently eliminated some narratives in which common steps produce uncommon results, which would be an interesting result. To investigate whether this was the case, we listed the narratives eliminated due to low step commonality where the last step had a step commonality of 1. This produced a common narrative leading to the establishment of geographical distribution (c.9), but none of the steps seemed particularly connected, at least not moreso than what was already covered by common narratives we included in our manual analysis.

We also found an eliminated narrative where the introduction of parallel web applications led to the introduction of serverless functions, which in turn led to the introduction of microservices. This narrative runs counter to our intuition, in which we expect more managed infrastructure (e.g. serverless) to follow less

managed infrastructure (e.g. microservices), not the inverse, as we see it here. It may, however, be a common narrative that was eliminated, and therefore we are including it in this discussion.

Identifying interesting common narratives We chose two metrics in order to assess the significance of a common narrative, weight and commonality. More heuristics would be interesting to consider, such as the length of a common narrative. Additionally, even after attempting to eliminate duplicate narratives, we had to consolidate several results representing the same common narrative in different ways (different subsections with a stronger commonality or weight than the full section). We were very careful not to eliminate potentially interesting common narratives, and therefore chose a conservative elimination strategy, but this could be further refined in the future.

Reasons for change Despite collecting data on reasons behind the architectural changes made, we found no useful way to include it in the analysis of architecture evolution paths. This seems like an obvious deficiency, as the driver of changes is entirely left out of our consideration, and much research in the architecture field indicates that drivers of change are significant to consider.

Classification of managed infrastructure Our classification of moving to managed infrastructure was vague and nondescript. A framework for this classification, allowing for a comparison of different levels of managed infrastructure, would have been helpful in shedding light on how architectures are migrated towards managed infrastructure environments—whether it is step-wise, or all at once, or something in-between.

5.4 Validity and reliability

The reliability and validity considerations mentioned in 3.4 apply wholesale to the findings of this chapter. Some further considerations can be made.

5.4.1 Reliability

Turning the concrete architecture timelines into something resembling architecture evolution paths was a wholly subjective process, and likely to yield different results if performed by different researchers. The identification of interesting common narratives, however, was performed based on objective measures and using a fully mechanical method.

5.4.2 Validity

When looking at validity, we consider the same types of validity as mentioned in 4.4.2, namely face validity, social validity, empirical and content validity, and external validity.

First, when looking into the applicability of other change frameworks to our study, we found that they were not useful. This runs counter to existing research, as both frameworks were established based on existing data. It is, however, what we expected to see from the outset, when applying the frameworks with a specific goal in mind that does not necessarily align with the (implicit) goal of the frameworks. The result is useful because it shows that none of these frameworks are universally applicable—only a single case is needed to show this. In terms of social validity, we consider it a useful result, as it may guide future work in architecture change classification, and open up the possibility of nuance and variety in frameworks.

Now, considering architecture evolution paths, we see some cases counter to what we would expect. For example, an expected result could have been seeing a move to managed infrastructure leading to the introduction of microservices, then leading to the introduction of serverless—as a natural progression towards more and more managed infrastructure—but we see the last two steps of this expected order reversed. On the other hand, many of the cases we found were explained by plausible (albeit untested) theories of architectural progression in the discussion, which supports face validity.

The results are not particularly actionable outside of academic circles, other than in providing a perspective on some common paths that layered web applications take when changing over time, which may inspire practitioners in their decisions. A method of determining the quality of various paths would have made the results much more useful in this regard.

With regards to content validity, we found no way of integrating reasons for changes into our analysis, which is a limitation of the analysis, meaning that it does not, in fact, cover all of the similarities we may find in architectural evolution. If we limit our expectation to that which is captured by operators, however, this concern goes away, and the study is indeed covering.

We found architecture evolution paths to be a useful lens through which to view the changes that happen to an architecture. As such, our research supports this existing research.

5.5 Summary of findings

We evaluated two change frameworks for completeness given our observed changes and usefulness for our study. We found one framework to be complete, but with many inadequate classifications. We found the other framework to be incomplete, as it was not possible to classify all the changes we observed. None of the frameworks were found to be useful in our context of study, as none of them classified changes in a way that allowed us to find patterns of changes shared between architecture evolution paths.

These findings support the view that different taxonomies and frameworks for classification provide different views, but are rarely universally applicable. In the end, a search for an ultimate change classification framework may be futile—a more fruitful approach may be found in better describing the scenarios

in which various classification frameworks provide value, and which value it is they provide, and thus allow software architecture researchers and practitioners to choose the relevant frameworks for their endeavours.

We posited some theories that might explain the narratives we identified. These are untested, and the obvious next step would be investigating these, before they are taken as established. The theories indicate only that we assess that there *may* be a reasonable explanation for the common narrative. A first step towards such an investigation would be performing more studies with a similar methodology to the one employed here, in order to collect a larger data set.

As the research method applied in this study is novel, we found several potential future improvements, from the way architecture evolution paths are constructed, to which facets are included (notably, we did not include reported reasons for change), and better heuristics for determining which narratives merit further analysis.

We found several common narratives which we could best explain by seeing certain steps as an indication of either willingness to make architectural changes, or an indication of complexity of an architecture. It may turn out that there are better explanations for these common narratives, given further study and more data points.

We found strong support for the common narrative that is writing a prototype of a web application, followed by a full rewrite. We found strong support that architectures commonly experiment with two parallel web applications before moving towards an architecture with more moving parts, such as a microservice architecture. Finally, we found that architectures that have moved components towards more managed infrastructure often do so again in the future.

Chapter 6

Data in web applications: generation, persistence and consistency

Summary This chapter first connects the fields of distributed systems research and database research to that of web engineering. First an argument that web applications are inherently at-best weakly consistent systems is provided, along with a perspective on how web applications should be considered in distributed systems terms.

Secondly, a taxonomy for data and data generation in web applications is presented, hinting at some semantics and allowing for some practices from database research to be applied to web applications. The taxonomy further hints at a larger theoretical framework for understanding web applications, which has been left for future work.

The taxonomy is evaluated through an application to the interview data from the study presented in chapter 3, providing an answer to research question RQ5 by illustrating some of the experimentation with data refinement and persistence present in web applications, as well as where there are opportunities for further exploration in this regard.

6.1 Data and consistency in web applications

In chapter 4 we established a breadth and variety of web application architectures, especially showing complexity of architectures of web application servers. We introduced, in the background, a definition of web applications allowing for such breadth. In this section we will use this definition to link the fields of database and distributed systems research with web engineering.

6.1.1 Web applications as database systems

We can draw some parallels between database system research and web applications. A typical web application is a tiered application, in which some state is stored in a database. Barring applications that rely on event sourcing, this state is usually limited to the current application state, e.g. entries representing users or products. Event sourcing gives us a way of understanding this state as a view over base relations (in this case events). Whether we persist the events that occur in a system (as event sourced systems do), or we do not, the events do occur and have an effect to the application state. In event sourced systems this change is made explicit, as the creation of a new event in the event store results in the modification of some aggregate, which is a view whose base relations are events in the system.

In applications that are not event sourced, the business logic that updates application state based on the event, can be seen as manual view maintenance, updating the user or product, based on events that would in an event sourced system record happenings such as *user created* or *order paid*. This view maintenance often happens after a request has been made, and before a response has been sent to the web client. This categorizes this type of maintenance as eager, but as current state is often taken into account it can also be seen as incremental.

The responses a web application generates can be seen as another, more refined, type of view. It is based on the state of the application, and usually generated once requested. In CGI, this is part of what the application program would do. These views are not materialized, as they are only generated once requested, and no maintenance happens on them.

Neither responses nor application state are ever stale from the application's point of view, as they are eagerly maintained or fully rebuilt on request. It leaves open the obvious question, however, of how database performance improvement techniques might be utilized in web applications. Event sourcing is an example of one such approach, in which events are explicitly captured, and a more automated form of view maintenance happens.

Likewise, we can draw a parallel between database systems caching data on the client, or guaranteeing certain consistency properties through client implementations, and the use of browsers as clients in web applications. The client in a web application can be used, for example, to ensure read-your-own-writes consistency from the user's perspective, showing always the most recent state for that client, even before it has propagated to the server, or before it has propagated to all nodes of the server-side application.

Often, propagation to the server, from the perspective of the client, will simply be equivalent to propagation to the server's underlying storage system, which will likely be one or more databases or storage systems in the sense of those systems discussed in the previous sections.

In the above, we have illustrated that many of the interactions of web applications can in fact be thought of in terms of distributed systems and database research. Doing so, would allow the application of practices from those fields to

be applied to web applications, few of which are in use in the industry currently.

6.1.2 Web applications are distributed systems with weak consistency

When seen from the eyes of the user, web applications, like databases and storage systems, act as an interface for interacting with state. Users will have similar expectations of consistency and correctness of the data they are presented, and therefore it is equally meaningful to talk about the consistency properties of web applications. Given the broad definition of web applications given in the background chapter, we will argue that web applications should be considered to be weakly consistent systems, and that attempts at creating more consistent web applications is generally untenable. This perspective can be used to inform decisions on consistency tradeoffs in the constituent parts of a web application. For example, the question of consistency within the backend of a web application is no longer simply which kind of consistency is acceptable, but which kind is acceptable given that the user will never perceive the system as wholly consistent.

With storage systems the consistency property describes how the state in the system is perceived or how it propagates through the system. Clients are seen as external to the system, and the system guarantees are about how queries for data will be perceived on the clients. The clients may be implemented in such a way that they provide stronger guarantees (e.g. read-your-writes or monotonic reads), but when considering the consistency of the state of the system, the client is not included as a consistent component of the system.

One could argue that the same should be true for the client-server relationship in web applications, that consistency is only considered on the server. In a web application, however, according to our definition, both client and server are seen as execution environments, potentially performing business related operations on business state. From a user perspective, and in accordance with the definition we have of web applications, execution on the client of a web application happens on state of the web application. The state of the web application is not simply from the server and inwards, but extends to the client, and so must consistency properties, in order to be considered properties of the application.

The opposite position, that clients should not be considered as part of the application when considering consistency, relies on a primitive and transactional notion of client behavior, in which users are explicitly aware of client transactions with the server. This makes sense for storage systems, in which transactions performed by the client are explicit, and made sense in web applications before clients became execution platforms; however in the rich user interfaces of many modern web applications this is not the case. In particular, read-your-writes and monotonic reads ensured by client-side caching in storage systems makes sense exactly because the client performs actions such as reading or writing values, and nothing more complex. A web application client commonly executes long-running code, and contains local state that is acted upon without verifying its freshness with the server every step of the way.

The consistency of a web server can be guaranteed in exactly the same way as that of any storage system, as a web server will typically depend exclusively on a storage system for its state, sometimes with some caching in the application code, in which cached elements are invalidated as new elements are written to the storage system. We can also ensure the consistency of the writes of a client with the server (i.e. that the client does not indicate a successful write before it has been confirmed by the server), and using techniques similar to read-your-writes we can ensure that these appear immediately consistent on the client, too, as view maintenance of the state on the server happens eagerly.

Consistency becomes infeasible when considering how users perceive state from web application clients. For example, if a client receives a response from the server, and allows a user to act on it, a scheduled job may change the state on the server. This change does not automatically propagate to the client, and so the user will be acting on stale data when they interact with the client, and a potential submission that feels like a change may overwrite other changes, never seen by the user, on the server.

A common example of the importance of consistency in applications is banking. Even under the assumption that the business logic in the server application functions as it should, ensuring consistent state and consistent transactions, users may make decisions they did not want to, due to the disconnect between client and server state. Shklar & Rosen discuss *dirty reads* from a database from the perspective of a web application server, where a request to read data is made while a transaction modifying the data is processing, noting that they "could be considered acceptable", but recommend avoiding them in critical applications, such as banking [61]. The concern is that a dirty read may lead user A to see an account balance higher than it actually will be, once the user acts, because user B's transferal of money to a different account is processing.

However, even without dirty reads, as should be evident from our discussion of server and client state above, this situation is possible. If user A requests an account overview, the server responds, and *then* user B performs the transferal, the state perceived by user A when making their decision (after user B's transferal has completed) will still be stale: the web application server does not propagate updated values to clients. That is, avoiding dirty reads does nothing to alleviate the problem, except in exceedingly rare edge cases. In fact, we would consider the case we describe to be more common than a read happening *during* a database transaction. The consequence remains the same: user A may perform an action that they would not have performed given accurate up-to-date information about the account balance.

While web applications do not commonly propagate state to clients, it is definitely possible. Technologies such as WebSockets would allow servers to propagate changes (or notices of data invalidation) to the relevant clients [75]. While this can be feasibly used to ensure eventual consistency (and solve many cases of the issue described above), it becomes infeasible to ensure strong consistency in this manner for any significant number of clients.

While a distributed system can ensure strong consistency given that the number of nodes written to for each change added to the number of nodes read

from for each read is greater than the total number of storage nodes in the system ($W + R > N$, as introduced in the background chapter [66]), this is not the case for web applications. In web applications, as noted before, clients do not expose transactions as their way of interacting with the system. Each client displaying information from the web application is an active client, from which users are perceiving state, not as the result of some previous request, but as interactive. As such, the system would have to propagate the updated values to *all* active client before confirming a write, in order to ensure freshness of all perceived values in the system. This would result in incredibly long write times in the case where any significant number of clients are connected to the system.

In other words, it is infeasible to wait for updated values to propagate to all active clients before confirming a write, but writes can still be used to ensure eventual consistency of clients.

Accepting that the web application will never be perceived by users as strongly consistent, and accepting clients as parts of the application system, opens up new possibilities. Clients can, for example, update state from each other, rather than relying solely on a server to hold state. Initiatives such as WebTorrent are potentially an early step in this direction [78].

To summarize, we presented in this section an argument that web applications can never be strongly consistent. While the web application's server-side can be considered in the same way as any other distributed system, the inclusion of active and executing clients changes the users' perception of web applications, and hence the way consistency should be conceived of in web applications. Web applications will at best be perceived by users as a type of eventually consistent.

6.2 A taxonomy of data in web applications

Earlier in this chapter, we introduced a perspective on web applications, viewing them through the lens of database and distributed systems research. In this section we expand on this perspective, introducing a taxonomy of data in web applications, that allows for the categorization along lines of these observations, focusing on how and when data is generated, and whether or not it is persisted in the system.

As we noted before, a holistic view of a web application will have to take into account both client and server parts, and the server of a web application may itself be any arbitrarily complex system. In order to focus our taxonomy, we center the terms we use around the communication between client and server, thus finding commonality between all web applications.

Web applications that employ caching may do so at many levels of the application, and at various degrees of aggregation and refinement, with each caching layer containing data in a format as close to the needed one as possible. For example, a Varnish reverse-proxy cache will cache responses exactly as they will be needed to send to the client [65].

In event sourced applications, the most basic data is referred to as events,

and what would traditionally be considered the application state is derived from the events. In event sourced systems, events can be replayed to regenerate the state, and while the state is persisted in the system, it is not considered the base truth [25]. We argued that these events exist whether or not they are persisted. In many applications they are not, and this means that state cannot be rederived, but it still is initially derived from events that occur in the system.

This distinction into more or less basic data types is also used in the *Lambda architecture* introduced by Marz & Warren in a 2015 book, where the most basic type of data is referred to as *core data*, and everything else in the system is derived from this [49]. We define the *rawness* of data in opposition to the *refinement*: the more basic data is, and the more accurately it represents input to a system, the more raw it is.[1] The more data has been changed for the purpose of representing what the system concerns itself with, the more refined it is. For example, a full HTTP request is more raw than an event, and the state typically stored in a web application database is more refined than events.

The most raw data in a system will typically be considered immutable. Once an event has happened it does not make sense that it should later be considered to not have happened. Refined data, such as current state of an application, is usually mutable.

We define as the *source of truth* of a system the most raw data that the system persists. That is, from the source of truth, more refined representations of data can be derived, but if the source of truth is somewhat refined, it is usually impossible to find more raw versions of the data. We can derive state from events, but not vice versa.

Marz & Warren abstractly represent the derivation of more refined data as a function being called with the entire core data set as input. That is, every refined type of data in the system can be described by some function that takes all the core data as input [49]. In event sourcing databases such as Event Store, this is represented by projections, which define a set of events that they expect to take as input, and describe how these events modify the state of the resulting object [69].

In practice it may make sense to view the refinement as a stepwise process. For example, some refined data may be more accurately described as depending on already-refined data, rather than the most raw data of a system. That is, the output of one refinement function may be part of the input of another. The ultimate result could technically be computed from the original data, but the mental model provided by seeing it as a stepwise process is desireable: it allows us to deconstruct what happens in web applications as a set of logical refinements that may be combined in various ways, not as black boxes with ultimate output. For example, both a user profile page (highly refined) and an overview of users (highly refined) may depend on the same user profile (constructed from events).

For consistency, we will henceforth refer to the rawest meaningful form of data in a web application (representing some request or event in the system,

[1] This definition of rawness is in agreement with that used by Marz & Warren, who propose how much data can be derived from data as a proxy measure of its rawness: "[t]he rawer your data, the more questions you can ask of it" [49].

from which all other data is derived) as *raw data*. Data with any degree of refinement we will term *refined data*. The most refined types of data in a web application we will term *responses*. Responses will typically be the exact content of a response sent from a web server to a client, or a prepared page on the client that it can display. Figure 6.1 shows the typical flow of refinement in a web application server, where data is created as raw data, and is refined into refined data and eventually responses.

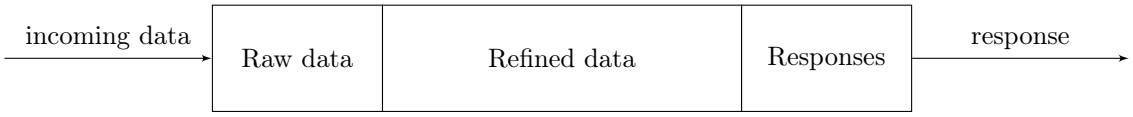


Figure 6.1: Data in a web application arrives to a web application server in the form of raw data, and is refined, eventually into the form of responses.

All web applications contain all three types of data—raw data, refined data and responses—but it varies from application to application which they choose to persist and where. Data that is not persisted we term *transient*. Refined data is a view in database terminology, and persisted refined data is therefore a materialized view. We can consider modifications to persisted refined data in the same manner as we consider view maintenance.

Responses are commonly persisted in caches such as Varnish, where the response is stored between the client and server on a reverse-proxy server (conceptually part of the server in the web application, but often in practice a separate server from the web application server). The reverse-proxy server short-circuits requests for unchanging content, or dynamic content which can safely be cached, and responds to request for this data without consulting the web application server, saving computation time [65].

Consider the example of a simple MVC web application (such as an application built with Ruby on Rails). In such an application, requests come in, but are not stored in any form. The state of the web application is modified based on the requests, and finally responses are built on the fly and sent back, but also never stored [63]. As such, these web applications persist only refined data, but raw data and responses are transient.

6.2.1 Refinement of data

We can define any refinement step of data as a *refinement function*, which takes as input the less-refined data on which it depends. In practice, a refinement will typically depend on one or more types of data filtered by some parameter. For example, a user profile (a domain object, refined data) will depend on user-events that concern the user with the ID of the user, whereas a user profile page (response) will depend on a user profile with a specific ID.

The approach taken by Marz & Warren of abstractly defining refinement functions over the entire dataset avoids having to deal with complex semantics

of the data and data dependencies [49]. In Event Store, the approach is different, and it is possible to pick the relevant streams based on any of their contents. Every time a new event arrives in Event Store, it will be tested by the filter to see if it should be passed to the refinement function (called a `when`-step). Finally, Event Store provides a Partition function, which, instead of including or excluding events based on a predicate, partitions by some field [69].

Partitioning provides a useful way of describing refinement functions that take several types of data as input and produces multiple results. In our example of user profiles, our refinement function would depend on relevant events (from the relevant streams), partitioned by that user ID. Listing 6.1 shows how the code for such a projection might look in Event Store.

```
fromStreams([ "user_created", "user_email_confirmed", "user_deleted" ])
    .partitionBy((event) => event.body.userId)
    .when({
        $initShared: () => {
            return {
                deletedAt: null,
                emailConfirmedAt: null
            };
        },
        user_created: (state, event) => {
            state.id = event.body.userId;
            state.username = event.body.username;
            state.email = event.body.email;
            state.passwordHash = event.body.passwordHash;
            state.passwordSalt = event.body.passwordSalt;
            state.createdAt = event.body.timestamp;
        },
        user_email_confirmed: (state, event) => {
            state.emailConfirmedAt = event.body.timestamp;
        },
        user_deleted: (state, event) => {
            state.deletedAt = event.body.timestamp;
        }
    })
    .outputState();
```

Listing 6.1: An Event Store projection describing a user based on events.

This covers abstractly how a refinement function should be seen: a function that depends on one or more types of data, potentially partitioned by some variable, producing one or more results. A refinement function may rely on any kind of data, and not exclusively events.[2]

[2]Event Store has a similar approach, where projections can result in new streams, on which further projections can be constructed [69].

To add more complexity, a refinement function may depend on one type of data that is partitioned by some variable, and another that is not. For example, a user profile page may have a side bar advertising the most sold products through the web application. The refinement function outputting the user profile page would then depend on users partitioned by ID, as well as the list of most sold products (a list itself refined from information about products and purchases), which would be the same list for every user profile page.

Finally, a refinement function may depend on several data sources partitioned by different variables, in which case it should would create every combination of the two variables. For example, a product page on a website that displays the current user's name when the user is logged in, will describe a page constructed from any combination of product and user.

Recall that refinement functions are abstractions, describing the way in which data is derived in web applications, and by no means practically applicable as presented. Constructing every product-user combination would be infeasible.

Instead, a refinement function, given the variables its input is partitioned on, can produce the exact result stemming from that variable combination. This will be more similar to what concretely happens in a web application when a logged in user requests to view a product: the page for that specific user and that specific product is generated, and then returned.

We use the term *response generation strategies* for decisions on how and when to execute various refinement functions involved in the production of a particular response. We will leave the formalization of refinement functions and further exploration of response generation strategies as future work.

For now, the importance of a refinement function is that it describes a refinement of some data into more refined data, which conceptually describes what happens when state changes occur in web applications, as well as when responses are generated.

In the case of a simple MVC web application, a request to create a new user will result in a virtual `user_created` event entering the web server, which results in the update of the application state, creating a user entry, theoretically seen as a refinement function reacting to the event, and finally a response is generated. If the resulting page is a user profile page for the newly created user, the page generation can seen theoretically as a refinement function refining the user entry in the database into a HTML response.

6.2.2 Data refinement triggers

We hinted briefly, when introducing the concept of response generation strategies, at the time of generation of a certain type of data being significant. Several types of data refinement triggers exist:

On request refined data types are generated exactly when they are requested. A great example of this is responses from web application servers which are typically generated when they are needed: when a user requests to

see the index page of a forum, the page is refined from the collections of forum posts and users that exist in the application.

Cached data types are generated on request, and then stored until they expiry. Expiry may be determined by a time-to-live or be managed manually by the application. Responses are commonly cached in reverse-proxy caches.

On change refined data types are generated when their underlying data changes. If a data type is generated on change, all the data types it depends on must also be generated on change (or the change would not propagate). This is the model we see in web application server state both with or without event sourcing: new events change the underlying data, when results in either the server's application code or the event store updating the state.

At intervals refined data types are generated from their underlying data types at set intervals.

Pre-requested data types are generated when a request for their generation comes in. They differ from data types generated on request in that the request for generation and the requisition of the actual result are disconnected. A typical example of pre-requested data types are reports that are generated from large amounts of aggregate data, to be viewed once generation has completed.

It is evident that certain of these data generation times are incompatible with strongly consistent systems. Data that is generated at intervals or that has pre-requested generation will necessarily be outdated when underlying data changes between generations. Caching (or, more precisely, cache invalidation) has to be applied carefully in order to ensure consistency.

From the perspective of a web application server, refinement on request will be strongly consistent with the underlying storage system as long as the generation itself does not allow write requests to finish. Even then, the generation time makes it harder to ensure consistency.

Likewise, web applications can still ensure consistency with data types refined on change, but this will require not confirming the write before it has propagated to those data types. Generally, as is the case in Event Store, some time delay in propagation (and thereby eventual consistency) is accepted in these cases [69].

6.2.3 Taxonomy

The taxonomy introduced in the previous sections can be summarized as follows. Data in web application can be described by its rawness, which is the inverse of its refinement. Data enters the system as raw data, and its refinement can be explained theoretically as passing through refinement functions, each function outputting a more refined type of data. The most refined data in a web application is responses. Raw and refined data, as well as responses, may or may not

be persisted in the application. All web applications have these types of data, and transient data is that which exists but is not persisted in an application. Data types refinement can be triggered at various times: on request, on request with caching, on change, at intervals, or pre-requested.

This taxonomy allows us to draw parallels between web applications and database-related research such as that on materialized views and view maintenance, as persisted refined data behaves like materialized views. It also provides a novel perspective for exploring how concrete web application architectures behave, and which performance tradeoffs they make.

6.3 Approach to evaluation of the taxonomy

In order to answer research question RQ5, we performed a separate coding of the interview transcriptions from the study presented in chapter 3.

RQ5 Which types of data is persisted in web applications, and what triggers its generation?

The coding was according to a coding legend mapping the taxonomy presented in section 6.2 to codes, investigating the data types, their generation triggers, and their persistence location in the interview data gathered through the interviews presented in chapter 3. The coding legend can be found in appendix C.2.

It is relevant to note that the study did not explicitly concern data types in the applications under study, but concerned instead architecturally significant changes. We are therefore not expecting to see exhaustive results, but rather an indication of those uses of specific data types that are architecturally significant.

6.4 Relevant results

We noted the generation triggers and persistence location (if any) of each data type interviewees mentioned in the interviews about their architectures' evolutions. Table 6.1 shows an enumeration (first column) of the types of data we encountered (second column) as well as how many of the cases we encountered them in (third column). Note that events and request data have no generation time listed as events are raw data, and therefore are not generated by the system.

We generally see that the most common case (6) are the data types found in a simple MVC web application: request data is transient (q.1), events are transient (e.1), refined data is persisted on change (r.3), and responses are transient and generated on request (s.1).

Two cases report or discuss a future of storing events in a database, which gives the ability to reconstruct the aggregates in question at a later time. Two cases had caching of refined data in the memory of an application, and one even cached the data on the client (in the browser), relying on the client to

#	Data type	Number of cases
q.1	Request data, transient	6
e.1	Events, transient	6
e.2	Events stored in database	2
r.1	Refined data, cached in application in-memory cache	2
r.2	Refined data cached in client	1
r.3	Refined data generated on change, persisted	6
r.4	Further refined data generated on change, persisted	2
r.5	Refined data generated at intervals, persisted	1
r.6	Refined data pre-requested, persisted	1
s.1	Responses generated on request, transient	6
s.2	Responses cached in cache-specific storage	1

Table 6.1: The data types observed, their generation triggers, and persistence location, found in the investigated architectures.

send the data back to the application for it to be used. The data in question was encrypted, such that the server could verify its integrity, allowing for faster processing of the client.

Two cases used second-level refinements, generated on change to lower-level refined data. These uses are similar to the cases of second-level refined data generated at intervals (1) and those of pre-requested second-level refined data (1). All of these approaches provide more refined values, closer to how they are going to be used in the future, reducing future generation time.

A single case reported caching of responses, by using a reverse proxy cache, which was the only case with non-transient responses.

6.5 Discussion

In order to further investigate which types of data are persisted in an application, we can compare the number of cases where request data, events, refined data and responses were persisted and those in which they appeared in transient forms. Table 6.2 shows an overview of the locations the different types were persisted in

	Transient	Persisted in database	Persisted in application	Persisted in cache store	Persisted on client
Request data	**6**	0	0	0	0
Events	**6**	2	0	0	0
Refined data	0	**6**	2	0	1
Responses	**6**	0	0	1	0

Table 6.2: Persistence location by data type

While the pattern reported above as the most common does indeed shine through again, we can also note something else. While events are persisted in some cases (2) and a single case mentioned persisting responses, we find no cases in which request data was persisted. (We also see that refined data was never mentioned as transient, although it likely also appeared in this format in all applications—but it was not thought architecturally significant by interviewees and therefore not mentioned.) The overview shows that there is some spread in where data is persisted, but most storage locations are limited to a few types of data.

Figure 6.3 shows an overview of data types and the data generation triggers reported for them. We see some spread on the types of generation triggers for refined data, but very little for responses. That is, responses are usually generated on request, and at most cached. Caching is the most common type of modification to the default MVC web application generation behavior.

	On request	**Cached**	**On change**	**At intervals**	**Pre-requested**
Refined data	0	3	**6**	1	1
Responses	**6**	1	0	0	0

Table 6.3: Data generation triggers by data type

The taxonomy provides a new lens through which we can view data in web applications and helps us pose some questions: could we find benefits in persisting more types of data (i.e. request data, events or responses), persisting them in a different location, or in triggering the generation of data types in a different manner? This leads us again to the concept of response generation strategies, and how such strategies may be formulated and evaluated, which, as stated earlier, is left for future work.

6.6 Validity and reliability

As the results and discussion in this chapter are based on the study presented in chapter 3, the validity and reliability considerations presented in section 3.4 apply to these results, too.

6.6.1 Reliability

The reliability of the method applied is considered fairly high, as it concerns primarily the coding of data types, generation triggers, and persistence locations of data mentioned in the interviews. The analysis applied consisted only of counting occurrences.

6.6.2 Validity

We consider the same validity measures as presented in section 4.4.2. The coding of the interviews was done using a legend corresponding exactly to the

taxonomy we wish to evaluate, which supports face validity. Whether or not the taxonomy supports its intended purpose is the subject of evaluation: our results are positive, that it was possible to categorize data in web applications according to it. We found that the results matched with what we would expect to find in MVC web applications, which further supports face validity.

The results presented in this chapter are mainly of academic character, and have little impact outside of academia, other than sketching a variety of options for web application developers. Further research into and formalization of the concepts of refinement functions and response generation strategies may prove to yield more useful results, as they would inform concrete design of web applications. The social validity of the current results is poor.

The taxonomy is new and experimental, and has not been applied to interview data from interviews concerning the subject matter of the taxonomy. As such, we have no reason to believe that the taxonomy in its current form is exhaustive, and it may be expanded in the future, following further studies. The content validity of the method is poor.

There is no prior research applying this or similar taxonomies, and thus we cannot evaluate the support of existing research.

6.7 Summary of findings

We presented an argument that it is infeasible to build strongly consistent web applications that scale to any significant number of clients.

We introduced a taxonomy of data in web applications, which allows for parallels to be drawn between web applications and database research in materialized views and view maintenance; and we proposed further work into refinement functions and response generation strategies as theoretical constructs that may inform the way we build web applications.

We applied the taxonomy, and found decent support for its use (albeit limited validity due to the mismatch in direction of interviews with the taxonomy), as well as showing the existence of some different approaches to both data type persistence and data generation triggers. Ultimately, this existence leads us to desire further work in applying and evaluating more approaches outside of those employed in simple MVC web applications.

Chapter 7

Conclusion

In this thesis, we set out to investigate and challenge perspectives of web applications. We explored the breadth and evolution of web application architectures, and argued that no change classification framework can be seen as ultimate, but should rather be promoted on merits of the cases where it provides useful insight. We presented a new perspective on architectural change based on architecture evolution paths, and a method for investigating concrete architectures through this perspective, documenting common narratives in six architectures' evolution paths. We also explored data as it exists in web applications, and found limited experimentation with persistence and generation triggers of data. Finally, we found that architecture-centric tools were in little use in the companies we studied, which is supported by previous research in the field.

We identified two new patterns of organizing web applications servers, parallel and chained web applications (in which two web applications use the same data store, and in which one web application uses another as data store, respectively), and several hybrid architectures, somewhere between traditional web application architectures and more complex orthogonal architectures, as well as long-lived efforts to slowly replace parts of the system, with an acceptance of added complexity at times.

Taken together with the variety of data generation methods (caching data on clients, generating refined data on change, storing events, storing refined data, caching responses), this paints a picture of an extremely complex set of practices, and the lack of use of architecture-centric tools shows that there is little help to find for practicioners, in the forms of useful perspectives or guidance.

With the inherently limited perspective we found in any change classification framework (which may be generalized to any classification framework), the complexity of web application development is decidedly poorly supported. We provided some new perspectives that may be found useful in the understanding of web applications, but nothing that has yet been shown to improve practicioners' understanding of the work they are performing.

As we explored the properties of web applications, we argued that they are inherently weakly consistent systems. This trait seems to be previously unrec-

ognized in web application research and practice, and may lead to improvements in web application design. Rather than doing nothing about weak consistency, we can now move towards building web applications with consistency properties more well-considered, such as considering different models for achieving eventual consistency between client and server than the one currently employed by most web applications.

We found parallels between database research and the research and practice of development of web applications, in particular that much of web application functionality is akin to manual view maintenance or view construction. By looking at web applications through this lens, we might in the future be able to describe web application functionality in a way that allows for semantic view maintenance, changing between maintenance strategies automatically, depending on load and constraints on the different data in the application.

The approach we used to identify architecture evolution paths was novel and experimental, and observations for future improvements were part of the results we reported. Additionally, the sample size of the study was very limited ($n = 6$) and the sample was poorly geographically distributed, so the results are poorly generalizable. The data we have presented provide an indication, but by no means conclusive evidence of any common patterns; rather we have strove to provide novel perspectives on web applications, leading to new directions in research.

We have taken the first steps down this path, introducing informally the notion of refinement functions—functions that aggregate one or more types of data into a more refined type of data—and the term response generation strategies—the consideration of how, when and where all the underlying refinement layers of a response are generated and persisted. Future work would involve further formalizations of these two terms, as well as exploration of how they may be used to inform web application design, both in theory and in practice. A framework for constructing and comparing response generation strategies and their impact on performance and consistency in theory would be an obvious next step, and practical applications of the ideas of refinement functions nad response generation strategies would follow that.

Bibliography

[1] Abadi, D. J. (2012). Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer, (2)*, (pp. 37-42).

[2] Abbott, A. (1992). What do cases do? Some notes on activity in sociological analysis. In Ragin, C. C. & Becker, H. S. (eds.), *What is a Case* (pp. 53-82). Cambridge University Press.

[3] Adzic, G., & Chatley, R. (2017, August). Serverless computing: economic and architectural impact. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (pp. 884-889). ACM.

[4] Alvarez, R. (2001, June). There's more to the story: using narrative analysis to examine requirements engineering as a social process. In *Proceedings of the 7th International Workshop on Requirements Engineering: Foundations for software quality* (pp. 4-5).

[5] Alvarez, R., & Urla, J. (2002). Tell me a good story: using narrative analysis to examine information requirements interviews during an ERP implementation. *ACM SIGMIS Database: the DATABASE for Advances in Information Systems*, 33(1), (pp. 38-52). ACM.

[6] Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing. *Communications of the ACM*, 53(4), (pp. 50-58). ACM.

[7] Barnes, J. M., Garlan, D., & Schmerl, B. (2014). Evolution styles: foundations and models for software architecture evolution. *Software & Systems Modeling*, 13(2) (pp. 649-678). Springer.

[8] Bartis, E., & Mitev, N. (2008). A multiple narrative approach to information systems failure: a successful system that failed. *European Journal of Information Systems*, 17(2) (pp. 112-124).

[9] Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. 2nd edition. Addison.

[10] Berners-Lee, T., Cailliau, R., Groff, J. F., & Pollermann, B. (1992). World-wide web: The information universe. *Internet Research*, 2(1), (pp. 52-58). Emerald Publishing.

[11] Berners-Lee, T.J., R. Cailliau & J.-F. Groff (1992). The world-wide web. *Computer Networks and ISDN Systems 25* (pp. 454-459). Elsevier.

[12] Biørn-Hansen, A., Majchrzak, T. A., & Grønli, T. M. (2017, April). Progressive web apps: The possible web-native unifier for mobile development. In *Proceedings of the 13th International Conference on Web Information Systems and Technologies (WEBIST)* (pp. 344-351). SCITEPRESS.

[13] Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., & Tan, W. G. (2001). Types of software evolution and software maintenance. *Journal of Software: Evolution and Process*, 13(1), (pp. 3-30). Wiley.

[14] Ben Christensen (2016). Don't Build a Distributed Monolith. `https://www.microservices.com/talks/dont-build-a-distributed-monolith/` (retrieved 2018-08-15)

[15] Clements, P., Garlan, D., Little, R., Nord, R., & Stafford, J. (2003, May). Documenting software architectures: views and beyond. In *Proceedings of the 25th International Conference on Software Engineering* (pp. 740-741). IEEE Computer Society.

[16] Conallen, J. (1999). Modeling Web application architectures with UML. *Communications of the ACM, 42*(10), (pp. 63-70). ACM.

[17] Devos, J., Van Landeghem, H., & Deschoolmeester, D. (2013, June). Narratives of an Outsourced Information Systems Failure in a Small Enterprise. In *International Working Conference on Transfer and Diffusion of IT* (pp. 57-72). Springer.

[18] Drevin, L. (2008). Making sense of information systems failures by using narrative analysis methods. In *Combined Proceeding of the 13th and 14th CPTS Working Conference.*

[19] Drevin, L., & Dalcher, D. (2011). Narrative methods: Success and failure stories as told by information system users. *2011 Proceedings 20 Years of Storytelling and sc'MOI: A Celebration, 69.* Philadelphia: sc'MOI.

[20] Dubé, L., & Robey, D. (1999). Software stories: three cultural perspectives on the organizational practices of software development. *Accounting, Management and Information Technologies*, 9(4), 223-259. Elsevier.

[21] Eugster, P. T., Felber, P. A., Guerraoui, R., & Kermarrec, A. M. (2003). The many faces of publish/subscribe. *ACM computing surveys* (CSUR), 35(2) (pp. 114-131). ACM.

[22] Evans, E. (2004). *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional.

[23] Fielding, R. T., & Taylor, R. N. (2002). Principled design of the modern Web architecture. *ACM Transactions on Internet Technology (TOIT)*, 2(2), (pp. 115-150). ACM.

[24] Fowler, M. (2004). Strangler Application. `https://www.martinfowler.com/bliki/StranglerApplication.html` (retrieved 2018-08-29)

[25] Fowler, M. (2005). Event Sourcing. `https://martinfowler.com/eaaDev/EventSourcing.html` (retrieved 2018-08-28)

[26] Fowler, M. (2011). CQRS. `https://martinfowler.com/bliki/CQRS.html` (retrieved 2018-09-01)

[27] Fowler, M. (2014). Microservices. `https://martinfowler.com/articles/microservices.html` (retrieved 2018-08-28)

[28] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., & Gauthier, P. (1997, October). Cluster-based scalable network services. In *ACM SIGOPS operating systems review* (Vol. 31, No. 5, pp. 78-91). ACM.

[29] Fox, A., & Brewer, E. A. (1999). Harvest, yield, and scalable tolerant systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems, 1999* (pp. 174-178). IEEE.

[30] Di Francesco, P., Lago, P., & Malavolta, I. (2018) Migrating towards Microservice Architectures: an Industrial Survey. *15th International Conference on Software Architecture, Proceedings of the*.

[31] Franklin, M. J., Carey, M. J., & Livny, M. (1993, August). Local disk caching for client-server database systems. In *Proceedings of the 19th International Conference on Very Large Data Bases* (pp. 641-655). Morgan Kaufmann Publishers Inc.

[32] Franzosi, R. (1998). Narrative analysis—or why (and how) sociologists should be interested in narrative. *Annual review of sociology*, 24(1), (pp. 517-554). Annual Reviews.

[33] Garlan, D. (1995, February). What is style. In *Proceedings of Dagshtul Workshop on Software Architecture*. Saarbruecken, Germany.

[34] Garrett, J. J. (2005). Ajax: A new approach to web applications. `http://adaptivepath.org/ideas/ajax-new-approach-web-applications/` (retrieved 2018-08-26).

[35] Gellersen, H. W., & Gaedke, M. (1999). Object-oriented web application development. *IEEE Internet Computing, 3*(1) (pp. 60-68). IEEE.

[36] Ginige, A., & Murugesan, S. (2001). Web engineering: An introduction. *IEEE multimedia, 8*(1), 14-18. IEEE.

[37] Golab, W., Rahman, M. R., AuYoung, A., Keeton, K., & Li, X. S. (2014). Eventually consistent: Not what you were expecting?. *Queue, 12*(1), 30. ACM.

[38] Gorton, I. (2006). *Essential software architecture*. Springer Science & Business Media.

[39] Gupta, A., & Mumick, I. S. (1995). Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull., 18*(2), 3-18. IEEE.

[40] Helland, P. (2005, January). Data on the Outside Versus Data on the Inside. In *CIDR* (pp. 144-153). CIDR Conference.

[41] Hill, M. D. (1990). What is scalability? *ACM SIGARCH Computer Architecture News*, 18(4) (pp. 18-21). ACM.

[42] Iyengar, A., & Challenger, J. (1997). Improving web server performance by caching dynamic data. *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (pp. 49-60). USENIX.

[43] Jónsson, B. Þ., Arinbjarnar, M., Þórsson, B., Franklin, M. J., & Srivastava, D. (2006). Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology (TOIT), 6*(3), 302-331. ACM.

[44] K. Krippendorff (2013). *Content Analysis: An Introduction to Its Methodology*. 3rd ed. Sage.

[45] Krishnan, S., Wang, J., Franklin, M. J., Goldberg, K., & Kraska, T. (2015). Stale view cleaning: Getting fresh answers from stale materialized views. *Proceedings of the VLDB Endowment, 8(12)* (pp. 1370-1381). VLDB Endowment.

[46] Labov, W. (1972). *Language in the inner city: Studies in the Black English vernacular (Vol. 3)*. University of Pennsylvania Press.

[47] Leff, A., & Rayfield, J. T. (2001). Web-application development using the model/view/controller design pattern. In *Enterprise Distributed Object Computing Conference, 2001. EDOC'01. Proceedings. Fifth IEEE International* (pp. 118-127). IEEE.

[48] Lloyd, W., Freedman, M. J., Kaminsky, M., & Andersen, D. G. (2014). Don't settle for eventual consistency. *Communications of the ACM, 57*(5), 61-68. ACM.

[49] Marz, N., & Warren, J. (2015). *Big Data: Principles and best practices of scalable real-time data systems*. New York; Manning Publications Co.

[50] Mikowski, M., & Powell, J. (2013). *Single page web applications: JavaScript end-to-end*. Manning Publications Co.

[51] Obasi, C.K., Asagba, P.O., & Silas, A.I. (2015). A Comparative Study of Consistency Theorems in Distributed Databases. In *African Journal of Computing & ICT* (pp. 205-208). IEEE.

[52] O'Reilly, T. (2007). What is Web 2.0? In Donelan, H., Kear, K., & Ramage, M. (Eds.), *Online communication and collaboration: A reader* (pp. 225-235). Routledge.

[53] Osterlie, T., & Wang, A. I. (2006, September). Establishing Maintainability in Systems Integration: Ambiguity, Negotiations, and Infrastructure. In *Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on* (pp. 186-196). IEEE.

[54] Ozkaya, I., Wallin, P., & Axelsson, J. (2010). Architecture knowledge management during system evolution: observations from practitioners. In *Proceedings of the 2010 ICSE Workshop on Sharing and Reusing Architectural Knowledge* (pp. 52-59). ACM.

[55] Paixao, M., Krinke, J., Han, D., Ragkhitwetsagul, C., & Harman, M. (2017). Are developers aware of the architectural impact of their changes? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering* (pp. 95-105). IEEE Press.

[56] Pallis, G., & Vakali, A. (2006). Insight and perspectives for content delivery networks. *Communications of the ACM*, 49(1) (pp. 101-106). ACM.

[57] Papazoglou, M. P. (2003, December). Service-oriented computing: Concepts, characteristics and directions. In *Web Information Systems Engineering, 2003, Proceedings of the Fourth International Conference on* (pp. 3-12). IEEE.

[58] Riessman, C. K. (1993). *Narrative analysis*. Sage.

[59] Rinaldi, B. (2015). *Static site generators*. O'Reilly.

[60] Rodriguez, A. (2008). Restful web services: The basics. *IBM developerWorks*, 33. IBM Press.

[61] Shklar, L., & Rosen, R. (2009). *Web application architecture*. John Wiley & Sons.

[62] Swanson, E. B. (1976, October). The dimensions of maintenance. In *Proceedings of the 2nd international conference on Software engineering* (pp. 492-497). IEEE Computer Society Press.

[63] Thomas, D., & Hansson, D.H. (2005). *Agile web development with Rails*. The Pragmatic Programmers LLC.

[64] Toolan, M. (2001). *Narrative: A critical linguistic introduction*. 2nd edition. Routledge.

[65] Velázquez, F., Lyngstøl, K., Heen, T.F., & Renard, J. (2016). *The Varnish Book*. Varnish Software AS.

[66] Vogels, W. (2009). Eventually consistent. *Communications of the ACM*, 52(1), (pp. 40-44). ACM.

[67] Williams, B. J., & Carver, J. C. (2010). Characterizing software architecture changes: A systematic review. *Information and Software Technology, 52(1)*, (pp. 31-51). Elsevier.

[68] Amazon Web Services website. `https://aws.amazon.com/` (retrieved 2018-08-17)

[69] Event Store Documentation. `https://eventstore.org/docs/` (retrieved 2018-08-28)

[70] Express documentation. `http://expressjs.com/` (retrieved 2018-09-01)

[71] Google Cloud website. `https://cloud.google.com/` (retrieved 2018-08-17)

[72] Gun.js website. `https://gun.eco/` (retrieved 2018-09-01)

[73] ISO/IEC 14764:2006: Software Engineering – Software Life Cycle Processes – Maintenance. International Organization for Standardization. `https://www.iso.org/standard/39064.html` (retrieved 2018-09-02).

[74] JAMstack website. `https://jamstack.org/` (retrieved 2018-08-15)

[75] Mozilla Developer Network WebSocket API documentation. `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API` (retrieved 2018-08-28)

[76] Node.js language HTTP documentation. `https://nodejs.org/api/http.html` (retrieved 2018-09-01)

[77] Rust langauge HTTP documentation. `https://docs.rs/http/0.1.5/http/index.html` (retrieved 2018-09-01)

[78] WebTorrent website. `https://webtorrent.io/` (retrieved 2018-08-28)

Appendix A

Company email

The following email was sent to potential interviewees in order to inquire about their participation in the study. Both an English and Danish language email was prepared. All the participants that responded were contacted using the Danish language email, as their primary communication on their company website was in Danish.

A.1 English version

Subject: Research project on web applications

Dear {{company}},

I am writing to hear if you might be interested in participating in an research project at the IT University of Copenhagen. We are investigating how web application architectures change over time when responding to changing requirements.

We would like to interview one of your employees for around an hour, talking about your web platform, how it has changed over time, and how you see it changing in the future.

The employee should preferably (though not necessarily) have worked on the platform since its inception, as we will be investigating decisions made that changed the architecture of the platform.

Specifically, we are investigating layered web applications (such as, for example, applications originally built with an MVC framework such as ASP.NET MVC, Ruby on Rails, or similar). We think your platform might fit the bill.

In the interview, we are going to have a conversation about the platform. The conversation will be recorded, and paper and pen are also available, so the employee can make illustrations, which will also be recorded. The interview recordings will not be shared with people not involved in the research. The results will be presented only in anonymized form.

I would appreciate any help getting in touch with a relevant employee at your company. Let me know if you have any questions or any need for clarification.

Best,

Niels Roesen Abildgaard
MSc Student at the IT University of Copenhagen
nroe@itu.dk | 40 92 06 41

A.2 Danish version

Subject: Forskningsprojekt om webapplikationer

Kære {{firma}},

Jeg skriver til jer, for at høre om I potentielt ville være interesserede i at deltage i et forskningsprojekt fra IT-Universitetet i København. Vi undersøger hvordan web applikationers arkitektur ændrer sig over tid efterhånden som krav til dem ændrer sig.

Vi vil meget gerne interviewe en af jeres medarbejdere i omkring en time, hvor vi vil snakke om jeres webplatform, hvordan den har ændret sig over tid, og hvordan I forudser at den vil ændre sig i fremtiden.

Medarbejderen bør helst (men ikke nødvendigvis) have arbejdet med platformen siden dens oprettelse, da vi gerne vil forstå de beslutninger der første til ændringer i platformens arkitektur.

Mere specifikt undersøger vi layered web applications (som for eksempel applikationer bygget med MVC frameworks som ASP.NET MVC, Ruby on Rails, eller lignende). Vi tror jeres platform falder i denne kategori.

Interviewet tager form at en samtale om platformen. Samtalen bliver optaget, og papir og skriveredskaber stilles til rådighed, så medarbejderen kan illustrere mens vi taler. Illustrationerne bliver også gemt. Interviewoptagelserne vil ikke blive delt med folk udenfor forskningsprojektet, og resultaterne præsenteres i anonymiseret form.

Jeg ville sætte stor pris på at blive sat i kontakt med en relevant medarbejder. Sig til hvis I har behov for uddybning af noget af det jeg har skrevet, eller har spørgsmål.

Bedste hilsner,

Niels Roesen Abildgaard
MSc-studerende ved IT-Universitetet i København
nroe@itu.dk | 40 92 06 41

Appendix B

Interview protocol

The interviews for this study will be semi-structured interviews touching on a number of themes. They are expected to last around an hour. The interviews are recorded, and notes and illustrations made during the interviews are captured by camera. The interviewer supplies materials for making illustrations.

Interviewees are asked to bring architectural artifacts in their possession to the interview.

In the beginning of the interview, the following script is read to interviewees, informing them of the procedure. They will be asked for explicit consent to the conditions of these interview conditions. The text was also made available to the company when first establishing contact:

> This interview is a part of a study exploring changes to web application architectures. We are going to have a conversation about your product, and the conversation will be recorded. Paper and pen are also available, so you can make illustrations, which will also be recorded. The interview recordings will not be shared with people not involved in the research. The results will be presented only in anonymized form.

Or, if interviews or contact are conducted in Danish, the following script will be used:

> Dette interview er en del af et studie af ændringer i web applikationsarkitekturer. Vi kommer til at have en samtale om jeres produkt, og samtalen bliver optaget. Der er også papir og kuglepen til rådighed til at lave illustrationer, og disse bliver også bevaret. Det materiale vi får fra interviewet bliver ikke delt med folk, der ikke er involverede i forskningsprojektet, og resultaterne bliver præsenteret i anonymiseret form.

During the interview, the following themes will be discussed, in prioritized and chronological order (meaning that if time runs out during an interview, the first themes will have been covered, but some later ones may not have been):

- The product's profile (when development started, date of first launch, the team then, the team now, number of users, interaction pattern of users)
- The interviewee's profile (their role in the company, their role while working on the product, their interaction with software architecture as a tool, how software architecture is used in the company)
- The product today (the significant components of architecture)
- The initial architecture of the product (the considerations when starting to develop the product, the significant components)
- Changes along the way (gradual changes, major changes, the causes of said changes, the effects of the changes to the architecture, and potential evaluations of the changes)
  The goal is to get an idea of what changes happened, as well as the order they happened in.
  If the interviewee indicates knowledge of early changes, these should be the focus. Otherwise, focus should be on the changes that are fresh in memory (recent, impactful).
- The interviewee's approach to scaling applications today (how would they go about responding to changing scalability requirements in their product as it first looked, if they were to do it today)
  To avoid leading questions, this could be investigated by asking the interviewee to evaluate the change, or if they received feedback on the change.

The interviews are expected to last an hour, depending on the availabilities of the employees in the companies.

Considerations regarding the interview protocol

After the first interview it became clear that asking the interviewee to describe their position and work in the company automatically led to technical descriptions of the product. To ensure getting basic information about the product, the order of the first two topics was swapped, such that the product profile is the first topic.

Appendix C

Legend for coding interviews

C.1 Architecture, patterns, and change

The following legend was used for coding of architectural direction and tool use, architecture patterns, and reasons and manner of architectural changes found in the interview transcripts.

During the coding, it was noted on the interview transcript every time an interviewee mentioned one of the terms or concepts in the coding legend. This allowed for a later lookup of e.g. what architectural practices a company had, which patterns were in use in their architecture, or which types of data they had in use (with each type characterized by type, generation trigger and persistence location).

T Architecture as a tool in the organization

1. Used informally, verbally
2. Used informally, verbally, with diagrams for external communication
3. Used informally, verbally and diagrams
4. Not used

AG Goals and direction for architecture

1. By target architecture
2. By principles
3. No long-term direction

AP Patterns in architectures

1. Bounded contexts
2. Strangler pattern: long-lived, slow replacement.
3. Layered web application

   (a) Shared-Nothing, as part of a larger system
   (b) Shared-Nothing, as the only part of the system
   (c) Stateful, as part of a larger system
   (d) Stateful, as the only part of the system
4. Microservices
   (a) True microservices
   (b) False microservices: large feature set
   (c) False microservices: not individual storage
   (d) Machine learning as microservice
5. Infrastructure or platform as a service
6. Parallel layered web applications
7. Chained layered web applications
8. Shared dependencies across services (distributed monolith)
9. Services in different programming languages/on different platforms
10. Serverless functions
11. ~~Monolith/layered web application~~[1]
12. Communication via message bus
13. Shared database (multi-tenant) (across services)
14. CQRS
15. Strangler completed
16. Replication of components
   (a) Load balancing
   (b) Geographic distribution
   (c) Failsafe
   (d) Other
17. Static site/JAMstack

ACR Architectural change: reasons for change

1. Migrating when infrastructure automation is ready
2. New feature requirement
3. Poor performance under current conditions
4. Need for future scalability
5. Developer experience improvement
6. Reduce infrastructure costs
7. Unknown

[1]This item, AP.11, was originally used in coding, but later merged with AP.3, and not used in the final coding. It is included here to keep the enumeration intact.

8. Industry popularity/"the right solution"
9. Eliminate error sources
10. Escape vendor lock-in
11. Adhere to bounding context

ACM Architectural change: manner of change

1. Extraction from existing part
   (a) Inside bounded context borders
   (b) Across bounded context borders
   (c) Without use of bounded contexts
   (d) Extract database to separate server
   (e) Establishing bounded contexts
2. New part from scratch
3. Removed part
4. Integration of parts into a single part.
5. Rewrote/replaced existing part
6. Moved to infrastructure as a service
7. Introduce replication of component
   (a) Load balancing
   (b) Other
8. Rewrote entire architecture
9. Change in communication
   (a) towards request/response
   (b) towards message-driven
10. Virtual/conceptual/theoretical architectural change

C.2 Data types in web applications

The coding legend for identifying data types in web applications overlaps the taxonomy presented in 6. During the coding, data types that were not originally present in the taxonomy, but observed in the interview transcripts, were added to the legend.

DT Data type

1. Request data
2. Raw data (event)
3. Aggregated data
4. Response data

DG Data generation time

1. On request for it
2. Cached
3. On change
4. At intervals
5. When generation is requested

DL Data location

1. In application
2. In ~~main~~ database
3. ~~In other database~~[2]
4. In cache-specific storage
5. On client (in browser/app)

[2]This category, DL.3, was originally coded for, but was merged with DL.2 (which was relabelled "In database"), as the distinction became impossible to make in systems with several databases, and no primary database.

Appendix D

Architecture evolution paths

This appendix contains the generic/abstracted version of the architecture evolution paths we identified from the interviews we conducted. Each case is listed in its own section, and the steps are listed in chronological order.

If there are significant notes about time (e.g. we know which year a step happened in, or the step was detached in time) it is underlined and noted first in the step. After that, in italics, is the description of what happened in the step, then, in bold, the codings relating to the architectural change that occurred, then a description of the full architecture at the time in question, and finally, in bold, the architectural patterns coded for that are in use in the architecture after the step.

In the case of more complex steps, where several smaller changes happened at the same time, these inner steps are listed in a nested list, with a description of the change performed in the step in italics and the relevant codings in bold.

For detached steps, we cannot know how the full architecture looks after their application, as we do not know when this change happened, so it has been left as a question mark.

D.1 Case a.

1. *First version*
   Unreplicated monolith layered web application. Single db. One server.
   AP.3.b

2. *Rewrite*
   ACM.8 ACR.7
   Unreplicated monolith layered web application. Single db. One server.
   AP.3.b

3. *Extract db to separate machine*
   ACM.1.d ACR.7 C-4
   Unreplicated monolith layered web application. Single db. Separate db

server.
AP.3.b

4. *Add new web application with new functionality, depending on same data-model*
   ACM.2 ACR.2
   Unreplicated parallel layered web applications. Single db. Separate db server. Shared dependencies across services.
   AP.3.a AP.6 AP.8 AP.13

5. 2015
   (a) *Move infrastructure to cloud*
   ACR.3 ACM.6
   (b) *Introduce load balancing*
   ACR.3 ACM.7.a

   Load-balanced parallel layered web applications hosted in cloud. Single db. Shared dependencies across services.
   AP.3.a AP.5 AP.6 AP.8 AP.13

6. *Introduce scheduled jobs*
   ACM.2 ACR.5 C-4
   Load-balanced parallel layered web applications hosted in cloud. Single db. Shared dependencies across services. Scheduled jobs.
   AP.3.a AP.5 AP.6 AP.8 AP.13

7. *Introduce message bus*
   ACM.2 ACM.9.b ACR.7
   Load-balanced parallel layered web applications hosted in cloud. Single db. Shared dependencies across services. Scheduled jobs.
   AP.3.a AP.5 AP.6 AP.8 AP.12 AP.13

8. 2017
   Start strangler replacing message bus
   ACM.5 ACR.9 C-4
   Load-balanced parallel layered web applications hosted in cloud. Single db. Shared dependencies across services. Scheduled jobs. Strangling old message bus.
   AP.2 AP.3.a AP.5 AP.6 AP.8 AP.12 AP.13

9. *Introduce microservices*
   ACR.2 ACR.5 ACR.8 ACM.2 ACM.7.a
   Load-balanced parallel layered web applications hosted in cloud. Single db. Shared dependencies across services. Scheduled jobs. Strangling old message bus. Microservices.
   AP.2 AP.3.a AP.4.a AP.4.d AP.5 AP.6 AP.8 AP.9 AP.12 AP.13

10. Not placed in time
    Introduced bounded contexts
    ACM.10 ACR.7
    ?
    AP.1

11. Not placed in time
    Introduced materialized views, CQRS
    ACM.1.e ACR.3
    ?
    AP.14

D.2 Case b.

1. 2015
   First version
   Unreplicated monolith layered web application; landing page as separate service, hosted in cloud. 8 databases.
   AP.3.a AP.5

2. 2017
   Merge databases
   ACM.4 ACR.6 ACR.5
   Unreplicated monolith layered web application; landing page as separate service, hosted in cloud. Single database.
   AP.3.a AP.5

3. *Introduced microservice*
   ACM.2 ACR.2
   Unreplicated monolith layered web application; landing page as separate service, hosted in cloud. Single database. Microservices.
   AP.3.a AP.4.a AP.4.d AP.5

4. Future
   (a) *Rewrite a component*
   ACM.5 ACR.4
   (b) *Move component to managed infrastructure*
   ACM.6 ACR.4

   Unreplicated monolith layered web application; landing page as separate service, hosted in cloud. Single database. Microservices. Parts have managed infrastructure.
   AP.3.a AP.4.a AP.4.d AP.5

5. Future
   Rewrite a component
   ACM.5 ACR.8

Unreplicated monolith layered web application; landing page as separate service, hosted in cloud. Single database. Microservices. Parts have managed infrastructure.
AP.3.a AP.4.a AP.4.d AP.5

6. Future
   Move database to managed infrastructure
   ACM.6 ACR.7
   Unreplicated monolith layered web application; landing page as separate service, hosted in cloud. Single database. Microservices. Parts have managed infrastructure.
   AP.3.a AP.4.a AP.4.d AP.5

7. Future
   Merge two web-facing components
   ACM.4 ACR.8
   Unreplicated monolith layered web application hosted in cloud. Single database. Microservices. Parts have managed infrastructure.
   AP.3.a AP.4.a AP.4.d AP.5

D.3 Case c.

1. 2004
   First version
   Unreplicated monolith layered web application (not shared-nothing). Single db.
   AP.3.d

2. 2008
   Rewrite
   ACM.8 ACR.7
   Unreplicated monolith layered web application (not shared-nothing). Single db.
   AP.3.d

3. 2009
   Introduce scheduled jobs
   ACM.2 ACR.2
   Unreplicated monolith layered web application (not shared-nothing). Single db. Scheduled jobs.
   AP.3.d

4. 2012
   Extract service from a scheduled job
   ACM.1.d ACR.3

Unreplicated monolith layered web application (not shared-nothing). Single db. Scheduled jobs. Microservices.
AP.3.d AP.4.c

5. 2012-2013
   Add mobile apps (new clients)
   ACM.2 ACR.2
   Unreplicated monolith layered web application (not shared-nothing). Single db. Scheduled jobs. Microservices. App clients.
   AP.3.d AP.4.c

6. 2014
   Add new web application strangling old service, depending on same database, some shared datamodels
   ACM.2 ACR.2
   Unreplicated parallel layered web application (one shared-nothing), one service strangling the other. Single db. Scheduled jobs. Microservices. App clients. Shared dependencies across services.
   AP.2 AP.3.a AP.3.c AP.4.c AP.6 AP.8 AP.13

7. *Introduce load balancing for one of the layered web applications*
   ACM.2 ACM.7.a ACR.3 ACR.4
   Partially load-balanced parallel layered web application (one shared nothing), one service strangling the other. Single db. Scheduled jobs. Microservices. App clients. Shared dependencies across services.
   AP.2 AP.3.a AP.3.c AP.4.c AP.6 AP.8 AP.13 AP.16.a

8. (a) *Introduce message bus*
      ACM.2 ACM.9.b ACR.3 ACR.4
   (b) *Extract services from scheduled jobs*
      ACM.1 ACR.3 ACR.4
   (c) *Move some services to cloud*
      ACM.6 ACR.3 ACR.4

   Partially load-balanced parallel layered web application, one service strangling the other. Single db. Scheduled jobs. Microservices, partially in managed hosting. Message bus. App clients. Shared dependencies across services.
   AP.2 AP.3.a AP.3.c AP.4.c AP.5 AP.6 AP.8 AP.12 AP.13 AP.16.a

9. *Geographically distribute database (shard)*
   ACR.3 ACR.6 ACM.1.b ACM.7.b
   Partially load-balanced parallel layered web application, one service strangling the other. Geographically distributed db. Scheduled jobs. Microservices, partially in managed hosting. Message bus. App clients. Shared dependencies across services.
   AP.2 AP.3.a AP.3.c AP.4.c AP.5 AP.6 AP.8 AP.12 AP.13 AP.16.a AP.16.b

10. 2016
    Introduce load-balancing for rest of the layered web applications
    ACM.2 ACM.7.a ACR.3
    Load-balanced parallel layered web application, one service strangling the other. Geographically distributed db. Scheduled jobs. Microservices, partially in managed hosting. Message bus. App clients. Shared dependencies across services.
    AP.2 AP.3.a AP.4.c AP.5 AP.6 AP.8 AP.12 AP.13 AP.16.a AP.16.b

11. 2017
    Introducing new style of microservices
    ACM.2 ACR.2 ACR.3
    Load-balanced parallel layered web application, one service strangling the other. Geographically distributed db. Scheduled jobs. Microservices, partially in managed hosting. Message bus. App clients. Shared dependencies across services.
    AP.2 AP.3.a AP.4.a AP.4.c AP.5 AP.6 AP.8 AP.12 AP.13 AP.16.a AP.16.b

12. Future
    Extract service from layered web application
    ACM.1.a ACR.3 ACR.9
    Load-balanced parallel layered web application, one service strangling the other. Geographically distributed db. Scheduled jobs. Microservices, partially in managed hosting. Message bus. App clients. Shared dependencies across services.
    AP.2 AP.3.a AP.4.a AP.4.b AP.4.c AP.5 AP.6 AP.8 AP.12 AP.13 AP.16.a AP.16.b

13. Future
    Move databases to cloud
    ACM.6 ACR.3
    Load-balanced parallel layered web application, one service strangling the other. Geographically distributed db in managed hosting. Scheduled jobs. Microservices, partially in managed hosting. Message bus. App clients. Shared dependencies across services.
    AP.2 AP.3.a AP.4.a AP.4.b AP.4.c AP.5 AP.6 AP.8 AP.12 AP.13 AP.16.a AP.16.b

14. Not placed in time
    Introduced bounded contexts
    ACR.8 ACR.5 ACM.1.e
    ?
    AP.1

D.4 Case d.

1. *First version*
   Unreplicated monolith layered web application. Single db.
   AP.3.b

2. (a) *Start strangler rewrite of entire application*
      ACM.5 ACR.2 ACR.5 ACR.9
   (b) *Introduce work queues*
      ACM.2 ACR.2 ACR.5

   Unreplicated monolith layered web application. Single db. Strangling old system. Work queues.
   AP.2 AP.3.b AP.14

3. *Finished strangler rewriting application*
   ACM.5 ACR.2 ACR.5 ACR.9
   Unreplicated monolith layered web application. Single db. Work queues.
   AP.3.b AP.14 AP.15

4. *Move work queues to different data store*
   ACM.2 ACR.3
   Unreplicated monolith layered web application. Work queues.
   AP.3.b AP.14 AP.15

D.5 Case e.

1. 2013
   First version
   Load-balanced monolith layered web application. Separate db with backup. Cloud hosted.
   AP.3.b AP.5 AP.16.a AP.16.c

2. 2016
   (a) *Rewrite*
      ACR.3 ACR.5 ACR.10 ACM.8
   (b) *Move to different cloud*
      ACR.3 ACR.5 ACR.10 ACM.6

   Load-balanced chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site.
   AP.3.a AP.5 AP.7 AP.9 AP.16.a AP.17

3. *Add new web application with new functionality*
   ACR.2 ACM.2
   Load-balanced parallel and chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site.
   AP.3.a AP.5 AP.6 AP.7 AP.9 AP.16.a AP.17

4. *Introducing serverless functions*
   ACR.2 ACM.2
   Load-balanced parallel and chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site. Serverless functions.
   AP.3.a AP.5 AP.6 AP.7 AP.9 AP.10 AP.16.a AP.17

5. *Introducing a microservice*
   ACM.2 ACR.2
   Load-balanced parallel and chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site. Microservice. Serverless functions.
   AP.3.a AP.4.a AP.4.d AP.5 AP.6 AP.7 AP.9 AP.10 AP.16.a AP.17

6. *Introducing message queue*
   ACM.9.b ACR.7
   Load-balanced parallel and chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site. Microservice. Message queue. Serverless functions.
   AP.3.a AP.4.a AP.4.d AP.5 AP.6 AP.7 AP.9 AP.10 AP.12 AP.16.a AP.17

7. Future
   Geographic distribution of architecture
   ACM.7.b ACR.3
   Geographically distributed, load-balanced parallel and chained layered web applications in managed hosting. Dbs in managed hosting. Cloud hosted. Static site. Microservice. Message queue. Serverless functions.
   AP.3.a AP.4.a AP.4.d AP.5 AP.6 AP.7 AP.9 AP.10 AP.12 AP.16.a AP.16.b AP.17

D.6 Case f.

1. *First version*
   Load-balanced chained layered web applications. Single, shared db, and data model. Scheduled jobs. Cloud hosted.
   AP.3.a AP.5 AP.7 AP.8 AP.13 AP.16.a

2. *Caching layer for some database data*
   ACR.4 ACM.2
   Load-balanced chained layered web applications. Single, shared db, and data model. Scheduled jobs. Cloud hosted.
   AP.3.a AP.5 AP.7 AP.8 AP.13 AP.16.a

3. *New web application with new functionality*
   ACR.2 ACM.2
   Load-balanced parallel and chained layered web applications. Single, shared

db, and data model. Scheduled jobs. Cloud hosted.
AP.3.a AP.5 AP.6 AP.7 AP.8 AP.13 AP.16.a

4. 2012
   Merge scheduled jobs functionality into web application
   ACM.4 ACR.7
   Load-balanced parallel and chained layered web applications. Single, shared db, and data model. Cloud hosted.
   AP.3.a AP.5 AP.6 AP.7 AP.8 AP.13 AP.16.a

5. *Introducing scheduled jobs*
   ACM.2 ACR.2
   Load-balanced parallel and chained layered web applications. Single, shared db, and data model. Scheduled jobs. Cloud hosted.
   AP.3.a AP.5 AP.6 AP.7 AP.8 AP.13 AP.16.a

6. 2013
   Introducing new database for part of the system
   ACM.2 ACR.3
   Load-balanced parallel and chained layered web applications. Shared data models. Two databases, shared. Scheduled jobs. Cloud hosted.
   AP.3.a AP.5 AP.6 AP.7 AP.8 AP.13 AP.16.a

7. 2013
   Start strangler rewrite of one layered web application
   ACM.5 ACR.7
   Load-balanced parallel and chained layered web applications. Strangling one web application. Shared data models. Two databases, shared. Scheduled jobs. Cloud hosted.
   AP.2 AP.3.a AP.5 AP.6 AP.7 AP.8 AP.13 AP.16.a

8. *Introducing microservices*
   ACM.2 ACR.2
   Load-balanced parallel and chained layered web applications. Strangling one web application. Shared data models. Two databases, shared. Scheduled jobs. Microservices. Cloud hosted. Bounded contexts.
   AP.1 AP.2 AP.3.a AP.4.a AP.4.c AP.5 AP.6 AP.7 AP.8 AP.13 AP.16.a

9. 2016
   Introduced serverless functions
   ACM.2 ACR.1 ACR.2 ACR.9
   Load-balanced parallel and chained layered web applications. Strangling one web application. Shared data models. Two databases, shared. Scheduled jobs. Microservices. Mixed langauge serverless functions. Cloud hosted. Bounded contexts.
   AP.1 AP.2 AP.3.a AP.4.a AP.4.c AP.5 AP.6 AP.7 AP.8 AP.9 AP.10 AP.13 AP.16.a

10. 2017
    Finished strangler rewrite of one layered web application
    ACM.5 ACR.7
    Load-balanced parallel and chained layered web applications. Shared data models. Two databases, shared. Scheduled jobs. Microservices. Mixed langauge serverless functions. Cloud hosted. Bounded contexts.
    AP.1 AP.3.a AP.4.a AP.4.c AP.5 AP.6 AP.7 AP.8 AP.9 AP.10 AP.13 AP.15 AP.16.a

11. Future
    Migrate one database to another technology
    ACM.5 ACR.5 ACR.8
    Load-balanced parallel and chained layered web applications. Shared data models. Two databases, shared. Scheduled jobs. Microservices. Mixed langauge serverless functions. Cloud hosted. Bounded contexts.
    AP.1 AP.3.a AP.4.a AP.4.c AP.5 AP.6 AP.7 AP.8 AP.9 AP.10 AP.13 AP.15 AP.16.a

12. Future
    Move responsibility for something to a better fitting bounding context, into new microservice
    ACM.1.b ACR.11
    Load-balanced parallel and chained layered web applications. Shared data models. Two databases, shared. Scheduled jobs. Microservices. Mixed langauge serverless functions. Cloud hosted. Bounded contexts.
    AP.1 AP.3.a AP.4.a AP.4.c AP.5 AP.6 AP.7 AP.8 AP.9 AP.10 AP.13 AP.15 AP.16.a

13. Future
    Migrate services to more managed hosting
    ACM.6 ACR.4 ACR.5
    Load-balanced parallel and chained layered web applications in managed hosting. Shared data models. Two databases, shared. Scheduled jobs. Microservices. Mixed langauge serverless functions. Cloud hosted. Boudned contexts.
    AP.1 AP.3.a AP.4.a AP.4.c AP.5 AP.6 AP.7 AP.8 AP.9 AP.10 AP.13 AP.15 AP.16.a

14. Not placed in time
    Introduced bounded contexts
    ACM.10 ACR.7
    ?
    AP.1

15. Not placed in time
    Added message queue
    ACM.2 ACR.7

?
AP.12

16. Not placed in time
Extracted service to remove shared dependency in applications/to adhere to bounded contexts
ACM.1.b ACR.5
?

17. Not placed in time
Distributed layered web applications geographically, and as a failsafe
ACM.7.b ACR.7
?
AP.16.b AP.16.c

Appendix E

Narrative finder program

The following program was used to find common narratives in architecture evolution paths. First, the architecture evolution paths were represented in a simple Javascript Object-format, where each step noted equal and similar steps. An example of this can be seen in listing E.1.

We load up the paths and use our program to construct a similarity table. We prime the paths, which in this case means representing each step in each path by the name of the alphanumerically first step that it overlaps with (either equal or similar). This is then used to find the different types of common narratives, which are saved as a JSON-file. The main program can be seen in listing E.2. The code for building a similarity table is in listing E.4, the code for priming paths is in listing E.5, the code for finding sections is in listing E.6, stories in listing E.7, and group precedences in listing E.8

```
module.exports = [
    { id: 1 },
    { id: 2 },
    { id: 3 },
    { id: 4 },
    { id: 5 },
    { id: 6 },
    { id: 7 },
    { id: 8 },
    { id: 9 }
];
```

Listing E.1: a.js — an example of a path encoded as Javascript Objects.

```
const paths = require("./paths");
const narrativeFinder = require("./index");
const fs = require("fs");

```

```
//Setup
let similarityTable = narrativeFinder.buildSimilarityTable(paths);
let equalPrimedPaths = narrativeFinder.primePaths(paths, similarityTable,
                                                                    "equal");
let similarPrimedPaths = narrativeFinder.primePaths(paths, similarityTable,
                                                                    "similar");

//Find sections
let equalSections = narrativeFinder.findSections(equalPrimedPaths);
let similarSections = narrativeFinder.findSections(similarPrimedPaths);

//Find stories
let equalStories = narrativeFinder.findStories(equalPrimedPaths);
let similarStories = narrativeFinder.findStories(similarPrimedPaths);

//Find group precedences
let equalGroupPrecedences = narrativeFinder
                                    .findGroupPrecedences(equalPrimedPaths);
let similarGroupPrecedences = narrativeFinder
                                    .findGroupPrecedences(similarPrimedPaths);

//Write result to file
fs.writeFile("narrative-finder-result.json", JSON.stringify({
    sections: {
        equal: equalSections,
        similar: similarSections
    },
    stories: {
        equal: equalStories,
        similar: similarStories
    },
    groupPrecedences: {
        equal: equalGroupPrecedences,
        similar: similarGroupPrecedences
    }
}), (error) => {
    if(error) {
        return console.error("Failed␣to␣save␣result", error);
    }
    console.log("Saved␣result.");
});
```

Listing E.2: run.js — the main program that finds narratives in the given data

```
module.exports = {
    buildSimilarityTable: require("./src/buildSimilarityTable"),
    primePaths: require("./src/primePaths"),
    findSections: require("./src/findSections"),
    findNarratives: require("./src/findNarratives"),
```

```
    findGroupPrecedences: require("./src/findGroupPrecedences")
};
```

Listing E.3: index.js — exposes the `narrativeFinder` object.

```
module.exports = buildSimilarityTable;

function buildSimilarityTable(paths) {
    let result = {};

    let pathNames = Object.keys(paths);

    pathNames.forEach((pathName) => {
        paths[pathName].forEach((step) => {
            let fullStepName = `${pathName}.${step.name}`;
            result[fullStepName] = [];

            pathNames.forEach((otherPathName) => {
                paths[otherPathName].forEach((otherStep) => {
                    if(stepsAreSimilar(pathName, step, otherPathName,
                                                            otherStep)) {
                        result[fullStepName].push({
                            type: "similar",
                            step: `${otherPathName}.${otherStep.name}`
                        });
                    }
                    if(stepsAreEqual(pathName, step, otherPathName,
                                                            otherStep)) {
                        result[fullStepName].push({
                            type: "equal",
                            step: `${otherPathName}.${otherStep.name}`
                        });
                    }
                });
            });
        });
    });

    let stepNames = Object.keys(result);

    stepNames.forEach((stepName) => {
        result[stepName].forEach((similarity, i1) => {
            result[stepName].forEach((otherSimilarity, i2) => {
                if(i1 == i2) return;
                ensureSimilarity(result, similarity, otherSimilarity);
            });
        });
    });

```

```
    stepNames.forEach((stepName) => {
        let similarities = result[stepName];

        let similarPrime = findPrime(similarities, stepName);
        let equalPrime = findPrime(
                            similarities.filter((s) => s.type == "equal"),
                            stepName);

        result[stepName] = { similarPrime, equalPrime, similarities };
    });

    return result;
}

function stepsAreSimilar(pathName, step, otherPathName, otherStep) {
    if(step.similarTo == `${otherPathName}.${otherStep.name}`) {
        return true;
    }
    if(otherStep.similarTo == `${pathName}.${step.name}`) {
        return true;
    }
    if(pathName === otherPathName) {
        if(step.similarTo == `~.${otherStep.name}`) {
            return true;
        }
        if(otherStep.similarTo == `~.${step.name}`) {
            return true;
        }
    }
    return false;
}

function stepsAreEqual(pathName, step, otherPathName, otherStep) {
    if(step.equalTo == `${otherPathName}.${otherStep.name}`) {
        return true;
    }
    if(otherStep.equalTo == `${pathName}.${step.name}`) {
        return true;
    }
    if(pathName === otherPathName) {
        if(step.equalTo == `~.${otherStep.name}`) {
            return true;
        }
        if(otherStep.equalTo == `~.${step.name}`) {
            return true;
        }
    }
    return false;
}

```

```
function ensureSimilarity(result, similarity, otherSimilarity) {
    let stepSimilarities = result[similarity.step];

    let existingSimilarity = stepSimilarities.find((stepSimilarity) =>
                                stepSimilarity.step == otherSimilarity.step);

    if(existingSimilarity) {
        if(similarity.type == "equal" && otherSimilarity.type == "equal") {
            existingSimilarity.type = "equal";
        }
        return;
    }

    if(similarity.type == "equal" && otherSimilarity.type == "equal") {
        stepSimilarities.push({ type: "equal", step: otherSimilarity.step });
        return;
    }
    stepSimilarities.push({ type: "similar", step: otherSimilarity.step });
}

function findPrime(similarities, ownName) {
    if(similarities.length == 0) {
        return ownName;
    }
    return similarities
                .map((similarity) => similarity.step)
                .concat([ ownName ])
                .sort()
                .find(() => true);
}
```

Listing E.4: buildSimilarityTable.js — constructs a similarity table from paths

```
module.exports = primePaths;

function primePaths(paths, similarityTable, type) {
    if(!type) {
        type = "similar";
    }
    if(type != "similar" && type != "equal") {
        throw new Error(`Attempting to prime paths without a valid given
                        'type'. Must be 'similar' or 'equal'.`);
    }
    let pathNames = Object.keys(paths);

    let result = {};

    pathNames.forEach((pathName) => {
        let path = paths[pathName];
```

```
17
18          let primedPath = path.map((step) =>
19                      primeStep(pathName, step,similarityTable, type));
20
21          result[pathName] = primedPath;
22      });
23
24      return result;
25  }
26
27  function primeStep(pathName, step, similarityTable, type) {
28      return similarityTable[`${pathName}.${step.name}`][`${type}Prime`];
29  }
```

Listing E.5: primePaths.js — primes paths such that each step refers to the alphanumerically first name of a step in its similarity cluster

```
1   module.exports = findSections;
2
3   function findSections(primedPaths) {
4       let pathNames = Object.keys(primedPaths);
5       let numPaths = pathNames.length;
6
7       let result = [];
8
9       pathNames.forEach((pathName, currentPathIndex) => {
10          console.log(`Finding sections in path ${currentPathIndex + 1}/${numPaths}`);
11
12          let possibleSections = findAllPossibleSections(primedPaths[pathName]);
13
14          console.log(` - Found ${possibleSections.length} possible sections`);
15
16          possibleSections.forEach((possibleSection) => {
17              if(resultContainsSection(result, possibleSection)) {
18                  return;
19              }
20
21              let sectionMatches = 1;
22
23              pathNames.forEach((otherPathName, otherPathIndex) => {
24                  if(otherPathIndex <= currentPathIndex) {
25                      return;
26                  }
27
28                  let otherPath = primedPaths[otherPathName];
29
30                  if(pathContainsSection(otherPath, possibleSection)) {
31                      sectionMatches++;
32                  }
```

```
            });

            if(sectionMatches >= 2) {
                let stepCommonalities = possibleSection
                                        .map((step) => {
                                            let occurrances = pathNames
                                                        .map((pathName) => primedPaths
                                                            [pathName].includes(step))
                                                        .filter((present) => present)
                                                        .length;
                                            return sectionMatches / occurrances;
                                        });

                let stepCommonalitiesSorted = stepCommonalities.sort();

                let commonalityHelp = stepCommonalities
                                        .map((stepCommonality) => {
                                            return { stepCommonality, sum: 1 };
                                        })
                                        .reduce((a, b) => {
                                            return {
                                                stepCommonality: a.stepCommonality +
                                                                 b.stepCommonality,
                                                sum: a.sum + b.sum
                                            };
                                        }, { stepCommonality: 0, sum: 0 });

                let commonalityAvg = commonalityHelp.stepCommonality / commonalityHelp.sum
                    ;
                commonalityAvg = commonalityAvg.toFixed(2);

                let numCommonalities = stepCommonalities.length;
                let commonalityMedian;
                if(numCommonalities % 2 == 0) {
                    commonalityMedian = (stepCommonalitiesSorted[Math.floor((
                        numCommonalities - 1) / 2)] +
                                         stepCommonalitiesSorted[Math.ceil((
                                             numCommonalities - 1) / 2)]   ) / 2;
                }
                else {
                    commonalityMedian = stepCommonalitiesSorted[numCommonalities / 2];
                }
                commonalityMedian = commonalityMedian.toFixed(2);

                result.push({ section: possibleSection, weight: sectionMatches,
                    commonalityAvg, commonalityMedian, stepCommonalities });
            }
        });
    });

```

```
78      return result;
79  }
80
81  function findAllPossibleSections(path) {
82      let sections = [];
83
84      for(let i = 0; i < path.length; i++) {
85          let remainder = path.length - i;
86          for(let j = 2; j < remainder; j++) {
87              sections.push(path.slice(i, i + j));
88          }
89      }
90
91      return sections;
92  }
93
94  function resultContainsSection(result, section) {
95      return result.some((entry) => {
96          let existingSection = entry.section;
97          if(section.length != existingSection.length) {
98              return false;
99          }
100         if(section.some((step, stepIndex) => existingSection[stepIndex] != step)) {
101             return false;
102         }
103         return true;
104     });
105 }
106
107 function pathContainsSection(path, section) {
108     let firstStep = section[0];
109     let start = path.findIndex((v) => v == firstStep);
110
111     if(start === -1) {
112         return false;
113     }
114
115     let allMatch = section.every((expectedStep, expectedOffset) => path[start +
            expectedOffset] == expectedStep);
116
117     if(allMatch) {
118         return true;
119     }
120
121     pathContainsSection(path.slice(start + 1), section);
122 }
```

Listing E.6: findSections.js

```
module.exports = findStories;

function findStories(primedPaths) {
    let pathNames = Object.keys(primedPaths);
    let numPaths = pathNames.length;

    let result = [];

    pathNames.forEach((pathName, currentPathIndex) => {
        console.log(`Finding stories in path ${currentPathIndex + 1}/${numPaths}`);

        let possibleStories = findAllPossibleStories(primedPaths[pathName]);

        console.log(` - Found ${possibleStories.length} possible stories`);

        possibleStories.forEach((possibleStory) => {
            if(resultContainsStory(result, possibleStory)) {
                return;
            }

            let storyMatches = 1;

            pathNames.forEach((otherPathName, otherPathIndex) => {
                if(otherPathIndex <= currentPathIndex) {
                    return;
                }

                let otherPath = primedPaths[otherPathName];

                if(pathContainsStory(otherPath, possibleStory)) {
                    storyMatches++;
                }
            });

            if(storyMatches >= 2) {
                let stepCommonalities = possibleStory
                                            .map((step) => {
                                                let occurrances = pathNames
                                                                    .filter((pathName) =>
                                                                        primedPaths[
                                                                        pathName].includes(
                                                                        step))
                                                                    .length;
                                                return storyMatches / occurrances;
                                            });

                let stepCommonalitiesSorted = stepCommonalities.sort();

                let commonalityHelp = stepCommonalities
```

```
                                        .map((stepCommonality) => {
                                            return { stepCommonality, sum: 1 };
                                        })
                                        .reduce((a, b) => {
                                            return {
                                                stepCommonality: a.stepCommonality + b.stepCommonality,
                                                sum: a.sum + b.sum
                                            };
                                        }, { stepCommonality: 0, sum: 0 });
                                        commonalityAvg = commonalityAvg.toFixed(2);

                let commonalityAvg = commonalityHelp.stepCommonality / commonalityHelp.sum;

                let numCommonalities = stepCommonalities.length;
                let commonalityMedian;
                if(numCommonalities % 2 == 0) {
                    commonalityMedian = (stepCommonalitiesSorted[Math.floor((numCommonalities - 1) / 2)] +
                                         stepCommonalitiesSorted[Math.ceil((numCommonalities - 1) / 2)]   ) / 2;
                }
                else {
                    commonalityMedian = stepCommonalitiesSorted[(numCommonalities - 1) / 2];
                }
                commonalityMedian = commonalityMedian.toFixed(2);

                result.push({ story: possibleStory, weight: storyMatches, commonalityAvg, commonalityMedian, stepCommonalities });
            }
        });
    });

    return result;
}

function findAllPossibleStories(path) {
    let stories = [];
    for(let i = 2; i < path.length; i++) {
        stories = stories.concat(pickStory(i, path));
    }

    return stories;
}

function pickStory(picksLeft, path) {
    if(picksLeft == 0) {
        return [ [] ];
```

```
    }

    let stories = [];

    for(let i = 0; i < path.length; i++) {
        let pick = path[i];
        let remainder = path.slice(i + 1);
        if(remainder.length >= picksLeft) {
            let endings = pickStory(picksLeft - 1, remainder);
            endings.forEach((ending) => {
                stories.push([ pick ].concat(ending));
            });
        }
    }

    return stories;
}

function resultContainsStory(result, story) {
    return result.some((entry) => {
        let existingStory = entry.story;
        if(story.length != existingStory.length) {
            return false;
        }
        if(story.some((step, stepIndex) => existingStory[stepIndex] != step)) {
            return false;
        }
        return true;
    });
}

function pathContainsStory(path, story) {
    if(story.length == 0) {
        return true;
    }

    let firstStep = story[0];
    let start = path.findIndex((v) => v == firstStep);

    if(start === -1) {
        return false;
    }

    let allMatch = pathContainsStory(path.slice(start + 1), story.slice(1));

    if(allMatch) {
        return true;
    }

    return pathContainsStory(path.slice(start + 1), story);
```

```
}
```

Listing E.7: findStories.js

```
module.exports = findGroupPrecedences;

function findGroupPrecedences(primedPaths) {
    let pathNames = Object.keys(primedPaths);
    let numPaths = pathNames.length;

    let result = [];

    pathNames.forEach((pathName, currentPathIndex) => {
        console.log(`Finding group precedences in path ${currentPathIndex + 1}/${numPaths}`);

        let possibleGroupPrecedences = findAllPossibleGroupPrecedences(primedPaths[pathName]);

        console.log(` - Found ${possibleGroupPrecedences.length} possible group precendences`);

        possibleGroupPrecedences.forEach((possibleGroupPrecedence) => {
            if(resultContainsGroupPrecedence(result, possibleGroupPrecedence)) {
                return;
            }

            let groupPrecedenceMatches = 1;

            pathNames.forEach((otherPathName, otherPathIndex) => {
                if(otherPathIndex <= currentPathIndex) {
                    return;
                }

                let otherPath = primedPaths[otherPathName];

                if(pathContainsGroupPrecedence(otherPath, possibleGroupPrecedence)) {
                    groupPrecedenceMatches++;
                }
            });

            if(groupPrecedenceMatches >= 2) {
                let stepCommonalities = possibleGroupPrecedence.preceeders
                                        .concat([ possibleGroupPrecedence.result ])
                                        .map((step) => {
                                            let occurrances = pathNames
                                                              .map((pathName) =>
                                                                  primedPaths[
                                                                  pathName].includes(
```

```
                                                                    step))
                                                                .filter((present) =>
                                                                    present)
                                                                .length;
                                                        return groupPrecedenceMatches /
                                                            occurrances;
                                                    });

                let stepCommonalitiesSorted = stepCommonalities.sort();

                let commonalityHelp = stepCommonalities
                                                    .map((stepCommonality) => {
                                                        return { stepCommonality, sum: 1 };
                                                    })
                                                    .reduce((a, b) => {
                                                        return {
                                                            stepCommonality: a.stepCommonality + b
                                                                .stepCommonality,
                                                            sum: a.sum + b.sum
                                                        };
                                                    }, { stepCommonality: 0, sum: 0 });

                let commonalityAvg = commonalityHelp.stepCommonality / commonalityHelp.sum
                    ;
                commonalityAvg = commonalityAvg.toFixed(2);

                let numCommonalities = stepCommonalities.length;
                let commonalityMedian;
                if(numCommonalities % 2 == 0) {
                    commonalityMedian = (stepCommonalitiesSorted[Math.floor((
                        numCommonalities - 1) / 2)] +
                                        stepCommonalitiesSorted[Math.ceil((
                                            numCommonalities - 1) / 2)]   ) / 2;
                }
                else {
                    commonalityMedian = stepCommonalitiesSorted[(numCommonalities - 1) /
                        2];
                }
                commonalityMedian = commonalityMedian.toFixed(2);

                result.push({ groupPrecedence: possibleGroupPrecedence, weight:
                    groupPrecedenceMatches, commonalityAvg, commonalityMedian,
                    stepCommonalities });
            }
        });
    });

    return result;
}

```

```
function findAllPossibleGroupPrecedences(path) {
    let groupPrecedences = [];
    for(let i = 2; i < path.length - 1; i++) {
        let numberOfPreceedersToPick = i;
        groupPrecedences = groupPrecedences.concat(pickGroupPrecedences(
            numberOfPreceedersToPick, path));
    }

    return groupPrecedences;
}

function pickGroupPrecedences(numberOfPreceedersToPick, path) {
    if(numberOfPreceedersToPick == 0) {
        return path.map((result) => {
            return { preceeders: [], result };
        });
    }

    let groupPrecedences = [];

    for(let i = 0; i < path.length; i++) {
        let pick = path[i];
        let remainder = path.slice(i + 1);
        if(remainder.length >= numberOfPreceedersToPick) {
            let endings = pickGroupPrecedences(numberOfPreceedersToPick - 1, remainder);
            endings.forEach((ending) => {
                groupPrecedences.push({
                    result: ending.result,
                    preceeders: [ pick ].concat(ending.preceeders)
                });
            });
        }
    }

    return groupPrecedences;
}

function resultContainsGroupPrecedence(result, groupPrecedence) {
    return result.some((entry) => {
        let existingGroupPrecedence = entry.groupPrecedence;
        if(groupPrecedence.preceeders.length != existingGroupPrecedence.preceeders.length)
            {
            return false;
        }
        if(groupPrecedence.preceeders.some((step, stepIndex) => existingGroupPrecedence.
            preceeders[stepIndex] != step)) {
            return false;
        }
        if(groupPrecedence.result != existingGroupPrecedence.result) {
            return false;
```

```
        }
        return true;
    });
}

function pathContainsGroupPrecedence(path, groupPrecedence, highestIndex) {
    if(!highestIndex) {
        highestIndex = 0;
    }

    if(groupPrecedence.preceeders.length == 0) {
        let resultIndex = path.lastIndexOf(groupPrecedence.result);
        return resultIndex !== -1 && resultIndex > highestIndex;
    }

    let firstStep = groupPrecedence.preceeders[0];
    let start = path.findIndex((v) => v == firstStep);

    if(start === -1) {
        return false;
    }

    if(start > highestIndex) {
        highestIndex = start;
    }

    let remainder = {
        preceeders: groupPrecedence.preceeders.slice(1),
        result: groupPrecedence.result
    };

    return pathContainsGroupPrecedence(path, remainder, highestIndex);
}
```

Listing E.8: findGroupPrecedences.js

Appendix F

Similarity table

Table F.1 shows the similarities between the steps identified in the architecture evolution paths under study. A full list of the architecture evolution paths we identified can be found in appendix D.

The table shows exact overlap (the steps describe the exact same change) by simply listing the overlapping step in the Overlap-column next to the step. For steps that are merely similar (not an exact overlap, but similar in intent and impact), the step is prefaced with a tilde ($\sim$), and likewise listed in the Overlap-column.

Step	Overlap	Step	Overlap	Step	Overlap
a.1	c.1, d.1	c.2	a.2, ∼ b.1, ∼ d.2, ∼ e.2	e.2	∼ a.2, ∼ b.1, ∼ c.2, ∼ d.2
a.2	∼ b.1, c.2, ∼ d.2, ∼ e.2	c.3	a.6, f.5	e.3	∼ a.4, ∼ c.6, ∼ f.3, ∼ f.7
a.3		c.4	∼ a.9, ∼ b.3, ∼ c.8.b, ∼ c.11, ∼ c.12, ∼ e.5, ∼ f.8, ∼ f.12	e.4	f.9
a.4	∼ c.6, ∼ e.3, f.3, ∼ f.7	c.5		e.5	∼ a.9, ∼ b.3, ∼ c.4, ∼ c.8.b, ∼ c.11, ∼ c.12, ∼ f.8, ∼ f.12
a.5		c.6	∼ a.4, f.7, ∼ e.3, ∼ f.3	e.6	a.7, c.8.a
a.6	c.3, f.5	c.7		e.7	∼ c.9
a.7	c.8.a, e.6	c.8.a	a.7, e.6	f.1	
a.8		c.8.b	∼ a.9, ∼ b.3, ∼ c.4, ∼ c.11, ∼ c.12, ∼ e.5, ∼ f.8, ∼ f.12	f.2	
a.9	b.3, ∼ c.4, ∼ c.8.b, ∼ c.11, ∼ c.12, ∼ e.5, f.8, ∼ f.12	c.8.c	∼ b.4.b, ∼ b.6, ∼ c.13, ∼ f.13	f.3	a.4, ∼ c.6, ∼ e.3, ∼ f.7
b.1	∼ a.2, ∼ c.2, ∼ d.2, ∼ e.2	c.9	∼ e.7	f.4	
b.2		c.10		f.5	a.6, c.3
b.3	a.9, ∼ c.4, ∼ c.8.b, ∼ c.11, ∼ c.12, ∼ e.5, f.8, ∼ f.12	c.11	∼ a.9, ∼ b.3, ∼ c.4, ∼ c.8.b, ∼ c.12, ∼ e.5, ∼ f.8, ∼ f.12	f.6	∼ d.4
b.4.a	b.5	c.12	∼ a.9, ∼ b.3, ∼ c.4, ∼ c.8.b, ∼ c.11, ∼ e.5, ∼ f.8, ∼ f.12	f.7	c.6, ∼ a.4, ∼ e.3, ∼ f.3
b.4.b	∼ b.6, ∼ c.8.c, ∼ c.13, ∼ f.13	c.13	∼ b.4.b, ∼ b.6, ∼ c.8.c, ∼ f.13	f.8	a.9, b.3, ∼ c.4, ∼ c.8.b, ∼ c.11, ∼ c.12, ∼ e.5, ∼ f.12
b.5	b.4.a	d.1	a.1, c.1	f.9	e.4
b.6	∼ b.4.b, ∼ c.8.c, ∼ c.13, ∼ f.13	d.2	∼ a.2, ∼ b.1, ∼ c.2, ∼ e.2	f.10	∼ d.3
b.7		d.3	∼ f.10	f.11	
c.1	a.1, d.1	d.4	∼ f.6	f.12	∼ a.9, ∼ b.3, ∼ c.4, ∼ c.8.b, ∼ c.11, ∼ c.12, ∼ e.5, ∼ f.8
		e.1		f.13	∼ b.4.b, ∼ b.6, ∼ c.8.c, ∼ c.13

Table F.1: Table of similarities of steps in our architecture paths.