Architectural Principles for Autonomous Microservices

Anders Mikkelsen
Systek
Oslo, Norway
anders.mikkelsen@systek.no

Tor-Morten Grønli
Kristiania University College
Mobile Technology Lab
Department of Technology
Oslo, Norway
tor-morten.gronli@kristiania.no

Damian A. Tamburri
Jheronimus Academy of Data Science
Technical University of Eindhoven
Netherlands
d.a.tamburri@tue.nl

Rick Kazman
University of Hawaii
Schidler College of Business
The United States of America
kazman@hawaii.edu

Abstract

In the last decade architectural patterns like microservices and event-driven architectures have emerged to meet the challenges of distributed computing. This paper explores recent practices and research in microservice- and event-driven architectures to identify the challenges in architecting such systems. With a proof-of-concept study approach, we distilled a set of design principles to address these challenges, creating asynchronous and agnostic microservice architectures. Further, we provide a generic toolkit for implementing such architectures. An example of this architecture was implemented in the company TechnipFMC. Concurrently, an architecture trade-off analysis was performed using a utility-tree based approach, highlighting the impact and importance of our proposed principles and their generalizability. The evaluation provides evidence for the viability of the proposed design principles.

1. Introduction

System and software architectures distributed across multiple machines are difficult to design and create [1, 2]. The most common reasons for implementing a distributed architecture are for scaling, either vertically or horizontally [1]. This strategy is employed for managing architectures [3] and separating concerns, and becomes clear with Microservices Architectures (MSAs) [4, 5, 6]. But what exactly is a MSA? Definitions are lacking but opinions and proposed implementations are not. In this paper, we survey the research on MSAs and describe a set of derived design principles and a reference implementation, in the form of a toolkit. Furthermore, we present our results on the realization of this toolkit for an industry partner—TechnipFMP. We also present a trade-off analysis featuring a utility tree extracted as part of the Architecture Tradeoff Analysis Method (ATAM) [7], [8] in the context of 6 industrial case-studies. Through an analysis of these experiences, we highlight how adherence to the design principles can increase the quality of MSAs. Finally, we conclude the paper by giving recommendations for future work.

2. Background

In a monolithic approach to software architecture every piece of business logic exists within a single process, often a single application. Even though the internal workings of the monolith might be service oriented it is not separated by process, so scaling is performed by starting additional instances of the entire application [9, 1]. Service orientation in the context of a monolith means that you can logically separate the monolith into several modules that have high cohesion and low coupling, without splitting it up into multiple separate services. Usually these end up being highly coupled because of the nature of the monolith, as shown by Gouigoux et al. [10].

As businesses have become more agile in delivering functionality, and as the number of users consuming web services has exploded recently, more scalable solutions were highly sought after. One of the approaches that grew organically from industrial practice has become known as Microservices Architecture [5, 6, 11]. As understood by practitioners and academics the three main pillars of MSA are those of: improving speed of change, having small cohesive units of functionality, and supporting scalability [12, 9, 5, 11].

Whether MSA is a separate style or a variant of Service Oriented Architecture (SOA) is contested [9, 4]. Some say MSA is a best-practices approach to SOA, while others [9] say it is a separate paradigm [5, 6, 13, 4]. There are also positions that embrace both ends of this spectrum, by defining multiple implementations of MSA; some closely resemble SOA in a fine-grained form; others, like Mulesoft, do not [12]. Thus there is no broadly accepted definition of MSA or even microservices themselves [13, 9, 5, 6]. However there appear to be some universals concerning microservices that practitioners do agree upon: microservices should be small with high cohesion [9, 11], each should run in its own process [11] and they communicate with each other via a lightweight interface, often a REST API [5, 6].

High cohesion in microservices takes the Domain model of SOA a step further. In many ways MSA is all about separating processes into as small services as possible—small not in the context of storage space or computational power but that of domain [9, 4, 14]. An MSA might, for example, separate an end-to-end process of placing orders into a shopping cart service, a line item service, an order service, and a check out service. This allows it to scale at a much more fine-grained level than a classical SOA. The importance put upon cohesion varies with implementation—maybe more than any other tenet of MSA—as shown by Shadija et al [9]. Even though they vary, all definitions speak of encapsulation, low coupling, autonomy and similar terms [11].

Although internal communication between services is normally performed with HTTP REST API's in SOA and MSA, other options are available, like message-based communication. Message-based communication can be synchronous or asynchronous variations, realized in three different implementations: Remote Procedure Calls (RPC), Object Request Broker (ORB), and Message oriented Middleware (MOM). These are explained in detail by Mahmoud [15]. RPC and ORB implementations are typically synchronous, meaning that when a procedure on one node calls another it must await the response before doing anything else. This is opaque to the programmer but does not scale well.

With the asynchronous variant of MOM there is generally an external broker passing messages around and the details of this is implemented by programmers. Asynchrony enables individual processes to continue working without waiting for a response, but it loses the simplicity of RPC or ORB [15]. For external communication with clients a typical pattern in a distributed architecture is running a Gateway that acts as a bridge. This was as true in the 70's when Farber was investigating Distributed Architectures [2], as it is today with companies like Netflix adopting this pattern [16]. Implementations vary but examples range from HTTP load balancers, with the chief objective being to forward requests to appropriate microservices, to a service (or collection of microservices) concatenating results from multiple microservices and returning the aggregated result to the client, as is the case with the Netflix API gateway [16]. Both of these approaches create an implicit coupling between a service and its gateway in terms of communicating with a client. In other words, any change in communication between a service and its clients usually creates a change in the gateway.

A MSA, operating with message-based communication, is at its core an Event-Based/Driven Architecture (EBA) [17]. EBAs are architectures where interactions are asynchronous and handled by a discrete data object known as an Event. This discrete data object is a representation of something that happened in the past, with a description of what happened and a timestamp. The state of any element of the architecture, like a model, is the aggregated result of all events pertaining to that element. An easy way to imagine this is to think of a bank account. The balance is the accumulation of all previous deposits and withdrawals [17]. A typical implementation of an EBA uses an external broker to route events to wherever they need to go [12]. This is similar to the Enterprise Service Bus of a SOA, although constrained to event management.

An EBA might only use these events as state updates that services use to modify their own state according to business rules, or it might employ what is known as Event Sourcing (ES) to structure the state of all data as aggregates of events [18]. ES is not new and is not necessarily tied to EBA's but plays well with them as discussed by Fowler [19]. The idea of ES is that every change in application state is represented by an event object and stored in timestamped order. This creates a log of everything that has happened in the application over its lifetime, which facilitates several operations. For example, you could discard the application state and completely rebuild it by replaying all events, and you can examine the application state in any point in time by replaying events up to a certain time. You can also remove incorrect events, then rebuild the state by replaying all events after the corrected one, as outlined by Fowler [19], and explained in detail by Overeem et al. [18]. It is important in this context to distinguish events that operate as commands in an EBA from events that describe a mutation of the application state. One service telling another service to perform some operation periodically might be a command event, but not necessarily an application mutation that warrants storage [12]. This is shown as a detailed flow by Yang for a data analysis task. The system receives an event that is interpreted as a command by the server which then performs the analysis and returns an aggregated data event [20].

ES has its own challenges not observed in traditional stateful data stores, mostly tied to faulty events and changes to the data models represented by the events [19, 18]. As is mentioned by Overeem et al. events are stored as schemaless entities but in reality, they have an implicit schema. The application assumes a schema when reading the events. This means that the store holding the events cannot supply tools for updating the schema as it is not aware of any schema, thus pushing the responsibility of schema changes to the application [18]. Fowler explains that ES obviously applies some constraints on your architecture which demand a different approach and mindset [19]. There appears to be consensus on this, but with a warning that this style of state management might seem alien to those who have not implemented it before [12, 19, 18].

EBAs are not new [2], and applying their principles to software architectures is gaining momentum [17, 21, 22]. An industrial study examines the challenges faced by LinkedIn when scaling up [23], and makes the case that events are essential building blocks of microservices. This mostly comes from the problem of managing state in large-scale distributed architectures. The foundation of an event-based architecture is to encapsulate a unit of change in the architecture as a standardized event abstraction [12]. A primary challenge of EBAs is that of ensuring event consistency: what happens when an event either is not sent correctly, or disappears due to hardware issues? The original sender service won't know that anything is wrong when the event is on the queue. So what can the service, which is ultimately responsible for the event, do about that?

3. Architectural Principles for Autonomous Microservices

Based on the Background described above, in this section we now outline the reasoning behind the four design principles for Autonomous Microservices (AM) architectures: Communication Independence, Organizational Agnosticism, Scalability, and Independence. These four principles define the core of what it means to be an AM architecture. Subsequently, this section describes our open-source toolkit, which serves as the reference implementation of AM, and realizes the four principles in production-ready code.

3.1. Communication Independence

The Communication Independence principle constrains inter- and intra-service communication as well as how the architecture should communicate with external clients.

1. All inter-service communication must, in all cases, communicate on non-blocking technologies. HTTP is an example of a blocking technology. Asynchronous TCP messages are an example of non-blocking technologies. At some point all traffic may be blocking, of course, but we are concerned with the architectural abstraction, not its infrastructural realization.
2. All inter-service communication is point-to-point and not reliant on external load balancers.
3. Services contact other registered handlers, e.g., other services or external clients, in a round robin fashion, to balance load and ensure access to all handlers in the event of retries or other communication issues.
4. All external communication negotiates its initial connection through a gateway, for unified access control, and all subsequent communications are performed point-to-point, in both client-to-server and client-to-client communication.

3.2. Organizational Agnosticism

Agnosticism dictates what should be provided by an architecture for AM, so that service creation and interaction is opaque to any developer team working on it. A developer should be able to confer with the consumer of their service to determine what the contract should be and deploy it with no architecture configuration required, and with no concept on where or how their service is run.

1. All message brokering is performed by a service and its recipient, dependent on whether it is important for it. For example, if a service is reliant on a response to a request, it must employ techniques such as circuit breakers, queues, retries etc. itself.
2. There are no masters controlling internal or external traffic, apart from regex filtering in the gateway for external clients. The filtering is a comprise for creating a demarcation line between external and internal consumers. This allows services to stay agnostic in their communication while having confidence that any information they send is not published outside the architecture unless they explicitly allow it.

3.3. Scalability

The scalability principle governs rules for service configuration and implementation to ensure that the architecture is as decentralized as possible, and to ensure linear scalability. No service knows anything above its own and its sibling's existence. This means that any coordination of communication cannot be controlled above the service scope. Scaled instances know of each other and may coordinate, as long as they do not employ distributed data structures that might block execution across instances. In our previous work on immutable architectures we see that in modern implementations scalability on an infrastructural level is enabled by the use of containerization, which in turn helps achieve portability [24]. Although portability through containerization is not a key architectural concern for autonomous microservices and in extension the scalability principle, it is a clear enabler.

1. Low resource consumption is key to scalability, and as such services should do the most amount of work on the least amount of kernel threads.
2. Treat termination and removal of any infrastructural components used by the application e.g., file shares or databases as normal occurrences, do not rely on graceful shutdown.
3. All operations not relying on third party technology restricted to another paradigm shall be asynchronous and non-blocking.
4. Any coordination is only permitted in the scope of a service, and the scaled out instances of said service. Direct message replies, in the form of asynchronous events within a service scope to coordinate incoming events that are published to all instances is fine, as there is always at least one instance available, even if it is the sender itself.

3.4. Independence

Independence dictates the life cycle of an AM. In this lies the parameters for scaling in and out, which is managed within the service scope, and the life time of the service. All services must be based upon acting on incoming data, and then responding to that by producing and publishing their own data. All communication is directed at addresses pointing to a single recipient. Such addresses can be TCP ports, bus addresses, HTTP endpoints etc. Services shall not call other services directly as part of their operations, but simply be content that any number of services might, or might not, be listening to their broadcasts.

1. Services are equal: there are no masters and no slaves.
2. Services are not terminated externally; they terminate themselves upon receiving information about scaling properties, either from another service, or from an orchestration system.
3. Services react to incoming data and produce their own outgoing data. They do not call other services directly, apart from acquiring outgoing data addresses.
4. Periodic tasks and time-based actions are performed and managed by the service itself.

3.5. Toolkit

The open source toolkit is meant as a reference implementation for the AM Design principles [1], but this toolkit is not required to produce AM-compliant architectures. An overview of the architectural elements of the toolkit is sketched in Figure 1, in the region under the Client/Internal Event Barrier. It is separated into three logical layers. In addition, we will refer to two *zones*, an external and internal. Internal is defined as any services running on the architecture, while external is defined as any other actor that can interact with the eventbus.

3.6. Layers

The gateway layer is distributed across all nodes in the autonomous microservices architecture. The gateway provides two-way communication between the external clients and the eventbus. Incoming traffic is handled by separate services as splitters, and outgoing traffic is handled by separate services as combiners. The gateway provides a regex-based approach for what base addresses are allowed from external clients but is otherwise agnostic to their interaction with internal services. This enables the addition and removal of services on the clustered architecture to expand features for existing or new users (internal services, customer systems etc.), without the need to reconfigure the architecture. This means we treat every microservice as a completely separate entity.

Furthermore, we can easily expose the addresses of a service to the public without any architectural configuration. The only entities that are aware of this implementation are the team responsible for creating and maintaining it, and the consuming client.

The data splitters and combiners are plugins that exist in this layer because they don't do direct computation on data. Their function is to take incoming data and split it into whatever components a particular splitter is designed for. This enables us to easily add representations of incoming data for some kind of data processing down the line. The data combiners are used for aggregating different data and creating complex objects based on it. This enables us to easily expand the API with functionality without changing the architecture, in a similar way to the architectural adapter pattern [25] by having the new combiners "implement" several other data services or combiners. This means a

[1] https://github.com/mikand13/autonomous-services

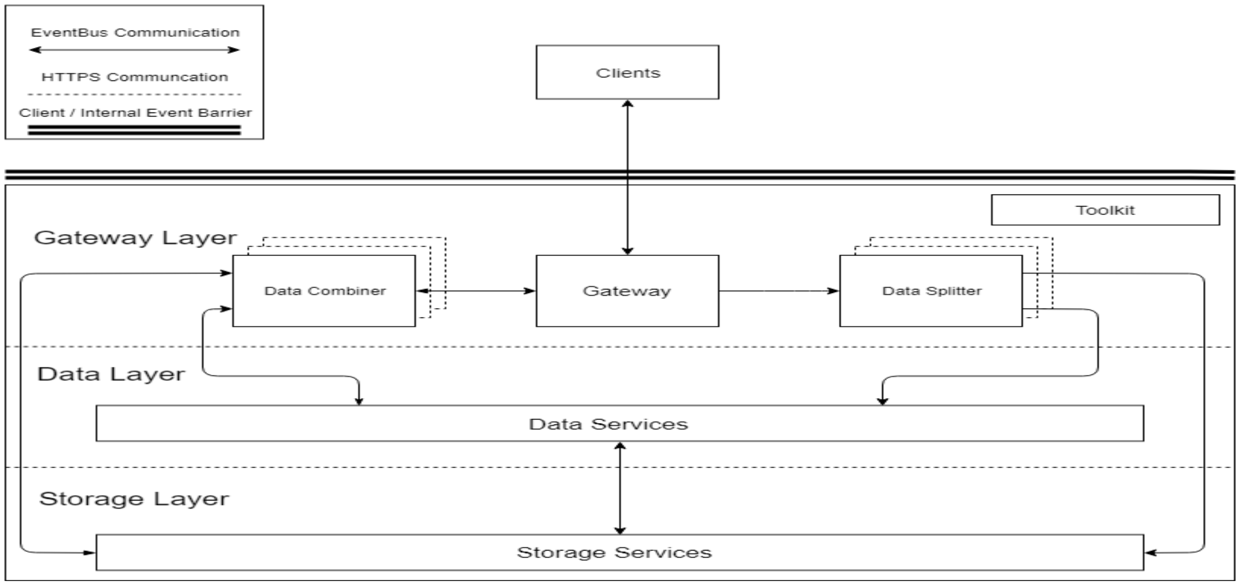

Figure 1. Autonomous Microservices: a Reference Architecture

particular development team could expand functionality for a particular use case autonomously. Splitters and combiners can be set up for both external and internal traffic. In the case where the input is similar but can be produced both externally and internally, you should take care to ensure there is no implicit coupling by handling both in the same splitter or combiner.

Behind the gateway layer is the data services layer. This layer comprises all data processing and computational services. These services accept data on a specific address, subscribe to any applicable splitter addresses, and report any results to specific addresses, both as publication (to any actor interested) and as a direct send to a storage address, when applicable. The data service itself does not concern itself with the result of this send operation. If it does it should implement logic for persisting the message and retrying it with an exponential backoff, like a normal microservice would. Data Services publish relevant results of data processing to external services, e.g. representations for customers, management etc. This allows the addition of processing services, without manipulating the API.

At the end of the chain we have the storage services layer. This layer comprises all services directly related to data at rest. These services accept data on specific addresses, like splitter/data service addresses, and store that data in whatever way they see fit. This also enables us to receive data across multiple services using a single address to store that data in different ways. This isolates the storage specifics of the data the service handles, and enables fine-grained storage control. Addresses are exposed for fetching this data directly by data combiners in the gateway layer, and combining it with other data before producing a final result.

Whenever data is changed in a storage service this information is published to addresses to notify any actors that require real-time information about that data. An example of this would be one data service doing some computation, which results in data being stored in a storage service. Upon publication of that information another data service might start computations on this new data, as well as a customer system responding to said information and performing some external work.

4. Results

Describing the entirety of the concrete solution produced for TechnipFMC would take up too much space and divert attention from the primary focus of the paper. Instead, we focus on a Trade-off analysis presented in section 4.2

4.1. TechnipFMC Architecture

The architecture produced for TechnipFMC employs the same technologies as the toolkit, namely Vert.x and nannoq-tools, and it is hosted on Amazon Web Services (AWS). The architectural diagram used for TechnipFMC is shown in Figure 2. Within the internal zone it bears close resemblance to the toolkit diagram seen in Figure 1, but it is more fine-grained in its representation of the outer zone. This is to contextualize what kind of traffic is expected in the short term and the disparity of data and clients to be expected in the future.

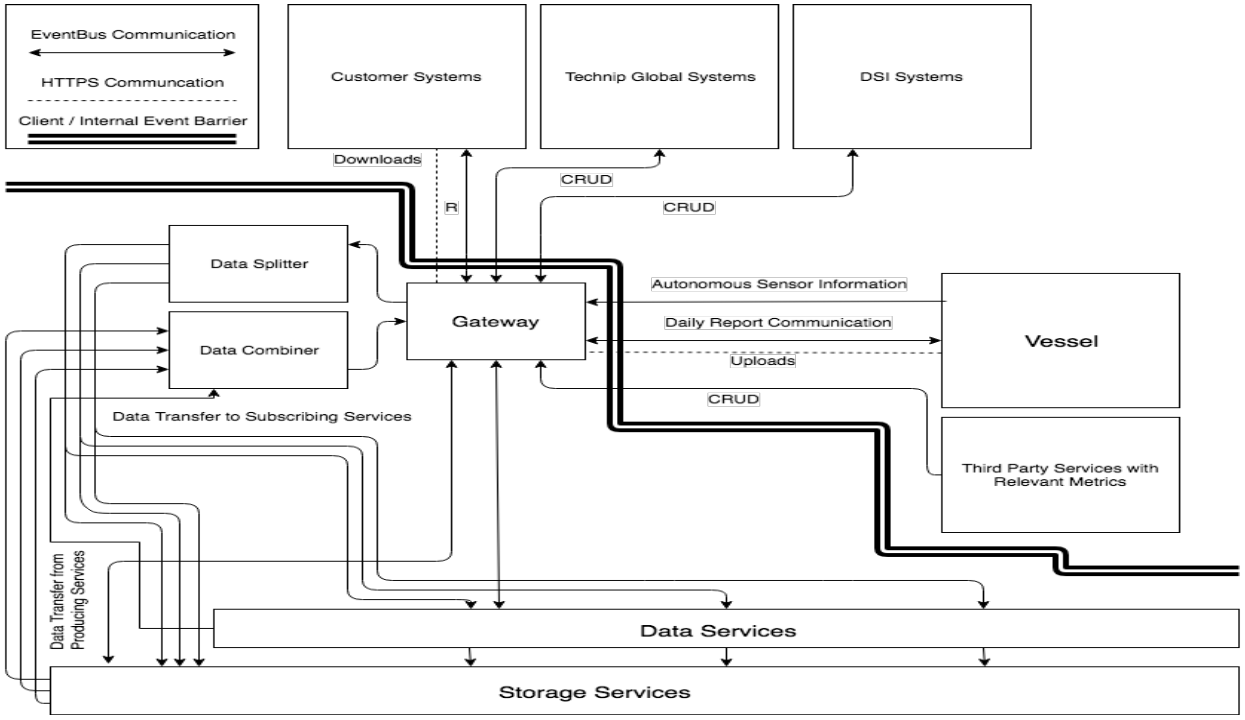
EventBus Communication
HTTPS Communcation
Client / Internal Event Barrier
Customer Systems
Technip Global Systems
DSI Systems
Downloads
R
CRUD
CRUD
Data Splitter
Data Combiner
Gateway
Autonomous Sensor Information
Daily Report Communication
Uploads
CRUD
Vessel
Third Party Services with Relevant Metrics
Data Transfer to Subscribing Services
Data Transfer from Producing Services
Data Services
Storage Services

Figure 2. TechnipFMC Autonomous Microservices Architecture

The architecture is based on a large set of splitters for a wide range of incoming data, and combiners for a myriad of different visualizations. With every added splitter or combiner another node, and in effect gateway, is added to the cluster. This, combined with the Claimer/Collector patterns with point-to-point communication to coordinate within the service scope we will not impact the rest of the cluster. This gives us near-linear scalability without needing to configure external brokers to meet the increased traffic.

4.2. Architecture-Based Trade-Off Analysis and Evaluation

To evaluate the extent to which the aforementioned design principles and practices are generalisable, we evaluated them in the context of the utility trees extracted from 6 industrial case-studies we conducted in previous work [26]. On the one hand, a utility tree [27] is a stakeholder-friendly way to organize software architecture quality principles and attributes and is used in the context of the Architecture Tradeoff Analysis Method (ATAM), a workshop based stakeholder-engagement activity aimed at eliciting and reasoning about the tradeoffs among quality attributes characteristics. On the other hand, in the scope of our evaluation, utility trees serve the purpose of understanding the extent to which the aforementioned design principles and practices were already used in these 6 industrial cases, where legacy assets were migrated to a cloud-native solution.

ID	Segment	#Stakeholders	#Services
1	Avionics	7	44
2	Automotive	5	31
3	Construction	4	23
4	Data-Processing	3	11
5	Resource-Mgt	5	18
6	Tourism	6	22

Table 1. An overview of the cases evaluated in our study; domains range from avionics to tourism.

For the sake of space, we cannot fully flesh out the details pertaining to each case but briefly describe the market segment and number of stakeholders engaged in each, as well as the size of the architecture in question. This information is summarized in Table 1.

The table highlights the variety of architectures targeted in our evaluation as well as the different sizes of the ATAM workshops we instrumented as part of our study. This sample allows us to control for size of the architecture, providing for small (10-19) to medium (20-29) as well as larger architecture sizes, in terms of the number of services.

Our research conjecture is that in every one of these cases there exists evidence for the application of the aforementioned principles and practices for the purpose of structuring the refactored architectures.

4.2.1. Research Approach

The research approach consisted of a comprehensive literature review highlighting the research domain and providing the foundation upon which we built the architectural principles. After distilling these architectural principles we proceeded to an implementation together with our industrial partner TechnipFMC. To elicit the aforementioned evidence of application of the proposed design principles in the target case-studies, we did a thematic coding [28] of the ATAM utility trees generated in 12 workshops. The utility tree is an elicitation device that helps to elicit and prioritize the quality attribute requirements for a system. Our aim was to associate one (or more) of the proposed design principles to the quality attributes in the utility trees elicited for the 6 case-studies. Subsequently, we operated content and frequency analysis to distill the quantities in question.

4.2.2. Evaluation Results

Figure 3 outlines our results from the ATAM-based evaluation. The tree in the figure is a collapsed version of the 6 utility trees that emerged from two ATAM workshop rounds conducted for each case study. This tree collapses together all of the major concerns reported as part of each of the workshops (the leaves of the tree) as well as the architectural quality attribute to which the concerns relate to (intermediate level of the tree). Finally, the tree is tagged with the frequencies of each of the concerns.

The tree shows (see the dotted boxes in Fig. 3) that several dimensions emerged from our case-studies, namely system security, its observability and security—where indications of the impact of the independence principle occur 22 and 17 times, respectively—as well as scalability (where indications of the impact from the scalability principle occur 16 times) and organizational structure (where indications of the impact from the agnosticism principle occur 18 times). These concerns are addressed by the four AM design principles proposed in the article. Furthermore, the frequency of these concerns accounts for 32% of all coded concerns throughout the 6 case studies—namely, we summed and ratioed the recurrence of concerns as shown in Fig. 3. This frequency highlights the impact and importance of the proposed principles and suggests that they are generalisable outside of the scope of the industrial environment in which the AM principles were

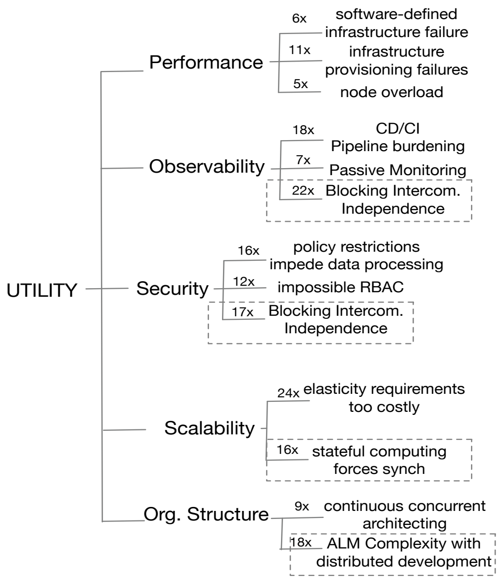
UTILITY
Performance
6x software-defined infrastructure failure
11x infrastructure provisioning failures
5x node overload
Observability
18x CD/CI Pipeline burdening
7x Passive Monitoring
22x Blocking Intercom. Independence
Security
16x policy restrictions impede data processing
12x impossible RBAC
17x Blocking Intercom. Independence
Scalability
24x elasticity requirements too costly
16x stateful computing forces synch
Org. Structure
9x continuous concurrent architecting
18x ALM Complexity with distributed development

Figure 3. A blended ATAM-based utility tree from all 6 case studies. Dotted boxes reflect shortcomings aligned with the AM design principles.

originally incepted.

5. Discussion

We have identified key challenges in the way microservices are viewed and employed today. These are mainly: HTTP as a synchronous request/response technology, load distribution and connection saturation, and external brokers, such as Kafka (a message queuing middleware [29]) that create strong coupling and establish a sort of master-slave relationship, which is counter to microservice principles. Coupled with this, we have identified shortcomings in real-world architectures, from the tradeoff analysis, and we now apply this understanding as a basis for our discussion.

As described by Mulesoft and other practitioners in an IEEE panel, there are quite a few challenges with microservices [12, 5, 6]. If you think that these challenges won't apply to you if you just "do it right", you will pay dearly. For example, our evaluation shows that 32% of the total concerns in our showcased examples can be linked back to our design principles, as outlined in Sec. 3. An example of how existing SOA principles do not scale up can be found in fine-grained SOA. This implementation is, in their words, the "big band" of MSA and was born out of the acknowledgement that traditional SOA which are relatively coarse-grained services are too difficult to meaningfully change without side-effects. That approach is holding development teams back. Their solution seems simple: break all the services up into finer-grained services with a single purpose. However, this introduces several impacts on your system [12, 5], such as increased traffic, a large number of services to manage and automatic orchestration for integration, testing and deployment. So in general you can say that while the initial ambition of MSA was to reduce the size of services, eventually it became clear that that in itself was not enough to fulfill the ambition of MSA.

Furthermore, from an infrastructure perspective, HTTP appears to be the main source of communication for microservice architectures (MSA) [5, 9, 4, 30, 11, 12]. That makes a lot of sense considering it is extremely well documented and tested. Through the DNS system of the Internet, URLs are very easy to reason with and most hosting providers operate on HTTP based load-balancers. While we did manage to identify a few methods of communication that differ from HTTP, like Gossip-Based communication, this approach has challenges. For example, monitoring the actual flow of execution is difficult.

Finally, Message-Based communication, which can be seen as the precursor for EBAs is interesting. There have been many implementations of this in the last 15 years, usually revolving around external brokers [12]. We view this as an anti-pattern for MSA as it undermines the foundation of SOA. All services should be able to communicate even if another external service has a catastrophic breakdown. In the scenario of an external broker, like the message queues and event queues of Mulesoft [12] and as described by Newman [30] and Shadija et al [9], they are "communication services" because at the instant they go down, the entire architecture breaks. This has motivated Newman's "Decentralize all the things" principle [30]. This single point of failure is a central problem, and addressing it is fundamental to the AM principles.

5.1. Revisiting the problem

By strictly enforcing asynchronous communication between services, as shown in the reference implementation, and removing the need for a master configuration by implementing new patterns like Claimer and Collector in the toolkit we achieve fully decentralized coordination. This in itself however is not necessarily enough to prevent the bottlenecks inherent in Master-Slave patterns. As stipulated in the Communication Independence principle we must also ensure that communication is point-to-point between services so that our coordination scales with the architecture. This is resolved by employing the Vert.x technology, and refraining from using any of the distributed data structures which are based on Hazelcast and concurrency locks across nodes.

The primary focus of the Agnosticism and Independence principles lies in increasing development speed, and reducing architectural complexity. Because of the decoupled nature of AM, a skilled team always knows the answer to questions such as the ones emerging in our case-studies, such as: "Where should we schedule batch tasks?", "Where should we configure the API to allow access to our service?" etc. Regarding questions of centralized control structures the answer is "just don't do it"; it must be managed within the service scope, preferably event-based without the use of permanent storage unless explicitly required. API access is always handled automatically by the Gateway layer at runtime by registering and unregistering services, so that no further configuration is needed. HTTP obviously breaks this contract because you must be able to route to an instance, and in real world applications you need to support this protocol, as discussed by the IEEE panel on typical implementations of microservice architectures [5, 6]. However, our principles are intended as a solution going forward, and do not take responsibility for being backwards compatible with everything that came before.

As a result of these technical challenges in contemporary architectures and varying opinions on exactly what a microservice is, we chose to contribute a concrete architecture embodying our four design principles for asynchronous and agnostic microservices. These principles are meant to be concrete in their representation, so as not to provide yet another vague definition of exactly what a microservice is. What these design principles do is define the architectural style of AM's in such a way that they can be reasoned about as concrete microservices and not as merely vague concepts. We now discuss how we make each of these principles concrete.

5.2. Adherence to the Communication Independence Design Principle in the Toolkit

For non-blocking communication we achieve this with point-to-point TCP messages sent asynchronously between instances of a service over the distributed eventbus. HTTP is only used for those services that have a client that demands it or when we are transferring files across the network. It would be an unnecessarily complicated operation to transfer large binaries over the eventbus, and would add unnecessary overhead. In these scenarios an initial request is sent over the eventbus to request an up- or down-load URL.

5.3. Adherence to the Organizational Agnosticism Design Principle in the Toolkit

Cohesion is key to adhering to this principle and agnosticism has been a primary concern in constructing the interfaces and classes used by the entirety of the Toolkit. All the services have been designed for, and been tested in, Docker containers. This gives easy encapsulation across different environments. This is explained in detail by Gouigoux et al. [10]. It allows us to deploy the service with all dependencies included directly, regardless of the host configuration. In addition they have been designed to be inward-looking, and only require the access address to any service that has information they need. With the Claimer and Collector patterns for communication all instances of services can coordinate between each other and interact with other services without breaking the Agnosticism principle, nor the Scalability principle.

It would be natural to construe the Gateway layer as a new Master-Slave configuration as discussed by Pautasso et al [5], and thus invalidating Autonomous Microservices as "just another Microservices approach", but it is in fact a logical Gateway which is distributed across all nodes in the reference implementation, by use of the BackgroundGatewayLauncher in the gateway module. This way the capacity for throughput in and out of the gateway scales with the architecture and is not reliant on a dedicated scaling of a "Gateway service", most commonly known as an API Gateway.

5.4. Adherence to the Scalability Design Principle in the Toolkit

Low resource consumption is a key element of scaling, especially when implementing fine-grained microservice architectures, as we do not want to pay for more hardware than necessary. As shown by Francesco [4] we gain a finer ability to scale our resources with this separation. But as detailed by Shadija et al [9] we also increase overhead for monitoring and distribution. For that reason we needed to investigate lightweight technologies for the reference implementation that provided both asynchrony and non-blocking technologies to maximize available performance. Due to extensive experience with it and the added bonus of the asynchronous and non-blocking nature of Vert.x, that is the technology we ended up using. One could argue that there are more tried and tested technologies out there, and while we agree, they are usually in the form of large frameworks or otherwise heavyweight enterprise platforms that are not natural to microservice based architectures. So even though Vert.x is not an absolute requirement for establishing low resource consumption for AM, it is the "weapon of choice" for our reference implementation. The way low resource consumption is achieved is with the multi-reactor pattern at the heart of Vert.x. In essence this means that all available kernel threads are utilized, and at the same time this constrains the application from threading above that count. This means we will not bloat memory with threads or experience thread starvation. We can then reserve more memory for application concerns and more realistically calculate the memory and CPU needs of a single service.

5.5. Adherence to the Independence Design Principle in the Toolkit

There are no concrete examples of the self-management of the lifecycle of a service in this principle in the toolkit, because that would inflate the scope a tremendous amount. We could have produced a notification based interface integrated with AWS but for this paper we choose to focus on the architectural elements of Autonomous Microservices and not those that border on infrastructural concerns. However, we did not think it would be correct to remove it from the principle because it is an integral part of the idea of AM. So it remains, not currently handled in the toolkit because of scope and resource limitations, but as an inspiration for future work.

5.6. Limitations

We acknowledge there are some non-trivial limitations to this work, in particular concerning its generalisability. Our research has introduced and elaborated on our 4 design principles and showcased them through an industrial implementation. Given space limitations in this article we could extensively elaborate on the implementation and acknowledge that doing so could have paved the way for an even higher degree of transparency, insight into the enforceability and general applicability of our design principles. We acknowledge that more research and more company implementations must be conducted to claim that our principles constitute a true design pattern or architectural style. However, we do think the results extensively point in this direction and the concepts presented are all worthy of further pursuit.

6. Conclusion

This paper set out to describe approaches for achieving linear scalability in MSAs, by eliminating centralized communication brokers, as presented in Section 1. This paper has produced two concrete contributions: 1) a set of AM design principles along with an open source toolkit as a reference implementation and 2) a trade-off analysis, employing a utility tree, highlighting impacts of AMs in several case studies. The four design principles presented in Section 3 and discussed in Section 5 specify the constraints that should be enforced on an AM architecture.

The toolkit was successfully expanded upon to create a proof of concept architecture for data collection at TechnipFMC. In this system we were able to demonstrate how an AM architecture diverges from the typical reliance on centralized coordination to facilitate near-linear scalability. As we have observed, the use of these AM principles leads to a dramatically less complex environment into which teams can introduce new services. The results of the implementation for TechnipFMC demonstrates the viability of the AM principles in a real project, and showcases the decentralized nature of the communication. The point-to-point nature of communication and adherence to the AM design principles resulted in near-linear scalability.

References

[1] M. P. Papazoglou and W.-J. van den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB Journal*, 2007.

[2] D. J. Farber, "Software considerations in distributed architectures," *Computer*, vol. 7, 1973.

[3] J. Soldani, D. A. Tamburri, and W.-J. van den Heuvel, "The pains and gains of microservices: A systematic grey literature review.," *Journal of Systems and Software*, vol. 146, pp. 215–232, 2018.

[4] P. D. Francesco, "Architecting microservices," *IEEE International Conference on Software Architecture Workshops*, 2007.

[5] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 1 : Reality check and service design," *IEEE Software*, 2017.

[6] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, and N. Josuttis, "Microservices in practice, part 2 : Service integration and sustainability," *IEEE Software*, 2017.

[7] P. Clements, R. Kazman, and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*. Addison Wesley, 2001.

[8] S. Bellomo, I. Gorton, and R. Kazman, "Toward agile architecture: Insights from 15 years of ATAM data.," *IEEE Software*, vol. 32, no. 5, pp. 38–45, 2015.

[9] D. Shadija, M. Rezai, and R. Hill, "Towards an understanding of microservices," *23rd International Conference on Automation and Computing, University of Huddersfield, Huddersfield*, 2017.

[10] J.-P. Gouigoux and D. Tamzalit, "From monolith to microservices - lessons learned on an industrial migration to a web oriented architecture," *IEEE International Conference on Software Architecture Workshops*, 2017.

[11] J. Lewis and M. Fowler, "Microservices: A definition of this new architectural term," 2014. Online; accessed 9 Nov. 2017.

[12] Mulesoft, "The top 6 microservices patterns," , Mulesoft, 2017.

[13] S. Hassan and R. Bahsoon, "Microservices and their design trade-offs: A self-adaptive roadmap," in *IEEE International Conference on Services Computing*, 2016.

[14] A. Sill, "The design and architecture of microservices," *IEEE Cloud Computing*, vol. 3, 2016.

[15] Q. H. Mahmoud, *Middleware for Communications*. The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England: John Wiley and Sons Ltd., 2004.

[16] B. Christensen, "Optimizing the netflix api," 2013. Online; accessed 1 Nov. 2017.

[17] P. Pietzuch, G. Mhl, and L. Fiege, "Distributed event-based systems: An emerging community," *IEEE Distributed Systems*, vol. 8, 2017.

[18] M. Overeem, M. Spoor, and S. Jansen, "The dark side of event sourcing: Managing data conversion," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017.

[19] M. Fowler, "Event sourcing," 2005. Online; accessed 14 Nov. 2017.

[20] D. Yang and J. Cao, "A scalable data warehouse model based on complex semantic event processing in distributed systems," *IEEE 28th International Conference on Data Engineering Workshops*, 2012.

[21] R. Manifesto, "Reactive manifesto v2.0," 2014. Online; accessed 14 Nov. 2017.

[22] A. Debski, B. Szczepanik, and M. Malawski, "In search for a scalable and reactive architecture of a cloud application: Cqrs and event sourcing case study," *IEEE Software (Not yet published 13.11.17)*, 2017.

[23] J. V. Vasques. Online; accessed 27 Aug. 2019.

[24] A. Mikkelsen, T.-M. Grønli, and R. Kazman, "Immutable infrastructure calls for immutable architecture," in *52nd Hawaii International Conference on System Sciences*, 2019.

[25] E. Gamma, J. Vlissides, R. Helm, and R. Johnson, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[26] A. Balalaie, A. Heydarnoori, P. Jamshidi, D. A. Tamburri, and T. Lynn, "Microservices migration patterns.," *Softw., Pract. Exper.*, vol. 48, no. 11, pp. 2019–2042, 2018.

[27] R. Kazman, M. Klein, and P. Clements, "Atam: Method for architecture evaluation," Tech. Rep. CMU/SEI-2000-TR-004, Carnegie Mellon Uiversity, Software Engineering Institute, 2000.

[28] K. Krippendorff, *Content Analysis: An Introduction to Its Methodology (second edition)*. Sage Publications, 2004.

[29] Apache, "Kafka," 2017. Online; accessed 16 Nov. 2017.

[30] S. Newman, "Building microservices," *O'Reilly Media*, 2015.