2019 IEEE/ACM 2nd International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)

Architecture Interoperability and Repeatability with Microservices: an Industry Perspective

Eric Yuan
Aerospace Corporation
Chantilly, VA
eric.yuan@aero.org

***Abstract*—Microservices, along with supporting technologies such as containers, have become a prevalent architecture approach for today's software systems, especially in enterprise environments. They represent the latest evolutionary step in the decades-old journey towards service- and component-based software architectures. Along with virtualization technologies, microservices have enabled the loose-coupling of both service *interfaces* (message passing) and service *integration* (form and fit). This paper attempts to explore the impact of microservices on software architecture interoperability and repeatability, based on our experiences in developing two microservice-based systems. Our central thesis is that, if we view software architecture as a set of principal design decisions, the microservices approach enable us to more elegantly separate these decisions from non-architectural, domain-specific ones, and thus make these decisions more interoperable, reusable, and repeatable across disparate problem domains. We therefore propose that a microservices based *reference architecture* (RA) and *reference implementation* (RI) be created for the community-wide infrastructure for software engineering and software architecture research, along with a set of detailed considerations.**

***Index Terms*—microservice, software architecture, DevOps, cloud computing**

I. Introduction

In the software industry, the growing popularity of microservices as a new architecture and design approach is hard to miss. Recent academic studies also confirms this trend [14], [23]. In many ways microservices can be seen as another evolutionary step towards service-based and component-based architecture styles, in the sense that each microservice implements a small, well-scoped piece of functionality with a single business or mission purpose [21]. The prevalent and rapid adoption of microservices, however, has a deeper root beyond more fine-grained modularity. Our own experiences show that the success of the microservices approach lies in the fact that it not only achieves the loose-coupling of software components via loosely coupled service *interfaces*, such as via RESTful APIs, but also is able to address the "*form and fit*" of the components, via modern encapsulation mechanisms such as virtualized containers. The latter offers the much-needed isolation of the components both *statically* (at time of development and deployment) and *dynamically* (at run-time), which traditional service-oriented architecture (SOA) implementations were not able to achieve.

This paper presents two case studies of real-world microservice based systems that we successfully implemented, and draws observations from our experiences and lessons. One case study is for a polyglot modeling and simulation platform for internal use, implemented in a private (on-premise) cloud. The other is a transportation data clearinghouse for exchanging data to and from intelligent vehicles in near real-time, delivered in the Amazon Web Services (AWS) public cloud for a US Government customer. Both are based on the Kubernetes container orchestration platform [6]. In the two projects we experienced first-hand the natural synergies of microservices architecture, decentralized Agile software development methodology, and DevOps best practices. From the software architecture perspective, we also started to see that from a software architecture perspective, there are more to microservices than meets the eye. In particular, we observe that:

- While microservices easily led to component-level reuse, a micro-services based architecture may be reused and repeated *wholesale*, resulting in architecture-level reuse;
- This is partly due to the fact that many non-functional concerns such as security, reliability and availability are decoupled and *externalized* to outside of the microservices;
- Infrastructure-as-code mechanisms are capable of codifying and enforcing basic architecture attributes and constraints to a level that traditional model-driven engineering (MDE) approaches were not able to attain, further boosting "wholesale" reuse at the architecture level.

These observations point to a deeper reason of the success of the microservices as an architectural approach – proper separation of concerns. Compared with making architectural decisions within the bounds of a monolithic system, microservices force a system to treat its architecture at two levels: the "micro" level for the intra-service design and the "macro" level for the collective and emergent behavior of participating microservices.

These insights led us to propose that the microservices approach be considered a key element to a community-wide research infrastructure. Its key benefits include enhanced tool reusability and interoperability, architecture repeatability, and instrumentability.

The rest of the paper is organized as follows. Section II provides further background on containerized microservices and related research work. Sections III and IV present the two

978-1-7281-2241-0/19/$31.00 ©2019 IEEE
DOI 10.1109/ECASE.2019.00013

case studies using cloud-native microservice implementations, respectively. Based on the observations from these development projects, section VI proposes a set of considerations and recommendations for building a community infrastructure for software architecture research.

II. Background

A microservice is "a small application that can be deployed independently, scaled independently, and tested independently and that has a single responsibility" [24]. Microservices started as an industry phenomenon but have started to gain academic interest. Current research efforts include strategies for migration away from monoliths [17]; applying existing architecture approaches to microservices, such as model-based engineering [13]; and exploring the natural synergy between microservices and DevOps and continuous integration / continuous delivery (CI / CD) [14].

Why microservices? Much has been said about the benefits of microservices such as improved modularity [12] and deployability [15], which enables independent development in parallel teams with the ability to release incrementally and individually. It is worthwhile to consider just exactly how such benefits are achieved. From the industry perspective, we believe an important part of the answer lies in the fact that microservices are not only a logical construct, but a physical one as well – it is a build unit, a deployment unit, and a runtime unit with its own dedicated compute resources (virtual or physical). In today's industry landscape, microservices have increasingly taken up containerized implementations, which is a form of lightweight virtualization, allowing each service or application to execute in its own isolated environment at runtime, even though it may still be sharing the same OS kernel with other applications. Unikernel applications appear to hold the promise of making microservices form factors even more compact [18].

The container-based form factor offers some key advantages:

- *Statically*, containers serve as a packaging mechanism (e.g., a Docker image) such that each microservice becomes an individually buildable and deployable unit – a component in its original true sense [16]. This greatly simplifies application integration and dependency management during the development lifecycle. Compared with a monolith application that can only be tested and released once several months, microservices support piecemeal, continuous delivery process.
- *Dynamically*, containers isolates the runtime environment of each microservice, including configurations, library dependencies and hardware resources, which results in greatly reduced global configuration complexity. As recent research shows, tight coupling of components directly impact maintainability [20].

Even though containers are not the only way to instantiate microservices, this paper discusses microservices architecture in the context of this prevalent form, and hope to extrapolate some deeper themes that can help shape a community-wide infrastructure for software architecture research.

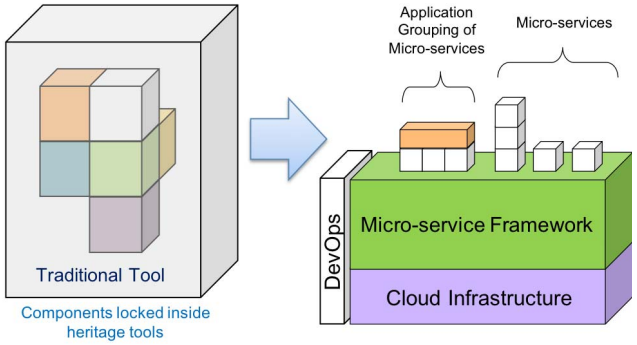

Fig. 1. FLAME Architecture Vision

III. Case Study 1: A Microservices-based Engineering Analysis Platform

A. Project Background

As a government-funded research and development center, our company has a rich heritage of performing aerospace engineering and analysis for the public sector. Currently, models and algorithms that embody corporate knowledge are difficult to discover, reuse, and integrate across projects, for a variety of reasons. First, our traditional tools are monolithic and not interoperable – in fact, most don't work outside of desktop workstations where they are installed. Useful functionality is buried inside these tools. Secondly, code and algorithms were written in different languages (C/C++, Python, Fortran, etc.) and environments (such as Windows, Linux, or HPC clusters) and are difficult to integrate. Perhaps the biggest obstacle of all is that engineers are trained to solve problems in their respective engineering disciplines, not in writing good software, let alone having software architecture foresight. As a result, most software implementations are ad-hoc and not fit for reuse. Answering new customer questions requires too much time to code new implementations for each project, which leads to slow response and high labor cost.

To change the status quo, we undertook a pilot initiative called Flexible Language-Agnostic Analysis and Modeling Environment (FLAME). The effort seeks to provide aerospace engineers with a Platform-as-a-Service (PaaS) environment that enables interoperability and sharing of model components, runs on any commodity computing platform, and lets developers continue to code in the language most comfortable and familiar to them. FLAME also employs modern software engineering methods to allow for Agile and collaborative development, automated build and testing, and continuous deployment.

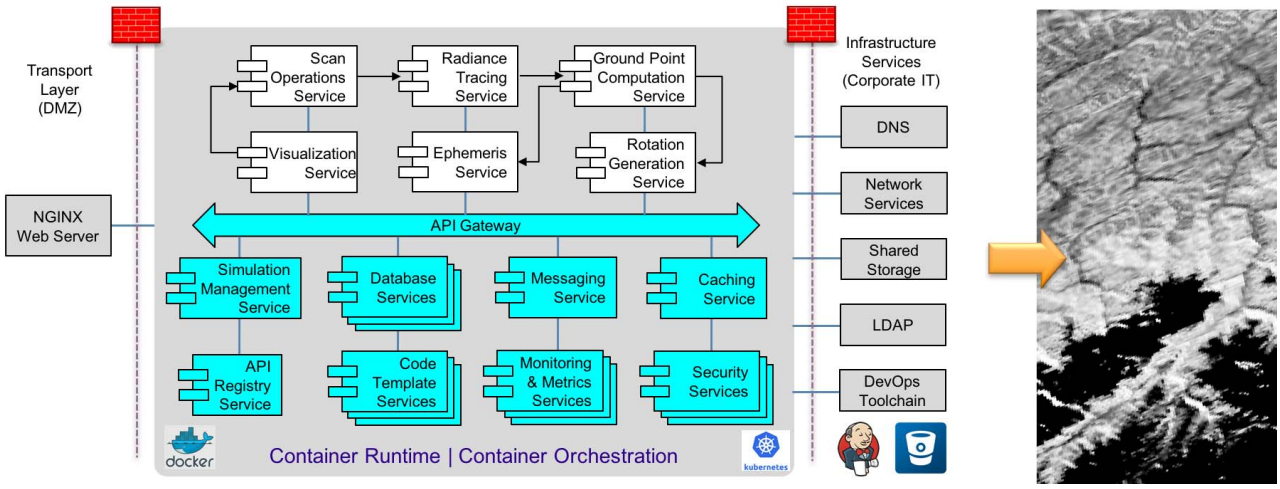

Fig. 2. FLAME Microservices Architecture

B. Architecture and Design

The architectural vision of FLAME is depicted in Fig. 1. We adopted the containerized microservices approach to decompose existing tools into independent components, each exposed via RESTful APIs and standard data formats (such as JSON). To support heterogeneous infrastructures including both on-premise hardware and public cloud service providers, we chose to use the Kubernetes container orchestration framework [6] that provides container load-balancing and elastic scalability, among other features.

FLAME also provided a DevOps tool chain that streamlines the processes from build, unit-test, containerization, container image publishing, to deployment into Kubernetes. To further reduce the burden of domain software engineers, we provided a set of code generation tools that can produce language-specific skeleton code from the RESTful API specification (using Swagger OpenAPI format [9]). A developer only needs to plug in the business logic (such as a C++ function) into the generated microservice wrapper.

C. Implementation and Results

To demonstrate the utility of the FLAME platform, we chose a use case based on a real customer need: space-based imaging performance assessment. It is a multi-disciplinary problem involving both sensor and radiometric modeling and vehicle platform motion assessment. The integrated analysis requires high-fidelity satellite platform motion models, projecting ground points to an imaging sensor, and applying instrument physical models. This is a complex modeling task that involves multiple computationally intensive steps, and had not been done before in an end-to-end integrated fashion. With modern cloud computing technologies, the FLAME platform makes it possible to modularize analysis and modeling capabilities as microservices that are integrated on-demand in a plug-and-play architecture. We were able to complete end-to-end simulation that produced a geometrically accurate image of what the scanning sensor would collect on orbit. The demonstration was implemented with a collection of microservices written in different programming languages (including C++, Java, Python, and JavaScript). The high-level architecture and the resulting image is shown in Fig. 2.

As shown in the architecture diagram, the transport layer on the left serves as the external interface of the system, using an NGINX web server to handle user requests. All other FLAME components are packaged as "dockerized" microservices running in the Kubernetes cluster. Kubernetes enables scaling-up / scaling-down of computational resources on demand – in fact, each microservice has become *individually scalable*, allowing us to address performance bottlenecks more effectively. The FLAME environment also leverages some corporately provided IT services such as DNS, LDAP, and DevOps tools, shown in the right side of the diagram.

During the implementation, our core team used open source software frameworks to implement a set of backend components such as WS02 API Gateway, Redis for data caching, etc. Concurrently, software engineers from other departments contributed other services using the programming language of their choice, such as the San Operations Service using Python and Ephemeris Service using C++. This kind of the collaborative and "polyglot" software development is hard to imagine in monolithic system environments.

Note also that in this highly computation-intensive problem scenario, the generation of every single pixel in the image may involve over 10^4 microservice calls. With the elastic and fault-tolerant FLAME architecture, we are able to perform a long-running simulation (sometimes over days) despite sporadic software and hardware failures without having to restart from the beginning.

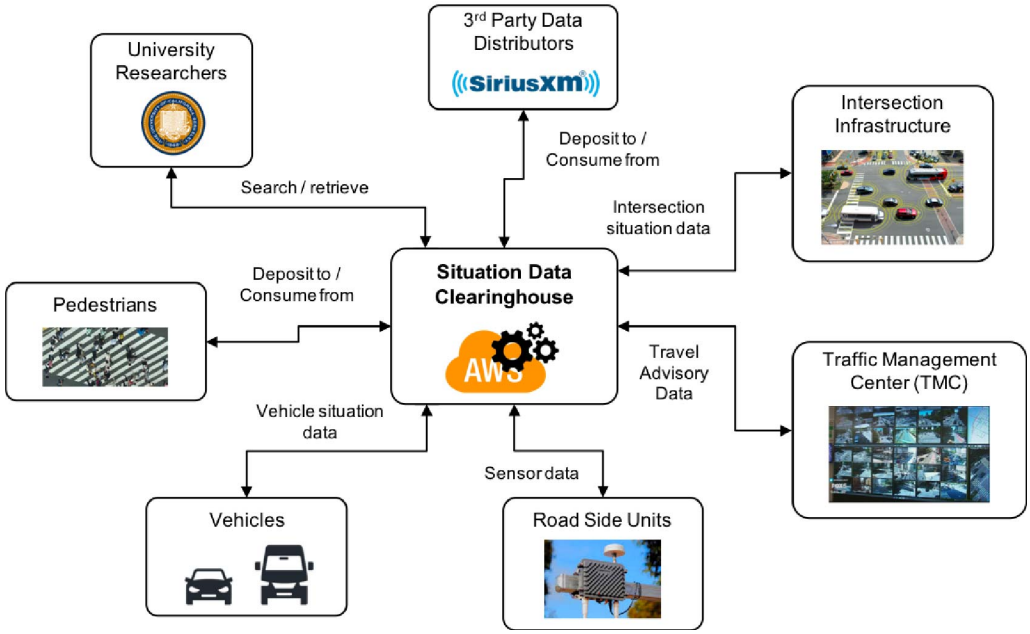

Fig. 3. SDC Context Diagram

IV. Case Study 2: A Microservices-based Data Clearinghouse for Intelligent Vehicles

A. Project Background

The US Department of Transportation (USDOT) has been pushing for intelligent transportation systems that digitally integrate intelligent vehicles and intelligent infrastructure. Such systems can serve many important use cases, such as improving driver safety by providing real-time situation awareness of road conditions. Connected Vehicles (CV) and Autonomous Vehicles (AV), for example, may use short-range radio signals to communicate with one another as well as with smart roadside units (RSU); cars would receive notifications of dangerous situations such as unsafe lane changes or traffic violations of nearby cars. With over 30,000 fatalities on the road every year, the USDOT is shifting its focus from helping people survive crashes towards preventing crashes from happening in the first place [3]. As a concrete example, one of the CV pilot programs involves the busy Highway I-80 in Wyoming, where extreme weather conditions including blowing snow in winter and fog and high winds in summer create dangerous conditions for truck drivers [11].

With increased connectivity among vehicles, organizations, systems, and people, unprecedented amounts of data are being generated. USDOT has identified a potential need to collect, fuse, and repackage operational data from disparate CV and AV deployments and sources for dissemination at a national level. A Situation Data Clearinghouse (SDC) system has been envisioned to provide data collection, fusion, transformation, and distribution capabilities across various data providers, distributors, and consumers. Fig. 3 depicts the system's operational context.

The key requirements of SDC include:

- flexible, open architecture and interoperability to enable data sharing and integration with various third party devices and applications;
- near real-time performance, low latency message exchanges;
- scalable on-demand to support regional and national level operations;
- low cost, incremental delivery using open source software and Agile development methodologies.

An earlier prototype of SDC demonstrated its utility but was implemented as a proprietary system that was tedious to maintain and difficult to integrate. Through a fortuitous opportunity, the project team learned about the FLAME microservices platform and thought that its modular, cloud-native architecture could serve as the foundation of a modernized SDC.

B. Architecture and Design

The SDC architecture, built on the domain-independent components of the FLAME platform, is depicted in Fig. 4. The new services are highlighted in red font. Compare with the FLAME architecture in Fig. 2, note the following changes:

- The transport layer now offers a more diverse set of mechanisms for inbound and outbound data exchanges. Vehicles and RSUs typically send data packages via Dedicated Short Range Communications (DSRC) messages over the UDP protocol due to unstable connections and limited bandwidth, whereas traffic management centers and other enterprise systems may use RESTful APIs or websockets streaming over TCP for better throughput.

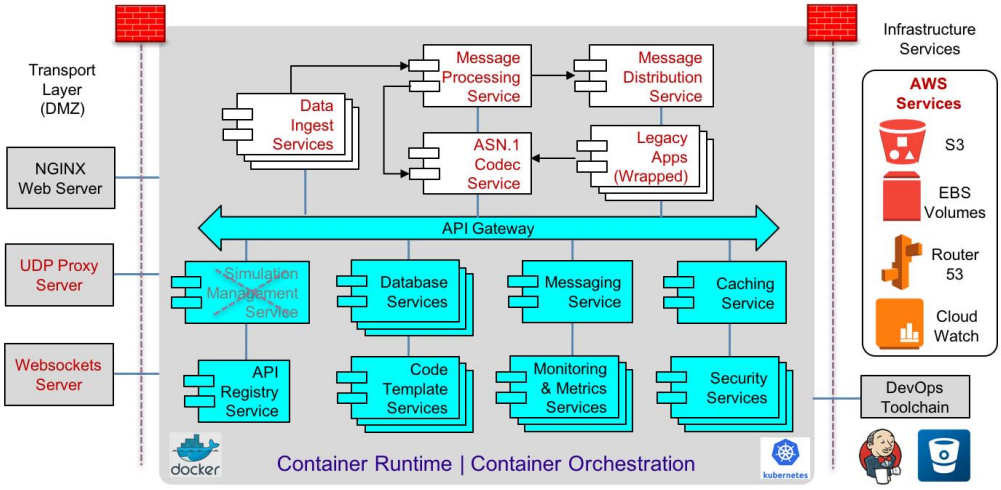

Fig. 4. SDC Architecture

Regardless of the external interface, situational data is encoded in SAE J2735 message formats, an industry standard. Messages may also be digitally signed and encrypted to ensure integrity and confidentiality.

- The backend architecture of the SDC is deployed within a Kubernetes cluster, on the foundation of the FLAME services described in the previous section, with the exception of the Simulation Management Service which is not applicable in this problem context. The new domain-specific microservices include data ingest services implemented based Apache NIFI [1], along with additional message handling services for vehicle data. We were also able to wrap software modules from the legacy SDC prototype system as microservices, with little modification. Just like in the FLAME project, these microservices are also accessible via RESTful interfaces, individually deployed and scalable, and continuously delivered via the DevOps pipeline.
- Unlike the original FLAME that was implemented in a private cloud, the SDC system is deployed in the AWS public cloud, therefore can take advantage of additional AWS service offerings such as S3 object storage and CloudWatch monitoring, shown on the right in Fig.4. The use of AWS services, however, are transparent to the microservices inside of the Kubernetes cluster – they are for the most part agnostic of the underlying infrastructure.

C. Implementation and Results

The initial release, also called the Minimally Viable Product (MVP), is scoped to facilitate an end-to-end data flow for the aforementioned Wyoming CV Pilot – Traveler Advisory messages from Wyoming DOT systems are ingested into SDC, decoded, validated, packaged and distributed to SiriusXM^{TM} [8], which in turn delivers the messages to vehicles via satellite communications. Due to the deliberate reuse of the FLAME platform, the delivery of the MVP into production took only 6 weeks, well ahead of schedule, and has so far maintained over 99% uptime.

V. Observations and Discussions

It is always exciting to see emerging software technologies respond to customer needs quickly and make a real-world impact. At the same time, our team also gained valuable insights into the microservices approach, especially from the software architecture perspective. Here we offer some key observations.

A. Continuous Architecting

First and foremost, during the course of the two projects software architecture was never an upfront activity early in the lifecycle, but something that continually evolve along with the system — services were frequently added and upgraded, workflows were being altered, and deployment topology kept being refined. The pace was accelerated by the DevOps practices and tools. This appears to be the norm for microservices-based architectures.

We saw a real need for modeling tools that can keep architecture definitions up to date and in sync with the running system, and are in the process of implementing an architecture reconstruction tool that can accurately depict service interactions within the Kubernetes cluster.

B. From Component Reuse to Architectural Repeatability

With well scoped functionality and containerized "form factor", a microservice easily leads to *component-level reuse*. Compared with reusing a Java library, which requires code-level integration and compatibility (e.g. JDK version, class-paths, data structures, etc.), reusing a microservice is as simple as downloading and deploying a container image from the registry, and interacting it through RESTful APIs. One of our design goals for the FLAME project was to make all microservices discoverable and accessible, so that other

engineers can make use of, say, an ephemeris calculation service without the need to understand its inner workings – in fact they do not even need to know what language it was written in.

The reuse story didn't just end there. The FLAME platform's overall architecture turned out to be largely domain independent, allowing it to be instantiated for entirely different problem domains. After the project started, it soon became evident to us that several components are not specific to the simulated image assessment use case at all, such as:

- messaging services based on Kafka, for distributed and asynchronous inter-service communications.
- parameter management service, for managing microservice instance-level configurations. This service can be used to manage the internal state of microservices (e.g. orbit of the spacecraft, attitude of the payload sensor, characteristics of the imaging chip, etc.) so that each microservice instance can be made stateless, following the 12-factor design principles [10].
- monitoring and metric service using Elasticsearch, for service status and health.
- in-memory database service using Redis, for caching interim computation results.

These services (together with other utilities and templates) formed the foundation of our technical strategy for the SDC system for intelligent vehicles, saving the development team about 50% of effort and time — this despite the fact that these services were developed for an an entirely different industry. Our SDC experience has demonstrated that with a microservice-based architecture, component reuse can go to a new level, that is, from ad-hoc, fortuitous to "wholesale" reuse and *architectural repeatability*, as shown in Fig. 5.

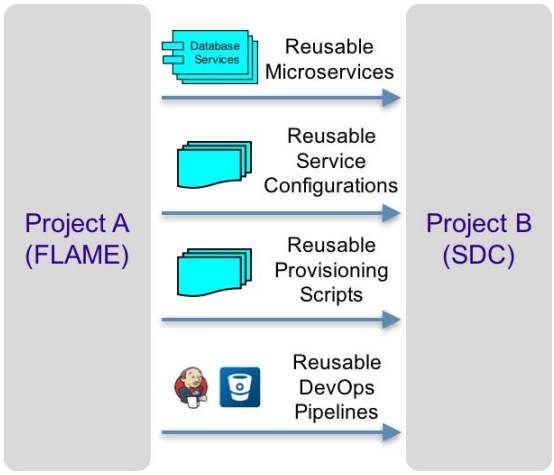

Fig. 5. Reuse and Repeatability at Architectural Level

This was quite an unexpected (but pleasant) outcome for the original team. We are currently applying the same architecture to jump start a number of other customer efforts at NASA and NOAA. Many enabling factors made it possible, including the use of a well-engineered container orchestration framework such as Kubernetes and modern DevOps tooling, but we believe the most crucial factor is that containerized microservices establish a natural boundary that separates the system into *macro* and *micro* level concerns, which leads to our next observation.

C. Externalization of Architectural Concerns

During the course of FLAME and SDC efforts, the software architect's attention has been almost entirely on the integration and interaction of services, while domain-specific software engineers from various departments focused on implementing the services themselves. Why? Because those microservices were primarily concerned with implementing *functional* requirements. In their statically and dynamically isolated design space, design decisions (either good or bad) have very limited impact to the overall system. On the other hand, typical quality attributes of the architecture – availability, scalability, security to name a few – have largely become the responsibilities of the outside framework, which can address them uniformly and more efficiently. This naturally becomes the center of attention for the software architect. In other words, we see that microservices effectively *externalize* architecture concerns.

To take a closer look, we use the SOA quality attributes taken from [22] and examine how they are implemented in the FLAME environment. As Table I shows, these attributes were satisfied by the underlying PaaS frameworks and supporting processes, rather than in the microservice code.

TABLE I
QUALITY ATTRIBUTES ADDRESSED IN FLAME ARCHITECTURE

Attribute	FLAME Approach
Interoperability	RESTful APIs for microservices are defined and managed by Swagger; FLAME further provides code generation tools for the service wrapper
Performance, Scalability and Availability	Elastic scalability, fail-over and load balancing provided by Kubernetes based on declarative policies; AWS auto-scaling also used
Security	Built-in network security in Kubernetes; WSO2 API Gateway provide fine-grained access control
Modifiability	Independent microservice deployment; CI/CD processes; zero downtime rolling upgrade
Testability	Automated testing enabled by Jenkins CI tool; end-to-end testability remains a challenge
Usability	User experience managed by dashboards and metrics services
SLAs, Lifecycle Management	SLAs managed by API Gateway; DevOps tool chain automates microservice lifecycle mgmt.

Recall that in a traditional SOA, a service developer usually need to pay attention to these quality attributes, such as the integration with WS-Transaction for messaging reliability, or application server clustering for high availability, etc. With microservices-style componentization coupled with a robust container management framework, these attributes are largely addressed outside (and therefore transparent to) the services. The industry refers to this as the "Inversion of Control" principle — that is, a service no longer attempts to be the "main" program in control of the overall system; on the contrary, it is now written with the assumption that

the encompassing framework or platform is responsible for how to instantiate and use it. It is our opinion that being able to successfully implement the Separation of Concerns and Inversion of Control principles marks a rather profound difference between the traditional SOA and the microservices styles, and partly explains the latter's resounding success.

With the externalization of the architecture concerns, we further hypothesize that the microservices approach enables a software architecture dichotomy at *macro* and *micro* levels, with the former focusing on domain-independent, non-functional traits, whereas leaving functional and domain-specific implementations to the latter. Fig. 6 depicts this two-level construct.

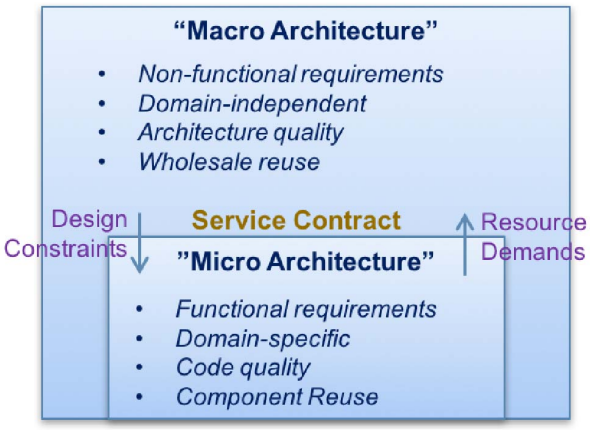

Fig. 6. Macro vs. Micro Software Architectures

At the macro level, the architecture is concerned with meeting the quality attributes or non-functional requirements for the system, such as those listed in Table I. As shown in our first-hand experiences, the strategies, patterns and tactics employed at the macro level are often domain-independent. For instance, microservices architectures are inherently distributed; all systems need to address common challenges such as consistency, latency, deployment topology, etc. Such issues need to addressed regardless of the industry or problem domain. As evidence, at trade conferences we often see microservices architectures from diverse industries with striking similarity. The domain-neutrality of macro architectures lead to two appealing benefits: first, best practices and patterns can be codified and automatically reapplied elsewhere, leading to systematic, wholesale reuse of frameworks and platforms. Second, solutions that address common architectural concerns have far-reaching impact and huge market potential, providing strong incentives for innovation. The emerging service mesh technologies (such as Istio [5] and Envoy [4]) is a good example.

At the micro level, software architecture should focus on strategy and tactics that efficiently implement functional, domain-specific requirements, and doing so within resource limits and design constraints dictated by the macro architecture. Our experiences show that uniform design constraints and implementation diversity can coexist – for example, while satisfying globally enforced design rules such as RESTful APIs, stateless design and standard logging levels, the developer of each microservice was able to choose his/her own preferred OS version, programming language or third-party libraries without impacting other services. While architecture quality is the central concern at the macro level, developers of a microservice focus on code quality, who can readily use existing tools such as static code analyzers and profilers.

Note that the macro vs. micro separation is not absolute. Even macro architecture may still take into account the domain or industry specific requirements – for instance, the aerospace industry generally puts a premium on reliability and availability over modifiability and usability. Likewise, a microservice still needs pay attention to non-functional requirements, such as following good coding practices and design patterns, avoid security vulnerabilities including command injection and buffer overflow, etc.

The advantage of the macro-micro separation, however, is that micro-level architectures can be treated essentially as blackboxes as far as overall architecture quality is concerned. Even some design flaws at the micro level may be compensated at the macro level. For example, during early FLAME test runs a microservice with memory leaks became less and less responsive over the course of time. The liveliness probes that we configured as part of the Kubernetes service proxy detected this anomaly, and started killing the sluggish instances and spawning new ones, allowing the overall simulation run to continue, albeit with degraded performance.

D. Architecture as Code

The final observation is somewhat related to the previous ones. As we made preparations to port FLAME from our private cloud to AWS, one team member went the extra mile of creating CloudFormation [2] templates to make the process repeatable. A closer look at the template, a software architect may find it eerily similar to some form of an Architecture Description Language – e.g., logical components, their relationships, and mapping to physical components. It is hardly a user-friendly ADL for human eyes, but most definitely a machine-readable and machine-executable one. CloudFormation is only one example of the flourishing family of so-called "infrastructure as code" tools emerging from the industry; other popular peers include Puppet, Chef and Ansible.

By codifying architecture information and artifacts in machine-readable formats, we can further enhance architectural interoperability and repeatability. One can simply execute the scripts or "recipes" to reuse or repeat a system's deployment process, as opposed to manually following its documentation.

One potential research opportunity is to see how to bridge the gap between industry and academia, and combine academic endeavors around model-driven engineering and executable architectures with industry's powerful tools for cloud provisioning and management.

VI. Considerations for a Community-Wide Infrastructure for Software Engineering Research

Microservices represent the latest evolutionary step in the decades-old journey towards service-based software architectures. Along with virtualization technologies, microservices have enabled the loose-coupling of both service *interfaces* (message passing) and service *integration* (form and fit). One may argue that we are closer than ever before in achieving a truly component based architecture.

This paper attempts to explore the impact of microservices on software architecture theory and practice, from a practitioner's perspective. From our experiences in developing two microservice-based systems, we have come to the realization that, if we view software architecture as a set of principal design decisions [19], the microservices approach enable us to more elegantly separate these decisions from non-architectural, domain-specific ones, and thus make these decisions more interoperable, reusable, and repeatable across disparate problem domains.

Therefore, we propose that a microservices based *reference architecture* (RA) (of which [7] is an example) and *reference implementation* (RI) be created for the community-wide infrastructure for software engineering / software architecture research, or at least be considered as one of the viable approaches.

Specifically, the RA/RI may provide the following benefits:

- *Tool Reusability* — Research tools from the community will be packaged as discoverable and API-accessible microservices, boosting their reusability;
- *Tool Interoperability* — Containerized microservices allow different researchers to use their programming language of choice, yet able to integrate their components together in unexpected ways in a common community-operated environment;
- *Architecture as Code* — Prescriptive architecture decisions and artifacts, such as structural relationships, the implementation of a quality attribute, or an architecture pattern, can be codified using machine-readable and executable scripts using modern automation tools, as discussed in Section V-D, making them reusable and repeatable. This is a meaningful step towards evolving architecture descriptions from human-centric to machine-understandable artifacts [25];
- *Instrumentability* — Modern cloud computing infrastructures, where microservices thrive in, provide common, open mechanisms for component-level observability and instrumentation. From logging tools such as logstash and fluentd, to container "sidecars" that are deployed in conjunction to the primary service (think tools such as Istio [5] and Envoy [4]), architecture concerns can be observed and managed transparently to the microservice of interest, with little or no performance impact. This not only allows open, repeatable implementation of architecture-based software engineering approaches (e.g. an open, interoperable MAPE-K framework), but also enables easy architecture recovery and comparison between prescriptive and descriptive architectures.

References

[1] Apache Nifi. http://nifi.apache.org/.
[2] AWS CloudFormation. https://aws.amazon.com/cloudformation/.
[3] Connected Vehicle Basics. https://www.its.dot.gov/cv_basics/cv_basics_what.htm.
[4] Envoy service proxy. https://www.envoyproxy.io/.
[5] Istio service mesh. https://istio.io/.
[6] Kubernetes container orchestration framework. https://kubernetes.io/.
[7] Reference architecture foundation for SOA. http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html.
[8] SiriusXM. https://www.siriusxm.com/.
[9] Swager OpenAPI specification. https://swagger.io/docs/specification/about/.
[10] The twelve factor app. https://12factor.net/processes.
[11] WYDOT CV Pilot. https://wydotcvp.wyoroad.info/.
[12] N. Alshuqayran, N. Ali, and R. Evans. A systematic mapping study in microservice architecture. In *Service-Oriented Computing and Applications (SOCA), 2016 IEEE 9th International Conference on*, pages 44–51. IEEE, 2016.
[13] N. Alshuqayran, N. Ali, and R. Evans. Towards micro service architecture recovery: An empirical study. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 47–4709. IEEE, 2018.
[14] A. Balalaie, A. Heydarnoori, and P. Jamshidi. Microservices architecture enables devops: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.
[15] L. Chen. Microservices: Architecting for continuous delivery and devops. In *IEEE International Conference on Software Architecture (ICSA)*, 2018.
[16] B. Councill and G. T. Heineman. Definition of a software component and its elements. *Component-based software engineering: putting the pieces together*, pages 5–19, 2001.
[17] P. Di Francesco, P. Lago, and I. Malavolta. Migrating towards microservice architectures: an industrial survey. In *2018 IEEE International Conference on Software Architecture (ICSA)*, pages 29–2909. IEEE, 2018.
[18] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Acm Sigplan Notices*, volume 48, pages 461–472. ACM, 2013.
[19] N. Medvidovic and R. N. Taylor. Software architecture: foundations, theory, and practice. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 471–472. ACM, 2010.
[20] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Decoupling level: a new metric for architectural maintenance complexity. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 499–510. IEEE, 2016.
[21] S. Newman. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.", 2015.
[22] L. O'Brien, P. Merson, and L. Bass. Quality attributes for service-oriented architectures. In *Proceedings of the international Workshop on Systems Development in SOA Environments*, page 3. IEEE Computer Society, 2007.
[23] C. Pahl and P. Jamshidi. Microservices: A systematic mapping study. In *CLOSER (1)*, pages 137–146, 2016.
[24] J. Thönes. Microservices. *IEEE software*, 32(1):116–116, 2015.
[25] E. Yuan. Towards ontology-based software architecture representations. In *Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE), 2017 IEEE/ACM 1st International Workshop on*, pages 21–27. IEEE, 2017.