Cloud Native Software Engineering

Brian S. Mitchell
Department of Computer Science
College of Computing and Informatics
Drexel University, Philadelphia, PA, USA
bmitchell@drexel.edu

***Abstract*—Cloud compute adoption has been growing since its inception in the early 2000's with estimates that the size of this market in terms of worldwide spend will increase from $700 billion in 2021 to $1.3 trillion in 2025 [1]. While there is a significant research activity in many areas of cloud computing technologies, we see little attention being paid to advancing software engineering practices needed to support the current and next generation of cloud native applications. By cloud native, we mean software that is designed and built specifically for deployment to a modern cloud platform. This paper frames the landscape of Cloud Native Software Engineering from a practitioners standpoint, and identifies several software engineering research opportunities that should be investigated. We cover specific engineering challenges associated with software architectures commonly used in cloud applications along with incremental challenges that are expected with emerging IoT/Edge computing use cases.**

I. Introduction and Context

Delivering managed computing services on hosted infrastructure started in the late 1990's with the introduction of the Software-as-a-Service (SaaS) model. One of the early pioneers of this model was Salesforce.com [2], which launched in 1999. Unlike other companies that licensed software deployed on customer-owned equipment, SaaS companies provide a pay-as-you-go subscription model. In this model, they manage all of the software and compute infrastructure, you pay a monthly charge that entitles access to the solution from any device at any time.

While SaaS solutions marked the start of shifting software license spend to usage-based spend, public cloud computing as we know it today can be attributed to the launch of AWS (Amazon) [3] in early 2006, with Azure (Microsoft) [4] and GCP (Google) [5] following in 2008. The primary early adopters of cloud computing were technology companies that innovated patterns, practices, and open sourced tools and frameworks that have become best practices for running resilient and scalable business services in the public cloud. Over the past 10 years cloud computing has been growing in organizations of all sizes across many different industries.

Larry Wall, the creator of Perl, once stated – *There is a saying in the software design industry: "Good. Fast. Cheap. Pick two."*. Software engineering involves making difficult decisions based on informed tradeoffs. For example, it would not be hard to argue that in order to move faster and build things cheaper, compromises on software features, software quality, and/or security would be required. Using the utility of the cloud, coupled with modern cloud computing tooling, one can now argue that you can build better software faster and cheaper. It's not that Larry Wall's insights were incorrect, but we can now have the technologies and practices to redefine *good* in terms of *fast* using the cloud. When computing components are deployed to the cloud, the simplest way (and thus the most popular way) to do this is via automation [4], [6]–[8]. The *automate everything* practice embraced by cloud computing not only allows deployments to be fast, but it also favors ephemeral computing components. These components by their nature are easier to test [9] and can be started, stopped, paused, or replaced at any time.

This combination of capabilities enables software engineers to rapidly deploy software to a known state at any time. With these building blocks new well-tested features can be quickly and consistently rolled out to users in very small batches. Goodness of the solution can now be validated via feedback from users, either directly, or via monitoring and instrumentation of their behavior. These cloud enabled capabilities have the potential to advance software engineering practices in many ways, but transforming these practices across the entire community comes with many challenges. We think this represents a significant opportunity for the software engineering field given the likelihood that most industrial systems moving forward will be deployed on cloud runtimes[1]. Specifically:

1) Helping Software Engineers manage the expanded cognitive load required to design, build, deploy and operate at scale applications in the cloud. We will discuss this throughout the remaining sections of this paper.
2) Identify opportunities to accelerate and scale software engineering skillsets needed to deploy a broader suite of applications to the cloud. Many organizations will want to move beyond deploying externally facing web and mobile applications to the cloud using their top engineers. This will require developing new skills for the broader engineering organization as more of their core business moves to cloud computing.
3) Investigate how software engineering and computer science education can expand to address the demands of industry to create new, and retool existing software engineers for the cloud.[2] Most cloud-proficient software

[1] By cloud runtime we include public, private and hybrid cloud infrastructure

[2] We will talk about cloud certifications later, but they are targeted towards using the services of a cloud provider, not on the design and architecture of cloud native applications

engineers appear to build their skillsets on the job and with online resources versus in formalized academic programs.

4) Understand software engineering needs for new architectures enabled by the cloud. For example, IoT and smart devices that run at the edge increase the complexity of software engineering given the distributed nature of these platforms. These challenges will be discussed in Section III-B.
5) Address non-technical challenges that organizations face with adopting cloud-centric engineering best practices. Consider Google who published in 2021 [10] that they ran over 700K experiments in production that resulted in over 4K search product changes. Netflix open sourced tools [11] that they use for chaos testing to validate platform resiliency. Comfort with strategies that involve testing and randomly breaking things in production are embedded in the DNA of technical companies, but are often met with caution in traditional organizations.

We will address a number of these opportunities in the subsequent sections of this paper. The next section will introduce *Cloud Native* from a software engineering vantage point. Throughout this paper, by cloud native, we are referring to systems designed specifically to favor managed cloud platform services (PaaS/Faas) [12], and not systems that are *lifted and shifted* [13] from an on premise virtual machine to a virtual machine that runs in the cloud (IaaS).

II. What is Cloud Native Computing?

Before we explore the software engineering landscape for the cloud, we need to address exactly what we mean by cloud native computing. According to the Cloud Native Computing Foundation (CNCF) [14] *"Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds."*. Amazon's definition is *"Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds"*. Google offers the definition *"Cloud native means adapting to the many new possibilities—but very different set of architectural constraints—offered by the cloud compared to traditional on-premises infrastructure."*. The primary theme in these definitions centers around the role that cloud platforms play in enabling the creation of cloud native applications. They also don't clearly define "Cloud Native", which we consider any application that is specifically designed to be deployed on a cloud platform.

We think a better definition of cloud native computing that focuses more on software engineering is *"Cloud native applications are well architected systems, that are "container" packaged, and dynamically managed"*. Specifically:

Well Architected Systems - By this we mean systems that adhere not only to established software engineering best practices but also embrace specific functional and non-functional capabilities offered by the cloud. For example, how the computing components such as services/APIs are identified, how they work with each other, how security requirements are met, and how the system is designed for resiliency and scale?

Container Packaged - The term *container* is overloaded in the cloud computing terminology landscape. In many places its equated to a standardized package [15] that is managed by Docker [16] technologies - aka "a docker container". We take a more generic view of container packaging. Specifically, we think container packaging is a mechanism to package and deploy code that is ephemeral, can operate across a variety of different hardware architectures (e.g., Intel, ARM, microcontrollers, etc), and at runtime is supervised. Supervision includes full lifecycle management associated with version identification, startup, shutdown, health checks, security scanning, and monitoring. Examples of container packaging and supervision include Docker, Docker Compose, Kubernetes, and serverless [17]. We also include in this category the emerging interest with using server-side web assembly [18], [19] as a way to package and deploy cloud native application services.

Dynamically Managed - Consider the cloud as a large, highly distributed, special purpose operating system. Just like any operating system, there are a number of resources like storage, compute, network and security services that are needed by applications. The job of an operating system is to dynamically manage and optimize the allocation of these resources to the realtime computing demand on the overall system. When done well, every process being managed by the OS will perceive that it has access to the resources it needs, when it needs it. In a similar context, a cloud service provider, via Application Programming Interfaces (APIs), provides and manages resources to cloud native applications dynamically. Classical operating systems manage physical resources on a single system, whereas cloud resources are virtualized and distributed, while also being resilient and scalable. For example, block storage that supports virtual machine reads and writes are automatically replicated across servers in different special purpose data centers. Outside of initial configuration, the user does not worry about how durability is provided given its dynamically managed by the cloud service provider. Other examples include using auto scaling of virtual machines with health checks, or more advanced services like Kubernetes [20] that can scale up or down dynamically based on demand. Function as a service (FaaS) solution's take this a step further by running code on demand when a certain event happens. AWS even open sourced a micro VM called Firecracker [21] they created to support dynamically managing serverless workloads at scale.

Now that we have provided a definition, we describe next a number of interesting software engineering problems that warrant investigation.

Managing cloud native technical assets. In 2011 Adam Wiggins authored a set of technical principals that enable software engineers to create, manage and release code in support of cloud native applications. These principals were branded "The Twelve-Factor App" [22]. Over the years they were updated and revalidated [23], [24], but consistently hold

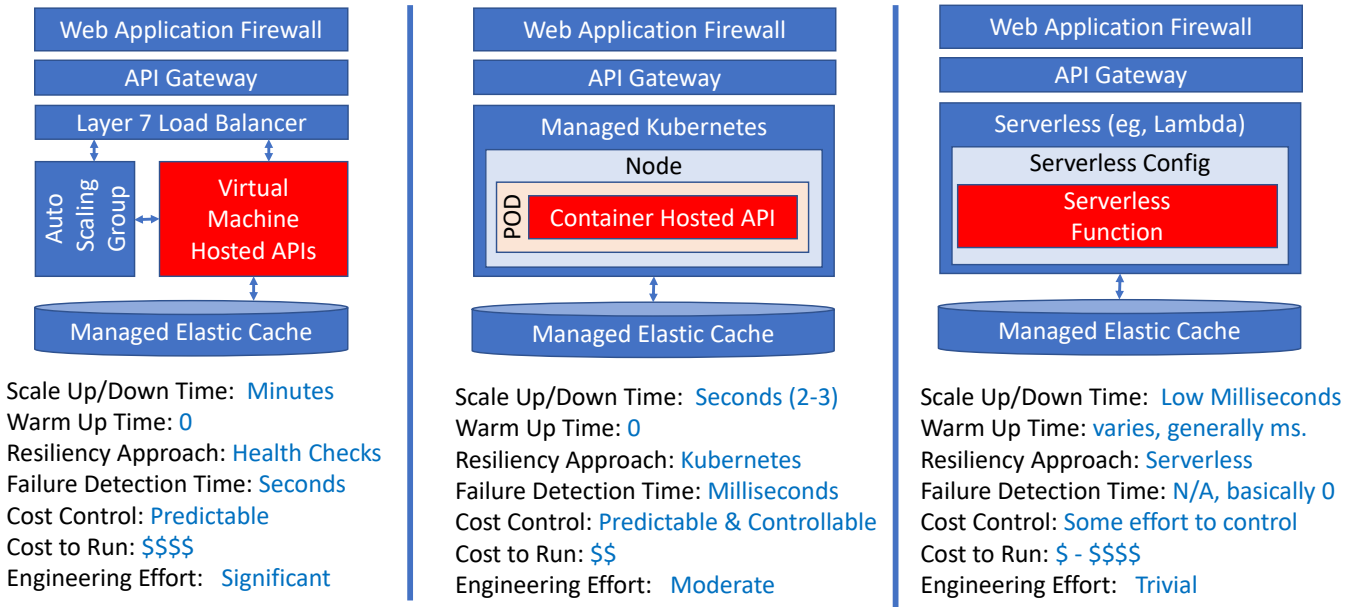

Fig. 1. Cloud Native Architectural Tradeoffs for a Hypothetical Suite of APIs

up as recommended engineering practices. One of the key challenges is that while none of these practices is overly complex, they require mastery and discipline. Also, existing standards enforced by enterprises might act as blockers to some or all of these practices. For example, some organizations might not have comfort or necessary controls to ensure that all deploys, to all environments, are driven through a source code control system.

Identifying an appropriate cloud native software architecture. The technical principals associated with 12 factor apps is a good start; however, these only focus on how to manage cloud native technical assets. Good cloud native solutions are generally architected as a suite of horizontally composable components. This model introduces several interesting challenges for cloud native software engineers that must be addressed. What are these components? How do identify them? What will guide how they should be built? We think some of the newer concepts around app meshes [25] and data meshes [26] provide a good start for shaping the overall architecture. As far as identifying the architecture components themselves, we think concepts from Domain Driven Design (DDD) [27], [28] can be refreshed to support this activity.

Upskilling Software Engineers on the use of Quality Attributes to make informed technology tradeoff decisions. Most cloud providers offer many different technology choices to create cloud native components. Applying discipline around quality attributes should guide making technology stack decisions. For example, Figure1 shows three different ways to deploy cloud native APIs along with some associated quality attributes. It should be noted, that the values of these quality attributes will change for different APIs, the figure assumes these hypothetical APIs are very light weight, event triggered, and manage their state via an elastic cache. Thus for the scenario shown in the figure, the VM option on the left does not make sense given the high cost, large scale up/down times, and complex engineering effort. The container option in the middle and the serverless option on the right both seem like good options. The ultimate decision will be driven by the desire to have a high degree of control over cost, and the nature of the workload. For example, if the expected traffic has massive near realtime spikes, a serverless solution might be preferred. If the workload is not event-driven, or has many runtime dependencies such as database connections, then a container based solution might be a better choice. As we move more towards computing at the edge, containers might not be possible since they depend on Linux kernel, so emerging lighter weight alternatives such as WebAssembly (WASM) with WASI [29] might be a good choice. Cloud native software engineers must be able to make these types of choices and resist over-standardizing based on personal or organizational preference and let software architecture practices using quality attributes guide cloud native platform choices.

Organizing teams for cloud native success. In 1967, Mel Conway published a paper called "How Do Committees Invent" - Fred Brooks cited this paper in the Mythical Man Month [30] calling it Conways law [31]. Conways law states *"Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure"*. In the early days of Amazon, Jeff Bezos introduced the idea of the "2 pizza team rule" [32] where a team size should be no larger than can be fed by two pizzas. The basic ideas are rooted in concepts that productive teams should be small, and independent. This aligns nicely with cloud native concepts in that one way to ensure that components are independent and interoperate only via their published interfaces, is that teams are also organized

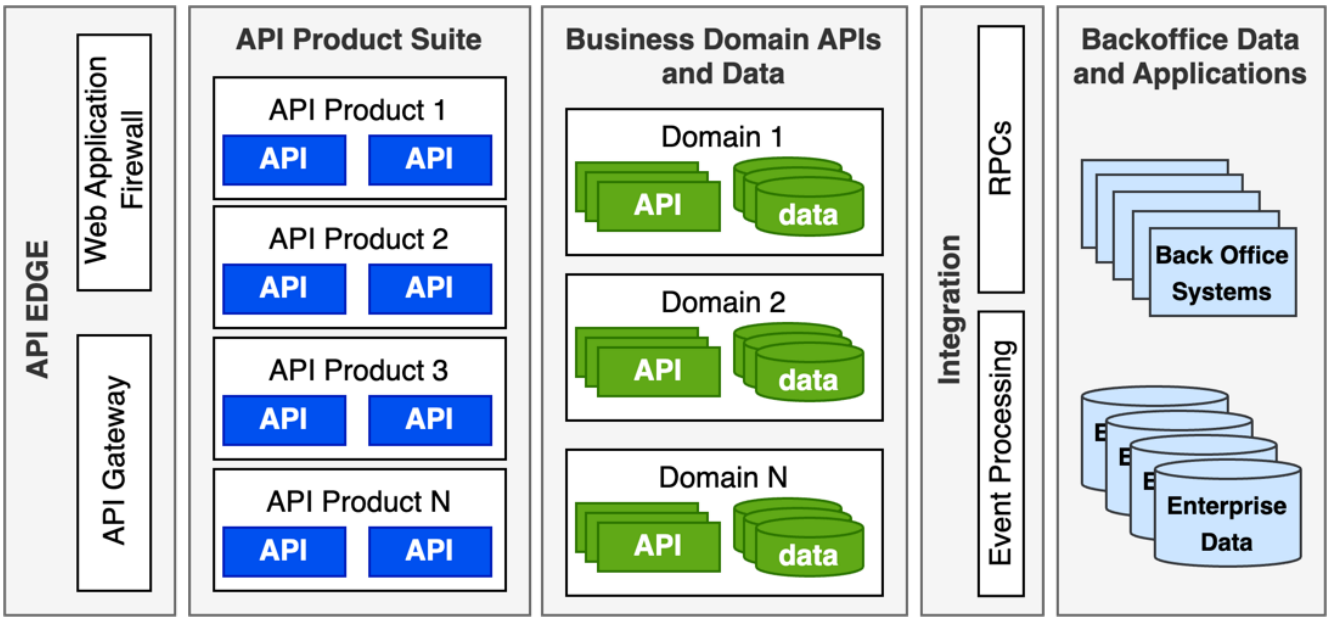

Fig. 2. Conceptual API-Based Architecture

this way. Over the years various organizational models and changes have been debated. *Full-stack teams* bring together front-end, back-end, testing and infrastructure professionals to a common team where they have full responsibility for their technical assets. *Shifting left* along with the emergence of *DevOps* brings a testing and automation focus to teams that allows them to increase quality and productivity. One problem is that while these concepts work well for driving individual team productivity, they are difficult to scale to larger organizations with multiple products. Several companies have also published their attempts to scale their practices. One popular model was published by Spotify [33], where "Squads" represent full-stack teams, but they also introduce concepts like "Tribes" for coordinating squads, and "Guilds" to address cross-cutting technical concerns. Commercial models have also been created to drive organizational changes to support cloud native architectures. One such example is the SAFe [34] framework, which is interesting in that it favors prescriptive organization and process rigor over streamlining software engineering practices in order to increase scale. We think there are some interesting problems in this space given the lack of alignment on the best ways to organize teams to work on cloud native applications at scale. Specifically, just copying a model used in one organization and using it in another organization does not address many of the challenges associated with things like politics and culture.

Software engineering demands for API based technical products. Historically, most applications are built to solve a targeted user or business problem end-to-end. In Mark Richards book entitled "Software Architecture Patterns" [35], Chapter 1 is focused on the *Layered Architecture Pattern*, which is shown in Figure 3. Richards layered pattern expands on the 3-tier architecture pattern [36] that calls for the isolation of the presentation, business logic, and database layers. An important attribute of these patterns is that the end-user interfaces only with the presentation layer, allowing the remaining layers to be hidden and secured against direct access. One of the foundational enablers of cloud computing is the plethora of first-class integration services provided by the platform. These capabilities open the door for new software architectures based on offering API-enabled services as a product. We propose a conceptual model for this type of architecture in Figure 2, while it is layered, the layers have different responsibilities

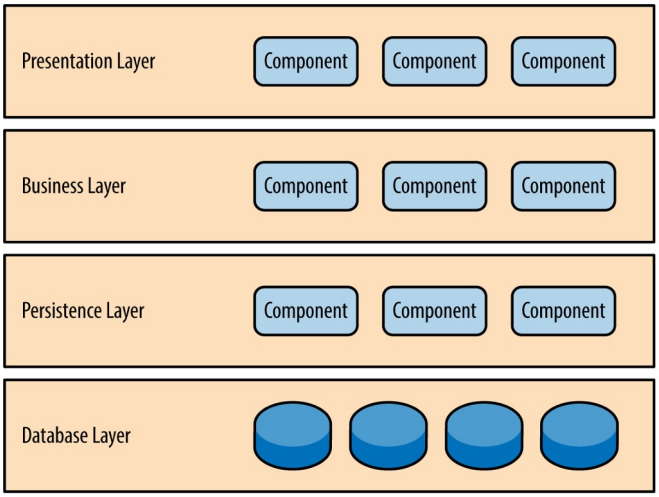

Fig. 3. Layered Software Architecture

than the ones discussed by Richards. Several well known examples of API products are Google Maps (for location services), Twilio (for messaging), and Stripe (for payment). These all represent API enabled capabilities designed specifically for embedding into other applications. As this model expands in popularity software engineers will have to become familiar new architecture patterns for designing API-centric products. For example, some of the best AI models are offered as a service by OpenAI [37], and the healthcare industry is being mandated to offer API services to support interoperability [38] regulations. Some of these companies are even taking the opportunities to use APIs as a competitive differentiator, one example is the developer portal provided by Cigna [39]. Prior to the mainstream adoption of the cloud native applications it was not possible to work with your bank, healthcare provider, auto insurer, and favorite retail stores with custom developed software.

III. The Cloud is Expanding to the Edge

The underlying services that the major cloud providers offer to their customers continues to expand and evolve. These advancements provide significant innovation capabilities to customers, but also put pressure on how to effectively engineer solutions that take advantage of these services cost effectively. Figure 4 shows the high level view of the computing services in the modern cloud. While it was once easy to identify the boundary of where the cloud started and ended, it is no longer easy to define the edge of the cloud. The following section will provide an overview of the evolution of the cloud, along with highlighting some of the challenges that have to be overcome by software engineers.

A. The Basic Cloud Architecture

At a high-level the basic cloud architectures deployed by major providers exhibit many similarities. The foundational infrastructure building block of cloud compute is called an **Availability Zone** (AZ). An AZ is a custom designed data center that hosts cloud infrastructure (compute, storage, and network) and runs cloud provider services on behalf of their customers. A **Region** is a physical location where a collection of 2 or more AZs are located. Each AZ within a region are connected together with a fully redundant high bandwidth low latency network. The goal of a region is to have AZs close enough so that they can behave like a single cluster, but also separate them by enough distance to isolate them from issues associated with power failure, earthquakes, tornados, and so on. AWS, as an example, uses the general guideline of 100km (60 miles) [40] for placing AZs within a region. The global footprint of a cloud provider is defined by the number of regions and locations they have across the globe, along with the purpose-built underlying network they use to interconnect them together.

Given the cost and complexity of deploying cloud regions around the globe, cloud providers expand their network reach through the use of edge locations. Edge locations are useful for a couple of reasons. First, they serve as a point of presence to lower connection latency to the cloud, and second, they can run services at the edge which offers additional benefits. One of the classical applications to run at the edge is a content delivery network (CDN). CDNs speed up web and mobile applications. For digital web and mobile applications the combination of Regions, AZs, and Edge Locations could be considered the boundary of the cloud. These are shown on the right side of Figure4. The major cloud providers continue to focus on expanding the number of services they support at the edge to drive even better performance and reduce network latency.

From a software engineering perspective, the basic cloud architecture described above introduces additional cognitive load on software engineers:

- Planning application deployment starts with the design of a virtual data center (VDC). VDCs logically carve out storage, compute, network and security policies from the cloud provider for customer usage. Traditional software engineers are not trained to think of the start of the software design process begins with the need to design a virtual data center and all of the complexity that comes with it. Historically, data centers are designed by specialty engineers and inherited "as is" into the final software architecture. The data center topology is now a critical software engineering concern.
- Quality attributes such as privacy, resiliency, reliability and scalability are foundational concepts that software architects use to reason about systems. These now move out of the conceptual realm and require a deeper understanding of technical constructs that now become part of the software design itself. For example, deploying microservices across different subnets, where each subnet is in a different AZ within a region. Also, declarative definition of the security policies that govern access to these microservices along with their entitlements to access other cloud resources becomes part of the software product itself.
- Although the cloud itself provides an infrastructure model to create resilient solutions that run at scale, its up to the software engineer to architect things properly to take advantage of these capabilities[3]. For example, it's still possible to deploy an application to a single virtual machine instance in the cloud, which without additional controls will not elastically scale, nor will it be resilient to failure. Thus, to enable the creation of cloud services software engineers must have mastery of newer patterns for distributed applications (*e.g.*, especially asynchronous event-based architectures).
- While cybersecurity has always been an important consideration of software engineers, the cloud materially expands these responsibilities. Everything in the cloud is secured by policy, but as mentioned earlier, software engineers now need to deal with security requirements across the entire OSI model [42] stack in addition to some

[3]Some patterns such as rehosting [41] *a.k.a.* "lift-and-shift" should not be considered cloud native patterns.

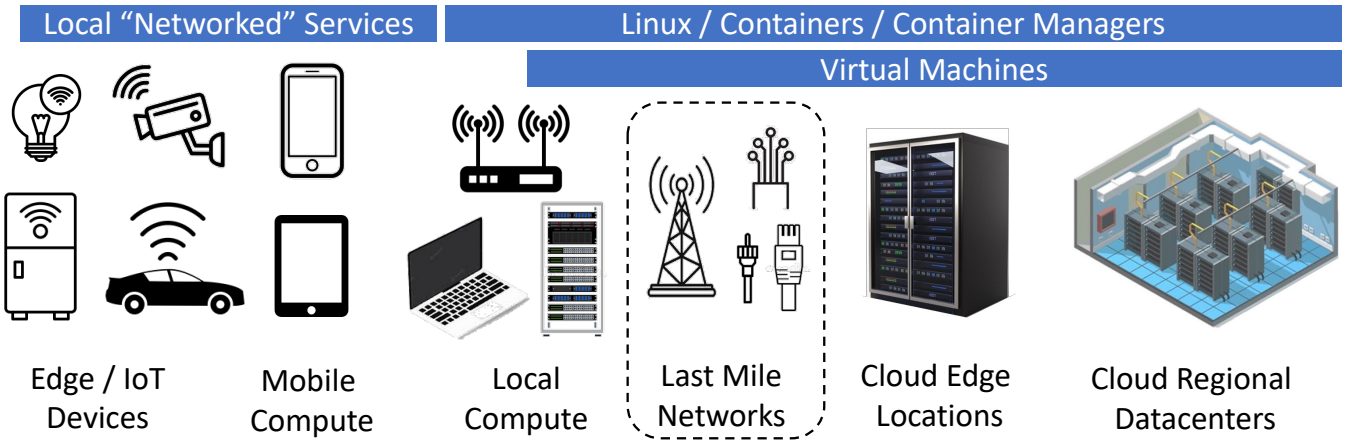

Fig. 4. The Modern Cloud

unique cloud requirements. Generally software engineers are comfortable with security at Layer 7 (the Application Layer) of the OSI model, using techniques such as OAuth 2 [43] to secure different types of digital assets. These responsibilities now expand to authoring and deploying policies to govern network access across subnets, and for attaching policies necessary to use managed services. In addition, software engineers must deploy and ensure proper configuration of virtualized security appliances such as web application firewalls (WAFs) and traditional firewalls that are now virtualized and software-defined. As attacks get more sophisticated, software engineers must also make decisions around introducing additional security capabilities into their solutions such as bot-detection, and defenses against credential-stuffing via MFA and supply chain attacks. The complexity of properly configuring, keeping track of, and managing cloud resources is also a new cloud-specific concern for software engineers. These problems themselves are also being addressed by software which needs to be deployed, configured and managed; for example Cloud Custodian [44] that was open sourced by Capitol One and donated to the CNCF.

- One clear benefit of deploying to the cloud is that the easiest path to do so requires everything to be automated. While software engineers are comfortable with automation associated with software tasks like testing, they are not accustomed to automating infrastructure deployment. This becomes even more challenging given many of the existing infrastructure automation tools were designed for non-programmers, relying on verbose, complex and error-prone configuration formats like YAML and JSON. Some progress in this space has been accomplished via DSLs like Terraform [6] and tools that use real programming languages like Pulumi [8].
- Another foundational software engineering cloud concern that design decisions have material influence on operational runtime costs. This is often referred to as *finops*, short for financial operations. At its core, the cloud transforms compute, network, storage, and security into a pay-as-you-go utility. Software engineers generally don't factor in things like programming language selection, database platforms, processor hardware architecture, frameworks, fully-managed services and so on into their design from the perspective of cost and carbon footprint impact. We will explore this topic more in Section IV.

B. The Emergence of Edge Computing

With the rapid growth of devices that are connected to the internet, we are now entering the era of edge computing [45]. Edge computing is different architecturally from traditional cloud computing. Consider a cloud-enabled web or mobile application. The architecture of these applications is often based on calling cloud-hosted APIs and then using the data returned from these to power the user experience. This architecture will not scale or meet the needs of all of the smart devices that connect to the internet. As its name implies, edge computing moves more computing services to the edge, with requirements not found in web or mobile applications: Specifically:

- They must be able to work autonomously. The cloud would not scale to support all device events, local processing is used to filter important events from less important events.
- They must be able to work fully disconnected, or with unreliable network connectivity.
- They must be able to perform compute locally, either independently, or in local clusters.

These added capabilities essentially extend the edge of the cloud all the way back to the client devices themselves as shown in Figure 4. The overall architecture of the modern cloud that extends to the edge is shown in Figure 5[4].

[4]Figure copied from https://www.spiceworks.com/tech/cloud/articles/edge-vs-fog-computing/

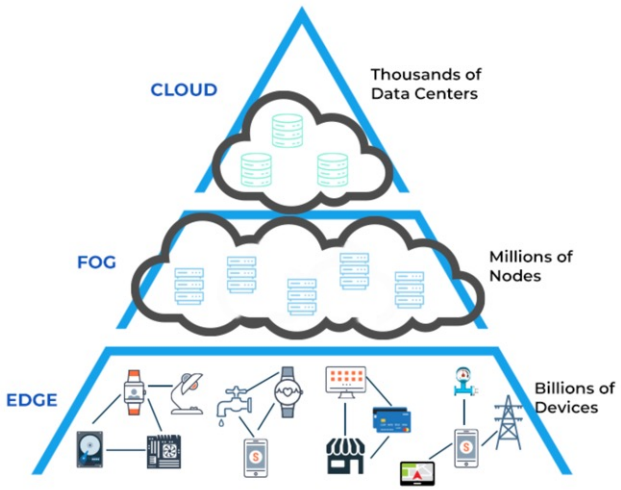

Fig. 5. Edge Compute Architecture

According to research by IoT Analytics, there were 14.4 million smart devices connected to the internet in 2022, expected to rise to almost 30 million by 2025 [46]. This many deployed devices could not be supported if they required connectivity to the large cloud data centers discussed earlier. Processing will need to move to the devices themselves supported by a new layer of cloud compute that is closer to the devices. This new layer of compute is often referred to as *fog computing*. The term comes from a play on the word cloud, given clouds are high up in the sky and fog is closer to the ground.

As cloud providers expand their footprint across the globe, creating new regions with multiple availability zones represents a major strategic decision because of the time, expense and other factors that go into rolling out multiple large data centers. Creating new regions is required to increase compute capacity as global cloud adoption expands, and to meet specific compliance requirements associated with conducting business in the cloud. For example, many countries are adopting data residency laws, which place controls over where data is stored at rest. The trend of cloud providers searching for strategic locations for new Regions, or to expand the number of AZs within a region will continue as cloud demand increases. Given the massive investments required, it is likely that the number of major cloud providers will continue to be small. A November 2022 TechTarget report [47] highlights that the big 3 providers – AWS, Microsoft, and Google – account for 62% of the overall cloud market.

While its unlikely that there will be significant disruption in the major cloud providers, the fog layer is likely to be federated across many different players. This layer needs to be deployed close to the edge devices themselves, and will likely be addressed by existing last mile internet service providers (ISPs), and by telecommunication companies offering 5G services who already have deployed infrastructure to meet these needs.

IV. The Edge and Expansion of Software Engineering Concerns

The evolution of cloud computing over the past decade has increased the decision landscape for software engineers. This section will highlight some of the new concerns that software engineers must address in cloud and edge computing design.

A. Processor Hardware Architecture Diversity

Ten years ago we did not have cloud providers creating custom processors for compute, special purpose AI applications, nor did we have all of the microcontrollers running at the edge of the Internet. Familiarity with making informed hardware architecture choices now becomes an important concern of software engineers. Some examples include:

- On May 23, 2023, AWS announced the third generation of their custom ARM-based microprocessor called Graviton 3. AWS claims that workloads running on Graviton 3 are 50% faster than Intel/AMD processors, consume 60% less power, and are 20% cheaper. From a software engineering perspective these benefits seem like a no brainer to take advantage of until you start to factor in other requirements such as being able to maintain ARM-based builds of your software, including all dependencies which may not be available or optimized for ARM. Additionally, organizations may impose other requirements such as running certain security products on VMs, these also must be available and certified.
- Since the realization that GPUs can improve the performance of training AI models, cloud providers have innovated further with custom AI microprocessors. In 2016 Google introduced the Tensor Processing Unit (TPU) to accelerate training deep learning models, and in 2018 AWS created the Inferentia chip to accelerate inference. AWS also entered the training space to compete with Google's TPU with the Trainium chip in 2020. With all of these new AI hardware choices, software engineers must be savvy with aligning hardware choices with software training and inference library requirements. For example, AWS announced a SDK for Trainium called Neuron to enable engineers to use popular AI frameworks such as Tensorflow and PyTorch.
- As we move to the edge, software engineers now more routinely have to create solutions for microcontollers, and other devices that have additional constraints. These devices might be battery powered, compute constrained, difficult to access or update, and/or have unreliable network connectivity. Programming frameworks and tools routinely available on modern servers might not be available or viable for these devices. Consider the popular trend of deploying code in containers. To a large extent, containers make assumptions that there is an underlying linux kernel, which might not be possible in these purpose-built devices. Instead of falling back to creating alternative versions of their software in lower level systems programming languages like C/C++, software engineers must become familiar with emerging solutions in

this space. Consider TinyGo [48], which is an alternative Go compiler specifically created to bring the Go ecosystem, which is popular in the cloud, to microcontrollers. Another example, is WasmEdge [49], which brings the power of Web Assembly(WASM) to the server and to edge devices. WasmEdge can run embedded WASM code created by modern compilers that are popular for creating cloud native applications such as Rust, Go, and Javascript.

- Managing tool chains for multiple hardware architectures. The Java programming language introduced the concept of *"Write Once, Run Anywhere"*. It accomplished this by creating Java Virtual Machines (JVMs) for different hardware platforms, and running compiled bytecode consistently across these platforms. While this works well, the Java ecosystem has some challenges in the broader cloud native space. Specifically, to use Java all dependencies must be Java-based, and although the JVM itself is an impressive, its size and compute requirements might be challenging to support on edge devices. Newer programming languages like Rust and Go have been adopting an open cross-compiler philosophy so that any compiler on any platform can create binaries for any other platform. Containers are another popular cloud native technology. With the need to support diverse processor architectures container packaging becomes more complex, and containers might not be practical on the edge given they assume the presence of a linux kernel. Docker recently released a technical preview to support web assembly that may help address this issue [50].

B. Polyglot Programming

We think the move to cloud native architectures requires software engineers to rethink the criteria for how programming languages are selected. In 2013 Meyerovich and Rabkin [51] reported on empirical human factors that impact programming language selection. Their findings cite reasons such as open source libraries, existing code, and programmer experience as the primary drivers for selecting programming languages for new projects. To complicate matters further, in some organizations approved programming languages are standardized removing the software engineering community from the decision making loop.

The general criteria to evaluate programming languages often examines attributes like object-oriented vs functional; high-level vs low-level; type safety vs dynamic; general purpose or domain specific, and so on. While these are good attributes to categorize programming languages they don't factor in criteria aligned to cloud native computing objectives. For example [52] did a comparative analysis of Java vs Kotlin. Kotlin has been increasing in popularity within the Java community because it is less verbose, introduces modern programming language features, while interoperating well with existing Java code. One of the criteria for good cloud native software discussed earlier is being able to move fast, thus adopting a language like Kotlin that is more productive and easier to test represents a good engineering tradeoff for organizations with significant investments and skillsets in Java.

We think an approach for programming language selection should be based on a careful tradeoff analysis using cloud native computing architecture decisions to guide the selection. This will often lead to a polyglot outcome, where more than one programming language is selected. One interesting study in this space was conducted by Cordingly et. al. [53] where they examined the Java, Python, Go, and Node.js against a collection of different Function as a Service (FaaS) workloads. We like their strategy using drivers such as performance and cost as the evaluation criteria. They also used specific FaaS concerns such as cold and warm start times in their analysis.

We think the approach used by Cordingly should be expanded to other cloud native architecture options. For example, with container based solutions, languages like Java tend to produce very large containers, and require significant resources associated with bringing along the JVM. Modern languages like Go, designed with the cloud in mind[5], produce very small containers, and have a robust and modern runtime. Languages like Javascript and Typescript coupled with the Node.js runtime is highly optimized to support asynchronous event-based architectures. Languages like Rust provide C/C++ performance, but have a modern runtime and provide compiler-enforced memory safety. In addition to the languages themselves, additional factors such as the maturity and completeness of cloud provider supplied language-specific SDKs should also be considered in the selection criteria.

C. Multi-Region and Multi-Cloud

The major cloud providers run massive infrastructures that have historically have provided availability that any individual enterprise would envy. However, while rare, cloud providers have had outages that have resulted in significant impact to customers. As cloud adoption continues to grow, the impact of these periodic outages will also continue to grow. Additionally, the major cloud providers compete with each other via their innovation investments into fully managed services. This has consequences of cloud vendor lock-in, making it hard for a customer to migrate from one cloud provider to another based on having to redevelop and redeploy their software on another providers platform. Software engineers need to be well versed in the options and consequences from a cost and scale perspective with respect to making Multi-Region and Multi-Cloud decisions. For example:

- As mentioned in Section III-A cloud providers offer resiliency and scale using the concept of a Region comprised of multiple Availability Zones. Most cloud providers handle the complexity of enabling services to run across AZs in a way that is transparent to customers. Little engineering effort is required by customers to continue to operate correctly when a single AZ fails within a region. However, if an entire region fails, additional engineering is required to continue correct operation. These

[5]Go is the primary language used to build significant cloud native platforms like Docker and Kubernetes

efforts increase the complexity of the software, software testing, and cost as the customer becomes responsible for data replication strategies and redundant storage costs to operate across regions. Netflix had authored engineering notes on the strategy that they use on their techblog [54], and also created open source modules to perform chaos testing to validate their platforms ability to survive various types of failures.

- While cloud providers offer many services that are similar, they also compete by trying to differentiate their suite of fully managed PaaS/FaaS offerings. Software engineers can lower the blast radius of failures by diversifying services across different cloud providers. For example, if a software engineer decided for their workloads that AWS has a superior FaaS solution with Lambda, and GCP has the best managed Kubernetes service GKE, consideration could be given to a best-of-breed strategy. While this would reduce risk of individual cloud failure exposure, it also comes with risk and complexity. Latency could suffer as intra-application calls have to leave one cloud and enter another. Also, many cloud providers provide tiered pricing models, so maximizing use on one cloud provider drives the best discounts.
- Even with being able to successfully run across different cloud providers or multiple regions software engineers must design applications that continue to operate with reduced function in the face of failure. These tradeoffs would need careful consideration and be domain focused. For example, on an e-commerce platform, favoring services that will allow customers to make and pay for orders can be prioritized over the ability to fulfill orders until normal cloud operations are restored.
- Being able to run applications across cloud providers, or to port to another cloud provider requires software engineers to make careful product selection decisions. For example, consider AWS Dynamo and Azure Cosmos. These are both database solutions that are often compared to each other with respect to resilience, hyper-scalability and performance. From a software interface perspective they are very different and would require a significant rewrite to port from one solution to another. Choosing an alternative technology like managed PostgreSQL for databases would be easier to support across different cloud providers. Kubernetes is another example of a platform that will be more similar to support across different cloud providers. In all cases, the processes to deploy and secure cloud assets across different cloud providers is not the same, so adopting a multi-cloud strategy, even with similar technologies, will still introduce cost and complexity.

V. Cloud Computing Education

One area we think is underserved by the research community is investigation into software engineering specific cloud education. Queries of Google Scholar [55] for research on Cloud Native Software Engineering produce very few results of relevance. Most of the results focus on work to advance specific technical areas within the field of software engineering, for example, running AI/ML workloads, automation, software testing, or supporting microservice based architectures.

So how do software engineers learn cloud computing? The cloud providers themselves have done a good job filling part of the need via education and certification programs. Cloud certifications are highly valued by industry and software engineers themselves, as they often post their certification accomplishments on personal LinkedIn pages. While cloud certifications are valuable, they focus on how to accomplish activities on a cloud platform versus how to engineer good software products using cloud services. We have discussed FaaS as an important cloud software engineering opportunity earlier. Certifications would enable engineers to understand how to deploy a FaaS component on AWS Lambda, Azure Functions, or Google Cloud Functions, but they would not cover important other considerations such is your architecture event-driven, how will state be managed, SDK robustness, component granularity (*e.g.*, is a function too small) and so on.

We think an important research question is to address if academic institutions play a leadership or partnership role to address the educational needs of software engineers working in the cloud. It appears that most of the collective knowledge in this space is acquired with on-the-job experience and via publications authored by industry professionals.

VI. Conclusion

This paper brings attention to the need for additional focus on expanding software engineering practices given the trend of moving to cloud and edge computing. We examined many cloud computing architectural concepts from the lens of a software engineering practitioner and summarized many opportunities that would benefit the community. Many organizations transition to the cloud by taking their top engineers and focusing them on moving existing digital assets like web and mobile applications to the cloud.

Early successes, coupled with the attractive technical capabilities liked by engineers, and a pay-as-you-go model liked by managers continue to drive acceleration of the cloud. We think this next wave of cloud will require more software engineering rigor as we need to scale the number of qualified engineers to work in the cloud, while at the same time, needing to support moving core enterprise applications to the cloud that don't have the same architectural characteristics as digital applications.

We also discussed new business opportunities that could only emerge with the cloud such as API-based products, and edge computing. These are complex architectures that will require software engineers to master additional skills.

References

[1] "IDC Forecasts Worldwide "Whole Cloud" Spending to Reach $1.3 Trillion by 2025". IDC. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS48208321

[2] "The History of Salesforce". Salesforce.com. [Online]. Available: https://www.salesforce.com/news/stories/the-history-of-salesforce

[3] "About AWS". Amazon Web Services. [Online]. Available: https://aws.amazon.com/about-aws/

[4] "The History of Microsoft Azure". Microsoft. [Online]. Available: https://techcommunity.microsoft.com/t5/educator-developer-blog/the-history-of-microsoft-azure/ba-p/3574204

[5] B. Stevens. "Google Cloud Platform: your Next home in the cloud". [Online]. Available: https://cloud.google.com/blog/products/gcp/google-cloud-platform-your-next-home-in-the-cloud

[6] "Automate Infrastructure on Any Cloud". Hashicorp. [Online]. Available: https://terraform.io

[7] "AWS CloudFormation". AWS. [Online]. Available: https://aws.amazon.com/cloudformation

[8] "Pulumi: Universal Infrastructure as Code". Pulumi. [Online]. Available: https://www.pulumi.com

[9] G. Kim, P. Debois, J. Willis, J. Humble, and J. Allspaw, *The DevOps Handbook: How to Create World-class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press, LLC, 2016. [Online]. Available: https://books.google.com/books?id=Kvq5zQEACAAJ

[10] Google how search works. Google. [Online]. Available: https://www.google.com/search/howsearchworks/how-search-works/rigorous-testing/

[11] Chaos monkey. Netflix Open Source. [Online]. Available: https://netflix.github.io/chaosmonkey/

[12] L. F. Albuquerque Jr, F. S. Ferraz, R. Oliveira, and S. Galdino, "Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas," in *ICSEA*, 2017, pp. 206–212.

[13] D. S. Linthicum, "Cloud-native applications and cloud migration: The good, the bad, and the points between," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 12–14, 2017.

[14] "Cloud Native Computing Foundation Homepage". CNCF. [Online]. Available: https://www.cncf.io/

[15] "Open Container Initiative". OCI. [Online]. Available: https://opencontainers.org/

[16] "Use containers to Build, Share and Run your applications". Docker.com. [Online]. Available: https://www.docker.com/resources/what-container

[17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski *et al.*, "Serverless computing: Current trends and open problems," in *Research advances in cloud computing*. Springer, 2017, pp. 1–20.

[18] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017, pp. 185–200.

[19] B. Bosshard, "On the use of web assembly in a serverless context," in *Agile Processes in Software Engineering and Extreme Programming–Workshops*, 2020, p. 141.

[20] "Production-Grade Container Orchestration". CNCF. [Online]. Available: https://www.kubernetes.io

[21] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *17th USENIX symposium on networked systems design and implementation (NSDI 20)*, 2020, pp. 419–434.

[22] A. Wiggins. "The Twelve-Factor App". Heroku. [Online]. Available: https://12factor.net

[23] K. Hoffman, *Beyond the Twelve-factor App: Exploring the DNA of Highly Scalable, Resilient Cloud Applications*. O'Reilly Media, 2016. [Online]. Available: https://books.google.com/books?id=Ib3iuQEACAAJ

[24] "12 Factor App Revisited". architecture notes. [Online]. Available: https://architecturenotes.co/12-factor-app-revisited/

[25] A. Thomas and A. Gupta, "Adopt a mesh app and service architecture to power your digital business," Gartner Research, Tech. Rep. G00392875, July 2022.

[26] Z. Dehghani. "Data Mesh Principles and Logical Architecture". Thoughtworks. [Online]. Available: https://martinfowler.com/articles/data-mesh-principles.html

[27] E. Evans, M. Fowler, and E. Evans, *Domain-driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. [Online]. Available: https://books.google.com/books?id=xColAAPGubgC

[28] V. Vernon, *Implementing Domain-Driven Design*. Pearson Education, 2013. [Online]. Available: https://books.google.com/books?id=X7DpD5g3VP8C

[29] "WASI – The WebAssembly System Interface". CNCF. [Online]. Available: https://wasi.dev/

[30] F. P. Brooks, *The mythical man-month – Essays on Software-Engineering*. Addison-Wesley, 1975.

[31] J. D. Herbsleb and R. E. Grinter, "Architectures, coordination, and distance: Conway's law and beyond." *IEEE Software*, vol. 16, no. 5, pp. 63–70, 1999. [Online]. Available: http://dblp.uni-trier.de/db/journals/software/software16.html#HerbslebG99

[32] A. Atlas, "Accidental adoption: The story of scrum at amazon.com," in *2009 Agile Conference*, 2009, pp. 135–140.

[33] "Discover the Spotify Model". Atlassian. [Online]. Available: https://www.atlassian.com/agile/agile-at-scale/spotify

[34] "Scaled Agile Framework (SAFe)". Scaled Agile. [Online]. Available: https://www.scaledagileframework.com/

[35] M. Richards, *Software architecture patterns*. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA ..., 2015, vol. 4.

[36] A. Aarsten, D. Brugali, and G. Menga, "Patterns for three-tier client/server applications," *Proceedings of Pattern Languages of Programs (PLoP'96)*, vol. 4, no. 6, 1996.

[37] Build next-gen apps with openai's powerful models. OpenAI. [Online]. Available: https://openai.com/api/

[38] Hl7 fhir. HL7. [Online]. Available: https://www.hl7.org/fhir/http.html/

[39] Develop for healthcare. Cigna. [Online]. Available: https://developer.cigna.com/

[40] "Regions and Availability Zones". AWS. [Online]. Available: https://aws.amazon.com/about-aws/global-infrastructure/regions_az

[41] A. Engelsrud, "Moving to the cloud: Lift and shift," in *Managing PeopleSoft on the Oracle Cloud*. Springer, 2019, pp. 229–242.

[42] Osi model. Wikipedia. [Online]. Available: https://en.wikipedia.org/wiki/OSI_model

[43] Rfc-6749. IETF RFC. [Online]. Available: https://www.rfc-editor.org/rfc/rfc6749

[44] Cloud custodian. CapitolOne Open Source. [Online]. Available: https://cloudcustodian.io/

[45] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE access*, vol. 8, pp. 85 714–85 728, 2020.

[46] M. Hasan. State of iot 2022: Number of connected iot devices growing 18% to 14.4 billion globally. IOT Analytics. [Online]. Available: https://iot-analytics.com/number-connected-iot-devices/

[47] B. Posey. (2022, Nov) Top public cloud providers of 2023: A brief comparison. TechTarget. [Online]. Available: https://www.techtarget.com/searchcloudcomputing/tip/Top-public-cloud-providers-A-brief-comparison

[48] Tinygo - a go compiler for small places. TinyGo. [Online]. Available: https://tinygo.org/

[49] Wasmedge - bring the cloud-native and serverless application paradigms to edge computing. CNCF. [Online]. Available: https://wasmedge.org/

[50] Introducing the docker+wasm technical preview. Docker. [Online]. Available: https://www.docker.com/blog/docker-wasm-technical-preview/

[51] L. A. Meyerovich and A. S. Rabkin, "Empirical analysis of programming language adoption," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 1–18.

[52] M. Flauzino, J. Veríssimo, R. Terra, E. Cirilo, V. H. Durelli, and R. S. Durelli, "Are you still smelling it? a comparative study between java and kotlin language," in *Proceedings of the VII Brazilian symposium on software components, architectures, and reuse*, 2018, pp. 23–32.

[53] R. Cordingly, H. Yu, V. Hoang, D. Perez, D. Foster, Z. Sadeghi, R. Hatchett, and W. J. Lloyd, "Implications of programming language selection for serverless data processing pipelines," in *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*, 2020, pp. 704–711.

[54] Active-active for multi-regional resiliency. Netflix. [Online]. Available: https://netflixtechblog.com/active-active-for-multi-regional-resiliency-c47719f6685b

[55] Google scolar. Google. [Online]. Available: https://scholar.google.com/

Brian S. Mitchell is an accomplished technologist, engineer, educator, software engineering researcher, speaker, strategist, leader, and enterprise-scale change agent. Brian is currently a member of the Department of Computer Science at Drexel University. His career has spanned both industry and academia, including holding the Distinguished Engineer role at a Fortune 15 company. He provided technical thought leadership and directed teams responsible for driving disruptive digital innovation that led to the creation of multiple generations of products that help millions of people every day. Brian also has more than 20 years of teaching experience in a variety of areas including Software Engineering, Software Architecture, Operating Systems, Networks, Computer Architecture, Programming Languages, and Distributed Systems. His recent research interests include exploring several interesting problems at the intersection of Software Engineering, Software Architecture and Cloud Native Computing. Previously he was one of the founders of the Search-Based Software Engineering research space, publishing many influential papers focused on recovering software architecture insights directly from source code. Dr. Mitchell holds BS, MS and PhD degrees in Computer Science, and a ME in Computer & Telecommunication Engineering.