The RESTless Cloud

Nathan Pemberton
UC Berkeley
nathanp@berkeley.edu

Johann Schleier-Smith
UC Berkeley
jssmith@berkeley.edu

Joseph E. Gonzalez
UC Berkeley
jegonzal@berkeley.edu

ABSTRACT

Cloud provider APIs have emerged as the de facto operating system interface for the warehouse scale computers that comprise the public cloud. Like single-server operating systems, they provide the resource allocation, protection, communication paths, naming, and scheduling for these large machines. Cloud provider APIs also provide all sorts of things that operating systems do not, things like big data analytics, machine learning model training, or factory automation. Somewhere, lurking within this menagerie of services, there is an operating system interface to a really big computer, the computer that today's application developers target. This computer works nothing like a single server, yet it also isn't a dispersed distributed system like the internet. It is something in-between. Now is the time to distill and refine a coherent "cloud system interface" from the multitude of cloud provider APIs, preferably a portable one. In this paper we discuss what goes in, what stays out, and the principles that inform these decisions.

CCS CONCEPTS

• **Computer systems organization** → **Cloud computing**; • **Software and its engineering** → *Operating systems.*

ACM Reference Format:
Nathan Pemberton, Johann Schleier-Smith, and Joseph E. Gonzalez . 2021. The RESTless Cloud. In *Workshop on Hot Topics in Operating Systems (HotOS '21), May 31–June 2, 2021, Ann Arbor, MI, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3458336.3465280

1 INTRODUCTION

The cloud is a diverse and complicated place. Cloud providers have added services one-by-one, their offerings growing organically to meet countless customer needs. Each service has its own set of interfaces and semantics, and the differences between cloud providers sometimes seem greater than their similarities. When deciding on a new feature, users must pick a set of services to commit to, then work out how to integrate and manage them. As needs evolve and platform services expand, they may be forced to rewrite their logic.

Contrast this with writing an application for a single server. While there are many languages and frameworks to choose from, a common set of underlying abstractions is found on just about every machine available. Whether the platform is Windows or Linux, files and processes work in roughly the same way. The portable operating system interface (POSIX) [38] arose to formalize these patterns, not just for portability as the name implies, but also as a model of how an operating system behaves. It is time for the cloud to have its own POSIX, a standard model for state and computation.

There are many options for how such an interface might be designed. Most commonly today, programmers focus on building applications that comprise multiple networked web services [75], using REST-based protocols [26] to access storage, compute, and other data center resources. Alternatively, we could try to design a single system image (SSI) operating system [15] that presents the cloud as if it were a single machine. Some proposals, like LegoOS, have proposed extending POSIX abstractions to disaggregated resources [64]. Others propose a departure from POSIX to abstractions more suitable for distributed systems [62]. In this paper, we contend that only these latter approaches are suitable for the cloud. The cloud is not a single computer and application designs need to reflect that. However, the cloud is also not a widely dispersed set of independent machines as assumed by web services.

In reality, the cloud is a collection of ever-changing, tightly managed resources shared by many independent users. It is also not a particular system, but a *category* of systems with many competing implementations. Likewise, we are not proposing a particular operating system, but a *category* of system interfaces that can be implemented in a portable way by any vendor. Furthermore, this interface should integrate the wide range of constantly evolving features and services available in the cloud today while providing an easier path to innovation in the future. Critically, it will need to reflect the physical reality of the cloud in a natural and intuitive way. This helps ensure that applications can grow without encountering artificial scalability bottlenecks, and that their costs and resource consumption remain commensurate to their actual needs.

HotOS '21, May 31–June 2, 2021, Ann Arbor, MI, USA

ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465280

Serverless computing, and Function-as-a-Service (FaaS) in particular, attempts to address many of these requirements [17, 60]. Cloud functions scale in accordance to the number of requests they receive and free users from concerns of provisioning and configuring individual servers. However, current serverless interfaces remain limited in scope [34], offering an alternative, rather than a unifying, paradigm (Section 2). In this paper, we will present a sketch of a unified cloud interface that builds on serverless abstractions to provide a portable and unified view of the cloud (Section 3). While this is just one possible proposal for a portable cloud interface, we hope that it will serve as a starting point for discussion. We will follow our proposal with a discussion of the benefits such an interface can provide (Section 4). Finally, we will present some remaining open questions and challenges for the community (Section 5).

Operation	Latency
2005 data center network RTT	1,000,000 ns
2021 data center network RTT	200,000 ns
Object marshaling (1k)	>50,000 ns
HTTP protocol	50,000 ns
Socket overhead	5,000 ns
Emerging fast network RTT	1,000 ns
KVM Hypervisor call	700 ns
Linux System call	500 ns
WebAssembly call - V8 Engine	17 ns

Table 1: Representative latency of various operations. Emerging network technologies have RTT times much lower than web service overheads. Hypervisor calls and system calls have similar latency, and WebAssembly isolation [31, 66] can have lower latency still.

2 TODAY'S ALTERNATIVES

Before we dive into the design of a new cloud interface, we first consider the inadequacies of existing solutions: the web services APIs that cloud providers offer today, UNIX-derived distributed operating systems, modern cross-cloud management solutions such a Kubernetes, and serverless computing as it exists today.

2.1 Why not web services?

Web services and the cloud are almost synonymous, and for good reason. Warehouse scale computing [8] is possible because it relies upon internet technologies with proven scalability. Internet Protocol provides routing, TCP provides flow control and congestion control, and HTTP load balancers distribute work across servers. Service endpoints provide stateless RESTful [25] interfaces, using another technology derived from the web.

While web services are excellent for scalability and interoperability, optimizing them for performance remains a stubborn problem. Table 1 shows the latency involved in various operations that might be invoked during a call into the cloud API. Web services APIs will always be adequate for certain things, such as provisioning servers, or even fetching large data objects from storage. However web service overheads will certainly become prohibitive on future fast networks [6], especially when supporting fine-grained operations such as small-block reads and writes. Part of the problem comes from protocol and data formatting requirements, part of it from stream oriented transport (cf. scatter-gather file system APIs), and part from the statelessness of REST. Statelessness is particularly fundamental, and has consequences such as repeated access control checks.

Building a distributed implementation of an application when an efficient single-machine implementation could meet the need can be tremendously wasteful [42], in part because of overheads such as those of web services. As a concrete example, we observe that fetching a 1KB object via the NFS protocol takes 1.5 ms and costs 0.003 USD/M (without the benefit of local caching), whereas fetching the same data from DynamoDB [68] takes 4.3 ms and costs 0.18 USD/M. We speculate that a part of the cost difference comes from the cloud provider passing the cost of providing a RESTful web service interface on to the customer.

At a minimum, cloud providers need a non-REST implementation of their existing APIs, but since performance problems are tied to the protocol statelessness, a simple translation is unlikely to suffice.

2.2 Why not POSIX?

Making a collection of computers work like one powerful computer is a longstanding goal of distributed operating systems research [63, 71]. A flurry of work ensued in the decades after inexpensive workstation hardware and local networks first became available [3, 4, 20, 24, 33, 45, 47, 51, 61, 78]. These efforts generally sought to provide a UNIX-like interface to a group of machines. However this line of work was largely eclipsed by the emergence of the internet, which ushered in a new era of distributed systems that operated on a far larger scale [7, 8]. The internet technologies won in the market with the help of tremendous investment, which makes it hard to conclude whether POSIX-like distributed operating systems suffered from technical failings, or whether they simply were not ready to meet the needs of gigantic internet services.

In [63], Schwarzkopf, Grosvner, and Hand argue that hardware trends have made warehouse-scale computers suitable for distributed operating systems. Indeed, there have been

several projects exploring designs in this direction [16, 54, 55, 62, 64, 81, 82].

The problem with POSIX and locality transparent operating system designs is the inverse of the problem with web services. While web services have a built in design assumption that everything is remote, POSIX has the built in assumption that everything is local. NFS provides a clear example of how interfaces designed in a local setting can prove troublesome in a distributed setting. For example, a remote file system that becomes unreachable may cause API responses not possible with a local file system [77]. Compliance with POSIX consistency guarantees [50], notably linearizability [35], has also been a perennial source of pain for distributed file system implementations [29, 37, 48, 79].

The assumption that everything is local infuses interface design and is even more pernicious than the assumption that everything is remote. A future-proof cloud system interface can make neither assumption—it must work well regardless of whether calls are serviced locally or remotely. We do not see an inherent trade-off, and believe it is possible to do both. We reinforce however, that we do not advocate full location transparency, where local operations and remote operations are indistinguishable, as in RPC [11] or distributed shared memory [49]. Such abstractions have long been known to be harmful [77]. Operations against memory or local storage are still local and always fast. Operations against the cloud API could be remote and slow, but they could also be local and fast.

2.3 Why not Kubernetes?

A number of systems have arisen to provide a more uniform abstraction for deploying services in the cloud and providing them with resources. Kubernetes [13, 14] has particularly strong industry adoption. Notably, all major cloud providers offer support for it, and since it also runs on-premise, it is the closest thing to a portable abstraction for the cloud. Kubernetes derives from the Borg [73, 74] cluster scheduler, which along with systems such as Mesos [36] and OpenStack [1] might be considered to offer a core functionality of an operating system at data center scale [87].

Kubernetes and its ilk have been quite successful within their domain: scheduling of lightweight server instances. However they have little to offer in the way of state management or security, and so represent a limited and incomplete slice of system functionality.

2.4 What about Serverless?

Serverless computing represents an exciting evolution of the cloud that we expect will help it deliver fully on its promise and potential [60]. FaaS with its autoscaling stateless functions gets the most attention [17], but other technologies such as cloud object storage share the essential characteristics of serverless computing: abstraction that hides servers, pay-per-use without capacity reservations, and autoscaling from zero to practically infinite resources. A major shortcoming of serverless computing as it exists today is that it comprises disparate technologies residing in their own silos. Programmers are burdened with using disjoint application paradigms, data models, and security policies. Performance and efficiency also suffer [69]. FaaS and other serverless technologies offer important lessons, but they do not yet provide the unifying paradigm that we seek.

3 A NEW INTERFACE

To move forward, we will need a new interface to the cloud. Let's refer to this new interface as the Portable Cloud System Interface (PCSI). What might this interface look like?

To begin answering this question, we now present a proof-of-concept design based around two key abstractions: state and computation. Separating state from computation has the advantage of allowing independent resource scaling, and has emerged as a popular design pattern for cloud applications. The boundary it creates is also a natural place for interposing a system interface, as demonstrated in established operating system designs.

3.1 Computation

We define computation as any transformation over state and refer to these transformations generically as "functions". Functions receive state as input and produce state as output. They may also may read and manipulate state as they execute, as described in Section 3.2.

In our PCSI proposal, functions are designed around three key properties:

- **Universal Compute Interface:** Functions provide the structure necessary for modular software [53]. A function can be reimplemented without changing its external interface, thus preserving an essential benefit of today's cloud web services. Drop-in replacement is possible, even when the new function relies on new underlying technology (e.g., hardware, programming language, runtime system). Thus PCSI provides an evolutionary path that enables rapid innovation in the cloud ecosystem. Multiple implementations of the same function can even be provided simultaneously, allowing an optimizer to choose dynamically among them to meet performance and cost goals [58].
- **No Implicit State:** Functions receive state, produce state, and interact with external state via the data abstraction, however they cannot rely on internal state beyond a single invocation. As with current serverless

FaaS offerings [17], or the vision of granular computing [41], this facilitates pay-per-use and allows functions to scale from a single invocation to thousands (or more).

- **Narrow and Heterogenous Implementations:** A wide and evolving range of platforms may be used to implement functions (e.g., accelerators, containers, unikernels [84], WebAssembly [31], etc.). However, each function should focus on a narrow and resource homogenous operation. This decoupling enables maximum innovation and helps resource allocation by isolating bottlenecks [52] and maximizing resource utilization.

Function arguments include explicit data layer inputs and outputs and a small pass-by-value request body. Users store functions themselves as objects in the data layer, allowing them to be invoked by other functions. In addition to invoking individual functions, users can build task graphs, which opens up optimization opportunities such as pipelining or physical co-location. Such task graphs can either be specified ahead-of-time, as in Cloudburst [69], or dynamically as in Ray [44] or Ciel [46].

PCSI functions are inspired by serverless FaaS and share similar design motivations and aims. However, PCSI pushes these abstractions toward a more universal and integrated system interface. For example, rather than require distinct services for things like model serving or data analytics, PCSI exposes these features through the same interface as any other function. Likewise, new hardware and software platforms can be introduced without requiring new system interfaces. While there are various serverless storage services that can be used with FaaS [60], in PCSI the interface between compute and state is deeply integrated into the model.

3.2 State

State in PCSI encompasses all information that is preserved beyond the lifetime of a single task, or that is transmitted outside of the scope of a single task. Access to state in PCSI is always explicit, which means that functions always access state over system interfaces. Our design centers around a few key principles:

- **Universal Storage Interface:** Applications interact with state through a common interface. This ensures that the system has full visibility into communication and storage patterns, allowing it to optimize scheduling and placement, and to provide fault-tolerance. This also provides a clear division between application and system, enabling implementations to evolve over time.
- **Everything is a File:** If applications must use a common state interface, then that interface must be able to express the wide range of functionality available in the cloud. We achieve this in much the same way as UNIX and its descendants [56, 57], by allowing various implementations of file system objects. While some objects may represent persistent data, others may represent network connections or interfaces to system services.
- **Simple Consistency Menu:** Cloud storage services offer a range of consistency models, and we can be sure that there is no "one size fits all" choice. We propose supporting just two consistency models, a strong one and a weak one, along with configurable restrictions on object mutability.

Objects in PCSI comprise several basic types including directories, regular files, FIFOs, sockets, and device interfaces to system services. This is analogous to POSIX, though the behaviors of each object type are somewhat different (see Section 3.3).

References are the primary method for accessing objects as names are optional in PCSI. References also provide a capability-oriented security mechanism, as Capsicum does for POSIX file descriptors [80]. PCSI makes object reachability explicit. An object is only accessible by functions that hold a reference to it or to a namespace containing it. In clear clear contrast to web services, references make the PCSI API stateful. One benefit is that object access possibilities are known and constrained, opening opportunities for optimization. Another benefit is automated resource reclamation for unreachable objects.

Naming in PCSI provides a secondary access method and a mechanism for indirection. PCSI has no global namespace, but rather each function has a directory object as its file system root. Functions access multiple namespaces via directories passed as arguments. File system layering has proven valuable in building cloud applications, e.g., it is one of the key features provided by Docker [9]. PCSI will include support for union file systems [85], allowing one namespace to be superimposed on top of another.

PCSI only describes an interface to state, underlying implementations may vary. For example, the cloud provider may use any type of underlying storage medium, or a combination of several of them, to meet target performance, cost, and availability criteria. This could mean storage on disk in multiple data centers or keeping just one copy in the memory of a GPU. The latter case exemplifies how an efficient PCSI implementation can keep keep associated compute and state resources close together, even though the abstract model separates them.

3.3 Concurrency and Consistency

Reconciling consistency with performance and availability is one of the persistently vexing challenges in distributed

systems [2, 12]. We acknowledge it will likely remain an active research area for some time to come and design PCSI with this in mind. We provide limited options that allow applications to choose between well understood paradigms, and are careful to remove implementation concerns from the interface.

PCSI allows objects to be configured to one of four mutability levels. These levels and the transitions allowed between them are shown in Figure 1. `IMMUTABLE` objects can be implemented with the proven efficiency and scalability of cloud object storage whereas `MUTABLE` objects allow more flexibility to applications that require it. Intermediate levels can still offer improved performance, e.g., once written, the content of an `APPEND_ONLY` object may be safely cached anywhere.

Operations against objects can execute at one of two consistency levels: linearizability [35] and eventual consistency [72, 76]. This sort of configurable consistency can be provided through quorum systems like DynamoDB [68], though we deliberately hide mechanism details like quorum sizes from the application.

We also believe that the separation of compute and state, a foundational assumption of PCSI, is at odds with some intermediate consistency models. For example, CRDTs [65] and lattice-based approaches [21, 22, 86] require the state management system to support a merge operation, in effect blending the notions of state and computation. We believe such techniques will play an important role in the cloud, however their implementations should be largely parallel to PCSI, as we discuss next.

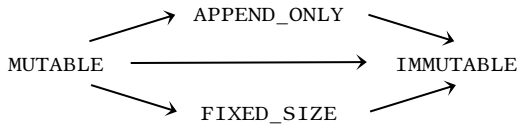

Figure 1: Allowable object mutability transitions.

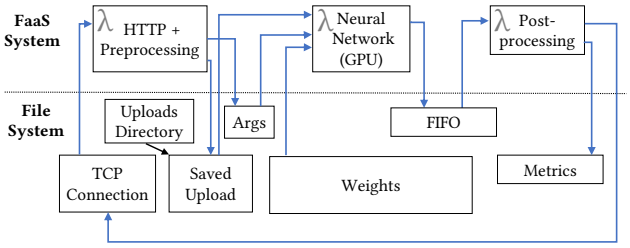

Figure 2: Model serving pipeline with separation of compute and state.

3.4 Limitations

In system design, what is *not* included is just as important as what is. We believe that PCSI will enable a broad range of cloud workloads, but we do not believe that all workloads will run well as a collection of functions interacting with one another and with the storage layer. Yet even those applications that run best with a server-based implementation can be integrated with the PCSI—we allow them to be invoked just like any other function. Things like OLTP databases and key-value stores benefit from detailed control over system resources [70], and can appear as part of a universal abstraction. The same is true of certain scientific computing applications and machine learning training systems, which can benefit from precisely coordinated scheduling and application specific network topology.

4 DISCUSSION

As a derivative of current serverless offerings, our design inherits the benefits of pay-per-use, simplified deployment, and autoscaling. It also benefits from being an evolutionary, rather than a revolutionary, path from current systems to as-yet unconceived improvements.

To understand PCSI's benefits further, we now review an example application: serving deep learning models. Figure 2 shows a pipelined composition of three FaaS functions. The first runs in response to input on a TCP connection and decodes an incoming HTTP request, including streaming user image uploads to a file. Next is a GPU-enabled prediction function which operates on the uploaded file. It also takes as input the model weights, which rarely change but need to be updated with strong consistency and replicated widely. Finally the output is sent through a FIFO to a post-processing function, which then uses the original TCP object to complete the HTTP response.

4.1 Making it Fast

While the abstractions in PCSI are designed to support distributed systems, logical disaggregation does not imply physical disaggregation [69]. A naive implementation might send intermediate data from the preprocessing function to remote storage before pulling it onto a remote GPU to run the model. However, a more sophisticated implementation could use knowledge of application behavior to make much better decisions [10]. Since the task graph indicates that these two functions will be composed, the system can schedule the first CPU function on a physical server that also contains a GPU. Since data were intended only for the next task (explicit inputs and outputs), data movement is reduced to a single `cudaMemcpy`. This implementation would achieve performance similar to a monolithic server-based service.

4.2 Making it Efficient

In the previous example, we described how user applications could be run with little performance overhead. However, being fast is not enough. After all, a dedicated bare-metal cluster would also be fast. The logically disaggregated design of PCSI also enables other optimization targets like cost or resource efficiency, even at the expense of performance. Rather than wait for a large enough server to handle the entire graph, the provider is free to scavenge underutilized resources from around the cluster for each function independently. Even though this may affect performance, it makes much more efficient use of expensive resources. Fortunately, the concept of "good enough" performance is prevalent in cloud workloads. Many applications come with service level objectives (SLOs) that stipulate a maximum acceptable latency, and experience little or no benefit from lower latency [43].

More generally, PCSI enables flexible scheduling and scaling of resources. Preprocessing functions can be scaled independently of the GPU-enabled model functions, precisely matching resource demands, even under rapidly varying load or skew. Since functions can be specialized to resource types, we can develop specialized hardware platforms with tailored thermal, packaging, and networking designs. Existing platforms like Microsoft's Catapult [18] or Google's TPU Pods [28] suggest that significant advantages can come from such specialization. While these existing systems require a specialized software environment, PCSI offers a unified interface that would enable more rapid development and deployment of specialized hardware platforms.

4.3 Making it Flexible

Cloud platforms launch new products at a rapid pace. Any successful cloud interface needs to be flexible enough to integrate new technologies and techniques with minimal application changes.

On the data side, we notice that our application has multiple inputs and outputs with differing consistency requirements, say strong consistency for model weights and eventual consistency for the upload archive and user metrics. PCSI supports these needs through a single unified interface.

While our example application utilized GPUs to execute the neural network, hardware for machine learning is advancing quickly [19, 39]. To take advantage of the latest accelerator, PCSI developers may need to modify their neural network function implementation, but the rest of the application would remain unchanged. It is not just the accelerators themselves that can see advancements. New hardware integration technologies are being developed that provide them with efficient memory hierarchies and networking support [18, 27, 28, 32, 83]. Since state management is explicit, the PCSI implementation can integrate these new technologies without requiring application changes. Even the non-accelerated functions can benefit from operating system advancements like unikernels [40, 84].

We observe that cloud-native interfaces can naturally take advantage of cloud-native hardware and operating systems while traditional interfaces are far more difficult to adapt.

5 THE PATH FORWARD

An abstraction is not useful if it is never deployed. Ultimately, we hope to see a common core of cloud interfaces that helps extend and sustain innovation. The path to a common model will be driven by user demands and open collaboration. We have seen successes before. Kubernetes [13] adoption has grown quickly, with user demand leading cloud vendors to release their own hosted Kubernetes services. Further afield, the computer architecture community has escaped the bonds of proprietary ISAs by coming together around the RISC-V open source ISA [5], enabling an explosion of industrial and academic innovation. We will need to learn from these experiences if we wish to have similar success.

In the immediate future, this means continuing to develop the core technologies and interfaces underlying the PCSI approach. The authors are currently building some of these components including serverless interfaces to GPUs, and file systems for cloud functions. Other challenges remain to be addressed. Are the proposed consistency models sufficient? Will existing security models for the cloud and warehouse-scale computers suffice or are new strategies needed [59, 62, 80]? Can techniques to drive performance and utilization of accelerators be broadened to a general multi-tenant setting [23, 30, 58, 67]? As these and other questions are answered, existing serverless offerings can evolve toward a common portable cloud system interface.

6 CONCLUSION

"What got you here won't get you there"
- Marshall Goldsmith

The cloud is a unique platform. The warehouse scale computers that power it are nothing like the individual servers that comprise it, but they also bear little resemblance to the global internet, the distributed system from which many of their technologies are drawn. A well defined core system interface for the cloud could unlock a great deal of innovation. Much as POSIX and REST brought sanity to their respective environments, a portable cloud system interface can tame the wild-west of cloud programming. The recent growth of serverless computing demonstrates that the community is ready and willing to redesign their applications around truly cloud-native interfaces—let's give them one.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their thoughtful feedback. This research was supported by NSF CISE Expeditions Award CCF-1730628 and gifts from Amazon Web Services, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, Nvidia, Scotiabank, Splunk and VMware.

REFERENCES

[1] [n.d.]. OpenStack. https://www.openstack.org/.

[2] Daniel Abadi. 2012. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer* 45, 2 (2012), 37–42.

[3] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. 1985. The Eden system: A technical review. *IEEE Transactions on Software Engineering* 1 (1985), 43–59.

[4] Thomas E. Anderson, David E. Culler, and David Patterson. 1995. A case for NOW (networks of workstations). *IEEE micro* 15, 1 (1995), 54–64.

[5] Krste Asanović and David A. Patterson. 2014. *Instruction Sets Should Be Free: The Case For RISC-V.* Technical Report UCB/EECS-2014-146. EECS Department, University of California, Berkeley. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[6] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. *Commun. ACM* 60, 4 (2017), 48–54.

[7] Luiz André Barroso, Jeffrey Dean, and Urs Holzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE micro* 23, 2 (2003), 22–28.

[8] Luiz André Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2018. The datacenter as a computer: Designing warehouse-scale machines. *Synthesis Lectures on Computer Architecture* 13, 3 (2018), i–189.

[9] David Bernstein. 2014. Containers and cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (2014), 81–84.

[10] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. 2012. Shredder: GPU-accelerated incremental storage and computation.. In *FAST*, Vol. 14. 14.

[11] Andrew D. Birrell and Bruce Jay Nelson. 1984. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)* 2, 1 (1984), 39–59.

[12] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (2012), 23–29.

[13] Eric A. Brewer. 2015. Kubernetes and the path to cloud native. In *Proceedings of the sixth ACM symposium on cloud computing*. 167–167.

[14] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes: Lessons learned from three container-management systems over a decade. *Queue* 14, 1 (2016), 70–93.

[15] Rajkumar Buyya, Toni Cortes, and Hai Jin. 2001. Single system image. *The International Journal of High Performance Computing Applications* 15, 2 (2001), 124–135.

[16] Michael Cafarella, David DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. 2020. DBOS: A Proposal for a Data-Centric Operating System. *arXiv preprint arXiv:2007.11112* (2020).

[17] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2019. The rise of serverless computing. *Commun. ACM* 62, 12 (2019), 44–54.

[18] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. 2016. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1–13.

[19] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[20] David Cheriton. 1988. The V distributed system. *Commun. ACM* 31, 3 (1988), 314–333.

[21] Alvin Cheung, Natacha Crooks, Matthew Milano, and Joseph M. Hellerstein. 2021. New directions in cloud programming. *CIDR* (2021).

[22] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*. 1–14.

[23] Feras Daoud, Amir Watad, and Mark Silberstein. 2016. GPUrdma: GPU-side library for high performance networking from GPU kernels. In *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ACM, 1–8. https://doi.org/10.1145/2931088.2931091

[24] Partha Dasgupta, Richard J. LeBlanc, Mustaque Ahamad, and Umakishore Ramachandran. 1991. The Clouds distributed operating system. *Computer* 24, 11 (1991), 34–44.

[25] Xinyang Feng, Jianjing Shen, and Ying Fan. 2009. REST: An alternative to RPC for Web services architecture. In *2009 First International Conference on Future Information Networks*. IEEE, 7–10.

[26] Roy T. Fielding. 2000. *Architectural styles and the design of network-based software architectures*. Vol. 7. University of California, Irvine Irvine.

[27] Gen-Z Consortium. 2018. *Gen-Z Overview*. Technical Report. Gen-Z Consortium. https://genzconsortium.org/wp-content/uploads/2018/05/Gen-Z-Overview-V1.pdf

[28] Google 2021. *Cloud TPU - Documentation - System Architecture*. Google. https://cloud.google.com/tpu/docs/system-architecture

[29] Cary Gray and David Cheriton. 1989. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. *ACM SIGOPS Operating Systems Review* 23, 5 (1989), 202–210.

[30] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. 2020. Serving DNNs like Clockwork: Performance predictability from the bottom up. *arXiv:2006.02464 [cs]* (Jun 2020).

[31] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 185–200.

[32] Mark Harris. 2017. *NVIDIA DGX-1: The Fastest Deep Learning System*. Technical Report. Nvidia. https://developer.nvidia.com/blog/dgx-1-fastest-deep-learning-system/

[33] Rober Haskin, Yoni Malachi, and Gregory Chan. 1988. Recovery management in QuickSilver. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 82–108.

[34] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless computing: One step forward, two steps back. *CIDR* (2019).

[35] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 463–492.

[36] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. 2011. Mesos: A platform for fine-grained resource sharing in the data center.. In *NSDI*, Vol. 11. 22–22.

[37] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. 1988. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 51–81.

[38] Andrew Josey, Eric Blake, Geoff Clare, et al. 2018. The Open Group base specifications issue 7. https://pubs.opengroup.org/onlinepubs/9699919799/.

[39] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[40] Ricardo Koller and Dan Williams. 2017. Will serverless end the dominance of Linux in the cloud?. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. 169–173.

[41] Collin Lee and John Ousterhout. 2019. Granular Computing. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 149–154.

[42] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST?. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[43] Jeffrey C. Mogul and John Wilkes. 2019. Nines are not enough: Meaningful metrics for clouds. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 136–141.

[44] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, et al. 2018. Ray: A distributed framework for emerging AI applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 561–577.

[45] Sape J. Mullender, Guido Van Rossum, AS Tananbaum, Robbert Van Renesse, and Hans Van Staveren. 1990. Amoeba: A distributed operating system for the 1990s. *Computer* 23, 5 (1990), 44–53.

[46] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. Ciel: A universal execution engine for distributed data-flow computing. In *Proc. 8th ACM/USENIX Symposium on Networked Systems Design and Implementation*. 113–126.

[47] Roger Michael Needham and Andrew J. Herbert. 1983. The Cambridge distributed computing system. (1983).

[48] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. 1988. Caching in the Sprite network file system. *ACM Transactions on Computer Systems (TOCS)* 6, 1 (1988), 134–154.

[49] Bill Nitzberg and Virginia Lo. 1991. Distributed shared memory: A survey of issues and algorithms. *Computer* 24, 8 (1991), 52–60.

[50] Gian Ntzik, Pedro da Rocha Pinto, Julian Sutherland, and Philippa Gardner. 2018. A concurrent specification of POSIX file systems. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[51] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. 1988. The Sprite network operating system. *Computer* 21, 2 (1988), 23–36.

[52] Kay Ousterhout, Christopher Canel, Sylvia Ratnasamy, and Scott Shenker. 2017. Monotasks: Architecting for performance clarity in data analytics frameworks. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 184–200.

[53] David L. Parnas. 1972. On the criteria to be used in decomposing systems into modules. In *Pioneers and Their Contributions to Software Engineering*. Springer, 479–498.

[54] Larry Peterson, Scott Baker, Marc De Leenheer, Andy Bavier, Sapan Bhatia, Mike Wawrzoniak, Jude Nelson, and John Hartman. 2015. XOS: An extensible cloud operating system. In *Proceedings of the 2nd International Workshop on Software-Defined Ecosystems*. 23–30.

[55] Fabio Pianese, Peter Bosch, Alessandro Duminuco, Nico Janssens, Thanos Stathopoulos, and Moritz Steiner. 2010. Toward a cloud operating system. In *2010 IEEE/IFIP Network Operations and Management Symposium Workshops*. IEEE, 335–342.

[56] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. 1995. Plan 9 from Bell Labs. *Computing systems* 8, 3 (1995), 221–254.

[57] Dennis M. Ritchie and Ken Thompson. 1974. The UNIX Time-Sharing System. *Communications* (1974).

[58] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2019. INFaaS: A Model-less Inference Serving System. *arXiv:1905.13348 [cs]* (Sep 2019).

[59] Ravi S. Sandhu. 1998. Role-based access control. In *Advances in computers*. Vol. 46. Elsevier, 237–286.

[60] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What serverless computing is and should become: The next phase of cloud computing. *Commun. ACM* 64, 5 (2021), 55–63.

[61] Frank Schmuck and Jim Wylie. 1991. Experience with transactions in QuickSilver. In *ACM SIGOPS Operating Systems Review*, Vol. 25. ACM, 239–253.

[62] Malte Schwarzkopf. 2015. *Operating system support for warehouse-scale computing*. Ph.D. Dissertation. University of Cambridge.

[63] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. 2013. New wine in old skins: The case for distributed operating systems in the data center. In *Proceedings of the 4th Asia-Pacific Workshop on Systems*. 1–7.

[64] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. 2018. LegoOS: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 69–87.

[65] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[66] Simon Shillaker and Peter Pietzuch. 2020. Faasm: lightweight isolation for efficient stateful serverless computing. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 419–433.

[67] Mark Silberstein. 2017. OmniX: An accelerator-centric OS for omni-programmable systems. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems - HotOS '17*. ACM Press, 69–75.

[68] Swaminathan Sivasubramanian. 2012. Amazon DynamoDB: A seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 729–730.

[69] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. 2020. Cloudburst: Stateful Functions-as-a-Service. *Proceedings of the VLDB Endowment* 13, 11 (2020).

[70] Michael Stonebraker. 1981. Operating system support for database management. *Commun. ACM* 24, 7 (1981), 412–418.

[71] Andrew S. Tanenbaum and Robbert Van Renesse. 1985. Distributed operating systems. *ACM Computing Surveys (CSUR)* 17, 4 (1985), 419–470.

[72] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. *ACM SIGOPS Operating Systems Review* 29, 5 (1995), 172–182.

[73] Muhammad Tirmazi, Adam Barker, Nan Deng, Md E. Haque, Zhijing Gene Qin, Steven Hand, Mor Harchol-Balter, and John Wilkes. 2020. Borg: The next generation. In *Proceedings of the Fifteenth European Conference on Computer Systems*. 1–14.

[74] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. 1–17.

[75] Werner Vogels. 2003. Web services are not distributed objects. *IEEE Internet computing* 7, 6 (2003), 59–66.

[76] Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44.

[77] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. 1996. A note on distributed computing. In *International Workshop on Mobile Object Systems*. Springer, 49–64.

[78] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. 1983. The LOCUS distributed operating system. *ACM SIGOPS Operating Systems Review* 17, 5 (1983), 49–70.

[79] Randolph Y. Wang and Thomas E. Anderson. 1993. xFS: A wide area mass storage file system. In *Proceedings of IEEE 4th Workshop on Workstation Operating Systems. WWOS-III*. IEEE, 71–78.

[80] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. 2012. A taste of Capsicum: Practical capabilities for UNIX. *Commun. ACM* 55, 3 (2012), 97–104.

[81] David Wentzlaff and Anant Agarwal. 2009. Factored operating systems (fos) the case for a scalable operating system for multicores. *ACM SIGOPS Operating Systems Review* 43, 2 (2009), 76–85.

[82] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, and Anant Agarwal. 2010. An operating system for multicore and clouds: Mechanisms and implementation. In *Proceedings of the 1st ACM symposium on Cloud computing*. 3–14.

[83] Bruce Wile. 2014. *Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems*. Technical Report. IBM Systems and Technology Group.

[84] Dan Williams and Ricardo Koller. 2016. Unikernel monitors: Extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*.

[85] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyer Zubair. 2006. Versatility and Unix semantics in namespace unification. *ACM Transactions on Storage (TOS)* 2, 1 (2006), 74–105.

[86] Chenggang Wu, Jose Faleiro, Yihan Lin, and Joseph Hellerstein. 2019. Anna: A KVS for any scale. *IEEE Transactions on Knowledge and Data Engineering* (2019).

[87] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joseph, Randy H. Katz, Scott Shenker, and Ion Stoica. 2011. The datacenter needs an operating system.. In *HotCloud*.