User-Defined Cloud

Yiying Zhang
UCSD

Ardalan Amiri Sani
UC Irvine

Guoqing Harry Xu
UCLA

ABSTRACT

Since its creation, cloud computing has always taken a provider-dictated approach, where cloud providers define and manage the cloud to accommodate the user needs they deem important. We propose “User-Defined Cloud”, or UDC, a new cloud scheme that allows users to define their own “clouds”, by defining hardware resource needs, system software features, and security requirements of their applications, and to do so without the need to build or manage low-level systems.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

user-defined cloud, cloud computing, system customization

ACM Reference Format:
Yiying Zhang, Ardalan Amiri Sani, and Guoqing Harry Xu. 2021. User-Defined 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, 8 pages. https://doi.org/10.1145/3458336.3465304

1 INTRODUCTION

Since the launch of AWS in 2006, the evolution of cloud ecosystems has so far been following a provider-dictated approach summarized in the following three steps: 1) the cloud provider identifies the need to support a new type of application workload or a new type of hardware; 2) the cloud provider develops new software and/or adapts an existing software/hardware infrastructure to support the need; and 3) the cloud provider launches a new service or a variation of an existing service to integrate the new hardware/software. This approach has successfully transformed the cloud from a niche market into a dominant computing platform that empowers small and large organizations to run their businesses at scale.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HotOS ’21, May 31–June 2, 2021, Ann Arbor, MI, USA
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8438-4/21/05.
https://doi.org/10.1145/3458336.3465304

Today’s cloud platforms see an unprecedentedly rapid growth in both *workload diversity* and *hardware heterogeneity*. On the one hand, a broader community of users are embracing the cloud and many come from domains with specific needs of efficiency, scalability, and security. Some example domains are military [1], hospitals [38], farms [24], financial firms [11], entertainment companies [13], and police departments [12]. On the other hand, new hardware devices emerge at an unprecedented speed. Examples are TPU [23] and other ASICs [4, 33] for compute acceleration, Optane [15, 20] and 3D-stacked [31] memory for making memory persistent and high-bandwidth, and programmable [9]/circuit [35] switches/NICs for offloading computation to network infrastructures and making them cheaper.

Looking forward, a critical question we ask is: would the aforementioned cloud evolution pattern still work? In particular, it boils down to the following two subquestions: 1) from the perspective of cloud users, would they be all happy with *a number of services each with a fixed configuration* even if new services keep getting added? 2) from the perspective of cloud providers, would they be able (and willing) to customize their infrastructures in a timely fashion when new hardware and workloads quickly emerge? Unfortunately, today’s cloud computing model falls short of both aspects.

From the cloud users’ perspective, there are three major issues. First, users pay for extra (35% according to [14]) computing resources they do not need because no cloud service matches their precise needs. For instance, to use 8 GPUs in a VM to run a big machine-learning workload, AWS users must select an EC2 p3.16xlarge or p3dn.24xlarge instance, which come with 64 and 96 vCPUs, respectively, even if they need only a small number of vCPUs to run the GPU orchestration software. Second, niche domain users are unable to run their workloads as desired in the cloud, often because the cloud does not provide the right combination of hardware or is too slow in incorporating new hardware features into their services. For example, many ML inference tasks are event-triggered and could benefit from serverless computing *and* GPU acceleration. Despite the high demand for such applications, no cloud provider has yet supported GPU in their serverless computing offerings. Finally, users cannot properly specify their security needs and they have to trust the cloud provider. Unfortunately, providing enhanced security and strong isolation often comes at the cost of reduced resource utilization or performance. As a result, security is often sacrificed, resulting in severe compromises and data

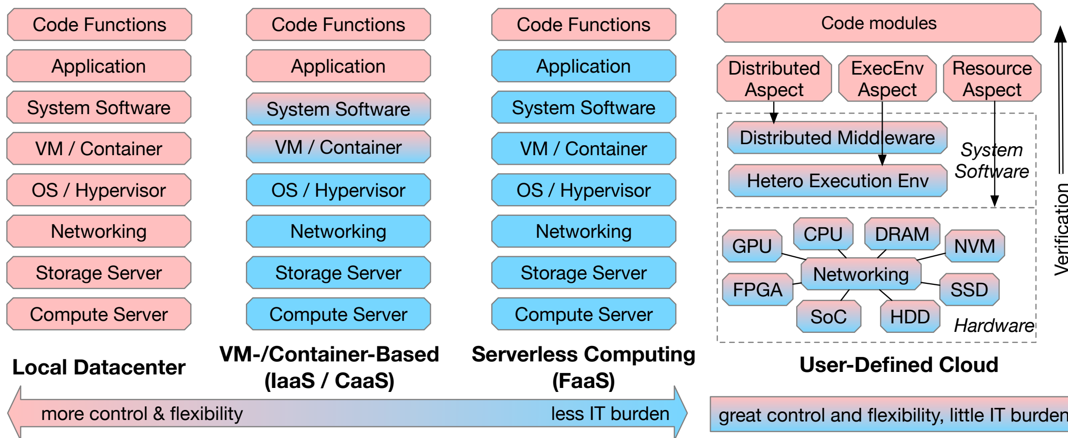

Figure 1: **UDC Architecture.** *Red boxes represent user-defined and user-managed modules. Blue boxes represent cloud-defined and cloud-managed modules. Red-blue boxes represent user-defined and cloud-managed modules.*

breaches [34]. For all the above cases, users have to build a dedicated cluster with their desired features, incurring high development and maintenance costs.

The way today's clouds run also creates problems for cloud providers. On the one hand, when there is new hardware to deploy or a security feature to add, the cloud provider needs to integrate them into every single one of its existing services. On the other hand, launching a new service dictates that the service must be compatible with different types of hardware, system software, and security features that users would want to access. These two problems collectively create a *"cloud DevOps matrix from hell"*, similar to the DevOps matrix from hell that motivated the creation of containers [22]. Every time a change is about to be made on the cloud, the provider must go through this matrix from hell, incurring exceedingly high development costs and slowing down the time to market.

We observe that the root cause of the aforementioned issues is that it is always the *cloud providers that define and manage the cloud* to accommodate the user needs *they deem popular*. The mere role of cloud users is to use the predefined cloud, *as is*. However, users are the ones who understand their workloads and know what is needed to run them. In this paper, we argue that *users can and should define their own clouds*. Cloud providers continue to create and manage the cloud, but in a way that is flexible enough for users to customize it. To be more specific, **each user defines what computing resources and features of these resources the cloud should provide** for their own workloads, and **cloud providers take care of *how* these resources are provided** by supplying software and hardware infrastructures under the hood. In doing so, users can actively customize software and hardware in a public cloud, and they only need to understand what their workloads need, as opposed to how to meet these needs with predefined service types. In the meantime, cloud providers only need to build a customizable (software and hardware) infrastructure that allows users to create their own "services", as opposed to tirelessly adding services for each emerging user group.

Building on this insight, we propose *User-Defined Cloud*, or *UDC*, a new cloud scheme that allows a user to define 1) the hardware resources they need to run their workloads (*e.g.*, number of CPU cores, type and number of GPUs, and amount of memory) *in arbitrary combinations and amounts*, 2) the execution environment and security requirements for their workloads (*e.g.*, the level of isolation, confidentiality, and integrity), and 3) system features for running their workloads in a distributed way (*e.g.*, the degree of replication, consistency level, and failure handling strategy). Furthermore, we propose a fine-grained approach that allows users to define what resources and features *each individual stage* of their workloads need, instead of claiming (and paying for) excessive resources for the entire workload. Based on these specifications, the cloud provider puts together a cloud service for the user *on the fly*, which includes the desired set of hardware resources, system configurations, as well as security features.

One implication of UDC is that when users have more freedom in defining their own "cloud", they need to understand both their own applications and to some extent, what computing resources and features they need. We envision a typical user of UDC to have a division of responsibility within them: application developers who write applications in a modularized way that fits their application, and a small IT team that defines various UDC specifications for each module. Users could also choose to not define any specifications, in which case the cloud provider makes the decisions instead (*i.e.*, falling back to today's cloud).

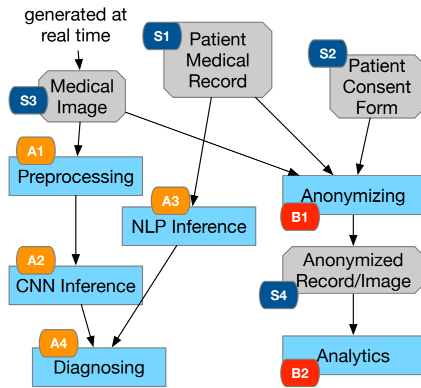

Figure 2: An Example of Medical Information Processing. *Each node represents a compute task or a data component, and arrows represent data flows.*

	Resource	Exec Env & Security	Distributed
A1	Fastest	Single-tenant (or SGX encalve if CPU)	No replication
A2	GPU	Single-tenant	No rep, Checkpoint
A3	GPU	Single-tenant	No rep, Checkpoint
A4	CPU	Single-tenant & SGX enclave	Rep 2×, Checkpoint
B1	Cheapest	Single-tenant (or SGX encalve if CPU)	No replication
B2	Cheapest	Containers	No rep, Checkpoint
S1	SSD	Encryption & integrity protection	Replicate 3×, Sequential consistency
S2	Cheapest	Encryption & integrity protection	Replicate 2×, Reader preference
S3	DRAM	Encryption & integrity prot.	Replicate 2×
S4	Cheapest	Integrity protection	No replication, Release consistency

Table 1: Example of User Definition. *Corresponds to Figure 2.*

2 A CASE FOR USER-DEFINED CLOUD

Figure 1 illustrates the high-level differences between UDC and existing cloud schemes. In today's clouds, VM-/container-based services (also known as IaaS and CaaS) give cloud users more (but not complete) control over how to run their workloads, but they require huge efforts from users to manage their IT. PaaS (Platform as a Service) and serverless computing eliminate most of the IT cost but leave users with no way to control their workloads. Our proposal takes a radically different approach to solving the pain points in today's cloud computing: *giving control to cloud users and keeping management for cloud providers at all the layers* in the cloud ecosystem — from hardware to system software and security. Overall, UDC will provide tremendous benefits to both cloud users and cloud providers in the following ways.

Benefits to cloud users. Users can choose[1] to customize the entire stack from software to hardware *of a public cloud* in a way that matches exactly what they need. They can quickly build their own cloud environments to launch their applications without waiting for cloud providers to create the service that may or may not fully meet their needs. Neither do they need to build or maintain any customized cluster software or hardware. Moreover, users obtain and pay only for the resources and features they need, instead of predefined packages that contain unnecessary resources. Finally, security can be granted according to the needs of the users, enabling security-critical applications to move to the cloud.

Benefits to cloud providers. By decoupling different layers in the software-hardware stack and allowing users to define each of them separately, cloud providers can independently add/remove a hardware/software feature without the need to change the rest of the system, essentially avoiding the cloud DevOps matrix from hell and saving development cost. Moreover, UDC enables significantly more users to use the cloud, which directly creates revenue for cloud providers. Indeed, supporting these new customers comes at almost no additional cost for cloud providers, as providers do not need to create new instance types or services for them. Finally, with UDC, although cloud providers cannot charge users for the resources they do not use, they can increase the unit price of their computing resources to the extent that still offers users a lower total cost than today's cloud. Moreover, without resource wastes, providers could potentially consolidate more applications to the same amount of computing resources and shutting down the remaining ones.

A motivating example. We motivate UDC using the following case in the healthcare industry (Figure 2). Table 1 shows the corresponding specifications for UDC, which we expect the user to create. To use UDC, there are two tasks the user must perform.

- Application Semantics Development: we expect a user *development team* to develop application code in the form of *modules* and specify the module relationship as shown in Figure 2 (more details in §3.1).
- UDC Aspect Specification: we expect a user *IT team* to specify how each module should be executed on UDC (*e.g.*, resource demands, security requirements, and distributed semantics).

In this example, a hospital wants to use the cloud to perform three tasks: securely storing patients' medical records, securely and quickly diagnosing patients' medical images (*e.g.*, CT scans), and occasionally performing analytics over anonymized patient data (*e.g.*, results of clinical trials).

First, the hospital needs to securely store all patients' medical records (S1, *e.g.*, previous diagnoses in a natural language) and consent forms (S2, *e.g.*, whether or not a patient is okay with providing their medical records, after anonymization, for research) in the cloud.

Second, the hospital performs image-based auto-diagnosis in a secure manner in the cloud. When a medical image is

[1] Users can also choose to not define one or more layers, in which case we fall back to traditional cloud solutions.

taken, it is sent to the cloud (S3), which launches a process of automatic diagnosis. This process involves A1 pre-processing of the medical image (*e.g.*, resizing and greyscaling), A2 object detection (*e.g.*, CNN inference) on the pre-processed image, A3 retrieving the patient's medical record and performing natural language analysis (*e.g.*, BERT inference) to automatically generate relevant previous diagnosis, and A4 automated diagnosis with detected objects and results of NLP on the medical record. All these steps must be done securely.

Finally, the hospital performs data analytics over anonymized patient records. This involves B1 fetching consent forms, filtering records/images based on user consents, and anonymizing them, and S4 passing the anonymized records/images to a data-processing system (possibly a third-party framework) for analytics (B2). The first step must be secured (*i.e.*, confidentiality and integrity), while the last step does not need security (although integrity-protection might be desirable).

3 A PROPOSAL OF UDC

We now offer one proposal for UDC; we do not claim it to be the only way to realize UDC or that it solves all challenges. Overall, our ideas center around three design principles.

Design Principle 1: Expressing definitions of low-level layers as runtime *aspects*. Today, programmers write their cloud applications in languages designed for local machines, which are then compiled and tested in a local (non-cloud) environments. A separate group (either an IT team or the cloud provider) builds the underlying systems and hardware to execute these applications. As a result, there is no way to customize how applications run on the cloud *with application-specific knowledge*. We propose a new *cloud-native application development model*, by exposing low-level system definitions to application developers in a way that is tied but orthogonal to application semantics. Specifically, programmers develop *program modules* based on application semantics. They (or a separate IT team) can then define the desired features of each module as different *aspects* of it (*i.e.*, inspired by aspect orientation [25, 40]). We include three types of aspects: 1) hardware resource demands, 2) execution environments including security specifications, and 3) distributed semantics. These aspects will be fed to the cloud runtime, which customizes the infrastructure, runs the program, collects the feedback, and performs adaptive optimizations.

Design Principle 2: Decouple specifications from their realization and decouple different aspects. When allowing the IT team to define aspects of their intended UDC, we should not expect them to implement these aspects (*i.e.*, build or manage low-level systems). Thus, we propose to let the IT team specify aspects in a declarative way and to decouple these specification from their low-level implementation. The cloud provider is the party responsible for the implementation, and they can choose different ways to implement a specification. Furthermore, we propose to decouple the three types of aspects. The user's IT team can freely define one aspect without changing others, and they can also choose to not define an aspect (*i.e.*, fall back to provider's default).

Design Principle 3: Fine granularity at each layer. To allow users to freely define and associate aspects to their applications, we argue to make *every layer of the application-software-hardware stack fine-grained*. Each fine-grained piece can be independently declared, configured, and managed. Users can then choose their desired combination of pieces and put them together to run their applications, similar to building Lego toys. Decomposing a layer into fine-grained pieces improves flexibility and resource utilization, but it increases the scale of hardware, system software, and user code that the cloud provider must manage. To tackle this challenge, we propose to *vertically* bundle layers of fine-grained pieces into a self-sustained *resource unit*. For example, we can combine some amount of compute resources (*e.g.*, a CPU core), an execution environment (*e.g.*, a container), and some distributed API library into one *low-level resource unit* for allocation, scheduling, and failure handling. We also propose to bundle a fine-grained code/data module and its aspects into a *high-level object*, which can be executed on one or more resource units. This vertical bundling reduces the complexity and overhead of resource management.

A recent proposal, Hydro [10], advocates a new cloud programming model that lets users specify four "facets" in a declarative way, including programming semantics, availability, consistency, and targets of optimization. Unlike Hydro, which focuses on new programming models and programming language supports on top of existing cloud infrastructures, UDC aims to support existing programming models but with a much more disruptive approach underneath these programming models: making every layer in the cloud infrastructure customizable and fine grained.

3.1 Specifying Application Semantics

What can users define? To use UDC, a user's development team write programs for their application logic. To meet our fine-granularity goal (Principle 3), a user program is expressed as a DAG of modules. A module could be a code block representing a task (*e.g.*, A1 to A4, B1 and B2) or one or more data structures representing a set of data (S1 to S4), and edges across modules represent their dependencies (*e.g.*, one task follows another task, one task module accessing a data module).

A key to efficiently executing workloads on a fine-grained distributed platform (§3.2) is good locality. To this end, we enhance the module DAG representation with locality relationship. For example, developers (or a compiler) can specify computation tasks that should be executed together on the same hardware unit (*e.g.*, A1 and A2). Similarly, they can

also hint that a data object (*e.g.*, S1) is frequently used by a computation task (*e.g.*, A3). Such information will be used to guide our runtime scheduler to make intelligent compute/data placement.

How to achieve the definition? There are many ways to represent an application as a DAG of modules, for example, by compiling an existing program into a DAG of code pieces, by letting users annotate their existing programs to define the boundaries of modules, or by using a new programming model that is native to this representation. For the third option, one promising model could be based on the Actor framework [3, 5–7, 17, 19], which is supported by many popular languages. Each actor represents a module that could run on a hardware resource unit. These (distributed) actors communicate via input and output messages and there is no shared state between actors. Evidence [36, 37] shows that explicit messages are more efficient for a disaggregated setting than shared-memory implementations. Furthermore, messages could be reliably recorded for faster recovery.

To allow developers to use their favorite languages for programming UDC applications, we could build libraries in different languages that offer annotations for expressing module scopes and locality hints, APIs for specifying actor-based operations, and/or language/compiler support for specifying aspects. We will then extend their compilers to compile them into a uniform intermediate representation (in units of *IR modules*) for resource allocation and execution. Our IR is defined as high-level modules and their relationships, *not* low-level code instructions. For example, each language can have a different type of IR module that specifies the execution environment for programs in this language. Different IR modules communicate via well-defined interfaces.

3.2 Defining Hardware Resources

What can users define? UDC allows users to specify the type and amount of computing resources they want/expect each module in their applications to use. However, how can users know their applications' resource usage? On the one hand, the amounts and types of resources that different parts of an application need are related to application logic (*i.e.*, partially known at static time by the application developers). On the other hand, input data also influences the resources needed to run a workload (*i.e.*, only known at runtime).

We believe a viable solution is a combination of developer knowledge, program analysis, and "dry-run" profiling, with the first performed by application developers and the latter two by the IT team or the cloud provider with UDC's tool support. Specifically, we expect the developer team to use their understanding of applications to define the scope of different tasks and to specify a set of possible hardware (*e.g.*, CPU, GPU) or the type of hardware (*e.g.*, compute) that each task *may* need. The IT team or the cloud provider will then use tools that UDC provides (*e.g.*, profilers, cross-platform compilers, *etc.*) to perform dry runs that execute the program with developer-supplied test inputs on different types of hardware within the developer-defined set. The actual resource usage observed for each task is then used as the resource aspect of the task.

How to achieve the user definition? UDC's hardware infrastructure needs to be fine grained to allow users to freely combine resources in the amount and type that they desire. We identify hardware resource disaggregation as the right fit for our goal. Resource disaggregation splits traditional servers into different types of network-attached devices, often organized as *resource pools*. Fulfilling users' resource demands would then simply be allocating the exact amount from the corresponding resource pools (instead of a bin-packing problem with traditional servers). Our runtime scheduler would use the user-supplied resource aspect, execution environment aspect (§3.3), and locality information from the application semantic aspect to decide the location(s) to execute a module and initialize it with the resource amount as user specified. Since user specified resources may be inaccurate when executing with real (and changing) inputs, UDC would perform fine tuning (enlarging or shrinking the amount of resources for a module, migrating modules across hardware units, etc.) based on telemetry data collected at the run time. If a user specifies a set of hardware that a module could potentially execute on or if users only provide a performance/cost goal, then UDC will select resources based on load and available hardware at the run time.

3.3 Defining Execution Environment

What can users define? With UDC, users can define the *execution environment* of their workloads in fine granularity (for each module), without the complexity of managing the environment. In addition, they can specify the *security requirements* of their execution environment without the need to trust cloud providers. They could also specify protection options for their data (*e.g.*, encryption, integrity protection, and replay protection) when these data leave the execution environment (to the network, storage, or another module).

Unlike other UDC aspects, security features should not be specified in a declarative way, as doing so allows cloud providers to choose how to implement the specification. Security features specified by a user should be verifiable by the user in case they do not trust the provider (e.g., for security-critical modules). However, high-level, declarative specifications lack preciseness and hence are hard to verify.

Below are some examples of how users could be more precise when specifying isolation features. For strongest isolation, users could specify a single-tenant Trusted Execution

Environments (TEE) environment. TEEs (*e.g.*, Intel SGX enclaves and AMD SEV VMs) provide protection against system software and physical attacks, while single-tenant execution (where the entire hardware is dedicated to one tenant) protects against hardware-based side-channel attacks [8, 21, 28, 29, 41]. For strong (but not the strongest) isolation, users could choose either TEEs or single-tenant, providing protection against a subset of the aforementioned attacks. For medium-level isolation, users could let providers choose from different options like unikernel [32], lightweight VMs [2], or sandboxed containers [18]. For weak isolation, users could choose containers. The first two options (strongest and strong) can enable verification by the user (§4). The last two require trust in the provider, which provides the system software.

How to achieve the user definition? Many existing execution environments like virtual machines, lightweight VMs, unikernels, containers, and TEEs could be used to fulfill different user requirements. Similarly, existing data confidentiality and integrity measurements could be used. One new challenge is the goal of allowing users to freely combine security/execution features with other aspects such as the resource aspect. For example, today's TEEs only work with CPUs, but with UDC, TEEs need to work with other hardware like GPUs and FPGAs. One possible solution is to add hardware support to specific hardware devices [39]. Another possibility is to create physically-isolated (disaggregated) device clusters that can only be occupied by one tenant at a time and keep data protected when leaving the cluster. Another new challenge relates to the goal of fine granularity. As secure environments are usually slower to start up, (cold) starting many environments for many modules can significantly slow down the entire application. Moreover, single-tenant environments could cause large resource wastes as a module is not likely to occupy the entire hardware unit.

3.4 Defining Distributed Semantics

What can users define? Users should be able to define how their applications run *distributedly*, but without the need to build complex distributed systems. For example, users (developers) can define the failure domains in their programs, with the understanding that different domains could fail independently while code and data within a domain will fail as a whole. The user IT team can then control the availability/reliability of each failure domain by specifying the replication factor, with the understanding that more replicas is more expensive. They can also define how failures are handled for each domain (*e.g.*, whether to re-execute a module or recover from a user-defined checkpoint). Finally, users can define the consistency level of concurrent accesses to their data modules (*e.g.*, sequential consistency), and what type of operations they want to give preferences to (*e.g.*, read preference over write).

How to achieve the user definition? There are a few challenges in realizing the goal of allowing users to freely define fine-grained distributed semantics in a UDC data center. First, users may define conflicting specifications for different modules, *e.g.*, two modules sharing data and one specified as sequential consistency and the other as release consistency. UDC needs to detect such conflicts and either chooses the strictest specification or returns an error to the user.

Second, unlike today's distributed systems that run on a cluster of servers, we expect UDC to run on a cluster of disaggregated devices some of which may not have computation power or could run any software. Thus, traditional software systems that implement distributed protocols would not directly work. A promising direction is to explore the programmability in the network to enforce the distributed specifications [26, 27, 30].

Finally, UDC's fine-grained and fully customizable approaches require a distributed execution environment that is highly scalable and flexible. Existing distributed management frameworks like Kubernetes [16] often take coarse-grained, application-oblivious approaches, *e.g.*, treating a container as the unit of replication, and thus will fall short for UDC.

4 DISCUSSION AND CONCLUSION

We proposed UDC, a radical approach to cloud computing that shifts the control from cloud providers to cloud users. UDC has many promising benefits and research opportunities. At the same, many key challenges remain to be solved. Below we pick four to elaborate further.

Verifying the fulfillment of user definitions. UDC must enable users to verify that the cloud vendor is correctly providing their selected features. This is especially important when the selected features have security implications. We believe this can be achieved through comprehensive remote attestation primitives, similar to the ones available in TEEs today. Using these primitives, users can verify important properties without trusting the vendor and by just trusting the hardware itself (i.e., hardware root of trust). Existing remote attestation primitives can help users verify some of the execution environments and the software running in them. However, many features that UDC allows users to define cannot be verified with today's remote attestation primitives (*e.g.*, whether or not resources were provided as specified).

Supporting legacy software. Most legacy cloud applications can run *as is* on UDC. However, without splitting these programs into smaller modules, their executions would not benefit from the fine-grained treatments UDC enables at each layer, leading to suboptimal performance and/or resource utilization. An interesting idea is to transform them into programs under our model. We could potentially develop static program analysis that performs semi-automated transformation of an existing program by involving developers

in the loop and with the help of a run-time profiler. For example, our static analysis can infer dependencies and cuts a program into segments to minimize the number of cross-segment dependencies, while developers can provide hints on where application semantics transition in their code and a profiling run could capture where resource usage patterns change in the code.

Economics and adoption. For cloud providers to adopt UDC, it needs to have economic incentives for them. At a glance, UDC may seem to impose additional costs for providers as they need to develop a new set of software and hardware infrastructures and UDC's fine-grained approach could incur higher management costs. However, deploying fine-grained application modules on disaggregated clusters would largely improve resource utilization (by 2× as shown by [36]). Meanwhile, providers only need to pay a one-time cost to develop a infrastructure that is flexible to fulfill different user definitions instead of repeatedly developing new services for new user needs. Finally, we expect UDC to enable significantly more workloads to migrate to the cloud, and providers could charge a higher unit price that is still attractive to users since they can tailor their cloud usages and only pay for what is used.

Deployment to existing clouds. Although UDC is a revolutionary idea, we believe that we should take an evolutionary approach to integrating it into existing clouds. Fortunately, today's clouds and data centers are already adopting some of the approaches we propose in UDC. For example, many data centers are already organizing servers into resource pools. Serverless computing and microservices are already making cloud users write modularized code for their applications. Cloud providers could also *partially* adopt UDC, *e.g.*, with a hybrid cluster that contains both regular servers and disaggregated devices; by combining the UDC service with existing cloud services.

Conclusion. This paper proposed User-Defined Cloud, a new cloud-computing paradigm that promises to increase the flexibility and customizability of today's public cloud by allowing users to define the computing resources their applications run on. Although there are many challenges in fully realizing the promises of UDC, we hope that this paper is a good starting point that will motivate future cloud-computing researchers and practitioners.

ACKNOWLEDGMENTS

We thank the HotOS reviewers for their comments. This work is supported by NSF grants CNS-2022675, CNS-1703598, CNS-1763172, CNS-1907352, CNS-2007737, CNS-2006437, CNS-2106838, ONR grants N00014-16-1-2913 and N00014-18-1-2037.

REFERENCES

[1] The Department of Defense and the Power of Cloud Computing - Weighing Acceptable Cost versus Acceptable Risk.

[2] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications . In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, Santa Clara, CA, February.

[3] Gul Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA, 1986.

[4] Amazon Web Services, Inc. AWS Nitro System. https://aws.amazon.com/ec2/nitro/.

[5] William C. Athas and Charles L. Seitz. Multicomputers: Message-passing concurrent computers. *Computer*, 21(8):9–24, August 1988.

[6] Russell Atkinson and Carl Hewitt. Synchronization in actor systems. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 267–280, 1977.

[7] Phil Bernstein, Sergey Bykov, Alan Geller, Gabriel Kliot, and Jorgen Thelin. Orleans: Distributed Virtual Actors for Programmability and Scalability. Technical Report MSR-TR-2014-41, March 2014.

[8] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proc. USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[9] A. Caulfield, P. Costa, and M. Ghobadi. Beyond smartnics: Towards a fully programmable cloud: Invited paper. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018.

[10] Alvin Cheung, Natacha Crooks, Joseph M. Hellerstein, and Matthew Milano. New Directions in Cloud Programming. In *11th Conference on Innovative Data Systems Research (CIDR' 21)*, January 2021.

[11] CloudSecureTech. Cloud Computing in the Financial Industry. https://www.cloudsecuretech.com/cloud-computing-financial-industry/, 12/2016.

[12] DSM. How the Cloud is Helping to Solve Law Enforcement Challenges. https://www.dsm.net/it-solutions-blog/how-the-cloud-is-helping-to-solve-law-enforcement-challenges, 09/2018.

[13] Sophia Fiorino. Cloud technology for TV and filmmakers. https://www.smpte.org/blog/cloud-technology-tv-and-filmmakers, 05/2020.

[14] Flexera Blog. Where Is the $10B in Waste in Public Cloud Costs?, 2017. https://www.flexera.com/blog/cloud/where-is-the-10b-in-waste-in-public-cloud-costs/.

[15] Google. Available first on Google Cloud: Intel Optane DC Persistent Memory. https://tinyurl.com/y4ghuvda.

[16] Google Inc. Kubernetes. http://kubernetes.io/.

[17] Irene Greif and Carl Hewitt. Actor semantics of planner-73. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '75, pages 67–77, 1975.

[18] gVisor. gVisor, Accessed 1/2021. https://gvisor.dev/.

[19] Carl Hewitt, Peter Bishop, Irene Greif, Brian Smith, Todd Matson, and Richard Steiger. Actor induction and meta-evaluation. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 153–168, 1973.

[20] Intel. INTEL OPTANE DC PERSISTENT MEMORY. https://www.intel.com/content/www/us/en/products/memory-storage/optane-dc-persistent-memory.html, 2019.

[21] G. Irazoqui, T. Eisenbarth, and B. Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM sandboxing–and its Application to AES. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2015.

[22] Jerome Petazzoni, dotCloud Inc. LXC, Docker, and the future of software delivery, 2013. Linuxcon.

[23] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[24] Arul karuppannan. The New Age: Cloud Computing In Agriculture Sectors. https://www.w2ssolutions.com/blog/new-age-cloud-computing-in-agriculture-sectors/.

[25] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[26] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, November 2016.

[27] Jialin Li, Jacob Nelson, Ellis Michael, Xin Jin, and Dan R. K. Ports. Pegasus: Tolerating skewed workloads in distributed storage with in-network coherence directories. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, November 2020.

[28] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proc. USENIX Security Symposium*, 2016.

[29] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *Proc. IEEE Symposium on Security and Privacy (S&P)*, 2015.

[30] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, Boston, MA, February 2019.

[31] Gabriel H. Loh. 3D-Stacked Memory Architectures for Multi-core Processors. In *2008 International Symposium on Computer Architecture (ISCA '08)*, Beijing, China, June 2008.

[32] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, Houston, Texas, March 2013.

[33] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor. ASIC Clouds: Specializing the Datacenter. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, August 2016.

[34] National Security Agency. Mitigating Cloud Vulnerabilities, 2020. https://media.defense.gov/2020/Jan/22/2002237484/-1/-1/0/CSI-MITIGATING-CLOUD-VULNERABILITIES_20200121.PDF.

[35] George Porter, Richard Strong, Nathan Farrington, Alex Forencich, Pang Chen-Sun, Tajana Rosing, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Integrating Microsecond Circuit Switching into the Data Center. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (SIGCOMM '13)*, Hong Kong, China, August 2013.

[36] Yizhou Shan, Yutong Huang, Yilun Chen, and Yiying Zhang. Legoos: A disseminated, distributed OS for hardware resource disaggregation. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)*, Carlsbad, CA, October 2018.

[37] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. Distributed shared persistent memory. In *Proceedings of the 8th Annual Symposium on Cloud Computing (SOCC '17)*, Santa Clara, CA, USA, September 2017.

[38] Arkenea Vinati Kamani. 5 Ways Cloud Computing Is Impacting Healthcare. https://www.healthitoutcomes.com/doc/ways-cloud-computing-is-impacting-healthcare-0001, 10/2019.

[39] S. Volos, K. Vaswani, and R. Bruno. Graviton: Trusted Execution Environments on GPUs. In *Proc. USENIX OSDI*, 2018.

[40] Guoqing Xu and Atanas Rountev. Regression test selection for aspectj software. In *ICSE*, pages 65–74, 2007.

[41] Y. Yarom and K. Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proc. USENIX Security Symposium*, 2014.