Towards Modern Development of Cloud Applications

Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker
Parveen Patel, Ivan Posva, Amin Vahdat
Google

Abstract

When writing a distributed application, conventional wisdom says to split your application into separate services that can be rolled out independently. This approach is well-intentioned, but a microservices-based architecture like this often backfires, introducing challenges that counteract the benefits the architecture tries to achieve. Fundamentally, this is because microservices conflate logical boundaries (how code is written) with physical boundaries (how code is deployed). In this paper, we propose a different programming methodology that decouples the two in order to solve these challenges. With our approach, developers write their applications as logical monoliths, offload the decisions of how to distribute and run applications to an automated runtime, and deploy applications atomically. Our prototype implementation reduces application latency by up to 15× and reduces cost by up to 9× compared to the status quo.

ACM Reference Format:
Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker, Parveen Patel, Ivan Posva, Amin Vahdat. 2023. Towards Modern Development of Cloud Applications. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595909

1 Introduction

Cloud computing has seen unprecedented growth in recent years. Writing and deploying distributed applications that can scale up to millions of users has never been easier, in large part due to frameworks like Kubernetes [25], messaging solutions like [7, 18, 31, 33, 40, 60], and data formats like [5, 6, 23, 30]. The prevailing wisdom when using these technologies is to manually split your application into separate microservices that can be rolled out independently.

HOTOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595909

Via an internal survey of various infrastructure teams, we have found that most developers split their applications into multiple binaries for one of the following reasons: (1) It improves *performance*. Separate binaries can be scaled independently, leading to better resource utilization. (2) It improves *fault tolerance*. A crash in one microservice doesn't bring down other microservices, limiting the blast radius of bugs. (3) It improves *abstraction boundaries*. Microservices require clear and explicit APIs, and the chance of code entanglement is severely minimized. (4) It allows for flexible *rollouts*. Different binaries can be released at different rates, leading to more agile code upgrades.

However, splitting applications into independently deployable microservices is not without its challenges, some of which directly contradict the benefits.

- **C1:** It hurts *performance*. The overhead of serializing data and sending it across the network is increasingly becoming a bottleneck [72]. When developers over-split their applications, these overheads compound [55].
- **C2:** It hurts *correctness*. It is extremely challenging to reason about the interactions between every deployed version of every microservice. In a case study of over 100 catastrophic failures of eight widely used systems, two-thirds of failures were caused by the interactions between multiple versions of a system [78].
- **C3:** It is hard to *manage*. Rather than having a single binary to build, test, and deploy, developers have to manage n different binaries, each on their own release schedule. Running end-to-end tests with a local instance of the application becomes an engineering feat.
- **C4:** It freezes *APIs*. Once a microservice establishes an API, it becomes hard to change without breaking the other services that consume the API. Legacy APIs linger around, and new APIs are patched on top.
- **C5:** It slows down application *development*. When making changes that affect multiple microservices, developers cannot implement and deploy the changes atomically. They have to carefully plan how to introduce the change across n microservices with their own release schedules.

In our experience, we have found that an overwhelming number of developers view the above challenges as a necessary part of doing business. Many cloud-native companies are in fact developing internal frameworks and processes that aim to ease some of the above challenges, but not fundamentally change or eliminate them altogether. For example,

continuous deployment frameworks [12, 22, 37] simplify how individual binaries are built and pushed into production, but they do nothing to solve the versioning issue; if anything, they make it worse, as code is pushed into production at a faster rate. Various programming libraries [13, 27] make it easier to create and discover network endpoints, but do nothing to help ease application management. Network protocols like gRPC [18] and data formats like Protocol Buffers [30] are continually improved, but still take up a major fraction of an application's execution cost.

There are two reasons why these microservice-based solutions fall short of solving challenges **C1-C5**. The first reason is that they all assume that the developer manually splits their application into multiple binaries. This means that the network layout of the application is predetermined by the application developer. Moreover, once made, the network layout becomes *hardened* by the addition of networking code into the application (e.g., network endpoints, client/server stubs, network-optimized data structures like [30]). This means that it becomes harder to undo or modify the splits, even when it makes sense to do so. This implicitly contributes to the challenges **C1**, **C2** and **C4** mentioned above.

The second reason is the assumption that application binaries are individually (and in some cases continually) released into production. This makes it more difficult to make changes to the cross-binary protocol. Additionally, it introduces versioning issues and forces the use of more inefficient data formats like [23, 30]. This in turn contributes to the challenges **C1-C5** listed above.

In this paper, we propose a different way of writing and deploying distributed applications, one that solves **C1-C5**. Our programming methodology consists of three core tenets:

(1) **Write monolithic applications** that are modularized into logically distinct components.
(2) Leverage a runtime to dynamically and automatically **assign logical components to physical processes** based on execution characteristics.
(3) **Deploy applications atomically**, preventing different versions of an application from interacting.

Other solutions (e.g., actor based systems) have also tried to raise the abstraction. However, they fall short of solving one or more of these challenges (Section 7). Though these challenges and our proposal are discussed in the context of serving applications, we believe that our observations and solutions are broadly useful.

2 Proposed Solution

The two main parts of our proposal are (1) a *programming model* with abstractions that allow developers to write single-binary modular applications focused solely on business logic, and (2) a *runtime* for building, deploying, and optimizing these applications.

The programming model enables a developer to write a distributed application as a single program, where the code is split into modular units called components (Section 3). This is similar to splitting an application into microservices, except that microservices conflate logical and physical boundaries. Our solution instead decouples the two: components are centered around logical boundaries based on application business logic, and the runtime is centered around physical boundaries based on application performance (e.g., two components should be co-located to improve performance). This decoupling—along with the fact that boundaries can be changed atomically—addresses **C4**.

By delegating all execution responsibilities to the runtime, our solution is able to provide the same benefits as microservices but with much higher performance and reduced costs (addresses **C1**). For example, the runtime makes decisions on how to run, place, replicate, and scale components (Section 4). Because applications are deployed atomically, the runtime has a bird's eye view into the application's execution, enabling further optimizations. For example, the runtime can use custom serialization and transport protocols that leverage the fact that all participants execute at the same version.

Writing an application as a single binary and deploying it atomically also makes it easier to reason about its correctness (addresses **C2**) and makes the application easier to manage (addresses **C3**). Our proposal provides developers with a programming model that lets them focus on application business logic, delegating deployment complexities to a runtime (addresses **C5**). Finally, our proposal enables future innovations like automated testing of distributed applications (Section 5).

3 Programming Model

3.1 Components

The key abstraction of our proposal is the **component**. A component is a long-lived, replicated computational agent, similar to an actor [2]. Each component implements an interface, and the only way to interact with a component is by calling methods on its interface. Components may be hosted by different OS processes (perhaps across many machines). Component method invocations turn into remote procedure calls where necessary, but remain local procedure calls if the caller and callee component are in the same process.

Components are illustrated in Figure 1. The example application has three components: *A*, *B*, and *C*. When the application is deployed, the runtime determines how to co-locate and replicate components. In this example, components *A* and *B* are co-located in the same OS process, and method calls between them are executed as regular method calls. Component *C* is not co-located with any other component

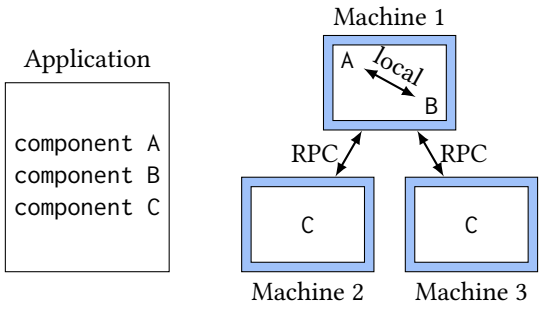

Figure 1: An illustration of how components are written and deployed. An application is written as a set of components (left) and deployed across machines (right). Note that components can be replicated and co-located.

and is replicated across two machines. Method calls on *C* are executed as RPCs over the network.

Components are generally long-lived, but the runtime may scale up or scale down the number of replicas of a component over time based on load. Similarly, component replicas may fail and get restarted. The runtime may also move component replicas around, e.g., to co-locate two chatty components in the same OS process so that communication between the components is done locally rather than over the network.

3.2 API

For the sake of concreteness, we present a component API in Go, though our ideas are language-agnostic. A "Hello, World!" application is given in Figure 2. Component interfaces are represented as Go interfaces, and component implementations are represented as Go structs that implement these interfaces. In Figure 2, the `hello` struct embeds the `Implements[Hello]` struct to signal that it is the implementation of the `Hello` component.

`Init` initializes the application. `Get[Hello]` returns a client to the component with interface `Hello`, creating it if necessary. The call to `hello.Greet` looks like a regular method call. Any serialization and remote procedure calls are abstracted away from the developer.

4 Runtime

4.1 Overview

Underneath the programming model lies a runtime that is responsible for distributing and executing components. The runtime makes all *high-level decisions* on how to run components. For example, it decides which components to co-locate and replicate. The runtime is also responsible for *low-level details* like launching components onto physical resources and restarting components when they fail. Finally, the runtime is responsible for performing *atomic rollouts*, ensuring

```go
// Component interface.
type Hello interface {
    Greet(name string) string
}

// Component implementation.
type hello struct {
    Implements[Hello]
}
func (h *hello) Greet(name string) string {
    return fmt.Sprintf("Hello, %s!", name)
}

// Component invocation.
func main() {
    app := Init()
    hello := Get[Hello](app)
    fmt.Println(hello.Greet("World"))
}
```

Figure 2: A "Hello, World!" application.

that components in one version of an application never communicate with components in a different version.

There are many ways to implement a runtime. The goal of this paper is not to prescribe any particular implementation. Still, it is important to recognize that the runtime is not magical. In the rest of this section, we outline the key pieces of the runtime and demystify its inner workings.

4.2 Code Generation

The first responsibility of the runtime is code generation. By inspecting the `Implements[T]` embeddings in a program's source code, the code generator computes the set of all component interfaces and implementations. It then generates code to marshal and unmarshal arguments to component methods. It also generates code to execute these methods as remote procedure calls. The generated code is compiled along with the developer's code into a single binary.

4.3 Application-Runtime Interaction

With our proposal, applications do not include any code specific to the environment in which they are deployed, yet they must ultimately be run and integrated into a specific environment (e.g., across machines in an on-premises cluster or across regions in a public cloud). To support this integration, we introduce an API (partially outlined in Table 1) that isolates application logic from the details of the environment.

API	Description
`RegisterReplica`	Register a proclet as alive and ready.
`StartComponent`	Start a component, potentially in another process.
`ComponentsToHost`	Get components a proclet should host.

Table 1: Example API between the application and runtime.

The caller of the API is a **proclet**. Every application binary runs a small, environment-agnostic daemon called a proclet that is linked into the binary during compilation. A proclet manages the components in a running binary. It runs them, starts them, stops them, restarts them on failure, etc.

The implementer of the API is the runtime, which is responsible for all control plane operations. The runtime decides how and where proclets should run. For example, a multiprocess runtime may run every proclet in a subprocess; an SSH runtime may run proclets via SSH; and a cloud runtime may run proclets as Kubernetes pods [25, 28].

Concretely, proclets interact with the runtime over a Unix pipe. For example, when a proclet is constructed, it sends a `RegisterReplica` message over the pipe to mark itself as alive and ready. It periodically issues `ComponentsToHost` requests to learn which components it should run. If a component calls a method on a different component, the proclet issues a `StartComponent` request to ensure it is started.

The runtime implements these APIs in a way that makes sense for the deployment environment. We expect most runtime implementations to contain the following two pieces: (1) a set of envelope processes that communicate directly with proclets via UNIX pipes, and (2) a global manager that orchestrates the execution of the proclets (see Figure 3).

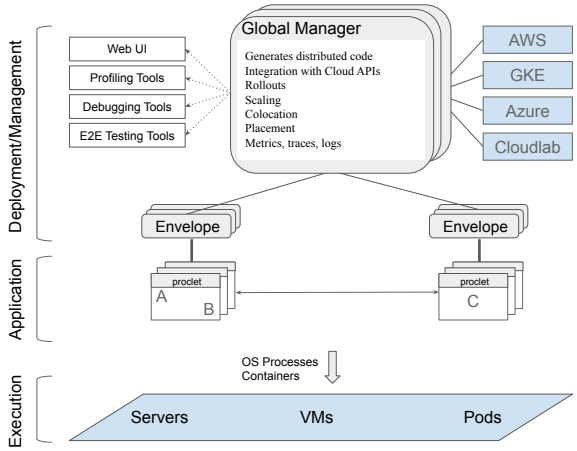

Figure 3: Proposed Deployer Architecture.

An envelope runs as the parent process to a proclet and relays API calls to the manager. The manager launches envelopes and (indirectly) proclets across the set of available resources (e.g., servers, VMs). Throughout the lifetime of the application, the manager interacts with the envelopes to collect health and load information of the running components; to aggregate metrics, logs, and traces exported by the components; and to handle requests to start new components. The manager also issues environment-specific APIs (e.g., Google Cloud [16], AWS [4]) to update traffic assignments and to scale up and down components based on load, health, and performance constraints. Note that the runtime implements the control plane but not the data plane. Proclets communicate directly with one another.

4.4 Atomic Rollouts

Developers inevitably have to release new versions of their application. A widely used approach is to perform rolling updates, where the machines in a deployment are updated from the old version to the new version one by one. During a rolling update, machines running different versions of the code have to communicate with each other, which can lead to failures. [78] shows that the majority of update failures are caused by these cross-version interactions.

To address these complexities, we propose a different approach. The runtime ensures that application versions are rolled out atomically, meaning that all component communication occurs within a single version of the application. The runtime gradually shifts traffic from the old version to the new version, but once a user request is forwarded to a specific version, it is processed entirely within that version. One popular implementation of atomic rollouts is the use of blue/green deployments [9].

5 Enabled Innovations

5.1 Transport, Placement, and Scaling

The runtime has a bird's-eye view into application execution, which enables new avenues to optimize performance. For example, our framework can construct a fine-grained call graph between components and use it to identify the critical path, the bottleneck components, the chatty components, etc. Using this information, the runtime can make smarter scaling, placement, and co-location decisions. Moreover, because serialization and transport are abstracted from the developer, the runtime is free to optimize them. For network bottlenecked applications, for example, the runtime may decide to compress messages on the wire. For certain deployments, the transport may leverage technologies like RDMA [32].

5.2 Routing

The performance of some components improves greatly when requests are routed with affinity. For example, consider an in-memory cache component backed by an underlying disk-based storage system. The cache hit rate and overall performance increase when requests for the same key are routed to the same cache replica. Slicer [44] showed that many applications can benefit from this type of affinity based routing and that the routing is most efficient when embedded in the application itself [43]. Our programming framework can

be naturally extended to include a routing API. The runtime could also learn which methods benefit the most from routing and route them automatically.

5.3 Automated Testing

One of the touted benefits of microservice architectures is fault-tolerance. The idea is that if one service in an application fails, the functionality of the application degrades but the app as a whole remains available. This is great in theory, but in practice it relies on the developer to ensure that their application is resilient to failures and, more importantly, to *test* that their failure-handling logic is correct. Testing is particularly challenging due to the overhead in building and running n different microservices, systematically failing and restoring them, and checking for correct behavior. As a result, only a fraction of microservice-based systems are tested for this type of fault tolerance. With our proposal, it is trivial to run end-to-end tests. Because applications are written as single binaries in a single programming language, end-to-end tests become simple unit tests. This opens the door to automated fault tolerance testing, akin to chaos testing [47], Jepsen testing [14], and model checking [62].

5.4 Stateful Rollouts

Our proposal ensures that components in one version of an application never communicate with components in a different version. This makes it easier for developers to reason about correctness. However, if an application updates state in a persistent storage system, like a database, then different versions of an application will indirectly influence each other via the data they read and write. These cross-version interactions are unavoidable—persistent state, by definition, persists across versions—but an open question remains about how to test these interactions and identify bugs early to avoid catastrophic failures during rollout.

5.5 Discussion

Note that innovation in the areas discussed in this section is not fundamentally unique to our proposal. There has been extensive research on transport protocols [63, 64], routing [44, 65], testing [45, 75], resource management [57, 67, 71], troubleshooting [54, 56], etc. However, the unique features of our programming model enable new innovations and make existing innovations much easier to implement.

For instance, by leveraging the atomic rollouts in our proposal, we can design highly-efficient serialization protocols that can safely assume that all participants are using the same schema. Additionally, our programming model makes it easy to embed routing logic directly into a user's application, providing a range of benefits [43]. Similarly, our proposal's ability to provide a bird's eye view of the application allows researchers to focus on developing new solutions for tuning applications and reducing deployment costs.

6 Prototype Implementation

Our prototype implementation is written in Go [38] and includes the component APIs described in Figure 2, the code generator described in Section 4.2, and the proclet architecture described in Section 4.3. The implementation uses a custom serialization format and a custom transport protocol built directly on top of TCP. The prototype also comes with a Google Kubernetes Engine (GKE) deployer, which implements multi-region deployments with gradual blue/green rollouts. It uses Horizontal Pod Autoscalers [20] to dynamically adjust the number of container replicas based on load and follows an architecture similar to that in Figure 3. Our implementation is available at github.com/ServiceWeaver.

6.1 Evaluation

To evaluate our prototype, we used a popular web application [41] representative of the kinds of microservice applications developers write. The application has eleven microservices and uses gRPC [18] and Kubernetes [25] to deploy on the cloud. The application is written in various programming languages, so for a fair comparison, we ported the application to be written fully in Go. We then ported the application to our prototype, with each microservice rewritten as a component. We used Locust [26], a workload generator, to load-test the application with and without our prototype.

The workload generator sends a steady rate of HTTP requests to the applications. Both application versions were configured to auto-scale the number of container replicas in response to load. We measured the number of CPU cores used by the application versions in a steady state, as well as their end-to-end latencies. Table 2 shows our results.

Metric	Our Prototype	Baseline
QPS	10000	10000
Average Number of Cores	28	78
Median Latency (ms)	2.66	5.47

Table 2: Performance Results.

Most of the performance benefits of our prototype come from its use of a custom serialization format designed for non-versioned data exchange, as well as its use of a streamlined transport protocol built directly on top of TCP. For example, the serialization format used does not require any encoding of field numbers or type information. This is because all encoders and decoders run at the exact same version and agree on the set of fields and the order in which they should be encoded and decoded in advance.

For an apples-to-apples comparison to the baseline, we did not co-locate any components. When we co-locate all

eleven components into a single OS process, the number of cores drops to 9 and the median latency drops to 0.38 ms, both an order of magnitude lower than the baseline. This mirrors industry experience [34, 39].

7 Related Work

Actor Systems. The closest solutions to our proposal are Orleans [74] and Akka [3]. These frameworks also use abstractions to decouple the application and runtime. Ray [70] is another actor based framework but is focused on ML applications. None of these systems support atomic rollouts, which is a necessary component to fully address challenges **C2-C5**. Other popular actor based frameworks such as Erlang [61], E [52], Thorn [48] and C++ Actor Framework [10] put the burden on the developer to deal with system and low level details regarding deployment and execution, hence they fail to decouple the concerns between the application and the runtime and therefore don't fully address **C1-C5**. Distributed object frameworks like CORBA, DCOM, and Java RMI use a programming model similar to ours but suffered from a number of technical and organizational issues [58] and don't fully address **C1-C5** either.

Microservice Based Systems. Kubernetes [25] is widely used for deploying container based applications in the cloud. However, its focus is orthogonal to our proposal and doesn't address any of **C1-C5**. Docker Compose [15], Acorn [1], Helm [19], Skaffold [35], and Istio [21] abstract away some microservice challenges (e.g., configuration generation). However, challenges related to splitting an application into microservices, versioned rollouts, and testing are still left to the user. Hence, they don't satisfy **C1-C5**.

Other Systems. There are many other solutions that make it easier for developers to write distributed applications, including dataflow systems [51, 59, 77], ML inference serving systems [8, 17, 42, 50, 73], serverless solutions [11, 24, 36], databases [29, 49], and web applications [66]. More recently, service meshes [46, 69] have raised networking abstractions to factor out common communication functionality. Our proposal embodies these same ideas but in a new domain of general serving systems and distributed applications. In this context, new challenges arise (e.g., atomic rollouts).

8 Discussion

8.1 Multiple Application Binaries

We argue that applications should be written and built as single binaries, but we acknowledge that this may not always be feasible. For example, the size of an application may exceed the capabilities of a single team, or different application services may require distinct release cycles for organizational reasons. In all such cases, it may be necessary for the application to consist of multiple binaries.

While this paper doesn't address the cases where the use of multiple binaries is required, we believe that our proposal allows developers to write fewer binaries (i.e. by grouping multiple services into single binaries whenever possible), achieve better performance, and postpone hard decisions related to how to partition the application. We are exploring how to accommodate applications written in multiple languages and compiled into separate binaries.

8.2 Integration with External Services

Applications often need to interact with external services (e.g., a Postgres database [29]). Our programming model allows applications to interact with these services as any application would. Not anything and everything has to be a component. However, when an external service is extensively used within and across applications, defining a corresponding component might provide better code reuse.

8.3 Distributed Systems Challenges

While our programming model allows developers to focus on their business logic and defer a lot of the complexity of deploying their applications to a runtime, our proposal does *not* solve fundamental challenges of distributed systems [53, 68, 76]. Application developers still need to be aware that components may fail or experience high latency.

8.4 Programming Guidance

There is no official guidance on how to write distributed applications, hence it's been a long and heated debate on whether writing applications as monoliths or microservices is a better choice. However, each approach comes with its own pros and cons. We argue that developers should write their application as a single binary using our proposal and decide later whether they really need to move to a microservices-based architecture. By postponing the decision of how exactly to split into different microservices, it allows them to write fewer and better microservices.

9 Conclusion

The status quo when writing distributed applications involves splitting applications into independently deployable services. This architecture has a number of benefits but also many shortcomings. In this paper, we propose a different programming paradigm that sidesteps these shortcomings. Our proposal encourages developers to (1) write monolithic applications divided into logical components, (2) defer to a runtime the challenge of physically distributing and executing the modularized monoliths, and (3) deploy applications atomically. These three guiding principles unlock a number of benefits and open the door to a bevy of future innovation. Our prototype implementation reduced application latency by up to 15× and reduced cost by up to 9× compared to the status quo.

References

[1] Acorn. https://www.acorn.io/.
[2] Actor model. https://en.wikipedia.org/wiki/Actor_model.
[3] Akka. https://akka.io.
[4] Amazon Web Services. https://aws.amazon.com/.
[5] Apache avro. https://avro.apache.org/docs/1.2.0/.
[6] Apache thrift. https://thrift.apache.org/.
[7] AWS Cloud Map. https://aws.amazon.com/cloud-map/.
[8] Azure Machine Learning. https://docs.microsoft.com/en-us/azure/machine-learning.
[9] Blue/green deployments. https://tinyurl.com/3bk64ch2.
[10] The c++ actor framework. https://www.actor-framework.org/.
[11] Cloudflare Workers. https://workers.cloudflare.com/.
[12] Continuous integration and delivery - circleci. https://circleci.com/.
[13] Dapr - distributed application runtime. https://dapr.io/.
[14] Distributed systems safety research. https://jespen.io.
[15] Docker compose. https://docs.docker.com/compose/.
[16] Google Cloud. https://cloud.google.com/.
[17] Google Cloud AI Platform. https://cloud.google.com/ai-platform.
[18] grpc. https://grpc.io/.
[19] Helm. http://helm.sh.
[20] Horizontal Pod Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
[21] Istio. https://istio.io/.
[22] Jenkins. https://www.jenkins.io/.
[23] Json. https://www.json.org/json-en.html.
[24] Kalix. https://www.kalix.io/.
[25] Kubernetes. https://kubernetes.io/.
[26] Locust. https://locust.io/.
[27] Micro | powering the future of cloud. https://micro.dev/.
[28] Pods. https://kubernetes.io/docs/concepts/workloads/pods/.
[29] Postgresql. https://www.postgresql.org/.
[30] Protocol buffers. https://developers.google.com/protocol-buffers.
[31] RabbitMQ. https://www.rabbitmq.com/.
[32] Remote direct memory access. https://en.wikipedia.org/wiki/Remote_direct_memory_access.
[33] REST API. https://restfulapi.net/.
[34] Scaling up the Prime Video audio/video monitoring service and reducing costs by 90%. https://tinyurl.com/yt6nxt63.
[35] Skaffold. https://skaffold.dev/.
[36] Temporal. https://temporal.io/.
[37] Terraform. https://www.terraform.io/.
[38] The Go programming language. https://go.dev/.
[39] To Microservices and Back Again - Why Segment Went Back to a Monolith. https://tinyurl.com/5932ce5n.
[40] WebSocket. https://en.wikipedia.org/wiki/WebSocket.
[41] Online boutique. https://github.com/GoogleCloudPlatform/microservices-demo, 2023.
[42] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.
[43] A. Adya, R. Grandl, D. Myers, and H. Qin. Fast key-value stores: An idea whose time has come and gone. In *HotOS*, 2019.
[44] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, R. Peon, L. Kai, A. Shraer, A. Merchant, and K. Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *OSDI*, 2016.
[45] D. Ardelean, A. Diwan, and C. Erdman. Performance analysis of cloud applications. In *NSDI*, 2018.
[46] S. Ashok, P. B. Godfrey, and R. Mittal. Leveraging service meshes as a new network layer. In *HotNets*, 2021.
[47] A. Basiri, N. Behnam, R. De Rooij, L. Hochstein, L. Kosewski, J. Reynolds, and C. Rosenthal. Chaos engineering. In *IEEE Software*, 2016.
[48] B. Bloom, J. Field, N. Nystrom, J. Östlund, G. Richards, R. Strniša, J. Vitek, and T. Wrigstad. Thorn: Robust, concurrent, extensible scripting on the jvm. In *OOPSLA*, 2009.
[49] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, 2012.
[50] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, 2017.
[51] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, 2004.
[52] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming heterogeneity - the ptolemy approach. In *Proceedings of the IEEE*, 2003.
[53] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. In *ACM Journal*, 1985.
[54] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou. Sage: Practical and Scalable ML-Driven Performance Debugging in Microservices. In *ASPLOS*, 2021.
[55] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*, 2019.
[56] Y. Gan, Y. Zhang, K. Hu, Y. He, M. Pancholi, D. Cheng, and C. Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *ASPLOS*, 2019.
[57] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.
[58] M. Henning. The rise and fall of corba: There's a lot we can learn from corba's mistakes. In *Queue*, 2006.
[59] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Eurosys*, 2007.
[60] K. Jay, N. Neha, and R. Jun. Kafka : a distributed messaging system for log processing. In *NetDB*, 2011.
[61] A. Joe. Erlang. In *Communications of the ACM*, 2010.
[62] L. Lamport. The temporal logic of actions. In *ACM TOPLS*, 1994.
[63] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, and Z. Shi. The quic transport protocol: Design and internet-scale deployment. In *SIGCOMM*, 2017.
[64] N. Lazarev, N. Adit, S. Xiang, Z. Zhang, and C. Delimitrou. Dagger: Towards Efficient RPCs in Cloud Microservices with Near-Memory Reconfigurable NICs. In *ASPLOS*, 2021.
[65] S. Lee, Z. Guo, O. Sunercan, J. Ying, T. Kooburat, S. Biswal, J. Chen, K. Huang, Y. Cheung, Y. Zhou, K. Veeraraghavan, B. Damani, P. M. Ruiz, V. Mehta, and C. Tang. Shard manager: A generic shard management framework for geo-distributed applications. In *SOSP*, 2021.
[66] B. Livshits and E. Kiciman. Doloto: Code splitting for network-bound web 2.0 applications. In *FSE*, 2008.
[67] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *SOCC*, 2021.

[68] N. A. Lynch. Distributed algorithms. In *Morgan Kaufmann Publishers Inc.*, 1996.
[69] S. McClure, S. Ratnasamy, D. Bansal, and J. Padhye. Rethinking networking abstractions for cloud tenants. In *HotOS*, 2021.
[70] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging ai applications. In *OSDI*, 2018.
[71] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *OSDI*, 2020.
[72] D. Raghavan, P. Levis, M. Zaharia, and I. Zhang. Breakfast of champions: towards zero-copy serialization with nic scatter-gather. In *HotOS*, 2021.
[73] F. Romero, Q. Li, N. J. Yadwadkar, and C. Kozyrakis. Infaas: Automated model-less inference serving. In *ATC*, 2021.
[74] B. Sergey, G. Allan, K. Gabriel, L. James, P. Ravi, and T. Jorgen. Orleans: Cloud computing for everyong. In *SOCC*, 2011.
[75] M. Waseem, P. Liang, G. Márquez, and A. D. Salle. Testing microservices architecture-based applications: A systematic mapping study. In *APSEC*, 2020.
[76] Wikipedia contributors. Fallacies of distributed computing.
[77] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[78] Y. Zhang, J. Yang, Z. Jin, U. Sethi, K. Rodrigues, S. Lu, and D. Yuan. Understanding and detecting software upgrade failures in distributed systems. In *SOSP*, 2021.