Toward Data-Centric Service Composition

Silvery D. Fu[1,2], Hong Zhang[3], Ryan Teoh[1,2], Taras Priadka[1,2], Sylvia Ratnasamy[2]
[1]Systems Design Studio, [2]UC Berkeley, [3]University of Waterloo

Abstract

Microservices are increasingly used in modern applications, leading to a growing need for effective service composition solutions. However, we argue that traditional *API-centric composition* mechanisms (e.g., RPC, REST, and Pub/Sub) hamper the modularity of microservices. These mechanisms introduce rigid code-level coupling, scatter composition logic, and hinder visibility into cross-service data exchanges. Ultimately, these limitations complicate the maintenance and evolution of microservice-based applications. In this paper, we propose a rethinking of service composition and present Knactor, a new *data-centric* composition framework to restore the modularity that microservices were intended to offer. Knactor decouples service composition from service development, allowing composition to be implemented as explicit *data exchanges* among multiple services. Our initial case study suggests that this approach not only simplifies service composition but also opens up opportunities for data-driven policies and optimizations.

CCS Concepts

• **Software and its engineering → Software architectures**.

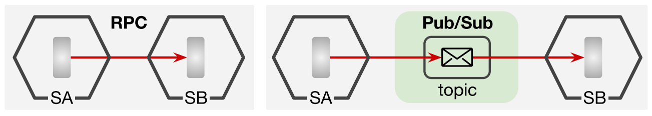
Keywords

Design Principles, Service Composition, Microservices

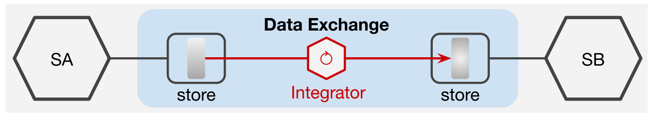
ACM Reference Format:
Silvery D. Fu, Hong Zhang, Ryan Teoh, Taras Priadka, Sylvia Ratnasamy. 2024. Toward Data-Centric Service Composition. In *The 23rd ACM Workshop on Hot Topics in Networks (HOTNETS '24), November 18–19, 2024, Irvine, CA, USA*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3696348.3702013

1 Introduction

The microservice architecture has been widely adopted in building modern applications [3, 8]. By breaking software systems down into smaller, independently deployable services, microservices make it easier to distribute development across teams and organizations such as SaaS providers and



(a) API-centric service composition with RPC and Pub/Sub.



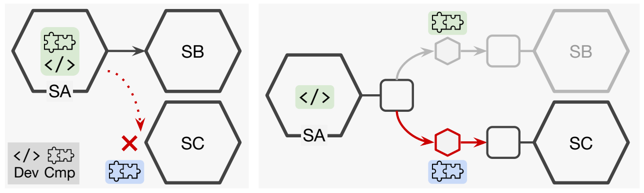
(b) Data-centric service composition with Knactor.

Figure 1: Comparison of service composition mechanisms. In RPC, Service S_A invokes the API defined by Service S_B. In Pub/Sub, S_A publishes messages to a topic where S_B subscribes and receives these messages. In Knactor, S_A and S_B externalize their states in data stores hosted on a data exchange. An integrator processes and syncs states between the data stores.

open-source communities. As more applications embrace this architecture, such as web and mobile back-ends, datacenter management, robot coordination, and IoT/smart space controllers [13, 15, 19, 24, 44, 46], the need for effective *service composition* becomes increasingly important. Service composition combines multiple microservices to create an end-to-end application (app), making it a crucial and challenging aspect in application development.

Today, service composition is typically achieved via remote procedure calls (RPCs) [21] or via publish-subscribe messaging (Pub/Sub) [10], as depicted in Fig.1a. With RPCs, service S_A invokes the API of another service S_B by incorporating the API endpoints and message schemas defined by S_B (e.g., gRPC and Protobuf [21]). Then, at run-time, S_A sends a request message via a synchronous call to S_B and, after running its procedure, S_B replies with a response message. Pub/Sub replaces synchronous communication with asynchronous delivery of messages. In Pub/Sub, S_B subscribes to a topic on a message broker (e.g., Kafka [10]). S_A can then send messages to this topic, which S_B receives asynchronously and decodes using a schema. In this case, the topic and the schema (predefined by either S_A or S_B) can be viewed as a variant of an API endpoint. We refer to these composition mechanisms as *API-centric composition* where services are composed using predefined APIs at the time of development.

However, the API-centric approach leads to three drawbacks for service composition. First, it introduces tight *coupling* between services. To compose services S_B to S_A, service developers incorporate S_B's schemas, client code stubs, invocation methods, response and error handling in the S_A's



(a) Cannot compose S_A and S_C w/o code changes in S_A. (b) With development and composition decoupled, one can compose S_A and S_C w/o modifying S_A.

Figure 2: Decoupling service development and composition. Existing composition mechanisms, shown in (a), couple the service development and composition. Knactor, shown in (b), allows these two to be decoupled. **Dev:** Development; **Cmp:** Composition.

code itself. Consequently, making composition changes, such as replacing S_B with S_C or adapting S_A to new schema of S_B, requires accessing and modifying the code of S_A as well as rebuilding and redeploying it. Second, in an application consisting of N services, the composition logic is scattered across $O(N)$ services. Each service may use $O(N)$ external APIs, while its own APIs are used by $O(N)$ other services, depending on the in/out-degree [51]. As a result, changes in the composition logic often involve and impact many services in the application, requiring extensive coordination and code-level changes across developers and teams responsible for each service. Third, the composition logic—especially the *data exchanges* among services—are *hidden* within the API invocations between individual pairs of services. This lack of visibility hinders customizing and optimizing composition at the app-level and at run-time.

The above drawbacks suggest that while microservices are modular, existing composition mechanisms hamper this modularity, complicating both the development and the composition of microservices. In particular, as modern applications increasingly consist of 10s, 100s, or even 1,000s of microservices (e.g., as seen in web apps [2, 5, 26]), implementing and evolving service composition that is coupled, scattered, and hidden among individual services can become painstaking and prone to errors (§2). Such complexity escalates when apps and services are developed by different companies or vendors (e.g., as seen in IoT apps [7, 33] and public API marketplaces [38]) due to the high communication and coordination costs for making service changes [1].

How can we simplify service composition and restore the modularity that microservices were designed to offer? In this paper, we propose two guiding principles for service composition: **(P1)** *Decouple service composition from service development*, enabling the flexibility to implement and consolidate service composition after the development phase; and **(P2)** *Make data exchanges explicit*, providing greater visibility and control to simplify service composition.

We present a new service composition mechanism that follows principles P1 and P2. Our key insight is that services should be composed *via states, not APIs*: we propose replacing API-centric composition with a new approach that we term *data-centric* composition. In this data-centric approach, there are no RPCs or messaging between services, i.e., no direct invocation of each others' APIs. Instead, each service externalizes its states to an associated *data store* (Fig.1b),[1] and an *integrator* module acts as the intermediary that composes services by processing and syncing states between their data stores. The integrator can be easily replaced or reconfigured, not only during development but also at run-time (P1). In the integrator, developers can use dedicated state processing primitives, such as data exchange graphs and dataflow operators (§3), to conveniently specify the desired data exchanges among services (P2).

We call this approach to developing and composing services as the "Knactor pattern."[2] The Knactor pattern aims to address the drawbacks of existing service composition mechanisms through its core design choices: **(i)** Knactor confines the interaction of each service to its own data store as opposed to introducing code-level coupling to other services. The composition logic is implemented in the integrator without modifying the services' code (Fig.2) and can be reconfigured even at run-time. **(ii)** Knactor consolidates the composition logic into a single or a few application-level integrator modules, as opposed to the $O(N)$ services and their codebases; thus making it easier to implement composition logic, reducing excessive coordination and communication across teams or companies. **(iii)** By having services externalize their states, developers gain the ability to implement composition in the form of data exchanges, with the help of dedicated state processing primitives; as opposed to dealing with intricate sequences of API invocations across different services. This approach also makes the composition code easier to change and maintain.

We further discuss the motivation for the Knactor pattern (§2) and propose our design for the *Knactor framework* to support this pattern in microservices (§3). We report on our early experiences with prototyping the Knactor framework and using it in example Web and IoT apps (§4). Finally, we discuss related work and outline future research directions on data-centric service composition (§5).

2 Revisiting Service Composition

While microservices, and more broadly the *service-oriented architecture* (SOA [37]), offer flexibility and scalability, they also create the need for effective composition and cooperation between services and teams. Today, composition is based on communication protocols such as RPC [11, 21], message

[1]Note that unlike in Pub/Sub, in our model, a service does not subscribe or publish to a different service but only to its *own* data store.
[2]Knactor (pronounced "connector") stands for Kubernetes-native actors in a nod to Kubernetes [24] and the actor model [39, 49] from which we drew inspiration (see §5).

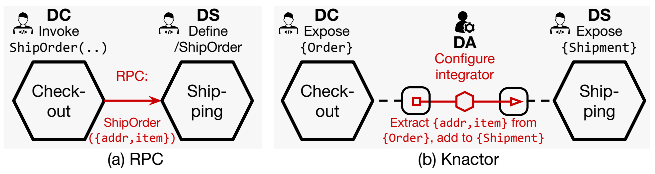


Figure 3: Comparison of composition mechanisms: RPC in (a) vs. Knactor in (b) for implementing an online retail app. **DX:** Developer for service or application X.

brokers [10, 16], and REST APIs [20, 36]. In what follows, we use examples from real-world apps to review the development workflow with existing composition mechanisms and motivate the case for a new approach. We'll use these apps in §3-§4 and show how they're handled with Knactor.

(1) Web: Online Retail. The first example is part of an online retail web application [19]. The original app uses gRPC to compose services [21]. Our focus is on two microservices, Checkout (C) and Shipping (S), and on the shipment request from C to S, which creates shipments for orders that have been checked out. With RPC, the developer of S (D_S) defines API endpoints with the API's name, version, and the message schema(s), etc. for the request messages in a Protobuf [30] definition file. As shown in Fig.3a, in this example, the API endpoint is `/ShipOrder` which takes item name and shipment address as its inputs. Then, D_S may share the API definition file with the developer of C (D_C), who uses the file to create the client stub code, imports and uses the code in C, and finally recompiles/builds C.

(2) IoT: Smart Home. The second example is from a home automation app [33, 44, 57] adapted from an open-source IoT app simulator [45]. The app includes a house service (developed by IoT company X, e.g., Samsung SmartThings [33]) that automatically adjusts the brightness level of lamps (from device vendor Y, e.g., Lifx [25]) based on occupancy sensor readings (from device vendor Z, e.g., Ring [9]) while monitoring the energy consumption of these devices. There are three services in this application—House (H), Motion (M), and Lamp (L)—that are composed via the message broker EMQX [16]. H subscribes to messages on M (the "motion" topic), and when H receives a message reporting "triggered: true", it publishes a message to L's topic to adjust the lamp's brightness level. For each service, the developer uses Protobuf to define schemas for the messages exchanged among devices. For example, H uses the schema of M and L to deserialize the messages from the two and vice versa.

Problem 1: Services are overly coupled. In both examples, we observe that API-centric composition introduces code-level coupling between services (e.g., C to S, H to L and M). Consequently, making any composition changes (e.g., switching C to a new shipping service) and ensuring compatibility with APIs (e.g., adapting C or H to the changes of schema of S or L) require modifications at the code level. This further leads to rebuilding and redeploying services, which also requires careful planning in the production environment to avoid application downtime [4, 6] and takes additional time and compute resources. For example, adapting C to an API schema change in S requires 69 lines of code and configuration updates (§4), followed by recompiling C, updating and uploading its container images, and redeploying C using a rolling update in Kubernetes [28]. Such changes are common in microservice-based applications [14, 27].

Problem 2: Composition logic is scattered. A direct consequence of implementing service composition within individual services is the scattering of composition logic across multiple services and service codebases. In the web app we studied, we identified 15 methods on handling API invocations scattered across 11 services, and 36 across 14 services in another well-studied social networking app [34, 47]. It is worth noting that these examples represent only small-scale apps developed for demonstration purposes. In production, applications may have composition logic scattered across more than 100s/1,000s of microservices [2, 5, 51].

Problem 3: Data exchanges are hidden. The examples also highlight that the data exchanges among services are hidden within pair-wise API invocations. In the online retail app, when an order checkout request is received by the C, it triggers an RPC request to the S. This request includes the order's states, which are not accessible outside this pair of services. Similarly, in the smart home system, state changes in the M or the L implicitly influence the state of the H through the messages exchanged among these services. This lack of visibility hinders the ability to add functionalities (e.g., implementing "conditional composition" where C should opt for air shipping when the order's price exceeds $1,000$ USD) and access control (e.g., H should not access the L during user-defined sleep hours).

3 Knactor Design

We present the rationale of the Knactor pattern (§3.1), the framework's designs (§3.2), and optimizations (§3.3).

3.1 The Rationale of the Knactor Pattern

Knactor's design is guided by two key principles aimed at enhancing modularity. The first is to **decouple service composition from service development**, which allows for the creation and updating of service composition outside the service development phase. This principle follows the classic design principle of *separating mechanism and policy*, i.e., the composition mechanism should not dictate which services *can* be composed. The rationale is that at the development stage, it can be challenging to anticipate all the ways the service might be used and extended in apps (i.e., the "composition policy"). Thus, an overly constraining composition mechanism can substantially increase the cost and delay involved in policy changes post-development.

Knactor achieves a cleaner separation of composition mechanism and policy by enabling *late-binding* of services via *two levels of indirection*: a per-service data store and integrator. In Knactor (Fig.1b), a service does not directly access other services' APIs nor their states but *its own data store only*. Then, an integrator module acts as an intermediary, responsible for processing and syncing states across the data stores of the services being composed.

The second guiding principle of Knactor is to replace API invocations with **explicit data exchanges** among services. The rationale is that developers, rather than calling and responding to APIs, can concisely specify data exchange patterns among multiple services and leverage the dedicated state processing primitives of a framework (§3.2) for simplified composition. This approach also streamlines the maintenance and evolution of composition logic, as updating the composition logic only requires altering the data exchange specification, eliminating the need to rewire API calls (§4). Finally, the declarative nature of data-centric composition separates the specification of composition from its execution, providing opportunities for optimization (§3.3).

3.2 Knactor Framework

The Knactor framework provides the programming libraries, tooling, and runtime to facilitate service composition with the data-centric approach. In Knactor, each microservice is represented as a *knactor*[3] that contains a *reconciler* component and one or multiple *data stores*.

Data store and exchange. A data store keeps the states relevant to the knactor's operation, such as order status in the Checkout service (Fig.5). The data stores are hosted on a logically centralized Data Exchange (DE) that provides state access and management capabilities such as data storage, caching, scaling, analytics, and access control. There can be different types of DEs that each specialized at handling a different type of states/data, e.g., API objects [22], logs [54], and database tables [29]. We envision the DEs will be taken off-the-shelf and the Knactor framework provides wrappers, tooling, and extensions around these DEs for simplifying and optimizing composition (§3.3). As a starting point, we focus on two types of DEs, "Object" and "Log". The former keeps states as attribute-value pairs in a k-v store and exposes APIs for CRUD operations over these states, while the latter keeps states as structured and semi-structured data as append-only logs and exposes data ingestion and analytics APIs. A knactor can have multiple data stores and thus use multiple DEs. For example, in Fig.4, the three knactors each have two data stores on Object and Log, with the ones on Object storing configuration states such as lamp's intensity level while the ones on Log storing sensor data such as motion readings.

[3]We overload the term "Knactor" to mean both the pattern and the framework, and the lowercase "knactor" the service abstraction.

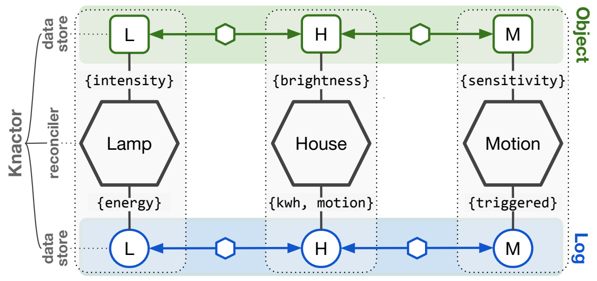


Figure 4: Building the smart home app in Knactor. There are three knactors, Lamp, House, and Motion each has two data stores, one on Object data exchange and one on Log data exchange.

```
schema: OnlineRetail/v1/Checkout/Order
items: object
address: string
cost: number
shippingCost: number # +kr: external
totalCost: number
currency: string
paymentID: string    # +kr: external
trackingID: string   # +kr: external
```

Figure 5: Schema of the Checkout knactor's data store.

```
Input:
  C: OnlineRetail/v1/Checkout/knactor-checkout
  S: OnlineRetail/v1/Shipping/knactor-shipping
  P: OnlineRetail/v1/Payment/knactor-payment
DXG:
  C.order:
    shippingCost: >
      currency_convert(S.quote.price,
                       S.quote.currency, this.currency)
    paymentID: P.id
    trackingID: S.id
  P:
    # other fields in the data store: id
    amount: C.order.totalCost
    currency: C.order.currency
  S:
    # other fields in the data store: id, quote
    items: '[item.name for item in C.order.items]'
    addr: C.order.address
    method: >
      "air" if C.order.cost > 1000 else "ground"
```

Figure 6: Specification of the data exchange graph (DXG) for the integrator in the online retail web app.

Reconciler. The reconciler is a code module that interacts with the knactor's data store(s) using the state access methods provided by the DE. It responds to state updates from the data store and initiates corresponding actions. For example, the reconciler inside the Shipping knactor may process a new shipment object that appears in the data store (e.g., by initiating a FedEx delivery [18]), and can also update the data store, such as posting the shipment's tracking ID. Service developers can adapt existing services by removing any external API invocations they aim to decouple from the service and adding required state access to the data store, or they can design and implement a new reconciler from scratch.

Integrator. An integrator syncs and processes states between data stores leveraging the APIs provided by the DEs. For instance, the integrator in Fig.3b can use the CRUD APIs provided by the Object DE to obtain the order states from the Checkout service, extract the shipment states `items` and `addr`, and update the Shipping knactor's data store. The framework provides *built-in integrators* specialized for processing states over a type of DE and data exchange patterns. Developers can use the state processing primitives from a built-in integrator to implement composition (described next). We focus on two built-in integrators, Cast and Sync, that handle states on Object and Log respectively.

Development workflow. The development workflow contains three logical steps: **(i) Externalize.** At development time, the developer registers the schema of the knactor's data store to the DE. **(ii) Express.** The developer indicates what states its data store can ingest by annotating the fields in the data store, e.g., in the Checkout knactor's data store (e.g., Fig.5) can indicate the "shippingCost", "paymentID", and "trackingID" are annotated to indicate they are to be filled externally by an integrator. Likewise, the House knactor can indicate it can ingest "kwh" and "motion" readings (Fig.4). **(iii) Exchange**. To compose services/knactors, the developer specifies the data exchanges among their data stores via programming or configuring the integrators. The state accesses and exchanges are subject to access control (§3.3).

Data exchange primitives. The integrator library provides primitives that simplify the expression of data exchange patterns for service composition. As shown in Fig.6, the Cast integrator supports data exchange graphs (DXGs), allowing the declarative description of data exchanges among multiple services. It can include references to states in each service's data store, state transformation and aggregation functions, and data-centric policies. Similarly, the Sync integrator offers dataflow operators like filter, rename, sort, and aggregation functions. For example, the House can retrieve motion sensor readings from the Motion service, and the Sync integrator can rename the "trigger" field to "motion" before loading the data into the House's data store.

3.3 Run-time Operation and Optimization

Integrator reconfiguration. Integrators, such as Cast and Sync, can be dynamically reconfigured at run-time to add new composition logic or modify existing configurations. This avoids service-level code changes, rebuilding, and redeployment for each composition update. For example, a new data-centric policy can easily be introduced to the DXG specification (Fig.6, line 22) to determine the shipment method based on the order price, without changing Checkout or Shipping.

State retention. By default, states in the data stores are preserved until they're no longer required by entities such as the knactor's reconciler or integrators. State retention can be managed via reference counting or similar mechanisms that track the usage of state objects. Once a reconciler or integrator has performed its operation on a state object, the object is marked as unused and the DEs can then perform garbage collection with standard recovery techniques. In addition, one can also specify customized state retention policies for archival or analytical purposes.

State access control. Knactor ensures only authorized entities can access the states in the data stores. First, developers can only view data store schemas, not actual states. At run-time, access to a knactor's data store is limited to its own reconciler and any integrators that have been granted access through access control policies. This can be done via the standard Role-based Access Control (RBAC), assigning roles to reconcilers and integrators to manage access [35]. Second, the data-centric approach allows finer-grained access control over states [58], e.g., granting access to certain state objects/fields but not others to specific roles.

Performance optimization. Knactor can optimize data exchange performance and efficiency by leveraging the modularity of the data store and integrator. First, one can use DEs optimized for high-performance such as in-memory k-v stores [32]. The integrators can perform *push-down optimization* to offload composition logic to the DE using features such as user-defined functions (UDFs) and stored procedures [31] common in these DEs. This can accelerate and reduce data movement between the DE and integrator. Second, when data stores are hosted on the DE, the DE and integrator can implement *zero-copy* data exchange to further minimize the data movement. Third, integrators can consolidate the state processing logic by combining multiple state processing operations into fewer and more efficient ones.

4 Prototype and Preliminary Studies

We built a prototype of the Knactor framework. It contains a programming library in Python for knactor development including communication packages for DEs, code generators, and a CLI for operating knactors. For the two DEs, we used open-source Kubernetes apiserver [22] (and Redis [32] as an alternative) for Object and Zed lake [54]) for Log; we implemented the built-in integrators, Cast and Sync, for Object and Log respectively. Using the Knactor prototype, we implemented the two example apps, online retail and smart home mentioned earlier. The online retail app consists of 11 knactors including the Checkout and Shipment discussed earlier, and a Cast-based integrator that composes the knactors.

We studied how well Knactor simplifies service composition and its implications on performance using the prototype and apps. We report the preliminary results in what follows.

App	Task	Operation		# File		SLOC	
	-	API	**KN**	API	**KN**	API	**KN**
Online Retail	1	c / f / b / d	**f**	8	**1**	109	**7**
	2	c / f / b / d	**f**	2	**1**	14	**1**
	3	c / f / b / d	**f**	4	**1**	93	**7**

Table 1: Comparison of composition cost: API-centric (API) vs. Knactor (KN). Annotations indicate required operations, **c:** code changes; **f:** config. changes. **b:** rebuild service; **d:** redeploy service.

Composition cost. We compare the service composition effort required for the Knactor and the API-centric approach using various tasks in the online retail app. The tasks include T_1: composing the Payment and Shipping services with the Checkout service; T_2: adding a shipment policy based on the order price; and T_3: updating the Shipping schema. We compare the required operations (e.g., code modifications, configuration updates, build and deployment steps), number of files, and the source lines of code (SLOC) changed or used to implement the task, including the services' source code, scripts, configurations, and schema definitions.

Takeaways. Table 1 presents the results. As shown, (i) In all tasks T_1-T_3, composition changes using Knactor require only reconfiguring the integrator module, while the API-centric approach requires code and configuration changes as well as service rebuilds and redeployments. This demonstrates Knactor's benefit of decoupling composition from development through externalized states and integrator, which facilitates composition without code-level changes at the individual services. (ii) Knactor enables the consolidation of composition logic, allowing modifications to be made in a single location (e.g., the DXG configuration) instead of multiple files across separate services/service codebases, which further simplifies the composition task. (iii) While the core composition logic remains the same in both approaches, compared to the API-centric approach, Knactor simplifies the *implementation* of composition logic, reducing the required SLOC (e.g., by 102 in T_1). We conjecture this is due to the *imperative* nature of the API-centric approach towards composition, which incurs not only the overhead of handling the "mechanics" (e.g., importing and handling schemas and client stubs), but also the complexity of expressing interactions across services (e.g., Checkout, Shipping, and Payment) as a sequence of API invocations. In contrast, Knactor enables the composition task to be *declaratively* and *concisely* expressed as data exchanges over externalized states (e.g., DXG in Fig.6), capturing operation ordering as state dependencies are resolved.

Impact on application performance. Compared to RPC, Knactor introduces two indirections for modularity: the data store and the integrator. To understand their impact on application performance, we benchmark the Cast between the Checkout and Shipping knactors deployed on a Kubernetes cluster. We measure the state propagation latency, with and without optimizations (§3.3), and repeat for the API-centric baselines and compare the results.

Setup	C-I	I	I-S	S	Prop. (ms)	Total (ms)
RPC	-	-	-	446	**1.8**	**447.8**
K-apiserver	20.6	0.01	12.5	453	33.1	486.1
K-redis	3.2	0.06	2.7	444	5.8	449.8
K-redis-udf	2.1	0.7	0.1	450	**2.9**	**452.9**

Table 2: Latency in the online retail app completing a shipment request, with breakdown by stage. **C-I**: Checkout and integrator. **I**: Integrator. **I-S**: Integrator and Shipping. **S:** Shipment processing.

Takeaways. Table 2 provides a breakdown of latencies in the online retail application. We compare three Knactor configurations: K-apiserver (using a strongly consistent k-v store with persistent storage [12] for Object exchange), K-redis (employing Redis, an in-memory data store [32]), and K-redis-udf (incorporating an integrator pushdown optimization using Redis's UDF [31]). First, the choice of DE substantially impacts the state propagation latency (33.1*ms* in K-apiserver vs. 5.8*ms* in K-redis); a high-performance DE can significantly reduce the data movement overhead from the integrator and reconciler. This overhead can be further reduced via integrator pushdown (e.g., from 2.7*ms* to 0.1*ms* between the integrator and Shipping's data store, with K-redis-udf). Second, the overhead has a relatively small impact on the performance of this app we studied, with the Shipment processing [18] as the primary bottleneck. However, note that Knactor does lead to higher latency overhead for state propagation; thus, for highly latency-sensitive workloads [53], direct RPC may still be the preferred approach.

5 Discussion and Future Work

In this section, we expand on related work beyond what was mentioned in §2, followed by future directions in Knactor.

Kubernetes and actor model. Knactor's design is influenced by Kubernetes [41], a cluster management system that manages applications via controllers [23]. The controllers read from and write to shared API/resource objects on an API server [22] as a means to interact. Compared to Knactor, Kubernetes does not contain the integrator abstraction, separation of data stores, and specialized data exchange designs. Knactor is also inspired by the actor model [39, 49], where actors are separate units of computation that interact via message passing. Knactor can be seen as consisting of two types of actors, knactors that express the main service/business logic and integrators that implement the composition logic and data exchanges between knactors.

Apps and applicability. Knactor is particularly beneficial for applications with many microservices and complex compositions, such as cellular EPC [48] besides the ones discussed in this paper. Currently, these systems rely on many inter-service APIs which leads to significant complexity [52] and is difficult to evolve. Knactor offers the benefits of enabling service composition to be performed by individuals who are not the original service developers. Similar to how API definitions and documentation convey information about the behaviors and semantics of today's services, Knactor developers can use the data store schema and other documentation to obtain required composition-related information. If these resources do not sufficiently describe the services' behaviors for effective composition, one can still fall back on engaging the original service developers to implement the composition task.

Framework support for composition. The visibility over states and data exchanges in Knactor allows developers to leverage tools such as formal methods and static analysis [55] as well as run-time primitives such as transactions [43] for implementing composition at large-scale. For example, the Cast can provide loop and unused state detection with static analysis to assist developers build robust data exchanges.

Compatibility and deployability. We expect the use of Knactor with existing systems can be facilitated through the use of proxies or porting mechanisms [17, 42]. Deployment issues such as load balancing, autoscaling, and observability, such as monitoring knactor SLOs through distributed tracing [40, 56] and telemetry [50], are also worth exploring.

Ecosystem. Knactor could potentially have far-reaching implications for the service and application ecosystem. For example, a marketplace for knactors and integrators could emerge, akin to current API marketplaces [38]. In such a marketplace, knactors and integrators, developed by various individuals or organizations, could be shared and reused, fostering more collaborative and streamlined app development.

Acknowledgments

The authors would like to thank the anonymous reviewers for their valuable feedback, which provided useful insights for our ongoing work. We also extend our gratitude to Wen Zhang and Ethan Jackson for their valuable discussions and support throughout the project. The lead author is especially grateful to Eric Brewer for the discussion they had on Kubernetes, which inspired the ideation of Knactor.

References

[1] 2014. Microservices: a definition of this new architectural term. https://martinfowler.com/articles/microservices.html.
[2] 2019. How Uber monitors 4,000 Microservices. https://www.cncf.io/blog/2019/02/05/how-uber-monitors-4000-microservices/.
[3] 2020. Microservices Architecture Market Statistics - 2026. https://www.alliedmarketresearch.com/microservices-architecture-market.
[4] 2021. 4 Microservice Deployment Patterns That Improve Availability. https://www.opslevel.com/resources/4-microservice-deployment-patterns-that-improve-availability.
[5] 2021. How Airbnb and Twitter Cut Back on Microservice Complexities. https://thenewstack.io/how-airbnb-and-twitter-cut-back-on-microservice-complexities.
[6] 2021. Zero Downtime Deployment Techniques: Rolling Update. https://www.encora.com/insights/zero-downtime-deployment-techniques-rolling-update.
[7] 2022. IoT Platform Companies Landscape 2021/2022. https://iot-analytics.com/iot-platform-companies-landscape/.
[8] 2023. Do you utilize microservices within your organization? https://www.statista.com/statistics/1236823/microservices-usage-per-organization-size/.
[9] 2024. Alarm Motion Detector. https://shop.ring.com/products/alarm-motion-detector-v2.
[10] 2024. Apache Kafka. https://kafka.apache.org/.
[11] 2024. Apache Thrift. https://thrift.apache.org/.
[12] 2024. APIs for building portable and reliable microservices. https://dapr.io/.
[13] 2024. Azure Kubernetes Service (AKS) Fabrikam Drone Delivery. https://github.com/mspnp/aks-fabrikam-dronedelivery.
[14] 2024. Creating, evolving, and versioning microservice APIs and contracts. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/maintain-microservice-apis.
[15] 2024. Custom Resources. https://spring.io/.
[16] 2024. EMQX MQTT Broker. https://www.emqx.io/.
[17] 2024. Envoy Proxy. https://www.envoyproxy.io/.
[18] 2024. FedEx Shipping API. https://www.shipengine.com/welcome-fedex-api/.
[19] 2024. GoogleCloudPlatform/microservices-demo: A web-based e-commerce app consists of an 11-tier microservices application. https://github.com/GoogleCloudPlatform/microservices-demo.
[20] 2024. GraphQL: A query language for your API. https://graphql.org/.
[21] 2024. gRPC: A high performance, open source universal RPC framework. https://grpc.io/.
[22] 2024. Kubernetes Apiserver. https://github.com/kubernetes/apiserver.
[23] 2024. Kubernetes Controllers. https://kubernetes.io/docs/concepts/architecture/controller/.
[24] 2024. Kubernetes: Production-Grade Container Orchestration. https://kubernetes.io/.
[25] 2024. LIFX Smart Home Light. https://www.lifx.com/.
[26] 2024. Netflix Architecture: How Much Does Netflix's AWS Cost? https://www.cloudzero.com/blog/netflix-aws.
[27] 2024. Panel: the Correct Number of Microservices for a System Is 489. https://www.infoq.com/presentations/number-microservices-system/.
[28] 2024. Performing a Rolling Update. https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/.
[29] 2024. PostgreSQL: The World's Most Advanced Open Source Relational Database. https://www.postgresql.org/.
[30] 2024. Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. https://developers.google.com/protocol-buffers.
[31] 2024. Redis functions. https://redis.io/docs/manual/programmability/functions-intro/.
[32] 2024. Redis: the open source, in-memory data store. https://redis.io/.
[33] 2024. SmartThings. https://smartthings.developer.samsung.com/.
[34] 2024. Social Network Microservices. https://github.com/delimitrou/DeathStarBench/tree/master/socialNetwork.
[35] 2024. Using RBAC Authorization. https://kubernetes.io/docs/reference/access-authn-authz/rbac/.
[36] 2024. What is a REST API? https://www.ibm.com/topics/rest-apis.
[37] 2024. What Is Service-Oriented Architecture? https://aws.amazon.com/what-is/service-oriented-architecture/.
[38] 2024. The World's Largest API Hub. https://rapidapi.com/.
[39] Gul Agha. 1986. *Actors: a model of concurrent computation in distributed systems*. MIT press.
[40] Jessica Berg, Fabian Ruffy, Khanh Nguyen, Nicholas Yang, Taegyun Kim, Anirudh Sivaraman, Ravi Netravali, and Srinivas Narayana. 2021.

Snicket: Query-driven distributed tracing. In *Proc. ACM HotNets*.

[41] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. 2016. Borg, Omega, and Kubernetes. *Commun. ACM* (2016).

[42] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In *Proc. USENIX NSDI*.

[43] Audrey Cheng, Xiao Shi, Lu Pan, Anthony Simpson, Neil Wheaton, Shilpa Lawande, Nathan Bronson, Peter Bailis, Natacha Crooks, and Ion Stoica. 2021. RAMP-TAO: layering atomic transactions on Facebook's online TAO data store. *Proc. VLDB Endowment* 14, 12 (2021), 3014–3027.

[44] Silvery Fu and Sylvia Ratnasamy. 2021. dSpace: Composable Abstractions for Smart Spaces. In *Proc. ACM SOSP*.

[45] Silvery Fu, Hong Zhang, Sylvia Ratnasamy, and Ion Stoica. 2022. The Internet of Things in a Laptop: Rapid Prototyping for IoT Applications with Digibox. In *Proc. ACM HotNets*.

[46] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proc. ACM ASPLOS*.

[47] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proc. ACM ASPLOS*.

[48] Shaddi Hasan et al. 2023. Building Flexible,{Low-Cost} Wireless Access Networks With Magma. In *Proc. USENIX NSDI*.

[49] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proc. IJCAI*.

[50] Greg Leffler. 2022. {OpenTelemetry} and Observability: What, Why, and Why Now? (2022).

[51] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proc. ACM SoCC*.

[52] Zhihong Luo, Silvery Fu, Mark Theis, Shaddi Hasan, Sylvia Ratnasamy, and Scott Shenker. 2021. Democratizing cellular access with CellBricks. In *Proc. ACM SIGCOMM*.

[53] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads.. In *Proc. USENIX NSDI*.

[54] Amy Ousterhout, Steve McCanne, Henri Dubois-Ferriere, Silvery Fu, Sylvia Ratnasamy, and Noah Treuhaft. 2021. Zed: leveraging data types to process eclectic data. In *Proc. CIDR*.

[55] Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2017. Verification in the age of microservices. In *Proc. ACM HotOS*.

[56] Benjamin H Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. Dapper, a large-scale distributed systems tracing infrastructure. (2010).

[57] Blase Ur, Elyse McManus, Melwyn Pak Yong Ho, and Michael L Littman. 2014. Practical trigger-action programming in the smart home. In *Proc. SIGCHI Conference on Human Factors in Computing Systems*. 803–812.

[58] Wen Zhang, Eric Sheng, Michael Chang, Aurojit Panda, Mooly Sagiv, and Scott Shenker. 2022. Blockaid: Data Access Policy Enforcement for Web Applications. In *Proc. USENIX OSDI*.