Toward Data-Centric Service Composition

Silvery D. Fu, Hong Zhang, Ryan Teoh, Taras Priadka, Sylvia Ratnasamy
Systems Design Studio LLC, UC Berkeley, University of Waterloo

Today we compose services via APIs

- A service is made of its app logic and APIs
- To compose two services:
  - Expose the API at the callee service
  - Invoke the API at the caller service
- Examples: RPC, REST, Pub/Sub

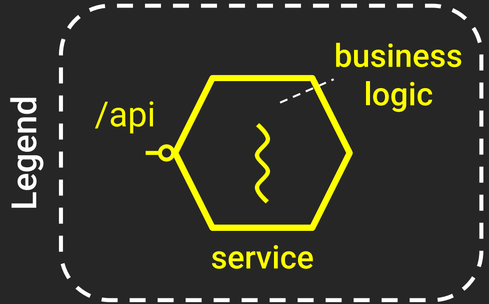
Today we compose services via APIs

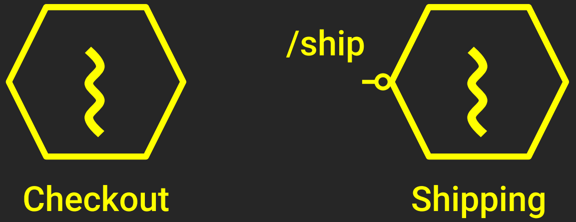
- Consider an online retail application:
  - Checkout, Shipping, Payment, .. services
  - Shipping exposes a /ship API
- Checkout requests /ship with order info
- Shipping responds with confirmation

Observation

API-centric composition makes services difficult to maintain and evolve.

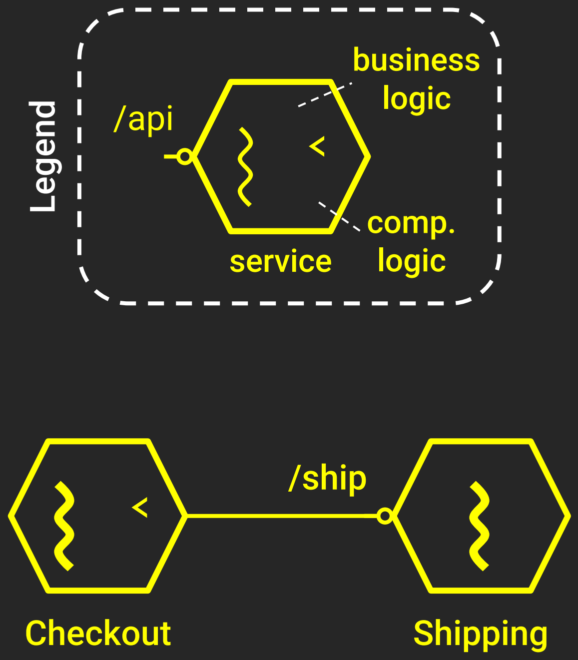
API-Centric Composition





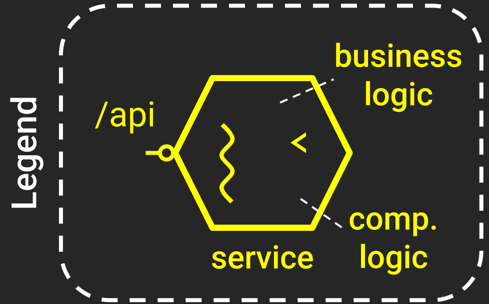
API-Centric Composition

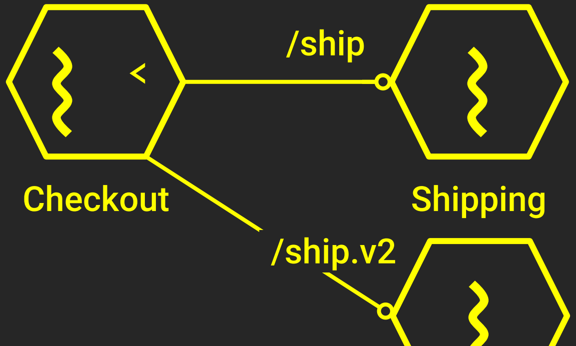
- Developers must embed message schemas, code stubs, and routines for requests, responses, and error handling directly in the service code.



API-Centric Composition

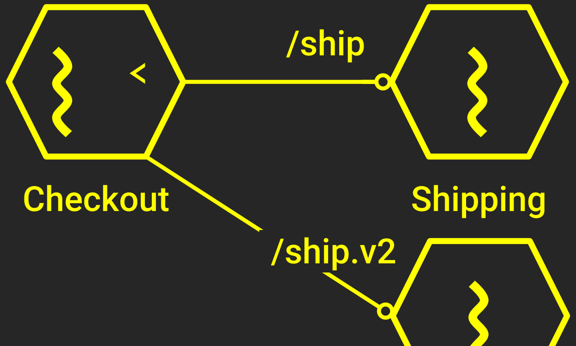
- **Problem 1:** service development and composition are coupled.
  - Composition changes must be made in the service.
  - Service rebuild and redeployment → interruptions and slow TTM.





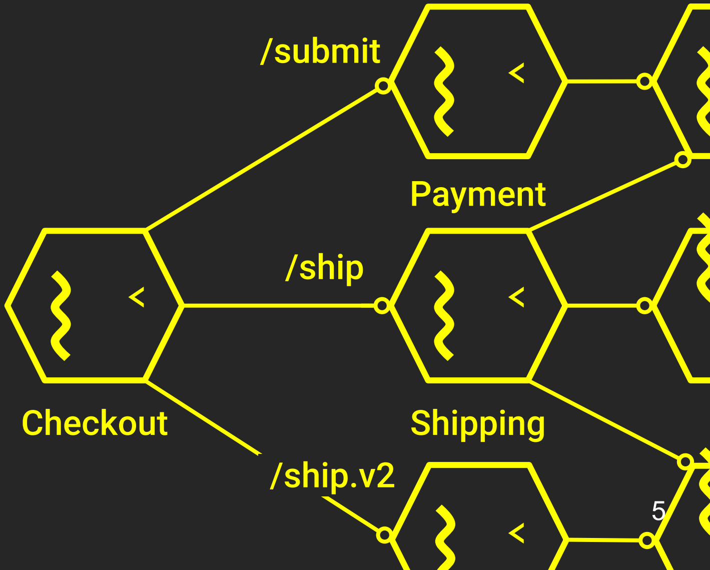
API-Centric Composition

- **Problem 2:** composition logic is scattered.



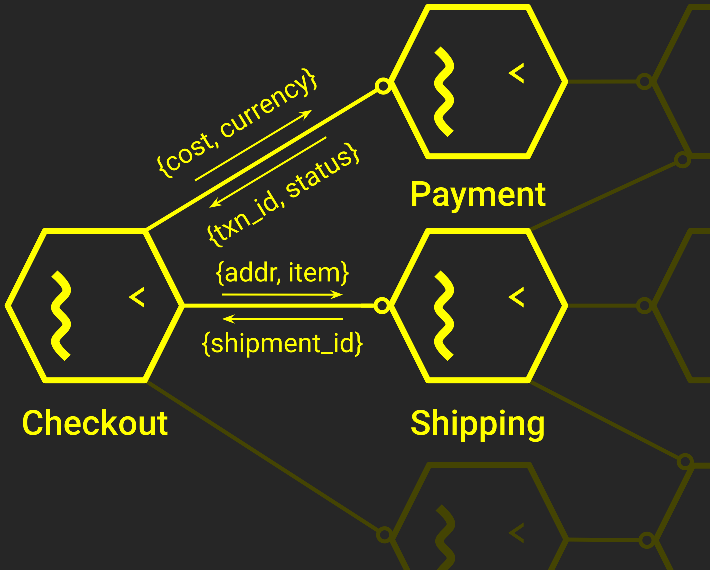
API-Centric Composition

- **Problem 2:** composition logic is scattered.
  - Composition logic spreads across multiple services; changes involve extensive team coordination.
  - Modern applications, such as Netflix and Uber, may contain 100s/1,000s services.



API-Centric Composition

- **Problem 3:** data exchanges are hidden.



API-Centric Composition

- **Problem 3:** data exchanges are hidden.
  - Data exchanges are hidden within API invocations between service pairs.
  - Lack of visibility hinders runtime monitoring, reconfiguration, and optimization.

API-Centric Composition

Hard to maintain and evolve service composition:

- Development and composition are coupled.
- Composition logic is scattered.
- Data exchanges are hidden.

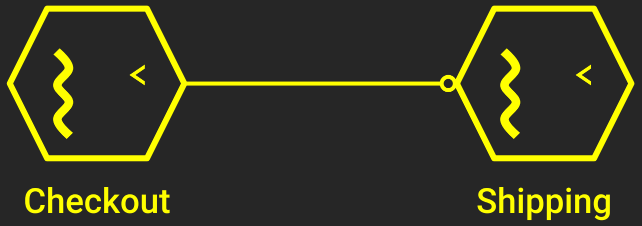
Rethinking Service Composition

Data-centric composition with two key principles:

- Principle 1: Decouple service composition from service development.
- Principle 2: Make data exchanges explicit.

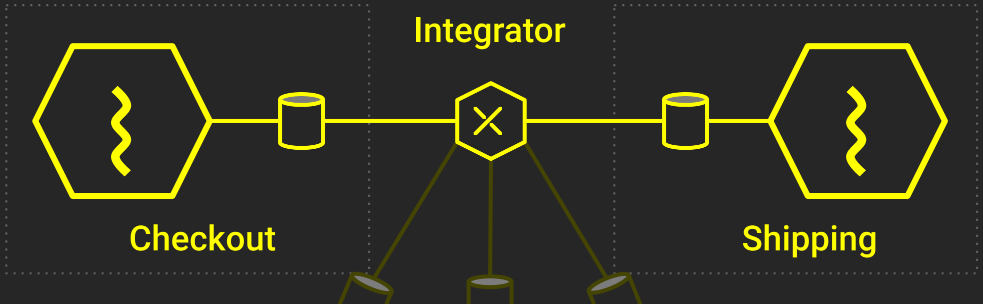
Data-Centric Composition

- Each **service** stores its composition-related states in a **data store** and reacts to updates.
- An **integrator** synchronizes states across data stores based on given data exchange graphs (DXGs).



Data-Centric Composition

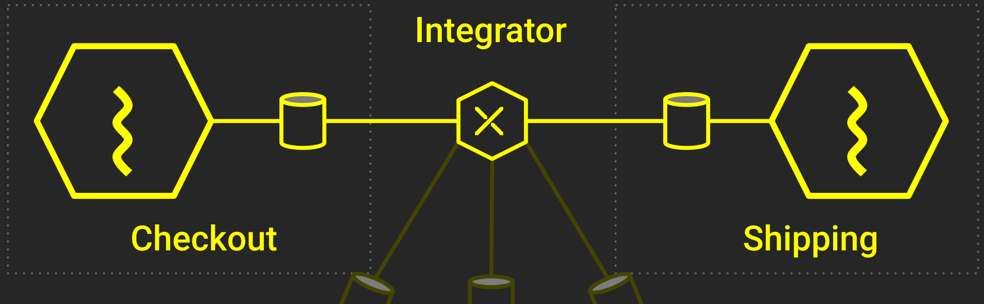
- Each **service** stores its composition-related states in a **data store** and reacts to updates.
- An **integrator** synchronizes states across data stores based on given data exchange graphs (DXGs).



Data-Centric Composition

Kubernetes-native actor

- We refer to this as the Knactor pattern:
  - Decoupled: services interact only with their own data store.
  - Consolidated: composition logic resides in the integrator.
  - Visible: data exchanges are explicit at the integrator.



Example: Online Retail Web App

- A web-based e-commerce app where users browse items, add to cart, and make purchases.
  https://github.com/GoogleCloudPlatform/microservices-demo/
- Contains 11 microservices, including Checkout, Shipping, and Payment composed with APIs (gRPC).
- Reproduce this application using Knactor.

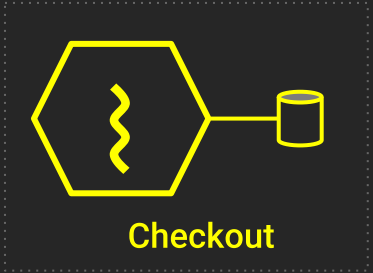
Knactor: Schema and Business Logic

Business logic (Python)

```python
@kr.on.update("OnlineRetail", "checkouts", "order")
def order_cost(states, name, **_):
    shipping_cost = kr.get(states, "shippingCost") or
         {
        "currencyCode": "USD",
        "units": 0,
        "nanos": 0,
    }
    cart_items = kr.get(states, "items", [])
    for item in cart_items:
        item_cost = money.multiply_slow(item["price"]
            , item["quantity"])
        cart_cost = money.sum(cart_cost, item_cost)

    total_cost = money.sum(cart_cost, shipping_cost)

    new_spec = {
        "states": {
            "totalCost": total_cost,
            "currency": "USD",
        }
    }

    kr.patch("OnlineRetail", "checkouts", n=name,
        spec=new_spec)
```

Data store schema (YAML)

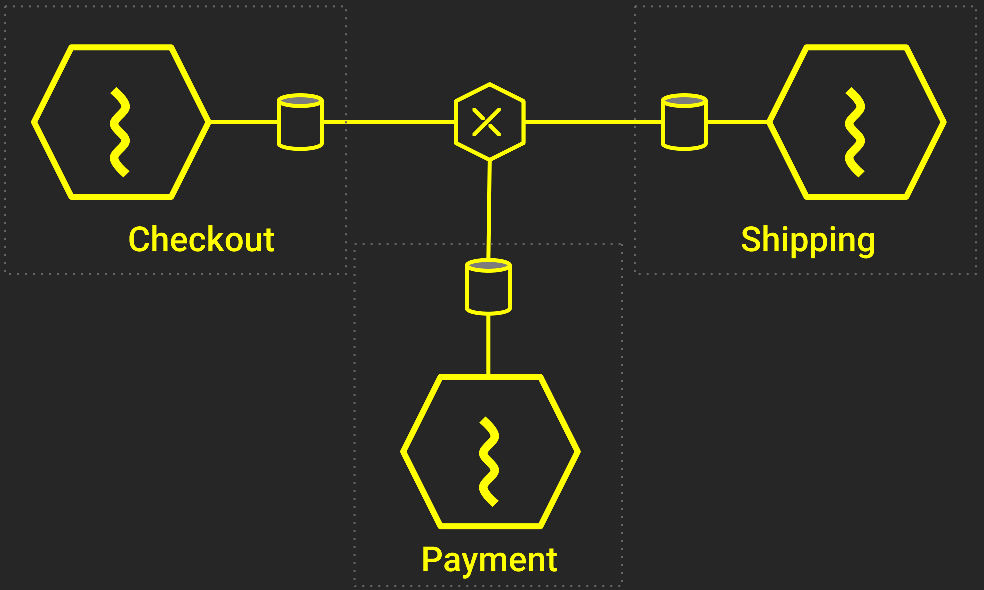
```yaml
schema: OnlineRetail/checkout/order
items: object
address: string
cost: number
shippingCost: number # +kr: external
totalCost: number
currency: string
paymentID: string     # +kr: external
trackingID: string    # +kr: external
```



Knactor: Data Exchange

Integrator (YAML)

```yaml
Input:
  C: OnlineRetail/checkout
  S: OnlineRetail/shipping
  P: OnlineRetail/payment

DXG:
  C.order:
    shippingCost: >
      currency_convert(
          S.quote.price,
          S.quote.currency,
          this.currency)
    paymentID: P.id
    trackingID: S.id
  P:
    amount: C.order.totalCost
    currency: C.order.currency
```




```yaml
S:
  items: '[item.name for item in C.order.items]'
  addr: C.order.address
  method: >
    "air" if C.order.cost > 1000 else "ground"
```

Comparing API vs. Knactor

Three implementation tasks:

1. Compose new Payment and Shipping services with the Checkout service.
2. Add a shipment policy based on the order price.
3. Update the Shipping schema.

Online Retail: API vs. Knactor

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

- **Operation:** APIs require code changes (c), configuration updates (f), rebuilds (b), and redeployments (d), whereas Knactor (due to decoupling) requires only integrator configuration updates.

Online Retail: API vs. Knactor

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

- **Number of files changed:** Knactor consolidates composition logic, allowing modifications in a single location (integrator DXG configuration file) instead of across multiple files in separate service codebases as with APIs.

Online Retail: API vs. Knactor

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

- **SLOC for Composition Logic:** Knactor simplifies composition through declarative data exchanges. Unlike APIs, which require handling schemas, stubs, and complex API sequences, Knactor captures operations more concisely in DXGs.

Takeaways

- API-centric composition couples development and composition, scatters composition logic, and hides data exchanges.
- To simplify maintenance and evolution, services should be composed over data, not APIs.

Check Out the Paper:

- Framework support for DXG programming.
- Performance implications and optimizations.
- State management and access control.

Reach Out: team@sd.studio

Backup

State Retention and Access Control

- Garbage collect states when no longer in use, and support custom policies for archival and analytics.
- Enforce access control with RBAC - only the reconciler and authorized integrators can access states.
  - Permissions are fine-grained that limit integrator access to specific state objects or fields.

Performance Implications

- Use high-performance data stores, such as in-memory key-value stores, to improve speed and efficiency.
- Offload composition logic to data stores with push-down optimizations like UDFs and stored procedures to reduce data movement.
- Minimize overhead with zero-copy data exchange and consolidate state processing into fewer operations.

Performance: API vs. Knactor

Setup	C-I	I	I-S	S	SP	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

- 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.