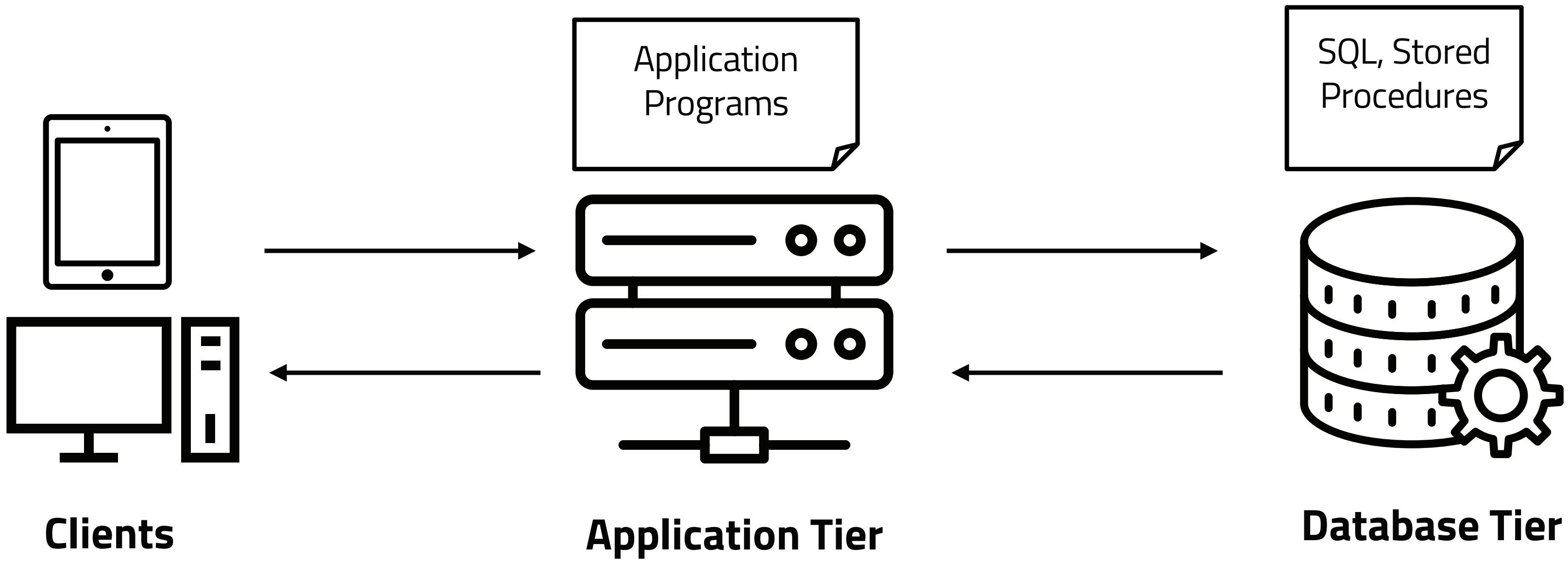
Programming Models

UNIVERSITY OF COPENHAGEN

TU Delft

Microservices

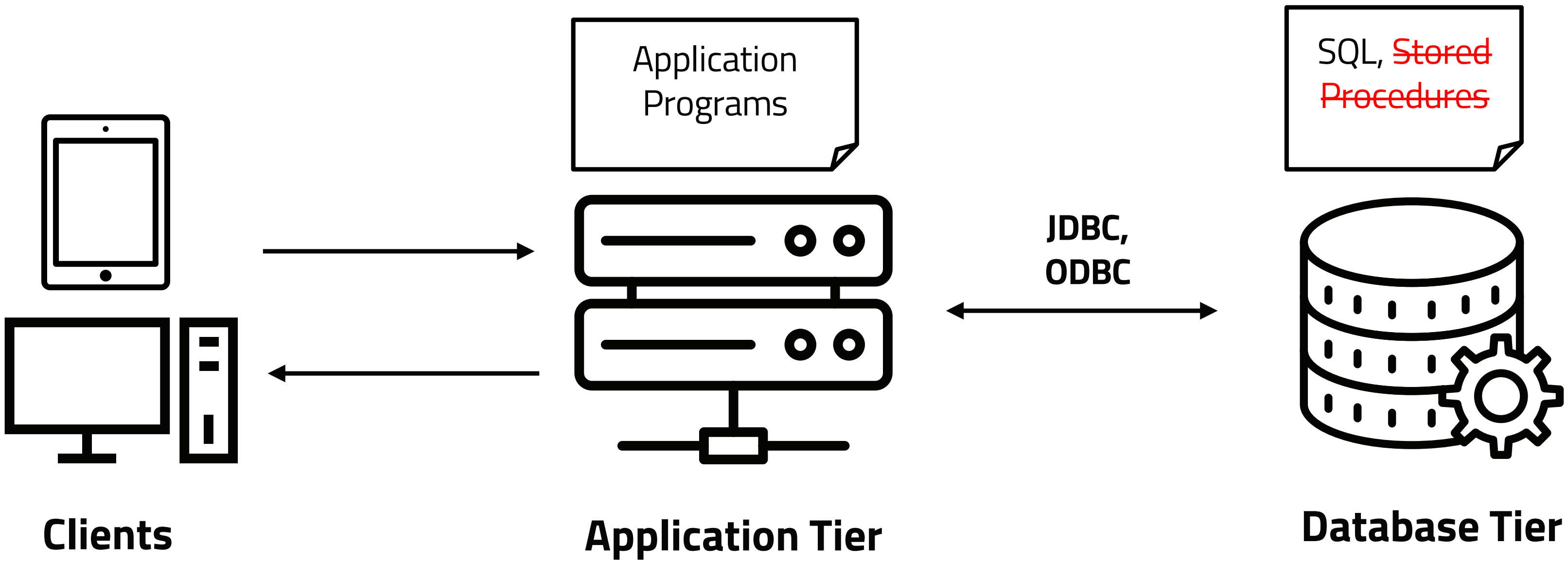
Service Architectures Background



- ✓ Durability
- ✓ Recoverability
- ✓ Data processing
- ✓ Data integrity

Figures partially extracted from Shah & Salles (2018)

Service Architectures Background

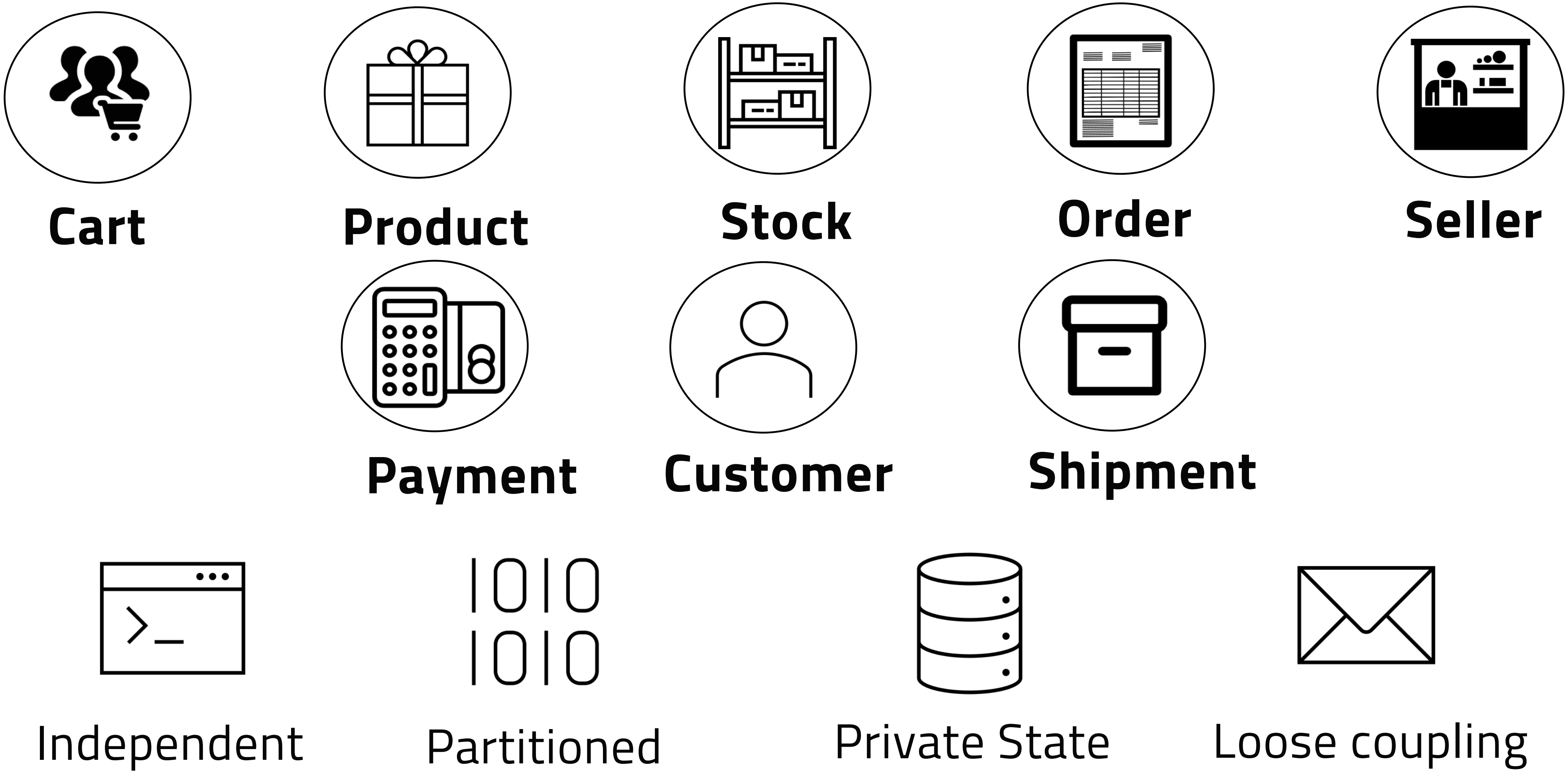


- ✓ Object-oriented programming
- ✓ Database technology abstraction
- ✓ Explicit connection management
- ✓ Exploit cloud-scale architectures

- ✓ Change tracking
- ✓ Code evolution
- ✓ Software maintenance
- ✓ Debuggers

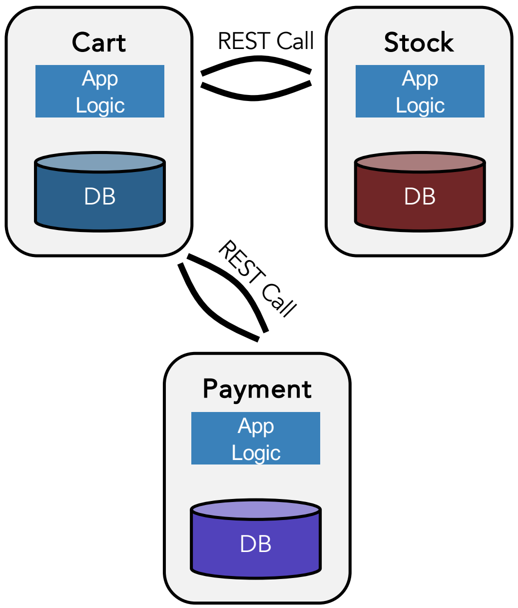
Figures partially extracted from Shah & Salles (2018)

Service Architectures Primer: Microservice



Service Architectures Primer: Microservice

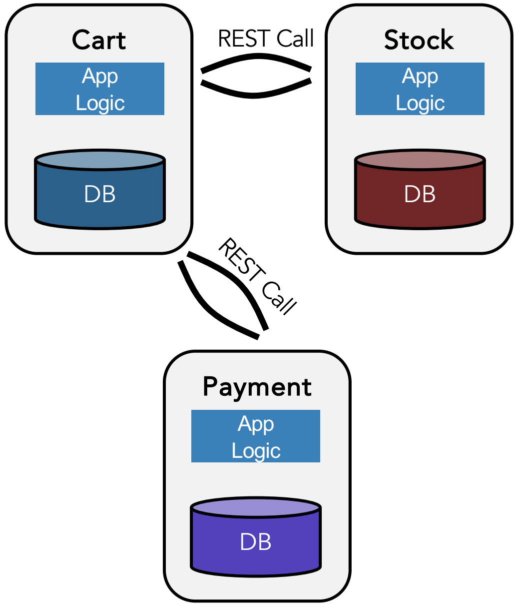
"Textbook" Microservice



- Multi-thread application servers
- App-level business logic
- Object-relational mapping
- Service communication
- DBMS-based concurrency control

Service Architectures Primer: Microservice

“Textbook” Microservice



Holistic resource provisioning

Scalability

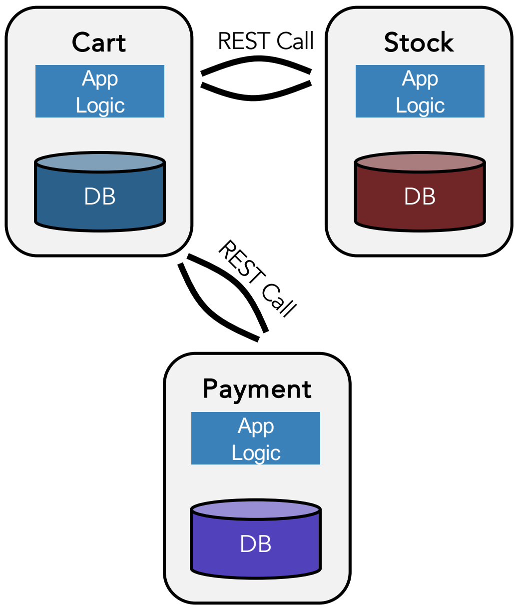
High data availability

Fault isolation

Team independence
- Schema changes
- Data models
- Workloads
- Deployment

Service Architectures Primer: Microservice

“Textbook” Microservice



Synchronous, stateless

- REST
- HTTP, gRPC

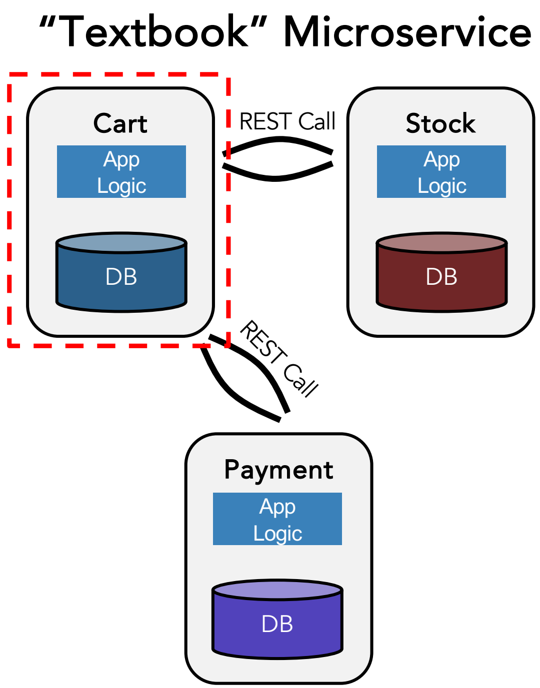
Asynchronous, persistent

- Message brokers
- Event log systems

Checkout code example w/ Spring

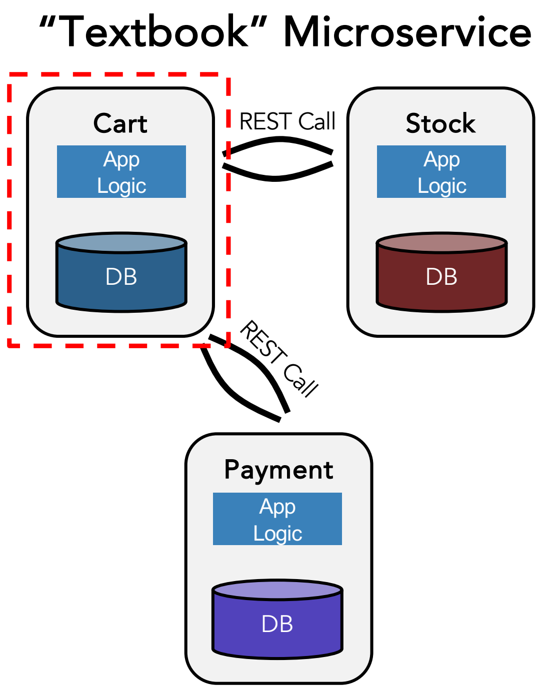
```java
@RestController
@RequestMapping("/cart")
public class CartController {

    @RequestMapping(value = "/{customerId}/add",
                    method = { RequestMethod.PUT, RequestMethod.PATCH })
    public ResponseEntity<?> addItem(@PathVariable int customerId,
                                     @RequestBody CartItem item) {
        // ...
        var res = httpClient
            .patch()
            .uri("https://example.com/stock/{itemId}", item.id)
            .contentType(APPLICATION_JSON)
            .body(item)
            .retrieve()
            .toBodilessEntity();
        if(res.getStatusCode().isError())
            return ResponseEntity.badRequest("Error adding item to cart");
        cartRepository.insert(customerId, item);
        return ResponseEntity.accepted();
    }
```

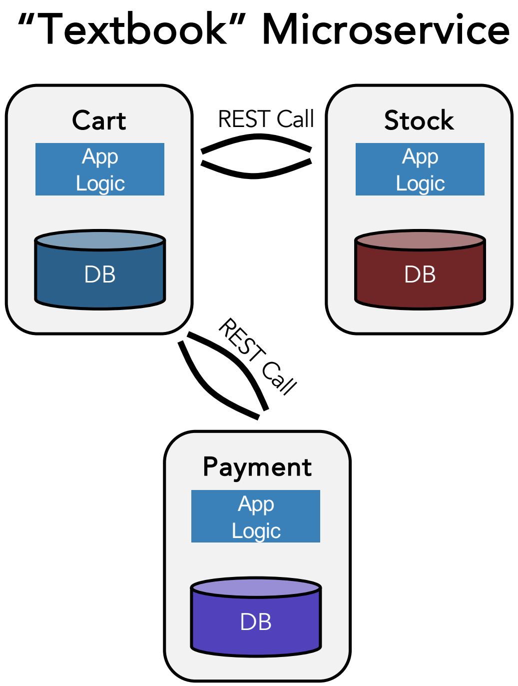


Checkout code example w/ Spring

```java
@PostMapping("/{customerId}/checkout")
public ResponseEntity<?> checkout(@PathVariable int customerId,
                                  @RequestBody CustomerCheckout checkout) {
    var custCart = cartRepository.getCart(customerId);
    custCart.signalCheckout(checkout);
    cartRepository.update(custCart);
    var res = httpClient
        .patch()
        .uri("https://example.com/payment/{customerId}", checkout.customerId)
        .contentType(APPLICATION_JSON)
        .body(custCart)
        .retrieve()
        .toBodilessEntity();
    if(res.getStatusCode().isError())
        return ResponseEntity.badRequest("Error processing payment");
    custCart.signalPayment();
    cartRepository.update(custCart);
    return ResponseEntity.accepted();
}
```



What can go wrong?



What people say about their deployments

Coordination mechanism	%
Orchestration	22.84
Choreography	20.37
Sagas (centralized approach, with a Saga coordinator)	14.81
The Back-end for Front-end Pattern (BFF)	13.58
Sagas (decentralized approach, i.e., no Saga coordinator)	8.64
Distributed transactions (e.g., via 2PC)	8.64
Others	9.88

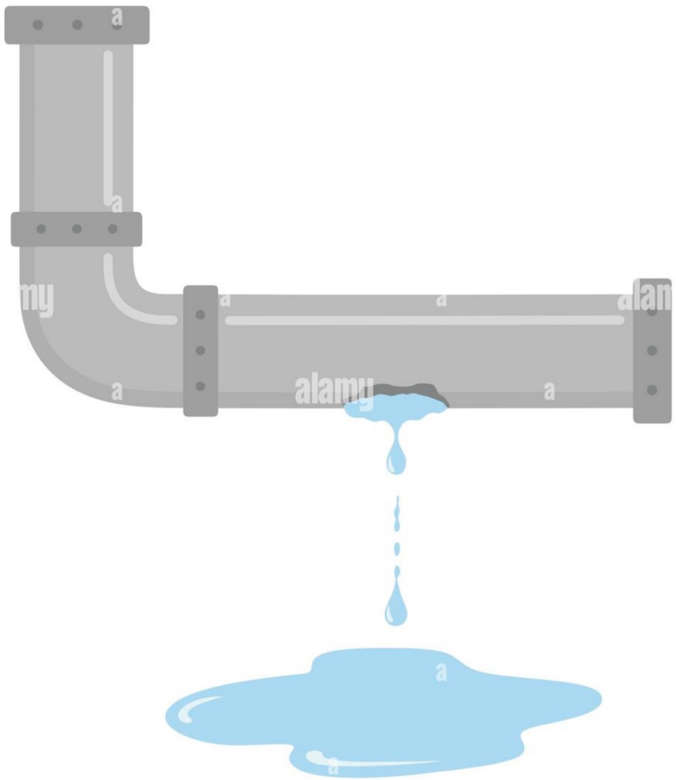
What is going on?

Laigner et al. "Data Management in Microservices: State of the Practice, Challenges, and Research Directions" VLDB (2021)

What people say about their deployments

Distributed commit protocols

- Interoperability
- Complex user code
- Blocking interfaces
- Broken encapsulation



What is being used then?

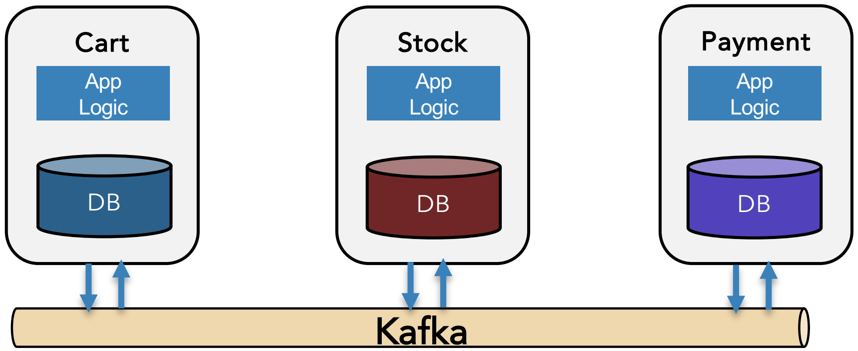
What people say about their deployments

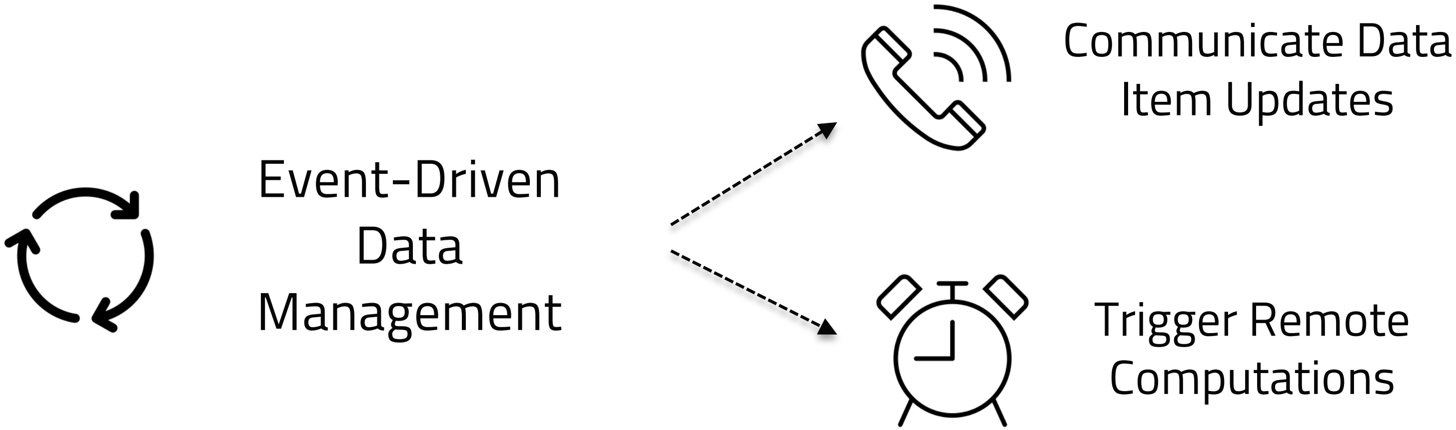
"I am absolutely against the business logic inside the database. Depending on the scale I would **refrain** from using **transactions** at all, favouring an **event-driven approach**, with **eventual consistency** and **micro-transactions**."

What does that mean?

Laigner et al. "Data Management in Microservices: State of the Practice, Challenges, and Research Directions" VLDB (2021)

Event-driven Microservices





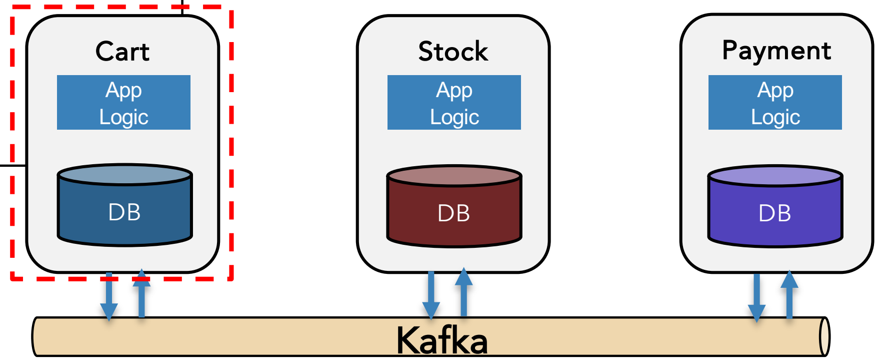
Event-driven Microservices

```java
@Component
public class CartKafkaListener {
    private final CartService cartService;
    public CartKafkaListener(CartService cartService) {
        this.cartService = cartService;
    }
    @KafkaListener(topics = "checkout", groupId = "cart-group")
    public void listenPriceUpdate(String payload) {
        cartService.handleCheckout(payload);
    }
    @KafkaListener(topics = "price-update-topic", groupId = "cart-group")
    public void listenPriceUpdate(String payload) {
        cartService.handlePriceUpdate(payload);
    }
}
```

Kafka Plumbing

Application Logic

Programmatic Configuration



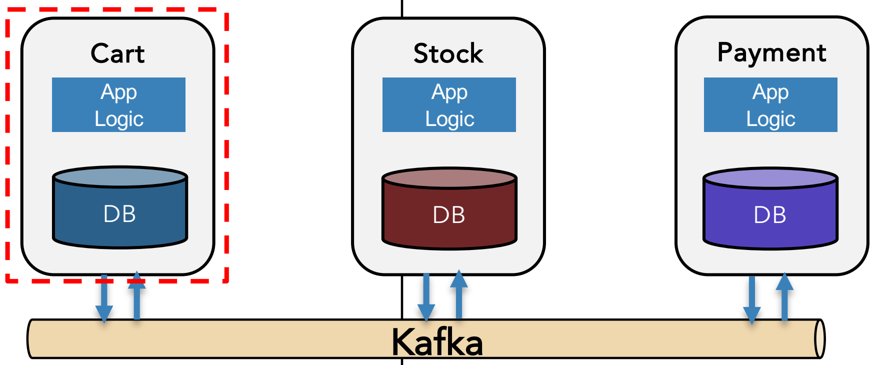
Event-driven Microservices

```java
public class CartService {

    private final ICartRepository cartRepository;
    private final ICartItemRepository cartItemRepository;
    private final IEventPublisher eventPublisher;

    public CartServiceCore(ICartRepository cartRepository,
                           ICartItemRepository cartItemRepository,
                           IEventPublisher eventPublisher) {
        this.cartRepository = cartRepository;
        this.cartItemRepository = cartItemRepository;
        this.eventPublisher = eventPublisher;
    }

    @Override
    public void handleCheckout(CustomerCheckout customerCheckout) {
        Cart cart = cartRepository.findByCustomerId(customerCheckout.getCustomerId());
        List<CartItem> items = cartItemRepository.findByCustomerId(customerCheckout.getCustomerId());
        cartItemRepository.deleteByCustomerId(customerCheckout.getCustomerId());
        ReserveStock checkout = new ReserveStock(customerCheckout, cartItems);
        eventPublisher.publishEvent("reserve-stock-topic", checkout);
    }
```



Kafka publisher abstraction

Interfaces for common database operations (e.g., CRUD)

Actual application logic

What can go wrong?

Actors

Akka, Orleans, etc.

The Actor Model

Actors

Isolated components

Communicate via *asynchronous message passing*

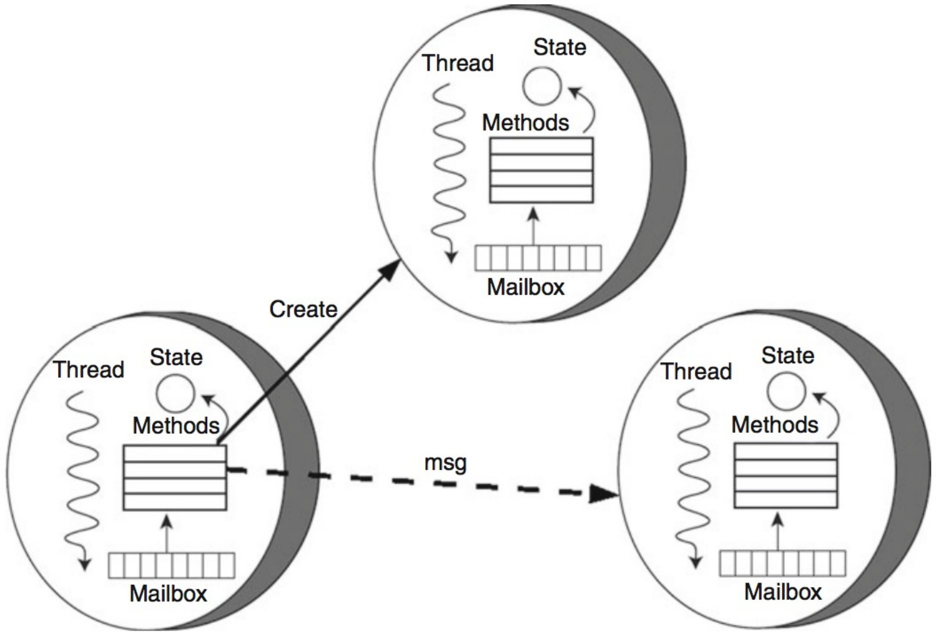
Upon receipt of a message, an actor can:

Create other actors

Send messages

Change its behavior

- Functional-style: Replace its function
- OO-style: Update its encapsulated state, thus affecting its behavior



Source: Rajesh K. Karmani, Gul Agha: Actors. Encyclopedia of Parallel Computing 2011: 1-11

The Actor Model

Key semantic properties

Encapsulation of state

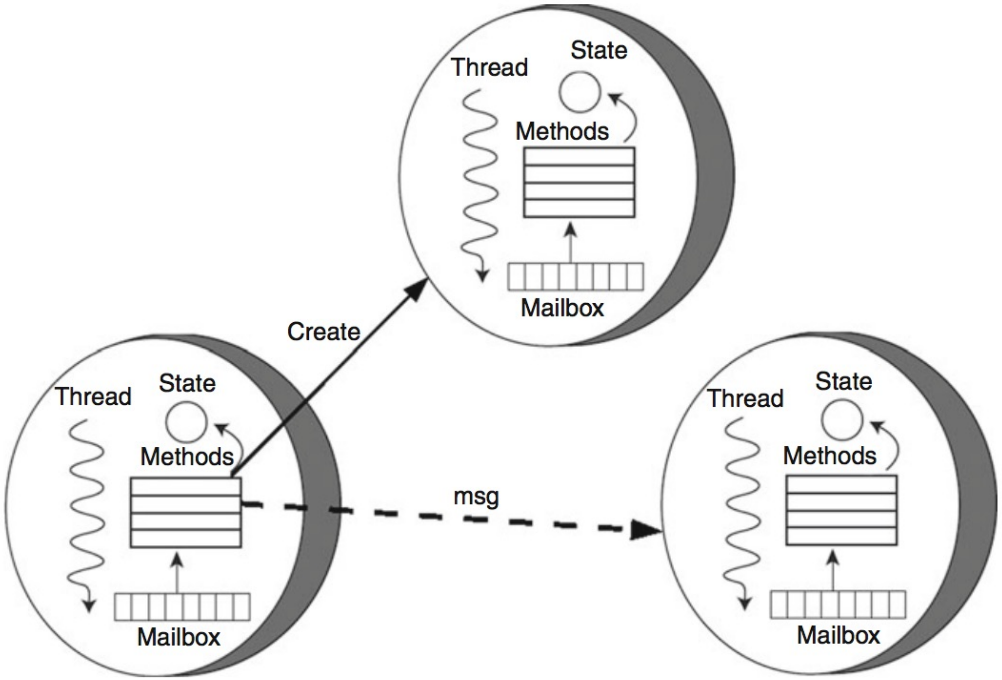
Atomic execution of methods

Location transparency

Fairness in scheduling

Asynchronous communication

- Unknown message delivery delay
- But messages will get delivered *eventually*



Source: Rajesh K. Karmani, Gul Agha: Actors. Encyclopedia of Parallel Computing 2011: 1-11

Coordination in the actor model

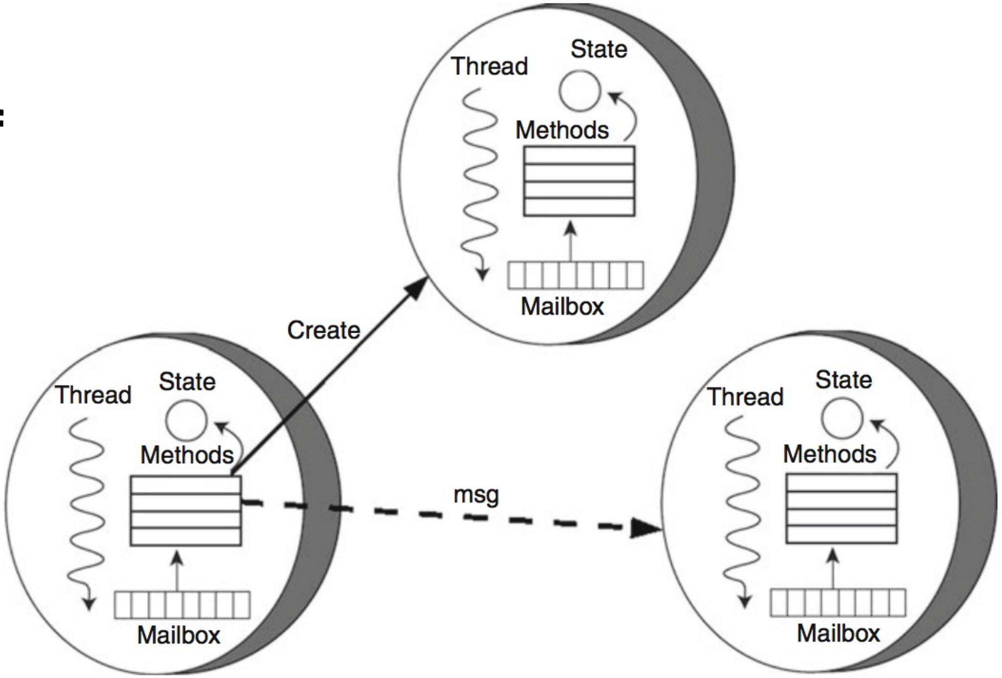
Reason about concurrency in terms of

The interleaving of messages to actors

Rather than interleaving access to shared variables

Key complexity

- Many possible interleavings of messages to groups of actors



Source: Rajesh K. Karmani, Gul Agha: Actors. Encyclopedia of Parallel Computing 2011: 1-11

Problems with Actor Model Frameworks

Too low level

- App manages lifecycle of actors, exposed to distributed races
- App has to deal with actor failures, supervision trees
- App manages placement of actors – resource management

Developer has to be a distributed systems expert

Source: *Philip A. Bernstein, Sergey Bykov: Developing Cloud Services Using the Orleans Virtual Actor Model. IEEE Internet Comput. 20(5): 71-75 (2016)*

Orleans: Virtual Actors

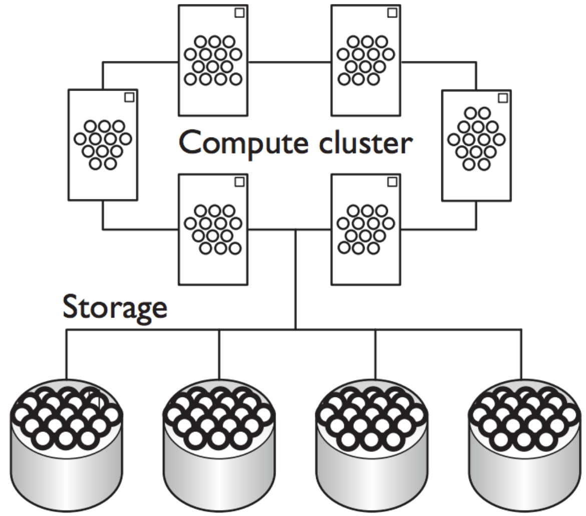
Actor instances always exist, virtually

Location transparency (logical actor reference)

Activations are created on demand (managed by the runtime)

The actor saves its state to storage whenever it wants

ACID transactions over multiple actors



Source: *Philip A. Bernstein, Sergey Bykov: Developing Cloud Services Using the Orleans Virtual Actor Model. IEEE Internet Comput. 20(5): 71-75 (2016)*

Checkout code example w/ Orleans

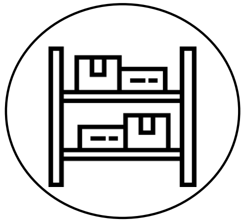
```
public class CartActor : Grain, ICartActor
{
    protected readonly IPersistentState<Cart> cart;

    public CartActor([PersistentState(
        stateName: "cart",
        storageName: Constants.OrleansStorage)] IPersistentState<Cart> state)
    {
        this.cart = state;
    }

    public virtual async Task AddItem(CartItem item)
    {
        var stockActor = this.GrainFactory.GetGrain<IStockActor>(item.ID);
        var res = stockActor.placeItem(item.ID, item.qty);
        if(!res.isError()){
            this.cart.State.items.Add(item);
            await this.cart.WriteStateAsync();
            return;
        }
        throw new Exception("Cart [" + this.customerId + "]: Error adding item to cart.");
    }
```

```
public class Cart
{
    public int customerId;
    public CartStatus status { get; set; } = CartStatus.OPEN;
    public List<CartItem> items { get; set; } = new List<CartItem>();
    public Cart() {}
    public Cart(int customerId) {
        this.customerId = customerId;
    }
}
```

Cart

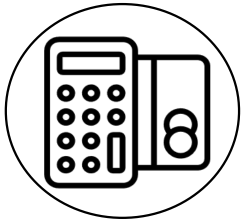


Stock

Checkout code example w/ Orleans

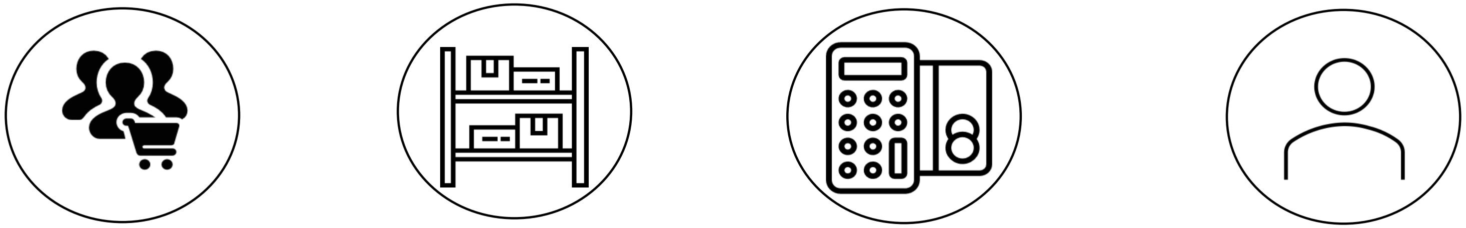
```
public virtual async Task NotifyCheckout(CustomerCheckout customerCheckout)
{
    this.cart.State.status = CartStatus.CHECKOUT_REQUESTED;
    await this.cart.WriteStateAsync();
    var paymentActor = this.GrainFactory.GetGrain<IPaymentActor>(customerId);
    var res = await paymentActor.processPayment(this.cart.State);
    if(res.isError()){
         throw new Exception("Cart [" + this.customerId + "]: Error processing payment.");
    }
    this.cart.State.status = CartStatus.OPEN;
    this.cart.State.items.Clear();
    await this.cart.WriteStateAsync();
}
```

Cart



Payment

Caveats of the Actor Model



Cart Stock Payment Customer

- Actor vs Object
  - E.g., should stock be an object or an actor?
- Should Cart and Customer be one or two actors?
  - One task -> one actor
  - Different actors capture different simultaneous tasks.
- Granularity of Actors
  - Coordination overhead vs. concurrency

Source: *Yiwen Wang, et al: Modeling and Building IoT Data Platforms with Actor-Oriented Databases. EDBT 2019.*

Serverless Functions

Function as a Service

A model of application function execution

Basic model: function isolation and invocation

- Logical: function identifier and URL, language runtime
- Physical: instance settings (memory, CPU)

Event-driven/trigger-based execution

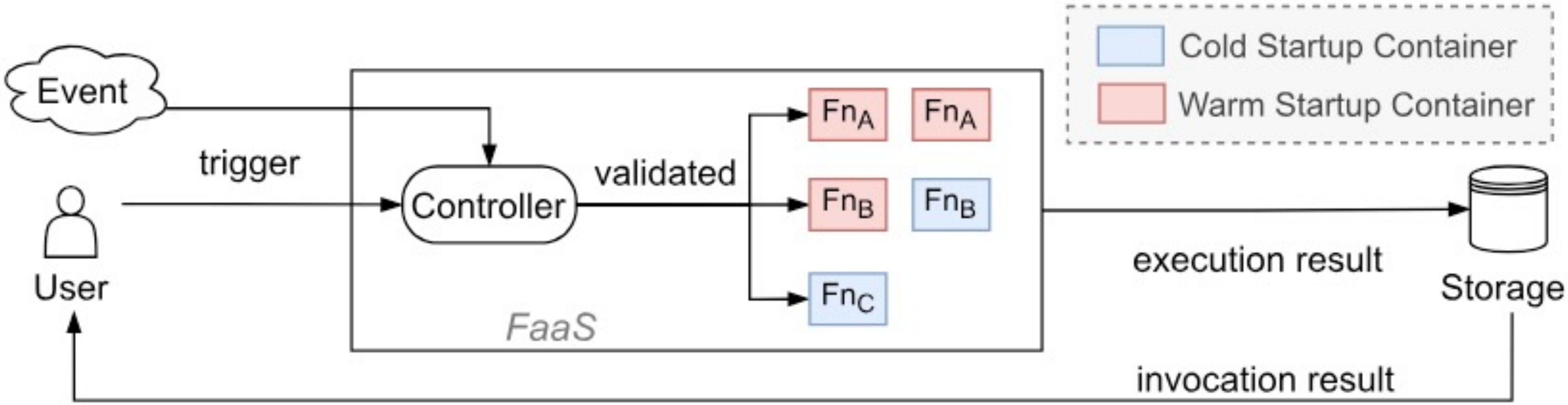
- External events (i.e., clicks, streams)

Application = collection of functions

- Chained *f-to-f* invocation $\{f_A, f_B, f_C\}$

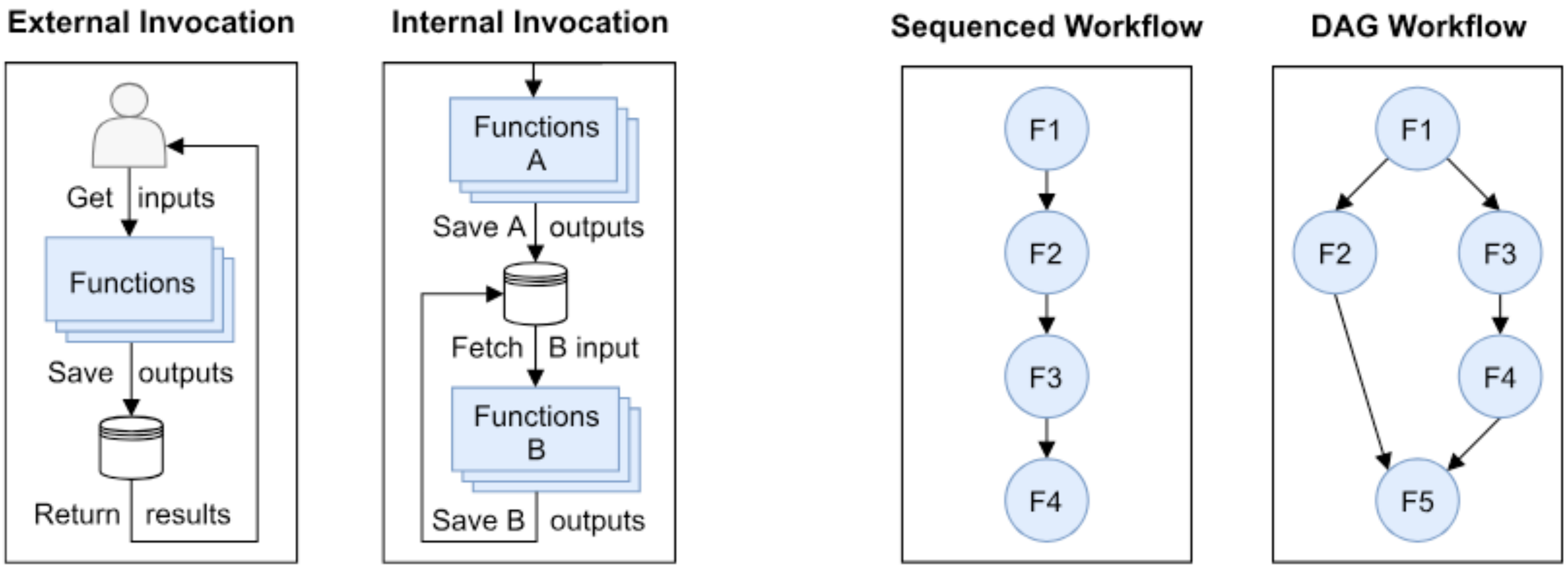
Z. Li et al. “The Serverless Computing Survey: A Technical Primer for Design Architecture.” ACM Comput. Surv. (2022)

Function as a Service



Z. Li et al. “The Serverless Computing Survey: A Technical Primer for Design Architecture.” ACM Comput. Surv. (2022)

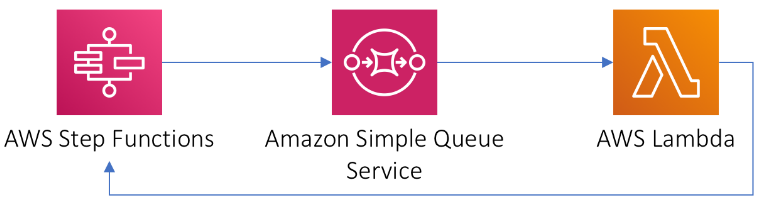
Function as a Service – Invocation Patterns

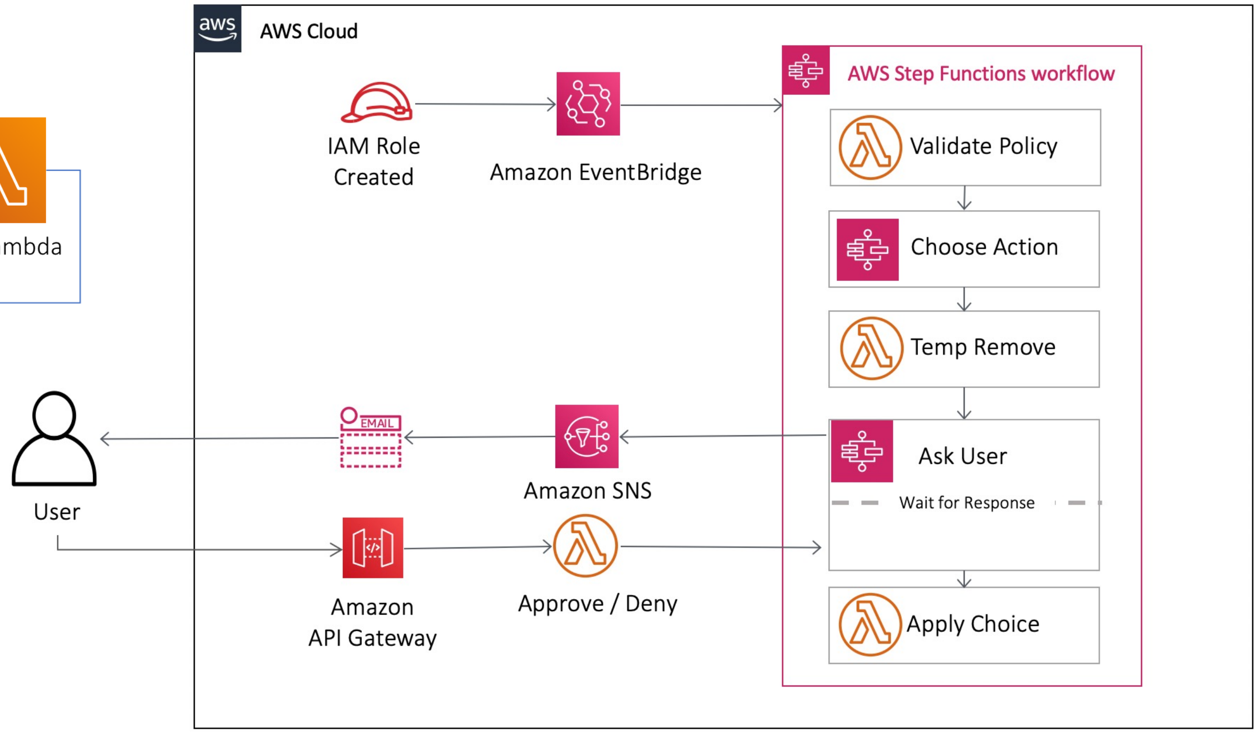


(a) Invocation patterns (b) Workflow execution models

Z. Li et al. "The Serverless Computing Survey: A Technical Primer for Design Architecture." ACM Comput. Surv. (2022)

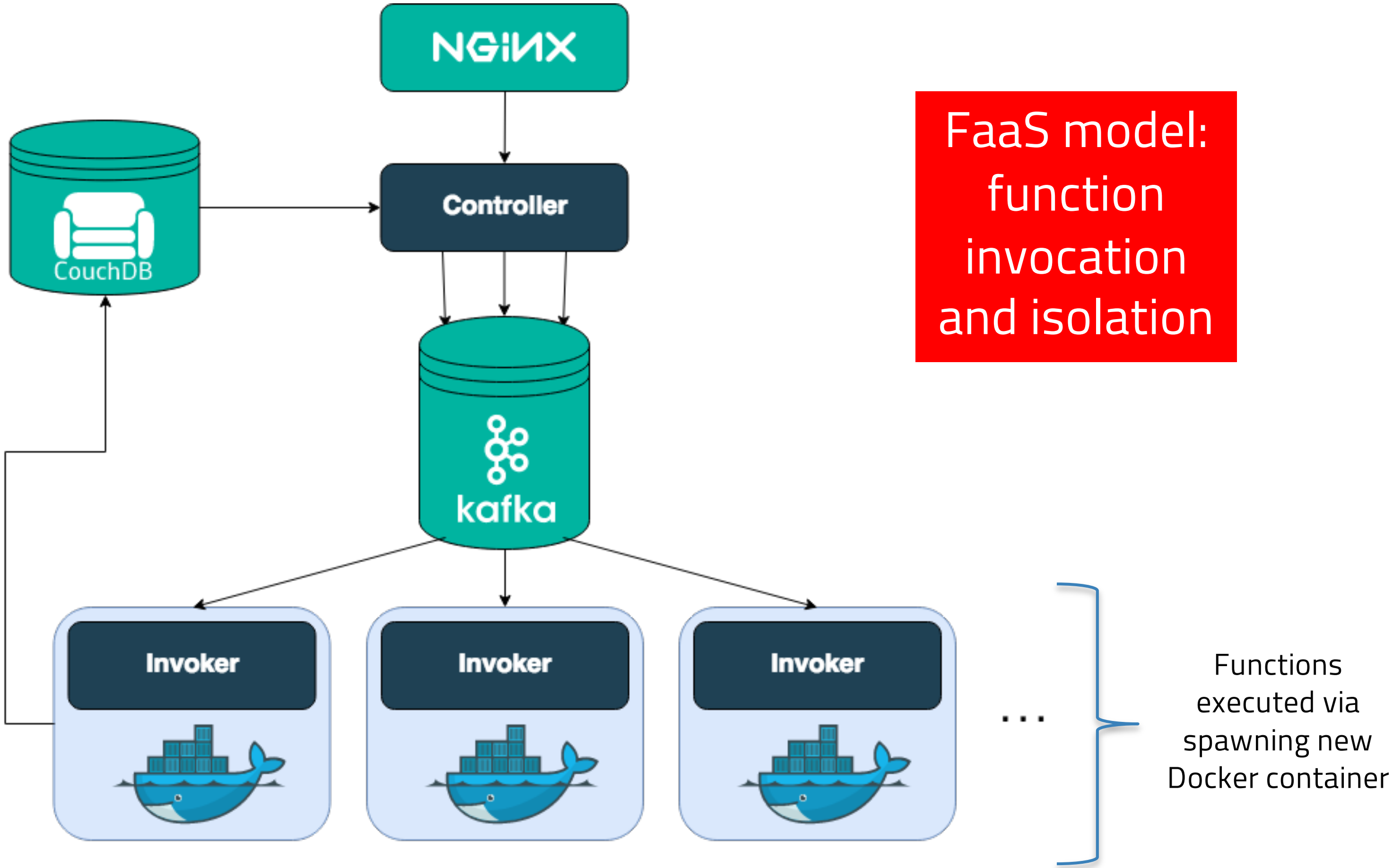
Function as a Service – Workflow Scenarios





Amazon AWS (2025)

Function as a Service – Typical System



FaaS model: function invocation and isolation

Apache OpenWhisk (2020)

Apache OpenWhisk

```
const openwhisk = require('openwhisk');

async function main(params) {
  const ow = openwhisk(); // uses env vars set in OpenWhisk

  const result = await ow.actions.invoke({
    actionName: 'greetUser',
    params: { name: 'Alice' },
    blocking: true,
    result: true // only return the result, not full activation
  });

  return { message: `Invoked greetUser and got: ${result.greeting}` };
}
```

Stateless functions

No state functionality

exposed to functions/users

CouchDB is used internally for

metadata

```
function main(params) {
  const name = params.name || 'stranger';
  return { greeting: `Hello, ${name}!` };
}
```

Apache OpenWhisk (2020)

https://openwhisk.apache.org/

AWS Step Functions

```javascript
exports.handler = async (event) => {
  const name = event.name || "World";
  const message = `Hello, ${name}!`;

  return {
    greeting: message,
    timestamp: new Date().toISOString()
  };
};
```

```json
{
  "Comment": "A workflow that composes two Lambda functions",
  "StartAt": "Invoke FunctionA",
  "States": {
    "Invoke FunctionA": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FunctionA",
      "ResultPath": "$.FunctionAResult",
      "Next": "Invoke FunctionB"
    },
    "Invoke FunctionB": {
      "Type": "Task",
      "Resource": "arn:aws:lambda:REGION:ACCOUNT_ID:function:FunctionB",
      "InputPath": "$.FunctionAResult",
      "ResultPath": "$.FunctionBResult",
      "End": true
    }
  }
}
```

```javascript
exports.handler = async (event) => {
  const originalMessage = event.greeting || "No greeting found.";
  const response = `${originalMessage} This was processed by FunctionB.`;

  return {
    finalMessage: response,
    processedAt: new Date().toISOString()
  };
};
```

Amazon AWS (2025)

https://aws.amazon.com/step-functions/

AWS Step Functions

```javascript
exports.handler = async (event) => {
  // 🟢 Retrieve state
  const name = event.name || "User";
  const count = event.count || 0;

  // 🔵 Modify state
  const newCount = count + 1;
  const message = `Hello, ${name}. You've been here ${newCount} times.`;

  // 🔴 Return updated state
  return {
    ...event,               // keep previous state
    count: newCount,        // overwrite count
    message: message        // add new field
  };
};
```

```javascript
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {
  const userId = event.userId;

  if (!userId) {
    throw new Error("Missing userId in input");
  }

  // 🟢 Get current count
  let visits = 0;
  try {
    const data = await ddb.get({
      TableName: 'UserVisits',
      Key: { userId }
    }).promise();
```

Amazon AWS (2025)

https://aws.amazon.com/step-functions/

Cloudburst – A stateful serverless platform

```python
from cloudburst import *
cloud = CloudburstClient(cloudburst_addr, my_ip)
cloud.put('key', 2)
reference = CloudburstReference('key')
def sqfun(x): return x * x
sq = cloud.register(sqfun, name='square')

print('result: %d' % (sq(reference))
> result: 4

future = sq(3, store_in_kvs=True)
print('result: %d' % (future.get())
> result: 9
```

Low composition overhead

Direct communication

Low-latency state access

Anna as a KV store

```python
>>> from cloudburst.client.client import CloudburstConnection
>>> local_cloud = CloudburstConnection('127.0.0.1', '127.0.0.1', local=True)
>>> cloud_sq = local_cloud.register(lambda _, x: x * x, 'square')
>>> cloud_sq(2).get()
4
>>> local_cloud.register_dag('dag', ['square'], [])
>>> local_cloud.call_dag('dag', { 'square': [2] }).get()
4
```

Sreekanti et al. “Cloudburst: Stateful Functions-as-a-Service.” VLDB (2020)

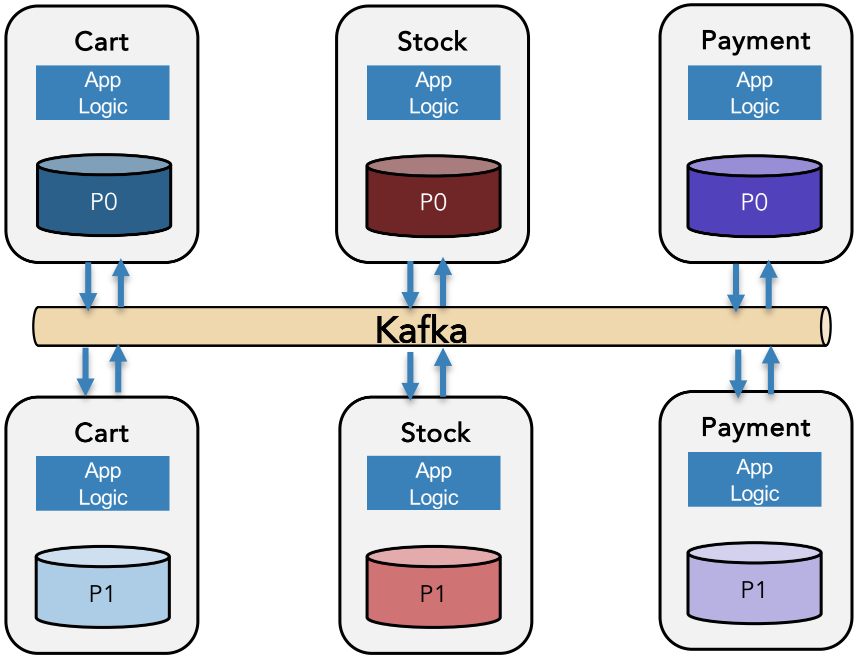
https://github.com/hydro-project/cloudburst

Impedance mismatch with data-centric apps?

- Developers follow opinioned software designs
  - Frameworks like Spring capture this essence
  - Implicit software design patterns, annotation-based config
- Complex highly modularized applications
  - Missing object-oriented constructs
  - Object-oriented relational mapping
- Adoption among enterprise application architectures
  - Latency-sensitive components often through runtimes (e.g., JVM)
  - 100%-based FaaS applications are not the de facto approach (yet)

Dataflow

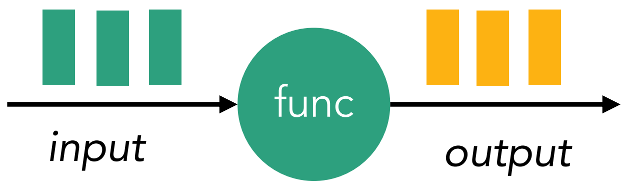
Scalable, event-driven μServices <u>are</u> parallel streaming dataflows



This is a partitioned, stateful, streaming dataflow graph. Built by hand.

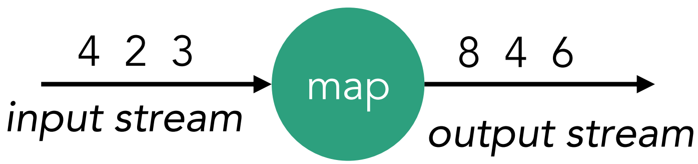
Dataflow Programming Primer (1)

Input, operator, output



Dataflow Programming Primer (2)

Example: read numbers and double them



In Scala

```
inputStream.map(number => number*2)
```

User-defined Function

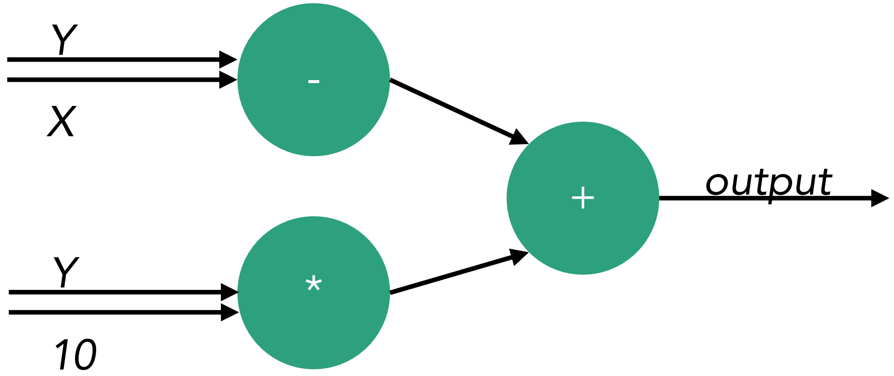
Dataflow Programming Primer (1)

A =X - Y
B = Y *10
C = A + B

Serially in von Neumann

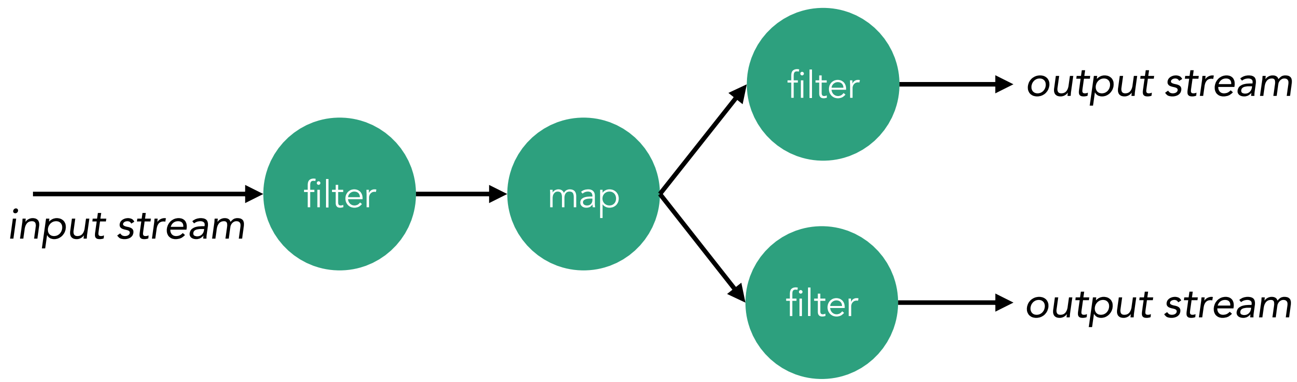
3 time units needed

2 time units needed

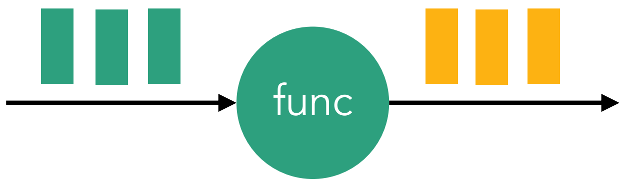


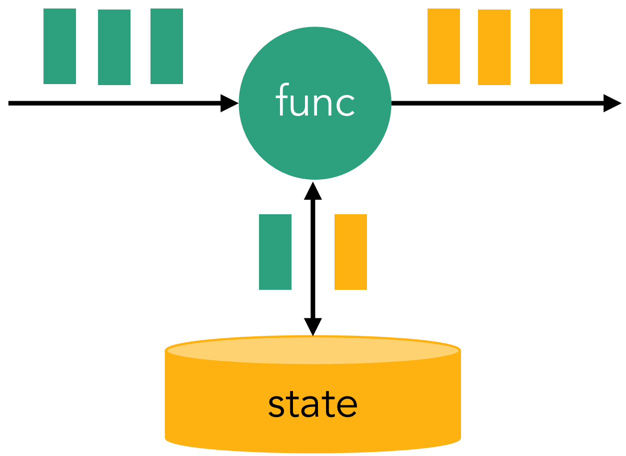
Complex Dataflows by Combining Functions

*In dataflow programming we model a program as a **directed graph** of the **data flowing** through **operators***



Stateless vs. Stateful Functions





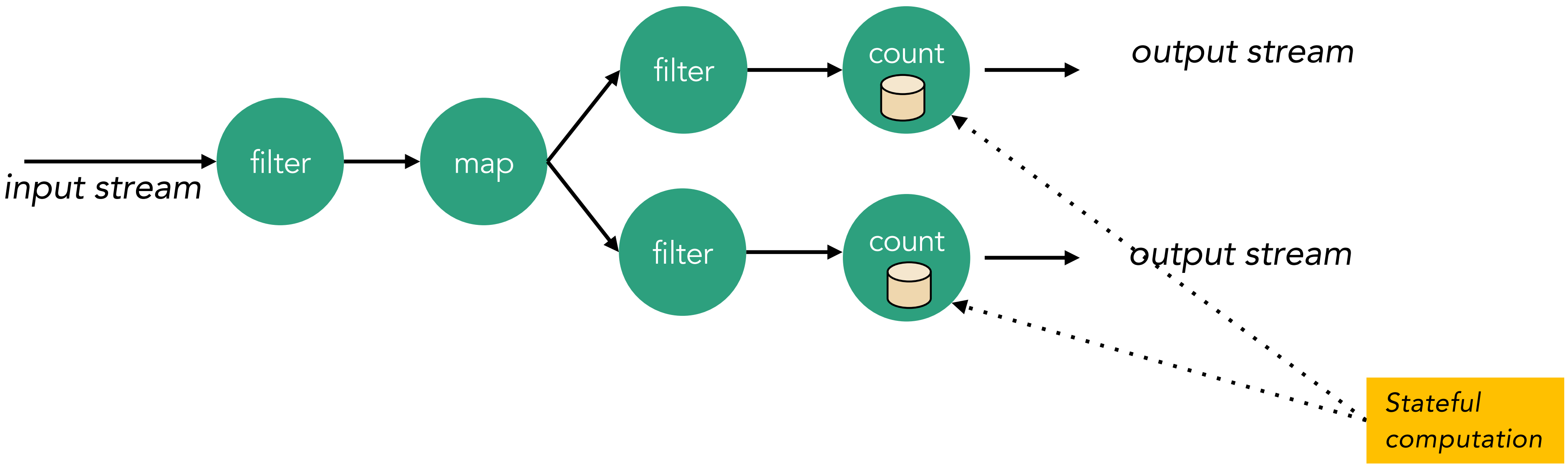
Stateless functions

Filters, simple maps, etc.

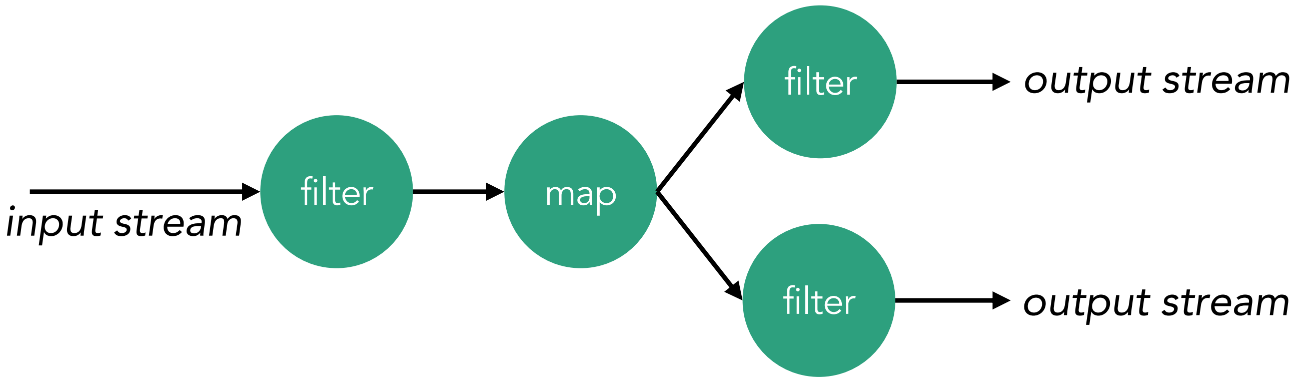
Stateful functions

Counters, sums, joins, etc.

Stateful vs. Stateless operators in Dataflows

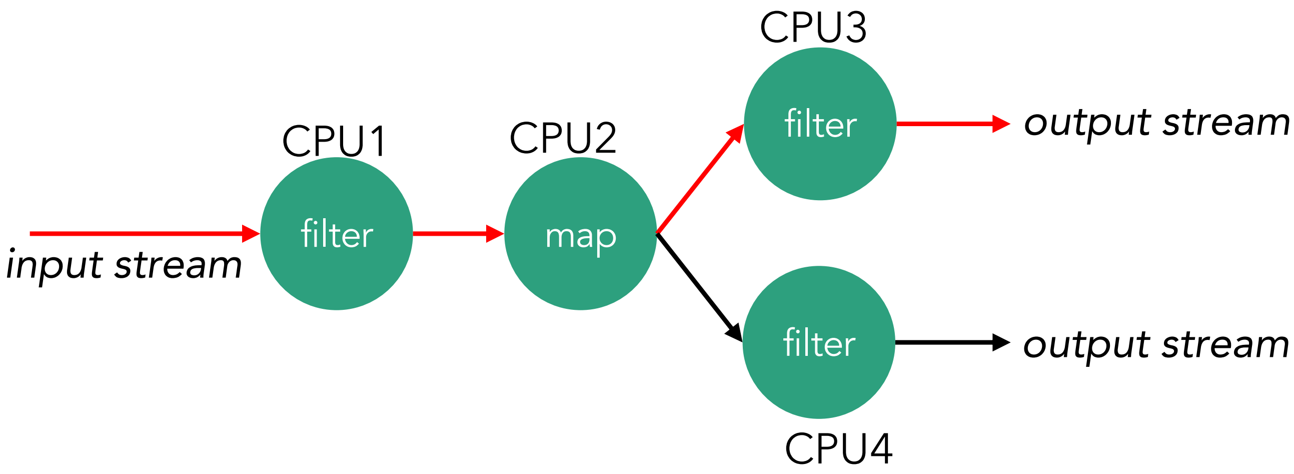


Parallelization of Dataflow Programs



Pipeline-Parallelism

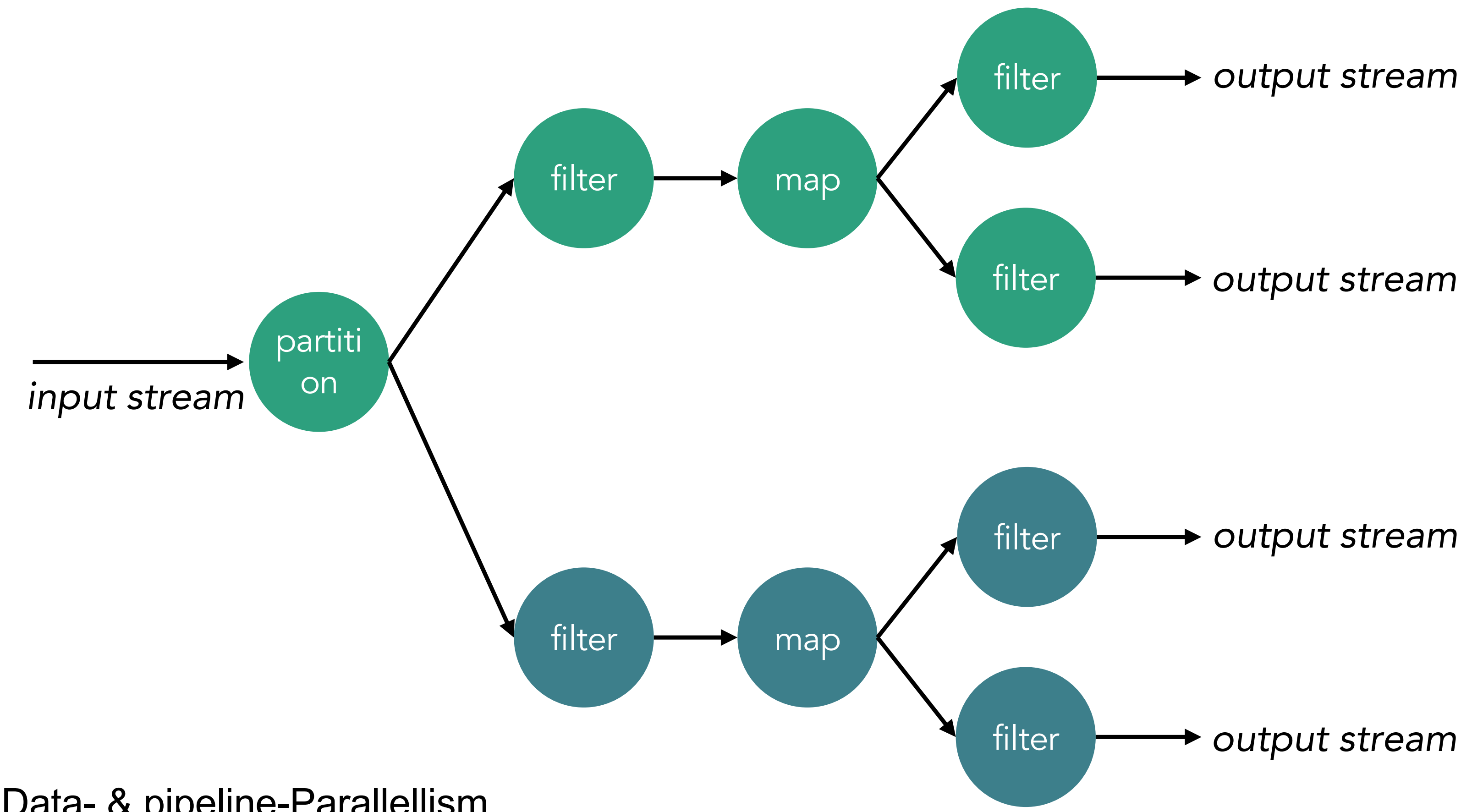
Parallelization of Dataflow Programs



Pipeline-Parallelism

4 CPUs max

Parallelization of Dataflow Programs



Data- & pipeline-Parallellism

9 CPUs max

Complex Dataflows by Combining Functions

```java
// Transformations: Count the words
DataStream<Tuple2<String, Integer>> wordCounts = text
        .flatMap(new LineSplitter())
        .keyBy(0)
        .timeWindow(Time.seconds(5))
        .sum(1);
```

Dataflows as substrate for Cloud Services

Decoupled software components

People “abuse” dataflow systems to implement microservices

No reliance on global program counter or global memory.

Cloud applications (often) need to be rewritten

Transactions over dataflows are not straight-forward

Summary

"Cloud apps are very awkward to program at the moment”

API abstractions leak system-oriented handling to the app logic

Developers should be able to not transform their imperative code