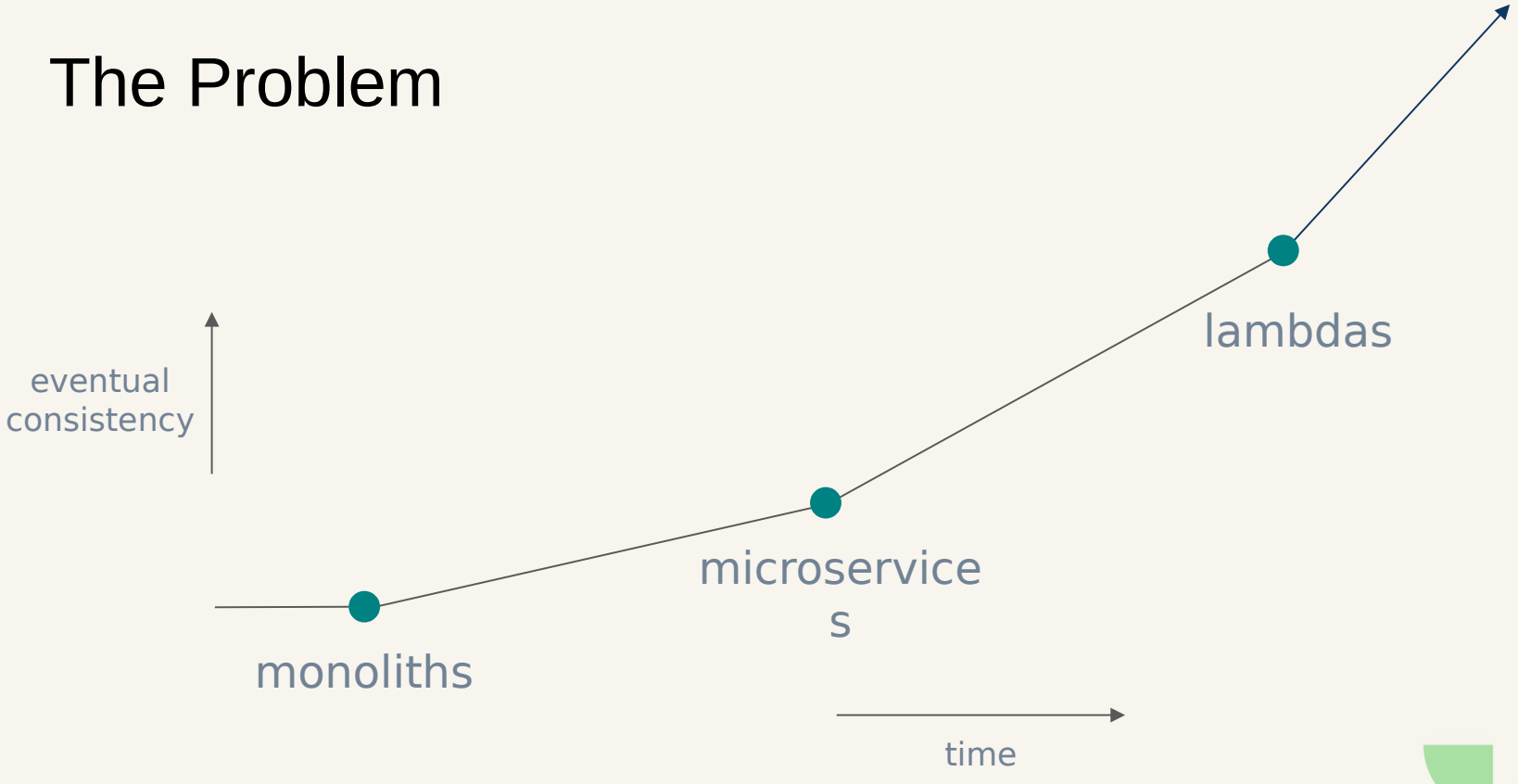
Reliable serverless needs **distributed transactions**



Riley Scheid
reboot.dev

The Problem

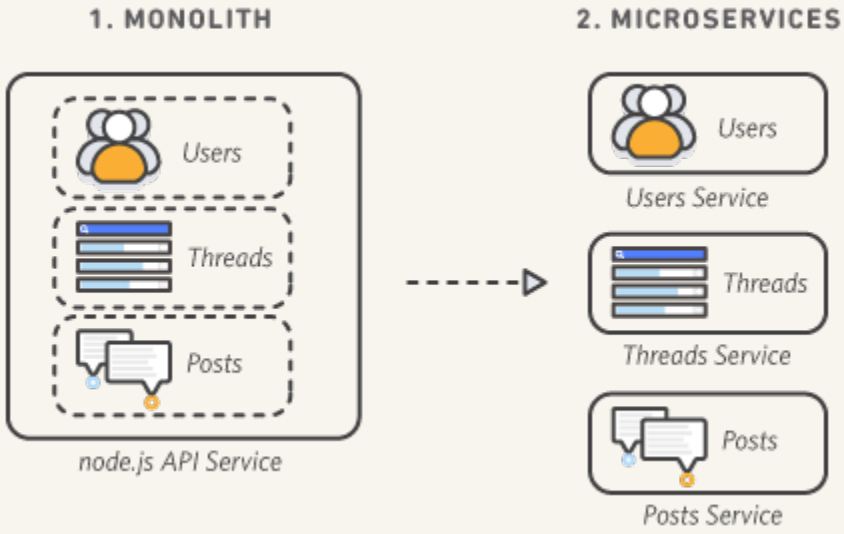




reboot

Microservices have a lot going for them

- Modularity makes it easier to understand, modify, and test.
- Engineering agility because teams can operate and deploy their software more independently/autonomously.
- Enable technology diversity behind well encapsulated API.
- Easier to scale the application and easier to scale the company.



reboot

Reflecting …

- Container orchestrators were *built for operators*, not application developers.
- Application developers were told to split their applications into microservices because it made it easier for us, the operators, to run their applications on container orchestrators so we could do software and hardware maintenance, we could provision and deprovision resource, etc…



Microservices have a lot going for them

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

✅ Independent teams can own different services: **Cart**, **Shipping**, **Payment**, **Emailer**, etc.



reboot

But microservices are hard to build correctly

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

❌

✅ Independent teams can own different services: **Cart**, **Shipping**, **Payment**, **Emailer**, etc.



reboot

But microservices are hard to build correctly

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

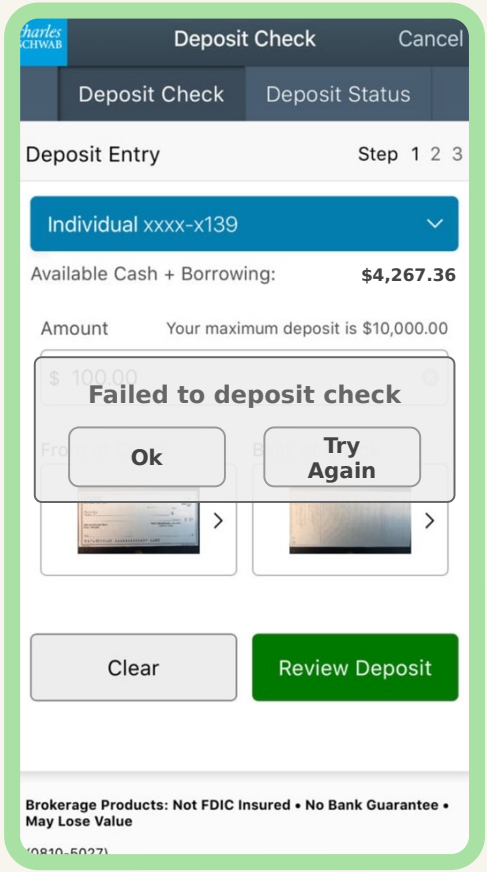
❌

✅ Independent teams can own different services: **Cart**, **Shipping**, **Payment**, **Emailer**, etc.

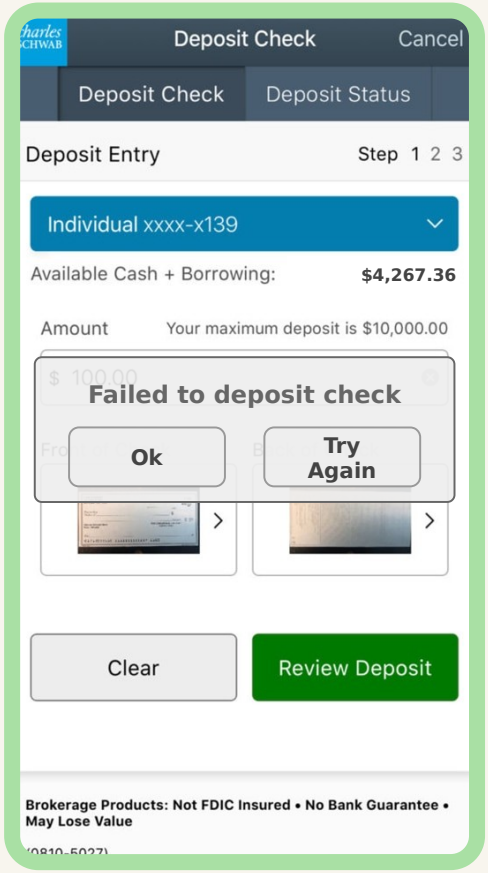


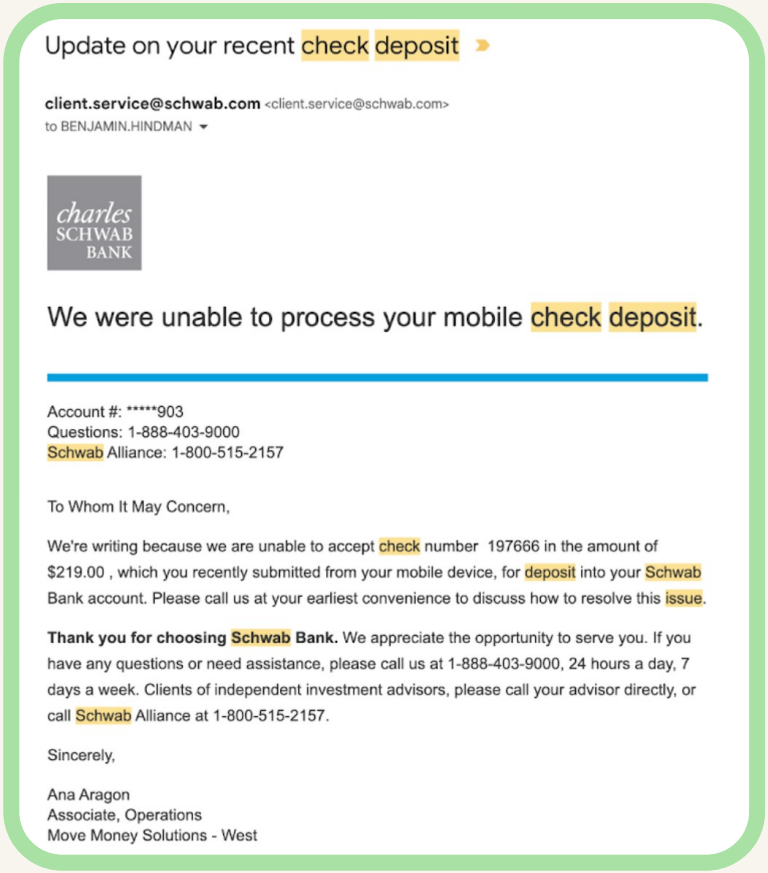
reboot

Real bugs


Deposit Check
Cancel
Deposit Check
Deposit Status
Deposit Entry
Step 1 2 3
Individual xxxx-x139
Available Cash + Borrowing:
$4,267.36
Amount
Your maximum deposit is $10,000.00
Failed to deposit check
Ok
Try Again
Clear
Review Deposit
Brokerage Products: Not FDIC Insured • No Bank Guarantee • May Lose Value

Real bugs


Deposit Check
Cancel
Deposit Check
Deposit Status
Deposit Entry
Step 1 2 3
Individual xxxx-x139
Available Cash + Borrowing:
$4,267.36
Amount
Your maximum deposit is $10,000.00
Failed to deposit check
Ok
Try Again
Clear
Review Deposit
Brokerage Products: Not FDIC Insured • No Bank Guarantee • May Lose Value


Update on your recent check deposit
client.service@schwab.com <client.service@schwab.com>
to BENJAMIN.HINDMAN
charles SCHWAB BANK
We were unable to process your mobile check deposit.
Account #: *****903
Questions: 1-888-403-9000
Schwab Alliance: 1-800-515-2157
To Whom It May Concern,
We're writing because we are unable to accept check number 197666 in the amount of $219.00 , which you recently submitted from your mobile device, for deposit into your Schwab Bank account. Please call us at your earliest convenience to discuss how to resolve this issue.
Thank you for choosing Schwab Bank. We appreciate the opportunity to serve you. If you have any questions or need assistance, please call us at 1-888-403-9000, 24 hours a day, 7 days a week. Clients of independent investment advisors, please call your advisor directly, or call Schwab Alliance at 1-800-515-2157.
Sincerely,
Ana Aragon
Associate, Operations
Move Money Solutions - West

But microservices are hard to build correctly

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

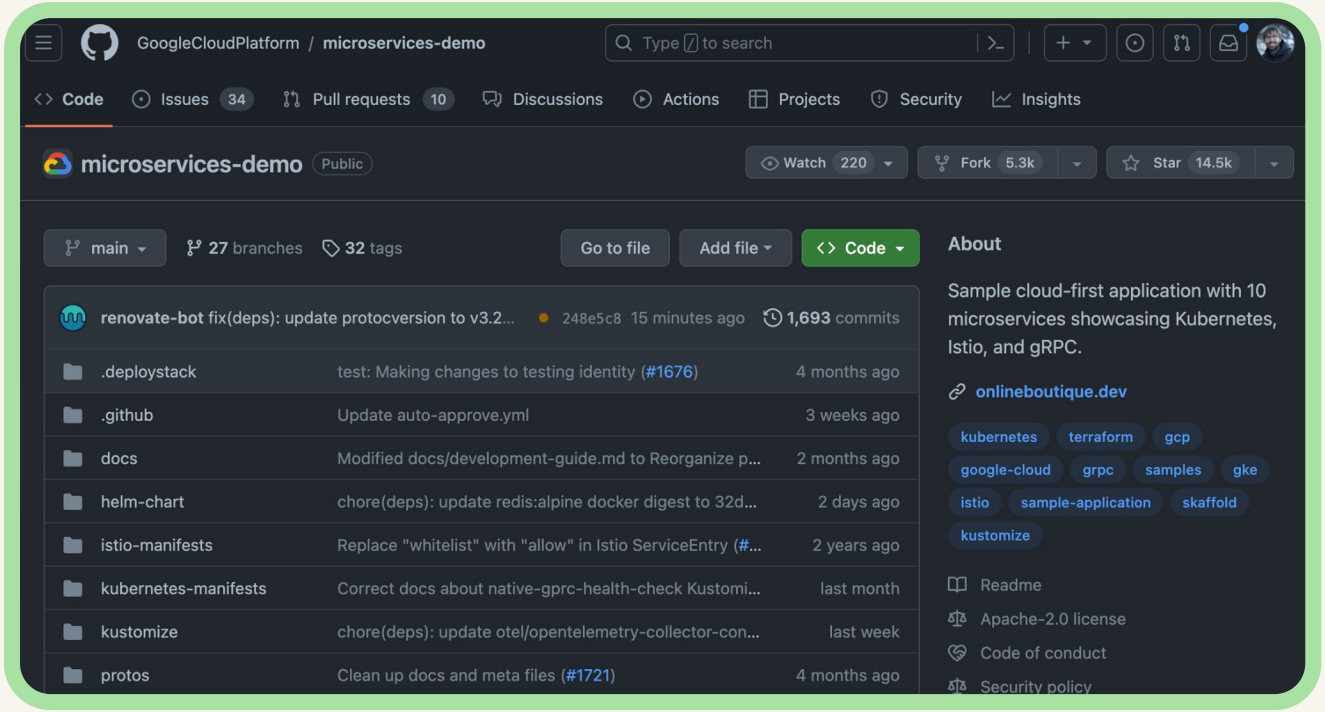
✅ Independent teams can own different services: **Cart**, **Shipping**, **Payment**, **Emailer**, etc.

❌ Anyone who does this has not run microservices in anger.



reboot

Microservices are hard to build correctly



reboot

From inconsistent to eventually consistent

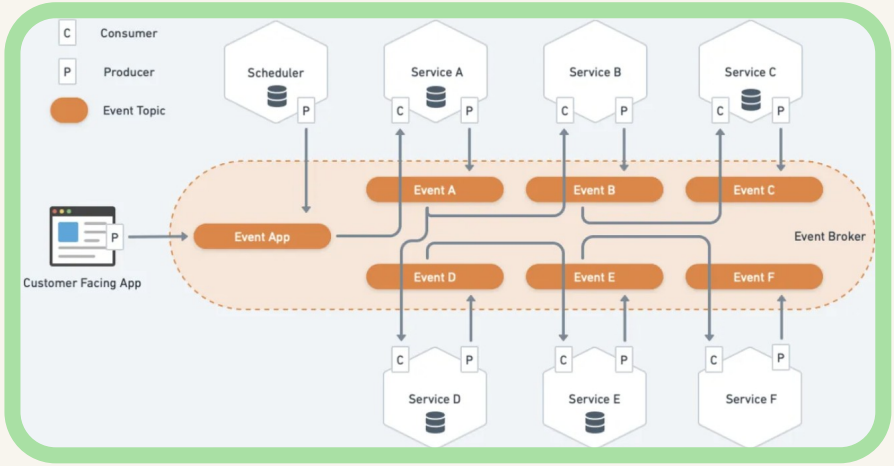
```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

✅ Event sourcing or durable execution can fix the inconsistencies of the last example.



reboot

From inconsistent to eventually consistent



- Operators need to manage a distributed queue.
- Inversion of control, “callback hell.”
- Business processes scattered throughout the codebase.
- “Queue glued”

reboot

From inconsistent to eventually consistent

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
✅  items = await cart.GetItems(...)
    ...
✅  await shipping.PrepareShipOrder(items, ...)
    ...
✅  await payment.AuthorizeCharge(items, ...)
    ...
❌  await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

✅ Event sourcing or durable execution can fix the inconsistencies of the last example.

❌ Devs often need to write compensating actions in case of failure.



reboot

From inconsistent to eventually consistent

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
✅  items = await cart.GetItems(...)
    ...
✅  await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

✅ Event sourcing or durable execution can fix the inconsistencies of the last example.

❌ Devs often need to write compensating actions in case of failure.



reboot

From inconsistent to eventually consistent

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
✅  items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

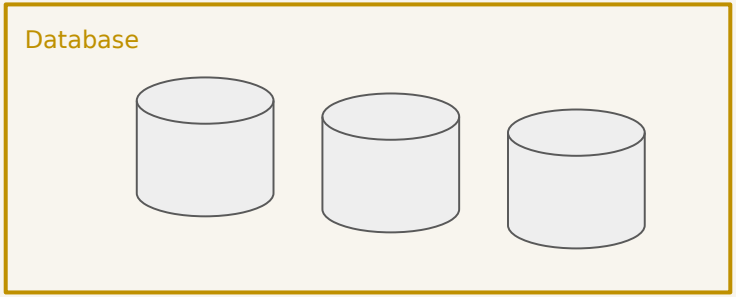
✅ Event sourcing or durable execution can fix the inconsistencies of the last example.

❌ Devs often need to write compensating actions in case of failure.



reboot

Eventual consistency in a strongly consistent world

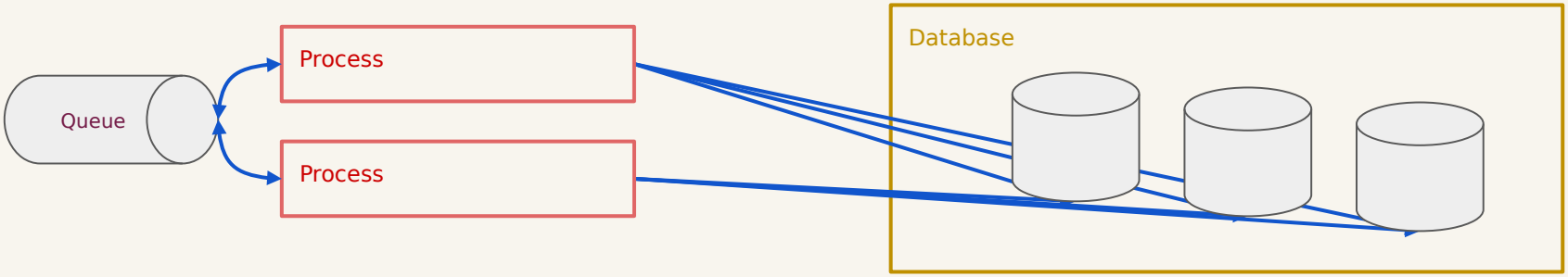


Strongly consistent distributed databases:

- Spanner
- CockroachDB
- Yugabyte
- ...



Eventual consistency in a strongly consistent world



Eventually consistent client applications, due to:

- queues
- event sourcing
- lambda architecture
- durable execution
- ...

Strongly consistent distributed databases:

- Spanner
- CockroachDB
- Yugabyte
- ...

reboot

This is not the architecture you're looking for …

"You shouldn't start a new project with microservices, even if you're sure your application will be big enough to make it worthwhile."

– Martin Fowler

Conventional wisdom: only adopt microservices when you graduate off of monoliths (i.e., when your product and company are scaling like crazy, because that's the perfect time to completely rewrite everything 🤪)



reboot

We gave you an insufficient abstraction

“[...] microservices conflate logical boundaries (how code is written) with physical boundaries (how code is deployed).”

– Sanjay Ghemawat et. al. (Towards Modern Development of Cloud Applications)

Developers *want to* write their applications **logically as monoliths**, but in such a way that a runtime can decide how to distribute and run the application code across resources.



reboot

From eventually consistent to strongly consistent

```
async def Checkout(
    self,
    request: CheckoutRequest,
    context: ServicerContext,
):
    ...
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

Developers intent is that this gets executed **atomically**!

(And ***idempotently*** if retried)



reboot

How to distribute transactions?

```
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest,
):
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

checkout

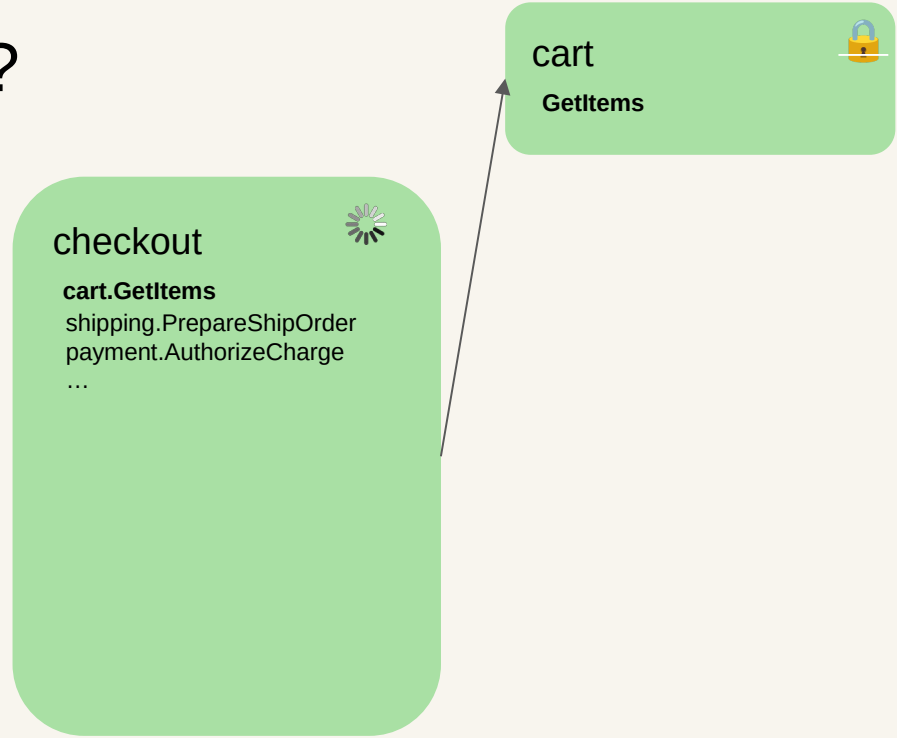
cart.GetItems
shipping.PrepareShipOrder
payment.AuthorizeCharge
…



reboot

How to distribute transactions?

```
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest,
):
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

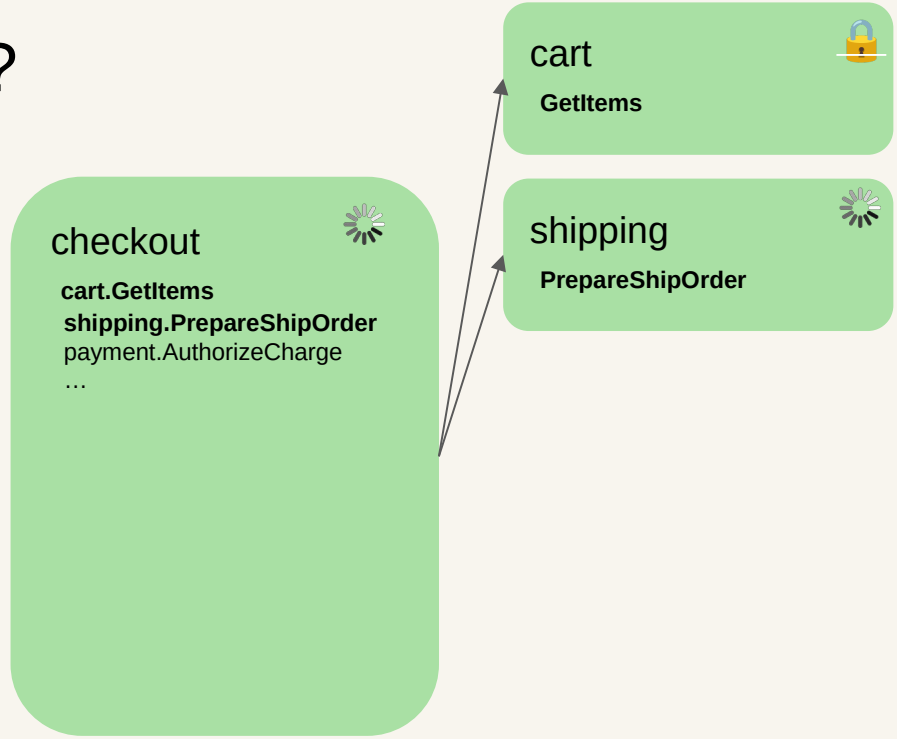




reboot

How to distribute transactions?

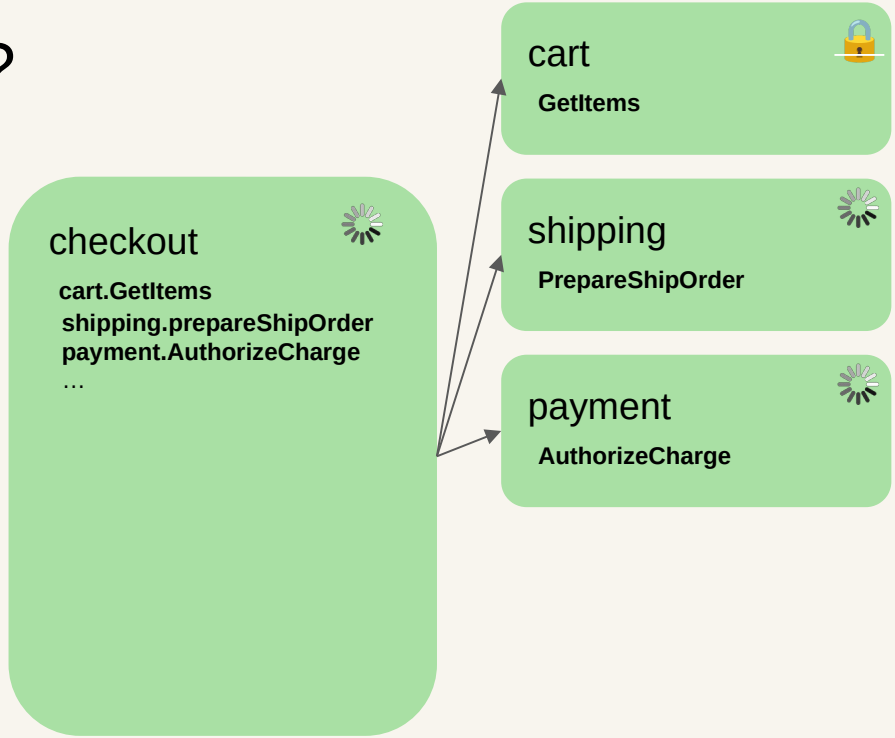
```
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest,
):
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```



reboot

How to distribute transactions?

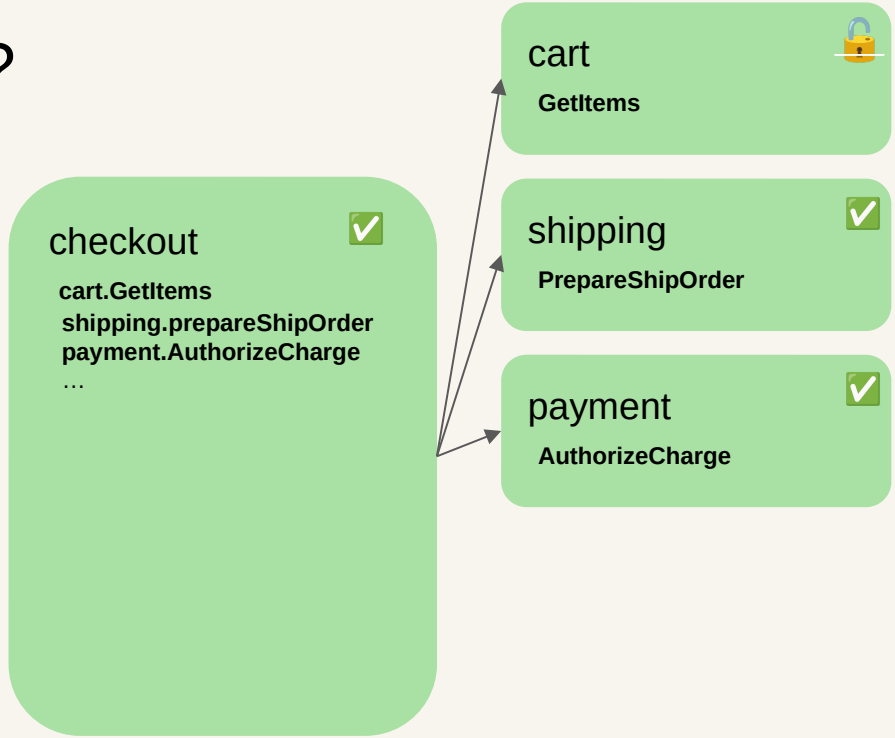
```
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest,
):
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```



reboot

How to distribute transactions?

```
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest,
):
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```



reboot

How to distribute transactions?

> “We believe it is better to have application programmers deal with performance problems due to overuse of transactions as bottlenecks arise, rather than always coding around the lack of transactions.”
>
> - Spanner paper, 2012



Transactions simplify application logic

```python
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest,
):
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```

✅

```python
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest
):
    items = await cart.EmptyCart(...)
    ...
    await asyncio.gather(*[
        shipping.PrepareShipOrder(items, ...),
        payment.AuthorizePayment(items, ...),
        shipping.ShipOrder(items, ...),
        emailer.SendOrderConfirmation(items, ...),
    ])
```

reboot

reboot

From eventually consistent to strongly consistent

```
async def Checkout(
    self,
    context: TransactionContext,
    state: Cart.State,
    request: CheckoutRequest,
):
    items = await cart.GetItems(...)
    ...
    await shipping.PrepareShipOrder(items, ...)
    ...
    await payment.AuthorizeCharge(items, ...)
    ...
    await shipping.ShipOrder(items, ...)
    ...
    await cart.EmptyCart(...)
    ...
    await emailer.SendOrderConfirmation(items, ...)
```



Operators: “It’s microservice. I know this.”

Application developers: “It’s a monolith. I know this.”

Interested in building the next generation of cloud-native abstractions?



Join us!



reboot.dev



Thank you!