Orleans

Microsoft’s Distributed Systems framework for .NET

Reuben Bond

.NET

Agenda

01

The hidden cost of statelessness

02

The stateful solution

03

Microsoft Orleans

04

What's new since Orleans 8.0

05

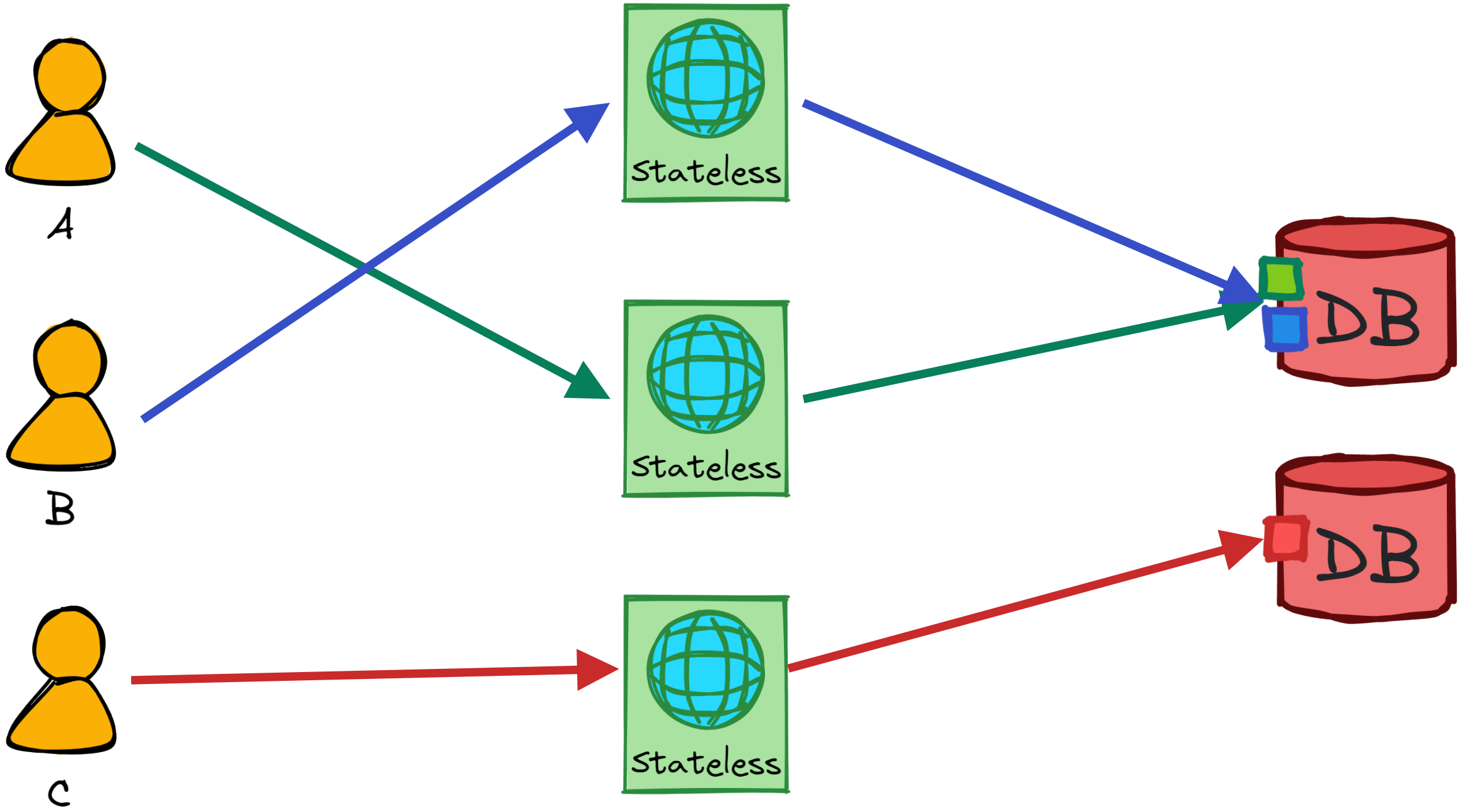
Placement & load balancing

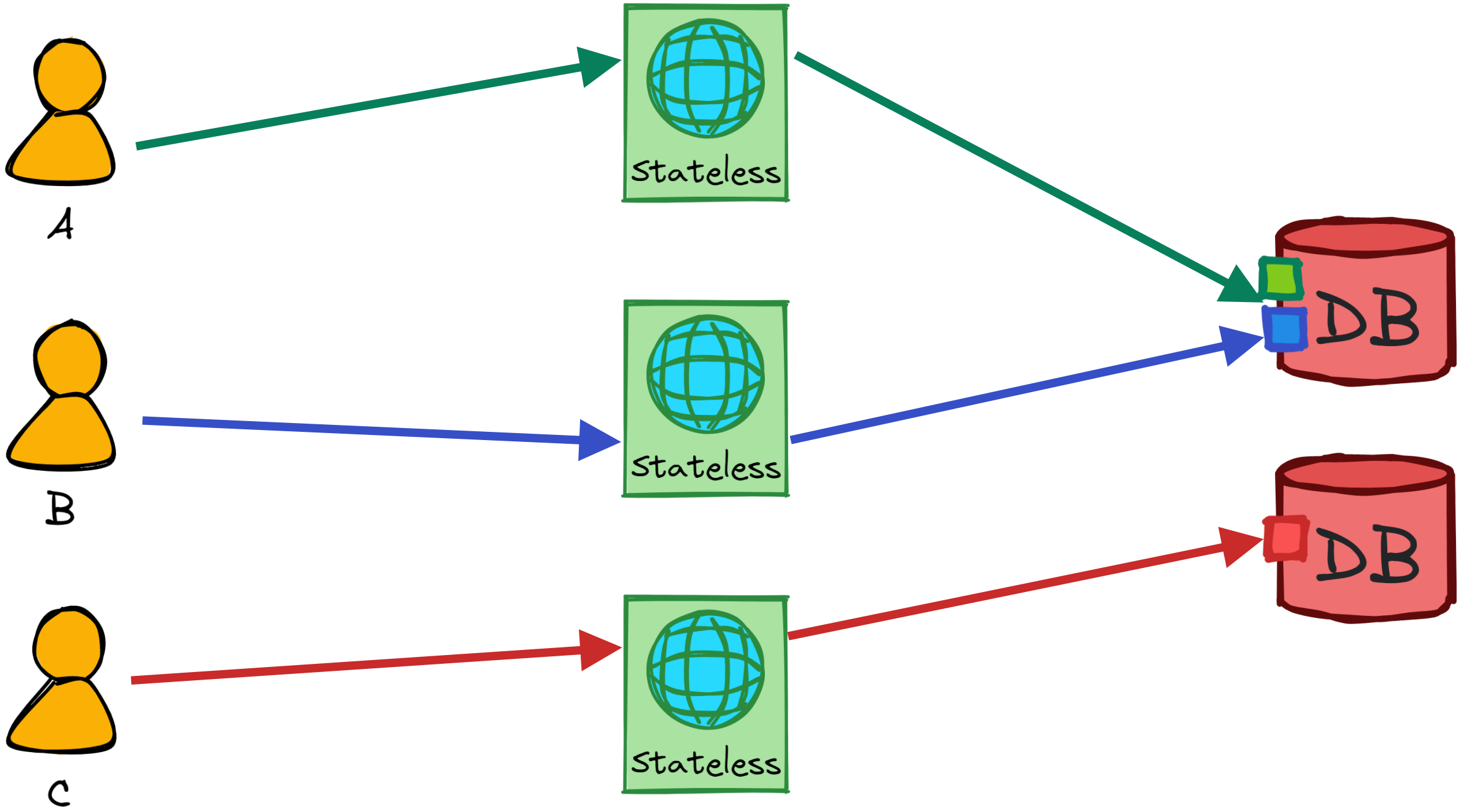
06

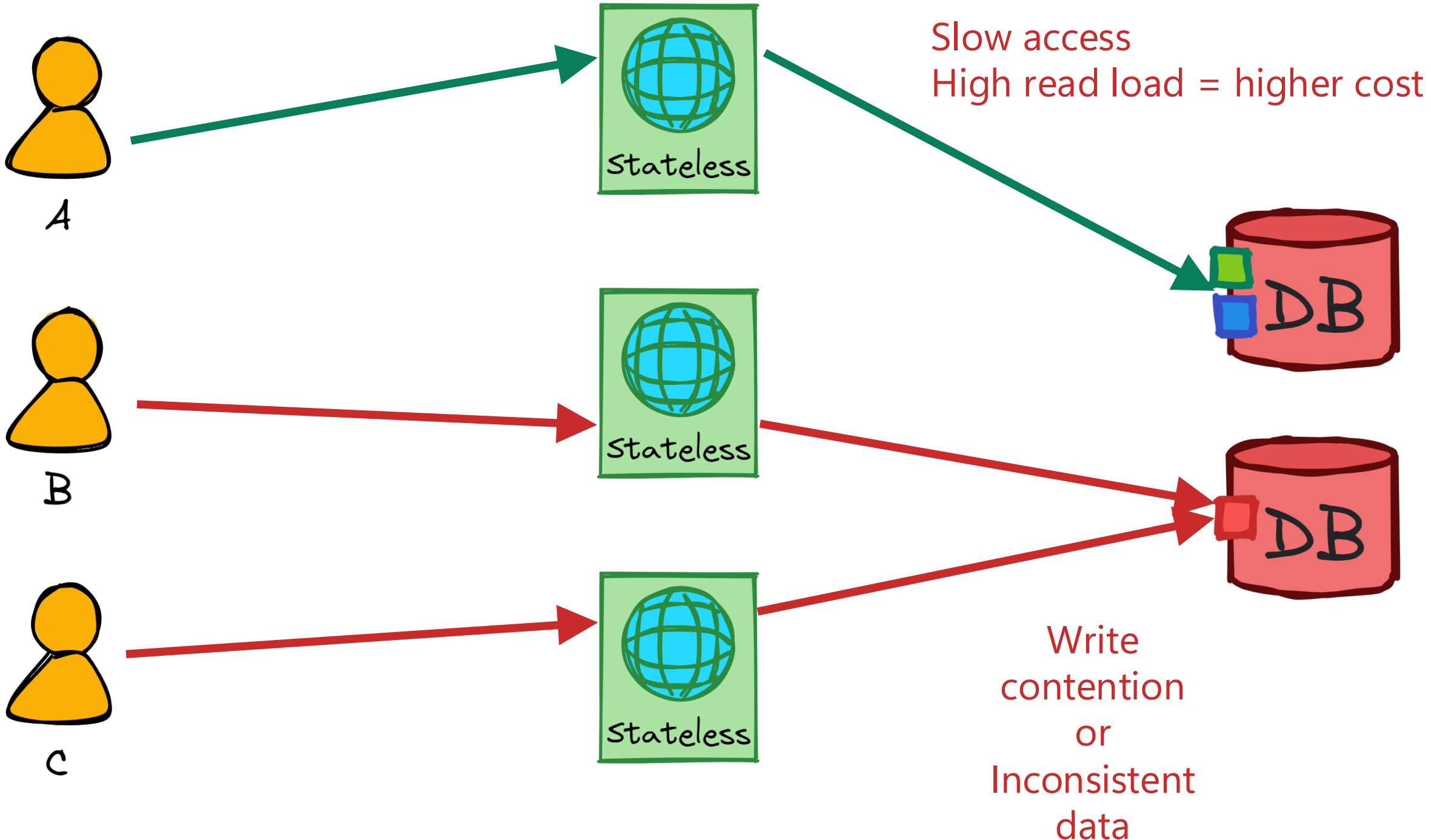
Q&A

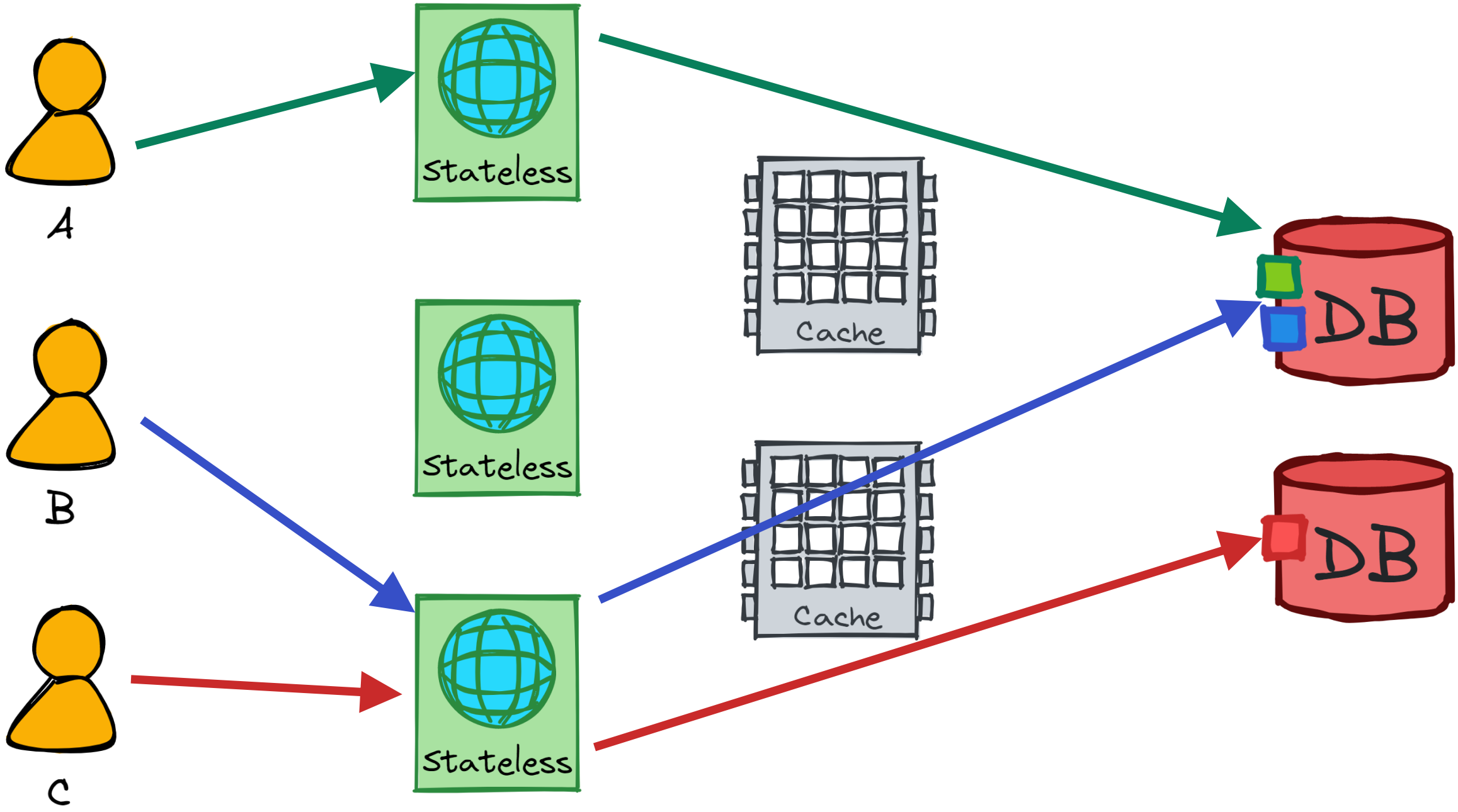
01

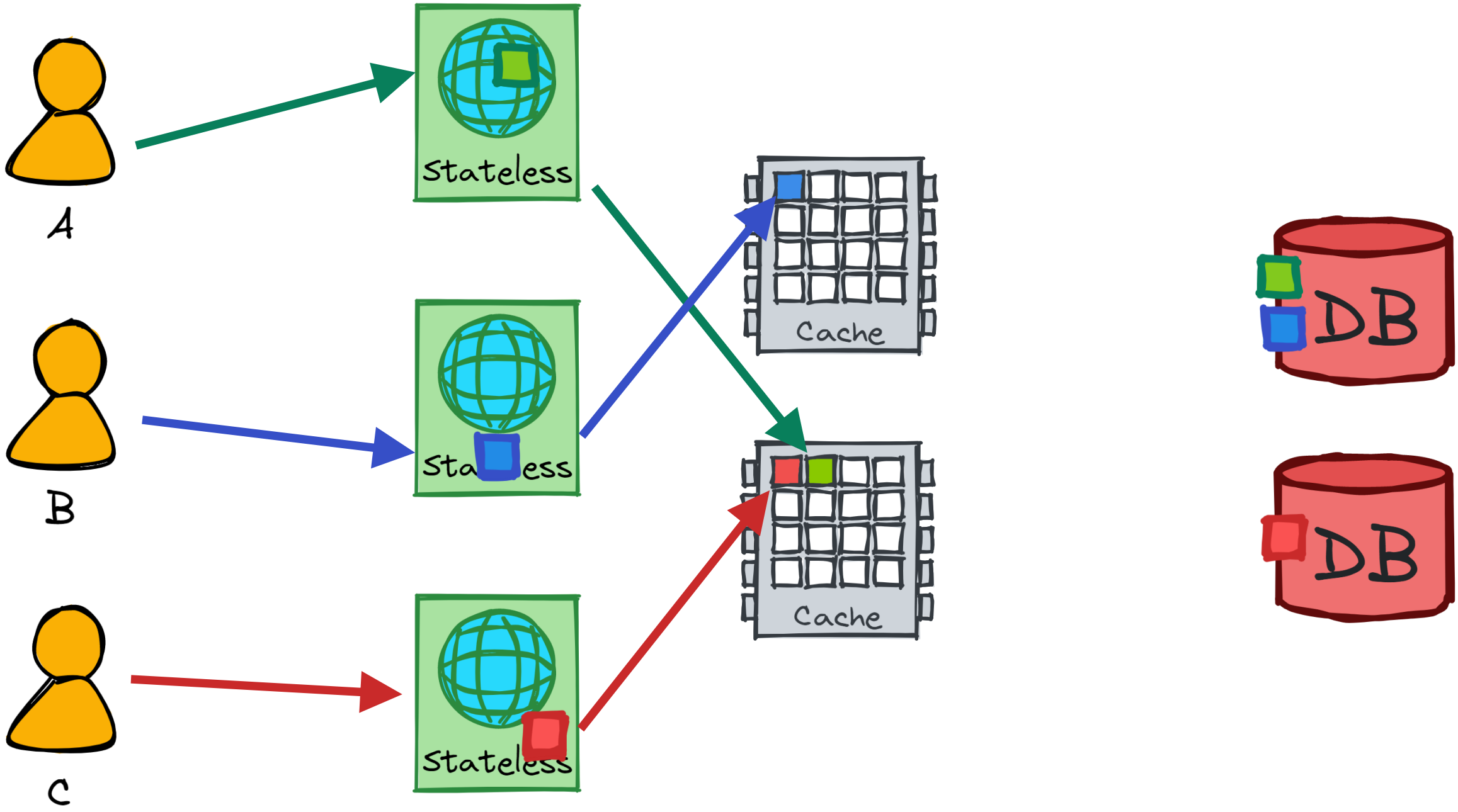
The hidden cost of statelessness

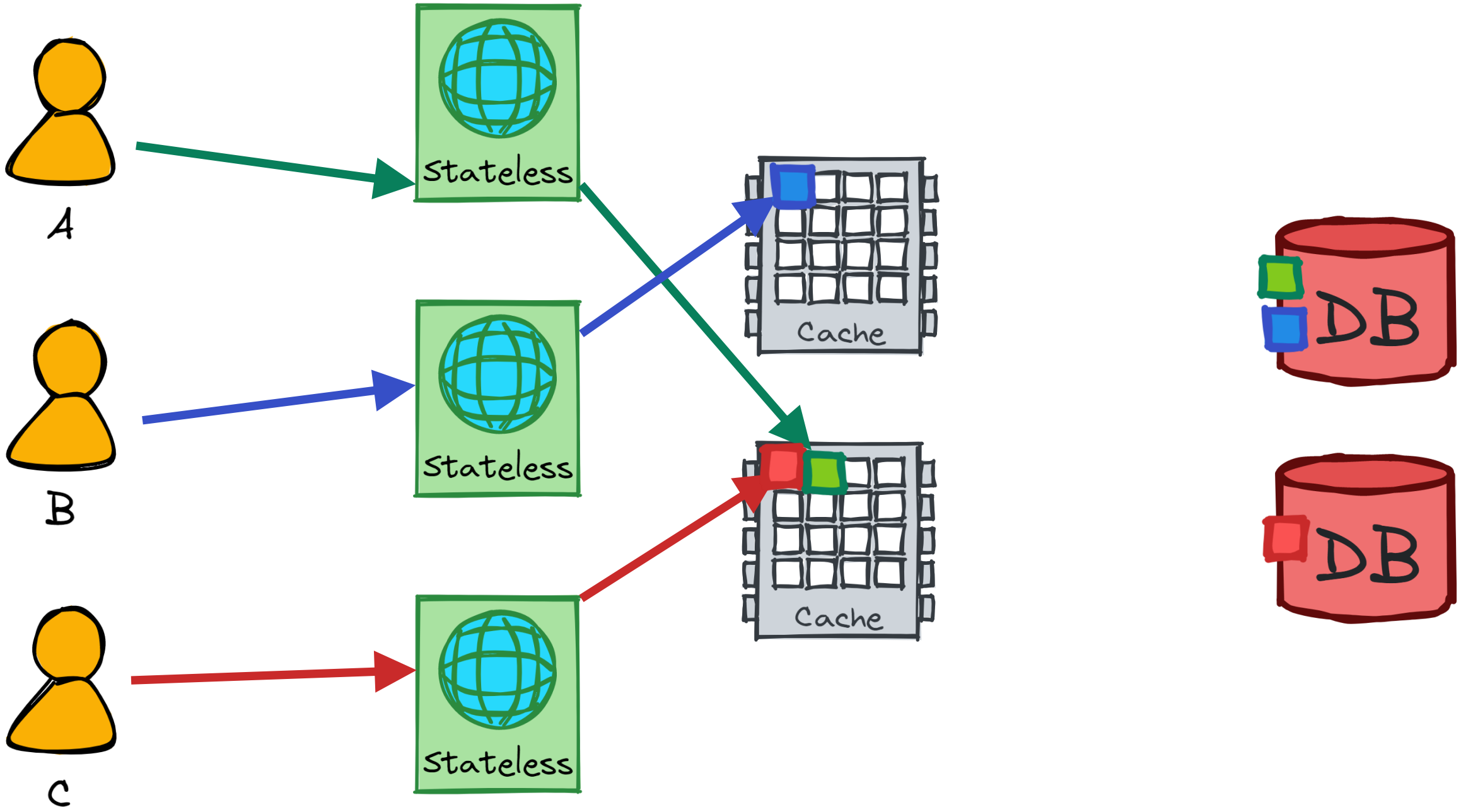

A
B
C
Stateless
Stateless
Stateless
DB
DB

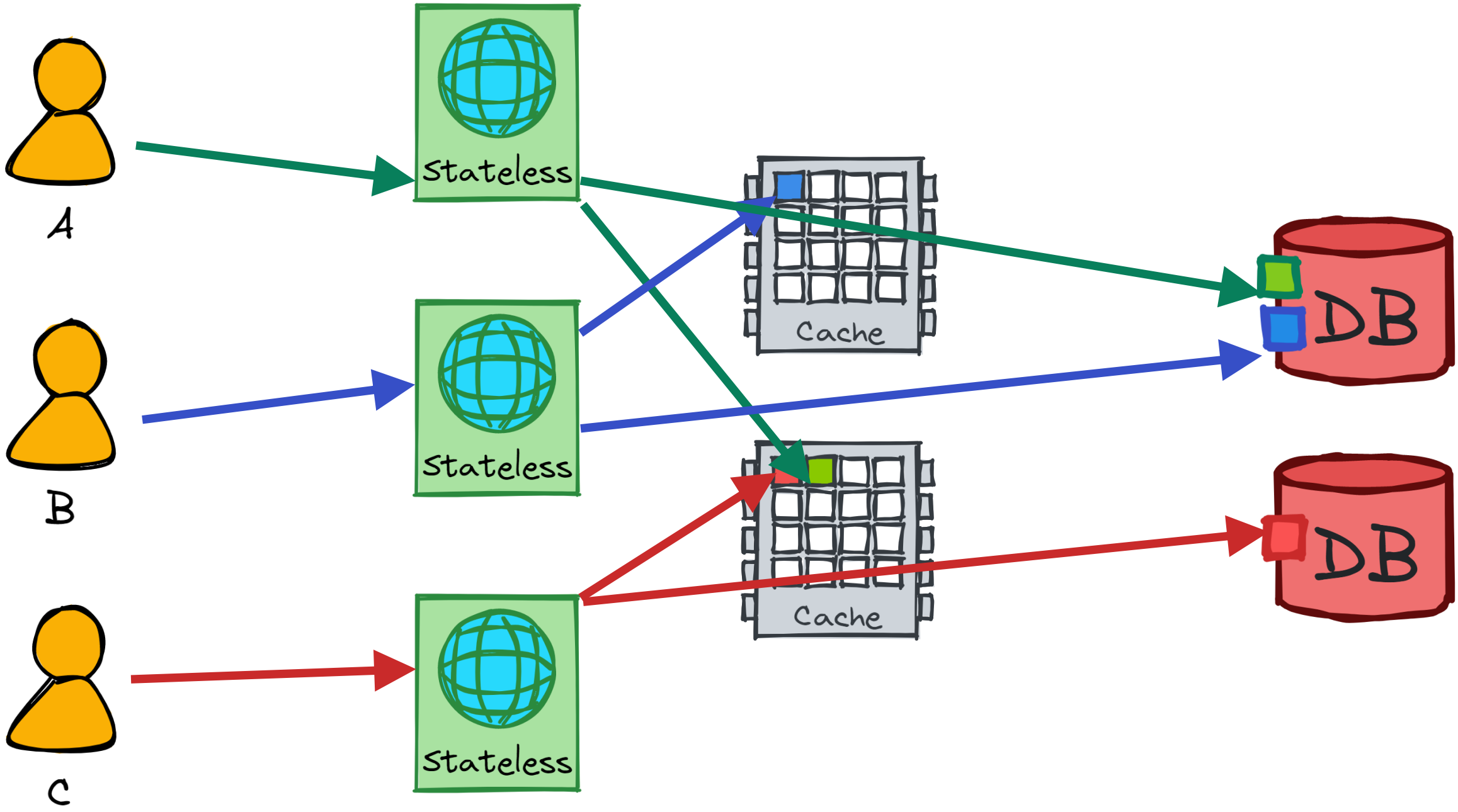

A
B
C
Stateless
Stateless
Stateless
DB
DB

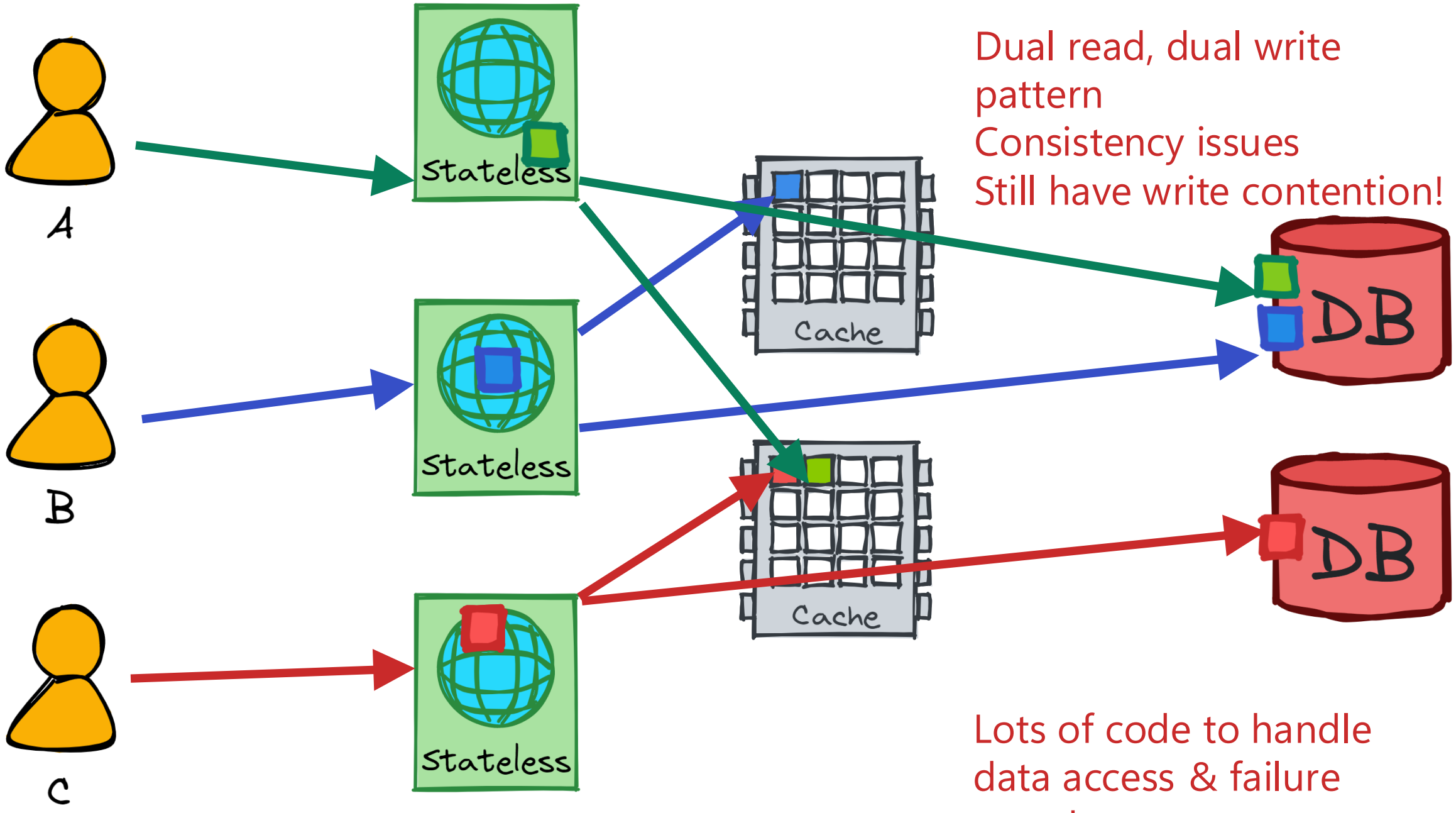

A
B
C
Stateless
Stateless
Stateless
DB
DB
Slow access
High read load = higher cost
Write
contention
or
Inconsistent
data


A
B
C
Stateless
Stateless
Stateless
Cache
Cache
DB
DB


A
B
C
Stateless
Stateless
Stateless
Cache
Cache
DB
DB


A
B
C
Stateless
Stateless
Stateless
Cache
Cache
DB
DB


A
B
C
Stateless
Stateless
Stateless
Cache
Cache
DB
DB



Dual read, dual write pattern
Consistency issues
Still have write contention!

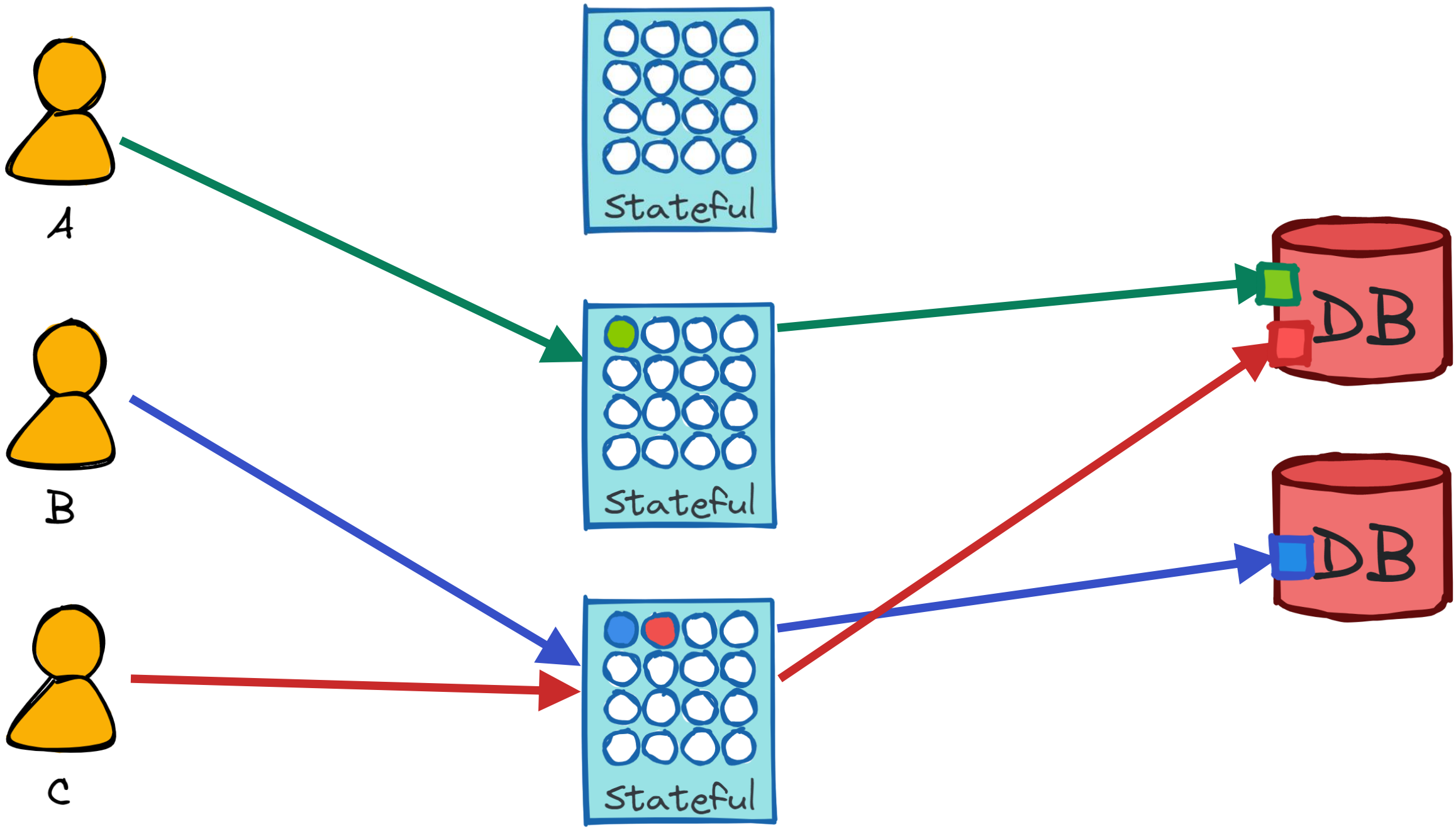
Lots of code to handle data access & failure cases!

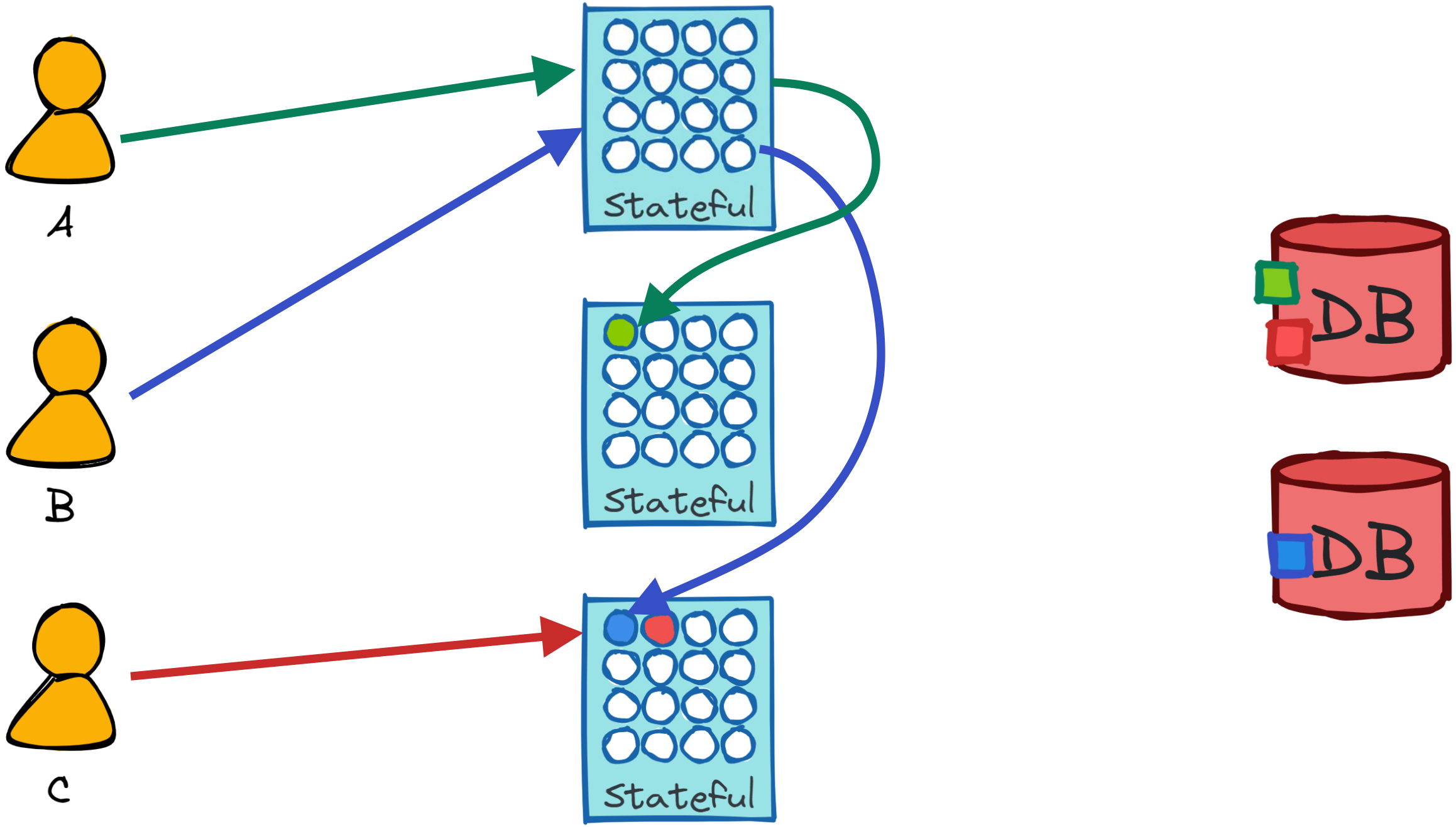
The hidden costs of statelessness

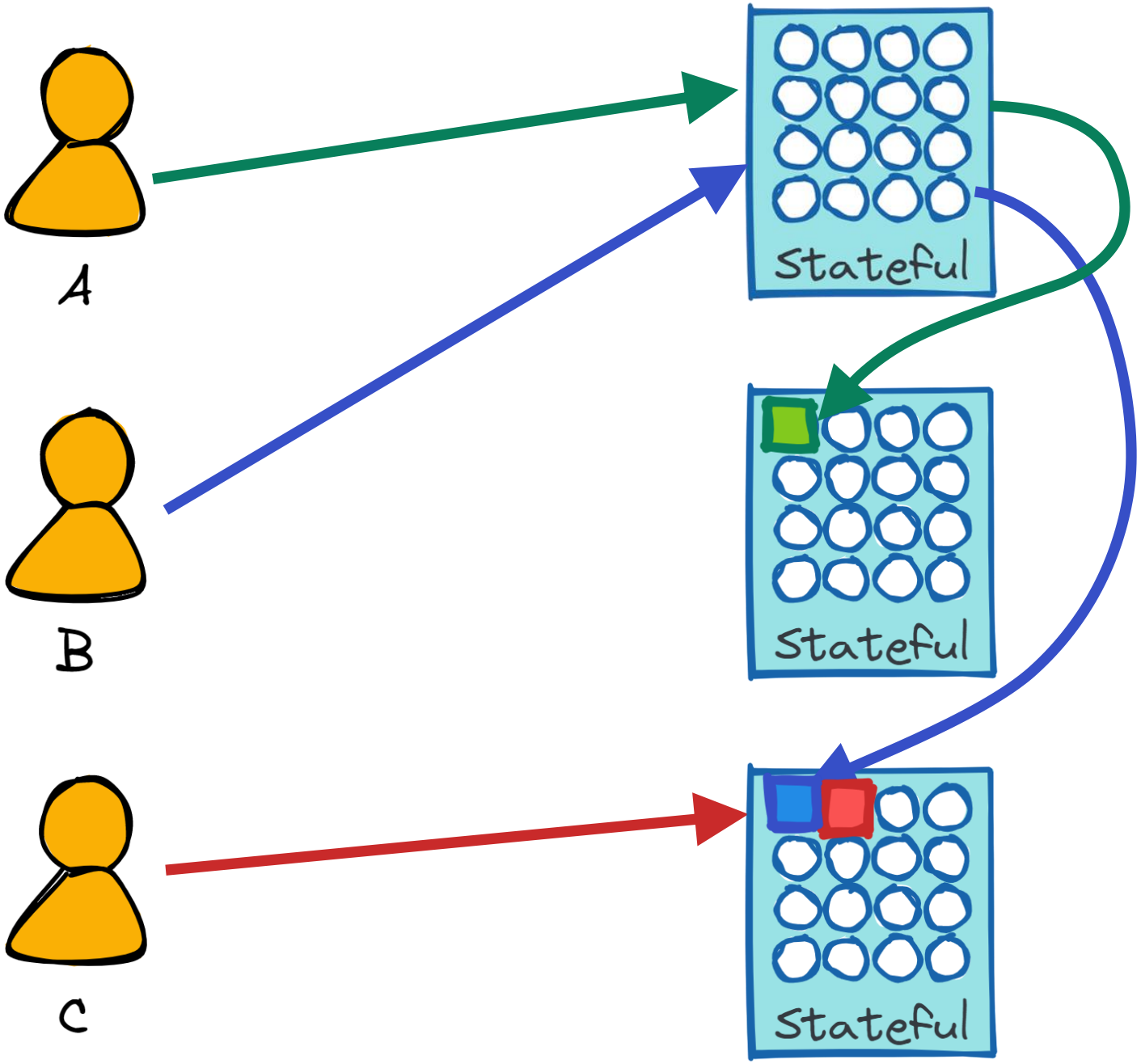
- Higher latency
  - Read, modify, write = 2+ I/O to cache/DB per request
  - Contention retries/backoff
- Lower throughput
  - (De)serialization cost for cache, I/O costs, read amplification, wasted I/O to resolve write contention
- Higher database cost, poor scalability
- Cache infrastructure costs
- Cache coherence complexities
- More code to maintain

02

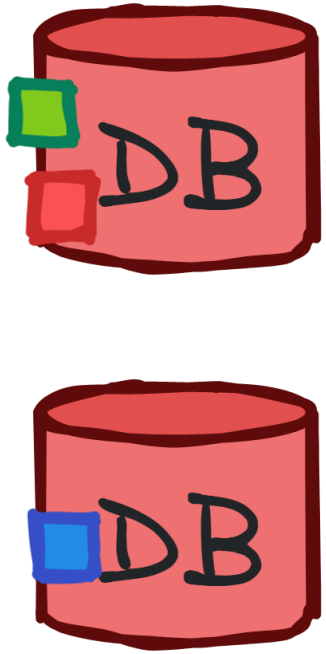
The stateful solution

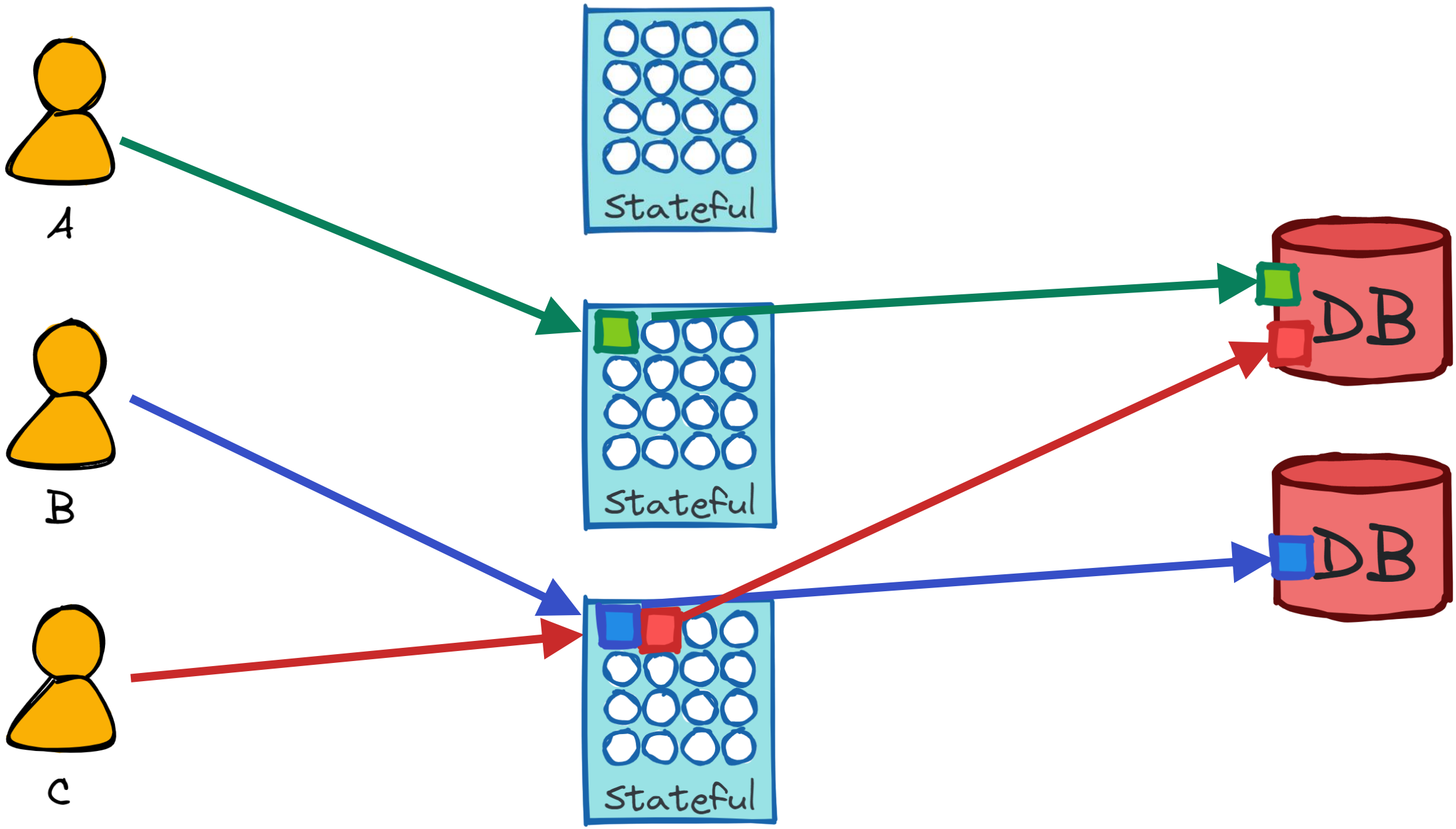

A
B
C
Stateful
Stateful
Stateful
DB
DB

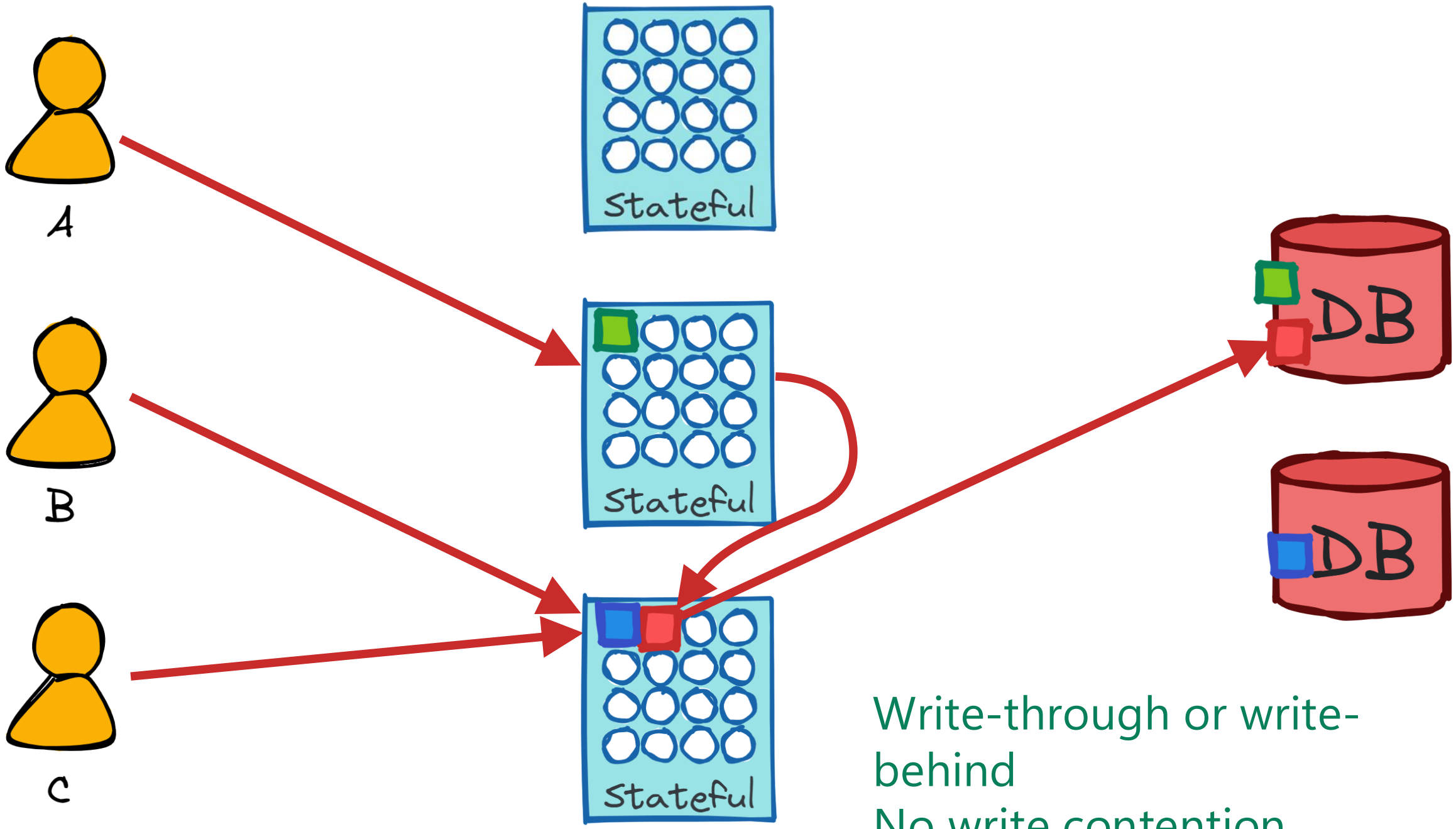

A
B
C
Stateful
Stateful
Stateful
DB
DB



Fast, in-memory reads!




A
B
C
Stateful
Stateful
Stateful
DB
DB



Write-through or write-behind
No write contention

Stateful services save cost, scale better

- Reduce database load
  - Hot/warm data is in memory, cold data is in database
- Eliminate need for separate cache servers
- Eliminate write contention
- Eliminate cache coherence issues
- Scales elastically
  - Load is balanced across your servers
  - More servers = more capacity
  - No partition keys

03

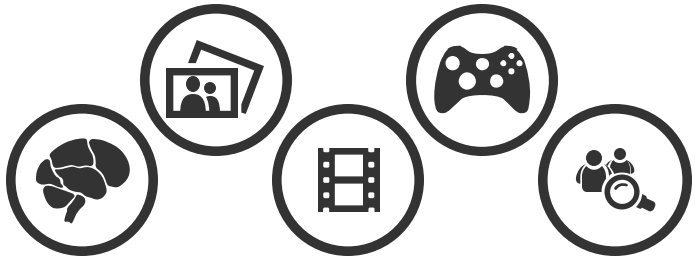
Microsoft Orleans

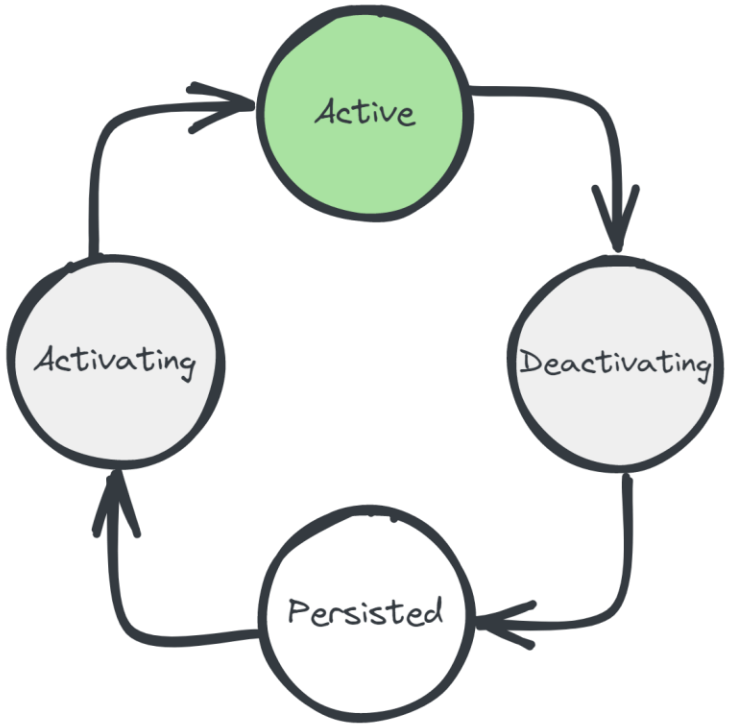
Microsoft Orleans

- .NET library for (stateful) cloud applications
- Core primitive: grain (next slide)
- Runtime handles lifecycle, fault tolerance, load balancing, messaging, serialization, etc
- OSS, maintained by .NET team alongside Aspire, gRPC, ASP.NET
- Adopted by many Microsoft product groups & external companies, underpinning production services for 10+ yrs
- Continually evolving & improving (later in this talk)

Grains

- Objects
  - Stateful, optionally persisted
  - Single-threaded by default
- Identified by type + key
  - Eg: `user/fred88`
- Loaded on-demand
  - Runtime managed lifecycle like virtual memory
- Communicate by RPC
  - Low-ceremony C# interface calls





```
public interface IUserGrain : IGrainWithStringKey
{
  Task SendMessage(IUserGrain from, string msg);

  Task<List<Message>> GetMessages();
}
```

```
public class UserGrain : Grain, IUserGrain
{
  IPersistentState<List<Message>> _msgs;

  public UserGrain([PersistentState("msgs")] IPersistentState<List<Message>> state)
    => _msgs = state;

  public Task<List<Message>> GetMessages() => Task.FromResult(_msgs.State);

  public async Task SendMessage(IUserGrain sender, string body)
  {
    _msgs.State.Add(new Message { Sender = sender, Body = body });
    await _msgs.WriteStateAsync();
  }
}
```

State is loaded by the runtime.

Methods are single-threaded. No locks necessary.

```
// Get grain references. Synchronous & cheap, like a URL
IUserGrain me = client.GetGrain<IUserGrain>("reuben");
IUserGrain friend = client.GetGrain<IUserGrain>("aditya");

// Call them
await friend.SendMessage(me, "hey :)");
var messages = await me.GetMessages();
```

Other Features

Persistence

Plugins for Azure Storage, Cosmos DB, SQL, DynamoDB, Redis, MongoDB, …

Pub/Sub

Decouple producers & consumers. Backed by Azure Event Hubs, Amazon SQS, GCP, in-memory

Scheduling

Ephemeral timers, persistent reminders for scheduling future event

Push notifications

Grains can send events (RPC calls) directly to clients. Eg, push notifications, streams

Deploy anywhere

Azure Container Apps, Azure App Service, Kubernetes, AWS, GCP, bare metal/VMs, etc

Custom placement & directory

Customize how the runtime decides where new grains are instantiated & how it finds them. “Move computation to data”

Call filters

Wrap incoming & outgoing requests & responses for logging, tracing, error handling

.NET integration

Aspire, Observability, Dependency Injection, Hosting, Configuration

04

What’s new since 8.0

What’s new since 8.0

Resource-aware placement

Places *new* grain activations on the least loaded hosts, with a locality bias

Locality-aware repartitioning

Minimizes network traffic by migrating *related* grains into the same hosts

Rebalancing

Rebalances already-active grains after scale-out, rolling upgrade, or changes in workload patterns

Strong-consistency directory

Replaces the in-built grain directory with a strong-consistency, resilient directory

Aspire integration

Streamlined inner-loop and “zero to cloud” experience using .NET Aspire

Much more

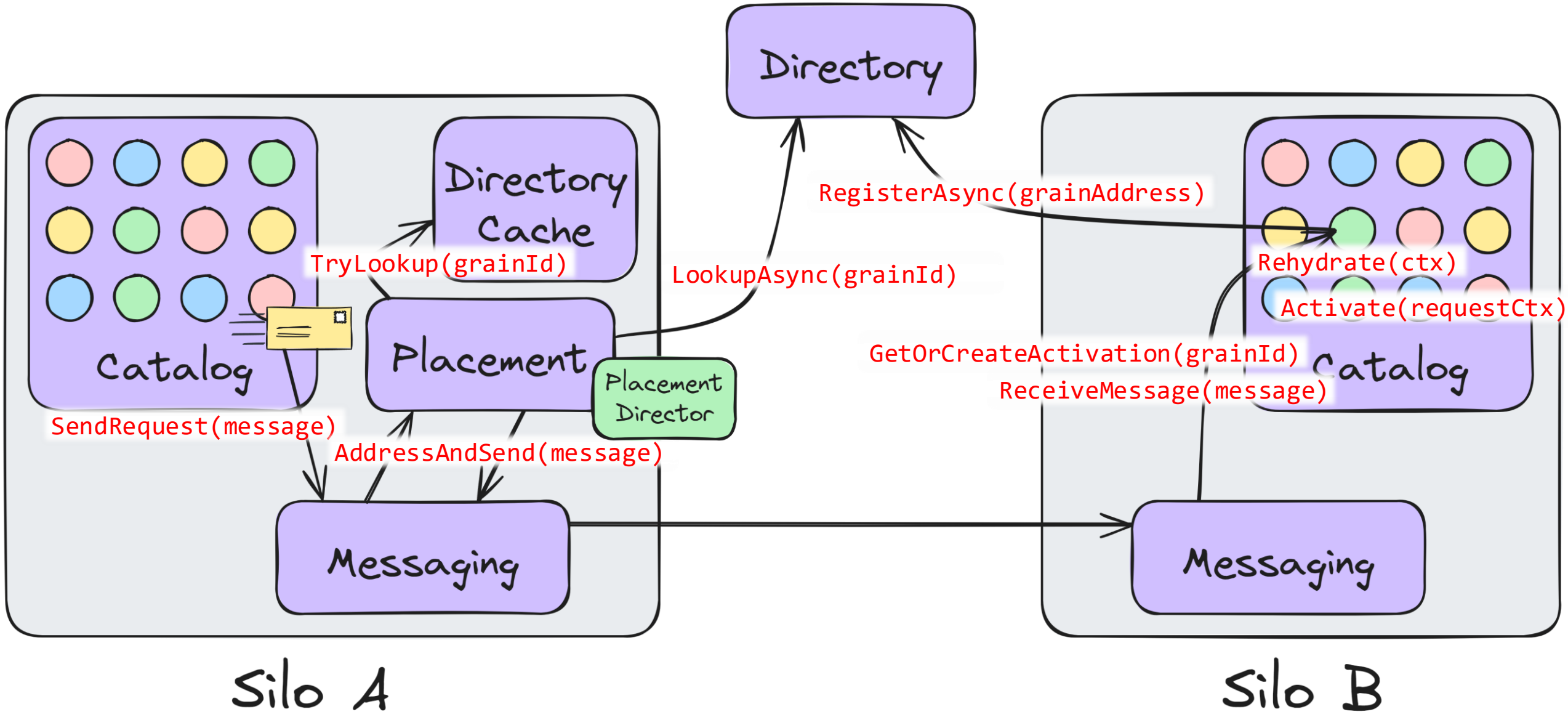
Reliability & performance improvements, improved grain timers, support for MessagePack, Cassandra

05

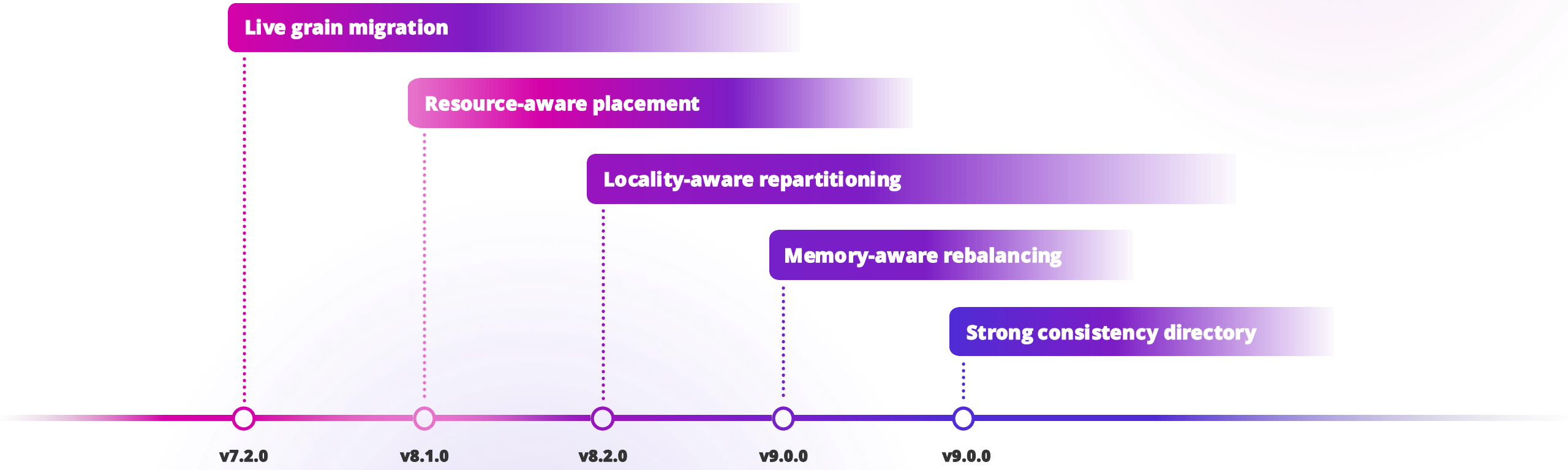
Placement & Load Balancing

Message routing overview

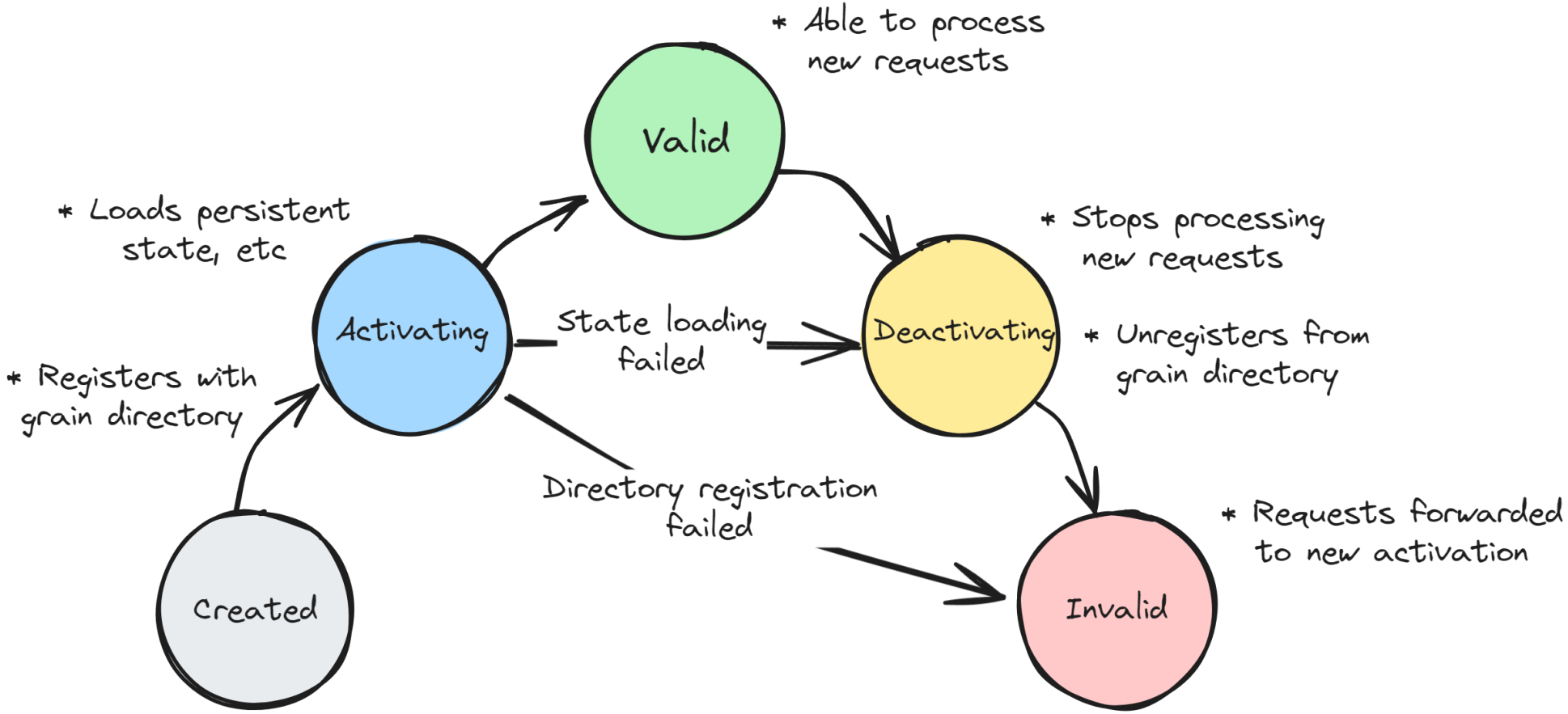
- Ask local directory **cache** which silo hosts the target grain
- If not found, ask **directory**, update cache
- If not found, run **placement** to pick a compatible silo
- Route message to that silo
- Silo **activates** target grain and **registers** it in the directory


Directory
Directory Cache
Catalog
Placement
Placement Director
Messaging
TryLookup(grainId)
LookupAsync(grainId)
RegisterAsync(grainAddress)
SendRequest(message)
AddressAndSend(message)
Rehydrate(ctx)
Activate(requestCtx)
GetOrCreateActivation(grainId)
ReceiveMessage(message)
Catalog
Messaging
Silo A
Silo B

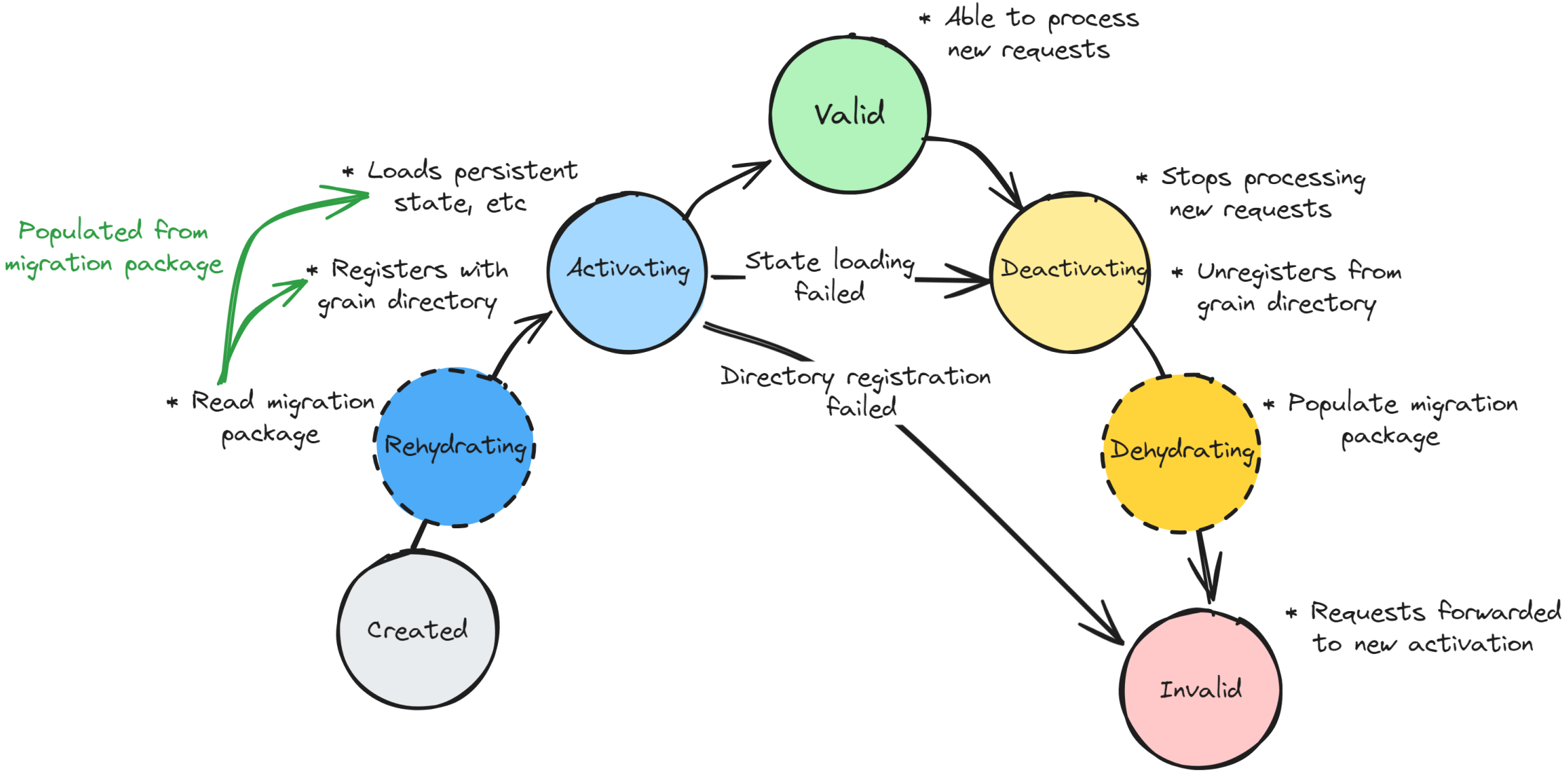
Placement & load balancing enhancements



Grain activation lifecycle

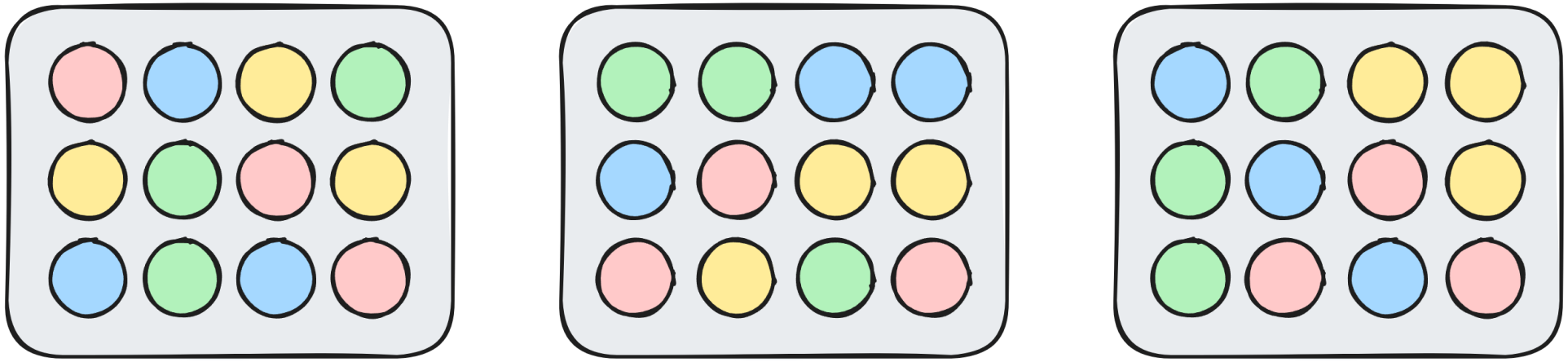


... with live grain migration



Resource-aware placement

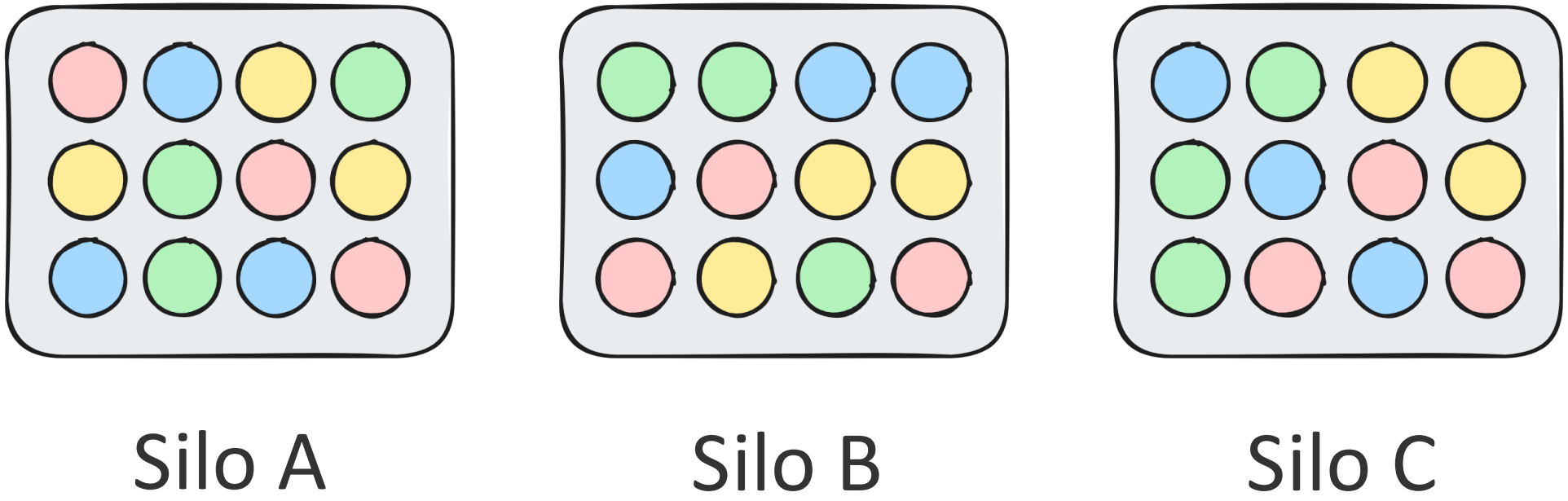
- Places activations on the least-loaded host
- Considers CPU usage, free/total memory, activation count
- Locality bias places related activations together



Silo A Silo B Silo C

Locality-aware repartitioning

- Migrates *related* activations to the same host to reduce network calls, maintaining balance
- Improves throughput (**30-110%**), reduces overhead & latency



Host 1 [1] 0

[4] 0 Host 4

Host 2 [2] 0

[3] 0 Host 3

[0] 0

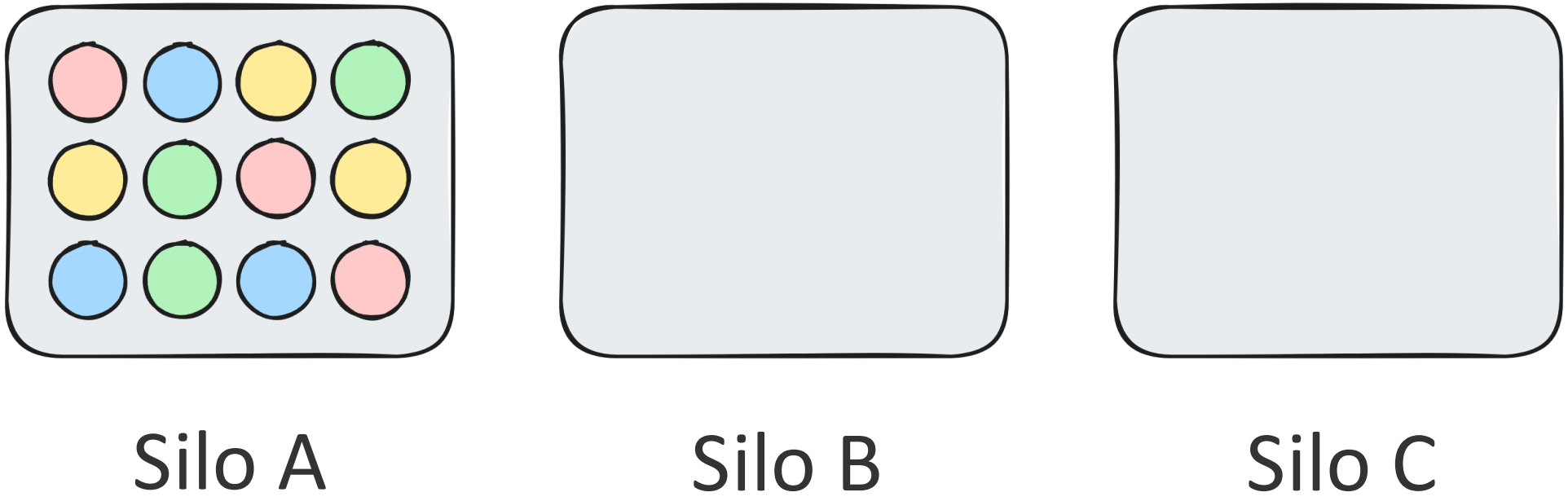
Host 0

Red line = network call

Adding load

Memory-aware rebalancing

- Rebalance load as capacity increases
- In-memory state is transferred – no hit to database



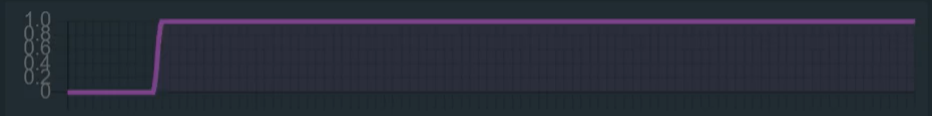
OrleansDashboard

v.8.2.0.0 Source

- Overview
- Grains
- Grain Details
- Silos
- Reminders
- Log Stream

Silos

ACTIVE SILOS
1



Silo Health

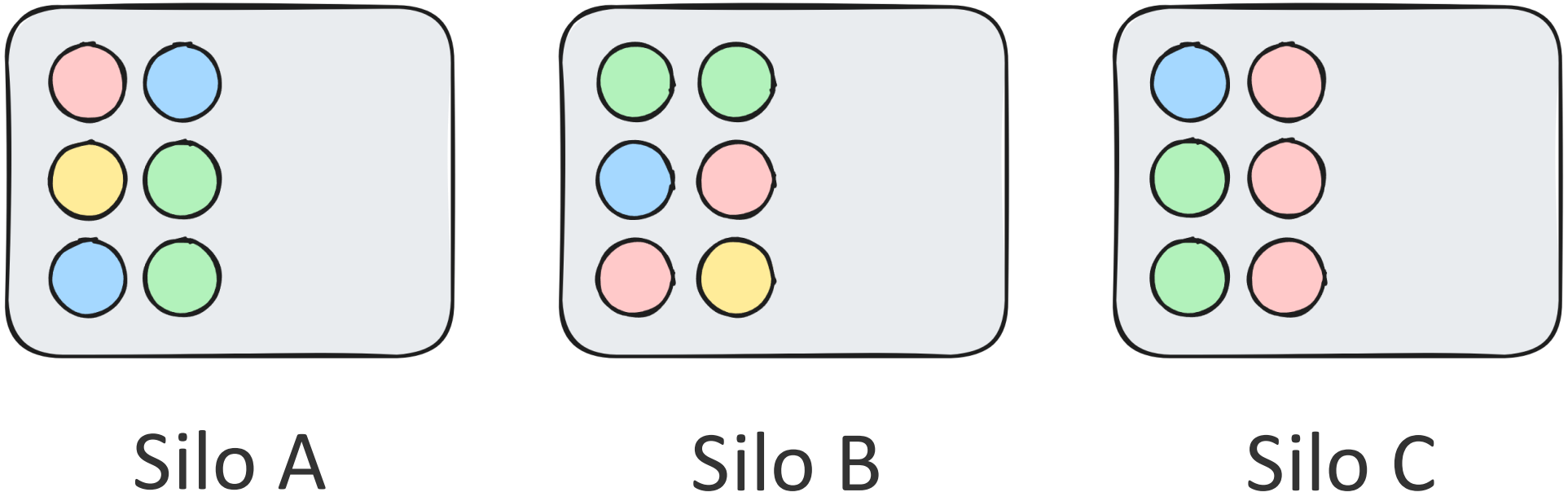
10.244.8.235:11111@88480289	Active	Silo_cb215	chaos-silo-859458c798-zlv6j	up for 2 minutes, 41 seconds	**99736** activations

Silo Map

	Fault Zone 0
Update Zone 0	10.244.8.235:11111@88480289 Active

Migrate on shutdown

- Move valuable grains to surviving hosts during shutdown by calling `MigrateOnIdle()` from `OnDeactivateAsync(…)`

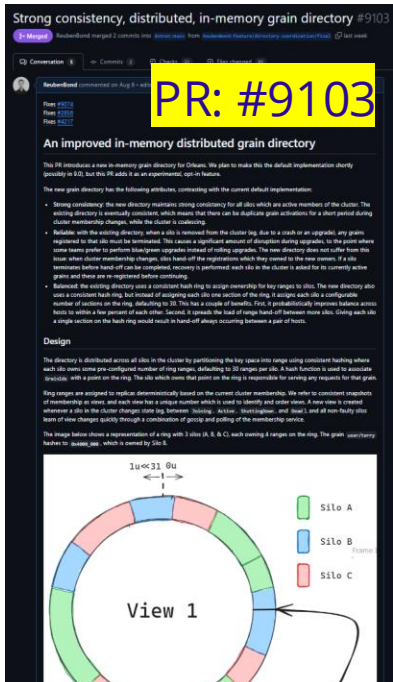


```
public sealed class MyRedGrain : Grain, IMyRedGrain
{
    public override Task OnDeactivateAsync(DeactivationReason reason, CancellationToken cancellationToken)
    {
        if (reason.ReasonCode == DeactivationReasonCode.ShuttingDown)
        {
            MigrateOnIdle();
        }

        return base.OnDeactivateAsync(reason, cancellationToken);
    }
}
```

New in-cluster grain directory

- Original directory:
  - Eventual consistency, temporarily allows duplicate activations
  - Distribution prone to hot spots
  - Loses registrations during cluster churn, magnifying impact
- New directory:
  - Strong consistency
  - Well balanced
  - Reliable – no registration/activation loss
- Result
  - Smooth rolling upgrades, scale out/in



Aspire integration

- Use Aspire to define infrastructure
- Aspire generates configuration for Orleans
- Orleans consumes configuration for various providers
  - Storage
  - Clustering
  - Reminders
  - Streams
- Aspire uses infra emulators/containers for local dev/test
  - Optionally provisions & configures real Azure infra
- Deploy to Azure Container Apps:
  - `azd init && azd up`

AppHost project

```
var builder = DistributedApplication.CreateBuilder(args);

var redis = builder.AddRedis("orleans-redis");

var orleans = builder.AddOrleans("cluster")
    .WithClustering(redis);

builder.AddProject<Projects.DashboardToy_Frontend>("frontend")
    .WithReference(orleans)
    .WaitFor(redis)
    .WithReplicas(5);

builder.Build().Run();
```

Frontend project

```
var builder = WebApplication.CreateBuilder(args);
builder.AddKeyedRedisClient("orleans-redis");
builder.UseOrleans();
var app = builder.Build();
```

Community Spotlight

Special thanks to Ledjon Behluli

Contributions & collaborations:

- Resource-aware Placement
- Locality-aware Repartitioning
- Memory-aware Rebalancing
- ...and more!

https://github.com/ledjon-behluli

Thank you, Ledjon!

github.com/dotnet/orleans

aka.ms/orleans/discord

Thank you