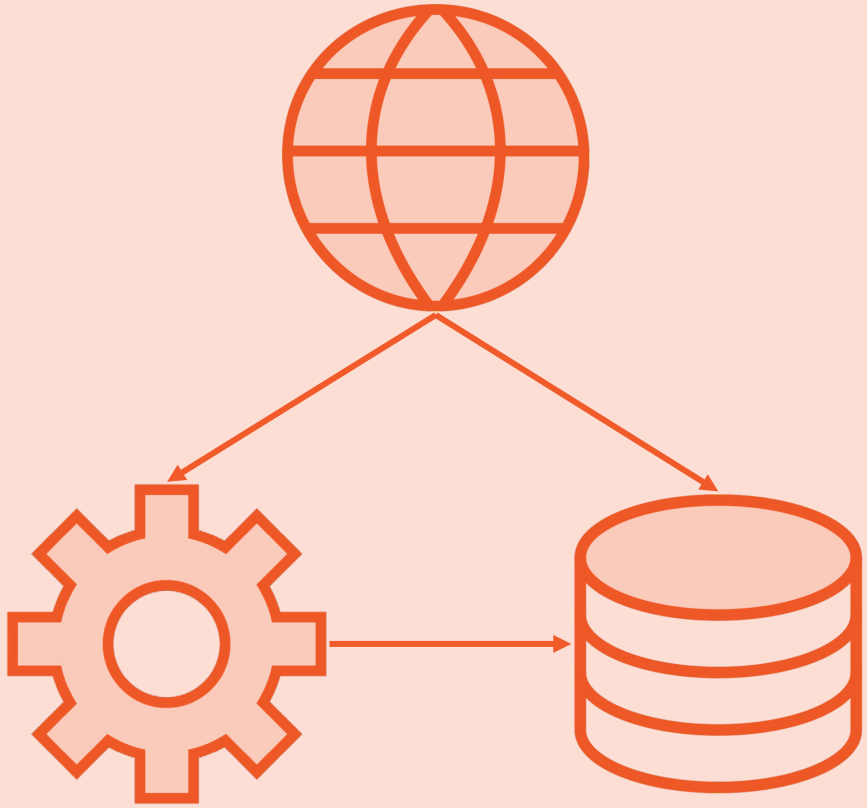
Incorporating Site Reliability Engineering in Your System Design

Architecting Systems for Reliability

Elton Stoneman
Consultant & Trainer

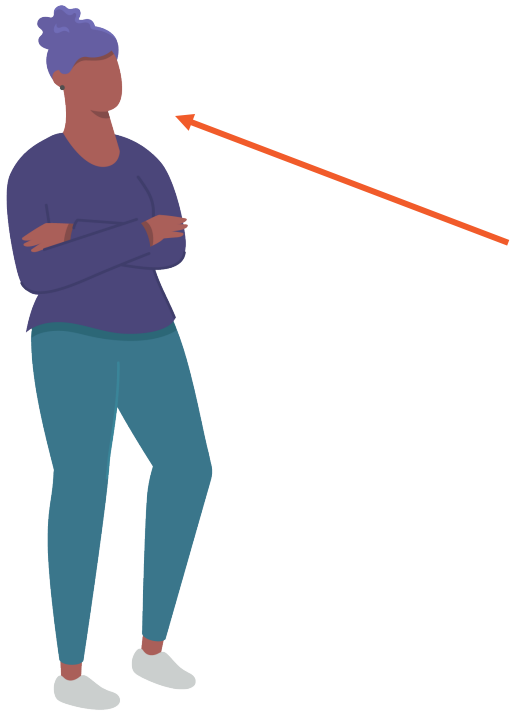
@EltonStoneman blog.sixeyed.com

Introducing the Course

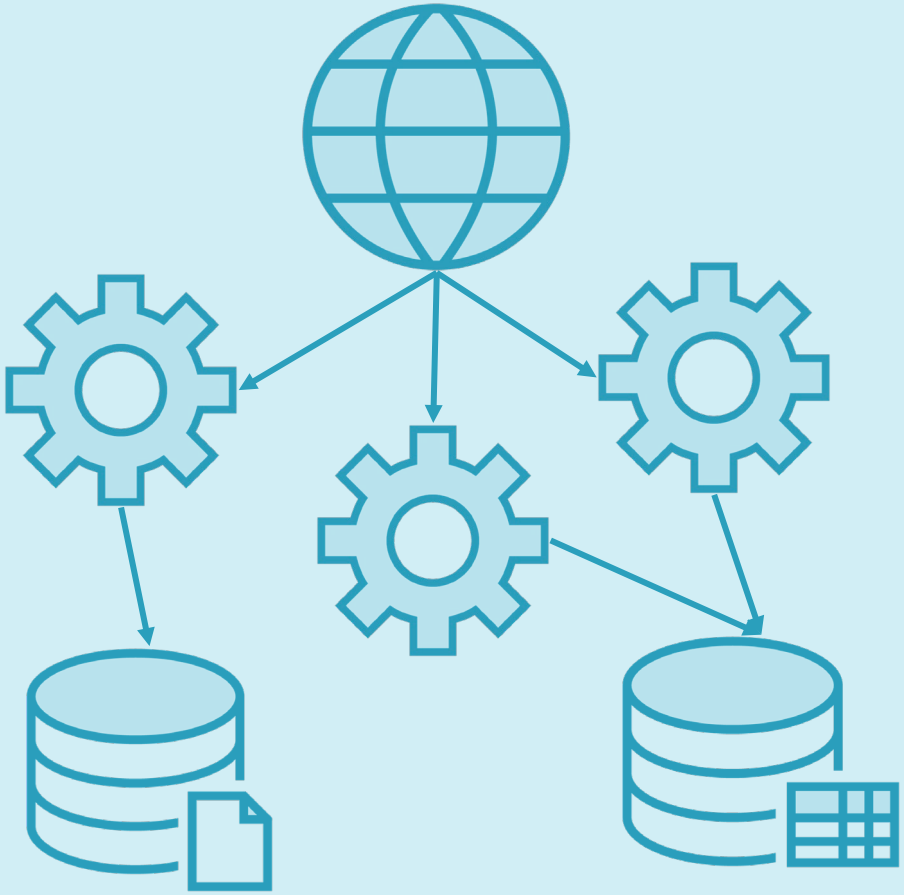


GLOBOTICKET

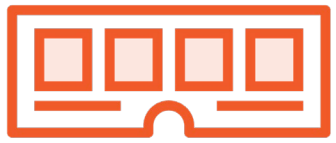
A Globomantics Company



Amila



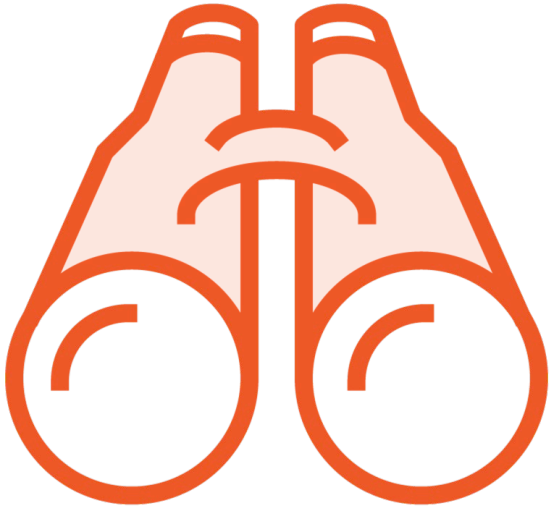
SRE Backlog



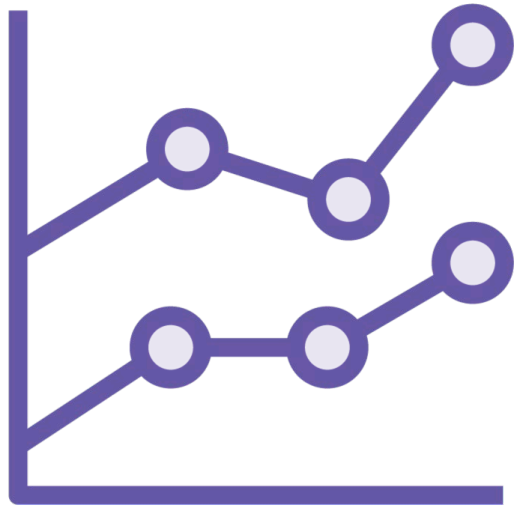
	Effort	*Team*	*Priority*
Safe restarts	**3 days**	**Dev**	**1**
Health checks	**2 days**	**SRE**	**2**
Caching	**5 days**	**SRE**	**3**

Course Layout

Architecture

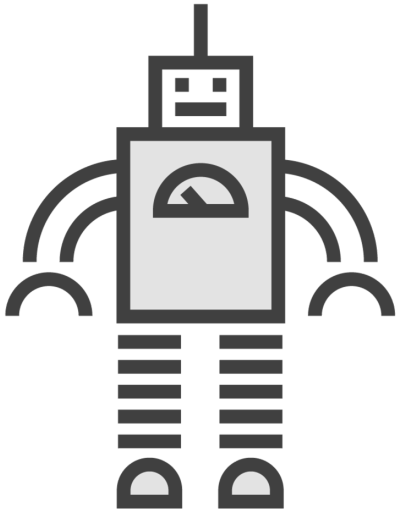


Observability



Improvement

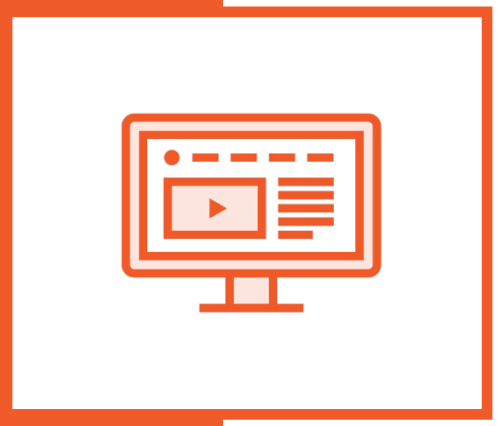
Principles of SRE



Eliminating toil

Managing risk

Handling failure



Fundamentals of SRE

Site Reliability Engineering (SRE): The Big Picture

Elton Stoneman

System Design and SRE

Design for mitigation

- **Incident step #1**
- **Remove investigation pressure**

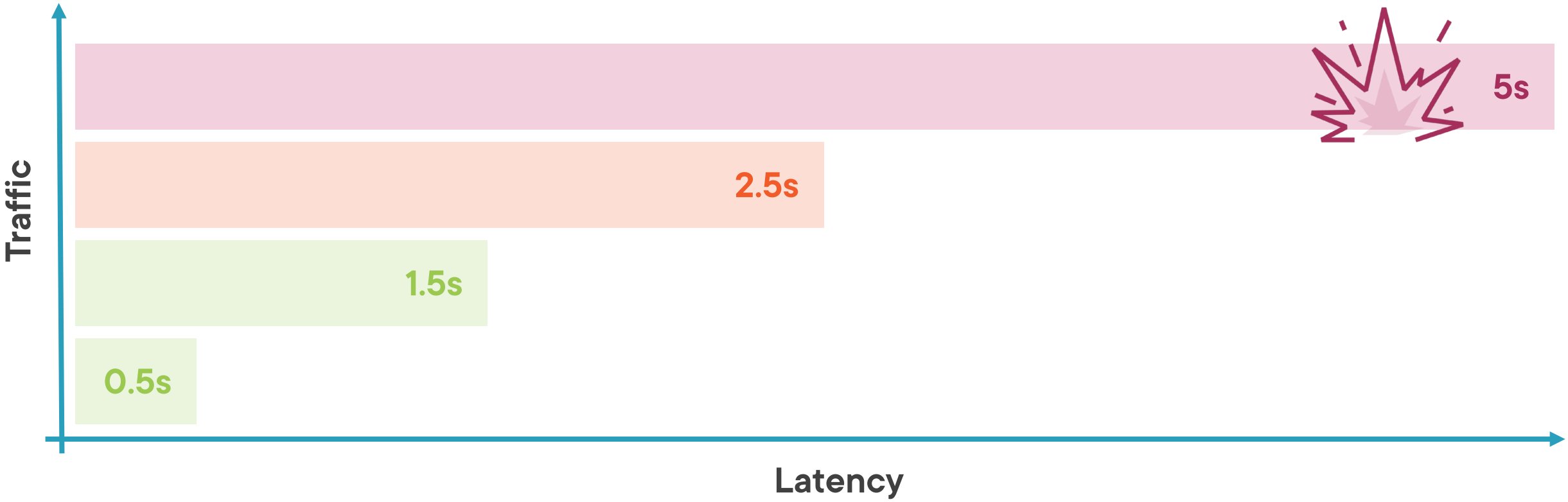
Design for failure scenarios

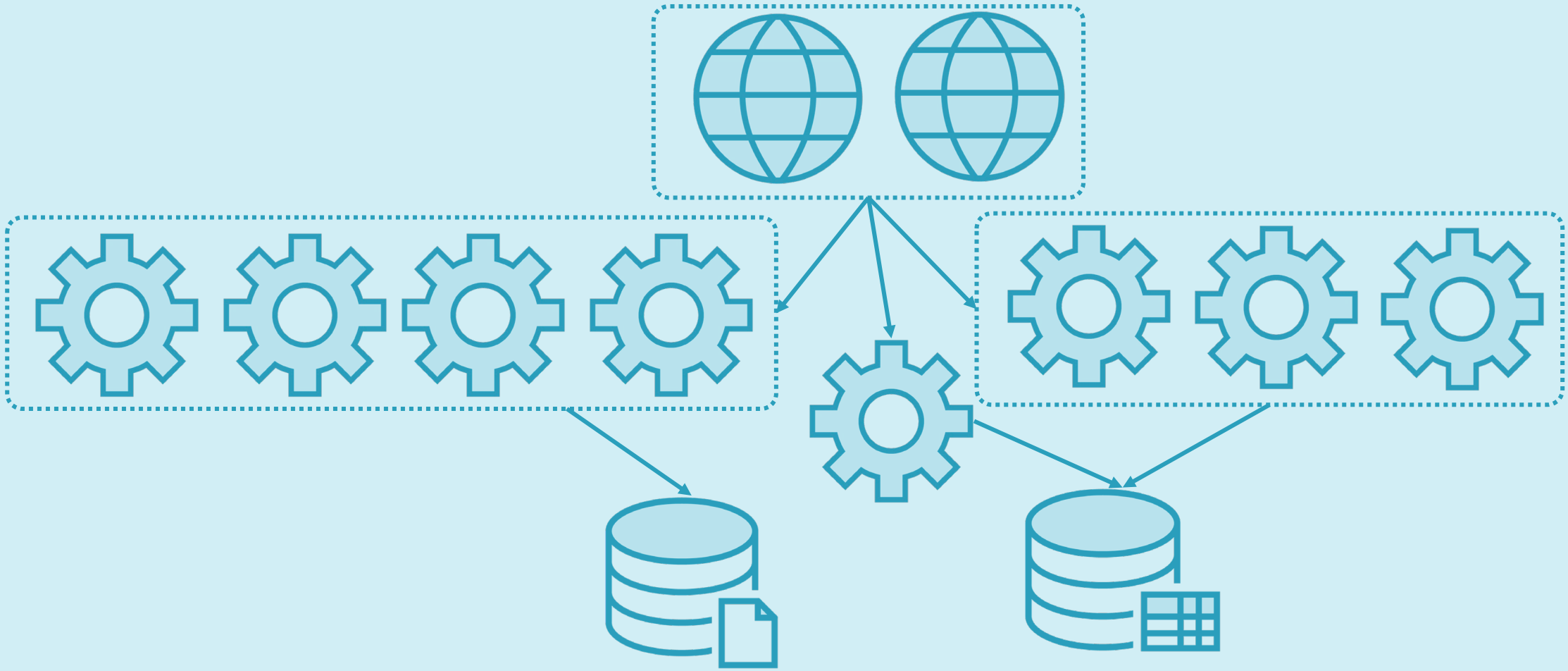
- **Scaling under high load**
- **Managing overload**
- **Introducing degradation**

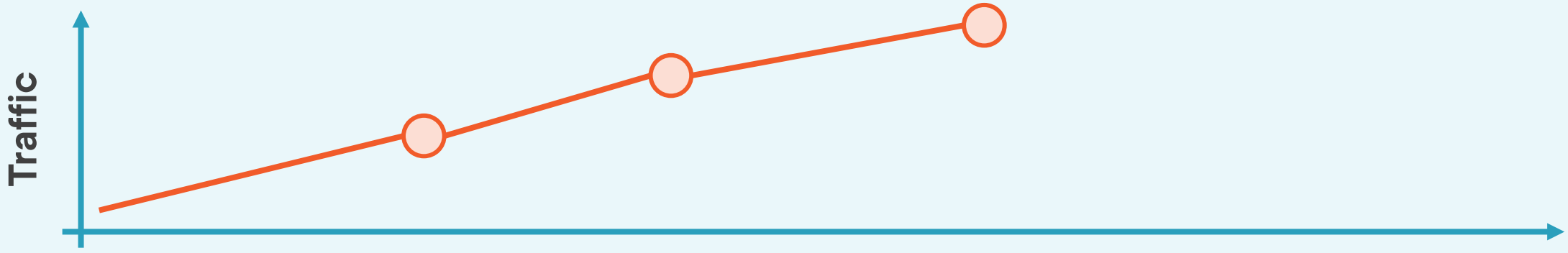
Designing for Scale and Load-balancing

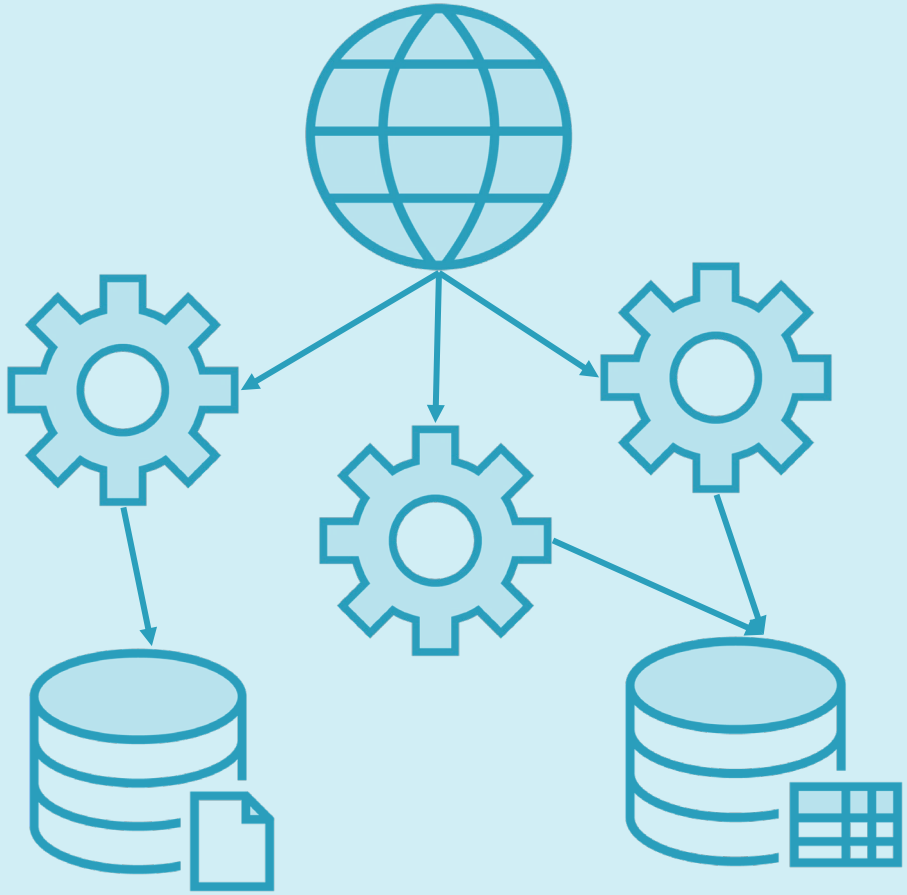
Service Level Objective

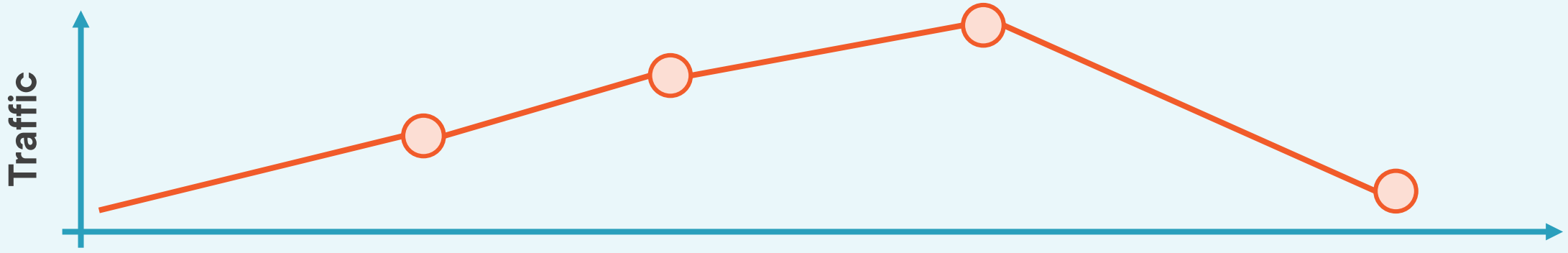
Response time **99.9% of requests within 2 seconds**



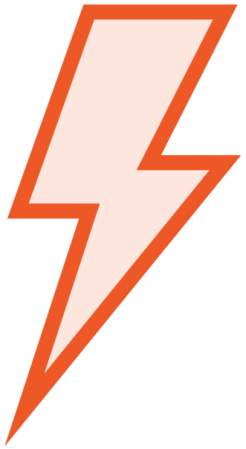



Traffic




Traffic

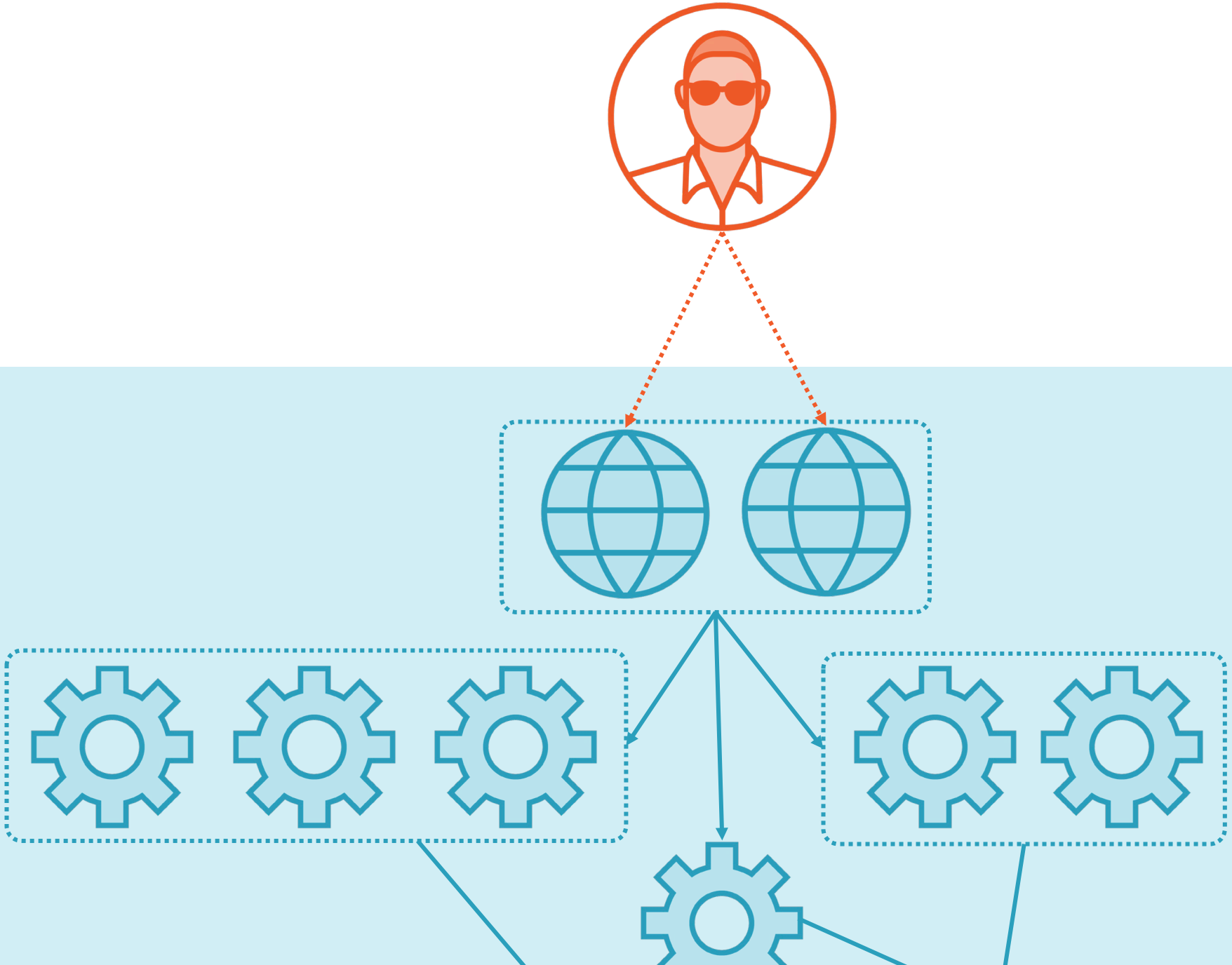
Automatic Scaling

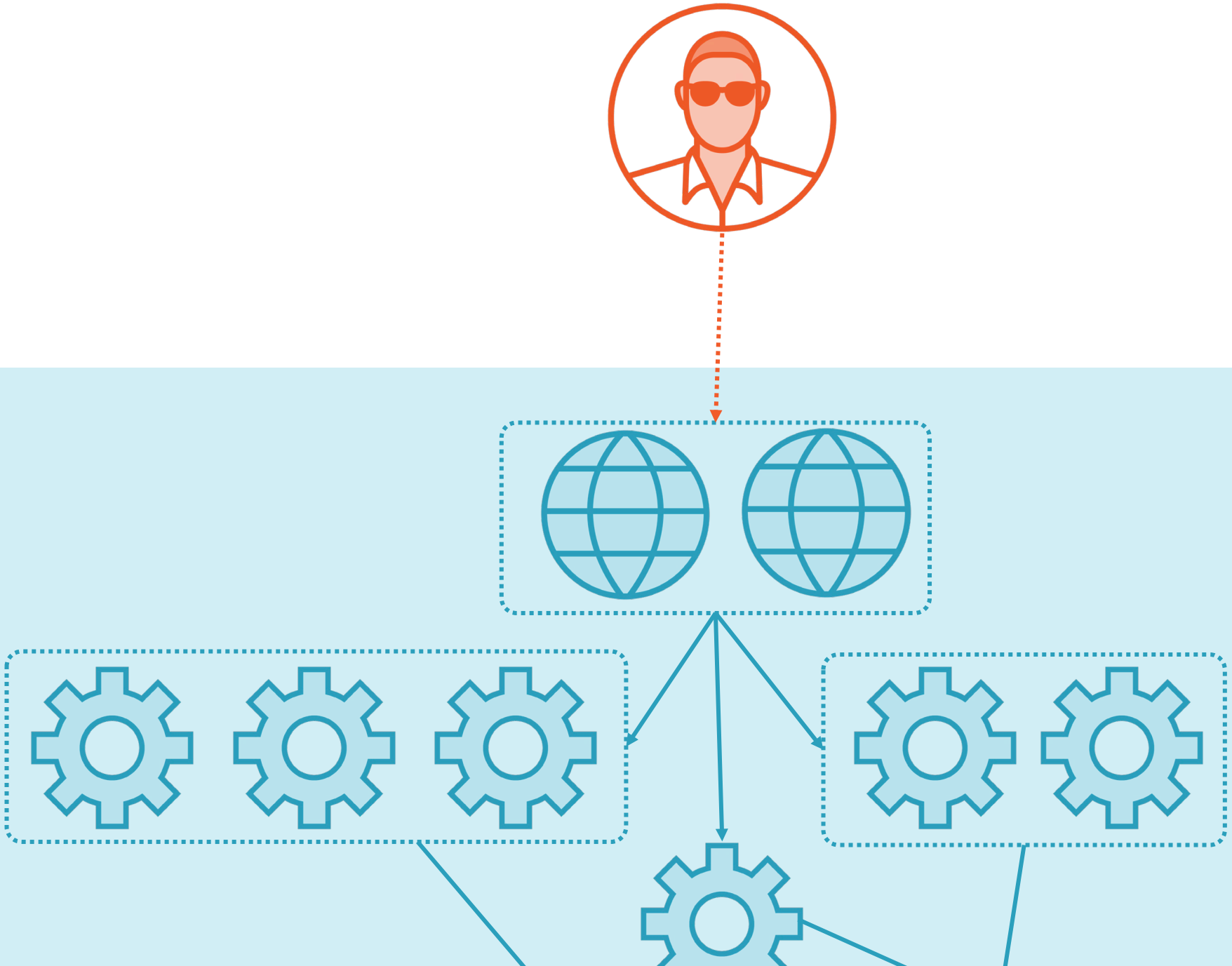


Scale event trigger

Instance startup time

New instance count

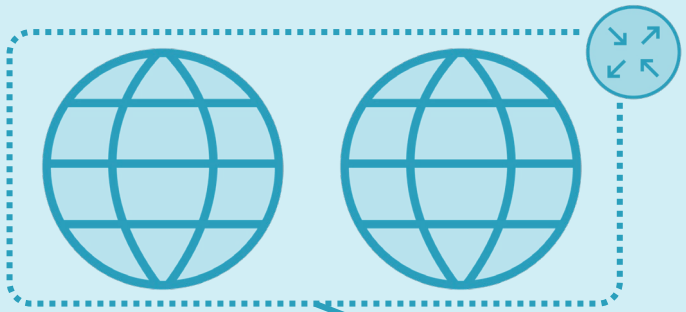


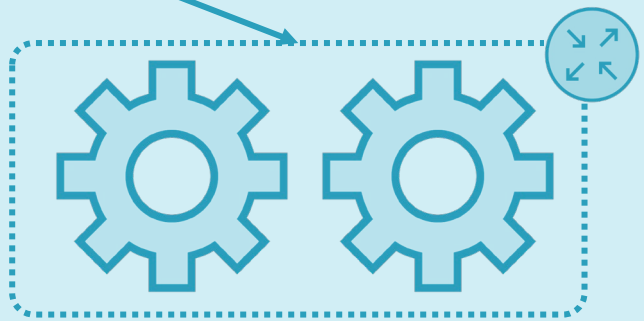


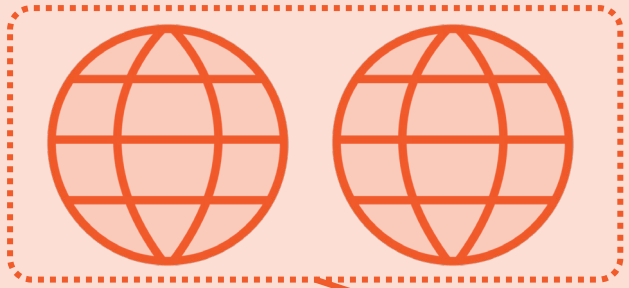
- DNS
- VIP + Network LB
- Software LB

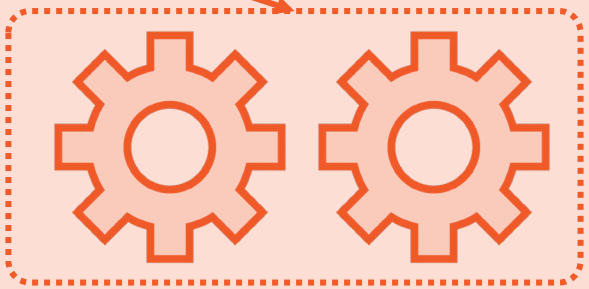
app.com

51.221.18.272





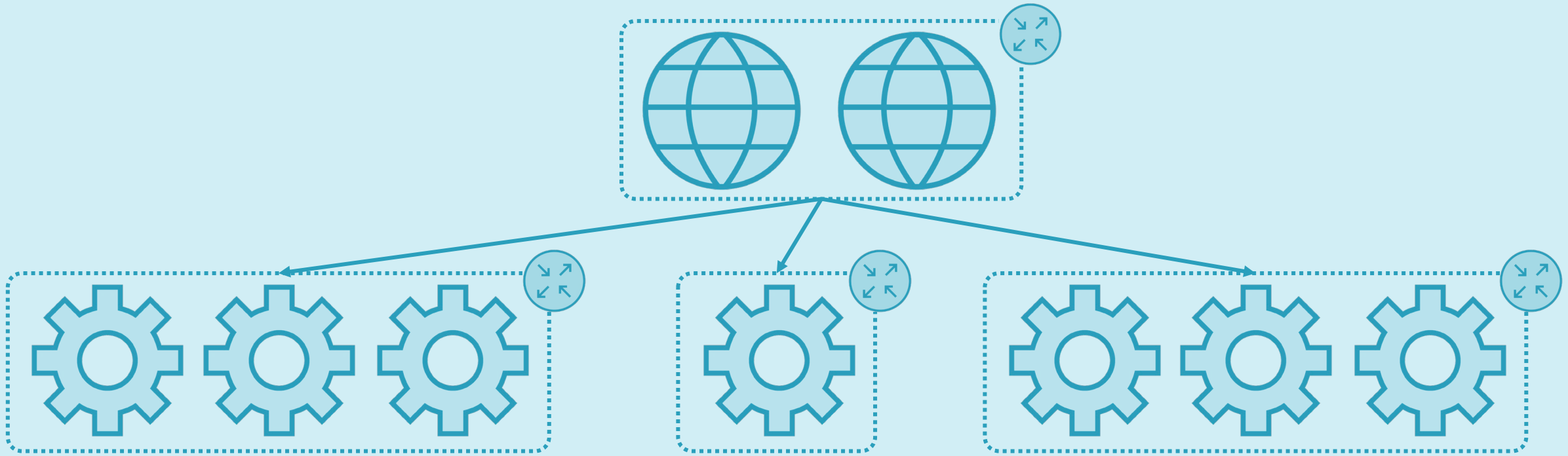




- DNS
- VIP + Network LB
- Software LB

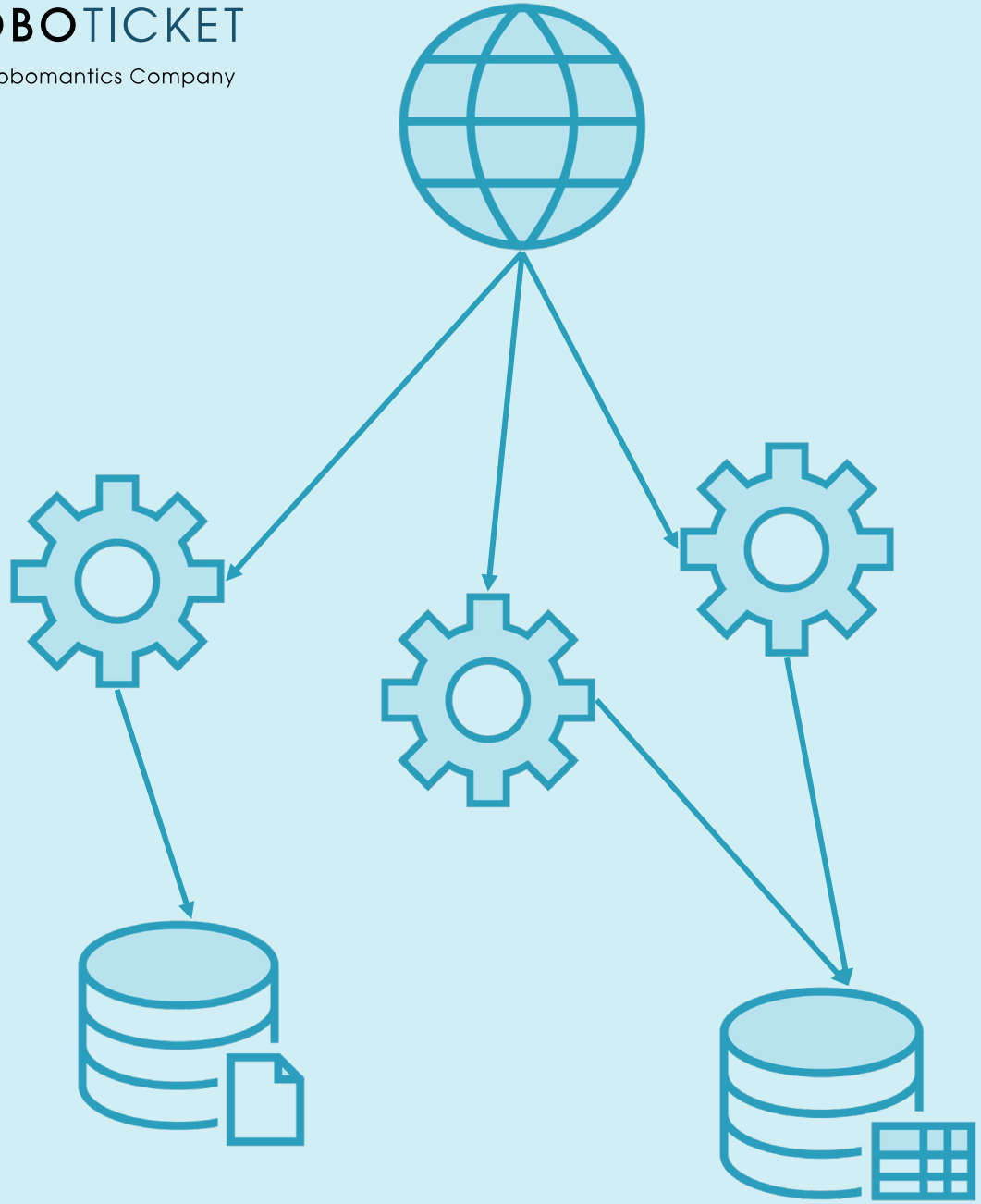
app.com

51.221.18.272



Scenario: Managing Demand with Scale

GLOBOTICKET
A Globomantics Company



Front-end

- Stateless
- Load-balanced

Back-end

- Stateless
- Load-balanced

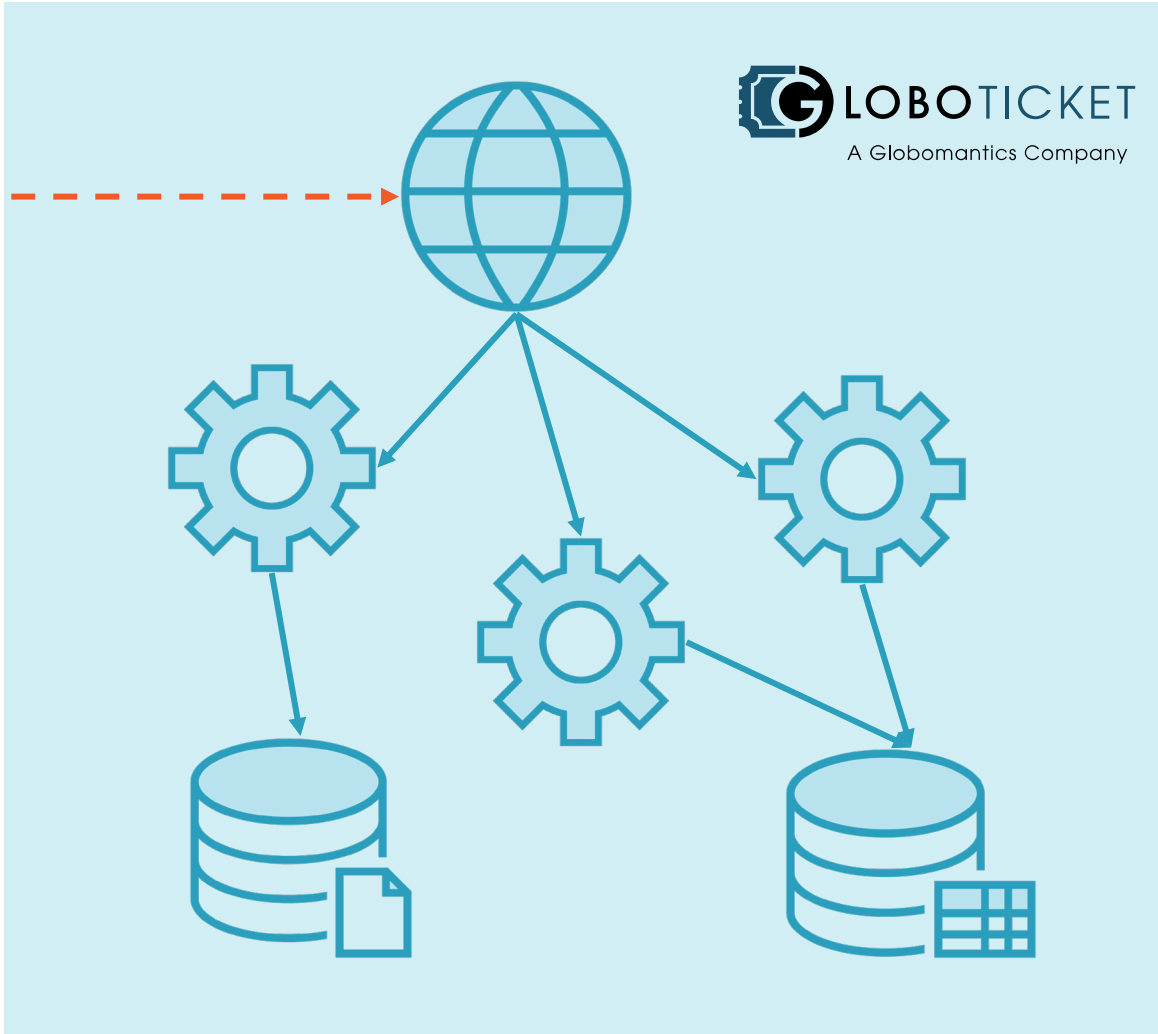
Database

- Transactional SQL
- Search no-SQL

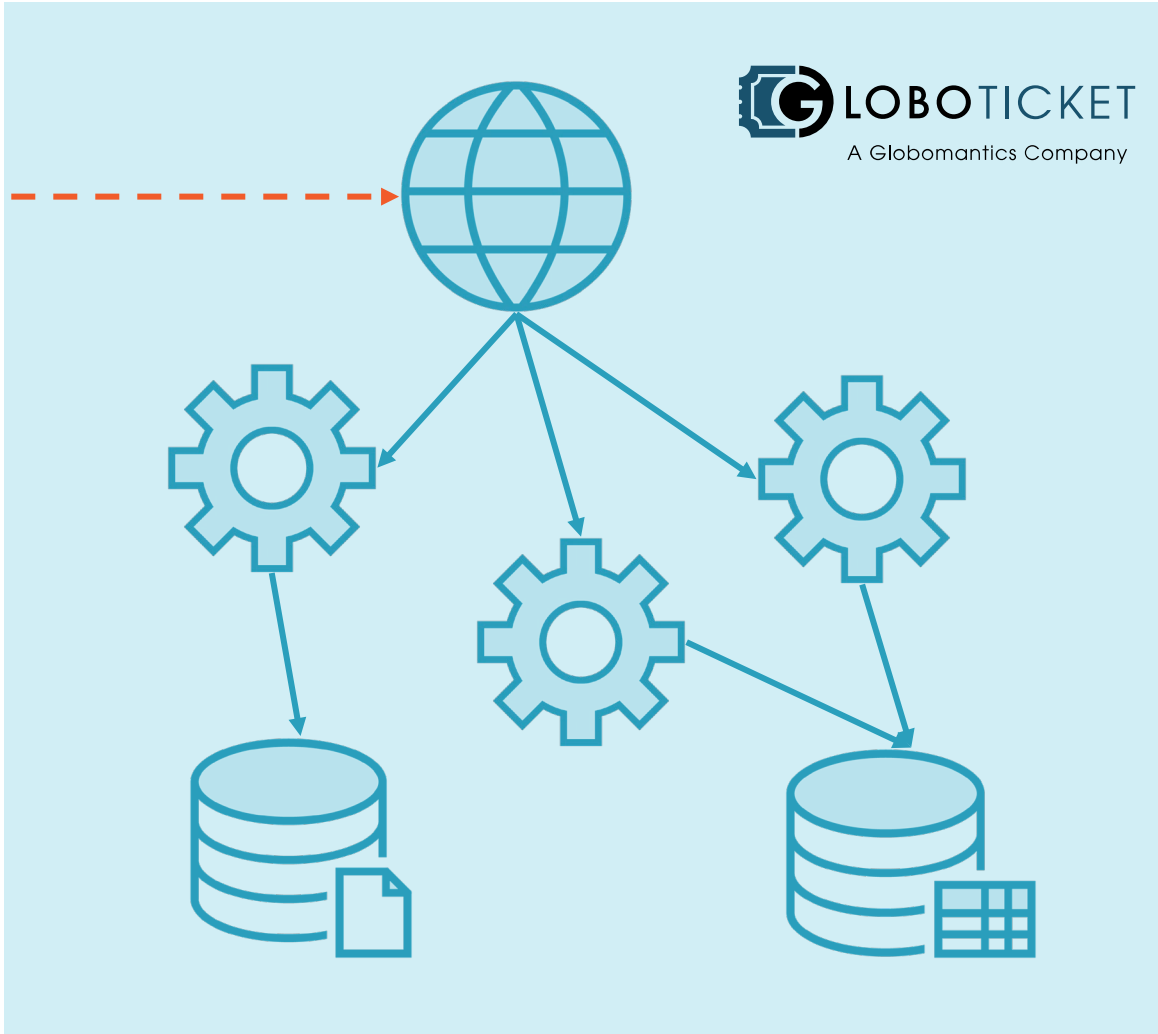
Amila

What happens when traffic spikes?

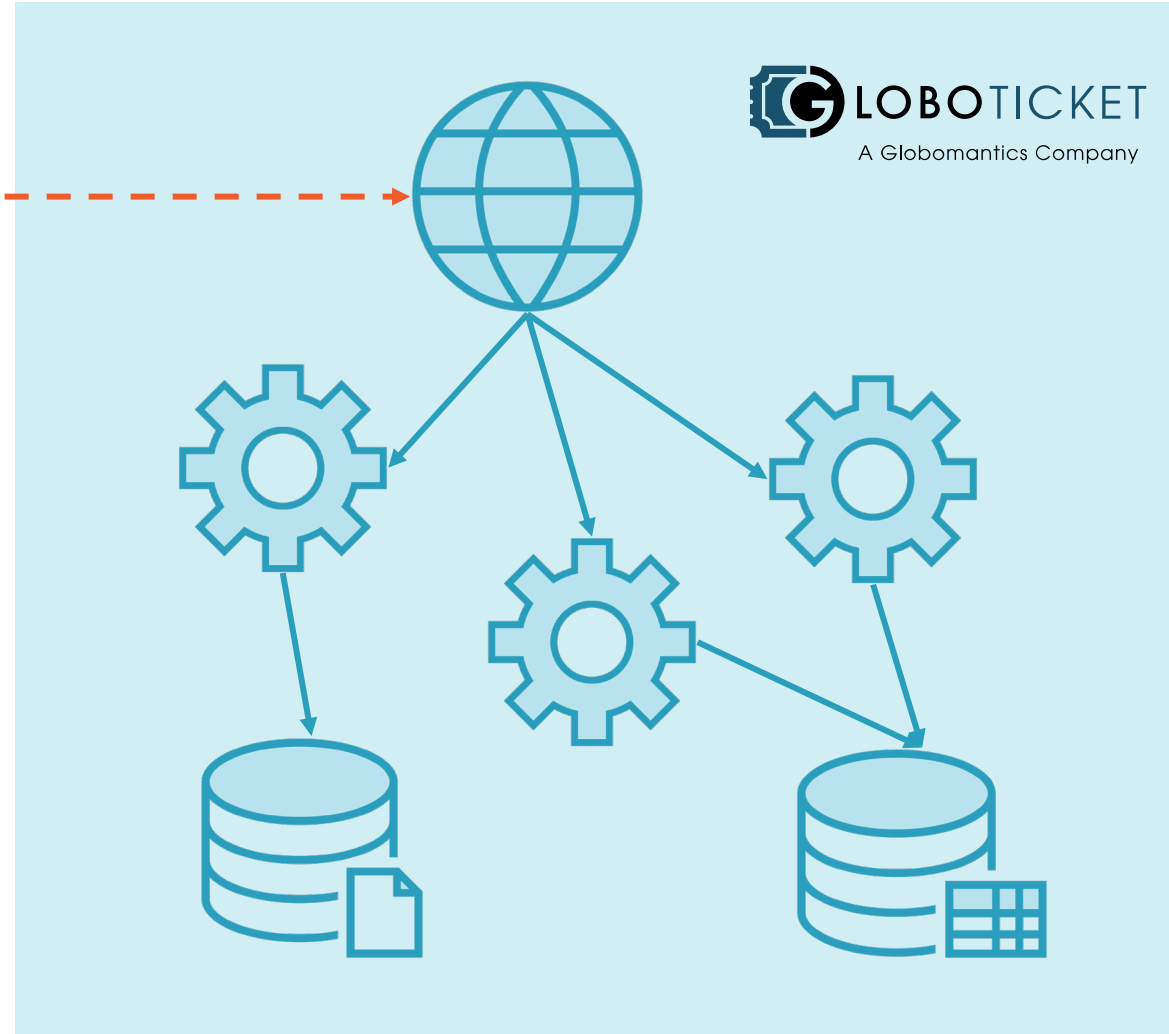
- Web response codes
- 503s trigger scale up
- 200s trigger scale down



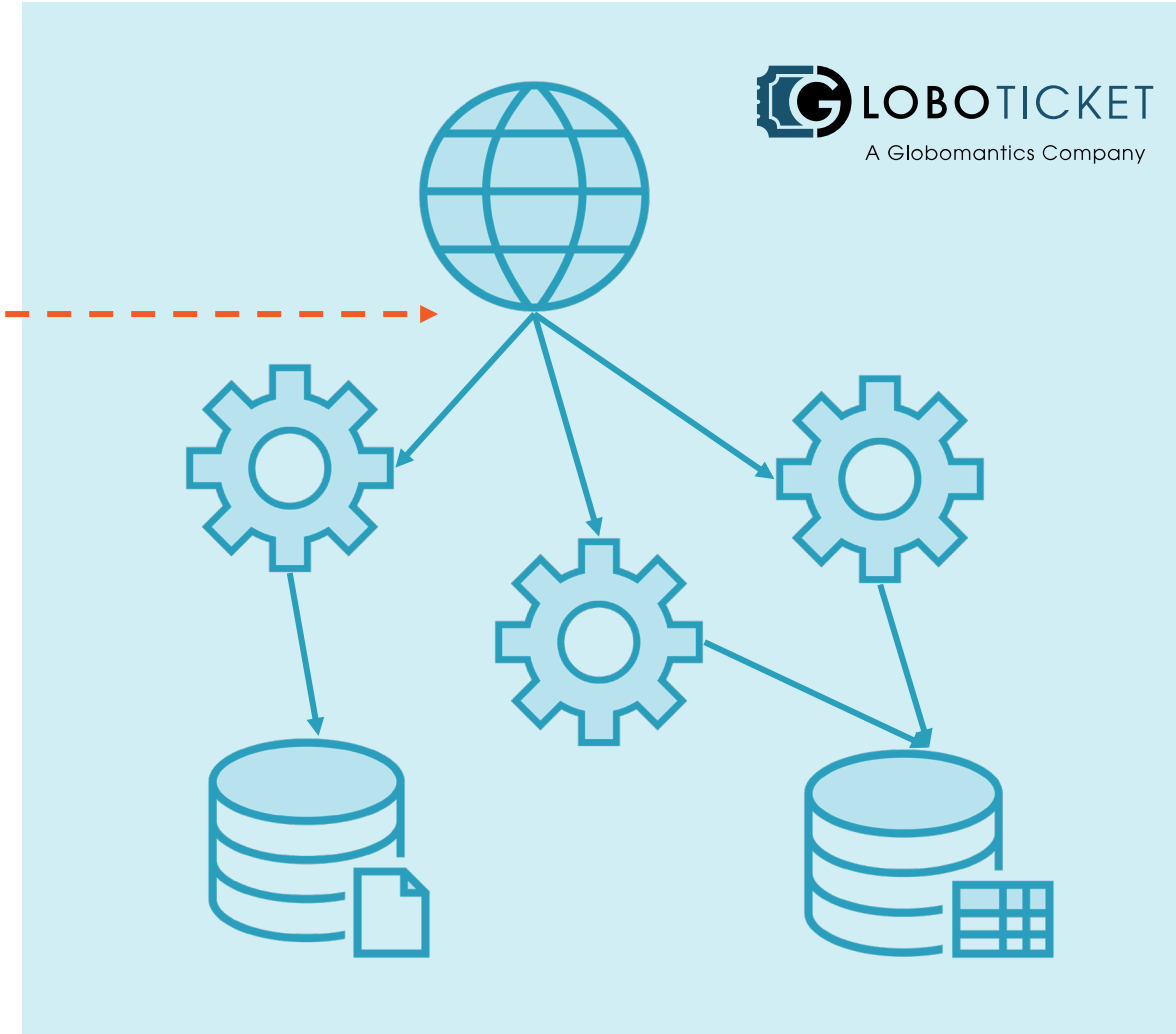
- Reactive
- Slow
- Poor UX



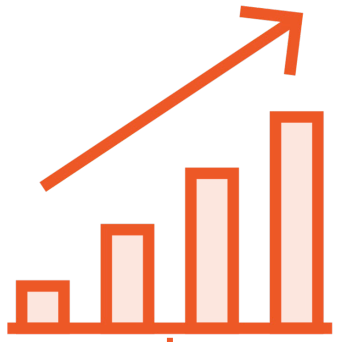
- Latency SLO
- Pre-emptive
- Positive UX



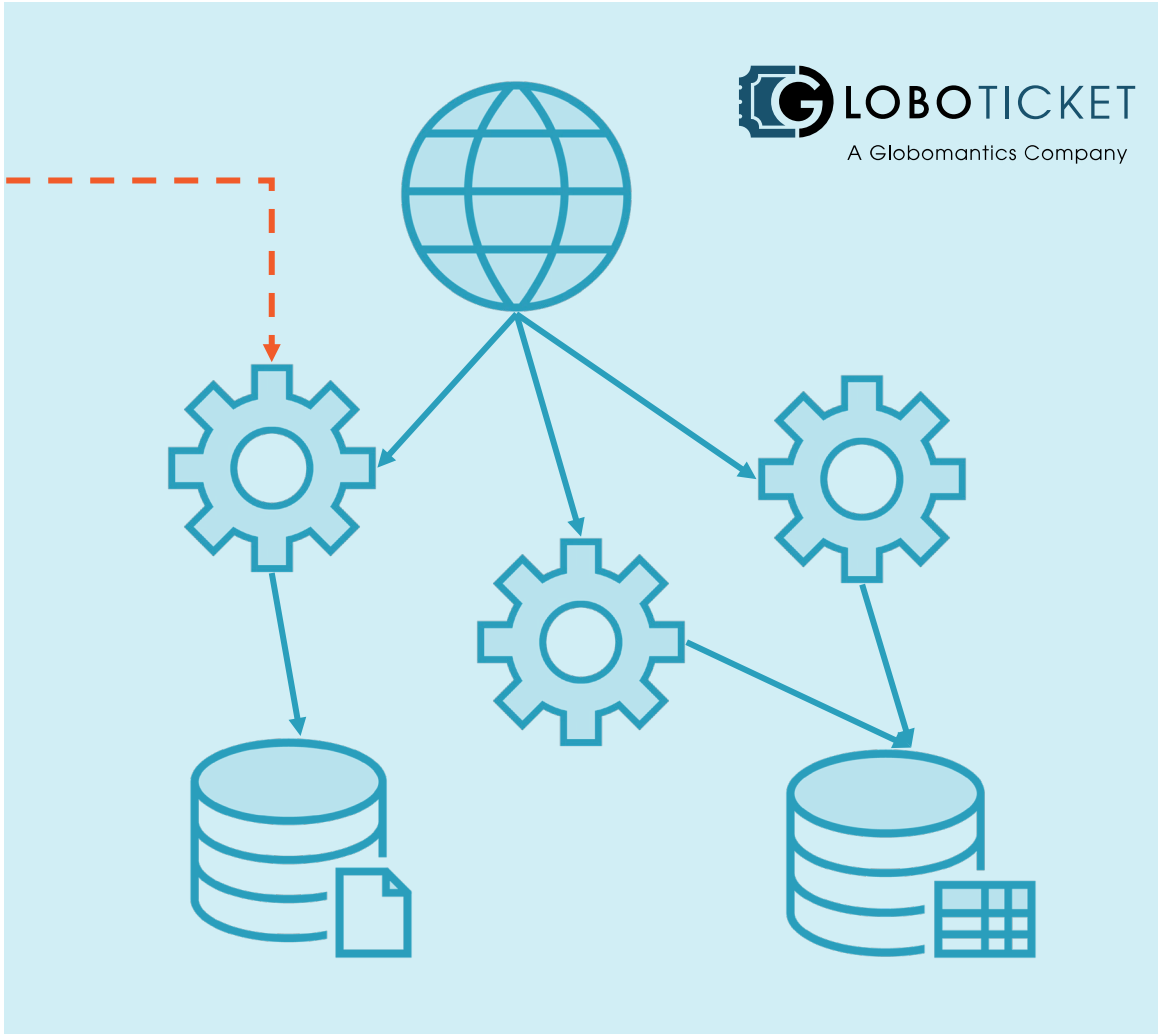
What if the APIs can't handle the load?



- Standard API client
- Tracks failures
- Automatic retries

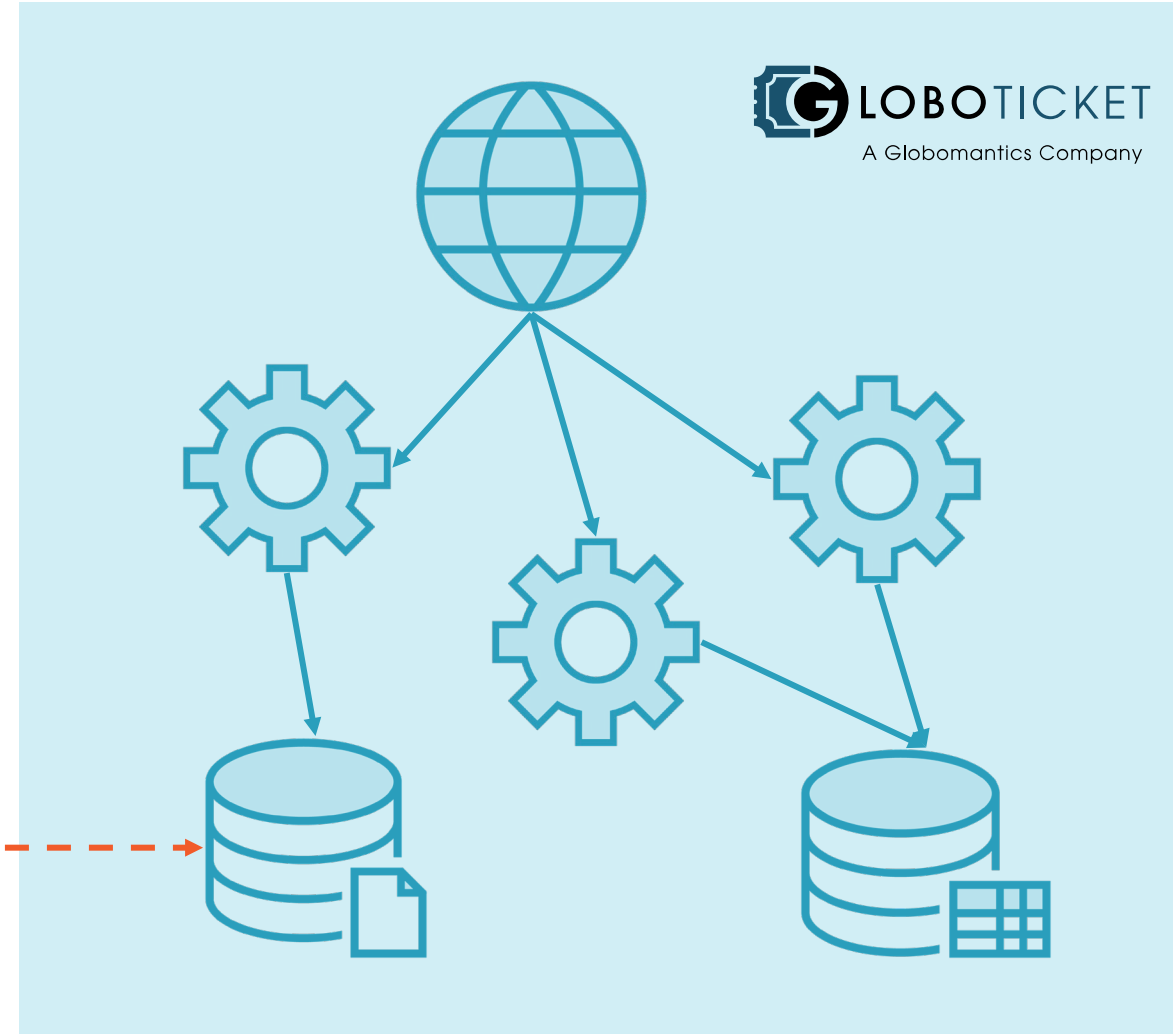


- Auto-scale
- Traffic SLO
- Circuit breaker

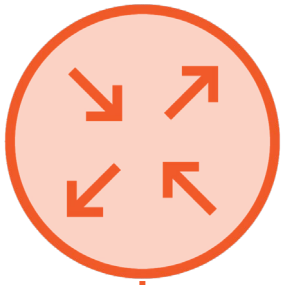


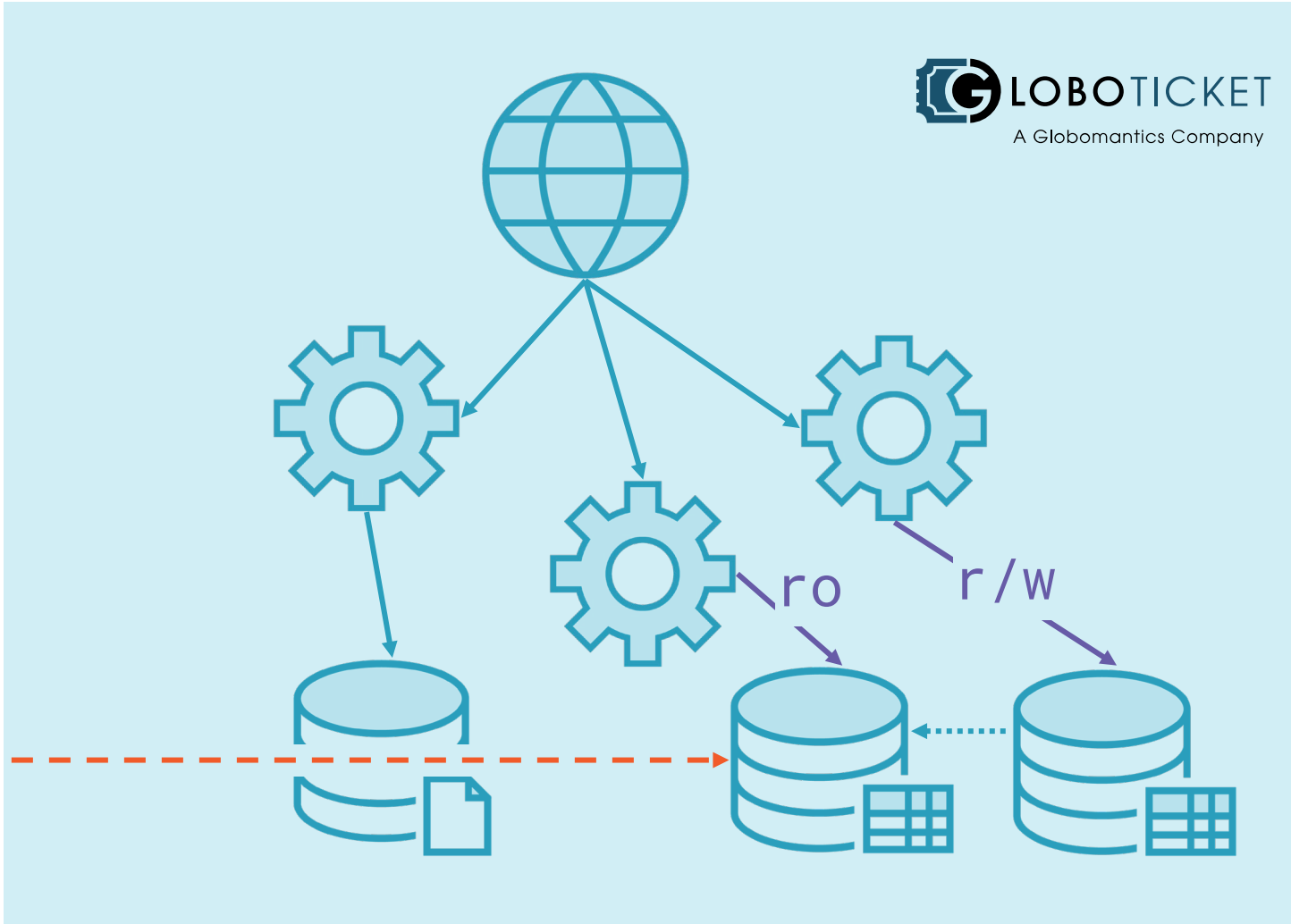
What if the APIs max out the database?

- Replicated
- Doc DB has headroom
- SQL DB not easily scaled

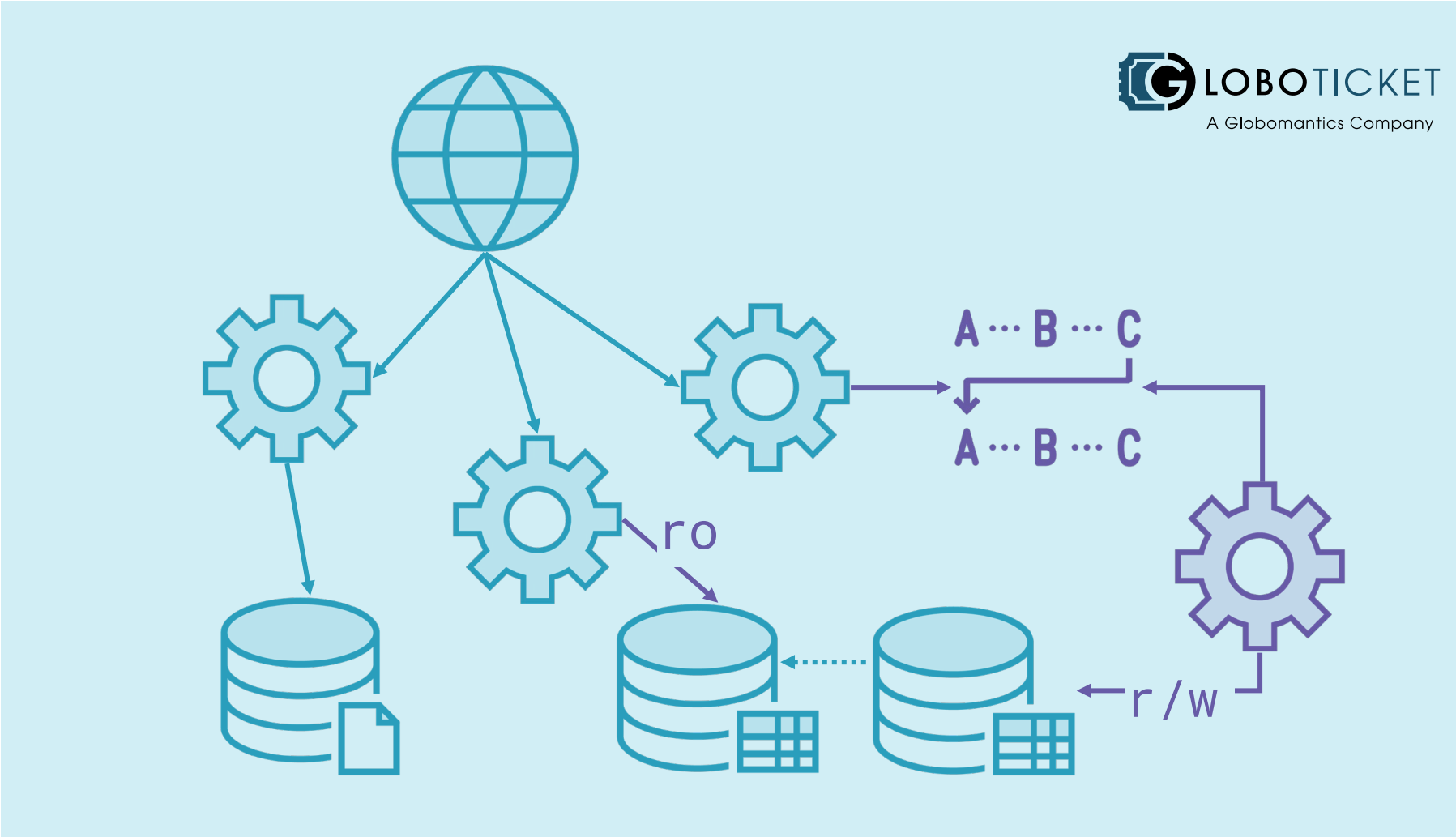


- Load-balance SQL
- Write primary
- Read secondary

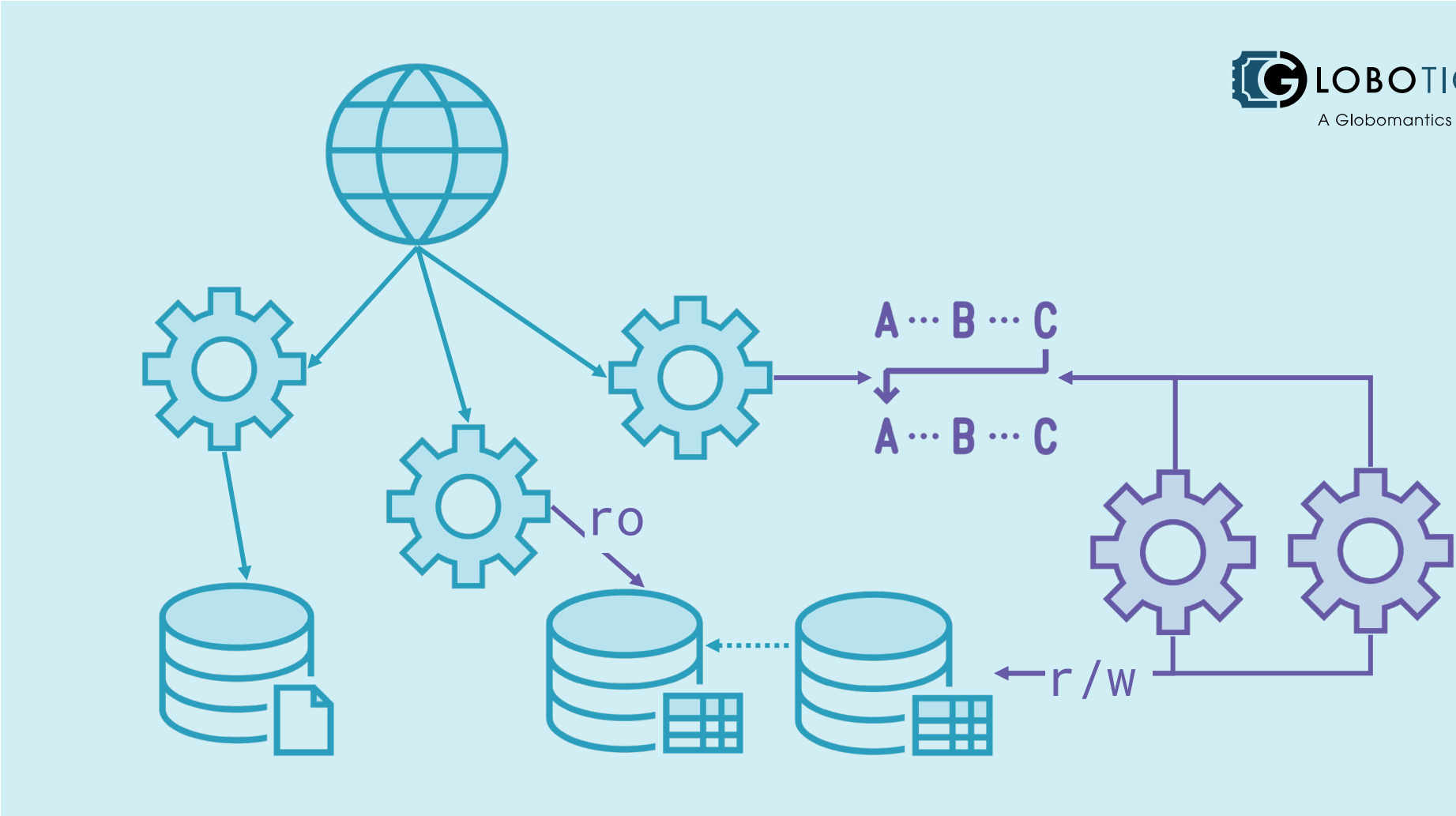




- Asynchronous messaging



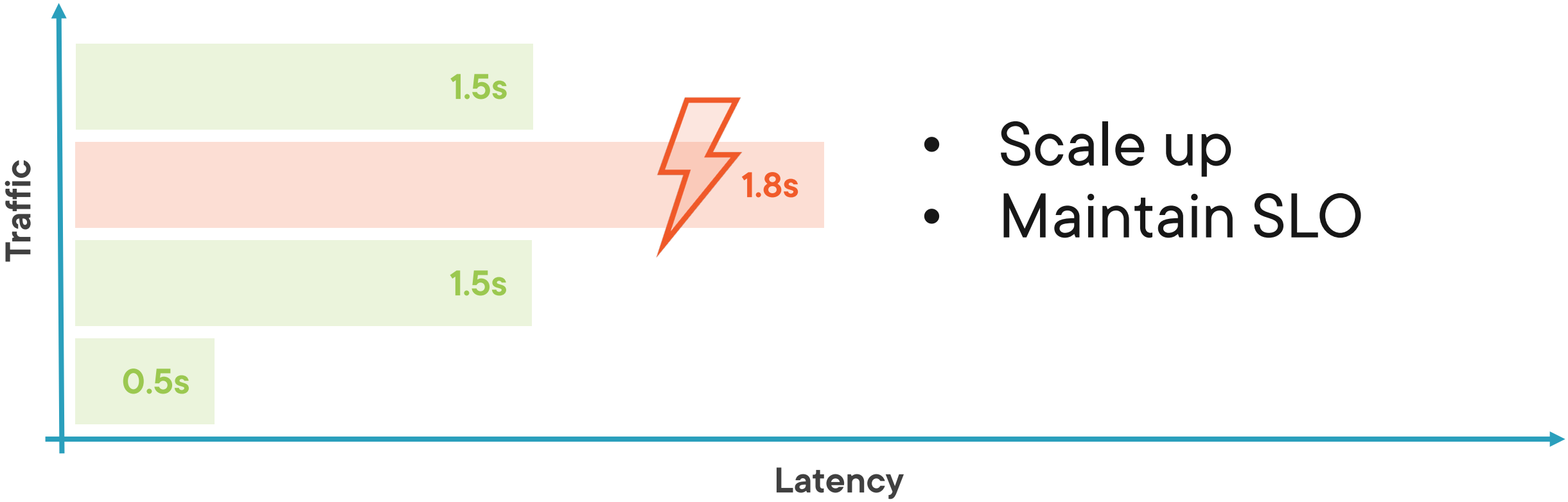
- Asynchronous messaging
- Control read/write scale



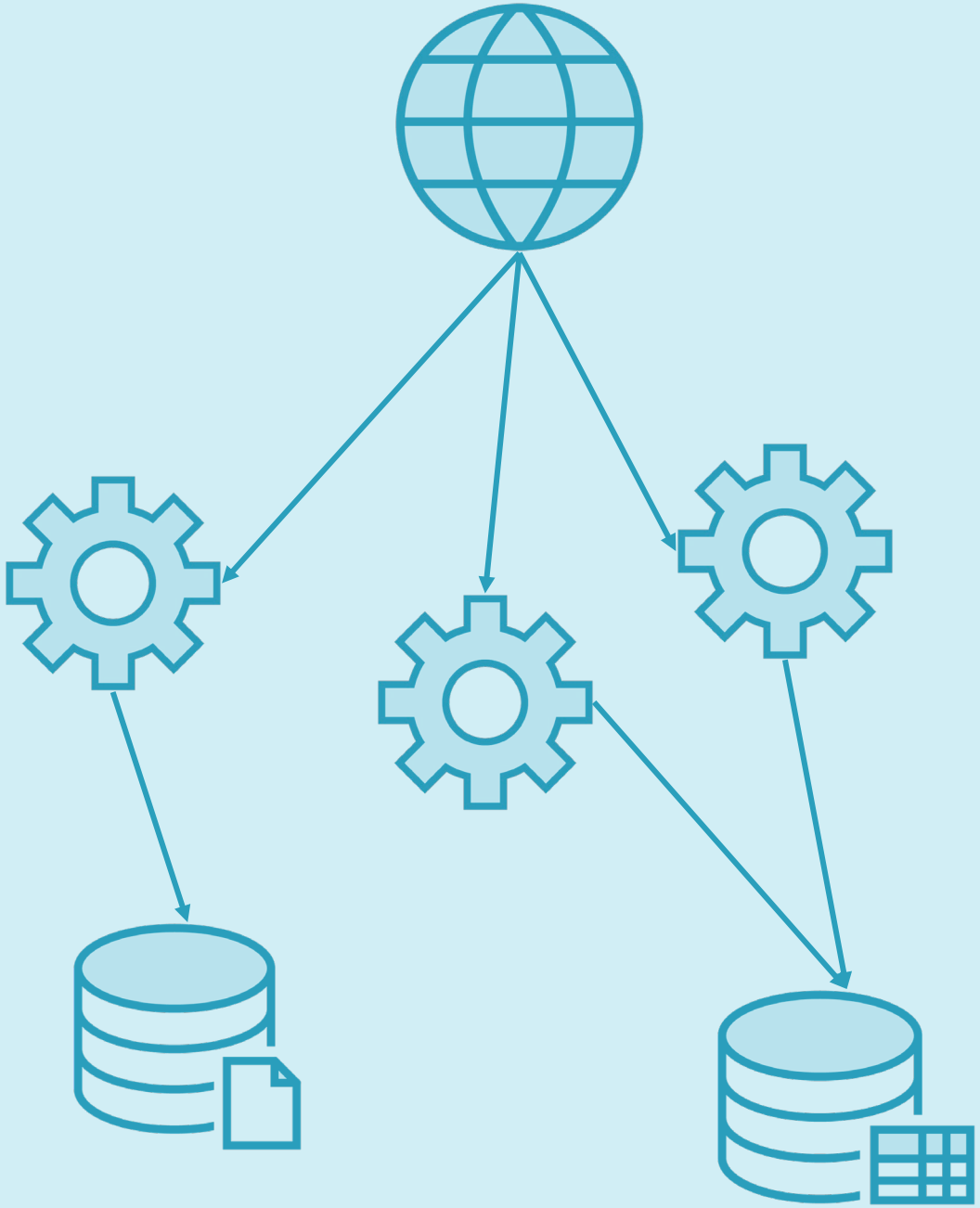
Managing Scale and Overload

Triggering Scale from SLOs

Response time **99.9% of requests within 2 seconds**



- Scale up
- Maintain SLO



Front-end

- Auto-scale
- Reactive trigger

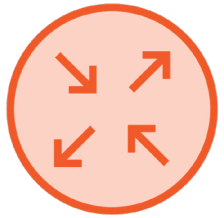
Back-end

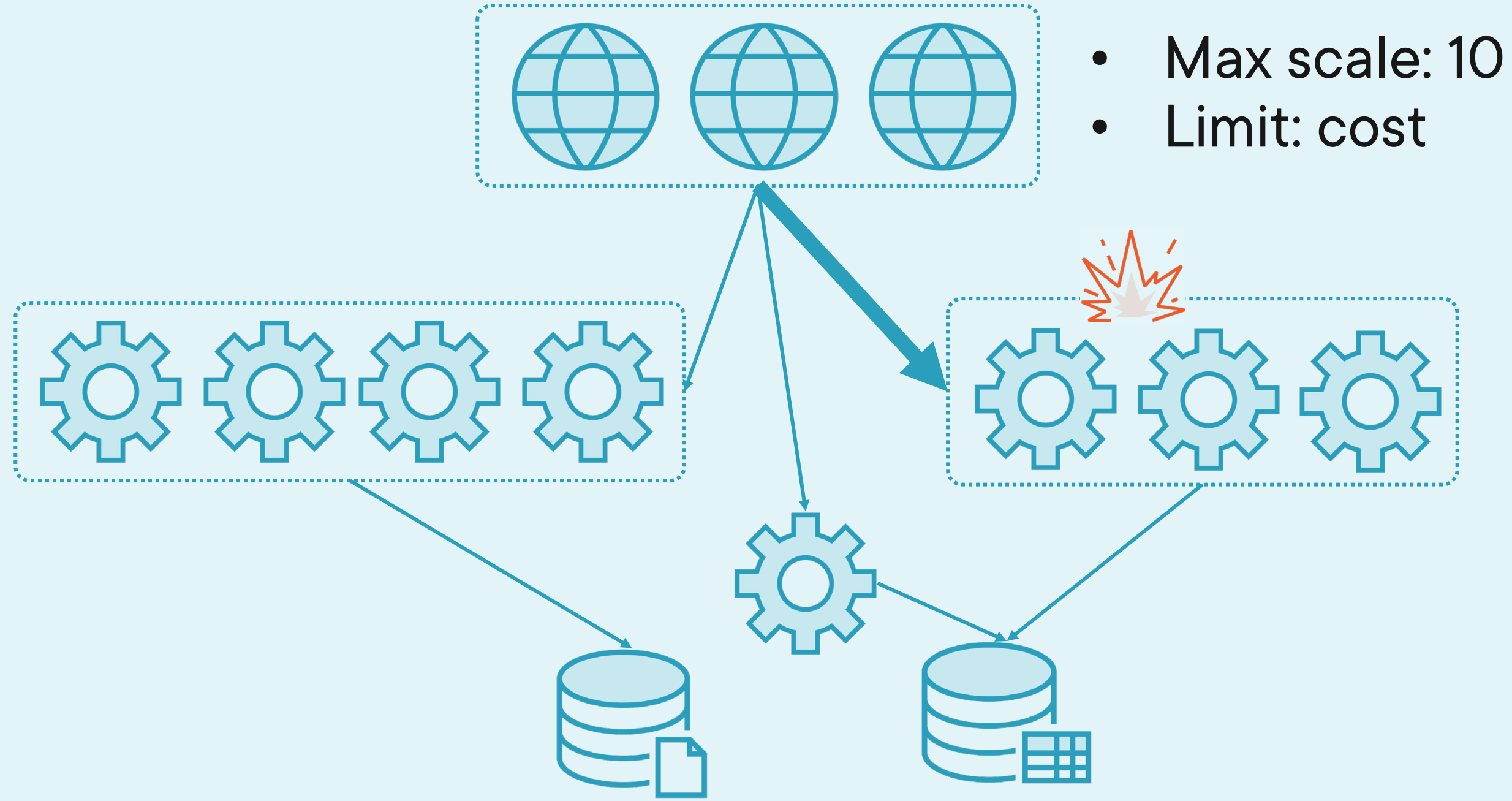
- No auto-scale
- Risk of overload

Database

- No auto-scale
- Limited options

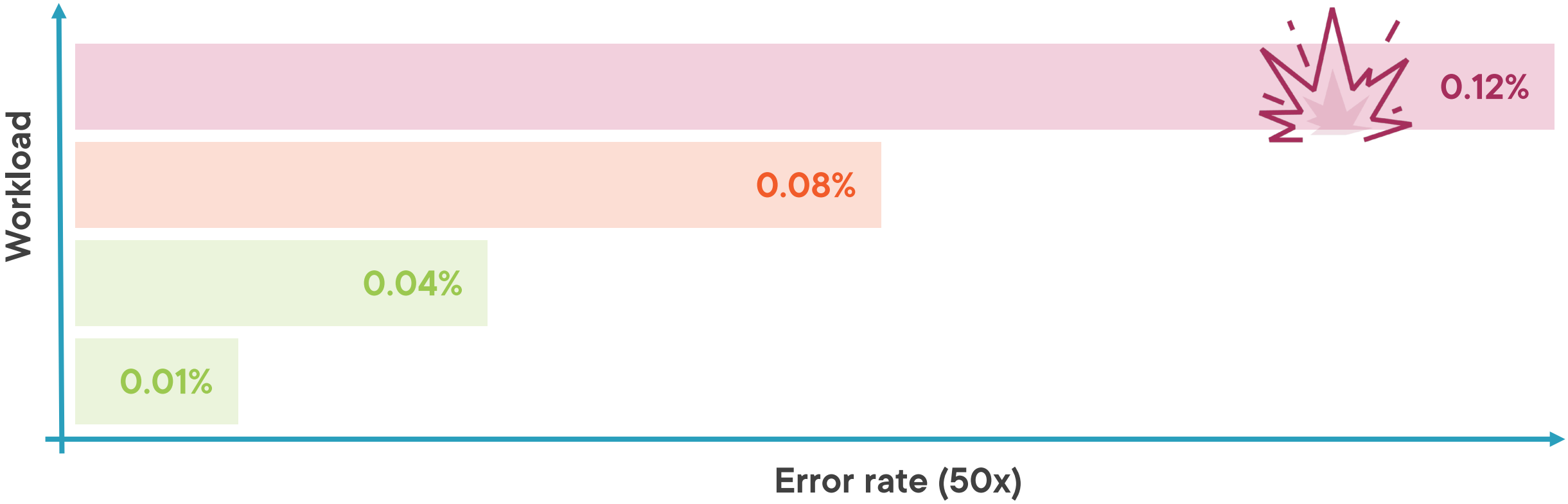
SRE Backlog

	Effort	*Team*	*Priority*
 **Scale web by latency SLO**	**? days**	**SRE**	**?**
 **Scale APIs by traffic SLO**	**? days**	**SRE**	**?**
 **Load-balance SQL reads**	**? days**	**Dev**	**?**


• Max scale: 10
• Limit: cost

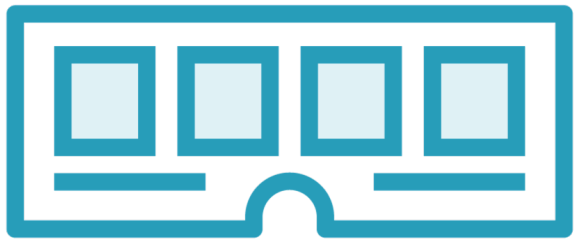
Service Level Objective

Success rate 99.9% of requests should succeed



Managing Overload

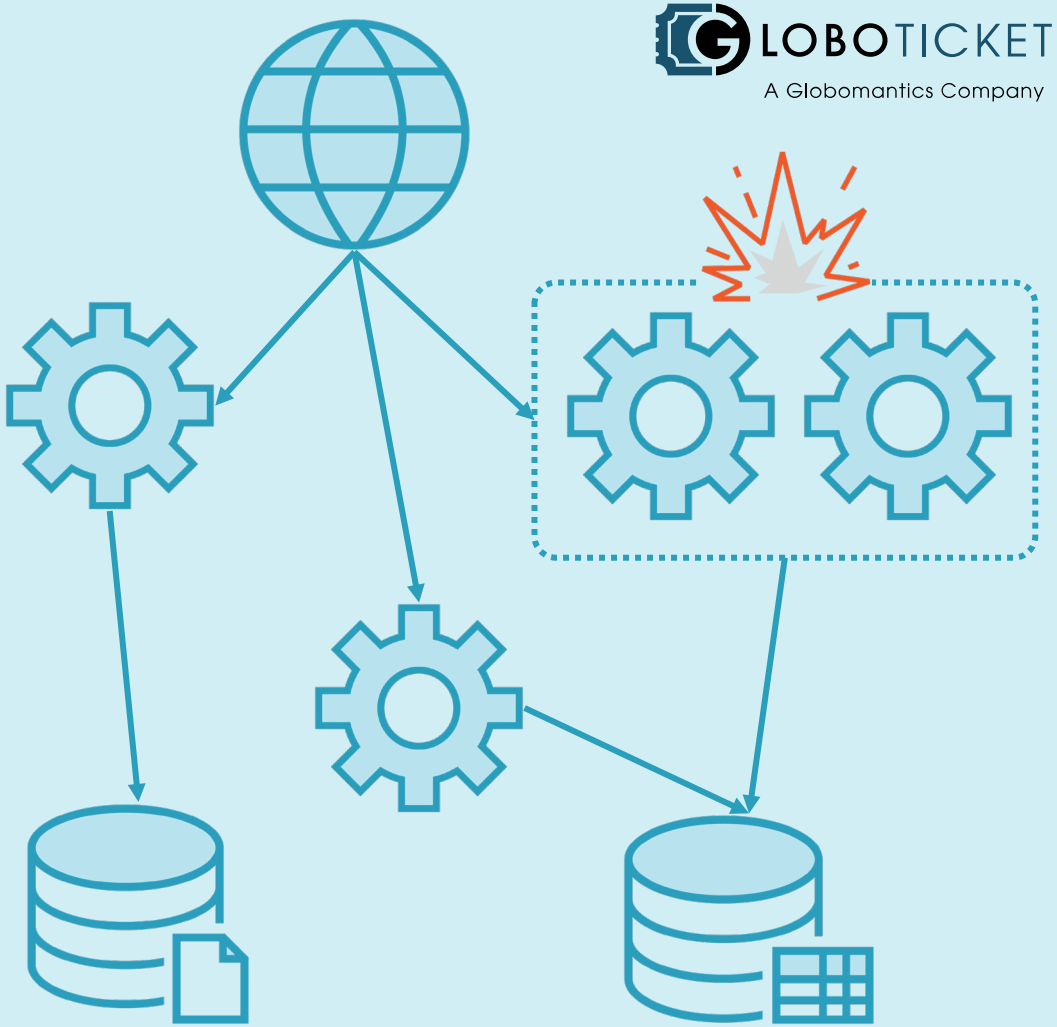
Reduce workload



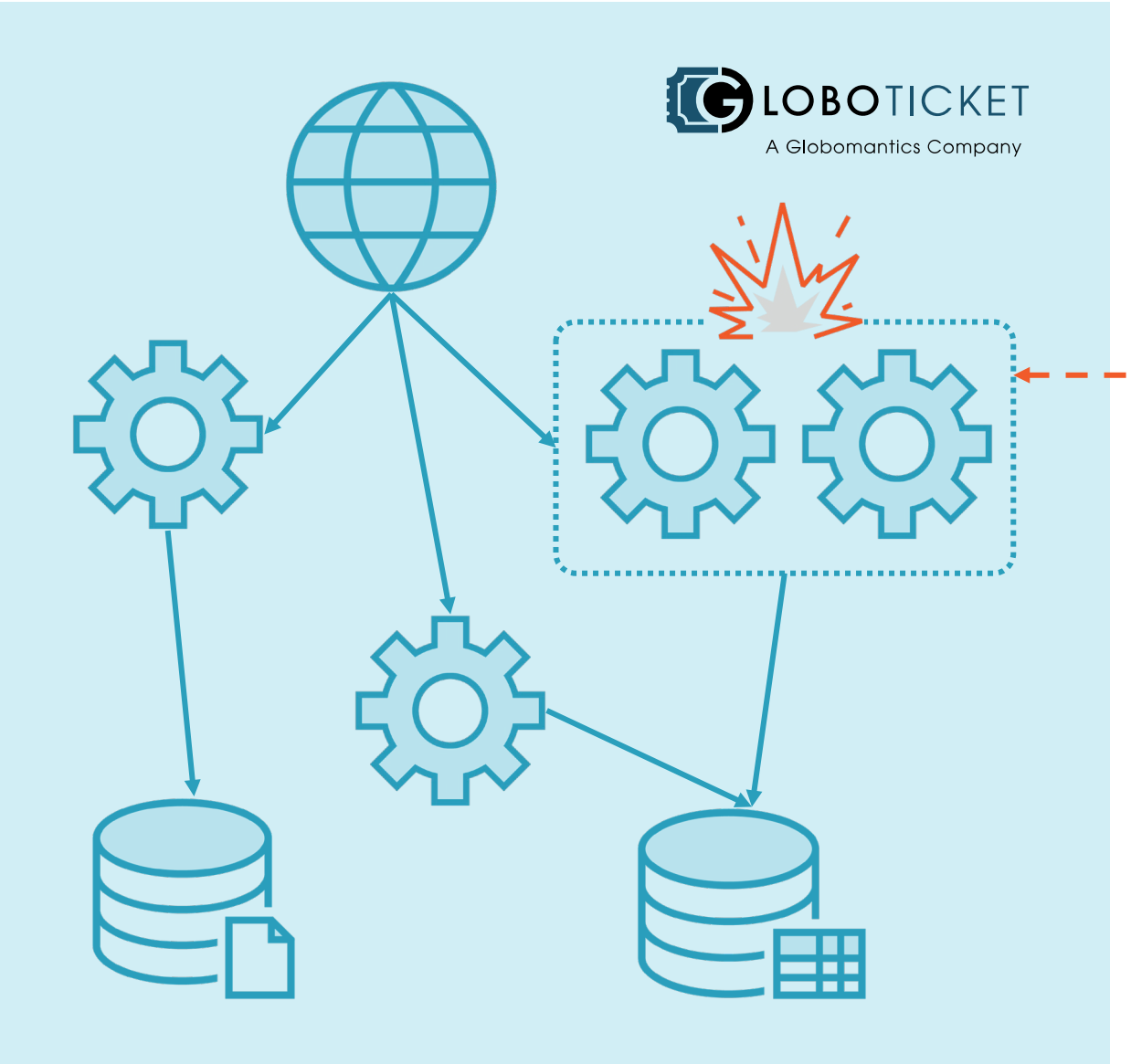
Cache responses

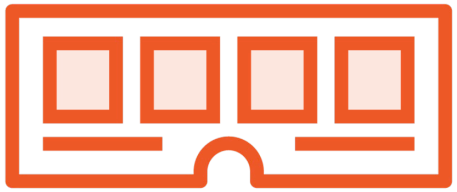
Switch features off

Scenario: Managing Overload

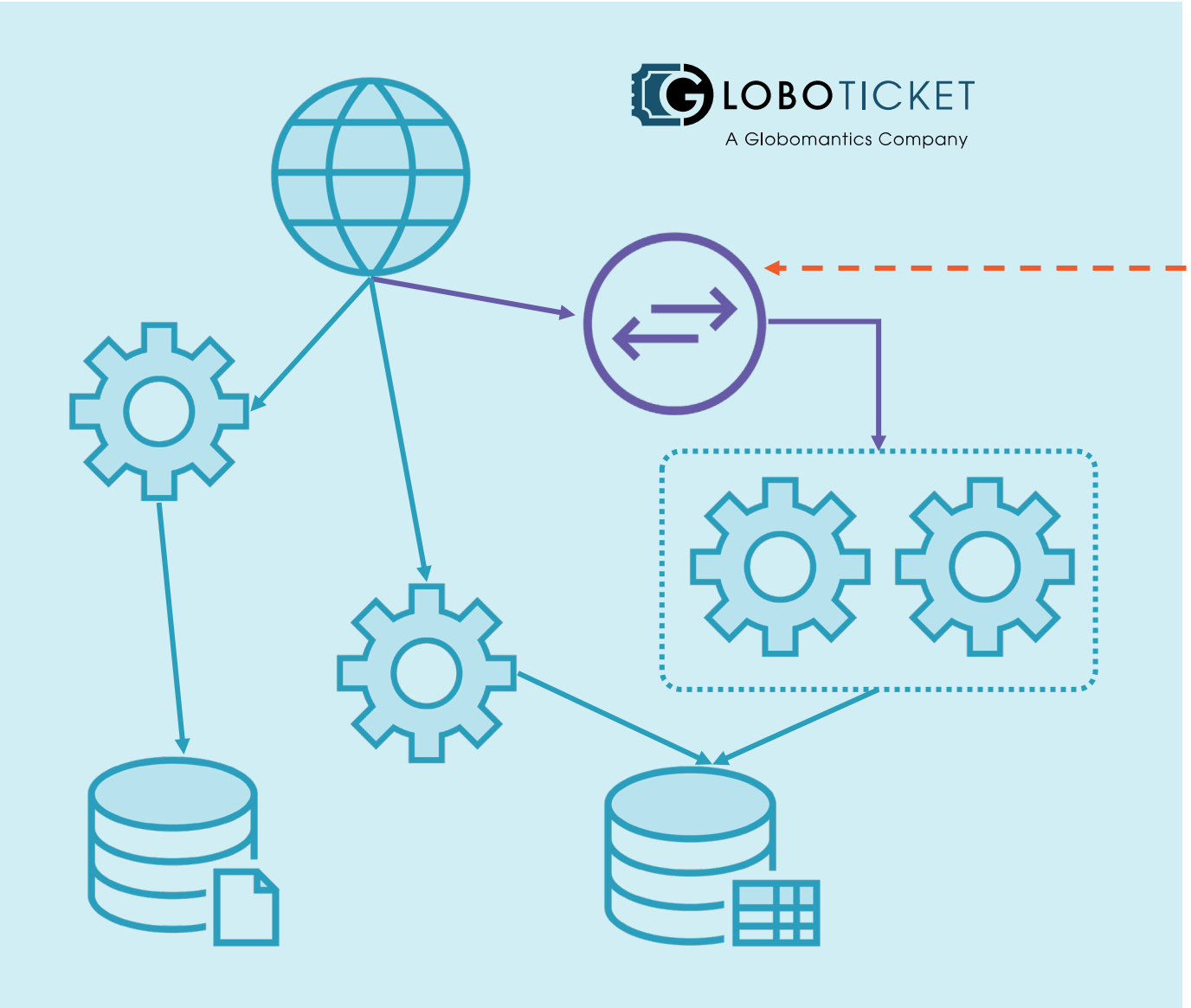


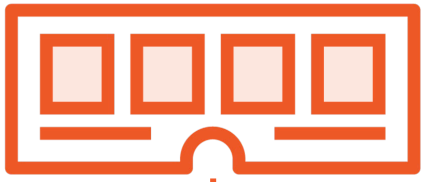
At max scale - what happens when APIs fail?



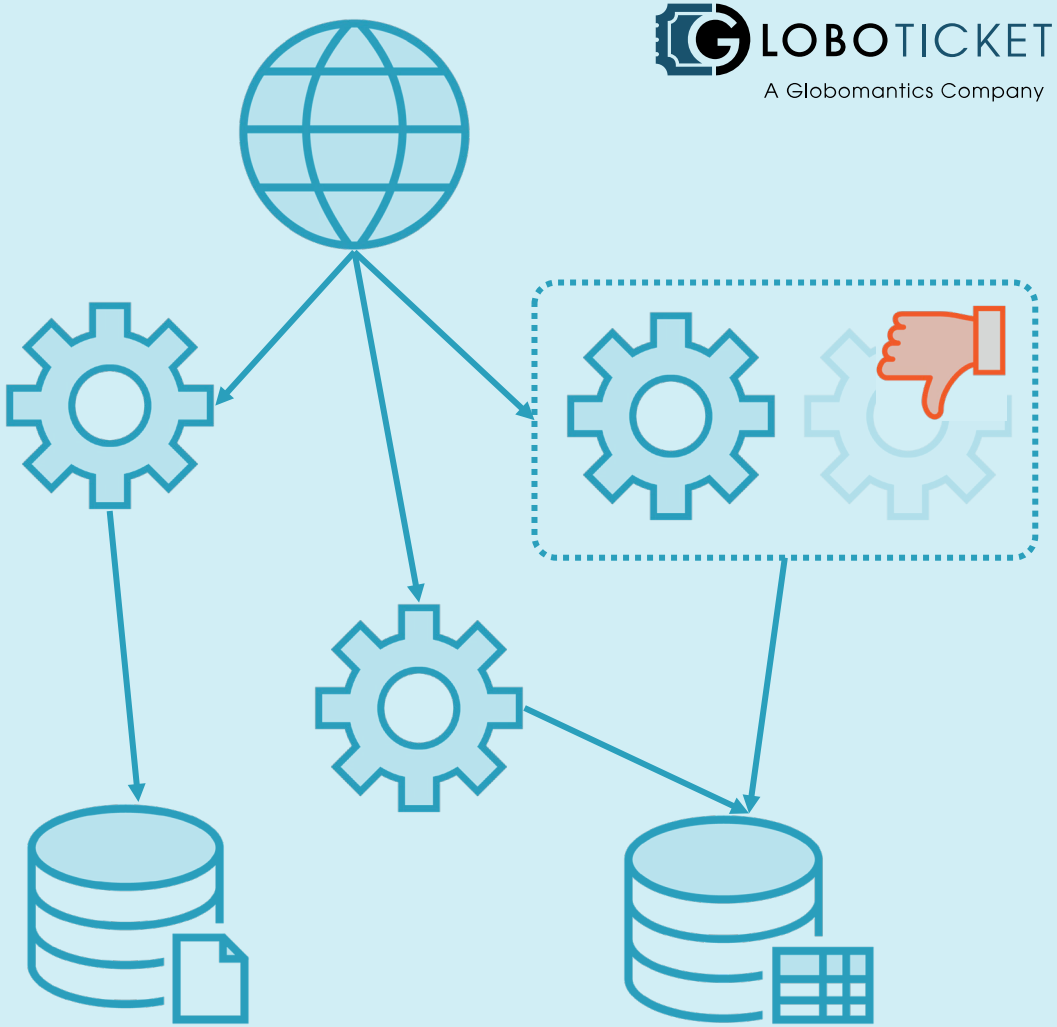


- Configurable cache
- Off by default
- No config reload
- Restart loses work

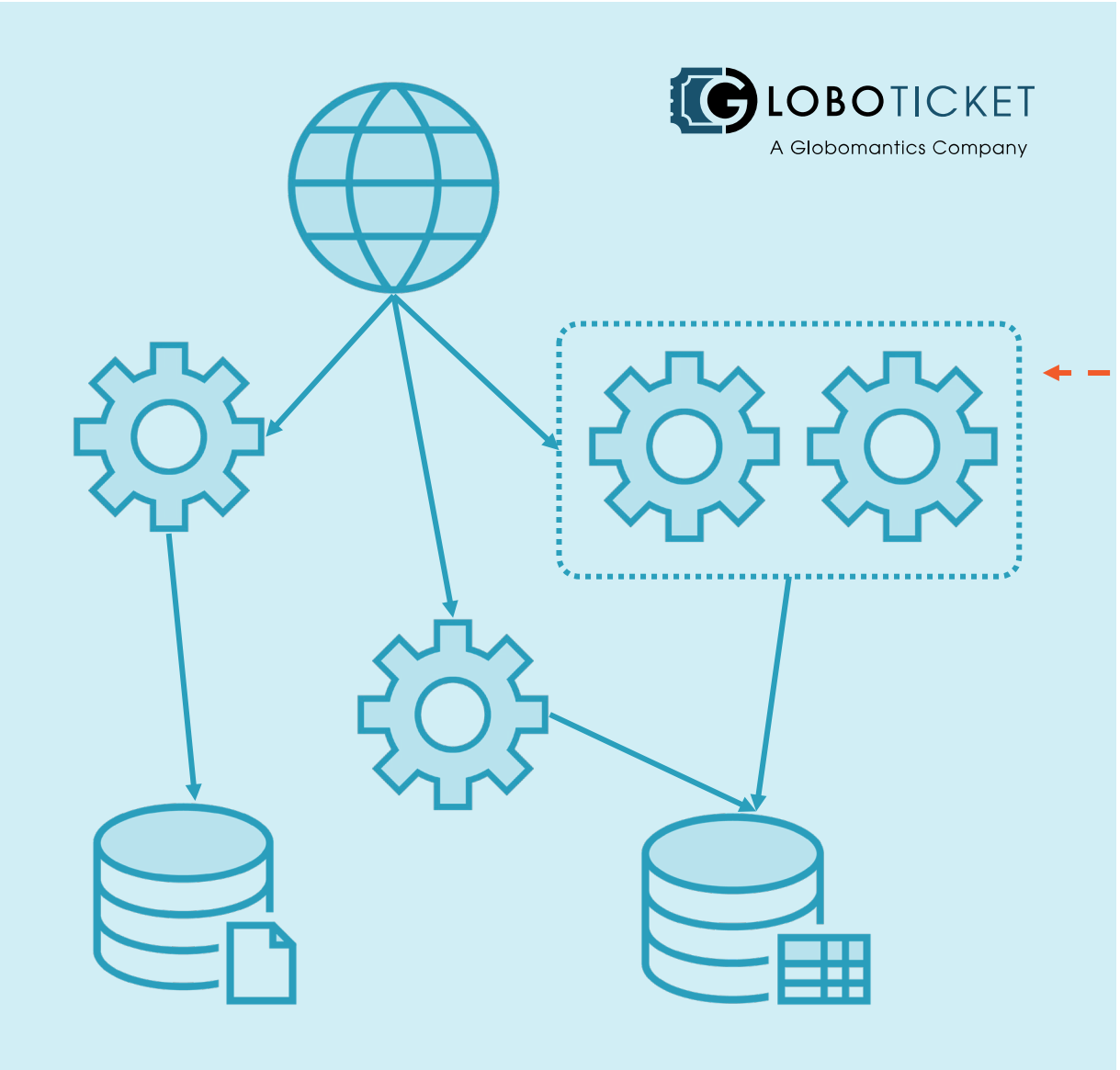




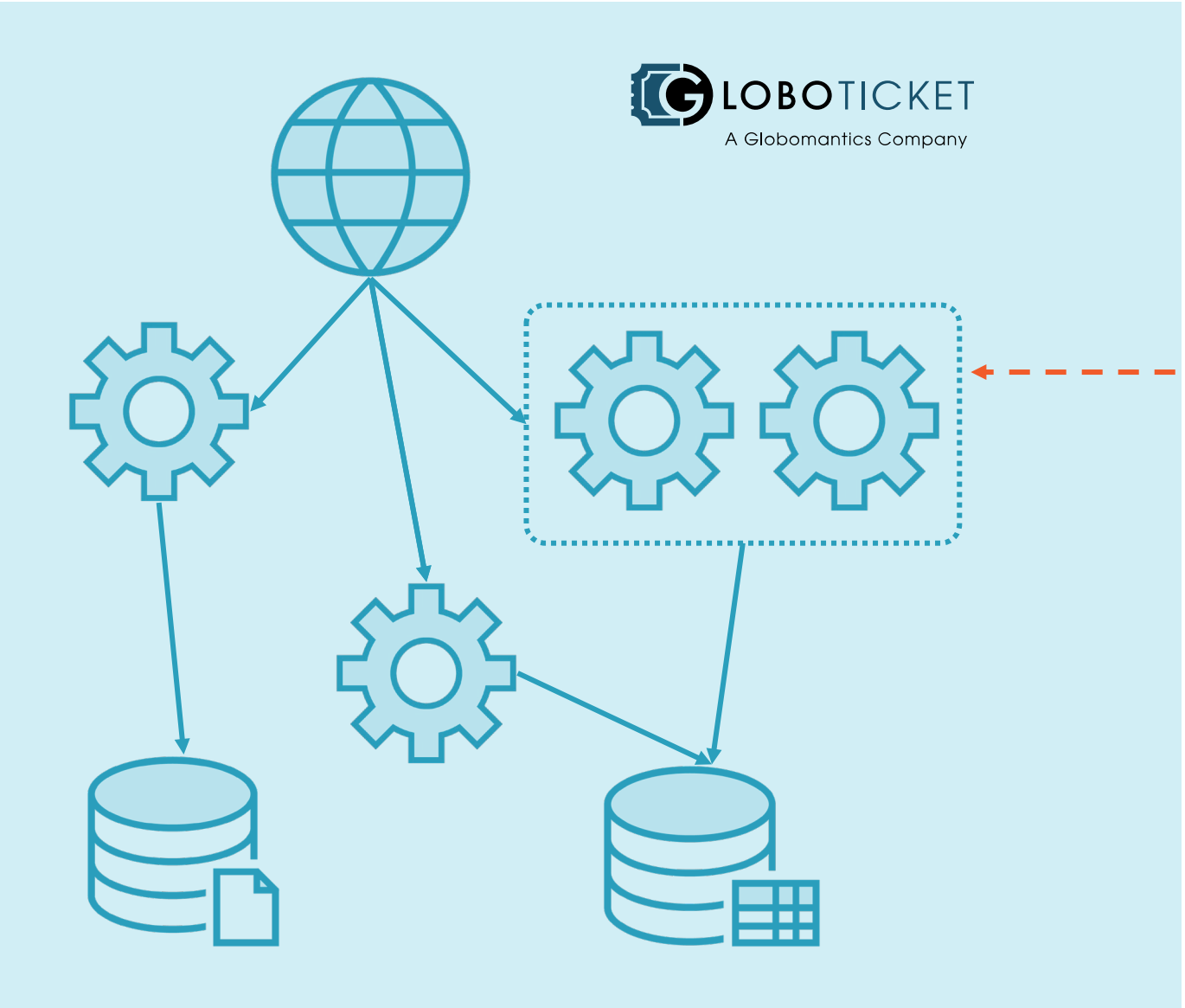
- Reverse proxy
- HTTP cache
- Config reload



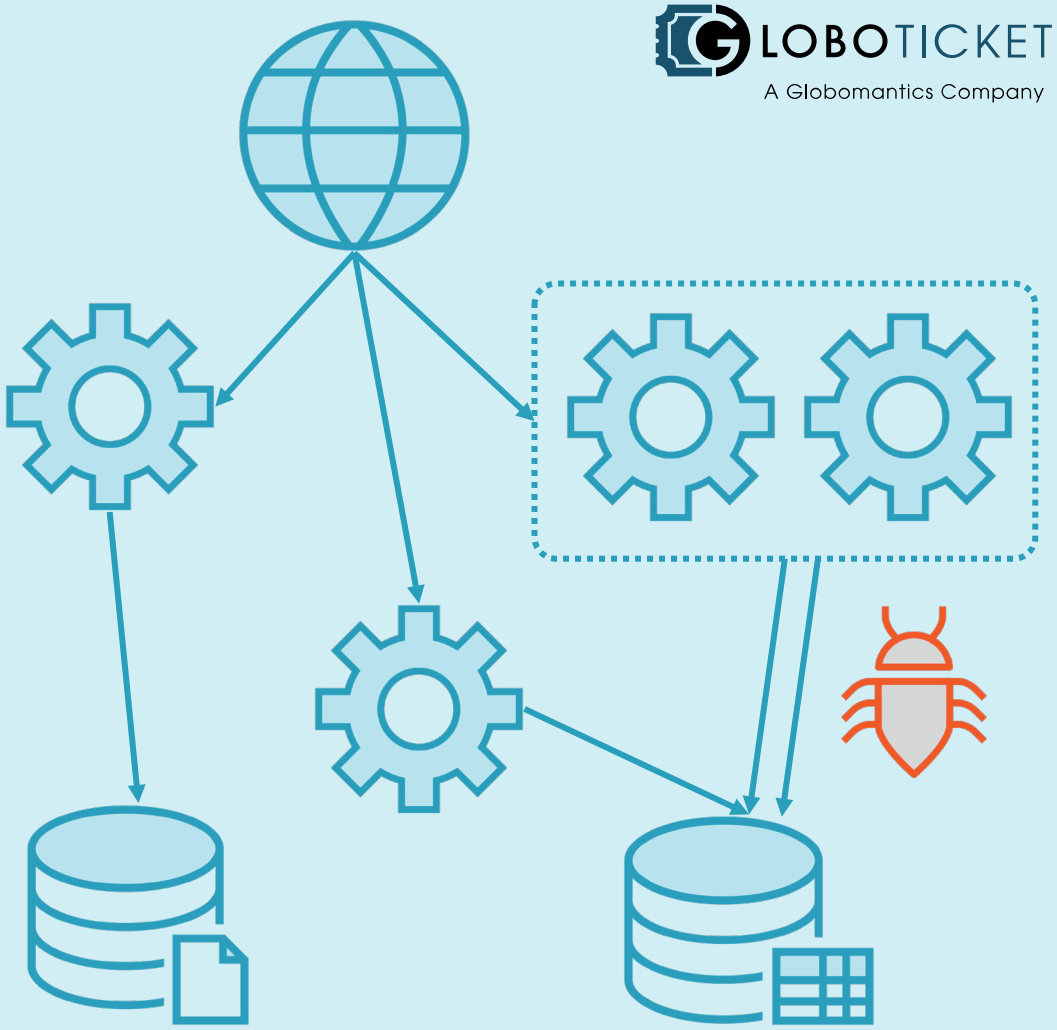
Do unhealthy instances get restarted?



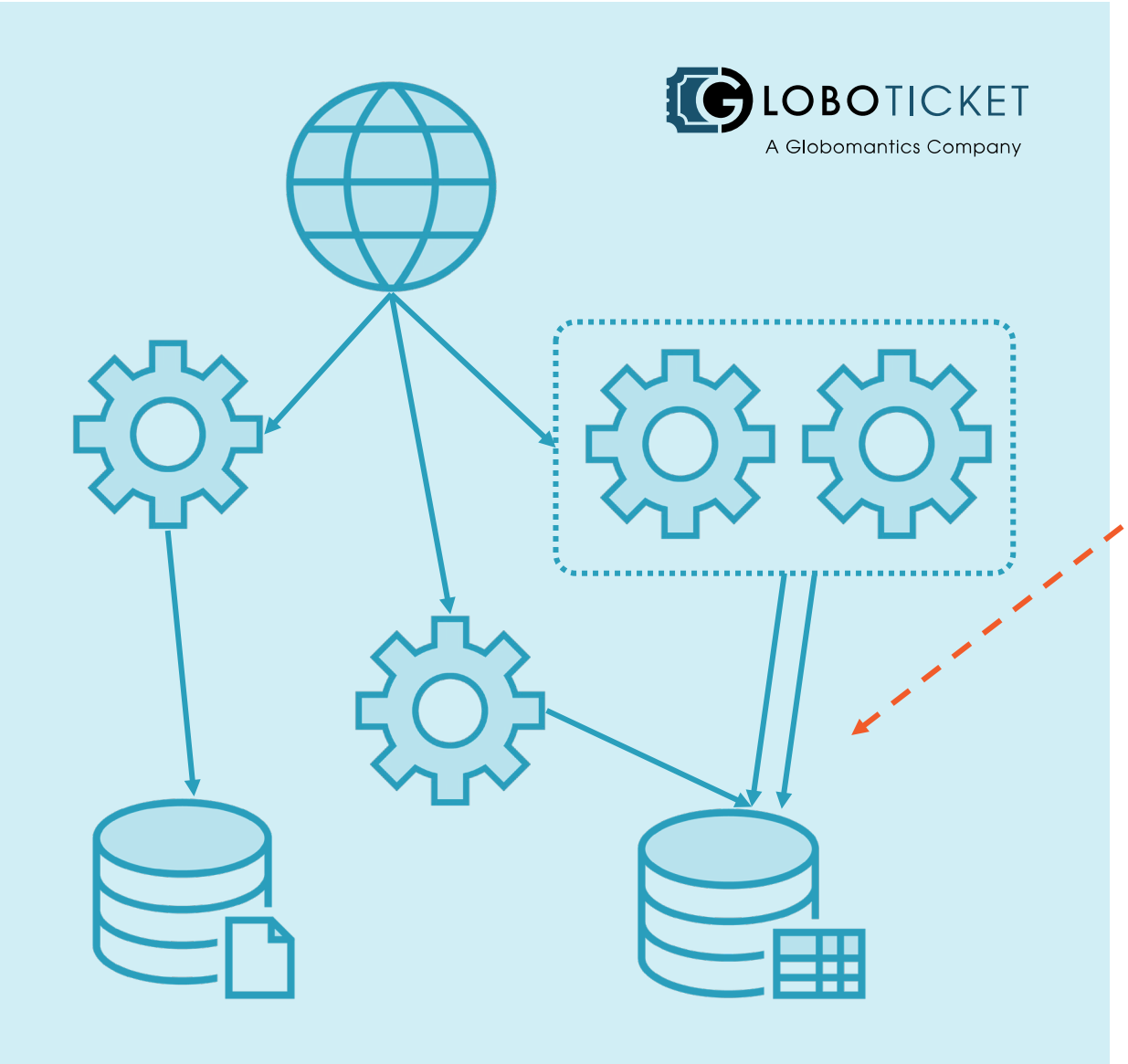
- No health checks
- Cannot be restarted
- End client connections



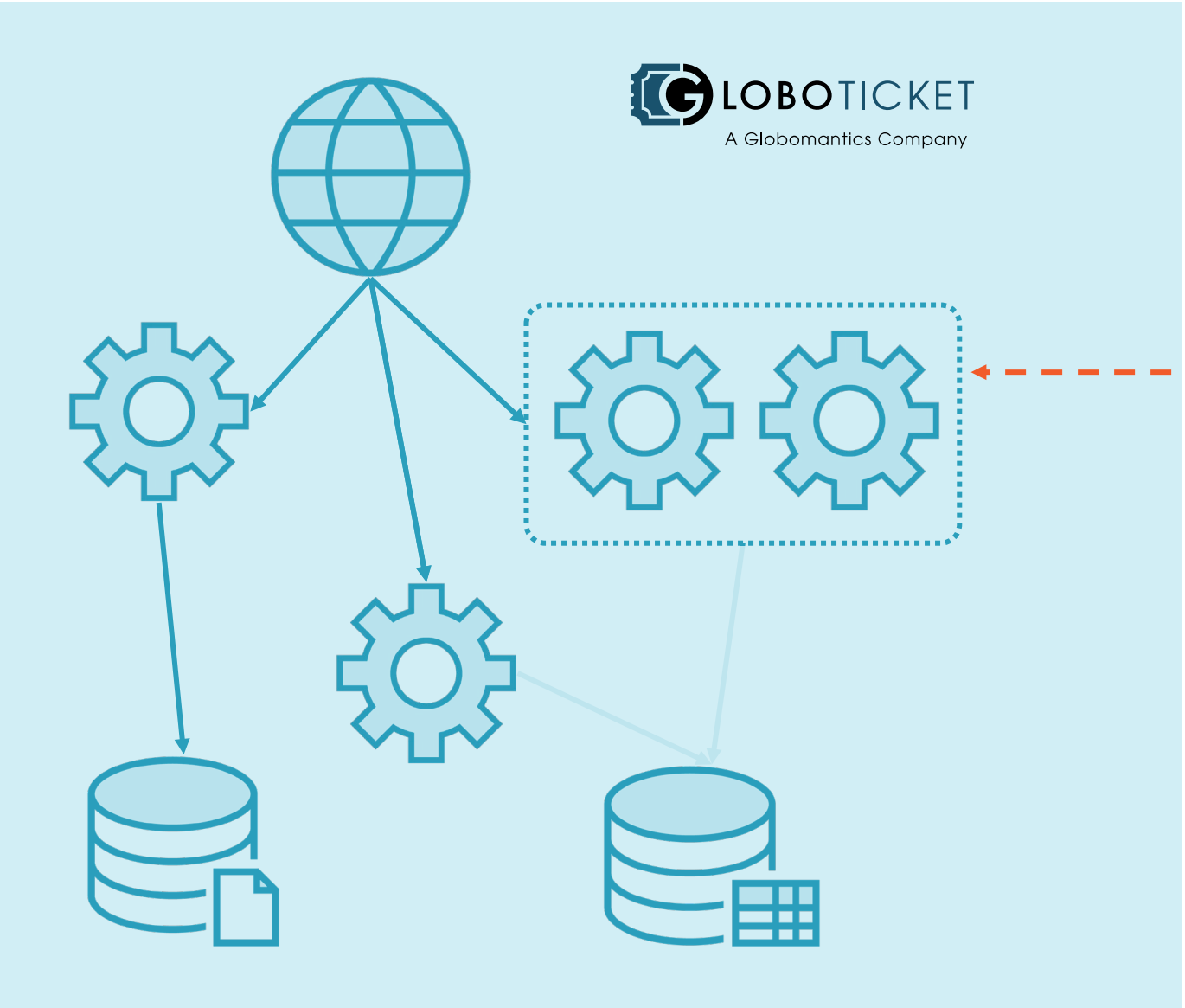
- Support restart
- Graceful exit
- No new load



Can you prevent data corruption?

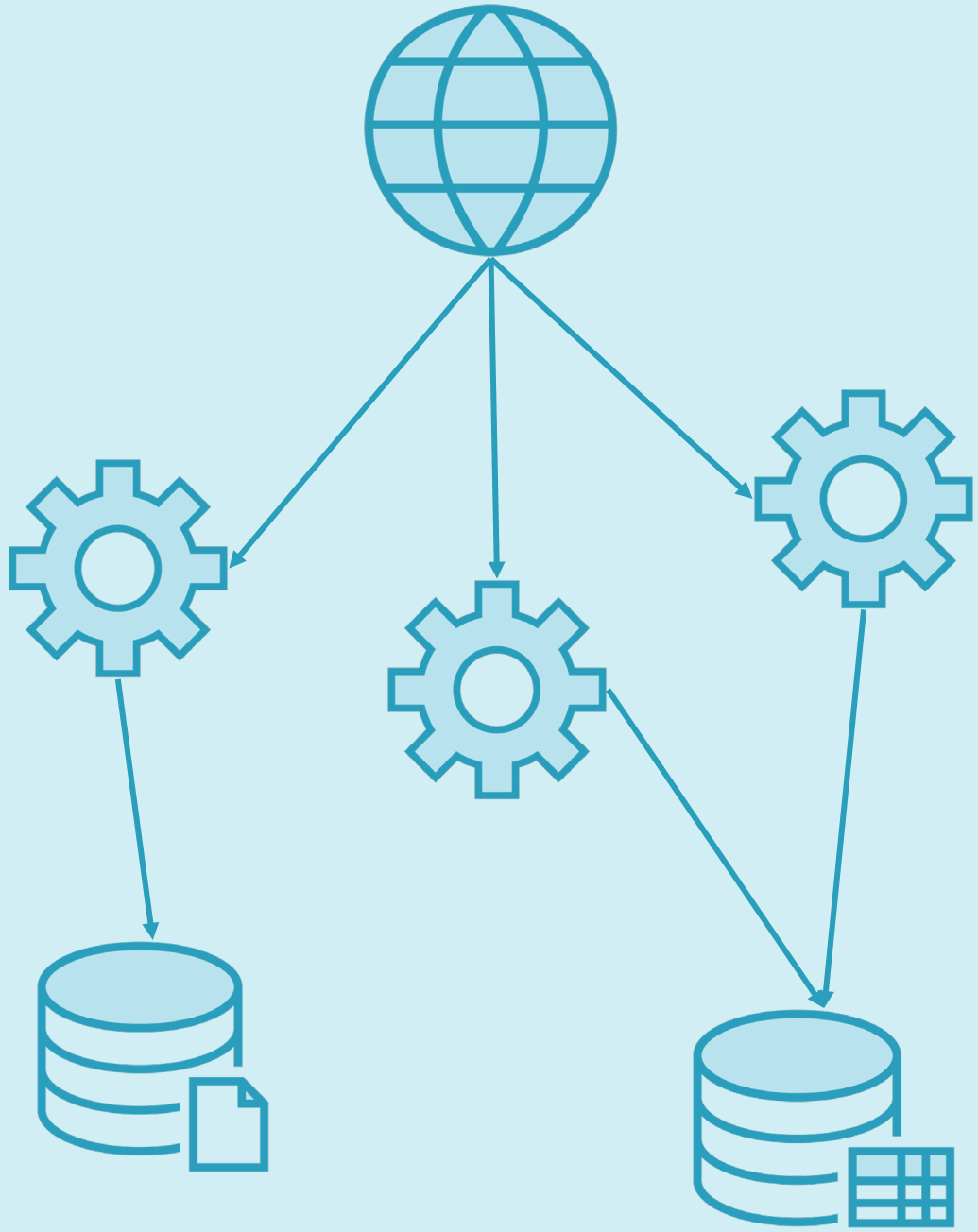


- Concurrency checks
- Data versioning
- Overwrites blocked



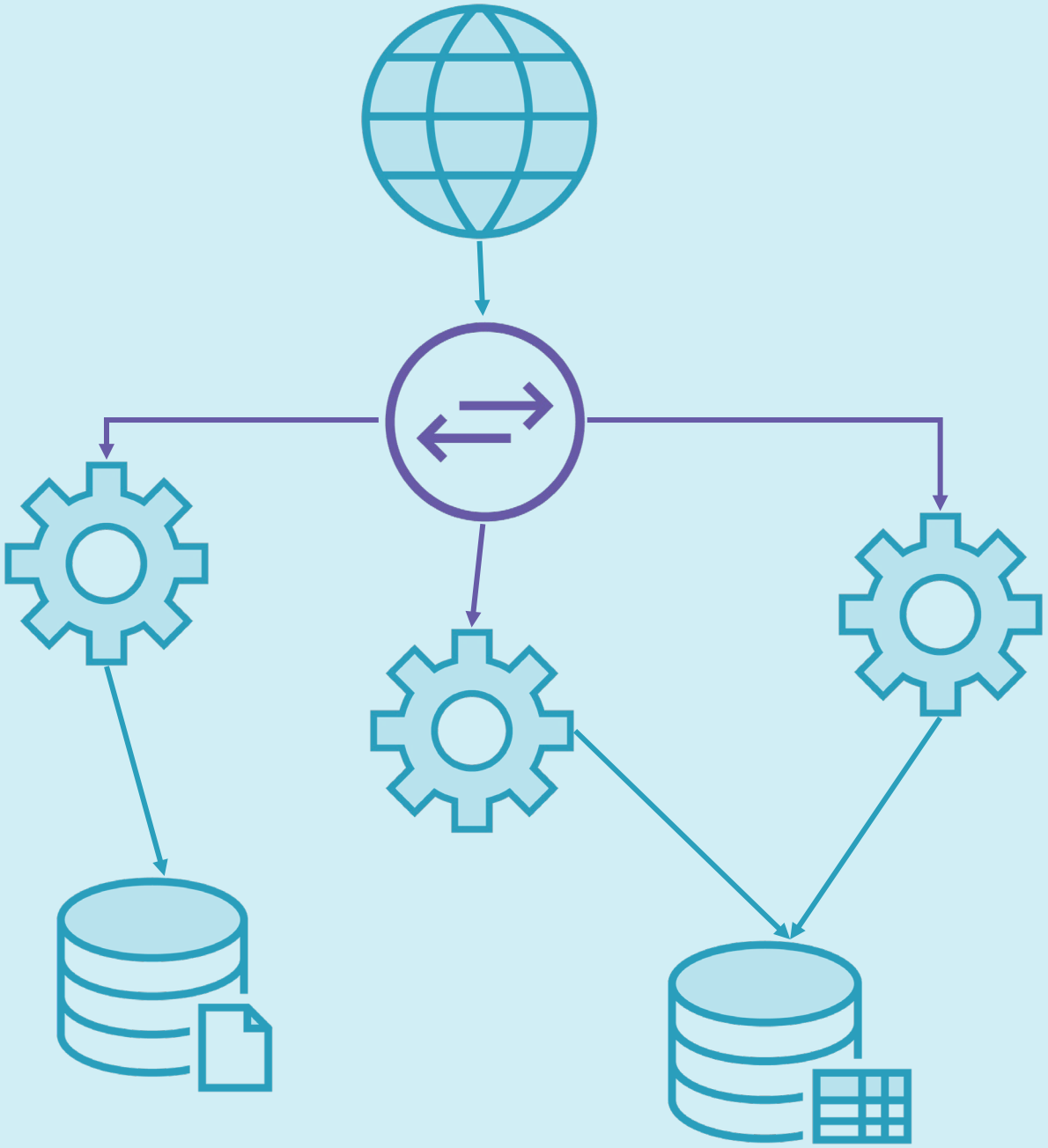
- Read-only mode
- Config switch
- UX stops writes
- Per feature

Mitigating and Preventing Overload



Reducing load

- Cache layer
- Separate config



Reducing load

- Cache layer
- Separate config

HTTP caching

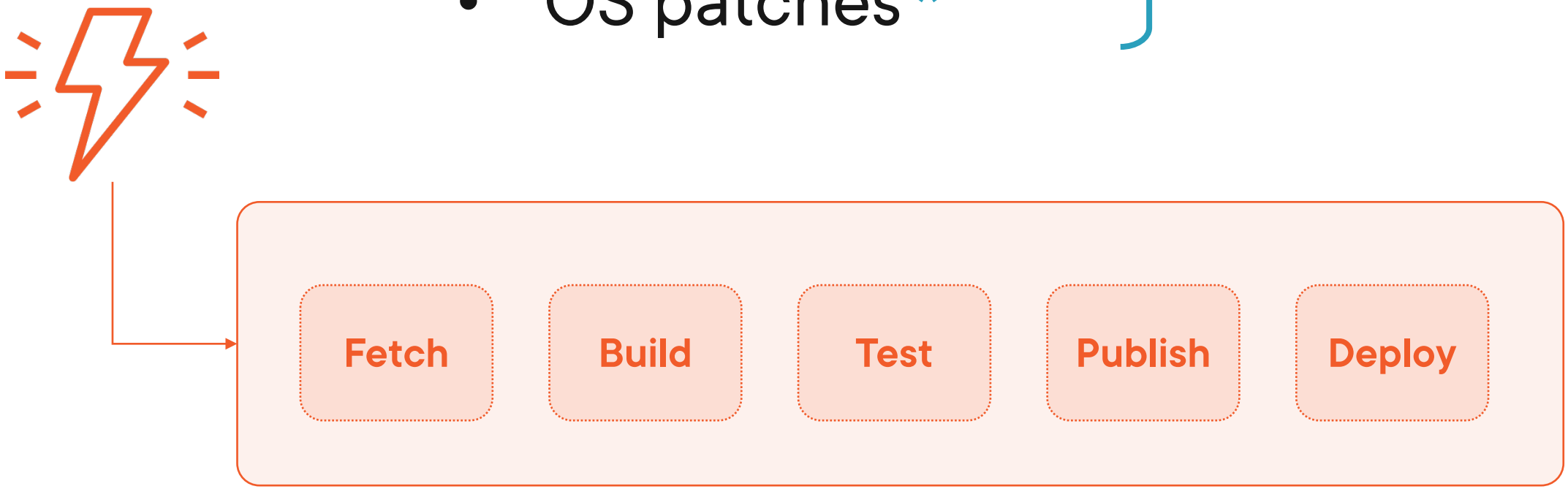
- Reverse proxy
- Stale responses

Supportability

- Disposable instances
- Graceful exits
- Safe restarts

- Code changes
- Library updates *
- OS patches *

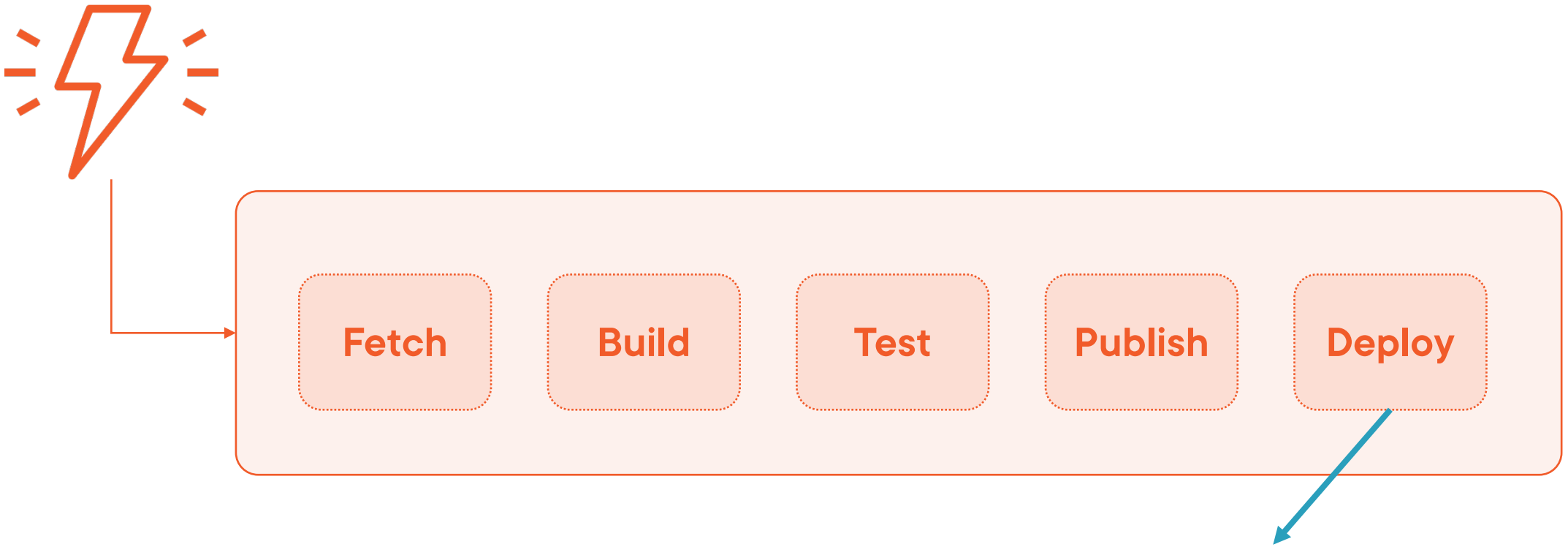
} monthly



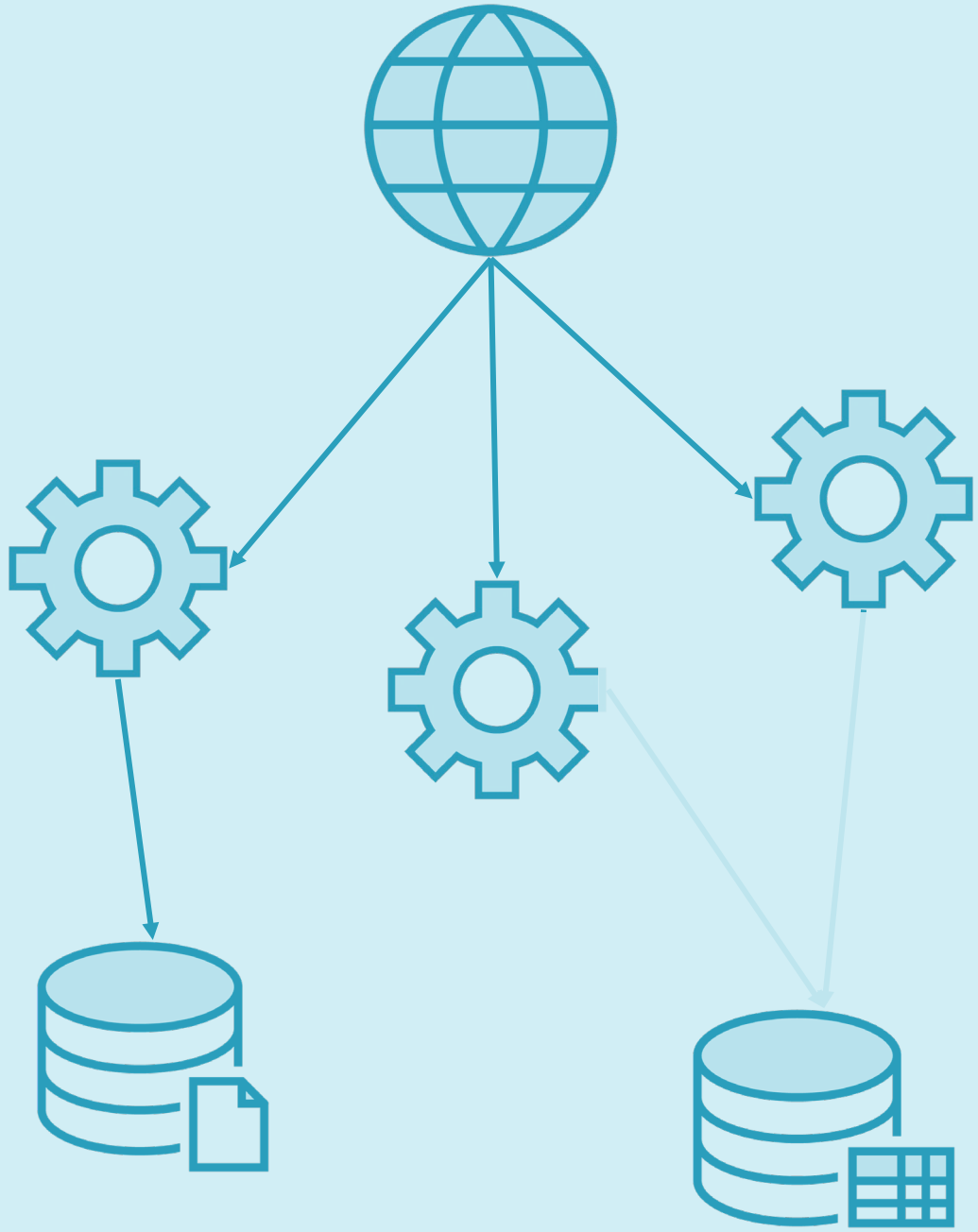
Fundamentals of CI/CD

Using Declarative Jenkins Pipelines

Elton Stoneman



- Rolling update
- Start *x* new instances
- Stop *y* old instances
- Repeat



Read-only mode

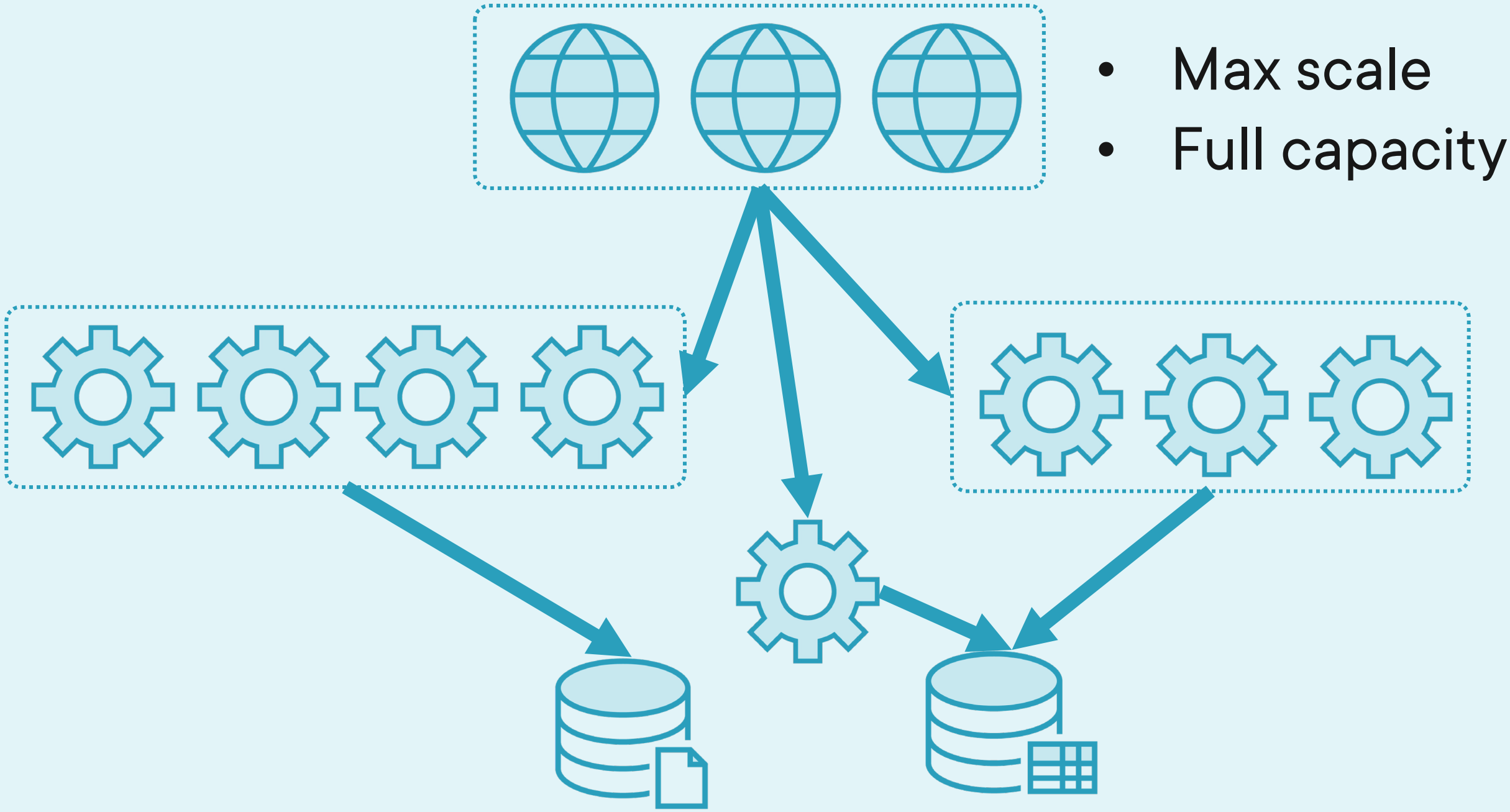
- Emergency option
- Fast mitigation

Data protection

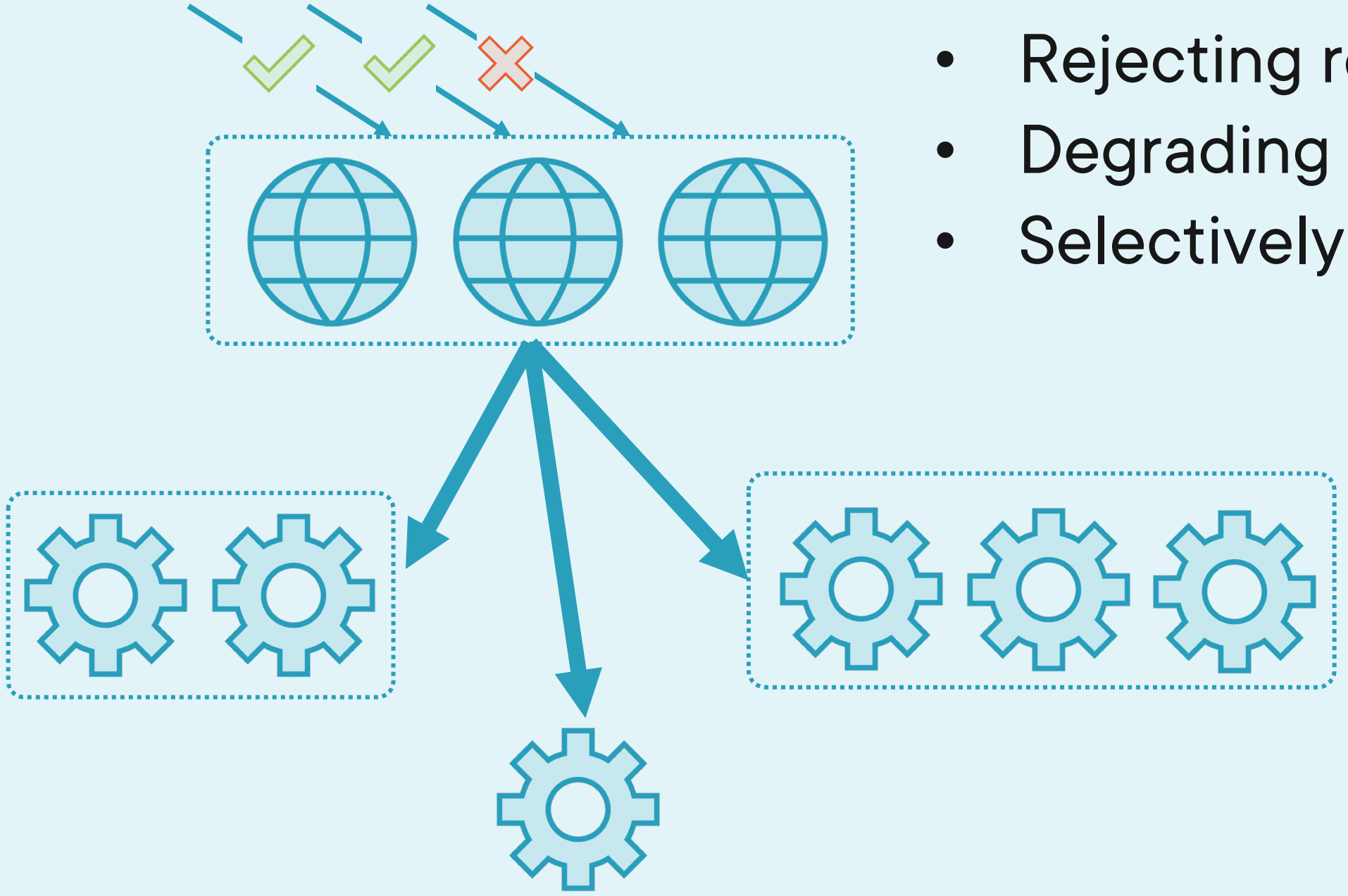
- Concurrency failsafe
- Rogue update

Partitioned by...

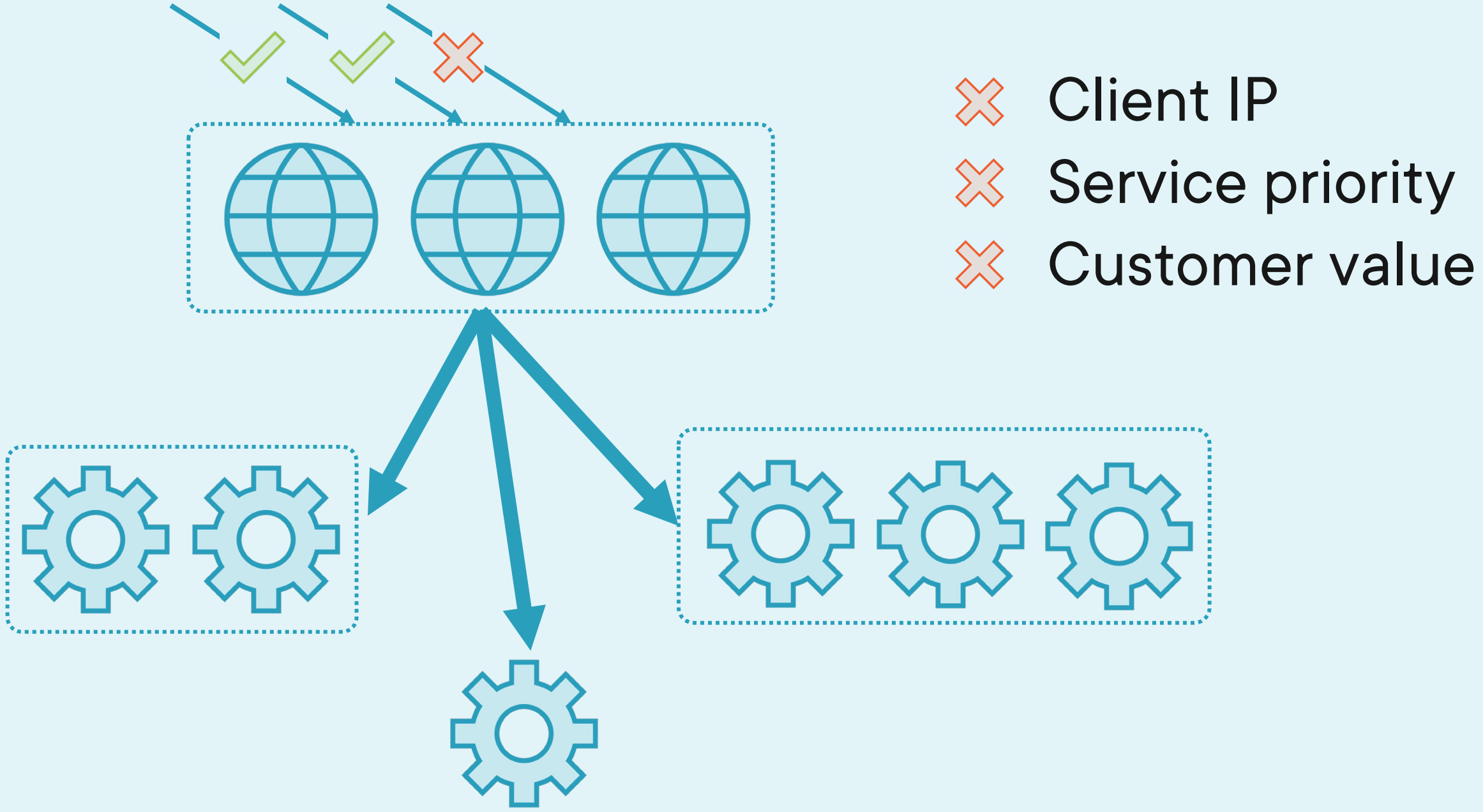
- Customer
- Feature
- Region



- Max scale
- Full capacity

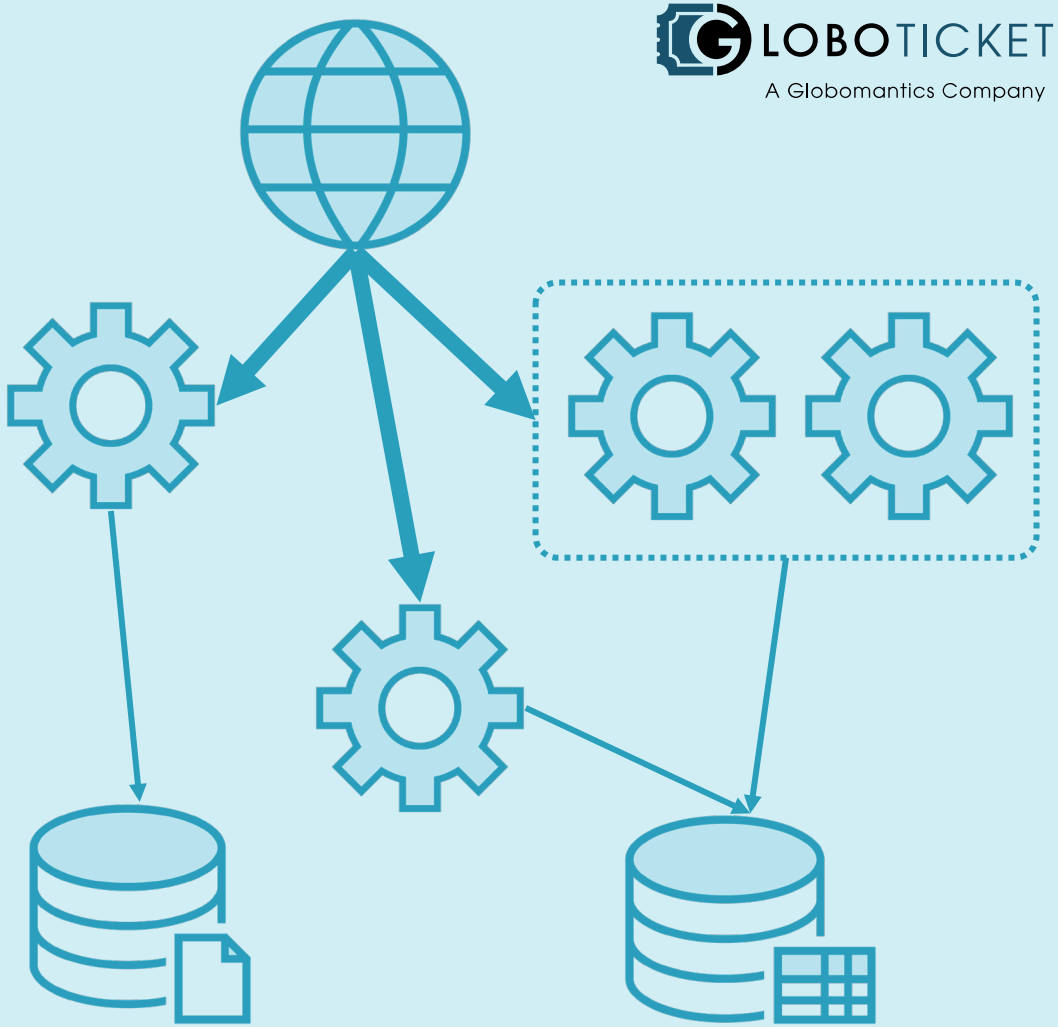


- Rejecting requests
- Degrading performance
- Selectively

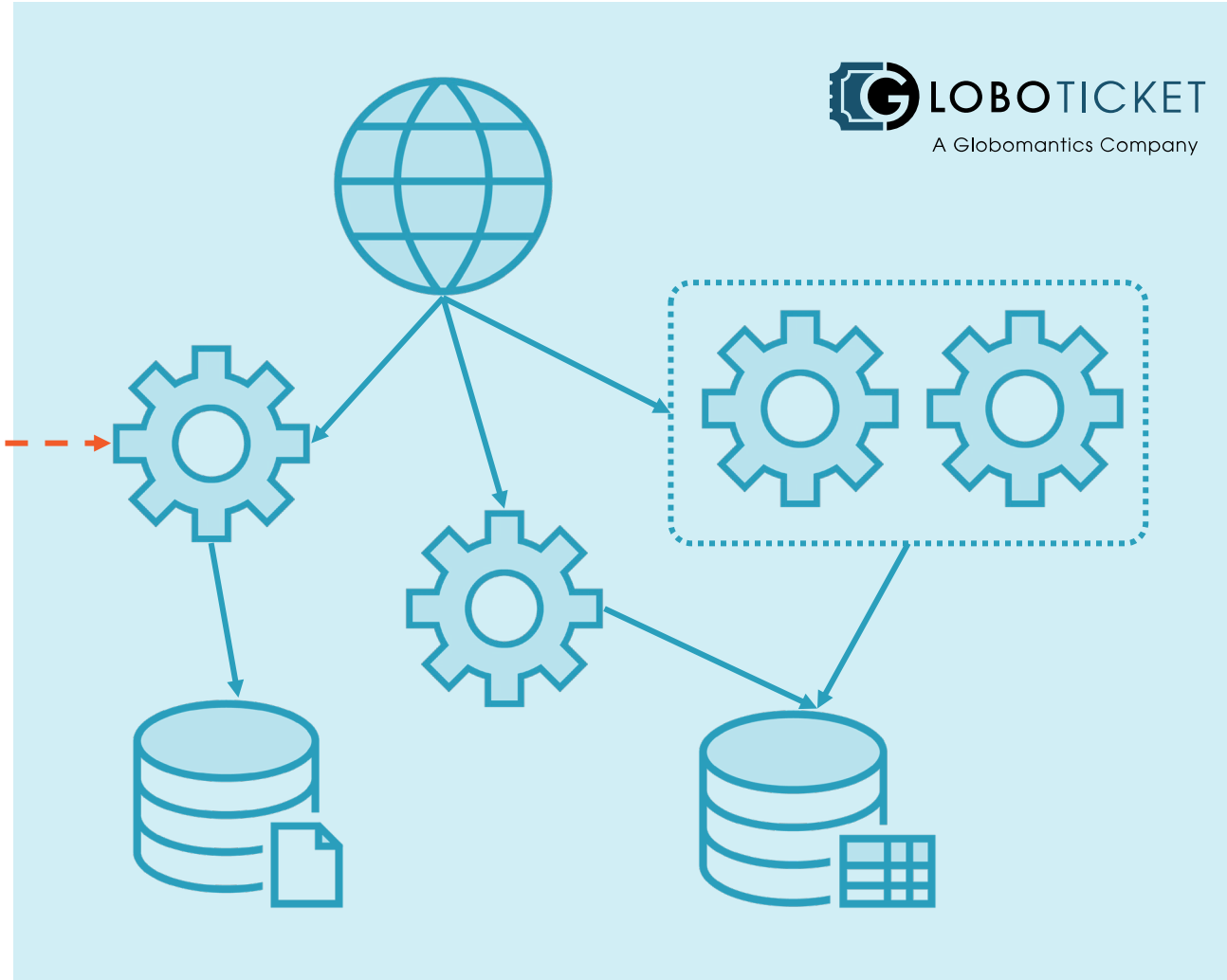

Client IP
Service priority
Customer value

Scenario: Throttling to Prevent Overload

At max capacity, can we prioritize to prevent overload?



- Service criticality
- Checked under load
- Low priority rejected



GLOBOTICKET
A Globomantics Company

```
Services:
---
ev-list
NORMAL
---
ev-details
SHEDDABLE
```

```
Services:
---
add-basket
NORMAL
---
checkout
CRITICAL
```

Criticality

- Service metadata
- Priority level
- Critical -> sheddable

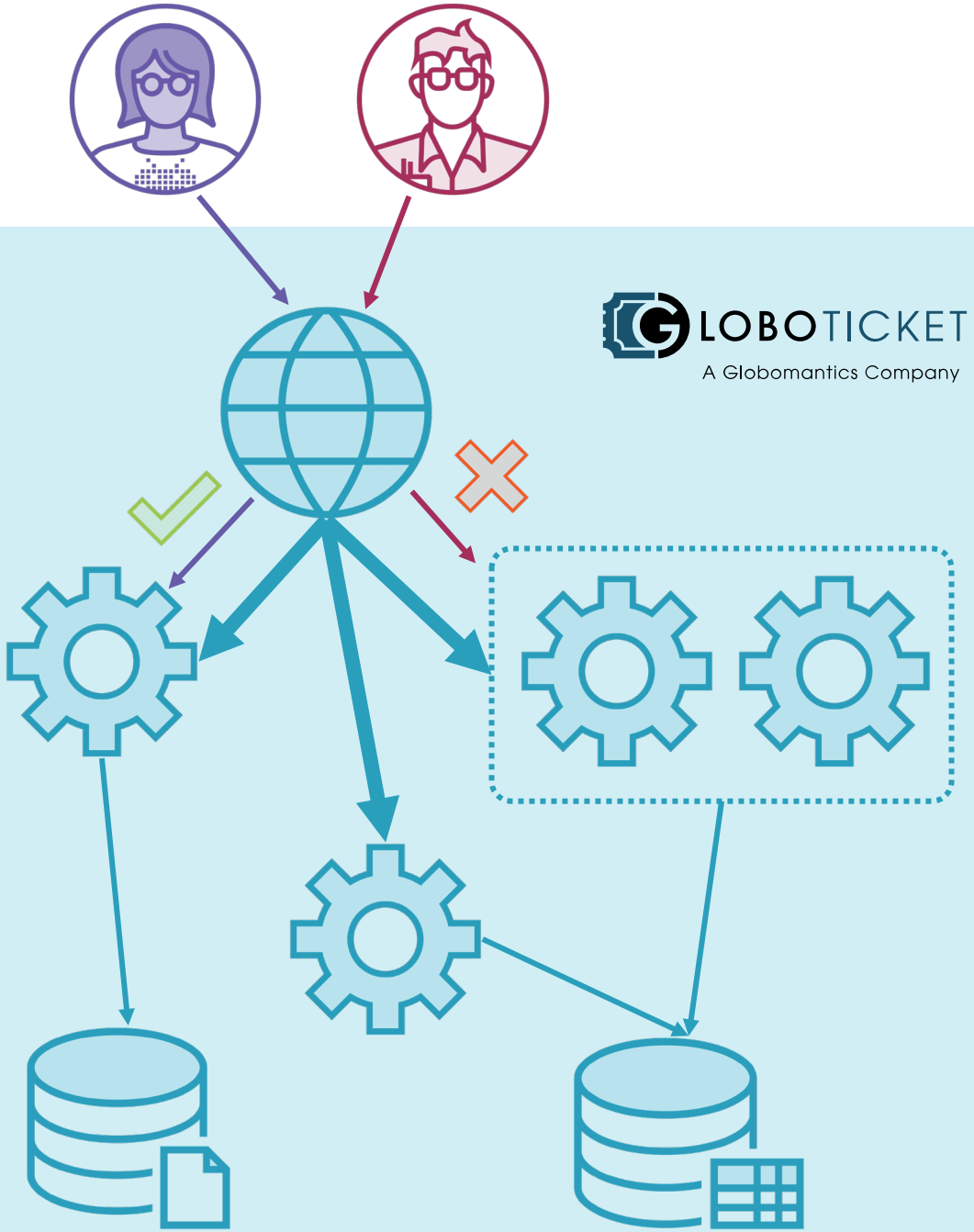
System performance

- All services
- Only critical

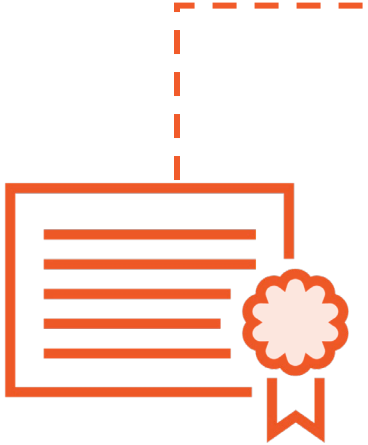
Manual mitigation

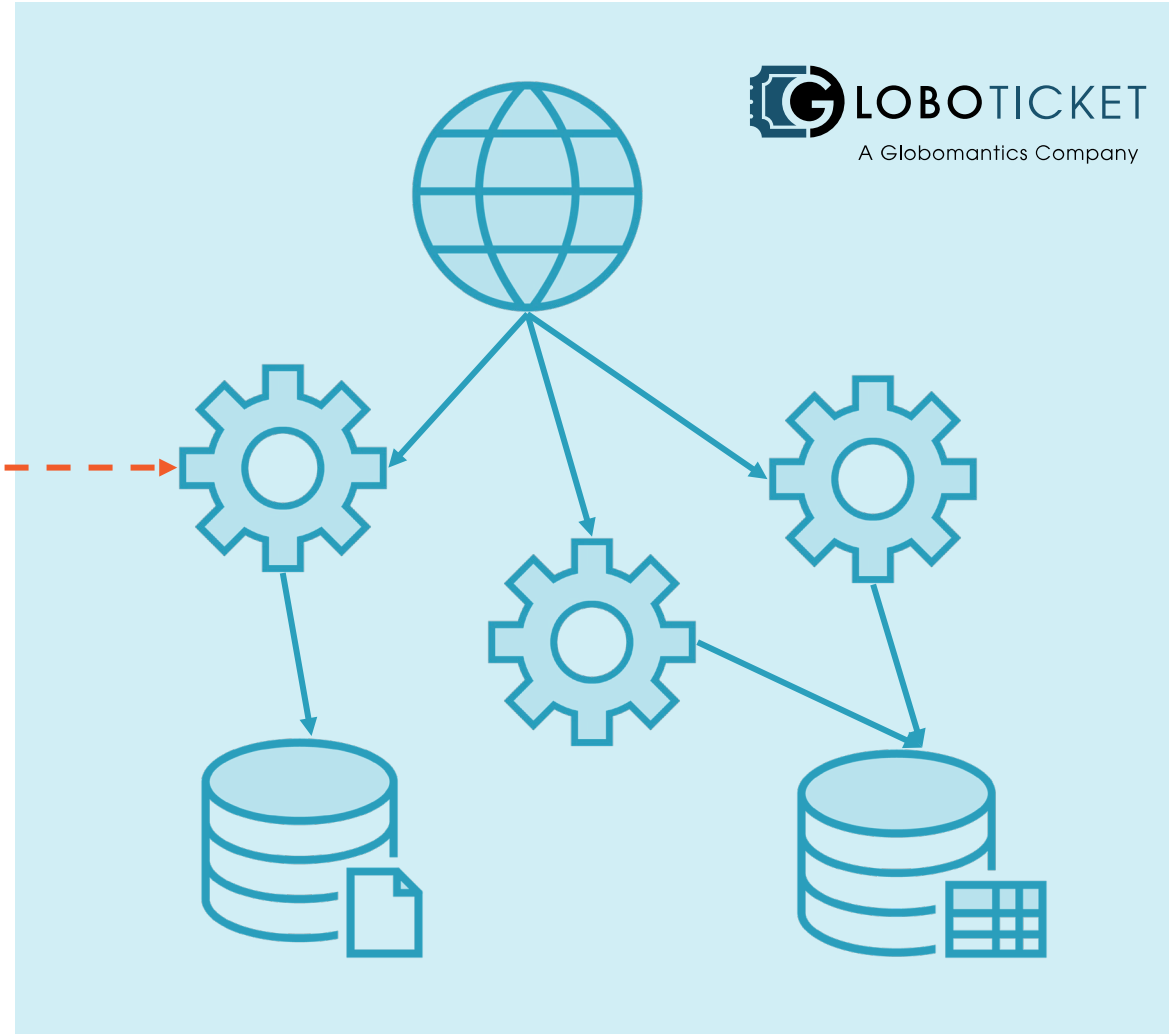
- Reject sheddable
- Proxy layer

At max capacity, can we prioritize certain customers?

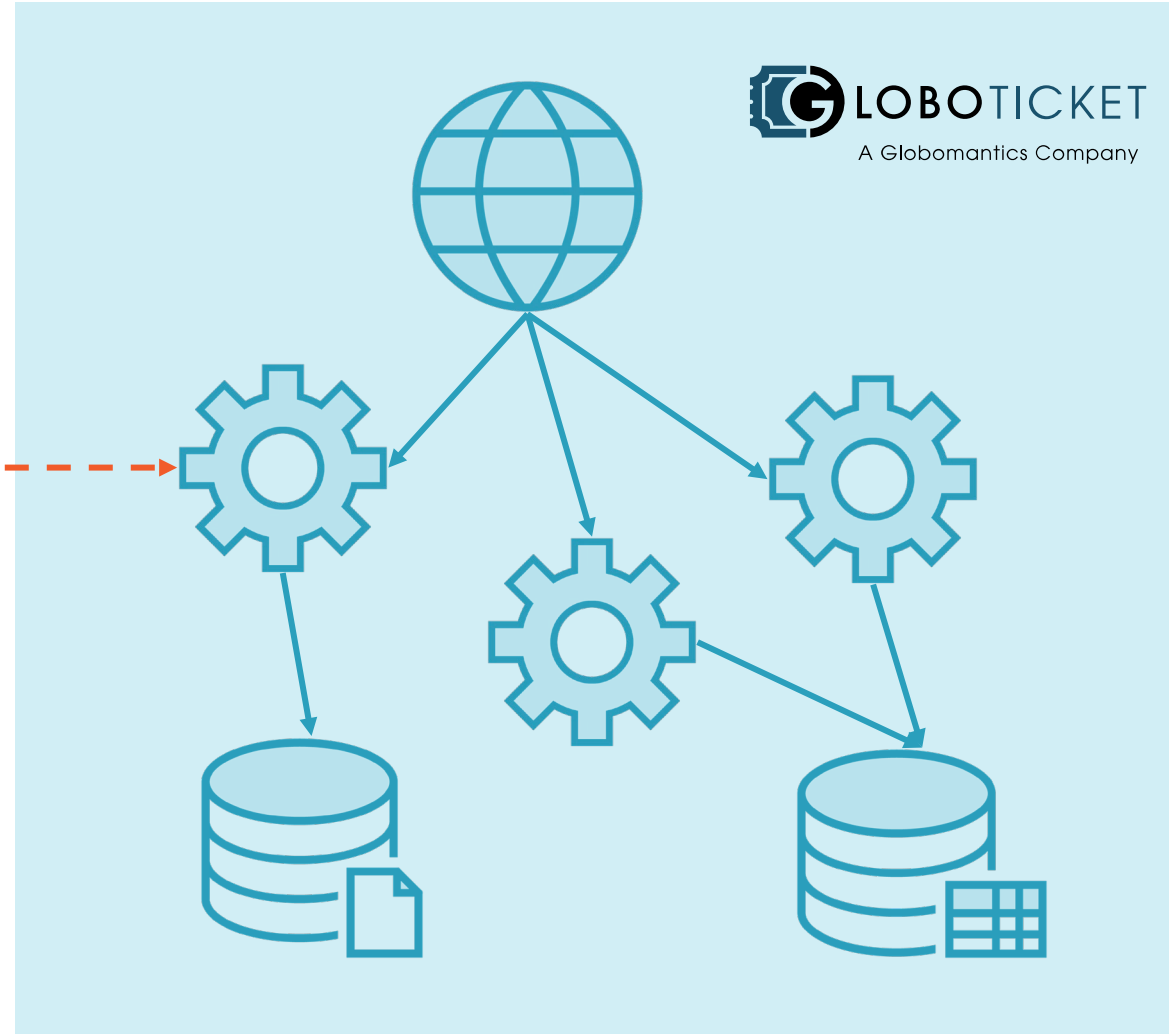


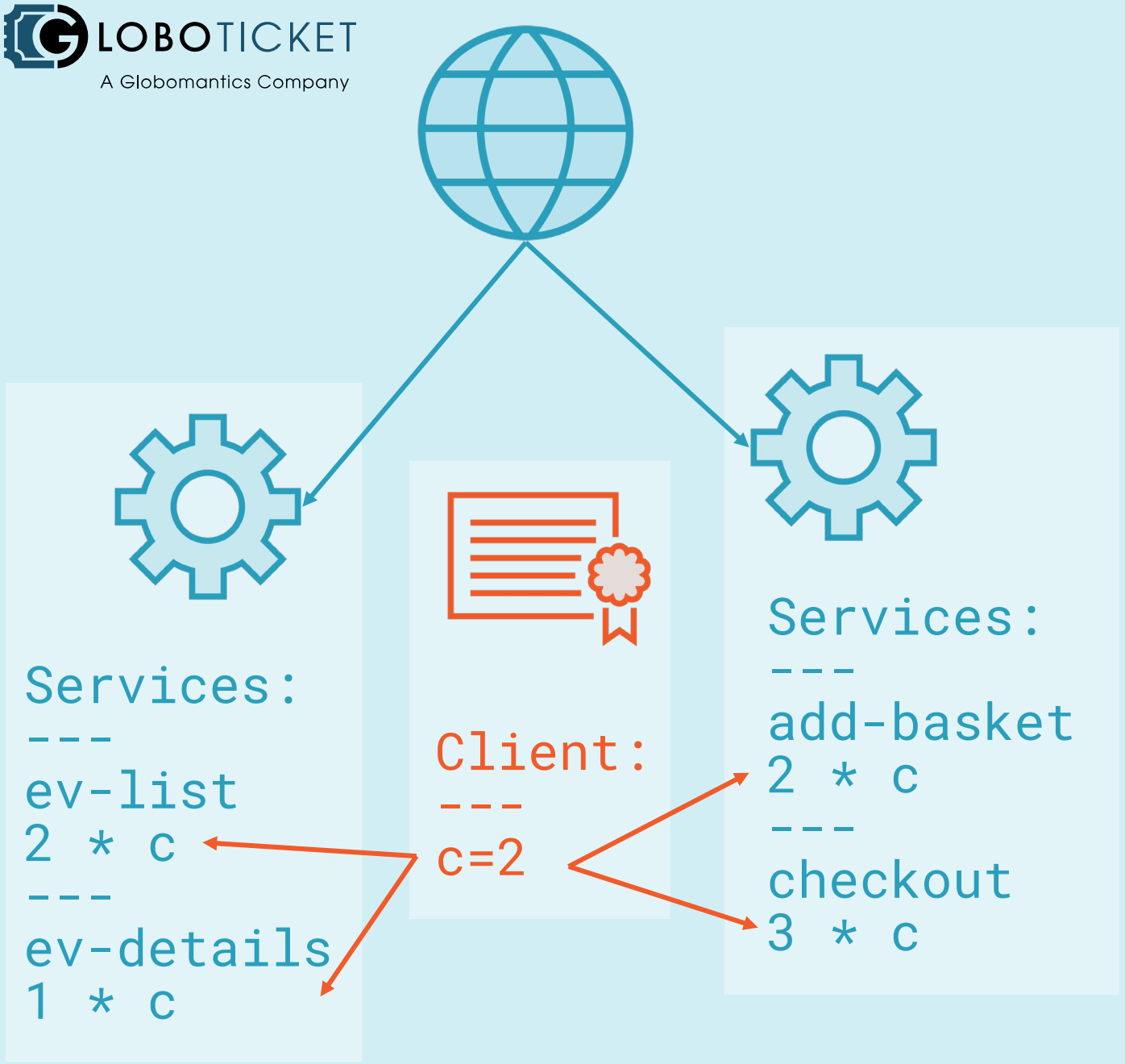
- Auth token
- Flows to API calls
- Contains customer info





- Criticality lookup
- Minimal impact
- Add claim to token





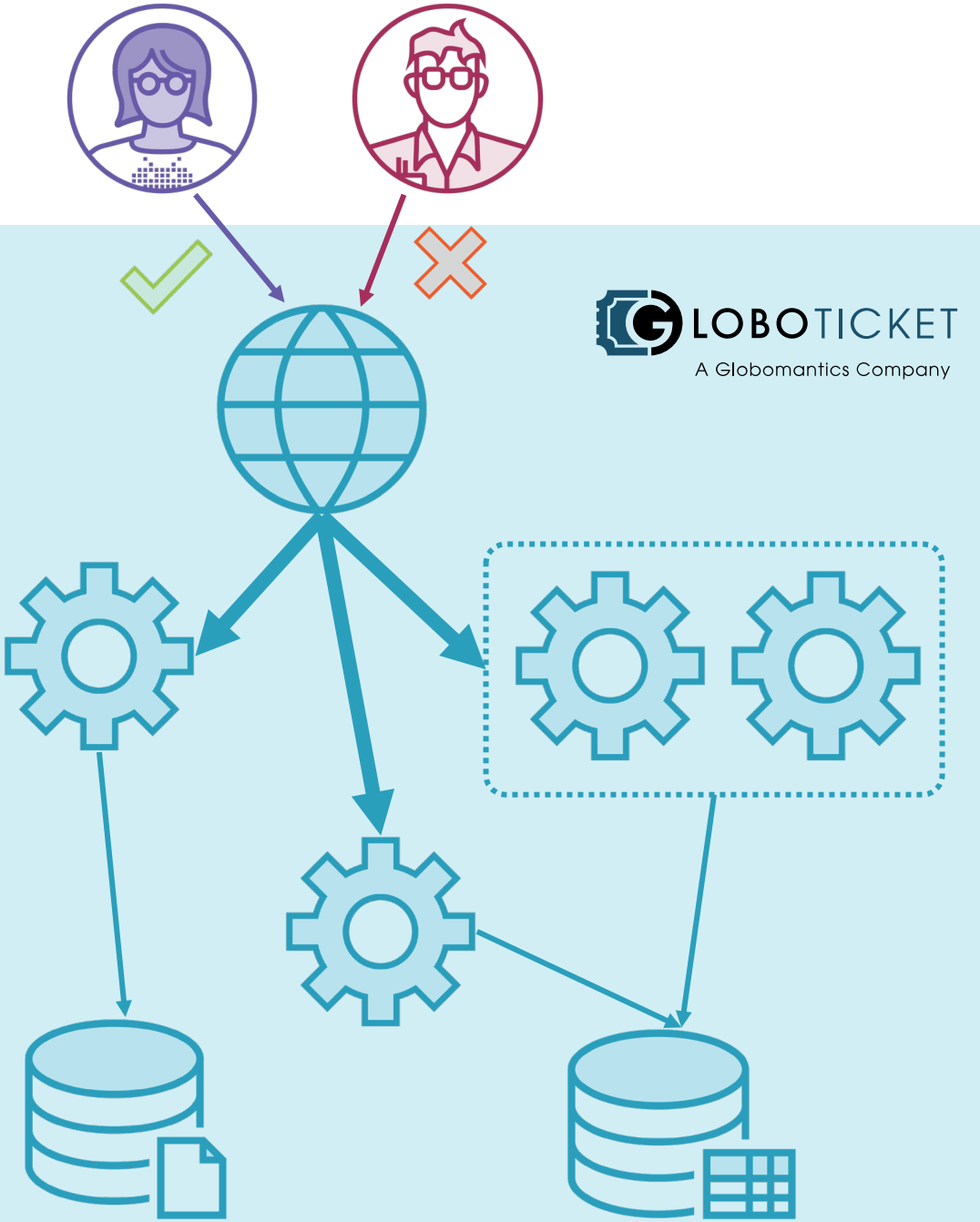
Criticality

- Service metadata
- Priority number
- Lower -> sheddable

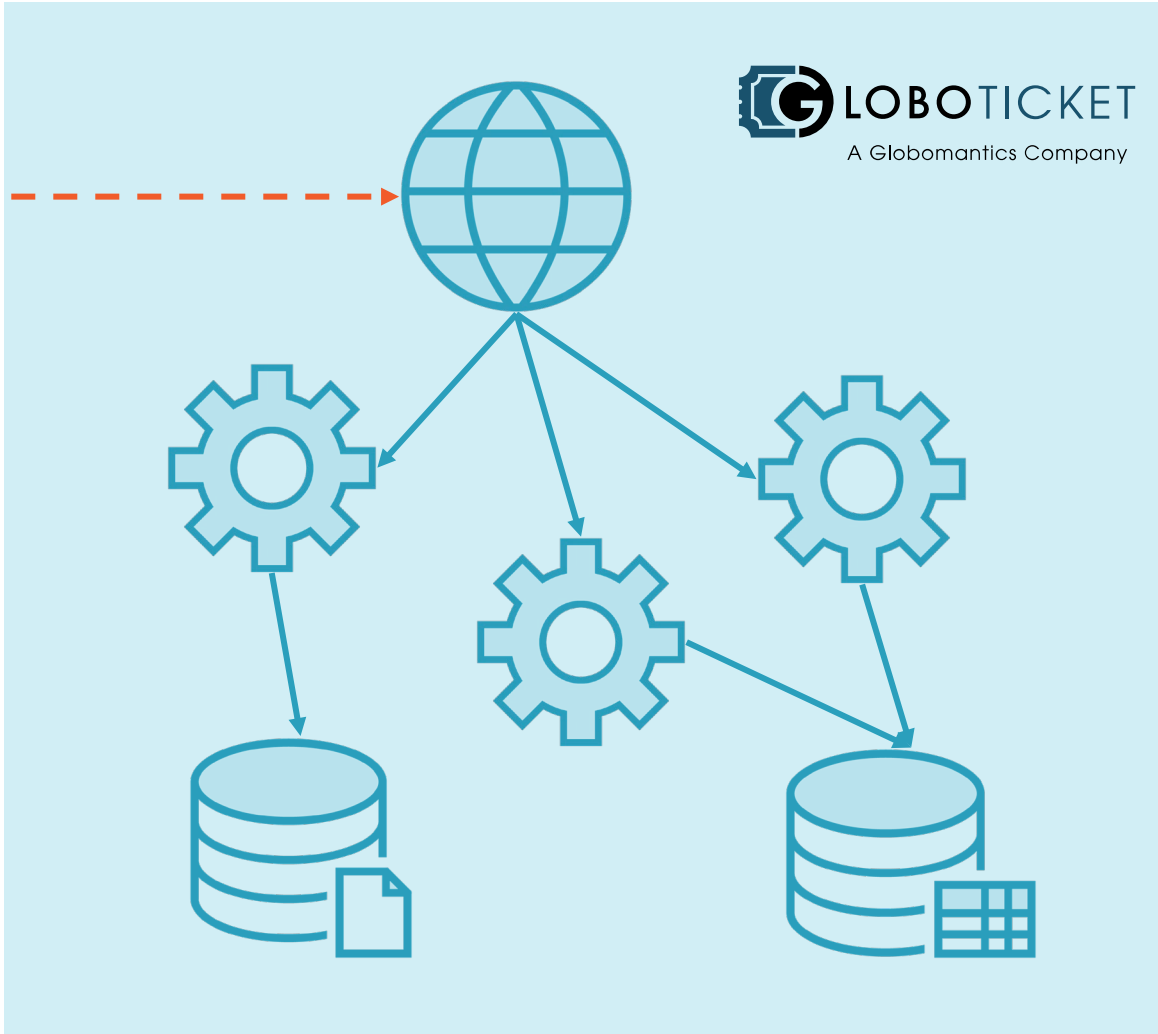
Customer value

- Criticality factor
- Multiply priority
- Sheddable upgraded

At max capacity, can we throttle incoming requests?

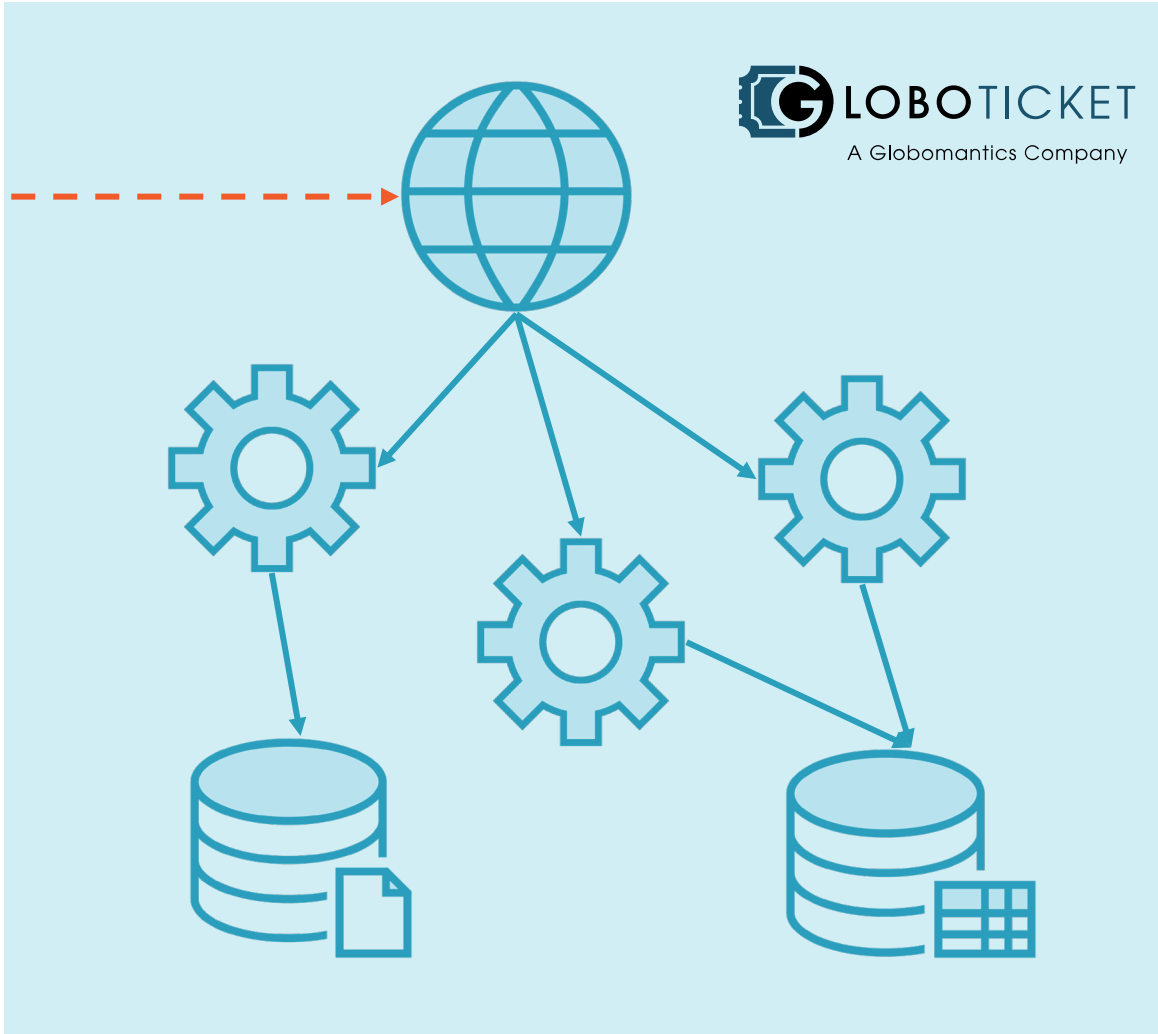


- Client-side rate limiting
- Used to stop automation
- Quotas not enforced

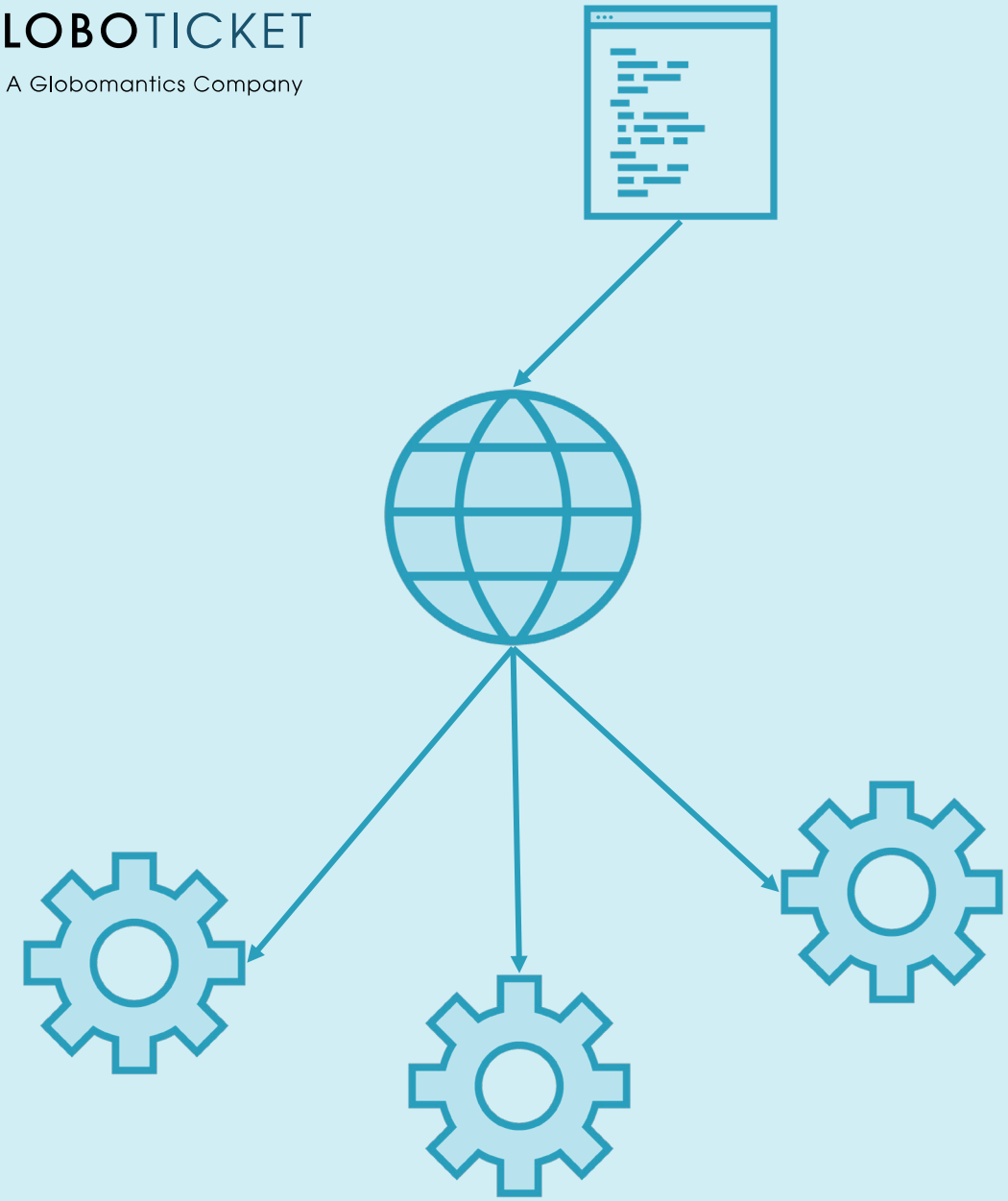


- Big customers:
- No throttling

- Others:
- Client throttling



GLOBOTICKET
A Globomantics Company



Client throttling

- Track reject rate
- Service 503s
- Client mimics 503

Degraded service

- Some customers
- Reduces load

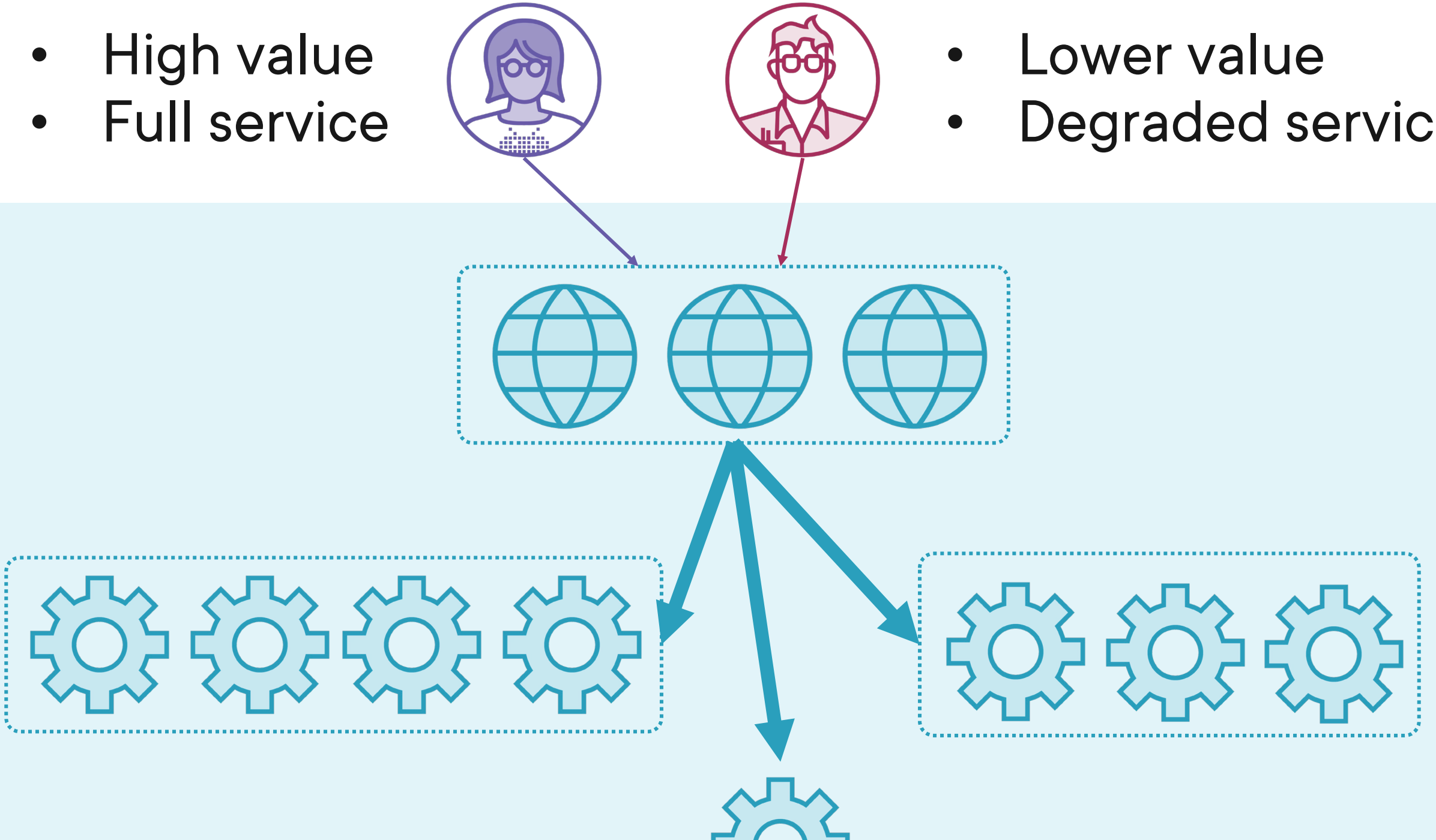
Automatic repair

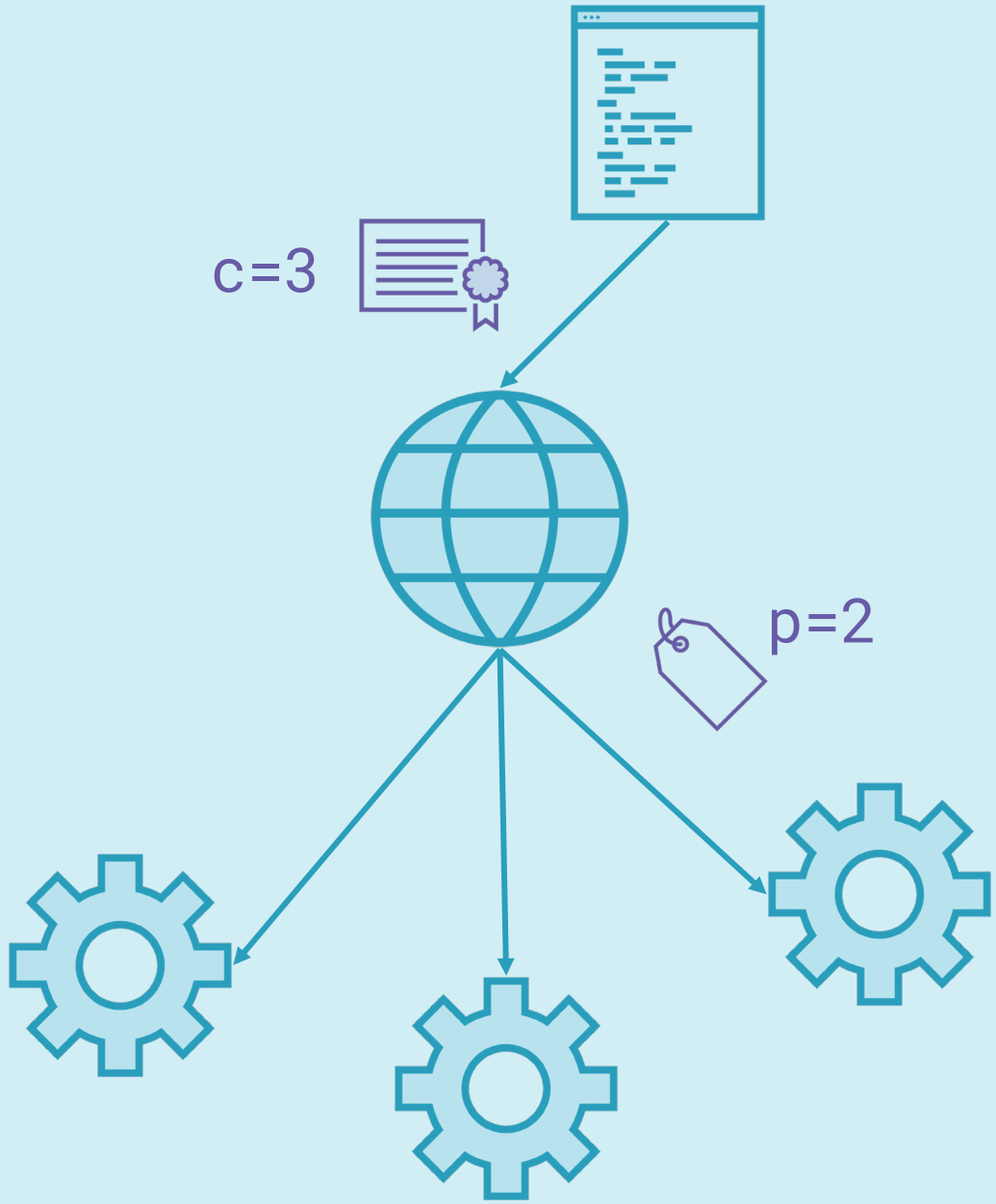
- Track success rate
- Reduce throttling

Module Summary

- High value
- Full service

- Lower value
- Degraded service





Service criticality

- Low-impact check
- HTTP header
- Proxy rejection

Client-side throttling

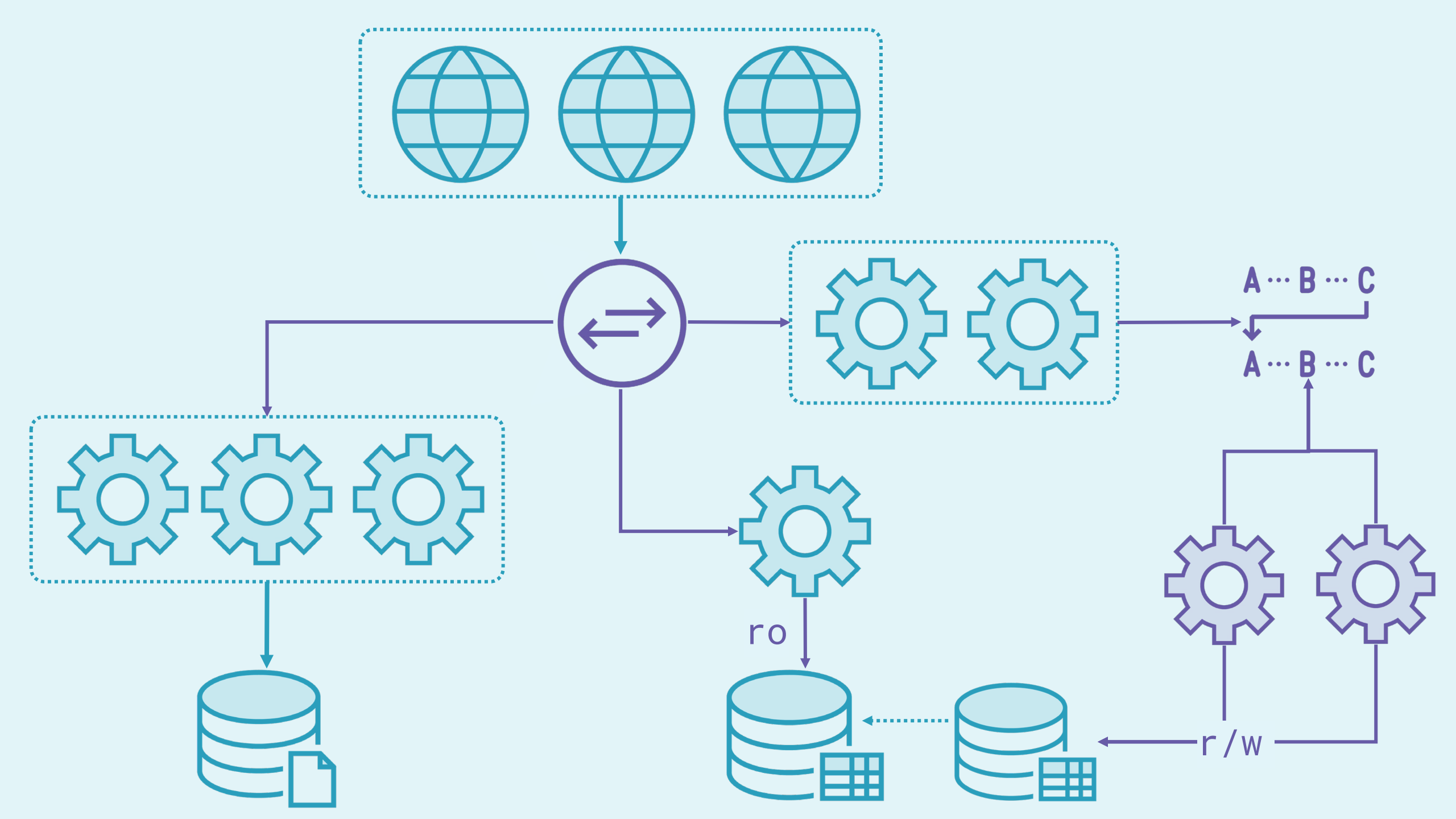
- Customer priority
- Cached value
- Speed over accuracy

Designing for Reliability

Automatic scale

Managing overload

Degrading performance


A ··· B ··· C
A ··· B ··· C
ro
r/w

Reliability Design and SRE

Design for SRE

- **Self-healing apps**
- **Fast mitigation**

Failure scenario walk-throughs

- **Drive out design gaps**
- **Identify SLOs and SLIs**
- **Build out backlogs**

Up Next:
Designing Observability for Fault Diagnosis