Brian Loomis

Director Of Architecture
at Progress

Surviving the Real-World: Building Resilient Cloud-Native Platforms for Intermittent Networks

Conf42 Golang

April 3 2025 • Online

CONF42

RELIABLE SYSTEMS BASED ON UNRELIABLE NETWORKS

The Eight Fallacies of Distributed Computing

Peter Deutsch

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

For more details, read the article by Arnon Rotem-Gal-Oz

CONF42 GOLANG 2025

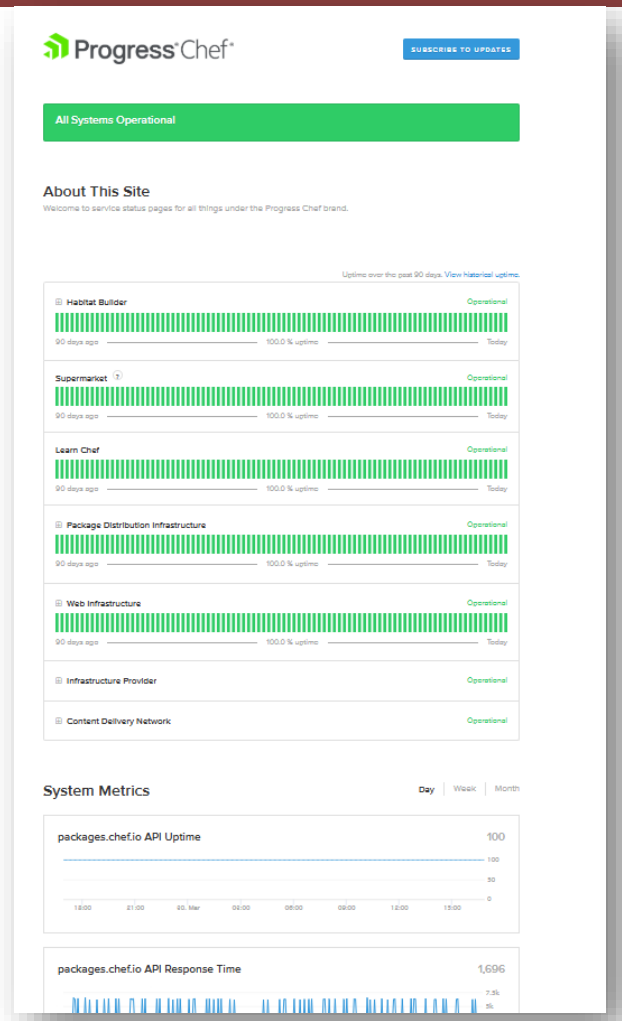
APRIL 3 • ONLINE

AGENDA

- Failure modes of distributed systems
- Service level agreements and objectives
- Defining quality aspects of availability & reliability/resilience, and scalability
- Key infrastructure and software architecture patterns
- Three examples from a simple set of microservices through to a devops solution
- How to do this in GoLang
- Testing your system
- Summary and references

DESIGN PROCESS SKETCH FOR QUALITY ATTRIBUTES

1. Start with a system diagram and business process to be implemented
2. Identify quality goals (SLA), measurements taken in the system for this business process, and failure modes (FMEA)
3. Plan specific design changes (as epics or tasks)
4. Implement infrastructure as code patterns and software patterns across the system of applications
5. Test and validate quality goals are improved
6. Continuously monitor and run automated playbooks to recover in production(operations team)
7. Iterate (RCA leads to more design…)



AVAILABILITY DEFINITION

90.0% (one nine)	36 days and 12 hours per year
99.0% (two nines)	87 hours and 46 minutes per year
99.9% (three nines)	8 hours and 46 minutes per year
99.99% (four nines)	52 minutes and 33 seconds per year
99.999% (five nines)	5 minutes and 35 seconds per year
99.9999% (six nines)	31.5 seconds per year

So how much availability is good enough? To put this question into perspective, consider the example of living in a "three nines" (99.9%) world. If everything around us were "three nines", the world would look like this:

- There would be a 99.9% turnout of all registered voters in an election
- You would have one rainy day every three years
- If you made 10 phone calls a day you would only have 3 dropped calls a year
- If you used your Windows PC 40 hours a week, you would only have to reboot it once every two weeks (once a year for a Mac!)

That sounds great, doesn't it? However, maybe a "three nines" world wouldn't be so great:

- The U.S. Postal Service would lose 2000 pieces of mail *each hour*
- 20,000 prescription errors would be made each year
- There would be 500 incorrect surgical procedures *per week*

$$Availability_{in-store} = \frac{Uptime}{(Uptime + Downtime)} = \frac{Uptime}{(Uptime + \sum_1^N downtimes)} = \frac{MTBF}{(MTBF + MTTR)}$$

WHAT IS AN SLA OR SLO?

- Service level agreement – typically a contractual document between service provider and customer(s) on how the service will be operated and what happens if these quantitative metrics are not met
- Service level objective – an organization-internal set of targets typically from an IT organization or cloud operations team; these often contribute key elements to the SLA for a SaaS product
- Not all systems need or have an SLA → this is business specific (more $$ or more risk involved may require this)

- An example may combine availability and reliability measures with caveats:
  - The customer should have an ability to submit DevOps jobs for 99.99% of the operating period
  - The customer jobs should complete successfully or with identified error in 100% of submissions; the identified error can be either customer inputs were invalid or required customer assets were not available, so job terminated. If the system crashes for internal reasons, no charge will be made to the customer.

A SAMPLE RETAIL STORE UPTIME SLA

- The critical goal of the point-of-sale system is that a store is able to maintain **100% uptime** in any situation other than a catastrophic event. It is assumed that point-of-sale could not reasonably overcome a catastrophic event to be able to keep the store taking, making and delivering customer's orders.
- The payment subsystem (applications talking to banks) has a target **of 99.999% uptime** for a given operational period

- Key Definitions:
  - Store Uptime - a store is defined as up and running if the store is able to take, make and deliver a customer's order. The store should not be delayed by any system for more than 1 minute taking, making or delivering an order to a customer. A store may be in a connected or disconnected state and still take, make and deliver a customer's order.
  - Disconnected Store - A store that cannot connect to the internet. The store is still able to take, make and deliver an order received from any instore channel (phone order or walk in order). Online ordering channels would be unavailable to this store. Menus, loyal customer discounts, and online ordering would be restored with eCommerce recovery
  - Catastrophic Event - any disruption to business continuity in an manner that either could not be reasonably anticipated or that is a disruption in scope that cannot be mitigated by the IT systems. An example would be more than one critical components in store infrastructure failing at the same time (a non-redundant switch or both clusters). An example of a disruption in scope beyond IT systems would be a total store power outage or physical damage to the store that prevents operation like a fire

LOOSE DEFINITION OF RESILIENCE

There is no quantitative definition of **resilience**, but generally, it is the ability of a system to withstand and recover from failures, disruptions, or threats, ensuring the availability, reliability, and performance of services and quickly recover to ensure uninterrupted service delivery to end-users.

1. Detect a failure has occurred
2. Report the failure (to an SRE or a dashboard)
3. Determine why it occurred or the scope of the failure
4. Adapt and recover

RELIABILITY

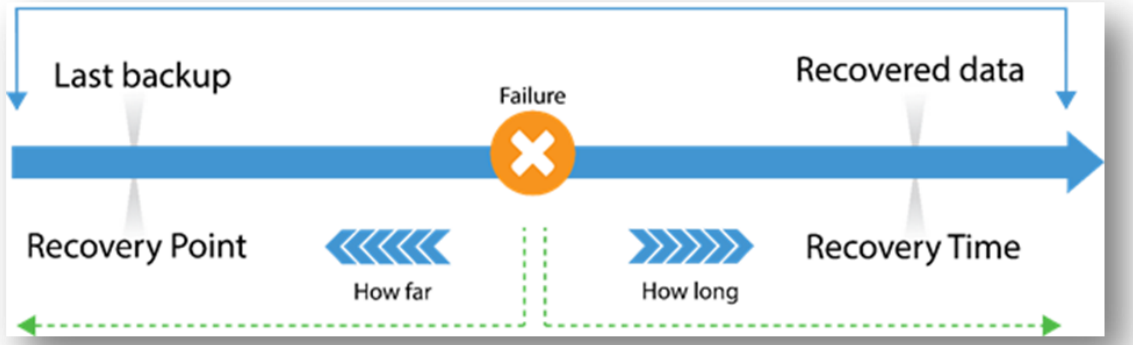
- **Reliability** measures the probability that a system, product, or service's actual behavior matches the expected behavior (correct results for inputs over a period of time – in other words, *it does what it says* (or what we expect) it to do…
  - No errors that cause crashes, no dependency issues that cause crashes or partial results, we might allow for eventual consistency (if delays cause the system to report outputs later than normal)

- SRE and service managers in ITIL may report on:
  - Mean Time to Failure (MTTF) = the average time a system or component takes to fail for non-repairable systems (wear-and-tear and hardware with lifecycle)
  - Mean Time Between Failures (MTBF) = the average time between failures for repairable systems (like software where we have outages and RTO)

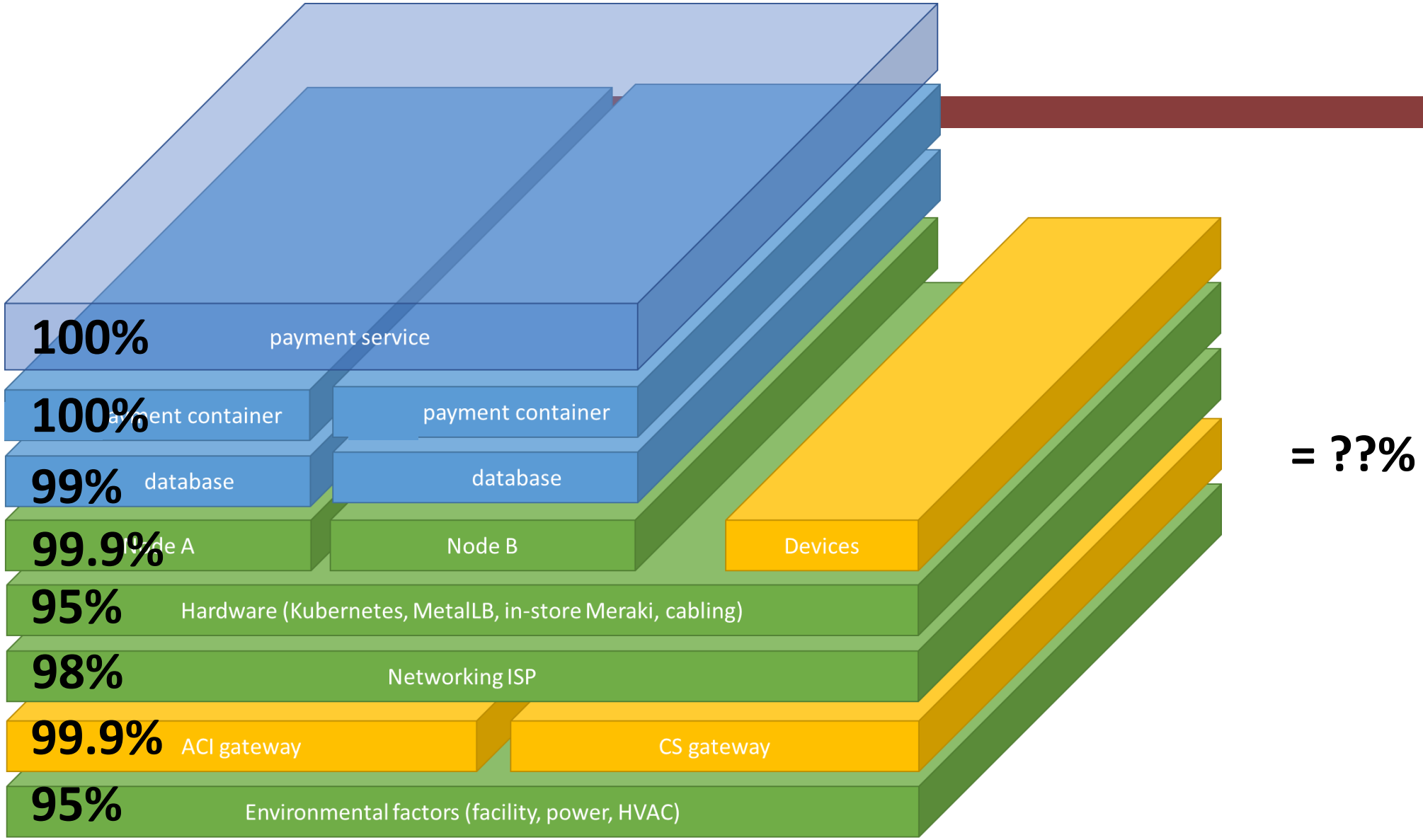
What is system reliability?

Reliability is the probability that a system or component consistently performs its intended function without failure over a specified period. Teams must understand how to measure and ensure reliability to make informed decisions about system performance and enhance customer satisfaction.

For instance, payroll systems must reliably process direct deposits within a set timeframe each month, while cold storage systems must detect power outages and switch to backup generators without fail. Across industries, maintaining reliability in automated processes and tracking performance



CALCULATING FOR A COMPLETE ENVIRONMENT



= ??%

FMEA HELPS BRAINSTORM ABOUT POSSIBLE FAILURES

- A failure might happen if…
  - A container cannot reach in-store devices… (switch down)
  - A container cannot reach bank… (internet or payment gateway to that bank down)
  - Configuration update or container maintenance at wrong time
  - A container is actually down… (the service cannot restart or failover does not work)
  - Hardware platform in-store is down
  - Service is degraded by devices offline…
  - (Cash drawer or printer issues)

- And we might reduce the likelihood of it happening by…
  - Hardware redundancy… blue-green deployments…
- And we might reduce the impact if it happens by…

RETAIL FAILURES BY OSI LAYER

OSI Layer	Generic example(s)	Typical payment system failure scenario(s)	How do we detect?	Impact to payment availability	Avoidance	Remediation
External client		• NextGenSystem client cannot reach payment system (or is restarted) • Security JWT token invalid or security service failure • Payment requests not from NextGenSystem client received	• No inbound API calls	• None	• Shared environment health check with restart mechanism	• NextGenSystem operations runbook • Client restart requires API support to determine transactions in flight (to restart at appropriate point)
7 (App)	Out-of-sync or overlapping business operations (e.g., HL7), race conditions, logic bug, performance of shared resources	• Different client completes payment than originating client • Split payments scenarios not adding up, order past store limit (or tip) • Payment requests out of order (finalize before auth) • No gateway is reachable, one gateway too much traffic by business rule • Primary gateway is no longer reachable (or responding slowly) • Device soft failure – appears to process • Logic bugs (handle with top-level error handling & alerting, retry) • Internal performance degradation • Multiple requests (same payment), badly formed requests (missing info), invalid CC number/PIN • Fraudulent request (bad credit card number) • Batch closed late (with too many transactions or mismatched & cannot reconcile automatically)			• Multiple gateway providers (circuit breaker on each, plus failover URLs, plus retry) • Devices protected by Polly/retry logic (should add CB) • Data loader values checked for consistency/breaking changes (LINT in JenkinsX) – scripts to move between versions of schema	• Incomplete transactions caught through daily batch operations • Manual resolution process required on complex batch irreconciliation • Critical bug fix (requiring new development) – redeploy via Octopus into stores as new container
6 (Presentation)	JSON packets mis-formed/MIME, API calls timeout, SSL, firewall rules blocking	• One gateway endpoint not reachable for short/long duration of time (possibly with in-flight transaction) • One device not available for short/long duration of time (possibly with in-flight transaction) • Firewall rule misconfigured			• Store & forward pattern (persisted checkpoint locally in case of need to restart) • Multiple gateway URL's (round-robin strategy)	• Manual replay of transactions (force SAF, manually gate resubmission of transactions) • Implement RTO or

OSI Layer	Generic example(s)	Typical payment system failure scenario(s)	How do we detect?	Impact to payment availability	Avoidance	Remediation
Container infrastructure	Container build issue (wrong .NET FX, missing DLL, packaging incompatibility with base container)	• Soft failure not caught by health check • Misconfiguration (appsettings or DB settings) • Expired passwords (database, gateway) • Gateway certificates expired	• K8S health check for components /DLLs, connectivity) • Global dashboard	• Potentially small time to redeploy (failover time)	• Round-robin to next available container endpoint	• Restart container (automatic or manual)
Compute infrastructure	Hardware error (bad drive), or hardware out of performance spec (memory, CPU, disk space)	• Error in deployed hardware • Resources not available (at limit) – CPU, memory, disk such as logs/DB full • Load balancer/ingress degraded • Inability to deploy in-store environment (bad helm chart, connectivity to CD resources)	• Initiated failover to backup	• Recreate payments in flight RPO • Availability impact for failover time (outage)	• Redundant hardware for hosting K8S, database and other containers • Scale-out containers on multiple LB's • Log-shipping database transactions to DW	• Manually reprovision or failover to cloud-hosted environment
5 (Session)	RPC's, sockets, stdout streams	• Change in container definition file, CI process	• Logs not available to Splunk			
4 (Transport)	TCP/IP packets out of order or TLS failure/IPSec	• TLS configuration change with gateway provider				
3 (Network)	Routing or general network failure, ICMP/ping	• Internal network not available between containers (through bad config or other) • Spanning tree or inability to failover routing • Database not reachable			• Database protected by .NET core best practice resiliency methods in EFCore (need Couchbase CB, retry)	
2 (Data)	Ethernet, WiFi unavailable or insecure	• Devices not reachable or do not authenticate or have bad tokens for gateways				
1 (Physical)	Fiber cut to store or natural disaster (store unavailable)	• Store connectivity not available or degraded • Environmental factors (building intactness, power, cooling, etc.)				

HOW TO APPLY TECHNIQUES

- What techniques can I use to:
  - Identify that a failure has occurred?
  - Checkpoint my work so that I can rollback to a known state?
  - Avoid the failure in the future?
  - Retry the failed operation (in the case of transient faults)?
  - Mitigate the failure (reduce or mask the failure's impact)?
  - Improve the recovery time (increase availability by moving the RTO as close after to the failure as possible)?

- We will apply **both** infrastructure and software-based resiliency techniques such as redundancy, detection, failover and isolation

KEY INFRASTRUCTURE PATTERNS

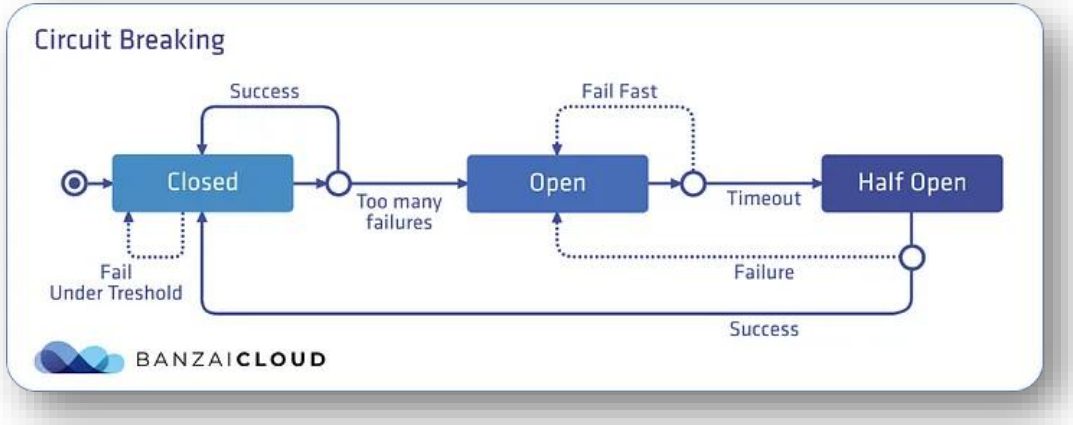
- Infrastructure patterns – either in deployment by terraform or helm charts, or by implementation with a managed service – can provide high degrees of redundancy and recoverability of services and data
- Platform availability is almost always preferred to rolling your own.
- If the platform you're building on has a pattern(s), select one which meets the solution availability goals
- Avoid mixing & matching patterns in a single application

1. Replicated/redundant subsystems (k8s replica sets/HPA/VPA, multiple clusters either with sharding customers or load-spreading in active-active) → horizontal scaling is preferred, pair with backup/restore strategy
2. Active-passive, warm standby, pilot light especially for persistence/databases with global DNS
3. Geode pattern
4. Rate limiting or throttling in infrastructure (e.g., at WAF)
5. Canary or synthetic transactions
6. Background data synchronization between instances
7. Redundant connections between cloud and on-prem datacenters or endpoints
8. Automatic recovery

SOFTWARE ARCHITECTURE PATTERNS

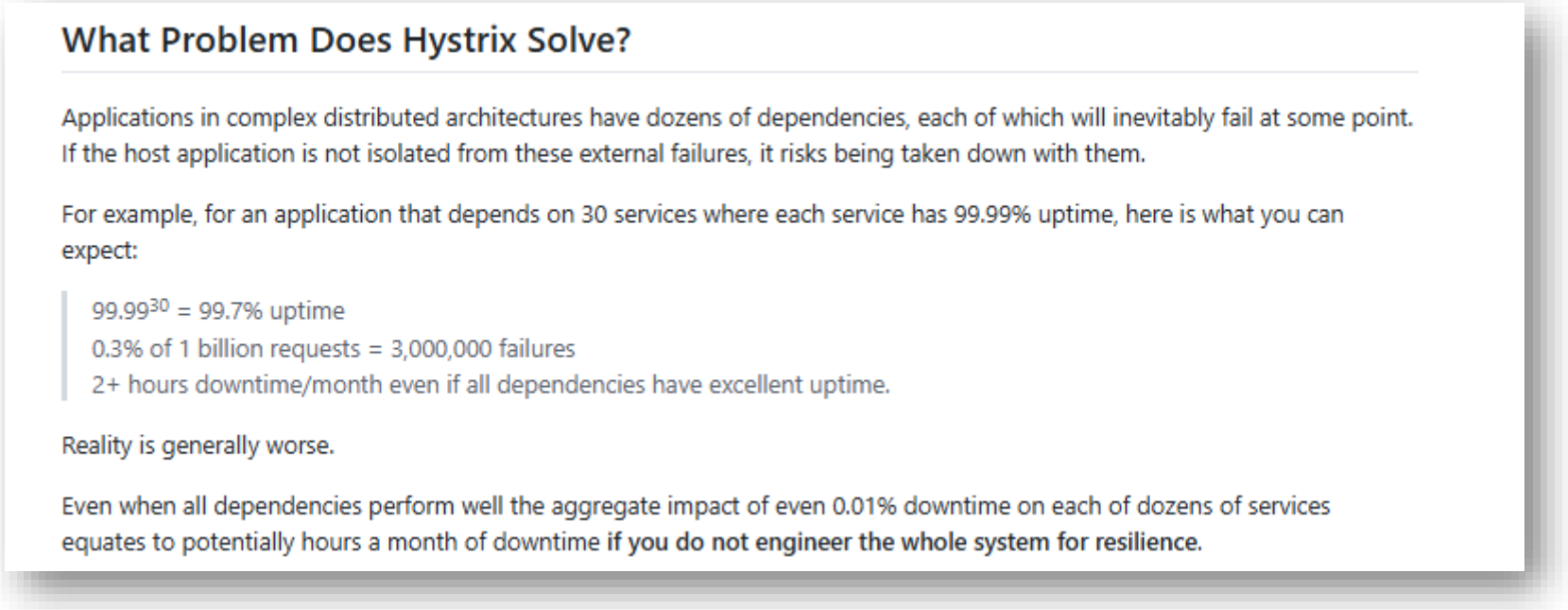
- Design the service architecture around loosely coupled, versioned services so that we can deploy and update parts of the system while it is running in production
- Beware of conflicts with database transaction methods and put transaction “close” to database
- Also think through idempotency – ability to retry operation with same effects – retrying a payment is not a good thing, retrying a read operation is fine

- Plan for graceful degradation with specific patterns like:
  1. (Cancel the operation and user gets to retry manually – always an option)
  2. Standard error processing (try/catch everywhere to avoid panic)
  3. Stateless services and load balancing (which leads to store and forward, fail-fast, write to disk before calling next service in chain)
  4. Backoff/retry (a “resilience layer”)
  5. Circuit breaker, which pauses dependent services when the caller senses the callee may be down and throttling
  6. Monitoring services (health checks on a dashboard, OTEL)
  7. Timeout
  8. Asynchronous calls or queued work (which then gets to short & long polling, and consistent error handling)
  9. Compensating transactions for workflows
  10. Bulkhead (fault isolation)
  11. Load management (HPA, bursting to a new cluster)
  12. Fallback



NETFLIX HYSTRIX AS A COMPLETE IMPLEMENTATION

- Came from Java to GoLang - https://github.com/Netflix/Hystrix/wiki/



What Problem Does Hystrix Solve?

Applications in complex distributed architectures have dozens of dependencies, each of which will inevitably fail at some point. If the host application is not isolated from these external failures, it risks being taken down with them.

For example, for an application that depends on 30 services where each service has 99.99% uptime, here is what you can expect:

> 99.99^{30} = 99.7% uptime
> 0.3% of 1 billion requests = 3,000,000 failures
> 2+ hours downtime/month even if all dependencies have excellent uptime.

Reality is generally worse.

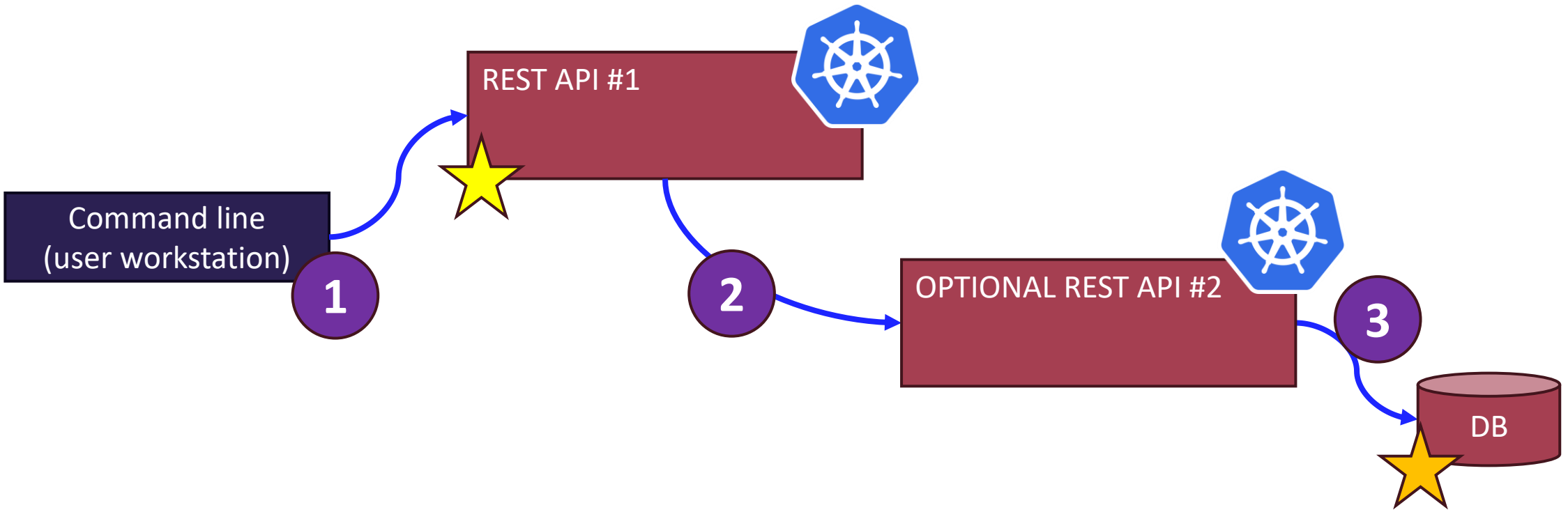
Even when all dependencies perform well the aggregate impact of even 0.01% downtime on each of dozens of services equates to potentially hours a month of downtime **if you do not engineer the whole system for resilience.**

MEASURING AND METRICS

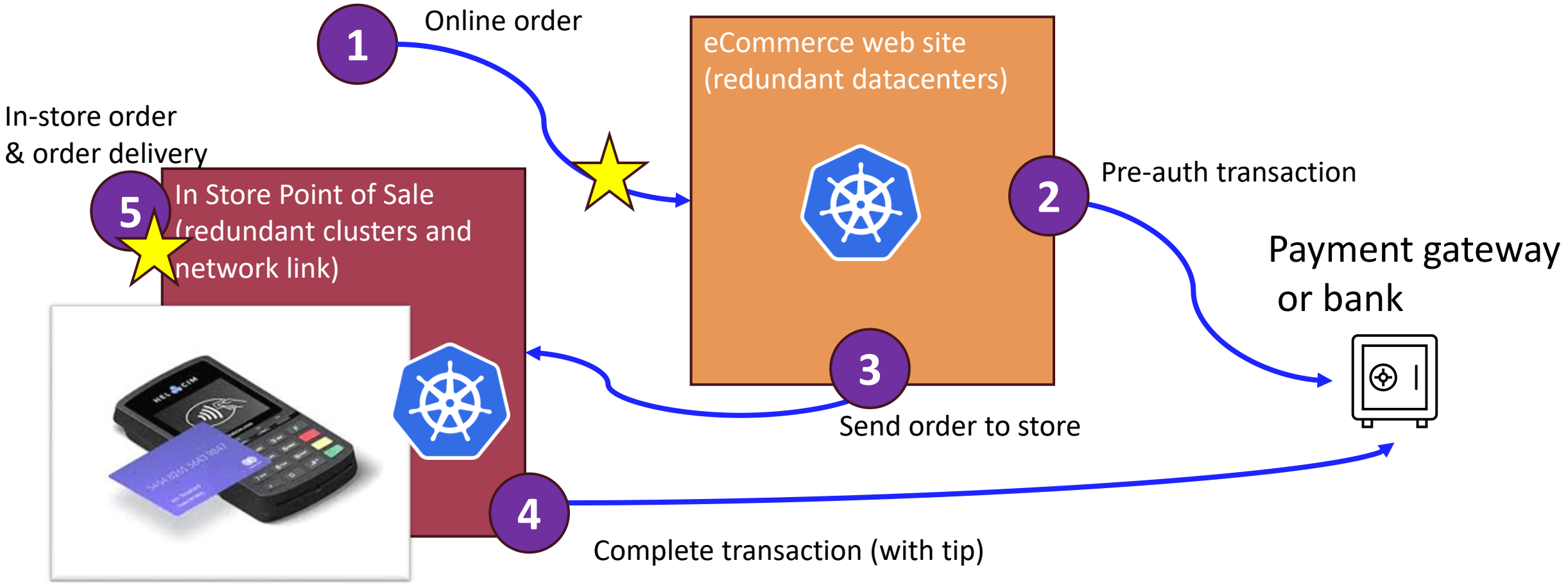
- Availability can be simple success/failure metric
- A true OTEL metric might be complete processes per tenant

- Internal SLO measurements might include
  - Latency of calls between services
  - Number of requests into service
- Often we look to indicators like performance counters (CPU, disk, IOPS, etc.) to determine “high-water marks”

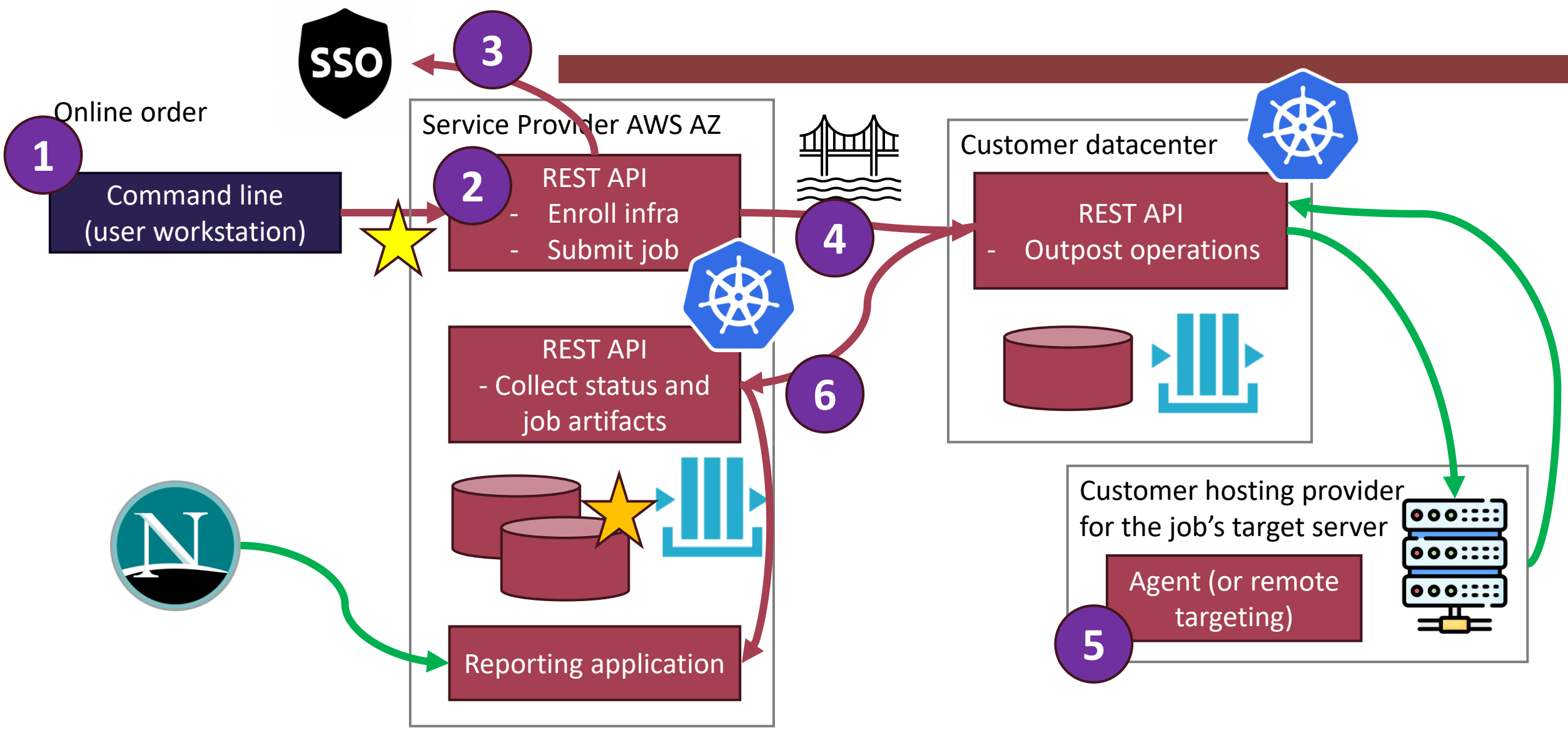
A SIMPLE CONNECTED MICROSERVICE EXAMPLE



RETAIL TO ECOMMERCE TO PAYMENT EXAMPLE



DEVOPS JOB RUNNER SYSTEM EXAMPLE



WHAT WE'RE DOING AT CHEF…

1. Global DNS routing to primary and failover EKS instances in AWS – the customer availability point, their DNS for API access has an active-passive environment
2. API gateway availability – multiple load-balanced gateways, customer sharding to limit damage for cluster failures (not yet geode), caching of AuthZ data, HPA/node-scaling inside and warm standby failover clusters
3. Internal infrastructure – using the built-in redundant services in AWS (DB, file/S3, message queues), with continuous monitoring and OTEL metrics; implementing **retries** between internal and external services (CLI can be manually retried, UI can be refreshed)
4. Agent resilience – **timeouts** on skills/jobs, **retries** for external deps (downloading plugins, remote server jobs, etc.), **circuit breaker** for communicating regular status back to state endpoints (turning communication back on by external signal), and a zero-trust "heartbeat" back to the APIs to check communication paths continuously
5. Job requests and agent results are written to disk immediately at the API so that post-processing and reporting can be offloaded
6. Monitoring by our operations team – on health checks, performance (Prometheus/Grafana), business metrics of both our services and customer-responsible assets (agented plus configured like SSO)
7. Development team reviewing improvements and testing each release

EVALUATING SOFTWARE COMPONENTS IN GOLANG

- Hystrix - https://github.com/afex/hystrix-go – wrapping methods with fallback behavior and timeouts as well as collecting statistics and load tests
- GoResilience (multiple) - https://github.com/slok/goresilience
- Go-resiliency - https://github.com/eapache/go-resiliency

- GoBreaker (circuit breaker) - https://github.com/sony/gobreaker/blob/master/README.md
- Circuit - https://github.com/cep21/circuit (recoverable panics)
- Rubyist CB - https://github.com/rubyist/circuitbreaker

- Heimdall - https://github.com/gojek/heimdall
- Failsafe-go - https://github.com/failsafe-go/failsafe-go
- HttpRetry (retry, backoff) - https://kdthedeveloper.medium.com/golang-http-retries-fbf7abacbe27
- GoRetryable (retries only) - https://github.com/hashicorp/go-retryablehttp
- Retry-Go - https://github.com/avast/retry-go
- Pester – retries and backoff - https://github.com/sethgrid/pester
- Circuit breaker - https://medium.com/@homayoonalimohammadi/circuitbreakers-in-go-d85f5297cb

```
func main() {
	runner := circuitbreaker.New(circuitbreaker.Config{
		//ErrorPercentThresholdToOpen:        50,
		//MinimumRequestToOpen:               20,
		//SuccessfulRequiredOnHalfOpen:       1,
		//WaitDurationInOpenState:            5 * time.Second,
		//MetricsSlidingWindowBucketQuantity: 10,
		//MetricsBucketDuration:              1 * time.Second,
	})

	for {
		time.Sleep(75 * time.Millisecond)

		err := runner.Run(context.TODO(), errorOnOddMinute)
		if err != nil {
			if err == errors.ErrCircuitOpen {
				fmt.Println("[!] circuit open")
			} else {
				fmt.Printf("[-] execution error: %s\n", err)
			}
		} else {
			fmt.Printf("[+] good\n")
		}
	}
}
```

TESTING SYSTEMS WITH REAL-WORLD CONDITIONS

- Use a tool like Chaos Monkey - https://netflix.github.io/chaosmonkey/ , fail-points, performance test tool (like k6, gatling, playwright, selenium, postman) or even manually turn off components to test your system's availability, simulate real-world scenarios like server failures, network outages, and high loads, and monitor your system's response and recovery times.

- Given availability requirements like uptime and reliability requirements like RTO and RPO or expected business process timelines:
  - Design tests for most likely failures contributing to an outage – based on load, failed business transaction, dependent service not available, complete datacenter down
  - Run tests in pre-production with integration test suite
  - Implement active monitoring with a dashboard of all production instances (Prometheus, or other dashboard)
  - Have a **canary tenant** or synthetic workflow to be constantly testing in production (this is what Dynatrace and other tools can report on)
  - Run your backup and restore components of business continuity on each update
  - Move lessons learned to SOP or runbooks and then to automated/scripted solutions

Sample tests

- Client goes away
  - Partial transactions in flight (never finished)
  - Transaction picked up by a new client (new token, etc.)
  - Split payments scenarios
- Gateway goes away
  - Primary down for minutes
  - Switch back to primary when redetected
  - Automated and manual SAF scenarios
  - (Circuit breaker operation)
- Device offline/soft failure
- Bad inputs to methods
  - Badly formed or missing inputs
  - Security token invalid
  - Content invalid (bad/fake CC number)
  - Duplicate request scenarios
  - Timeout non-responsive (various causes)
- Infrastructure
  - Database not reachable
  - Firewall rule change causes network outage (full or partial)
  - Bad health check
  - Resources overconsumed (CPU, memory)
  - Race conditions, load balancer
  - Bad passwords – db, gateway…
  - Logs, DB fill up
  - Certificates expired on sites

REPORT ON THE SERVICE LEVEL AGREEMENT (EVEN IF IT'S ONLY INTERNAL)

- ITIL processes require collection of data, share with customers when robust

1. Service health checks
2. Performance counters (CPU utilization, k8s performance)
3. Availability (uptime, PagerDuty or similar public page for status – calculate outages observed from incident management process)
4. Logging and OTEL metrics on "business operations" may be shared back with customers for audit purposes* (see previous talk on OTEL)

- Define DR plans and practice failover
- Compare the availability and reliability implementation with the theoretical model

- Perform RCA on all outages of services, drive the action plan back into the service

SUMMARY

- Availability, scalability and reliability cannot be solved by software or platform alone, it’s a combination of techniques
- Look out for examples in other systems, read the platform design sections from your cloud supplier
- Go as deep as you need to meet the expectation or SLA
  - Easy solutions win out, so if you can afford to drop operations and let the user retry, do so! (no need for a complicated solution)
- Don’t guess at an SLA – test it before you advertise!

REFERENCES

- Polly Project - https://www.thepollyproject.org/ (.NET)
- Netflix Hystrix - https://github.com/afex/hystrix-go (original in Java https://github.com/Netflix/Hystrix/wiki)
- Netflix Chaos Monkey - https://netflix.github.io/chaosmonkey/

- Azure Well-Architected - https://azure.microsoft.com/en-us/solutions/cloud-enablement/well-architected
- AWS Well-Architected - https://aws.amazon.com/architecture/well-architected/
- Uber SRE role - https://www.uber.com/blog/site-reliability-engineering-talks-feb-2016/
- ITIL v4 - https://wiki.en.it-processmaps.com/index.php/ITIL_4

- Chef 360™ Documentation – https://docs.chef.io/360/1.2/

THANK YOU

HTTPS://WWW.LINKEDIN.COM/IN/BRIANWLOOMIS/