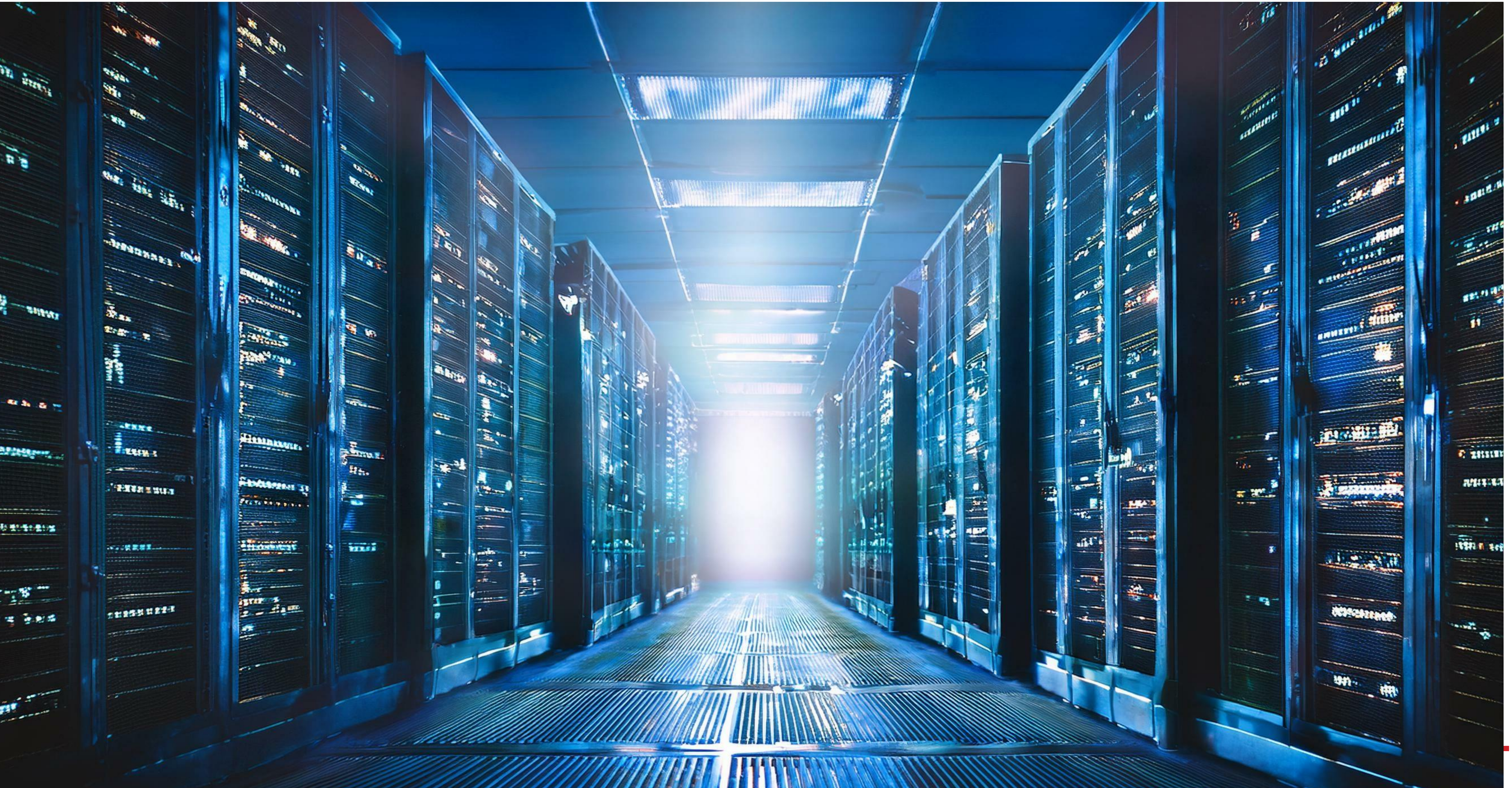
SITE RELIABILITY ENGINEERING – TESTING FOR RELIABILITY & ADVANCE TOPICS

© 2025 by Innovation In Software Corporation



TEKsystems Global Services

WORKFORCE DEVELOPMENT

PARTICIPANT GUIDE

REVIEW: DAY 2

Release Engineering, Automation, and Reliability Practices:

- Release Engineering
- Simplicity
- Alerting
- Troubleshooting
- Emergency Response
- Incident Management
- Tracking Outages

Day 1:

Introduction to SRE and Core Principles.

Day 2:

Release Engineering, Automation, Incident Management Practices

Day 3:

Testing for Reliability and Advanced Topics.

© 2025 by Innovation In Software Corporation

AGENDA FOR DAY 3

- Testing for Reliability
- Load Balancing
- Overload
- Cascading Failures
- Managing Critical Sate
- Conclusion

© 2025 by Innovation In Software Corporation

TESTING FOR RELIABILITY

© 2025 by Innovation In Software Corporation

TESTING FOR RELIABILITY – TESTING TYPES

Test Type	Description	Purpose for Reliability
Unit Testing	Tests small, individual functions or methods in isolation.	Prevents basic logic errors in code that could cause crashes or unexpected behavior.
Integration Testing	Tests how different modules or services interact with each other.	Ensures that components exchange data and perform workflows as expected.
System Testing	Tests the entire application in a production-like environment.	Validates that the system meets functional and performance requirements end-to-end.
User Acceptance Testing (UAT)	Validates that the application meets user and business needs.	Helps catch functional gaps or unexpected behavior from a business workflow perspective.
Production Testing	Lightweight, controlled testing in the live production environment (like synthetic tests or health checks).	Detects real-world issues (e.g., network latency, DNS failures) that test environments may miss.

© 2025 by Innovation In Software Corporation

TESTING FOR RELIABILITY - SRE & AZURE

Unit Tests

- Run in Azure DevOps Build Pipelines using MSTest, xUnit, etc.

Integration Tests

- Executed in Azure Test Plans or during Deployment Stages

System Tests

- Run in a Staging Environment deployed via Azure Pipelines

UAT

- Managed by product teams using Azure DevOps Test Plans

Production Tests

- Implemented via Azure Application Insights Availability Tests or synthetic monitoring

© 2025 by Innovation In Software Corporation

TESTING FOR RELIABILITY - TESTING AT SCALE

Load Testing

Simulates expected peak user load to check performance and stability.

Stress Testing

Pushes system beyond its capacity limits to observe failure behavior.

Soak Testing

Runs a heavy load over an extended period (hours/days) to identify memory leaks or resource exhaustion

Spike Testing

Simulates sudden large surges in traffic to test scaling and elasticity.

© 2025 by Innovation In Software Corporation

TESTING FOR RELIABILITY - TESTING AT SCALE

- Uncover Latency Issues
  Performance bottlenecks may not appear at low load.

- Test Resource Limits
  CPU, memory, network, and storage usage under heavy load.

- Find Concurrency Bugs
  Deadlocks, race conditions, and thread safety issues emerge at scale.

- Validate Auto-Scaling
  Ensure services scale up/down as expected in cloud environments.

- Simulate Failover Scenarios
  Test how load balancers, redundant systems, and backup services behave

© 2025 by Innovation In Software Corporation

POP QUIZ:

During a system test of your web application, you aim to validate the entire user journey from the frontend to the backend database.
Which Azure service combination is most appropriate for automating this end-to-end testing process?

A. Azure Pipelines and Azure Functions
B. Azure DevOps Pipelines and Azure Test Plans
C. Azure Deployment Center and Azure Chaos Studio
D. Azure Resource Manager (ARM) Templates and Azure Monitor Application Insights

© 2025 by Innovation In Software Corporation

POP QUIZ:

During a system test of your web application, you aim to validate the entire user journey from the frontend to the backend database.
Which Azure service combination is most appropriate for automating this end-to-end testing process?

A. Azure Pipelines and Azure Functions
B. Azure DevOps Pipelines and Azure Test Plans
C. Azure Deployment Center and Azure Chaos Studio
D. Azure Resource Manager (ARM) Templates and Azure Monitor Application Insights

© 2025 by Innovation In Software Corporation

LAB 06: IMPLEMENTING TESTING WITH AZURE DEVOPS

Goal: In this lab, you will build a pipeline that implements automated post-release checks to test that our API endpoints are properly available for our users. You will then implement the sending of the post-release test metrics to Azure log analytics along with the azure devops pipelines metrics.

Skills Covered:

- Task 1: Implement a Post-Deployment Testing Pipeline
- Task 2: Send API testing metrics to Azure Log Analytics
- Task 3: Setup metrics visualizations and alerts
- Task 4: Send all API Pipelines metrics to Azure Log Analytics

Instructions: AZ_SRE_lab_06.md

© 2025 by Innovation In Software Corporation

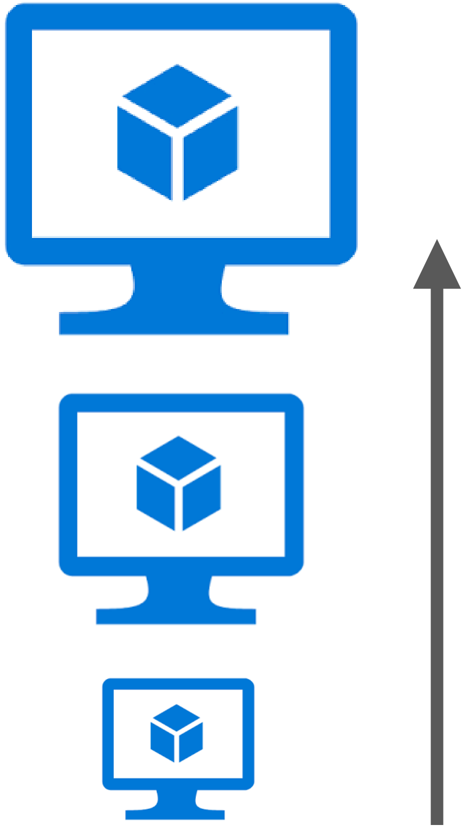
LOAD BALANCING

© 2025 by Innovation In Software Corporation

LOAD BALANCING - POWER ISN'T THE ANSWER

Vertical Scaling / Scale UP

(Increase size (CPU, RAM, etc.)



- **Single Point of Failure (SPOF)**
  If your giant server crashes, your entire service goes down.
- **Hardware Limitations**
  Physical servers have upper bounds for CPU, memory, and storage.
- **Maintenance Complexity**
  You can't easily take down a single massive server for upgrades or patching without causing downtime.
- **Limited Redundancy**
  No failover path exists if there's only one machine serving traffic.
- **Scaling is Expensive**
  Large, high-end servers are disproportionately expensive compared to multiple smaller ones.

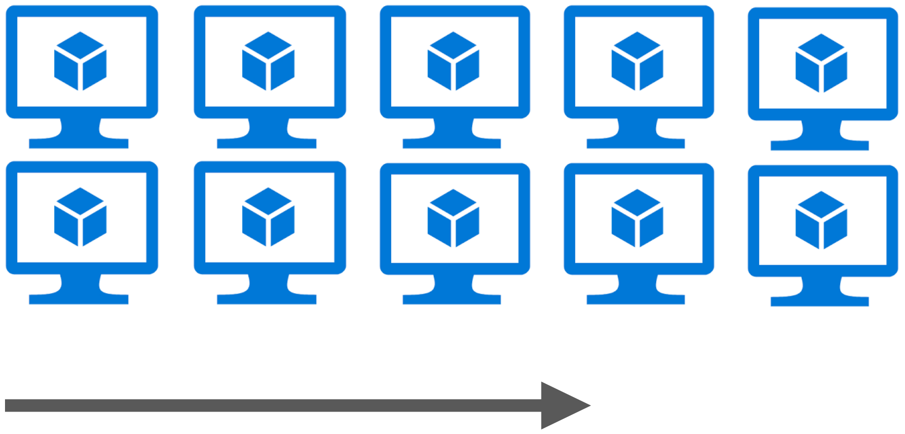
© 2025 by Innovation In Software Corporation

SRE'S PREFERRED APPROACH: HORIZONTAL SCALING

- **Multiple Smaller Servers/Instances**
  Distribute traffic across multiple nodes using load balancers.
- **Elastic Capacity**
  Easily add or remove nodes based on demand.
- **Improved Fault Tolerance**
  If one node fails, others continue to handle traffic.
- **Better Maintenance Windows**
  Take down one node for maintenance without full-service interruption.

Horizontal Scaling / Scale OUT

(Add more instances)



© 2025 by Innovation In Software Corporation

LOAD BALANCING - DNS

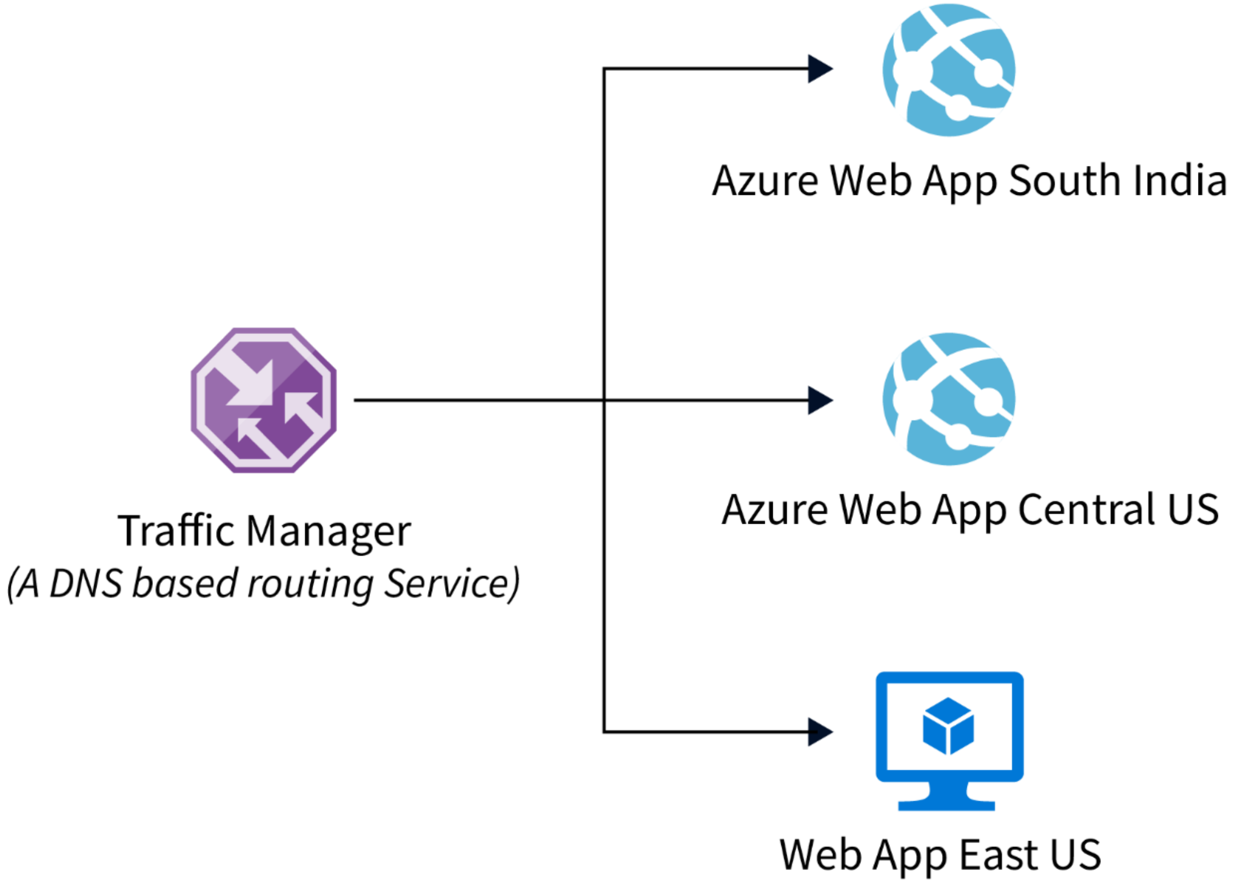
- Distribute traffic across geographically distributed endpoints
- Use DNS round-robin or geo-aware routing
- Simple but has limitations in health awareness
- Best for global load distribution, not real-time failover
- Use with other load balancing layers for resilience

© 2025 by Innovation In Software Corporation

LOAD BALANCING - DNS

- Azure Traffic Manager to provide DNS-level traffic distribution with different routing methods.

Routing Methods	
1	Priority
2	Weightage
3	Performance
4	Geographic
5	Subnet



© 2025 by Innovation In Software Corporation

LOAD BALANCING - VIP

- Use a Virtual IP (VIP) as a single front-end entry point
- Distribute incoming traffic across multiple backend instances
- Supports real-time health checks and load-based routing
- Enables automatic failover and fault isolation
- Provides better reliability and low-latency request handling
- VIPs are Managed in Azure Load Balancer

© 2025 by Innovation In Software Corporation

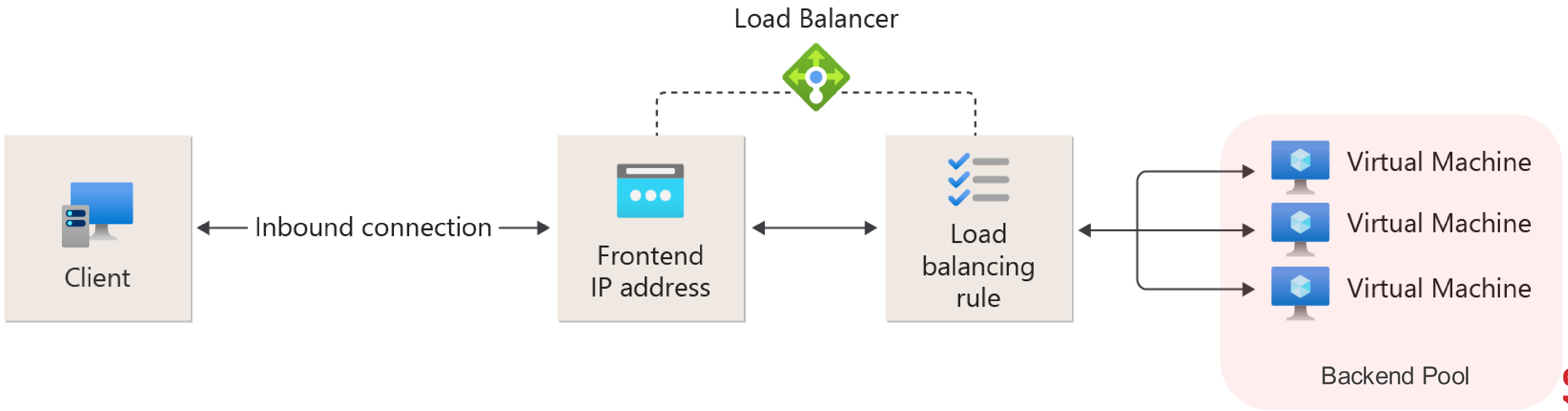
LOAD BALANCING – VIP - CONCEPTS

- **VIP (Virtual IP Address)**
  A single static IP address clients connect to for accessing the service.

- **Load Balancer Logic**
  The VIP is managed by a load balancer (hardware or software) that distributes incoming requests across healthy backends based on routing rules.

- **Health Probes**
  The load balancer regularly checks backend health and removes unhealthy nodes from traffic rotation.

- **Backend Pool**
  Multiple servers, instances, or containers that handle requests.



© 2025 by Innovation In Software Corporation

POP QUIZ:

You are an SRE transitioning from AWS to Azure, tasked with configuring Azure API Management (APIM) to handle frontend traffic for a retail application. The application experiences unpredictable traffic spikes during sales events, risking overload. Which configuration best prevents overload while maintaining user experience?

A. Set a fixed rate limit of 500 requests/second per subscription key with no response headers for throttling.
B. Enable client-side throttling with exponential backoff and configure subscription usage quotas of 1000 requests/hour per user.
C. Use Azure WAF to block all traffic exceeding 1000 requests/second at the gateway, without considering per-user limits.
D. Enable response caching for static content but remove all rate-limiting policies to prioritize speed.

© 2025 by Innovation In Software Corporation

POP QUIZ:

You are an SRE transitioning from AWS to Azure, tasked with configuring Azure API Management (APIM) to handle frontend traffic for a retail application. The application experiences unpredictable traffic spikes during sales events, risking overload. Which configuration best prevents overload while maintaining user experience?

A. Set a fixed rate limit of 500 requests/second per subscription key with no response headers for throttling.
B. Enable client-side throttling with exponential backoff and configure subscription usage quotas of 1000 requests/hour per user.
C. Use Azure WAF to block all traffic exceeding 1000 requests/second at the gateway, without considering per-user limits.
D. Enable response caching for static content but remove all rate-limiting policies to prioritize speed.

© 2025 by Innovation In Software Corporation

POP QUIZ:

Your Azure-based frontend uses an Azure Application Gateway to distribute traffic to virtual machine scale sets (VMSS) running a web application. A sudden spike in traffic causes one instance to fail, triggering a cascading failure across the fleet. Which strategy best prevents this?

A. Implement bulkheads by isolating VM instances into separate Virtual Machine Scale Sets across different Availability Zones.
B. Increase the Application Gateway’s idle timeout to 120 seconds to accommodate slow client connections.
C. Use Azure DDoS Protection Standard to mitigate traffic surges that cause instance failures.
D. Configure Azure Service Bus to queue all incoming HTTP requests before routing them to VM instances.

© 2025 by Innovation In Software Corporation

POP QUIZ:

Your Azure-based frontend uses an Azure Application Gateway to distribute traffic to virtual machine scale sets (VMSS) running a web application. A sudden spike in traffic causes one instance to fail, triggering a cascading failure across the fleet. Which strategy best prevents this?

A. Implement bulkheads by isolating VM instances into separate Virtual Machine Scale Sets across different Availability Zones.
B. Increase the Application Gateway's idle timeout to 120 seconds to accommodate slow client connections.
C. Use Azure DDoS Protection Standard to mitigate traffic surges that cause instance failures.
D. Configure Azure Service Bus to queue all incoming HTTP requests before routing them to VM instances.

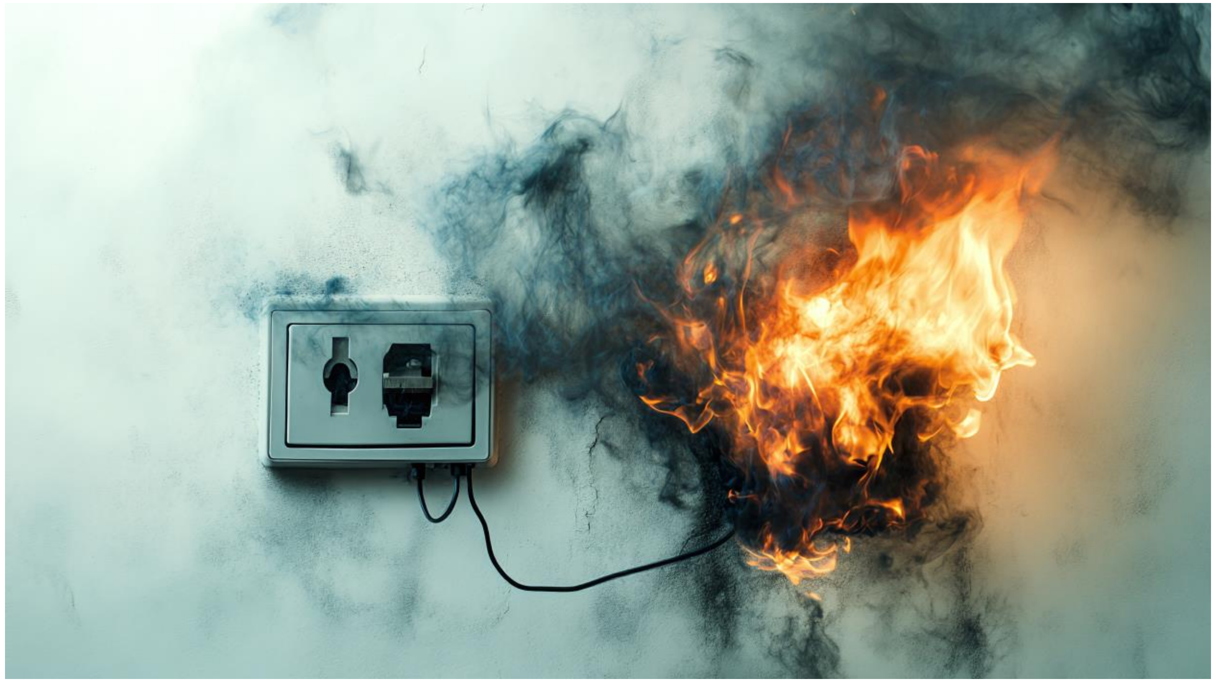
© 2025 by Innovation In Software Corporation

OVERLOAD

© 2025 by Innovation In Software Corporation

INTRODUCTION

- **Overload Definition**: Occurs when system demand exceeds capacity, causing performance degradation or failure.
- **Common Triggers**: Includes traffic spikes (e.g., marketing campaigns), resource exhaustion, or misconfigurations.
- **Impact on Reliability**: Threatens SLOs by increasing latency or causing outages if not managed.
- **Goal**: Maintain stability and user satisfaction during high-demand periods.



© 2025 by Innovation In Software Corporation

OVERLOAD - QPS PITFALLS

Query per Second is a blunt metric for load measurement.

- **Doesn't account for request complexity or resource cost**
  Two systems with identical QPS could consume drastically different CPU, memory, or I/O resources.

- **Focusing on QPS alone can lead to overload and missed bottlenecks**
  A system handling 1,000 lightweight queries may perform better than one handling 100 heavy queries concurrently.

- **Doesn't Reflect User Experience**
  High QPS doesn't mean good performance; latency and error rates matter more.

© 2025 by Innovation In Software Corporation

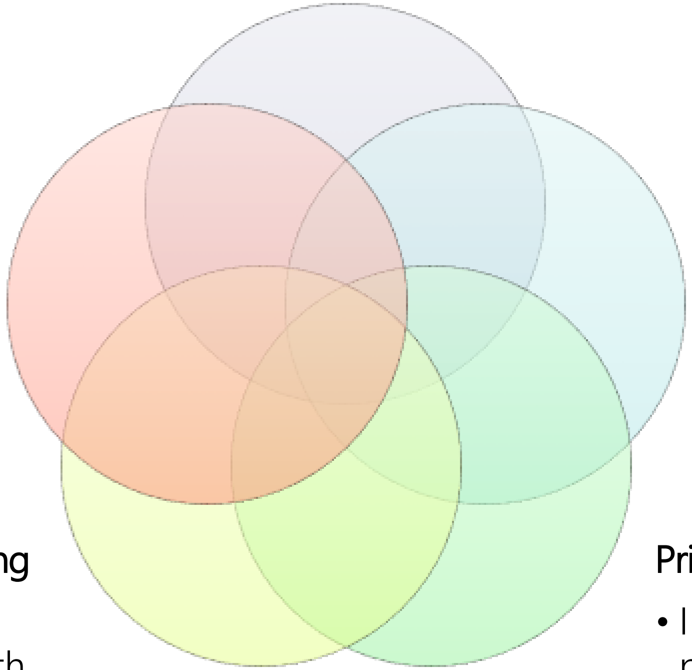
BEST PRACTICES FOR SRE OVERLOAD PREVENTION

Monitor Multi-Dimensional Metrics

- Track QPS alongside CPU, latency, memory, and error rate.

Test Different Request Types Separately

- Profile system performance for lightweight vs. heavy queries.



Define Overload Thresholds Based on Resources

- Use CPU utilization or request latency, not just QPS counts.

Implement Client-Side Throttling

- Prevent clients from overwhelming the system with too many requests.

Prioritize Critical Traffic

- Implement load shedding or prioritization to protect high-value requests during spikes.

© 2025 by Innovation In Software Corporation

OVERLOAD - QPS IN AZURE

Azure Monitor

- Monitor QPS alongside CPU and Memory metrics using Azure Monitor Metrics.

Azure Autoscale Triggers

- Set Autoscale triggers based on CPU utilization and request latency, not QPS alone, for services like Azure App Service, AKS, or Azure Functions.

Azure Insight

- Use Application Insights to track request dependencies and resource consumption per request type.

Azure Load Testing

- Leverage Azure Load Testing to simulate varied request loads and measure true capacity.

© 2025 by Innovation In Software Corporation

OVERLOAD - LIMITS/THROTTLING

Definition: Throttling is a mechanism used to limit the number of requests a system or service receives within a specific timeframe.

- Protect critical services from resource exhaustion
- Prevent noisy “neighbors” from degrading performance

Throttling techniques:

- **Per-Customer Limits**
  Prevent large customers from overwhelming shared services (e.g., 100 requests/second per tenant).

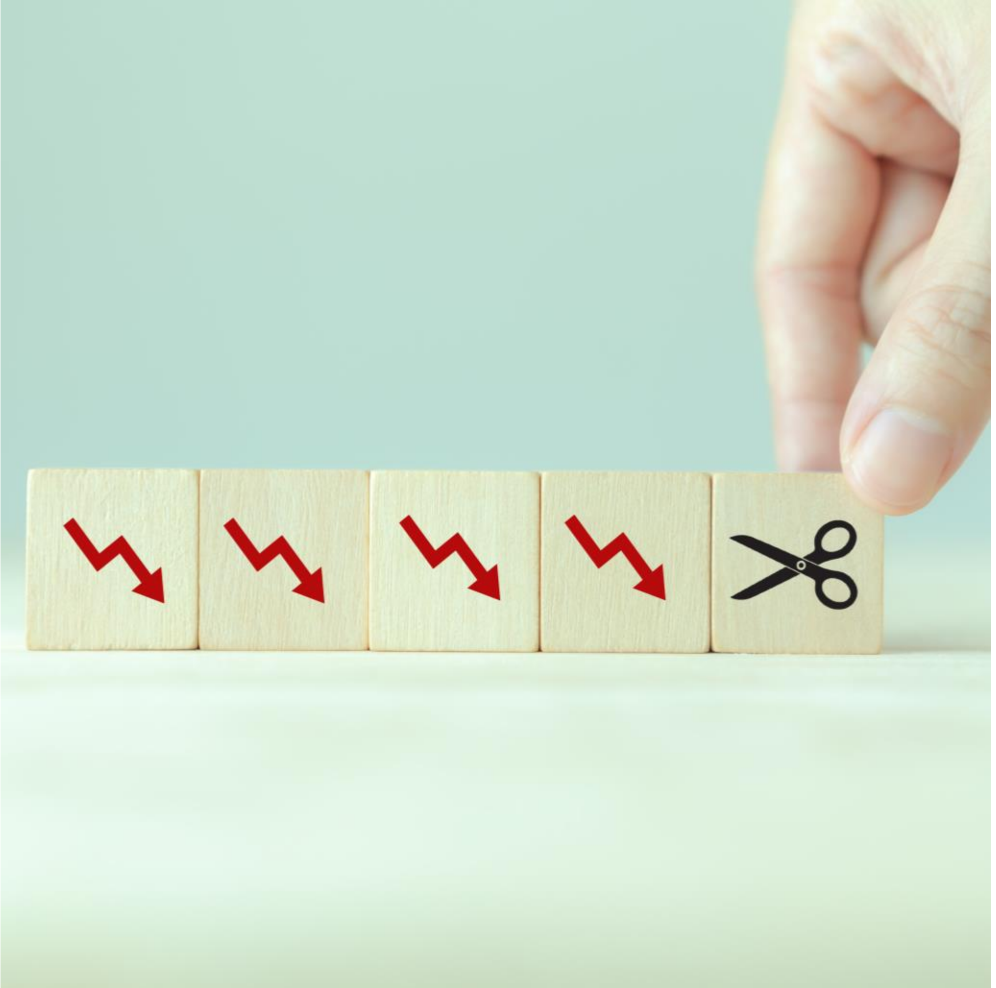
- **Per-User Limits**
  Avoid abusive individual user behavior (e.g., API keys limited to X requests/minute).

- **Per-IP or Region Limits**
  Protect against DDoS or concentrated bursts from a single location.

- **Global System-Wide Limits**
  Enforce absolute caps on total request volume to protect infrastructure.

© 2025 by Innovation In Software Corporation

BEST PRACTICES FOR SRE OVERLOAD PROTECTION WITH LIMITS



- **Implement Server-Side Rate Limiters**
  Use API gateways to enforce limits automatically.
- **Send Clear Error Responses**
  Use standard HTTP status codes (e.g., 429 Too Many Requests) with Retry-After headers.
- **Use Adaptive Throttling**
  Adjust rate limits dynamically based on system load.
- **Monitor Limit Breaches**
  Set alerts on throttle rates and limit violations to detect potential abuse or misconfigurations.
- **Communicate Limits to Customers**
  Provide clear API documentation so clients design around limits.

© 2025 by Innovation In Software Corporation

IN AZURE : MANAGING CONNECTION LOAD

Azure API Management

- Set per-subscription rate limits and quotas for API consumers

Azure Application Gateway

- Implement App Gateway Web Application Firewall (WAF) policies to block or throttle abusive traffic patterns.

Azure Monitor

- Monitor Metrics for 429 response codes to track rate limit enforcement.

Azure Front Door

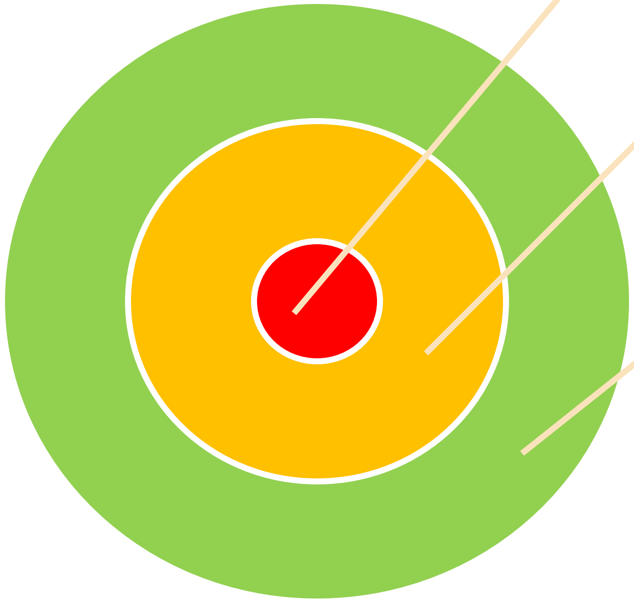
- Use Azure Front Door for global request throttling and custom rate limiting rules.

© 2025 by Innovation In Software Corporation

OVERLOAD - CRITICALITY/OVERLOAD ERRORS

Criticality refers to the **relative importance of different types of traffic or workloads** to the business and end users.

1. Prioritize critical traffic over non-essential workloads
2. Maintain service quality for key user actions
3. Use service-level objectives (SLOs) to guide overload protection



High-Criticality (Must Succeed)

- User login, payment processing, order placement

Medium-Criticality (Important but Deferrable)

- Background data sync, analytics

Low-Criticality (Can Fail Gracefully)

- Email reports, dashboard refreshes, batch jobs

© 2025 by Innovation In Software Corporation

HOW TO PRIORITIZE TRAFFIC DURING OVERLOAD:

Request Classification

Tag incoming requests by priority level (critical, standard, low

Rate Limiting by Priority

Apply stricter limits to non-critical traffic during load spikes.

Queue Management

Preferentially process high-priority requests first in queues

Circuit Breaking for Low Priority Services

Temporarily disable low-priority features if the system nears overload

Differentiated Resource Allocation

Reserve system resources (CPU, memory, threads) for critical workloads.

© 2025 by Innovation In Software Corporation

BEST PRACTICES OVERLOAD ERROR MANAGEMENT

1. **Predefine Traffic Criticality Classes**
   Avoid deciding what's critical during a live incident.

2. **Implement Request Dropping Logic Early**
   Don't wait for total resource exhaustion before shedding load.

3. **Communicate with Clients**
   Help downstream services and clients handle overload gracefully (via error codes and retry policies).

4. **Alert on Overload Errors**
   Monitor HTTP 429 and 503 rates as leading indicators of system stress.

5. **Incorporate into SLO Reporting**
   Track overload error rates as part of error budget consumption.

© 2025 by Innovation In Software Corporation

OVERLOAD - RETRY DECISIONS

Retrying failed requests can worsen overload.

- **Retry Storms**
  Thousands of clients retrying at once after initial failures cause even more load.

- **Amplified Traffic**
  Each failed request can generate multiple retry attempts, multiplying the load.

- **Increased Latency and Errors**
  System stays overloaded longer, impacting all users.

- **Service Collapse**
  Unchecked retries can bring down dependent services as well.

© 2025 by Innovation In Software Corporation

OVERLOAD – RETRY DECISIONS - RISKS

Safe to retry

- GET, HEAD, PUT (idempotent)
- Transient network failures
- HTTP 429, 503 with Retry-After

Not Safe to Retry

- POSTs that perform non-idempotent actions (e.g., payments)
- Permanent logic errors
- HTTP 400 (client error), 500 (internal error unless explicitly marked retriable)

© 2025 by Innovation In Software Corporation

SRE BEST PRACTICES FOR RETRY LOGIC

Exponential Backoff	• Space out retries progressively (e.g., wait 1s, then 2s, then 4s, etc.).
Add Jitter (Randomized Delay)	• Prevent retry spikes from synchronized clients by adding randomness to wait times.
Set a Maximum Retry Limit	• Prevent infinite retry loops by capping the number of attempts per request.
Retry Only Idempotent Operations	• Ensure retries won't cause unintended side effects (e.g., don't retry payment charges blindly).
Respect Server Signals	• Honor Retry-After headers and specific error codes like HTTP 429 and 503.
Implement Circuit Breakers	• Stop retrying entirely if the system remains overloaded for a set duration.

© 2025 by Innovation In Software Corporation

OVERLOAD - LOAD FROM CONNECTIONS

Idle or hanging clients can cause resource exhaustion:

- **Memory Usage**
  Each open connection consumes buffer space and session state memory.

- **File Descriptors**
  Operating systems have limits on simultaneous open sockets.

- **CPU Context Switching**
  Thousands of concurrent connections increase scheduling overhead.

- **Thread or Event Loop Limits**
  Servers with thread-per-connection models (or limited event loops) can be overwhelmed.

© 2025 by Innovation In Software Corporation

BEST PRACTICES : MANAGING CONNECTION LOAD

Best Practice	Benefit
Set Maximum Concurrent Connection Limits	Prevents connection storms from exhausting server capacity.
Implement Connection Timeouts	Disconnect idle clients after a reasonable period to free up resources.
Use Keep-Alive Timeouts Wisely	Tune keep-alive settings to balance performance and resource usage.
Use Connection Multiplexing	Tools like **HTTP/2**, **gRPC**, and **database connection pooling** reduce total connection count while supporting high request volume.
Apply SYN Flood Protection	Protects against low-level connection exhaustion attacks.
Monitor Active Connection Metrics	Track connection counts in dashboards and set alerts for sudden spikes.

© 2025 by Innovation In Software Corporation

IN AZURE : MANAGING CONNECTION LOAD

Azure App Service

- Configure connection timeout limits and monitor concurrent connections using Azure Monitor.

Azure Application Gateway

- Supports connection idle timeouts to clean up inactive sessions.

Azure Load Balancer

- Can help distribute TCP connection load across multiple backend instances.

Azure SQL Database

- Use connection pooling from application side (via ADO.NET, JDBC, etc.) to avoid connection exhaustion.

Azure Front Door

- Terminates client connections at the edge, reducing load on backend services.

© 2025 by Innovation In Software Corporation

POP QUIZ:

Your Azure application experiences frontend overload due to synchronous API calls overwhelming virtual machines or app services. Which queue-based strategy best mitigates this while ensuring reliability?

A. Use Azure Queue Storage or Azure Service Bus to buffer incoming API requests, scaling Azure Functions consumers based on queue length.
B. Configure Azure Event Grid to fan out API requests to multiple Azure VMs for parallel execution.
C. Implement Azure Event Hubs to batch API requests and process them at fixed intervals.
D. Route all API requests directly to Azure Container Instances (ACI) for faster execution, skipping any queuing layer.

© 2025 by Innovation In Software Corporation

POP QUIZ:

Your Azure application experiences frontend overload due to synchronous API calls overwhelming virtual machines or app services. Which queue-based strategy best mitigates this while ensuring reliability?

A. Use Azure Queue Storage or Azure Service Bus to buffer incoming API requests, scaling Azure Functions consumers based on queue length.
B. Configure Azure Event Grid to fan out API requests to multiple Azure VMs for parallel execution.
C. Implement Azure Event Hubs to batch API requests and process them at fixed intervals.
D. Route all API requests directly to Azure Container Instances (ACI) for faster execution, skipping any queuing layer.

© 2025 by Innovation In Software Corporation

POP QUIZ:

Your Azure-based application occasionally experiences overload during peak usage, leading to transient failures when calling backend services. What is the most effective strategy to implement retries while minimizing additional strain on the overloaded system?

A. Use immediate retries with short fixed delays in Azure API Management policies.
B. Implement exponential backoff with jitter in client-side retry logic or Azure Functions.
C. Increase the request timeout duration to give backend services more time to recover.
D. Retry failed requests instantly using Azure Front Door’s automatic failover feature.

© 2025 by Innovation In Software Corporation

POP QUIZ:

Your Azure-based application occasionally experiences overload during peak usage, leading to transient failures when calling backend services. What is the most effective strategy to implement retries while minimizing additional strain on the overloaded system?

A. Use immediate retries with short fixed delays in Azure API Management policies.
B. Implement exponential backoff with jitter in client-side retry logic or Azure Functions.
C. Increase the request timeout duration to give backend services more time to recover.
D. Retry failed requests instantly using Azure Front Door's automatic failover feature.

© 2025 by Innovation In Software Corporation

CASCADING FAILURES

© 2025 by Innovation In Software Corporation

CASCADING FAILURES - CAUSES/PREVENTION

Cascading failure:
Failure in one component triggers failures in others

Common causes:
Overload, unhandled errors, tight coupling

Prevention:
- Isolation, circuit breakers, rate limits
- Design for graceful degradation
- Implement backpressure and load shedding

© 2025 by Innovation In Software Corporation

COMMON CAUSES OF CASCADING FAILURES:

- **Overload Propagation**
  One overloaded service sends excess traffic or slow responses downstream, overloading other services.
- **Tight Coupling Between Services**
  Failure in one service leads to immediate failures in all services that depend on it, especially without retries or fallbacks.
- **Lack of Resource Isolation**
  Shared resources (like databases, message queues) get saturated due to one misbehaving component.
- **Improper Error Handling**
  One service's failure leads to timeouts, retry storms, or unhandled exceptions in others.
- **Excessive Retries Without Backoff**
  Causes increased load on already stressed systems, amplifying failure impact.

© 2025 by Innovation In Software Corporation

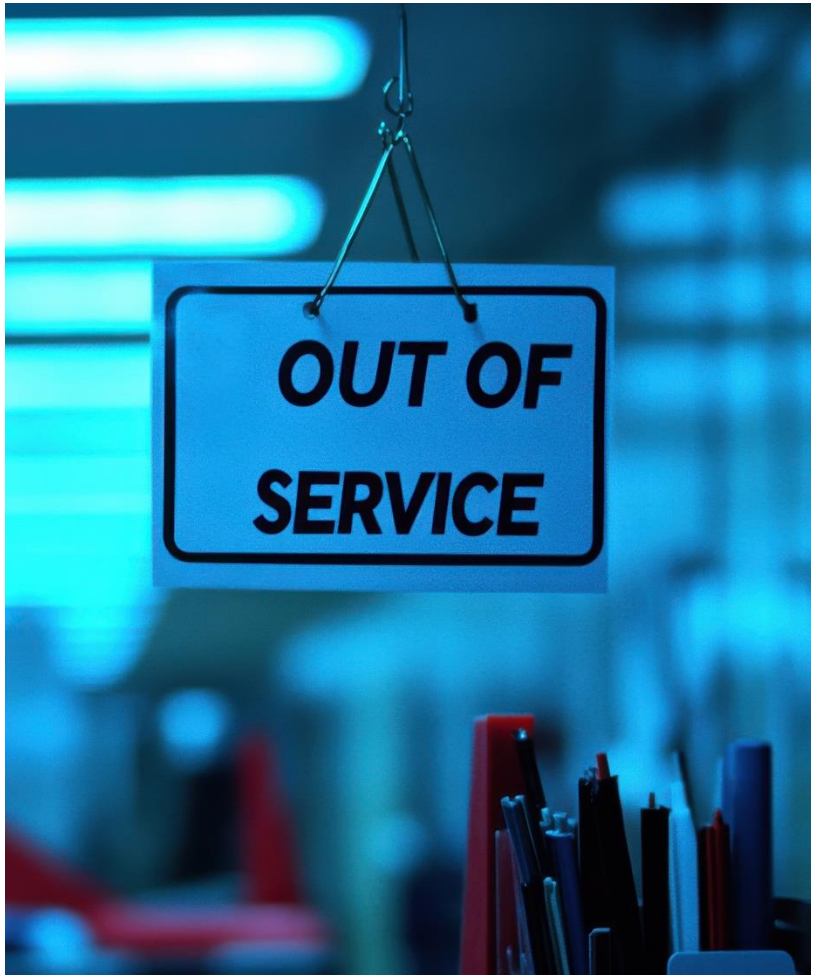
CASCADING FAILURES – SERVER OVERLOAD

- Overload and resource exhaustion trigger cascading failures
- Causes: High traffic, memory leaks, CPU saturation, thread pool exhaustion
- Leads to increased latency, dropped requests, or total server crash
- Prevention: Auto-scaling, rate limiting, resource monitoring

© 2025 by Innovation In Software Corporation

CASCADING FAILURES - SERVICE UNAVAILABILITY

- A critical service outage can trigger downstream failures
- Causes: software bugs, deployment errors, infrastructure issues
- Leads to retry storms, dependency timeouts, and client-side failures



© 2025 by Innovation In Software Corporation

SRE STRATEGIES TO PREVENT CASCADING FAILURES

Service Isolation	• Ensure that one service failure doesn't automatically bring down others (design for fault domains).
Circuit Breakers	• Automatically stop sending requests to unhealthy services to avoid further damage.
Rate Limiting and Throttling	• Prevent any single client or service from overloading others.
Timeouts and Retries with Backoff	• Limit retry storms and fast-fail on downstream failures.
Load Shedding	• Drop low-priority requests during high load to preserve critical functions.
Graceful Degradation	• Allow services to operate in a reduced capacity when dependencies fail (e.g., show cached data when database is unavailable).
Bulkhead Pattern	• Isolate resources for different service areas to contain failure impact.

© 2025 by Innovation In Software Corporation

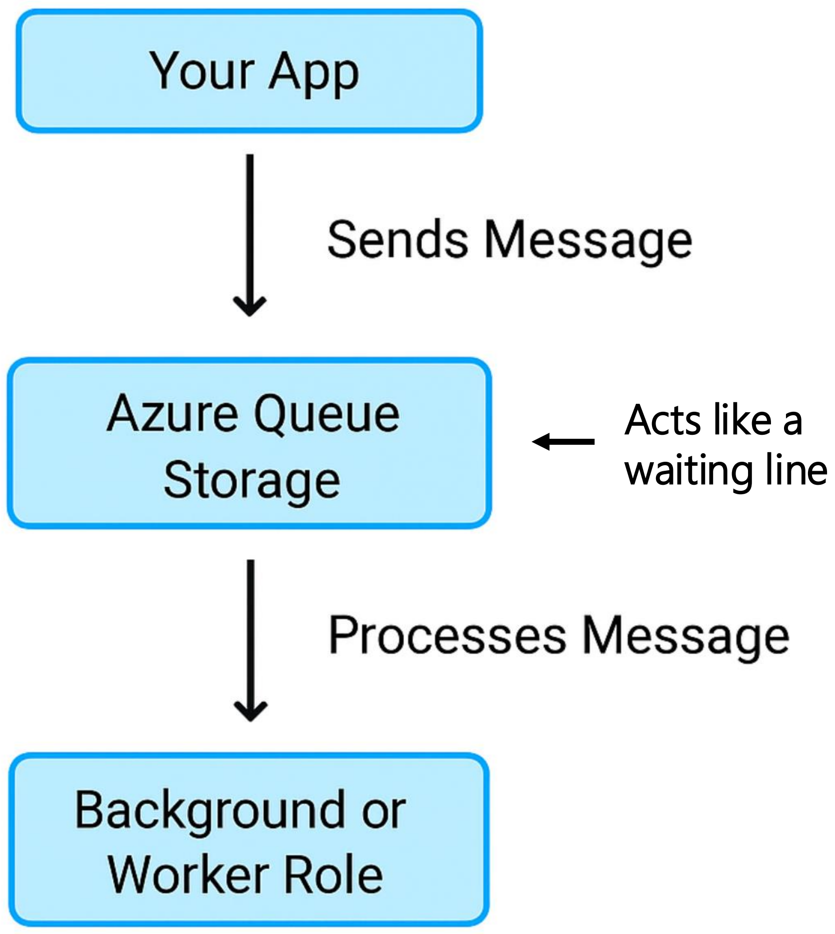
AZURE-SPECIFIC EXAMPLE

- Use Azure Traffic Manager or Azure Front Door for regional failover.
- Deploy multi-region active-active setups to avoid single region dependency.
- Monitor dependency availability via Application Insights Dependency Tracking.
- Implement retry policies and circuit breakers using Azure API Management (APIM) or in microservice frameworks on AKS.
- Leverage Azure Chaos Studio to simulate service unavailability and validate system behavior.

© 2025 by Innovation In Software Corporation

CASCADING FAILURES - QUEUE MANAGEMENT

- Queues absorb traffic spikes and smooth load
- Poorly managed queues can cause backlog and latency buildup
- Monitor queue length, processing rates, and age of messages
- Implement backpressure and rate limiting
- Use timeouts and circuit breakers to prevent queue-induced failures



© 2025 by Innovation In Software Corporation

BEST PRACTICE FOR QUEUE MANAGEMENT

Practice	Benefit
Set Queue Length Thresholds	Alert when queues grow beyond healthy limits.
Implement Backpressure on Producers	Slow down message producers when queue depth increases.
Limit Retry Attempts	Avoid infinite retries that can worsen backlog.
Use Dead-Letter Queues (DLQs)	Redirect poisoned or repeatedly failing messages to a DLQ for manual review.
Monitor Message Age (Time-in-Queue)	Helps identify processing slowdowns and latent backlogs.
Autoscale Consumers	Dynamically scale the number of workers based on queue length or processing lag.
Timeout Slow Consumers	Kill or restart consumers that hang or fail to process messages within a timeout window.

© 2025 by Innovation In Software Corporation

AZURE-SPECIFIC EXAMPLE

Azure Service Bus

- Supports message TTL, dead-letter queues, and peek-lock mechanisms to prevent message loss and control retries.

Azure Storage Queues

- Suitable for lightweight queueing with visibility timeouts and message expiration.

Azure Event Grid

- Offers durable event delivery with dead-lettering for failed message delivery.

Azure Functions Queue Triggers

- Allow autoscaling of queue consumers based on queue length.

Azure Monitor

- Can track queue depth, message age, and trigger alerts when thresholds are exceeded.

© 2025 by Innovation In Software Corporation

CASCADING FAILURES - TESTING FAILURES

- Deliberately introduce failures to test system resilience
- Simulate service outages, latency, resource exhaustion, or error spikes
- Validate fault tolerance mechanisms (circuit breakers, timeouts, retries)
- Identify hidden dependencies and failure paths

© 2025 by Innovation In Software Corporation

BEST PRACTICES FOR FAILURE INJECTION

Start Small

- Begin with low-impact tests in staging or non-critical environments.

Move Toward Production Testing

- Once mature, conduct controlled tests in production with careful scope limits.

Test One Failure Mode at a Time

- Avoid introducing too many simultaneous failures initially.

Monitor Closely

- Ensure robust monitoring during chaos experiments to measure impact and catch unintended side effects.

Have a Rollback Plan

- Be prepared to quickly revert any changes or halt the test if needed.

© 2025 by Innovation In Software Corporation

AZURE CHAOS STUDIO

- **Service Disruption Simulation**
  Test behavior when Azure VMs, App Services, or AKS nodes are stopped or degraded.

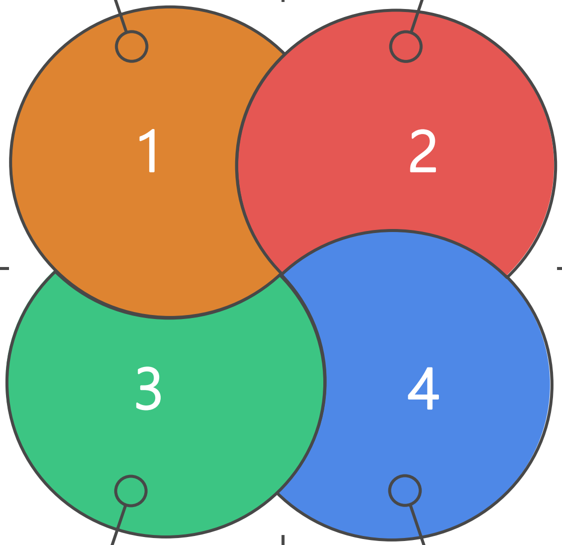
- **Network Latency Injection**
  Introduce delays between Azure resources.

- **CPU and Memory Stress**
  Simulate resource exhaustion conditions on target VMs.

- **Real-Time Experiment Tracking**
  Observe impact during and after chaos experiments via **Azure Monitor** and **Application Insights** dashboards.

© 2025 by Innovation In Software Corporation

BEST PRACTICES FOR DESIGNING RESILIENT SYSTEMS



Fail Fast

Detect and abort failing requests quickly to prevent resource drain.

Limit Concurrency

Prevent too many parallel requests overwhelming downstream services.

Dependency Health Checks

Proactively monitor critical service dependencies.

Testing for Cascading Failures

Run controlled failure injection tests (chaos engineering) to validate system resilience.

© 2025 by Innovation In Software Corporation

POP QUIZ:

You're managing a set of Azure App Services hosting microservices that recently became unresponsive during a flash sale event. Investigation reveals that the services were tightly coupled and lacked elasticity under load.
Which strategy best improves resilience and helps mitigate similar failures in the future?

A. Enable autoscaling for all Azure App Services and increase timeout thresholds.
B. Deploy all services in a single App Service Plan to simplify deployment and logging.
C. Route traffic through Azure Front Door with retries set to zero to reduce backend load.
D. Replace App Services with Azure Functions using fixed execution quotas.

© 2025 by Innovation In Software Corporation

POP QUIZ:

You’re managing a set of Azure App Services hosting microservices that recently became unresponsive during a flash sale event. Investigation reveals that the services were tightly coupled and lacked elasticity under load.
Which strategy best improves resilience and helps mitigate similar failures in the future?

A. Enable autoscaling for all Azure App Services and increase timeout thresholds.
B. Deploy all services in a single App Service Plan to simplify deployment and logging.
C. Route traffic through Azure Front Door with retries set to zero to reduce backend load.
D. Replace App Services with Azure Functions using fixed execution quotas.

© 2025 by Innovation In Software Corporation

POP QUIZ:

As part of your resilience engineering strategy in Azure, you plan to deliberately introduce failures to validate your system's ability to recover from disruptions.
Which approach is most appropriate for safely executing this in a controlled and observable way?

A. Use Azure Chaos Studio to inject faults into services, virtual machines, and dependencies during non-peak hours.
B. Randomly terminate Azure resources in production without alerting the operations team.
C. Increase traffic loads using Azure Load Testing without monitoring the impact.
D. Disable autoscaling features temporarily to simulate resource exhaustion under load.

© 2025 by Innovation In Software Corporation

POP QUIZ:

As part of your resilience engineering strategy in Azure, you plan to deliberately introduce failures to validate your system's ability to recover from disruptions.
Which approach is most appropriate for safely executing this in a controlled and observable way?

A. Use Azure Chaos Studio to inject faults into services, virtual machines, and dependencies during non-peak hours.
B. Randomly terminate Azure resources in production without alerting the operations team.
C. Increase traffic loads using Azure Load Testing without monitoring the impact.
D. Disable autoscaling features temporarily to simulate resource exhaustion under load.

© 2025 by Innovation In Software Corporation

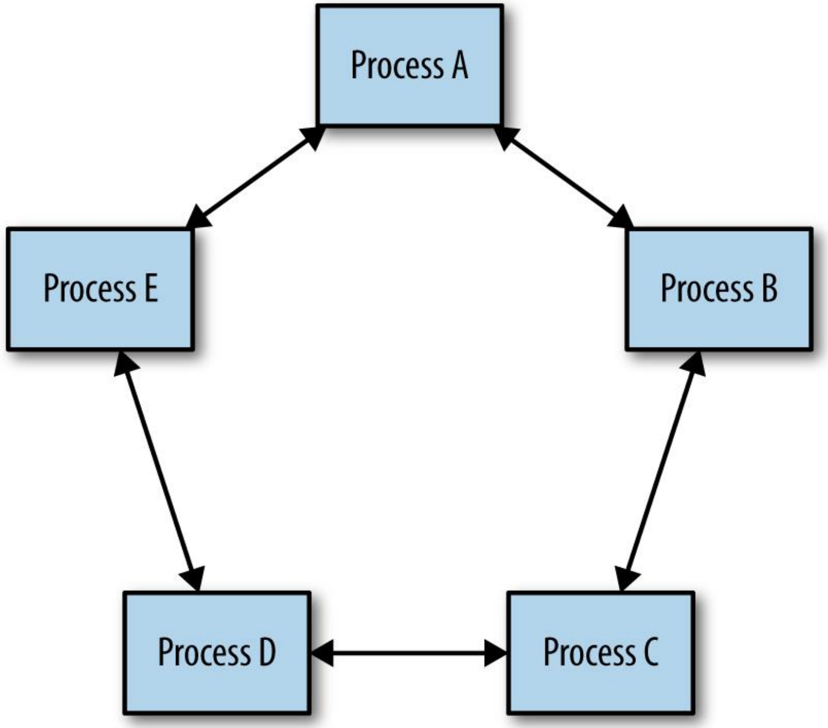
MANAGING CRITICAL STATE: DISTRIBUTED CONSENSUS

© 2025 by Innovation In Software Corporation

CRITICAL STATE - DISTRIBUTED CONSENSUS

Distributed consensus is the process that ensures **multiple nodes in a distributed system agree on a single value or state**, even in the presence of **failures, network partitions, or message delays**.

- Ensure agreement across distributed systems on critical state
- Used in leader election, replication, failover coordination
- Requires majority (quorum) agreement for any state change



© 2025 by Innovation In Software Corporation

WHY DISTRIBUTED CONSENSUS MATTERS

Why is it Important for Organizations?

- **Data Consistency:** In distributed systems, multiple nodes process data independently. Consensus algorithms guarantee that all nodes agree on the same data, preventing inconsistencies and errors.

- **Fault Tolerance:** Distributed systems are susceptible to failures (network issues, node crashes). Consensus protocols help these systems tolerate a certain number of failures while remaining operational.

- **Coordination and Synchronization:** Nodes need to synchronize their actions and make decisions based on the same data. Consensus algorithms facilitate this coordination.

- **Scalability:** As systems grow, consensus protocols enable them to handle increased load and complexity efficiently.

© 2025 by Innovation In Software Corporation

CHALLENGES IN DISTRIBUTED CONSENSUS

- **Network Partitions**
  Some nodes may temporarily lose communication, but consensus must remain consistent.

- **Latency**
  Achieving consensus requires multiple network round-trips across nodes.

- **Quorum Requirements**
  A minimum number of nodes (quorum) must agree before action is taken.

- **Failure Recovery**
  Systems must handle node restarts and state re-synchronization gracefully.

© 2025 by Innovation In Software Corporation

FAILURE MODES CONSENSUS HELP PREVENT

Split-Brain	Only one leader elected at any time.
Data Inconsistency	Agreement ensures that only committed data is accepted.
Lost Updates	Ensures updates are replicated and acknowledged by a quorum before confirming.
Multiple Masters	Prevents more than one master node from making state changes simultaneously.

© 2025 by Innovation In Software Corporation

SYSTEM ARCHITECTURE PATTERNS FOR DC

- Leader-based Replication with quorum voting
- Use odd number of nodes for quorum fault tolerance
- Support leader election, failover, and state synchronization
- Design for partition tolerance and high availability
- Examples: **Raft**, Paxos, ZAB (ZooKeeper Atomic Broadcast)

© 2025 by Innovation In Software Corporation

CASE STUDY: THE SPLIT-BRAIN PROBLEM

Consider a cluster of five nodes that gets split by a network failure in two partitions:

- Partition A: Leader, Follower 1 (2 nodes)
- Partition B: Follower 2, Follower 3, Follower 4 (3 nodes)

- Since Partition B forms a majority, they elect a new leader among themselves in higher term.
  This new leader can accept client requests and commit log entries.

- Two nodes believe they are the leader ?
  It would lead to data inconsistency, conflicting writes, and service disruption

- Prevented by distributed consensus algorithms (e.g., Paxos, Raft)
- Implementation of quorum-based decision making

© 2025 by Innovation In Software Corporation

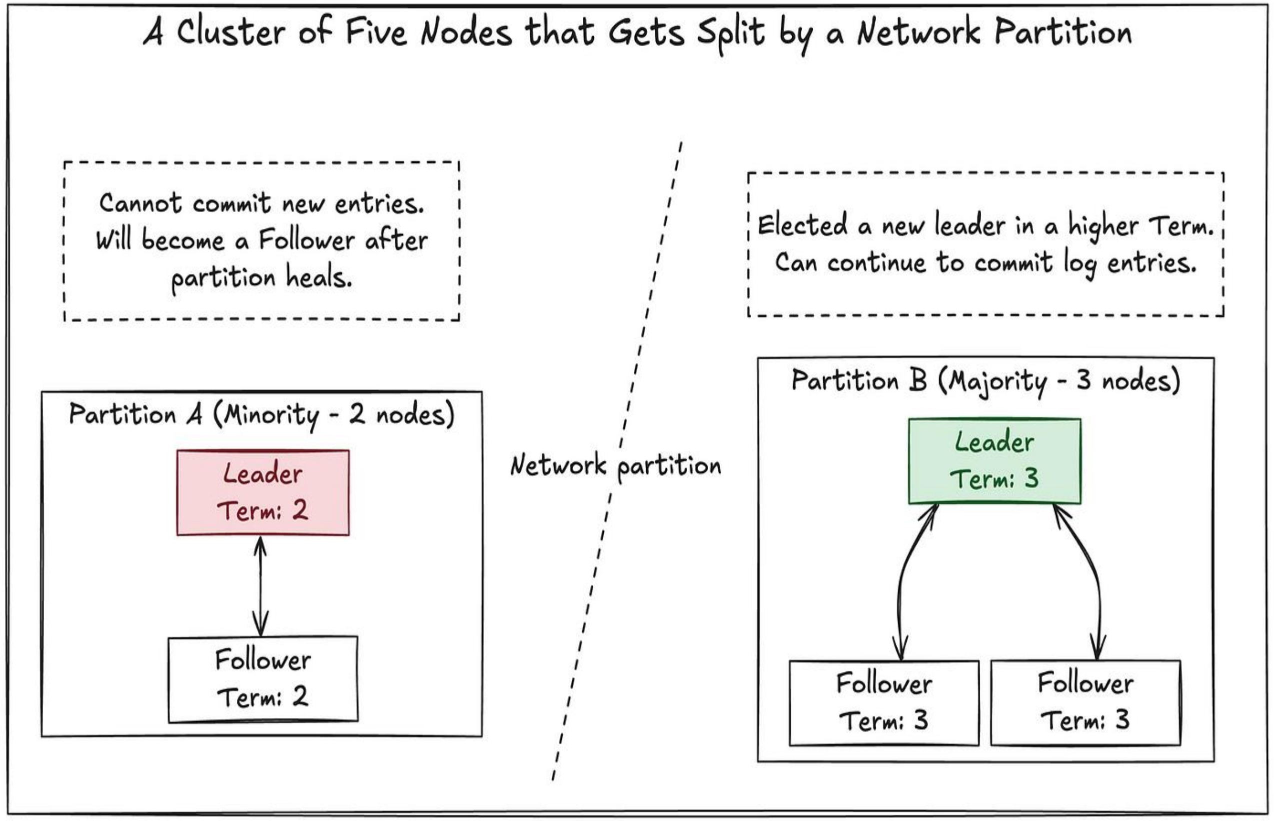
CASE STUDY: THE SPLIT-BRAIN PROBLEM (RAFT)

- The leader in Partition A is now in a minority partition.
  It cannot receive acknowledgements from a majority (3) of the nodes.

- When the network partition heals, the old leader from Partition A will receive a message from the new leader in Partition B

- Seeing the higher term number, it will recognize that it is stale, step down to become a follower

- Only the partition with a majority of nodes can make progress.



© 2025 by Innovation In Software Corporation

RELIABLE REPLICATED STATE MACHINES

- **Reliable Replicated State Machines (RSMs)** are a fundamental concept in distributed systems. They provide fault tolerance and high availability by replicating the same state machine across multiple nodes.
- This replication ensures that if one node fails, others can continue operating, maintaining system consistency and reliability.

Practical examples of Usage:

- **Data Storage:** RSMs are used to replicate data across multiple storage nodes, ensuring data availability and durability.
- **Configuration Management:** RSMs can be used to manage distributed configurations, ensuring that all nodes have the same configuration.

© 2025 by Innovation In Software Corporation

HOW RELIABLE REPLICATED STATE MACHINES WORK

1. Client Sends Command
Client sends a request to modify system state (e.g., write data, update configuration).

2. Leader Logs the Command
The leader node appends the command to its local log and proposes it to followers.

3. Consensus Agreement
Through consensus protocols like Raft or Paxos, the system ensures a quorum agrees on the order and content of the command.

4. State Machine Execution
Once agreed, all replicas execute the command on their local state machines in the exact same order.

5. Acknowledge Success
After state change is applied, the leader responds to the client confirming success.

© 2025 by Innovation In Software Corporation

BEST PRACTICES FOR MANAGING REPLICATED STORES

Practice	Benefit
Monitor Quorum Health	Track how many nodes are online and participating in consensus.
Test Failover Regularly	Simulate node and network failures to validate data resilience.
Use Reliable Storage Backends	Ensure disks and network layers meet latency and durability requirements.
Backup Regularly	Even with replication, backups protect against accidental data deletion or corruption.
Tune Timeouts and Heartbeats	Prevent false failure detection or delayed leader elections.

© 2025 by Innovation In Software Corporation

POP QUIZ:

Which scenario best illustrates the need for distributed consensus in a cloud-native application deployed on Azure?

A. Synchronizing configuration changes across stateless Azure App Services
B. Coordinating leader election among nodes in a distributed database like Cosmos DB or etcd running in Azure Kubernetes Services
C. Scaling out Azure Functions to handle more event-driven requests
D. Routing traffic using Azure Front Door across multiple geographic regions

© 2025 by Innovation In Software Corporation

POP QUIZ:

Which scenario best illustrates the need for distributed consensus in a cloud-native application deployed on Azure?

A. Synchronizing configuration changes across stateless Azure App Services
B. Coordinating leader election among nodes in a distributed database like Cosmos DB or etcd running in Azure Kubernetes Services
C. Scaling out Azure Functions to handle more event-driven requests
D. Routing traffic using Azure Front Door across multiple geographic regions

© 2025 by Innovation In Software Corporation

CONCLUSION

© 2025 by Innovation In Software Corporation

SRE IN A NUTSHELL

- SRE bridges software engineering and operations
- Focus on reliability, scalability, and performance
- Combines engineering practices with operational discipline
- Embraces failure as a learning opportunity
- Drives automation, monitoring, testing, and incident management

© 2025 by Innovation In Software Corporation

KEY TAKEWAYS

SRE Is a Mindset Shift:

- Moves organizations from **reactive firefighting** to **proactive engineering for reliability**.
- It applies **software engineering principles** to operations problems.

Reliability is Everyone's Responsibility:

- SRE creates **shared accountability** between **developers, operations teams, and business leaders**.
- It helps organizations balance **feature velocity and system stability**, using tools like **SLOs and Error Budgets**.

Failure is Expected and Managed:

- **SRE teams design for failure**, building systems that can **detect, isolate, and recover from faults automatically**.
- Post-incident reviews are conducted **blamelessly**, focusing on learning and system improvement.

Automation and Monitoring Are Core Practices:

- **SREs automate manual tasks** to reduce toil and human error.
- They invest heavily in **observability (metrics, logs, traces)** and **actionable alerting** to ensure early detection of problems.

Operational Excellence Comes from Engineering Discipline:

- Reliability is not left to chance. It is **measured, monitored, tested, and engineered** at every layer of the system.
- Teams use **robust testing strategies**, **failure injection**, and **load balancing techniques** to maintain service health.

© 2025 by Innovation In Software Corporation

INDIVIDUAL KEY TAKEAWAYS

- Write down three key insights from today’s session.
- Highlight how these take aways influence your work.

© 2025 by Innovation In Software Corporation

Q&A AND OPEN DISCUSSION

© 2025 by Innovation In Software Corporation


THANK
YOU