Failure Handling in Distributed Systems:

Retries, Circuit Breakers, and Resilience Engineering

Subtitle: Learn how large scale systems survive failures using retries, timeouts, circuit breakers, and resilience patterns in distributed architectures.

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app

Introduction

Most beginner developers assume:

- if code is correct, system will not fail
- failures are rare
- errors are exceptions only

Real production systems behave completely differently.

At scale:

- networks fail
- databases slow down
- services crash
- APIs timeout
- partial failures happen constantly

Failure is not an exception in distributed systems:

failure is normal behavior

This is why modern engineering focuses heavily on:

failure handling and resilience engineering

This PDF explains how production systems survive failure without collapsing.

Chapter 1: Why Failures Are Inevitable

In distributed systems:

- machines fail
- networks fail
- services degrade
- dependencies break

Important Truth

You cannot prevent all failures.

You can only:
• design systems that survive them

Chapter 2: Types of Failures

1. Network Failures

• packet loss
• latency spikes
• connection drops

2. Service Failures

• service crash
• overload
• memory leaks

3. Dependency Failures

• third party API down
• database slow
• microservice unavailable

4. Partial Failures

Only part of system fails while rest works.

Chapter 3: Timeouts

Timeout means:

stop waiting after a defined time limit

Why timeouts are important

Without timeouts:
• system waits forever
• resources get stuck
• cascading failures happen

Example

API call waits max:
• 2 seconds

If no response:
• request fails safely

Chapter 4: Retries

Retry means:

trying failed operation again

Example

Network request fails
→ retry automatically

Problem with retries

Retries can:
• increase system load
• worsen failures

Solution

Use:
• limited retries
• exponential backoff

Chapter 5: Exponential Backoff

Instead of retrying instantly:

Wait times increase gradually:

• 1s
• 2s
• 4s
• 8s

Benefit

Reduces system overload during failure spikes

Chapter 6: Circuit Breaker Pattern

Circuit breaker means:

stop sending requests to failing service temporarily

States

Closed

• normal operation

Open

• requests blocked

Half Open

• testing recovery

Benefit

Prevents:
• cascading failures

Chapter 7: Cascading Failures

One small failure can break entire system.

Example

Payment service slows down
→ order service waits
→ API gateway overloads
→ entire system collapses

Circuit breakers prevent this chain reaction

Chapter 8: Bulkhead Pattern

Bulkhead means:

isolate system resources into compartments

Example

Separate thread pools for:

• payments
• notifications
• analytics

Benefit

One failure does not affect all services

Chapter 9: Fail Fast Principle

Instead of waiting too long:

Fail immediately when:
• system is unhealthy

Benefit

Prevents resource waste

Chapter 10: Graceful Degradation

System should not fully break.

Instead:
• reduce functionality

Example

If recommendations fail:
• show basic feed instead

Benefit

User experience remains stable

Chapter 11: Fallback Systems

Fallback means:

alternative response when primary system fails

Example

Database fails
→ return cached data

Chapter 12: Idempotency

Idempotency means:

repeating operation gives same result

Example

Payment request sent twice
→ only one transaction processed

Why important

Prevents duplicate operations in retries

Chapter 13: Load Shedding

Load shedding means:

rejecting requests when system is overloaded

Example

During high traffic:
• reject low priority requests

Benefit

Protects core system stability

Chapter 14: Queue Based Buffering

Instead of processing immediately:

Requests go into queue.

Benefits

• smooth traffic spikes
• prevents overload

Chapter 15: Health Checks

Systems continuously check:

• service health
• database health
• dependency status

If unhealthy:

Traffic is redirected

Chapter 16: Redundancy

Redundancy means:

multiple copies of system components

Example

Multiple servers running same service

Benefit

If one fails:
• others continue

Chapter 17: Failover Systems

Failover means:

switching to backup system automatically

Example

Primary database fails
→ secondary takes over

Chapter 18: Retry Storm Problem

Bad retry logic causes:

• too many repeated requests
• system overload

Solution

• backoff
• jitter
• rate limiting

Chapter 19: Chaos Engineering

Chaos engineering means:

intentionally breaking systems to test resilience

Example

Shutting down servers randomly

Goal

Find weaknesses before real failures

Chapter 20: Observing Failure Behavior

Engineers analyze:

• how system reacts
• how fast recovery happens

• where bottlenecks appear

Chapter 21: Distributed System Reality

In distributed systems:

• nothing is perfectly reliable
• everything can fail anytime

Chapter 22: Recovery Engineering

Systems must recover quickly:

• automatic restarts
• state restoration
• failback mechanisms

Chapter 23: Tradeoff in Failure Handling

Stronger reliability requires:

• more infrastructure
• more complexity
• more cost

Chapter 24: Beginner vs Senior Thinking

Beginner

• “System should not fail.”

Senior Engineer

• “System will fail, how do we survive it?”

Chapter 25: Real Production Failure Flow

Request arrives
→ service fails
→ retry triggered
→ circuit breaker activates
→ fallback used
→ system stabilizes

Chapter 26: Most Important Principle

Systems are not designed to avoid failure
they are designed to handle failure gracefully

Key Takeaways

- Failures are normal in distributed systems
- Timeouts prevent infinite waiting
- Retries must be controlled with backoff
- Circuit breakers prevent cascading failures
- Graceful degradation protects user experience
- Redundancy and failover improve reliability
- Chaos engineering tests system resilience
- Real engineering focuses on survival, not perfection

Website Name: haas.dev
Website Link: https://dev-roast-app.vercel.app