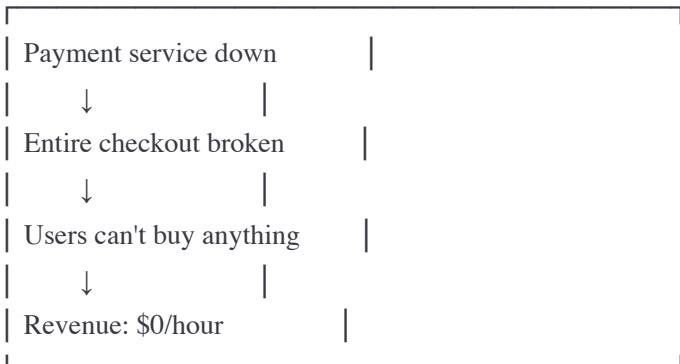


Chapter 13: Fault Tolerance and Resilience

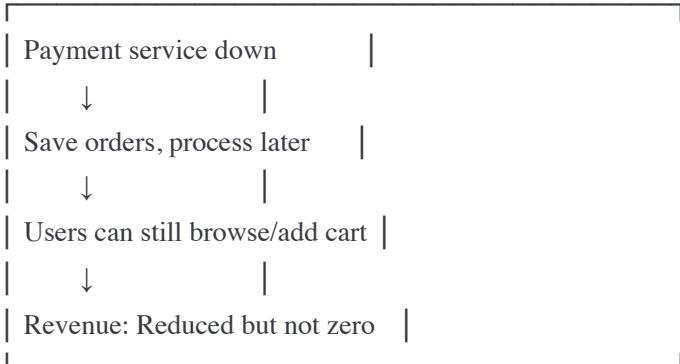
Introduction: Why Fault Tolerance Matters

Fault Tolerance: The ability of a system to continue operating even when parts fail.

Without Fault Tolerance:



With Fault Tolerance:



Key Principle: Degrade gracefully, don't crash completely!

1. Graceful Degradation

What is Graceful Degradation?

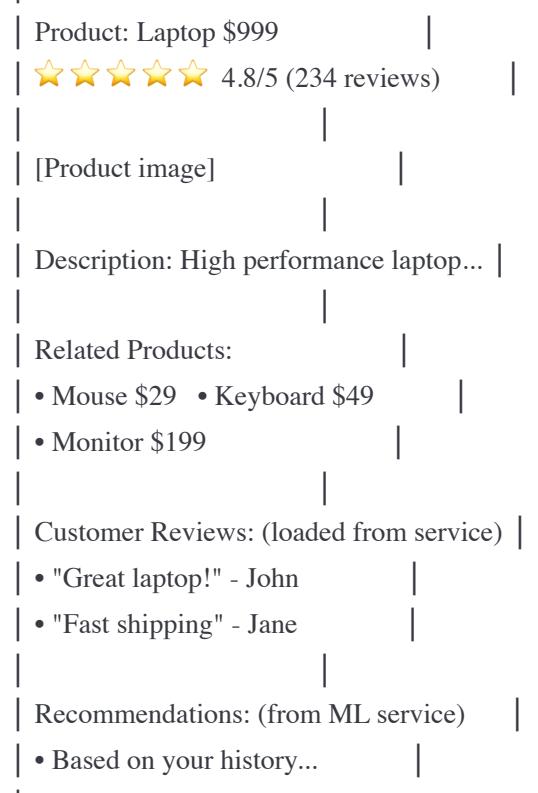
Concept: When a feature fails, provide reduced functionality instead of complete failure.

Analogy: Car with broken air conditioning

- Bad response: Car won't start
- Good response: Car works, just no AC (degraded but functional)

Example 1: E-Commerce Product Page

Full-Featured Product Page:

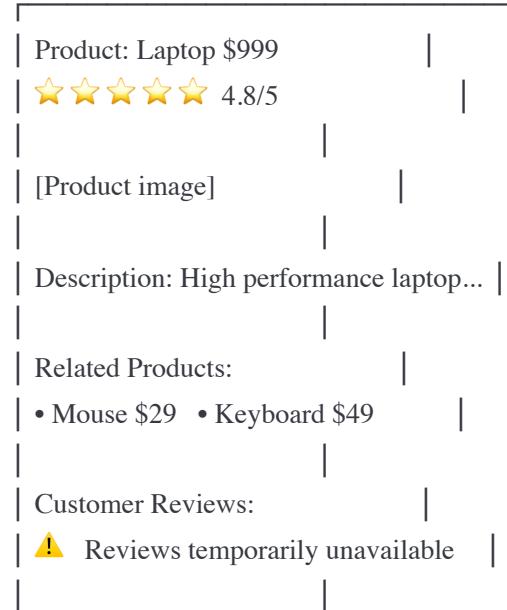


If Review Service Fails (Without Graceful Degradation):



Result: Lost sale!

If Review Service Fails (With Graceful Degradation):



| Recommendations:
|  Recommendations unavailable
|
|[Add to Cart] ← Still works!

Result: User can still buy!

Implementation

javascript

```
class ProductPageService {
  constructor() {
    this.productService = new ProductService();
    this.reviewService = new ReviewService();
    this.recommendationService = new RecommendationService();
  }

  async getProductPage(productId) {
    const page = {
      product: null,
      reviews: null,
      recommendations: null,
      warnings: []
    };

    try {
      // Core feature: Product details (MUST work)
      page.product = await this.productService.getProduct(productId);

    } catch (error) {
      // Core feature failed - can't degrade this
      throw new Error('Product not found');
    }

    // Non-critical feature: Reviews (can degrade)
    try {
      page.reviews = await this.reviewService.getReviews(productId);
    } catch (error) {
      console.error('Review service failed:', error);
      page.reviews = {
        error: true,
        message: 'Reviews temporarily unavailable'
      };
      page.warnings.push('Reviews service degraded');
    }

    // Non-critical feature: Recommendations (can degrade)
    try {
      page.recommendations = await this.recommendationService.getRecommendations(productId);
    } catch (error) {
      console.error('Recommendation service failed:', error);
      page.recommendations = {
        error: true,
        message: 'Recommendations temporarily unavailable'
      };
      page.warnings.push('Recommendation service degraded');
    }
  }
}
```

```

    }

    return page;
}
}

// Frontend rendering
function renderProductPage(pageData) {
    // Always show product (core feature)
    renderProduct(pageData.product);

    // Conditionally show reviews
    if (pageData.reviews.error) {
        renderWarning('Reviews are temporarily unavailable');
    } else {
        renderReviews(pageData.reviews);
    }

    // Conditionally show recommendations
    if (pageData.recommendations.error) {
        renderWarning('Recommendations are temporarily unavailable');
    } else {
        renderRecommendations(pageData.recommendations);
    }

    // Show warnings if any services degraded
    if (pageData.warnings.length > 0) {
        renderSystemStatus('Some features are temporarily limited');
    }
}

```

Example 2: Search Service with Fallback

javascript

```
class SearchService {
  constructor() {
    this.elasticsearchClient = new ElasticsearchClient();
    this.databaseClient = new DatabaseClient();
    this.cacheClient = new RedisClient();
  }

  async search(query) {
    // Try primary search (Elasticsearch - best results)
    try {
      console.log('Trying Elasticsearch...!');
      const results = await this.elasticsearchClient.search(query);
      return {
        results,
        source: 'elasticsearch',
        quality: 'high'
      };
    } catch (error) {
      console.warn('Elasticsearch failed, trying database...');

      // Fallback 1: Database (slower but works)
      try {
        const results = await this.databaseClient.search(query);
        return {
          results,
          source: 'database',
          quality: 'medium',
          warning: 'Search may be slower than usual'
        };
      } catch (error) {
        console.warn('Database failed, trying cache...');

        // Fallback 2: Cache (limited results)
        try {
          const results = await this.cacheClient.getPopularResults();
          return {
            results,
            source: 'cache',
            quality: 'low',
            warning: 'Showing popular results only'
          };
        } catch (error) {
          // All failed - return empty but don't crash!
        }
      }
    }
  }
}
```

```

        console.error('All search backends failed');

        return {
            results: [],
            source: 'none',
            quality: 'none',
            error: 'Search is temporarily unavailable. Please try again later.'
        };
    }
}

}

}

}

}

// Usage
const searchService = new SearchService();
const result = await searchService.search('laptop');

// Elasticsearch working: Fast, accurate results
// Elasticsearch down: Slower database results
// Database also down: Popular/cached results
// Everything down: Empty results, but site still works!

```

Graceful Degradation Strategies

1. FEATURE TOGLING

Disable non-critical features during overload

Example:

- Core: Add to cart ✓ (always on)
- Optional: Reviews ✘ (can disable)
- Optional: Recommendations ✘ (can disable)

2. Fallback Values

Use cached or default values when service fails

Example:

- Primary: Get real-time stock price
- Fallback: Use 15-minute delayed price
- Last resort: Show "Price unavailable"

3. REDUCED FUNCTIONALITY

Provide limited version of feature

Example:

- Full search: Elasticsearch with filters, sorting

- Degraded: Simple SQL LIKE query
- Minimal: Show popular items only

4. CACHED RESPONSES

Serve stale data better than no data

Example:

- Fresh data: 5 minutes old
- Degraded: 1 hour old
- Emergency: 1 day old

5. STATIC CONTENT

Pre-rendered pages as ultimate fallback

Example:

- Dynamic homepage: Personalized
- Degraded: Generic homepage
- Static: Cached HTML file

Implementation with Feature Flags:

javascript

```
class FeatureFlags {
  constructor() {
    this.flags = {
      'reviews': true,
      'recommendations': true,
      'real-time-inventory': true,
      'personalization': true
    };
  }

  // Monitor system health
  setInterval(() => this.adjustFlags(), 10000);
}

async adjustFlags() {
  const systemLoad = await this.getSystemLoad();

  // Disable features based on load
  if (systemLoad > 90) {
    console.log('⚠️ High load, disabling non-critical features');
    this.flags['reviews'] = false;
    this.flags['recommendations'] = false;
    this.flags['personalization'] = false;
  } else if (systemLoad > 70) {
    console.log('⚠️ Medium load, disabling heavy features');
    this.flags['recommendations'] = false;
  } else {
    // Normal load - all features on
    this.flags['reviews'] = true;
    this.flags['recommendations'] = true;
    this.flags['real-time-inventory'] = true;
    this.flags['personalization'] = true;
  }
}

isEnabled(feature) {
  return this.flags[feature] || false;
}

async getSystemLoad() {
  // Get CPU, memory, request queue depth
  // Return percentage (0-100)
  return 50; // Simplified
}
```

```
// Usage in product page
const flags = new FeatureFlags();

app.get('/product/:id', async (req, res) => {
  const product = await getProduct(req.params.id);
  const page = { product };

  // Conditionally load features
  if (flags.isEnabled('reviews')) {
    page.reviews = await getReviews(req.params.id);
  } else {
    page.reviews = { disabled: true, message: 'Reviews temporarily disabled' };
  }

  if (flags.isEnabled('recommendations')) {
    page.recommendations = await getRecommendations(req.params.id);
  }

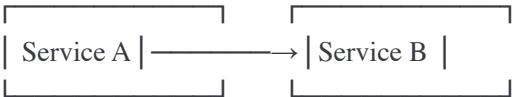
  res.json(page);
});
```

2. Circuit Breaker Pattern

What is a Circuit Breaker?

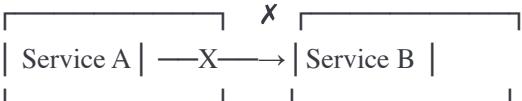
Analogy: Electrical circuit breaker in your home.

Normal Operation (Circuit CLOSED):



Requests flow normally

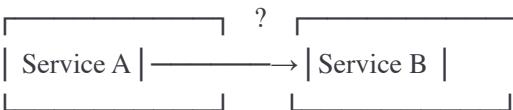
Too Many Failures (Circuit OPEN):



Fail fast

(don't wait)

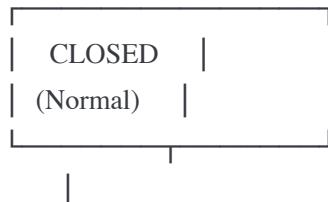
After Timeout (Circuit HALF-OPEN):



Try a few requests to test recovery

Circuit States

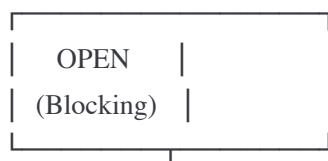
STATE MACHINE:



↓

Failures exceed

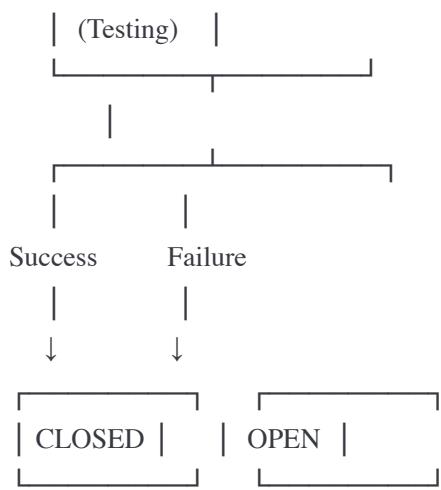
threshold (5 in 10s)



Timeout expires

(30 seconds)





Complete Circuit Breaker Implementation

javascript

```
class CircuitBreaker {
  constructor(options = {}) {
    this.failureThreshold = options.failureThreshold || 5;
    this.successThreshold = options.successThreshold || 2;
    this.timeout = options.timeout || 60000; // 60 seconds
    this.monitoringPeriod = options.monitoringPeriod || 10000; // 10 seconds

    this.state = 'CLOSED'; // CLOSED, OPEN, HALF_OPEN
    this.failures = [];
    this.successes = 0;
    this.nextAttempt = Date.now();
    this.stats = {
      totalRequests: 0,
      successfulRequests: 0,
      failedRequests: 0,
      rejectedRequests: 0
    };
  }

  async execute(operation) {
    this.stats.totalRequests++;

    // Check circuit state
    if (this.state === 'OPEN') {
      // Check if timeout has expired
      if (Date.now() < this.nextAttempt) {
        this.stats.rejectedRequests++;
        throw new Error('Circuit breaker is OPEN - request rejected');
      }
    }

    // Timeout expired, move to HALF_OPEN
    this.state = 'HALF_OPEN';
    console.log('Circuit breaker: OPEN → HALF_OPEN (testing recovery)');
  }

  try {
    // Execute the operation
    const result = await operation();

    // Success!
    this.onSuccess();
    return result;
  } catch (error) {
    // Failure!
    this.onFailure(error);
  }
}
```

```
        throw error;
    }
}

onSuccess() {
    this.stats.successfulRequests++;
    this.failures = []; // Reset failure count

    if (this.state === 'HALF_OPEN') {
        this.successes++;

        if (this.successes >= this.successThreshold) {
            // Enough successes, close circuit
            this.state = 'CLOSED';
            this.successes = 0;
            console.log(✓ Circuit breaker: HALF_OPEN → CLOSED (recovered));
        }
    }
}

onFailure(error) {
    this.stats.failedRequests++;

    const now = Date.now();
    this.failures.push(now);

    // Remove old failures outside monitoring period
    this.failures = this.failures.filter(
        timestamp => now - timestamp < this.monitoringPeriod
    );

    // Check if should open circuit
    if (this.state === 'HALF_OPEN') {
        // Failed during testing, back to OPEN
        this.state = 'OPEN';
        this.successes = 0;
        this.nextAttempt = now + this.timeout;
        console.log(✗ Circuit breaker: HALF_OPEN → OPEN (still failing));
    } else if (this.failures.length >= this.failureThreshold) {
        // Too many failures, open circuit
        this.state = 'OPEN';
        this.nextAttempt = now + this.timeout;
        console.log(🔴 Circuit breaker: CLOSED → OPEN (${this.failures.length} failures));
    }
}
```

```
getState() {
  return {
    state: this.state,
    failures: this.failures.length,
    stats: this.stats
  };
}

reset() {
  this.state = 'CLOSED';
  this.failures = [];
  this.successes = 0;
  console.log('Circuit breaker manually reset');
}
}

// Usage with external service
const paymentServiceBreaker = new CircuitBreaker({
  failureThreshold: 5,
  successThreshold: 2,
  timeout: 30000 // 30 seconds
});

async function processPayment(orderId, amount) {
  try {
    const result = await paymentServiceBreaker.execute(async () => {
      // Call external payment service
      const response = await fetch('https://payment-api.example.com/charge', {
        method: 'POST',
        body: JSON.stringify({ orderId, amount }),
        timeout: 5000
      });

      if (!response.ok) {
        throw new Error('Payment failed');
      }

      return await response.json();
    });
  }

  console.log('Payment successful:', result);
  return result;
}

} catch (error) {
  if (error.message.includes('Circuit breaker is OPEN')) {
    console.log(`⚠️ Payment service is down, using fallback`);
  }
}
```

```
// Graceful degradation: Queue for later processing
await queuePaymentForLater(orderId, amount);

return {
  status: 'queued',
  message: 'Payment will be processed when service recovers'
};

throw error;
}

}

// Monitor circuit breaker
setInterval(() => {
  const state = paymentServiceBreaker.getState();
  console.log('Circuit Breaker Status:', state);

  if (state.state === 'OPEN') {
    // Alert operations team
    alertOps('Payment service circuit breaker is OPEN!');
  }
}, 10000);
```

Circuit Breaker with Metrics

javascript

```
class CircuitBreakerWithMetrics extends CircuitBreaker {  
    constructor(serviceName, options) {  
        super(options);  
        this.serviceName = serviceName;  
        this.metrics = {  
            stateChanges: [],  
            recentLatencies: []  
        };  
    }  
  
    onSuccess() {  
        super.onSuccess();  
  
        // Record state change  
        if (this.state === 'CLOSED' && this.previousState === 'HALF_OPEN') {  
            this.metrics.stateChanges.push({  
                from: 'HALF_OPEN',  
                to: 'CLOSED',  
                timestamp: new Date().toISOString()  
            });  
        }  
  
        this.previousState = this.state;  
    }  
  
    onFailure(error) {  
        const previousState = this.state;  
        super.onFailure(error);  
  
        // Record state change  
        if (this.state !== previousState) {  
            this.metrics.stateChanges.push({  
                from: previousState,  
                to: this.state,  
                timestamp: new Date().toISOString(),  
                error: error.message  
            });  
        }  
  
        // Send metrics to monitoring system  
        this.sendMetrics();  
    }  
  
    this.previousState = this.state;  
}  
  
async execute(operation) {
```

```
const startTime = Date.now();

try {
  const result = await super.execute(operation);

  // Record latency
  const latency = Date.now() - startTime;
  this.recordLatency(latency);

  return result;
}

} catch (error) {
  const latency = Date.now() - startTime;
  this.recordLatency(latency);
  throw error;
}
}

recordLatency(latency) {
  this.metrics.recentLatencies.push(latency);

  // Keep last 100 measurements
  if (this.metrics.recentLatencies.length > 100) {
    this.metrics.recentLatencies.shift();
  }
}

sendMetrics() {
  // Send to Prometheus, Datadog, CloudWatch, etc.
  const avg_latency = this.metrics.recentLatencies.reduce((a, b) => a + b, 0)
    / this.metrics.recentLatencies.length;

  console.log(`Metrics for ${this.serviceName}:`, {
    state: this.state,
    success_rate: (this.stats.successfulRequests / this.stats.totalRequests * 100).toFixed(2) + '%',
    avg_latency: avg_latency.toFixed(2) + 'ms',
    total_requests: this.stats.totalRequests,
    rejected: this.stats.rejectedRequests
  });
}

getMetrics() {
  return this.metrics;
}

// Usage
```

```

const userServiceBreaker = new CircuitBreakerWithMetrics('UserService', {
  failureThreshold: 5,
  timeout: 30000
});

// Simulate traffic and failures
async function simulateTraffic() {
  for (let i = 0; i < 100; i++) {
    try {
      await userServiceBreaker.execute(async () => {
        // Simulate failing service (50% failure rate)
        if (Math.random() < 0.5) {
          throw new Error('Service unavailable');
        }
        return { userId: i, name: `User ${i}` };
      });
    } catch (error) {
      // Circuit breaker will open after threshold
    }
  }

  await new Promise(resolve => setTimeout(resolve, 100));
}

// Check metrics
console.log('Final metrics:', userServiceBreaker.getMetrics());
}

```

3. Retry Logic and Exponential Backoff

Why Retry?

Transient failures can succeed on retry:

- Network hiccup
- Temporary overload
- Database connection pool full
- Brief service restart

Timeline:

10:00:00 - Request fails (timeout)

10:00:01 - Retry → Success!

Without retry: Request failed (bad UX)

With retry: Request succeeded (good UX)

Simple Retry

javascript

```
async function simpleRetry(operation, maxRetries = 3) {
  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      const result = await operation();
      return result; // Success!
    } catch (error) {
      console.log(`Attempt ${attempt} failed: ${error.message}`);
      if (attempt === maxRetries) {
        throw error; // Final attempt failed
      }
      // Wait before retry
      await new Promise(resolve => setTimeout(resolve, 1000));
    }
  }
}

// Usage
const data = await simpleRetry(async () => {
  return await fetch('https://api.example.com/data');
}, 3);
```

Exponential Backoff

Problem with fixed delay: Thundering herd

Problem: 1000 requests fail simultaneously

All retry after 1 second

All hit server at same time again!

Server still overloaded → All fail again

Solution: Exponential backoff with jitter

Attempt 1: Wait 1 second

Attempt 2: Wait 2 seconds

Attempt 3: Wait 4 seconds

Attempt 4: Wait 8 seconds

...

Plus random jitter to spread out requests

Implementation:

javascript

```
class ExponentialBackoff {
  constructor(options = {}) {
    this.baseDelay = options.baseDelay || 1000; // 1 second
    this.maxDelay = options.maxDelay || 60000; // 60 seconds
    this.maxRetries = options.maxRetries || 5;
    this.jitter = options.jitter !== false;
  }

  calculateDelay(attempt) {
    // Exponential: 1s, 2s, 4s, 8s, 16s, 32s, 60s (capped)
    let delay = Math.min(
      this.baseDelay * Math.pow(2, attempt - 1),
      this.maxDelay
    );

    // Add jitter (random ±25%)
    if (this.jitter) {
      const jitter = delay * 0.25 * (Math.random() * 2 - 1);
      delay = delay + jitter;
    }

    return Math.floor(delay);
  }

  async retry(operation) {
    let lastError;

    for (let attempt = 1; attempt <= this.maxRetries; attempt++) {
      try {
        console.log(`Attempt ${attempt}/${this.maxRetries}`);
        const result = await operation();

        if (attempt > 1) {
          console.log(`✓ Succeeded on attempt ${attempt}`);
        }
      } catch (error) {
        lastError = error;
        console.log(`✗ Attempt ${attempt} failed: ${error.message}`);

        if (attempt < this.maxRetries) {
          const delay = this.calculateDelay(attempt);
          console.log(`Waiting ${delay}ms before retry...`);
          await new Promise(resolve => setTimeout(resolve, delay));
        }
      }
    }
  }
}
```

```

    }
}

// All retries exhausted
console.log(`❌ All ${this.maxRetries} attempts failed`);
throw lastError;
}
}

// Usage
const backoff = new ExponentialBackoff({
  baseDelay: 1000,
  maxDelay: 30000,
  maxRetries: 5,
  jitter: true
});

async function callUnreliableService() {
  return await backoff.retry(async () => {
    const response = await fetch('https://unreliable-api.example.com/data');

    if (!response.ok) {
      throw new Error(`HTTP ${response.status}`);
    }

    return await response.json();
  });
}

// Retry timeline with exponential backoff:
// Attempt 1: Immediate (0ms)
// Attempt 2: Wait ~1000ms (1s ± jitter)
// Attempt 3: Wait ~2000ms (2s ± jitter)
// Attempt 4: Wait ~4000ms (4s ± jitter)
// Attempt 5: Wait ~8000ms (8s ± jitter)
// Total time: ~15 seconds

```

Retry with Circuit Breaker

javascript

```
class ResilientClient {
  constructor(serviceName, serviceUrl) {
    this.serviceName = serviceName;
    this.serviceUrl = serviceUrl;

    this.circuitBreaker = new CircuitBreaker({
      failureThreshold: 5,
      timeout: 30000
    });

    this.backoff = new ExponentialBackoff({
      maxRetries: 3,
      baseDelay: 1000
    });
  }

  async call(endpoint, options = {}) {
    // Circuit breaker wraps retry logic
    return await this.circuitBreaker.execute(async () => {

      // Retry with exponential backoff
      return await this.backoff.retry(async () => {
        const response = await fetch(
          `${this.serviceUrl}${endpoint}`,
          {
            ...options,
            timeout: 5000
          }
        );

        if (!response.ok) {
          throw new Error(`HTTP ${response.status}: ${response.statusText}`);
        }

        return await response.json();
      });
    });
  }
}

// Usage
const userService = new ResilientClient('UserService', 'https://user-api.example.com');

try {
  const user = await userService.call('/users/123');
```

```
console.log('User:', user);

} catch (error) {
  if (error.message.includes('Circuit breaker is OPEN')) {
    console.log('Service is down, using cached data');
    // Fallback to cache
    const cachedUser = await cache.get('user:123');
    return cachedUser;
  }

  throw error;
}
```

When NOT to Retry

DON'T retry on:

- ✗ 400 Bad Request (client error, won't change)
- ✗ 401 Unauthorized (need new credentials)
- ✗ 403 Forbidden (permission issue)
- ✗ 404 Not Found (resource doesn't exist)
- ✗ 422 Validation Error (invalid input)

DO retry on:

- ✓ 408 Request Timeout
- ✓ 429 Too Many Requests (with backoff)
- ✓ 500 Internal Server Error
- ✓ 502 Bad Gateway
- ✓ 503 Service Unavailable
- ✓ 504 Gateway Timeout
- ✓ Network errors (ECONNRESET, ETIMEDOUT)

Implementation:

javascript

```
function isRetriableError(error) {
  // HTTP status codes
  const retriableStatusCodes = [408, 429, 500, 502, 503, 504];

  if (error.response) {
    return retriableStatusCodes.includes(error.response.status);
  }

  // Network errors
  const retriableNetworkErrors = [
    'ECONNRESET',
    'ETIMEDOUT',
    'ENOTFOUND',
    'ECONNREFUSED'
  ];

  return retriableNetworkErrors.includes(error.code);
}

async function smartRetry(operation, maxRetries = 3) {
  for (let attempt = 1; attempt <= maxRetries; attempt++) {
    try {
      return await operation();

    } catch (error) {
      // Check if error is retriable
      if (!isRetriableError(error)) {
        console.log('Non-retriable error, not retrying');
        throw error;
      }

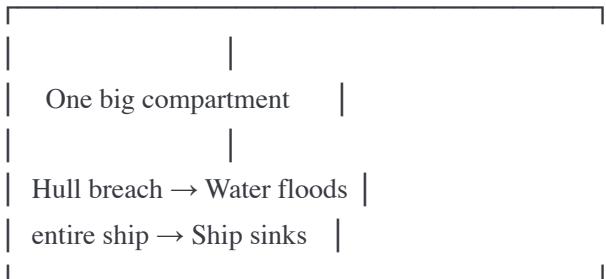
      if (attempt < maxRetries) {
        const delay = 1000 * Math.pow(2, attempt - 1);
        console.log(`Retriable error, waiting ${delay}ms...`);
        await new Promise(resolve => setTimeout(resolve, delay));
      } else {
        throw error;
      }
    }
  }
}
```

4. Bulkhead Pattern

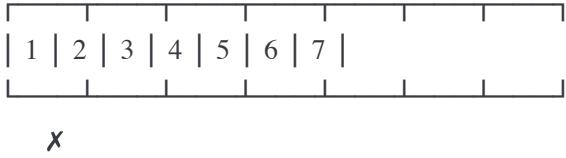
What is a Bulkhead?

Analogy: Ship compartments

Ship without Bulkheads:



Ship with Bulkheads:



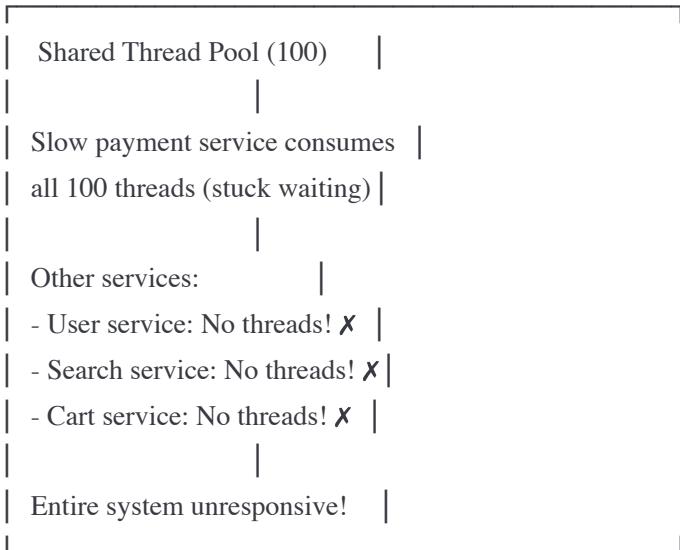
Water contained to compartment 2

Other compartments fine

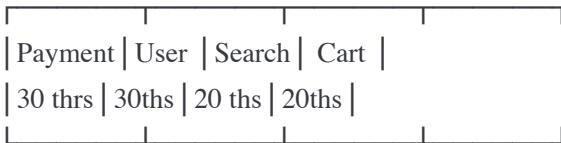
Ship stays afloat!

In Software:

Without Bulkhead:



With Bulkhead:



X

Payment slow
(uses all 30)

Other services still have threads!

- User service: 30 threads available ✓
- Search service: 20 threads available ✓
- Cart service: 20 threads available ✓

Fault isolated!

Bulkhead Implementation

javascript

```
class Bulkhead {
  constructor(name, maxConcurrent) {
    this.name = name;
    this.maxConcurrent = maxConcurrent;
    this.currentlyExecuting = 0;
    this.queue = [];
    this.stats = {
      totalRequests: 0,
      rejectedRequests: 0,
      queuedRequests: 0
    };
  }

  async execute(operation) {
    this.stats.totalRequests++;

    // Check if at capacity
    if (this.currentlyExecuting >= this.maxConcurrent) {
      this.stats.rejectedRequests++;
      throw new Error(`Bulkhead ${this.name} at capacity (${this.maxConcurrent})`);
    }

    this.currentlyExecuting++;
    console.log(`${this.name}: ${this.currentlyExecuting}/${this.maxConcurrent} active`);

    try {
      const result = await operation();
      return result;
    } finally {
      this.currentlyExecuting--;
    }
  }

  getStats() {
    return {
      name: this.name,
      capacity: this.maxConcurrent,
      current: this.currentlyExecuting,
      available: this.maxConcurrent - this.currentlyExecuting,
      stats: this.stats
    };
  }
}

// Create separate bulkheads for each service
```

```
const paymentBulkhead = new Bulkhead('Payment', 30);
const userBulkhead = new Bulkhead('User', 30);
const searchBulkhead = new Bulkhead('Search', 20);
const cartBulkhead = new Bulkhead('Cart', 20);

// Route requests through appropriate bulkhead
async function callPaymentService(data) {
  return await paymentBulkhead.execute(async () => {
    return await fetch('https://payment.example.com/charge', {
      method: 'POST',
      body: JSON.stringify(data)
    });
  });
}

async function callUserService(userId) {
  return await userBulkhead.execute(async () => {
    return await fetch(`https://user.example.com/users/${userId}`);
  });
}

// Simulate: Payment service becomes slow
async function simulateSlowPayment() {
  const promises = [];

  // Send 50 payment requests (slow service)
  for (let i = 0; i < 50; i++) {
    promises.push(
      callPaymentService({ amount: 100 }).catch(err => {
        console.log('Payment rejected:', err.message);
      })
    );
  }

  // Send 50 user requests (fast service)
  for (let i = 0; i < 50; i++) {
    promises.push(
      callUserService(i).catch(err => {
        console.log('User request rejected:', err.message);
      })
    );
  }

  await Promise.all(promises);

  // Payment bulkhead: 30/30 (rejected 20)
  // User bulkhead: 30/30 (all succeeded)
}
```

```
console.log('Payment stats:', paymentBulkhead.getStats());
console.log('User stats:', userBulkhead.getStats());
}
```

Thread Pool Bulkhead

javascript

```
class ThreadPoolBulkhead {
  constructor(name, poolSize, queueSize) {
    this.name = name;
    this.poolSize = poolSize;
    this.queueSize = queueSize;
    this.activeCount = 0;
    this.queue = [];
  }

  async execute(operation) {
    // Check if pool has capacity
    if (this.activeCount < this.poolSize) {
      return await this._executeImmediately(operation);
    }

    // Pool full, try to queue
    if (this.queue.length < this.queueSize) {
      return await this._queueOperation(operation);
    }

    // Pool and queue full, reject
    throw new Error(`${this.name} bulkhead saturated (pool: ${this.poolSize}, queue: ${this.queueSize})`);
  }

  async _executeImmediately(operation) {
    this.activeCount++;
    try {
      const result = await operation();
      return result;
    } finally {
      this.activeCount--;
      this._processQueue(); // Try to process queued items
    }
  }

  async _queueOperation(operation) {
    return new Promise((resolve, reject) => {
      this.queue.push({
        operation,
        resolve,
        reject
      });

      console.log(`Queued operation (queue: ${this.queue.length}/${this.queueSize})`);
    });
  }
}
```

```

}

_processQueue() {
  // Process next queued operation if pool has capacity
  if (this.queue.length > 0 && this.activeCount < this.poolSize) {
    const { operation, resolve, reject } = this.queue.shift();

    this._executeImmediately(operation)
      .then(resolve)
      .catch(reject);
  }
}

// Usage
const paymentPool = new ThreadPoolBulkhead('Payment', 10, 20);

// 100 concurrent payment requests
const promises = [];
for (let i = 0; i < 100; i++) {
  promises.push(
    paymentPool.execute(async () => {
      await new Promise(resolve => setTimeout(resolve, 1000));
      return { paymentId: i };
    }).catch(err => {
      console.log(`Request ${i} rejected: ${err.message}`);
    })
  );
}

await Promise.all(promises);

// Output:
// First 10: Execute immediately (pool)
// Next 20: Queued (queue)
// Remaining 70: Rejected (saturated)

```

Semaphore Pattern (Limiting Concurrent Requests)

javascript

```
class Semaphore {
  constructor(maxConcurrent) {
    this.maxConcurrent = maxConcurrent;
    this.currentCount = 0;
    this.waiting = [];
  }

  async acquire() {
    if (this.currentCount < this.maxConcurrent) {
      this.currentCount++;
      return;
    }

    // Wait for available slot
    await new Promise(resolve => {
      this.waiting.push(resolve);
    });
  }

  release() {
    this.currentCount--;

    // Notify next waiting operation
    if (this.waiting.length > 0) {
      const resolve = this.waiting.shift();
      this.currentCount++;
      resolve();
    }
  }

  async execute(operation) {
    await this.acquire();

    try {
      return await operation();
    } finally {
      this.release();
    }
  }
}

// Limit database connections
const dbSemaphore = new Semaphore(20); // Max 20 concurrent DB queries

async function queryDatabase(sql) {
  return await dbSemaphore.execute(async () => {
```

```

const conn = await pool.getConnection();
try {
    return await conn.query(sql);
} finally {
    conn.release();
}
});

// Even with 1000 concurrent requests, max 20 hit DB at once

```

5. Chaos Engineering Principles

What is Chaos Engineering?

Definition: Deliberately inject failures to test system resilience.

Traditional Testing:

"Does it work when everything is fine?" ✓

Chaos Engineering:

"Does it work when things break?" 🤔

Principle: Break things in production (in controlled way)
to find weaknesses BEFORE they cause real outages.

Netflix Chaos Monkey

Chaos Monkey randomly terminates instances:

09:00 - Chaos Monkey starts

09:15 - Kills web-server-23

↓

System response:

- Load balancer detects failure
- Routes traffic to other servers
- Auto-scaling launches replacement
- No user impact!
- ✓ System is resilient

09:30 - Kills database replica-2

↓

System response:

- Master continues serving

- Replica-1 handles read traffic
- New replica provisioned
- ✓ System is resilient

If system can't handle these failures:

- Found a weakness!
- Fix it before real outage

Chaos Engineering Tools

Chaos Monkey Family (Netflix):

Tool	What It Breaks
Chaos Monkey	Randomly terminates instances (VMs, containers)
Chaos Gorilla	Kills entire availability zone (simulates data center failure)
Chaos Kong	Kills entire AWS region (ultimate test)
Latency Monkey	Adds artificial delays (slow network, slow service)
Exception Monkey	Causes applications to throw random exceptions
Conformity Monkey	Finds instances that don't follow best practices

Simple Chaos Monkey Implementation

javascript

```
class ChaosMonkey {
  constructor(options = {}) {
    this.enabled = options.enabled || false;
    this.killProbability = options.killProbability || 0.01; // 1%
    this.latencyProbability = options.latencyProbability || 0.05; // 5%
    this.errorProbability = options.errorProbability || 0.02; // 2%
    this.maxLatency = options.maxLatency || 5000; // 5 seconds
  }

  async maybeInjectFailure(operation) {
    if (!this.enabled) {
      // Chaos disabled, run normally
      return await operation();
    }

    const random = Math.random();

    // Inject instance termination
    if (random < this.killProbability) {
      console.log(`✖️ Chaos Monkey: Simulating instance termination!`);
      throw new Error('CHAOS: Instance terminated');
    }

    // Inject latency
    if (random < this.latencyProbability) {
      const delay = Math.random() * this.maxLatency;
      console.log(`✖️ Chaos Monkey: Injecting ${delay.toFixed(0)}ms latency`);
      await new Promise(resolve => setTimeout(resolve, delay));
    }

    // Inject error
    if (random < this.errorProbability) {
      console.log(`✖️ Chaos Monkey: Injecting random error!`);
      throw new Error('CHAOS: Random service failure');
    }

    // Normal execution
    return await operation();
  }
}

// Usage
const chaosMonkey = new ChaosMonkey({
  enabled: process.env.ENABLE_CHAOS === 'true',
  killProbability: 0.001, // 0.1% chance
  latencyProbability: 0.01, // 1% chance
```

```
errorProbability: 0.005 // 0.5% chance
});

// Wrap service calls
async function callExternalService(data) {
  return await chaosMonkey.maybeInjectFailure(async () => {
    return await fetch('https://api.example.com/endpoint', {
      method: 'POST',
      body: JSON.stringify(data)
    });
  });
}

// During normal operation:
// - Most requests succeed
// - Occasionally: latency injected
// - Rarely: errors or "termination"
// Tests if application handles these failures gracefully!
```

Chaos Engineering Experiments

Experiment 1: Database Failure

javascript

```
class ChaosExperiment {
  constructor(name) {
    this.name = name;
    this.results = {
      started: null,
      ended: null,
      failures: [],
      metrics: {}
    };
  }

  async run(setup, fault, validation, rollback) {
    console.log(`\n${'='.repeat(50)}`);
    console.log(` Starting chaos experiment: ${this.name}`);
    console.log(`=.${repeat(50)}`);

    this.results.started = new Date().toISOString();

    try {
      // Step 1: Setup
      console.log(`\n[1/4] Setup: Establishing baseline...`);
      await setup();

      const baselineMetrics = await this.collectMetrics();
      console.log('Baseline metrics:', baselineMetrics);

      // Step 2: Inject fault
      console.log(`\n[2/4] Injecting fault...`);
      await fault();

      // Step 3: Observe and validate
      console.log(`\n[3/4] Observing system behavior...`);
      await new Promise(resolve => setTimeout(resolve, 30000)); // Observe 30s

      const faultMetrics = await this.collectMetrics();
      console.log('Metrics during fault:', faultMetrics);

      // Validate system behavior
      console.log(`\n[4/4] Validating...`);
      const passed = await validation(baselineMetrics, faultMetrics);

      if (passed) {
        console.log(`✅ Experiment PASSED - System is resilient!`);
      } else {
        console.log(`❌ Experiment FAILED - System not resilient!`);
        this.results.failures.push('System did not handle fault gracefully');
      }
    } catch (error) {
      console.error(`Error during experiment: ${error}`);
      await rollback();
    }
  }
}
```

```
}

} catch (error) {
  console.error(X Experiment failed with error:', error);
  this.results.failures.push(error.message);

} finally {
  // Step 4: Rollback (always execute)
  console.log('\nRolling back...!');
  await rollback();

  this.results.ended = new Date().toISOString();
  this.results.metrics = await this.collectMetrics();

  console.log('\nExperiment completed');
  console.log('='.repeat(50));
}

return this.results;
}

async collectMetrics() {
  // Collect system metrics
  return {
    responseTime: await this.measureResponseTime(),
    errorRate: await this.measureErrorRate(),
    throughput: await this.measureThroughput()
  };
}

async measureResponseTime() {
  // Measure average API response time
  const start = Date.now();
  try {
    await fetch('https://api.example.com/health');
    return Date.now() - start;
  } catch {
    return -1; // Failed
  }
}

async measureErrorRate() {
  // Get error rate from monitoring system
  return 0.01; // 1% error rate
}

async measureThroughput() {
```

```
// Get requests per second
return 1000; // 1000 req/s
}

}

// Experiment: Kill database replica
async function databaseReplicaFailureExperiment() {
const experiment = new ChaosExperiment('Database Replica Failure');

await experiment.run(
// Setup
async () => {
  console.log('Verifying all database replicas are healthy...');

  // Check health of primary and replicas
  },
  // Fault injection
  async () => {
    console.log(`🔥 Terminating database replica...`);

    await terminateInstance('db-replica-1');
  },
  // Validation
  async (baseline, fault) => {
    // System should handle replica failure gracefully

    // Check 1: Response time shouldn't increase much
    if (fault.responseTime > baseline.responseTime * 1.5) {
      console.log(`✖ Response time increased too much!`);
      return false;
    }

    // Check 2: Error rate shouldn't increase
    if (fault.errorRate > baseline.errorRate * 1.2) {
      console.log(`✖ Error rate increased!`);
      return false;
    }

    // Check 3: Throughput should be maintained
    if (fault.throughput < baseline.throughput * 0.9) {
      console.log(`✖ Throughput dropped significantly!`);
      return false;
    }

    console.log(`✅ All validation checks passed!`);
    return true;
  },
}
```

```

// Rollback
async () => {
  console.log('Restoring database replica...');

  await launchInstance('db-replica-1');

  // Wait for replication to catch up
  await waitForReplication();

}

);

return experiment.results;
}

// Run experiment
databaseReplicaFailureExperiment();

```

Chaos Engineering Best Practices

1. START SMALL

- ✓ Test in development first
- ✓ Then staging
- ✓ Finally production (controlled)
- ✗ Don't start with production!

2. HAVE ROLLBACK PLAN

- ✓ Automated rollback
- ✓ Big red button to stop experiment
- ✓ Monitoring in place

3. MINIMIZE BLAST RADIUS

- ✓ Affect small percentage of users
- ✓ Use canary deployments
- ✗ Don't affect all users at once

4. INFORM TEAM

- ✓ Schedule chaos experiments
- ✓ Team knows what's happening
- ✓ Can distinguish chaos from real issues

5. MEASURE EVERYTHING

- ✓ Metrics before, during, after
- ✓ Customer impact
- ✓ System behavior

6. AUTOMATE

- ✓ Run regularly (weekly/monthly)
- ✓ Continuous testing
- ✓ Catch regressions early

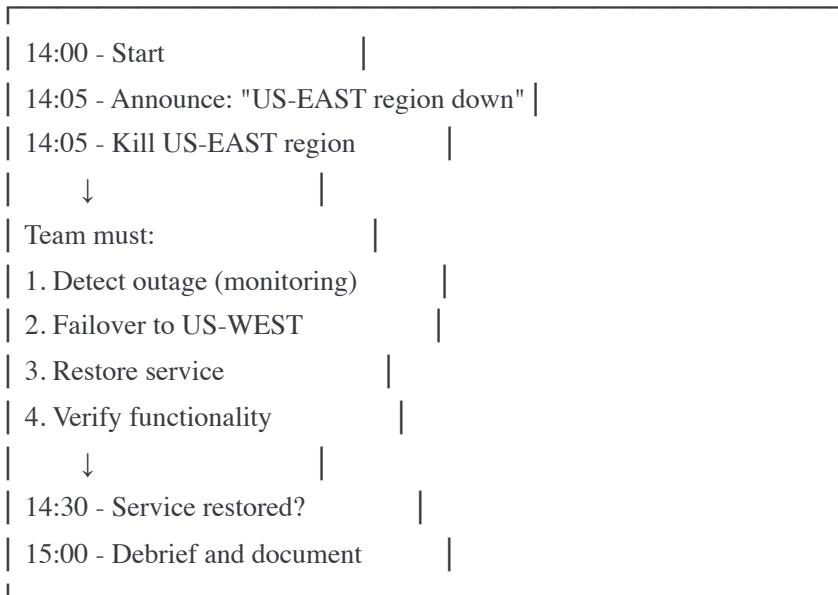
GameDay Exercise

GameDay: Simulated disaster recovery drill

Schedule: First Tuesday of every month, 2-4 PM

Team: Engineers, SRE, Product, Management

Scenario: Region Failure



Learnings:

- What went well?
- What went wrong?
- What can be automated?
- Update runbooks

Combining All Patterns

Resilient Service Client

```
javascript
```

```
class ResilientServiceClient {
  constructor(serviceName, serviceUrl, options = {}) {
    this.serviceName = serviceName;
    this.serviceUrl = serviceUrl;

    // Circuit breaker
    this.circuitBreaker = new CircuitBreaker({
      failureThreshold: options.failureThreshold || 5,
      timeout: options.timeout || 30000
    });

    // Retry with backoff
    this.backoff = new ExponentialBackoff({
      maxRetries: options.maxRetries || 3,
      baseDelay: options.baseDelay || 1000
    });

    // Bulkhead
    this.bulkhead = new Bulkhead(
      serviceName,
      options.maxConcurrent || 10
    );

    // Timeout
    this.requestTimeout = options.requestTimeout || 5000;
  }

  async call(endpoint, data, options = {}) {
    // Layer 1: Bulkhead (limit concurrent requests)
    return await this.bulkhead.execute(async () => {

      // Layer 2: Circuit breaker (fail fast if service down)
      return await this.circuitBreaker.execute(async () => {

        // Layer 3: Retry with exponential backoff
        return await this.backoff.retry(async () => {

          // Layer 4: Timeout (don't wait forever)
          return await this.withTimeout(async () => {

            const response = await fetch(
              `${this.serviceUrl}${endpoint}`,
              {
                method: options.method || 'GET',
                headers: {
                  'Content-Type': 'application/json',
                }
              }
            );
          });
        });
      });
    });
  }
}
```

```
    ...options.headers
  },
  body: data ? JSON.stringify(data) : undefined
)
);

if (!response.ok) {
  throw new Error(`HTTP ${response.status}`);
}

return await response.json();
}, this.requestTimeout);
};

});

});

async withTimeout(operation, timeout) {
  return Promise.race([
    operation(),
    new Promise((_, reject) =>
      setTimeout(() => reject(new Error('Request timeout')), timeout)
    )
  ]);
}

async callWithFallback(endpoint, data, fallback) {
  try {
    return await this.call(endpoint, data);

  } catch (error) {
    console.warn(`${this.serviceName} failed, using fallback:`, error.message);

    // Execute fallback
    if (typeof fallback === 'function') {
      return await fallback();
    }

    return fallback;
  }
}

getHealth() {
```

```
return {
  service: this.serviceName,
  circuitBreaker: this.circuitBreaker.getState(),
  bulkhead: this.bulkhead.getStats()
};

}

}

// Usage
const paymentClient = new ResilientServiceClient(
  'PaymentService',
  'https://payment.example.com',
  {
    maxRetries: 3,
    maxConcurrent: 20,
    requestTimeout: 5000,
    failureThreshold: 5
  }
);

// Call with all resilience patterns applied
async function processPayment(orderId, amount) {
  try {
    const result = await paymentClient.callWithFallback(
      '/charge',
      { orderId, amount },
      // Fallback: Queue for later
      async () => {
        await queuePayment(orderId, amount);
        return { status: 'queued', message: 'Payment queued for processing' };
      }
    );
    return result;
  } catch (error) {
    console.error('Payment processing failed:', error);
    // Ultimate fallback: Save order, notify user
    await saveFailedPayment(orderId, amount);

    return {
      status: 'failed',
      message: 'Payment failed. We will contact you to complete the order.'
    };
  }
}
```

```
}
```

```
// Health check endpoint
app.get('/health', (req, res) => {
  const health = {
    status: 'up',
    services: {
      payment: paymentClient.getHealth(),
      user: userClient.getHealth(),
      inventory: inventoryClient.getHealth()
    }
  };
  // Check if any circuit breakers are open
  const openCircuits = Object.values(health.services)
    .filter(s => s.circuitBreaker.state === 'OPEN');

  if (openCircuits.length > 0) {
    health.status = 'degraded';
    health.degradedServices = openCircuits.map(s => s.service);
  }

  res.json(health);
});
```

Real-World Examples

Example 1: Netflix Resilience

Netflix Architecture Resilience:

1. CIRCUIT BREAKERS

Every service call wrapped in circuit breaker

2. FALLBACKS

Video service down?

- Show cached recommendations
- Show trending list
- Show "Recently Watched"

3. GRACEFUL DEGRADATION

Personalization service down?

- Show generic homepage
- User can still browse and watch

4. BULKHEADS

Separate thread pools per service

If recommendation service slow

→ Doesn't affect streaming

5. CHAOS ENGINEERING

Continuous testing in production

Chaos Monkey runs daily

Simian Army (multiple chaos tools)

Result:

- Can lose entire AWS region

- Still serve 100% of customers

- Availability: 99.99%+

Example 2: Amazon's Resilience

Amazon Architecture Patterns:

1. RETRY WITH BACKOFF

All service calls retry automatically

Exponential backoff prevents thundering herd

2. TIMEOUT EVERYWHERE

Every call has timeout

Don't wait indefinitely for slow service

3. BULKHEAD

Separate resources per service

Cell-based architecture limits blast radius

4. GRACEFUL DEGRADATION

Search service down?

→ Show category pages instead

→ User can still browse

Recommendation service down?

→ Show bestsellers

→ User can still shop

5. STATIC FALLBACKS

Ultimate fallback: Pre-rendered pages

Even if entire backend down, show something useful

Result:

- 99.99% availability during peak (Prime Day, Black Friday)
- Handle 10x normal traffic

Key Takeaways

1. Graceful Degradation:

- Fail partially, not completely
- Core features always work
- Optional features can be disabled
- Better than total failure

2. Circuit Breaker:

- Fail fast when service down
- Three states: CLOSED, OPEN, HALF_OPEN
- Prevents cascading failures
- Automatic recovery testing

3. Retry Logic:

- Exponential backoff (1s, 2s, 4s, 8s...)
- Add jitter (prevent thundering herd)
- Only retry retriable errors
- Combine with circuit breaker

4. Bulkhead Pattern:

- Isolate resources
- Limit concurrent requests
- Prevent cascade failures
- Maintain thread pools per service

5. Chaos Engineering:

- Test failures in production
- Find weaknesses proactively
- Automate failure injection
- Regular GameDay exercises

Practice Problems

1. Design a resilient payment processing system. Which patterns would you use and where?
2. Calculate: With circuit breaker (5 failures trigger open, 30s timeout), how many failed requests before service marked unhealthy?
3. Design chaos experiments to test your system's resilience to:
 - Database failure
 - Network partition
 - Service slowdown
 - Memory pressure

Next Chapter Preview

In Chapter 14, we'll explore **Monitoring and Observability**:

- Logging best practices
- Metrics collection (Prometheus, Grafana)
- Distributed tracing (Jaeger, Zipkin)
- Alerting strategies
- SLIs, SLOs, SLAs

Ready to continue?