

Graceful Degradation

1. The Real Problem This Exists to Solve

When systems experience partial failures or overload, they must choose between complete failure (all features down) or degraded service (critical features work, non-critical features disabled). Graceful degradation ensures systems remain partially operational instead of completely failing.

Real production scenario:

- Netflix during regional AWS outage
- **Without graceful degradation:**
 - Recommendation service down → entire homepage fails to load
 - Cannot render page without recommendations
 - Users see error page
 - 100% service unavailability
 - Customer impact: Entire service appears down
- **With graceful degradation:**
 - Recommendation service down → detected via health check
 - Homepage loads without personalized recommendations
 - Shows "Popular Titles" instead (cached data)
 - Search, playback, account management all work
 - Only recommendation feature degraded
 - Service availability: 95% (only recommendations affected)
 - Customer impact: Slightly worse experience, but can still watch content

The fundamental problem: Complex systems have many dependencies. If we require all dependencies to be healthy for any functionality, a single component failure causes complete outage. Instead, identify core functionality and shed non-critical features during failures.

Without graceful degradation:

- All-or-nothing approach
- Single component failure = complete outage
- Cascading failures
- Poor availability
- Users see error pages

With graceful degradation:

- Core functionality always works
- Non-critical features fail silently or show fallbacks
- Partial availability better than no availability
- Failures contained
- Users still get value from service

2. The Naive / Incorrect Approaches (IMPORTANT)

✗ Incorrect Approach #1: Fail-Fast Without Fallbacks

```
// Incorrect: Fail entire request if any component fails
async function renderHomepage(userId: string): Promise<Homepage> {
```

```

// All components required, no fallbacks
const recommendations = await recommendationService.getRecommendations(userId);
const trending = await trendingService.getTrending();
const userProfile = await profileService.getProfile(userId);
const watchHistory = await historyService.getHistory(userId);

return {
  recommendations, // Required
  trending,        // Required
  userProfile,     // Required
  watchHistory,    // Required
};
}

app.get('/homepage', async (req, res) => {
  try {
    const homepage = await renderHomepage(req.user.id);
    res.json(homepage);
  } catch (error) {
    // Any component failure = complete failure
    res.status(500).json({ error: 'Service unavailable' });
  }
});

```

Why it seems reasonable:

- Simple error handling
- Consistent experience (all features or nothing)
- Easy to implement
- Clear success/failure states

How it breaks:

- Recommendation service down → homepage completely broken
- User cannot see anything despite other services healthy
- Profile service slow → entire page times out
- Single point of failure in every dependency
- Availability = product of all component availabilities
 - If 5 components each 99% available: $0.99^5 = 95\%$ (worse than any single component)

Production symptoms:

- Complete outage despite most services healthy
- Monitoring shows: 4/5 services healthy, but 0% homepage availability
- Users see: "Service unavailable"
- Can't even show basic content despite having data
- Customer support flooded with "site is down" complaints

✗ Incorrect Approach #2: Silent Failures Without User Communication

```

// Incorrect: Return partial data without telling user
async function renderHomepage(userId: string): Promise<Homepage> {
  const homepage: Partial<Homepage> = {};

```

```

    try {
      homepage.recommendations = await
recommendationService.getRecommendations(userId);
    } catch {
      // Silent failure - user doesn't know recommendations unavailable
    }

    try {
      homepage.trending = await trendingService.getTrending();
    } catch {
      // Silent failure
    }

    return homepage as Homepage;
  }

```

Why it seems reasonable:

- Page still loads
- No error thrown
- Gracefully handles failures
- User sees something

How it breaks:

- User sees empty recommendations section, thinks "no recommendations for me"
- User doesn't know it's a temporary failure vs permanent state
- User might think service is broken specifically for them
- No indication that feature will be back
- User experience: Confusion ("Why is my page different today?")

Production symptoms:

- Support tickets: "Where are my recommendations?"
- Users think their account is broken
- No way to distinguish between "no recommendations available" vs "service down"
- Analytics show high bounce rate (users think page broken)

✗ Incorrect Approach #3: All Features Marked "Optional" Without Prioritization

```

// Incorrect: Treat all features equally, no priority
async function renderHomepage(userId: string): Promise<Homepage> {
  const [recommendations, trending, profile, history] = await Promise.allSettled([
    recommendationService.getRecommendations(userId),
    trendingService.getTrending(),
    profileService.getProfile(userId),
    historyService.getHistory(userId),
  ]);

  return {
    recommendations: recommendations.status === 'fulfilled' ? recommendations.value
: null,

```

```

    trending: trending.status === 'fulfilled' ? trending.value : null,
    profile: profile.status === 'fulfilled' ? profile.value : null,
    history: history.status === 'fulfilled' ? history.value : null,
  };
}

```

Why it seems reasonable:

- All failures handled gracefully
- Partial functionality maintained
- No single point of failure

How it breaks:

- Treats critical features (profile) same as non-critical (recommendations)
- User profile failure = user not logged in (critical)
- Recommendations failure = missing suggestions (not critical)
- Both handled identically
- May show page without user name/account info (bad experience)
- No differentiation between must-have and nice-to-have

Production symptoms:

- Homepage loads but user name missing (profile service down)
- Looks like user is not logged in
- Users try to log in again (already logged in)
- Confusion and poor experience

✗ Incorrect Approach #4: Degradation Without Monitoring

```

// Incorrect: Fallbacks implemented but no visibility
async function getRecommendations(userId: string): Promise<Item[]> {
  try {
    return await recommendationService.getRecommendations(userId);
  } catch {
    // Fallback to popular items (good)
    return await getPopularItems(); // But no logging/metrics
  }
}

```

Why it seems reasonable:

- Has fallback (good)
- User experience preserved
- Service keeps working

How it breaks:

- No metrics on how often fallback used
- Team doesn't know recommendation service failing
- Fallback may run for days without team noticing
- Degraded experience becomes permanent
- No alerts, no visibility

- Can't measure impact of degradation

Production symptoms:

- Recommendation service down for 2 weeks
- Nobody notices (fallback working)
- Finally discovered when checking why personalization metrics dropped
- Users experiencing degraded service for weeks
- No incident created, no fix applied

3. Correct Mental Model (How It Actually Works)

Graceful degradation is a tiered approach to handling failures: preserve core functionality, degrade non-critical features, communicate status to users.

The Feature Priority Hierarchy

Tier 1: Critical (Must work, no fallback acceptable)

- User authentication
- Payment processing
- Core content delivery
- Data consistency

Tier 2: Important (Should work, fallback to cached/default)

- User profile display
- Search functionality
- Navigation menus
- Recent history

Tier 3: Enhanced (Nice-to-have, can fail silently)

- Personalized recommendations
- Related content
- Social features
- Live updates

Tier 4: Optional (Fail silently, no user impact)

- Analytics tracking
- A/B tests
- Ad serving
- Telemetry

The Degradation Decision Tree

Component fails

↓

Is it Tier 1 (Critical)?

Yes → Return error, can't proceed

No ↓

↓

Is fallback available?

Yes → Use fallback (cached, default, simplified)

No ↓

↓

Can we omit feature?

Yes → Hide feature, show notice

No → Return error

The Fallback Strategy Hierarchy

1. Cached data (stale but recent)
Example: Show yesterday's recommendations
2. Default/popular data (generic but relevant)
Example: Show trending content
3. Simplified functionality (reduced but working)
Example: Basic search instead of full-text
4. User notification (tell user feature unavailable)
Example: "Recommendations temporarily unavailable"
5. Empty/hidden (silently omit non-critical feature)
Example: Don't show A/B test variant

The Communication Model

Feature degraded:

- Log internally (for ops team)
- Emit metrics (for monitoring)
- Notify user if visible impact
- Degrade gracefully (fallback > error)

User sees:

"Some features temporarily unavailable"
[Show what's working with fallback]

4. Correct Design & Algorithm

Strategy 1: Circuit Breaker + Fallback

```
try:
    result = call_service()
except:
    if has_fallback():
        return fallback()
    else:
        if is_critical():
            raise error
        else:
            return empty
```

Strategy 2: Tiered Timeout

```
Critical services: timeout 1s
Important services: timeout 500ms
Enhanced services: timeout 200ms
Optional services: timeout 100ms

Faster timeout = faster degradation
```

Strategy 3: Parallel with Race

```
primary = call_service()
fallback = call_fallback() after 500ms

return first_success(primary, fallback)
```

Strategy 4: Feature Flags for Dynamic Degradation

```
if feature_flag('recommendations_enabled'):
    show_recommendations()
else:
    show_trending() # Fallback

During incident: flip flag to disable recommendations
```

5. Full Production-Grade Implementation

```
interface DegradationConfig {
    name: string;
    tier: 1 | 2 | 3 | 4;
    timeout: number;
    fallback?: () => Promise<any>;
    required: boolean;
}

interface DegradationMetrics {
    totalCalls: number;
    successfulCalls: number;
    fallbackCalls: number;
    failures: number;
    avgLatency: number;
    degradationActive: boolean;
}

class DegradableService<T> {
    private config: DegradationConfig;
    private metrics: DegradationMetrics;
    private circuitBreaker: CircuitBreaker;
    private latencies: number[] = [];
```

```

constructor(config: DegradationConfig) {
  this.config = config;
  this.metrics = {
    totalCalls: 0,
    successfulCalls: 0,
    fallbackCalls: 0,
    failures: 0,
    avgLatency: 0,
    degradationActive: false,
  };
  this.circuitBreaker = new CircuitBreaker({
    failureThreshold: 5,
    successThreshold: 2,
    timeout: config.timeout,
  });
}

/**
 * Execute service call with degradation support
 */
async execute(fn: () => Promise<T>): Promise<DegradableResult<T>> {
  this.metrics.totalCalls++;
  const startTime = Date.now();

  try {
    // Check circuit breaker
    if (this.circuitBreaker.isOpen()) {
      this.metrics.degradationActive = true;
      return await this.handleDegradation('Circuit breaker open');
    }

    // Execute with timeout
    const result = await this.withTimeout(fn(), this.config.timeout);

    const latency = Date.now() - startTime;
    this.recordLatency(latency);
    this.metrics.successfulCalls++;
    this.circuitBreaker.recordSuccess();
    this.metrics.degradationActive = false;

    return {
      success: true,
      value: result,
      source: 'primary',
      latency,
    };
  } catch (error) {
    const latency = Date.now() - startTime;
    this.recordLatency(latency);
    this.circuitBreaker.recordFailure();
  }
}

```

```

        return await this.handleDegradation(error.message);
    }
}

/**
 * Handle service degradation
 */
private async handleDegradation(reason: string): Promise<DegradableResult<T>> {
    // If service is required (tier 1), propagate error
    if (this.config.required) {
        this.metrics.failures++;
        throw new RequiredServiceUnavailableError(
            `Required service ${this.config.name} unavailable: ${reason}`
        );
    }

    // Try fallback if available
    if (this.config.fallback) {
        try {
            this.metrics.fallbackCalls++;
            const fallbackValue = await this.config.fallback();

            return {
                success: true,
                value: fallbackValue,
                source: 'fallback',
                degraded: true,
                reason,
            };
        } catch (fallbackError) {
            // Fallback also failed
            this.metrics.failures++;
            return {
                success: false,
                source: 'none',
                degraded: true,
                reason: `Primary and fallback failed: ${reason}`,
            };
        }
    }

    // No fallback available, return degraded result
    this.metrics.failures++;
    return {
        success: false,
        source: 'none',
        degraded: true,
        reason,
    };
}

/**

```

```

    * Execute with timeout
    */
    private withTimeout(promise: Promise<T>, timeoutMs: number): Promise<T> {
        return Promise.race([
            promise,
            new Promise<T>((_, reject) =>
                setTimeout(() => reject(new TimeoutError('Service timeout')), timeoutMs)
            ),
        ]);
    }

    /**
     * Record latency for metrics
     */
    private recordLatency(latency: number): void {
        this.latencies.push(latency);
        if (this.latencies.length > 100) {
            this.latencies.shift();
        }

        const sum = this.latencies.reduce((a, b) => a + b, 0);
        this.metrics.avgLatency = sum / this.latencies.length;
    }

    /**
     * Get metrics
     */
    getMetrics(): DegradationMetrics {
        return { ...this.metrics };
    }

    /**
     * Check if currently degraded
     */
    isDegraded(): boolean {
        return this.metrics.degradationActive || this.circuitBreaker.isOpen();
    }
}

interface DegradableResult<T> {
    success: boolean;
    value?: T;
    source: 'primary' | 'fallback' | 'none';
    degraded?: boolean;
    reason?: string;
    latency?: number;
}

class CircuitBreaker {
    private state: 'closed' | 'open' | 'half-open' = 'closed';
    private failureCount = 0;
    private successCount = 0;

```

```

private lastFailureTime?: number;
private config: {
  failureThreshold: number;
  successThreshold: number;
  timeout: number;
};

constructor(config: { failureThreshold: number; successThreshold: number; timeout:
number }) {
  this.config = config;
}

recordSuccess(): void {
  if (this.state === 'half-open') {
    this.successCount++;
    if (this.successCount >= this.config.successThreshold) {
      this.state = 'closed';
      this.failureCount = 0;
      this.successCount = 0;
    }
  } else {
    this.failureCount = 0;
  }
}

recordFailure(): void {
  this.failureCount++;
  this.lastFailureTime = Date.now();

  if (this.failureCount >= this.config.failureThreshold) {
    this.state = 'open';
    this.successCount = 0;
  }
}

isOpen(): boolean {
  if (this.state === 'open') {
    // Check if timeout elapsed, move to half-open
    if (this.lastFailureTime && Date.now() - this.lastFailureTime >
this.config.timeout) {
      this.state = 'half-open';
      return false;
    }
    return true;
  }
  return false;
}
}

class RequiredServiceUnavailableError extends Error {
  constructor(message: string) {
    super(message);
  }
}

```

```

        this.name = 'RequiredServiceUnavailableError';
    }
}

class TimeoutError extends Error {
    constructor(message: string) {
        super(message);
        this.name = 'TimeoutError';
    }
}

// Example: Homepage with graceful degradation
interface Homepage {
    user: UserProfile;           // Tier 1: Required
    content: Content[];          // Tier 1: Required
    recommendations?: Item[];     // Tier 3: Enhanced (optional)
    trending?: Item[];           // Tier 2: Important (fallback available)
    social?: SocialFeed;         // Tier 3: Enhanced (optional)
    ads?: Ad[];                  // Tier 4: Optional (can fail silently)
    degradationNotice?: string;  // Shows if any service degraded
}

// Service wrappers with degradation
const recommendationService = new DegradableService<Item[]>({
    name: 'recommendations',
    tier: 3,
    timeout: 500,
    required: false,
    fallback: async () => {
        // Fallback to popular items from cache
        return await cache.get('popular_items') || [];
    },
});

const trendingService = new DegradableService<Item[]>({
    name: 'trending',
    tier: 2,
    timeout: 1000,
    required: false,
    fallback: async () => {
        // Fallback to cached trending
        return await cache.get('trending_items') || [];
    },
});

const userProfileService = new DegradableService<UserProfile>({
    name: 'user_profile',
    tier: 1,
    timeout: 2000,
    required: true, // No fallback, must work
});

```

```

const contentService = new DegradableService<Content[]>({
  name: 'content',
  tier: 1,
  timeout: 2000,
  required: true,
});

const socialService = new DegradableService<SocialFeed>({
  name: 'social',
  tier: 3,
  timeout: 300,
  required: false,
  fallback: async () => null, // Just hide social feed
});

const adService = new DegradableService<Ad[]>({
  name: 'ads',
  tier: 4,
  timeout: 100,
  required: false,
  fallback: async () => [], // No ads if service down (acceptable)
});

// Render homepage with graceful degradation
async function renderHomepage(userId: string): Promise<Homepage> {
  const degradedServices: string[] = [];

  // Tier 1: Required services (will throw if fail)
  const [userResult, contentResult] = await Promise.all([
    userProfileService.execute(() => fetchUserProfile(userId)),
    contentService.execute(() => fetchContent()),
  ]);

  if (!userResult.success || !contentResult.success) {
    throw new Error('Critical services unavailable');
  }

  // Tier 2-4: Optional services (degrade gracefully)
  const [recommendationsResult, trendingResult, socialResult, adsResult] = await
Promise.all([
    recommendationService.execute(() => fetchRecommendations(userId)),
    trendingService.execute(() => fetchTrending()),
    socialService.execute(() => fetchSocialFeed(userId)),
    adService.execute(() => fetchAds(userId)),
  ]);

  // Track degraded services
  if (recommendationsResult.degraded) degradedServices.push('recommendations');
  if (trendingResult.degraded) degradedServices.push('trending');
  if (socialResult.degraded) degradedServices.push('social');
  if (adsResult.degraded) degradedServices.push('ads');
}

```

```

return {
  user: userResult.value!,
  content: contentResult.value!,
  recommendations: recommendationsResult.success ? recommendationsResult.value :
undefined,
  trending: trendingResult.success ? trendingResult.value : undefined,
  social: socialResult.success ? socialResult.value : undefined,
  ads: adsResult.success ? adsResult.value : undefined,
  degradationNotice: degradedServices.length > 0
    ? `Some features temporarily unavailable: ${degradedServices.join(', ')}`
    : undefined,
};
}

// Express route
app.get('/homepage', async (req, res) => {
  try {
    const homepage = await renderHomepage(req.user.id);

    // Add header indicating degradation
    if (homepage.degradationNotice) {
      res.setHeader('X-Service-Degraded', 'true');
      res.setHeader('X-Degraded-Services', homepage.degradationNotice);
    }

    res.json(homepage);
  } catch (error) {
    if (error instanceof RequiredServiceUnavailableError) {
      res.status(503).json({
        error: 'Service temporarily unavailable',
        message: 'Core services are down, please try again later',
      });
    } else {
      res.status(500).json({ error: 'Internal error' });
    }
  }
});

// Metrics endpoint
app.get('/metrics/degradation', (req, res) => {
  const services = [
    { name: 'recommendations', service: recommendationService },
    { name: 'trending', service: trendingService },
    { name: 'user_profile', service: userProfileService },
    { name: 'content', service: contentService },
    { name: 'social', service: socialService },
    { name: 'ads', service: adService },
  ];

  const metrics = services.map(({ name, service }) => {
    const m = service.getMetrics();
    return {

```

```

        name,
        degraded: service.isDegraded(),
        successRate: (m.successfulCalls / Math.max(m.totalCalls, 1) * 100).toFixed(2)
+ '%',
        fallbackRate: (m.fallbackCalls / Math.max(m.totalCalls, 1) * 100).toFixed(2) +
'%',
        avgLatencyMs: m.avgLatency.toFixed(0),
    });
});

res.json({ services: metrics });
});

// Health check with degradation status
app.get('/health', (req, res) => {
    const criticalServices = [userService, contentService];
    const allHealthy = criticalServices.every(s => !s.isDegraded());

    if (allHealthy) {
        res.status(200).json({ status: 'healthy' });
    } else {
        res.status(200).json({
            status: 'degraded',
            message: 'Some non-critical features may be unavailable'
        });
    }
});

```

6. Correct Usage Patterns (Where This Shines)

Pattern 1: E-Commerce with Tiered Features

```

// Critical: Checkout must always work
const checkoutResult = await checkoutService.execute(() => processCheckout(order));
if (!checkoutResult.success) {
    throw new Error('Cannot process checkout');
}

// Important: Product details with fallback
const detailsResult = await productService.execute(() => getProductDetails(id));
const details = detailsResult.success ? detailsResult.value : getCachedDetails(id);

// Optional: Recommendations
const recsResult = await recommendationService.execute(() =>
getRecommendations(userId));
if (recsResult.success) {
    page.recommendations = recsResult.value;
}
// If fails, just don't show recommendations

```

Pattern 2: Multi-Region Failover

```
const primaryRegion = new DegradableService({
  name: 'us-east-1',
  tier: 1,
  timeout: 1000,
  fallback: async () => {
    // Fallback to secondary region
    return await callRegion('us-west-2');
  },
});
```

Pattern 3: Cached Stale Data Fallback

```
const liveDataService = new DegradableService({
  name: 'live-data',
  tier: 2,
  timeout: 500,
  fallback: async () => {
    const cached = await cache.get('data');
    if (cached) {
      return { ...cached, stale: true };
    }
    throw new Error('No cached data available');
  },
});
```

7. Failure Modes & Edge Cases

Fallback Service Also Fails

Problem: Primary fails, fallback also fails.

Mitigation: Tertiary fallback (cached data), or accept failure for non-critical.

Degraded State Becomes Permanent

Problem: System runs in degraded mode for days, team doesn't notice.

Mitigation: Alert on degradation >30 minutes, force review.

Cascading Degradation

Problem: Service A degrades → Service B depends on A → B also degrades.

Mitigation: Break dependencies, use cached data, circuit breakers.

User Confusion from Inconsistent State

Problem: User sees different features on different requests (A/B testing appearance).

Mitigation: Clear messaging when degraded, consistent degradation per user session.

8. Performance Characteristics & Tradeoffs

Availability

Without degradation:

- Availability = $99\% \times 99\% \times 99\% \times 99\% = 96\%$

With degradation:

- Critical: 99% (checkout)
- Optional: 95% (recommendations)
- Overall: 99% (only critical matters)

Latency

Fail-fast: 5s timeout → 5s latency before error

Degradation: 200ms timeout → immediate fallback → 250ms total

Complexity

Trade-off: More complex code for better availability

9. Foot-Guns & Common Mistakes (DO NOT SKIP)

Mistake 1: No Differentiation Between Feature Tiers

Fix: Explicitly categorize features as required/important/optional.

Mistake 2: Stale Cache Without TTL

Fix: Expire cached fallbacks, show "Last updated X hours ago".

Mistake 3: No User Communication

Fix: Show banner: "Some features temporarily unavailable".

Mistake 4: Degradation Without Monitoring

Fix: Alert on fallback usage >5%, dashboard showing degradation.

Mistake 5: Fallback More Expensive Than Primary

Fix: Fallback should be cheaper/simpler (cached data, default values).

10. When NOT to Use This (Anti-Patterns)

Anti-Pattern 1: Financial Transactions

Don't degrade accuracy/consistency for payments.

Anti-Pattern 2: All Features Critical

If all features equally important, degradation doesn't help.

Anti-Pattern 3: Data Integrity Operations

Don't degrade writes/updates that must be consistent.

11. Related Concepts (With Contrast)

Circuit Breaker

Together: Circuit breaker detects failures, degradation provides fallback.

Retry Logic

Difference: Retry attempts same operation. Degradation uses fallback.

Failover

Difference: Failover switches to backup system. Degradation reduces functionality.

12. Production Readiness Checklist

Feature Classification

- ☐ Identify tier 1 (critical) features
- ☐ Identify tier 2-3 (degradable) features
- ☐ Identify tier 4 (optional) features
- ☐ Document dependencies per feature

Fallback Implementation

- ☐ Cached data fallbacks for tier 2
- ☐ Default/popular data for tier 3
- ☐ Silent omission for tier 4
- ☐ Test each fallback independently

User Communication

- ☐ Degradation notice in UI
- ☐ HTTP headers indicating degradation
- ☐ Status page updates during incidents
- ☐ Clear messaging per feature

Monitoring

- ☐ Track degradation state per service
- ☐ Alert on degradation >5 minutes
- ☐ Dashboard showing fallback usage
- ☐ SLO for critical services only

Testing

- ☐ Chaos testing (disable services randomly)
- ☐ Verify fallbacks work
- ☐ Load test degraded state

- ☐ Ensure critical features always work