📚 RESUMO COMPLETO: DESIGN TACTICS - DISPONIBILIDADE & PERFORMANCE

RAS-slidesEN-9-ARCH-Tactics.pdf - Análise Estruturada Completa

Data: 17 de Janeiro de 2026
Nível: Mestrado em Engenharia Informática - Universidade do Minho
Autor Base: JM Fernandes
Compilação: Análise Completa, Rigorosa e Estruturada

ÍNDICE

INTRODUÇÃO A TACTICS

Motivação: Conexão entre Requisitos e Design

Facto:

> "**Architects precisam de técnicas para alcançar qualidades particulares**."

Significado:

- Requisitos especificam qualidades desejadas
- Arquiteto precisa saber COMO alcançar essas qualidades
- Tactics são as técnicas para isso

O Propósito Central

Definição de Propósito:

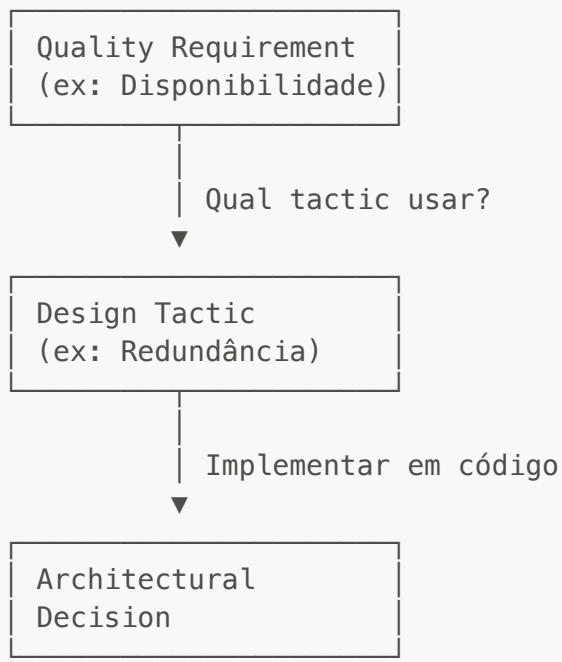
> "**Tactics são usadas pelo arquiteto para criar um design.**"

> "**Tactics conectam requisitos de atributos de qualidade com decisões arquiteturais.**"

Significado:

- Bridge entre especificação (o que) e implementação (como)
- Requisitos de qualidade são transformados em decisões design
- Cada tactic é uma opção de design concreta

Exemplo Conceptual



Contexto: Stimulus → Response

Modelo Conceptual:

```
STIMULUS (Evento chega)
    ↓
TACTICS (Como responder?)
    ↓
RESPONSE (Sistema responde dentro requisitos)
```

Exemplo Concreto:

```
STIMULUS: "Servidor falha"
TACTICS: Redundância + Health Monitoring + Recovery
RESPONSE: "Sistema recupera automaticamente"
```

DEFINIÇÃO DE TACTIC

Conceito Formal

Definição:

> "Uma **tactic é uma decisão de design que impacta em atributos de qualidade específicos**."

Características:

- Design decision (escolha consciente)
- Impacto identificável (afeta qualidades)
- Específica (não genérica)
- Testável (resulta em behavior observável)

Tipos de Decisões

Sistema Design = Coleção de Decisões:

```
├─ Decisões de Qualidade (Tactics)
│  ├─ Aumentar disponibilidade (redundância)
│  ├─ Melhorar performance (caching)
│  ├─ Aumentar segurança (criptografia)
│  └─ ...
│
└─ Decisões de Funcionalidade
   ├─ Como implementar feature X
   ├─ Que algoritmo usar
   └─ Que dados estrutura
```

Facto:

> "**Algumas decisões ajudam controlar as respostas de atributos de qualidade.**" "**Outras garantem realização de funcionalidade do sistema.**"

Significado:

- Separação clara entre qualidade e funcionalidade
- Tactics focam em qualidade
- Funcionalidade é separada

Tactic Como Design Option

Conceito:

> "**Cada tactic é uma opção de design para o arquiteto.**"

Significado:

- Não há uma única forma de resolver

- Múltiplas options (tactics) para mesma qualidade
- Arquiteto escolhe baseado em contexto

Exemplo:

> "Uma tactic pode **introduzir redundância para aumentar disponibilidade** do sistema."

Alternativas:

```
Disponibilidade pode ser aumentada por:
- Redundância (backup hardware)
- Health monitoring (detectar falhas)
- Recovery rápido (reconstruir state)
- Combinação dos 3

Escolha do arquiteto baseado em:
- Custo
- Criticidade
- Ambiente
- Requisitos específicos
```

TIPOS DE TACTICS

Categoria Geral

Existem 6 categorias principais de tactics:

#	Categoria	Objetivo	Foco
1	**Availability**	Manter sistema operacional	Falhas, recuperação
2	**Modifiability**	Fácil mudar sistema	Mudanças, manutenção
3	**Performance**	Responder rapidamente	Latência, throughput
4	**Security**	Proteger dados/acesso	Criptografia, autenticação
5	**Testability**	Fácil testar	Testes, verificação
6	**Usability**	Fácil usar	Interface, UX

Nota: Este ficheiro focua em **Availability** e **Performance**

DISPONIBILIDADE (AVAILABILITY)

Definição

Conceito Formal:

> "**Availability quantifica a percentagem de tempo durante o qual um sistema está operacional e funcionando corretamente.**"

Fórmula:

$$\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}$$

Componentes:

- **MTBF** = Mean Time Between Failures (tempo médio entre falhas)
- **MTTR** = Mean Time To Repair (tempo médio para reparar)

Interpretação:

```
Alta MTBF (falhas raras):
- Sistema é fiável
- Tempo entre falhas é longo

Baixa MTTR (recuperação rápida):
- Quando falha, recupera rapidamente
- Impacto minimizado

Exemplo:
┌──────────────────────────────────────────┐
│ Falha #1          Falha #2       Falha #3 │
├──────────┬──────────────────┬──────────┬──┤
│   MTBF   │      MTBF        │  MTBF    │  │
│  10 dias │     10 dias      │ 10 dias  │  │
│  MTTR: 1h│     MTTR: 1h     │MTTR: 1h  │  │
│          │                  │          │  │
│ Avail = 10*24/(10*24 + 1) ≈ 99.6%  │
└──────────────────────────────────────────┘
```

Importância Prática

Ramificações de Availability:

1. **Confiança de Utilizadores**
   - Sistema confiável = utilizadores confiam
   - Sistema instável = utilizadores não voltam
2. **Valor da Informação**
   - Dados inacessíveis = valor zero
   - Dados sempre disponíveis = valor máximo
3. **Eficiência de Processos**
   - Sistema down = processos param
   - Sistema operacional = processos continuam

4. **Produtividade Organizacional**
    - Downtime = perda produtividade
    - Uptime = máxima produtividade

Exemplo Real:

```
Uma loja online com 99% availability:
- 3,65 dias de downtime por ano
- Cada dia de downtime = perda vendas

Com 99.9% availability:
- 8,76 horas de downtime por ano
- Muito melhor para negócio
```

Fault vs Failure

Distinção Crítica:

Fault (Falha interna)

- Componente interno não funciona
- Não necessariamente visível para user
- Pode ser mascarado

Exemplo:

```
Disco servidor tem bad sector
→ Fault (hardware falha)
→ Mas RAID replica dados
→ User não percebe
```

Failure (Falha observável)

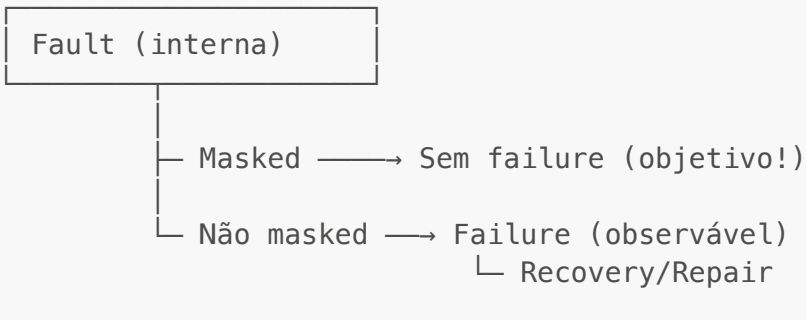
- Sistema não entrega serviço conforme especificação
- **Observável pelo utilizador**
- Impacto direto

Exemplo:

```
Página web não carrega
→ Failure (user percebe)
→ Possível causa: disco bad sector (fault)
```

Relação Fault → Failure

Modelo Conceptual:



Objetivo de Availability Tactics:

> "**Availability tactics têm objectivo de evitar faltas (faults) de se tornarem falhas (failures), ou pelo menos reduzir efeitos da falha e tornar reparação possível.**"

Três Abordagens:

1. **Prevent Failure:** Fault não vira failure (mascarar)
2. **Reduce Effect:** Falha existe, mas impacto minimizado
3. **Enable Repair:** Falha identificada e reparada rapidamente

---

Tres Pilares de Availability

Facto:

> "**Todas abordagens endereçando availability envolvem algum tipo de:**
>
> 1. **Redundancy** (duplicação)
> 2. **Health Monitoring** (deteção)
> 3. **Recovery** (recuperação)
>
> **Monitorização ou recuperação é automática ou manual.**"

Pilar 1: Redundancy (Redundância)

Conceito:

- Múltiplas cópias de componente
- Se um falha, outro continua
- Invisível para utilizador

Tipos:

- Hardware redundancy (múltiplos servidores)
- Software redundancy (múltiplas instâncias)
- Data redundancy (backups)

Tradeoff:

```
Vantagem: Alta disponibilidade
Desvantagem: Custo (múltiplas cópias)
```

Pilar 2: Health Monitoring (Monitorização)

Conceito:

- Detectar quando falha ocorre
- Rápida detecção = rápida recuperação

Métodos:

- Ping/echo (está vivo?)
- Heartbeat (bate o coração?)
- Exceptions (levantou alerta?)

Pilar 3: Recovery (Recuperação)

Conceito:

- Quando falha detectada, recuperar
- Automático (sistema reage) ou manual (admin intervém)

Tipos:

- Failover (mudar para backup)
- Restart (reiniciar componente)
- Rollback (voltar a estado anterior)

FAULT DETECTION TACTICS

Overview

Objetivo:

> "Detectar quando **fault ocorre** para poder **reagir rapidamente**."

Desafio:

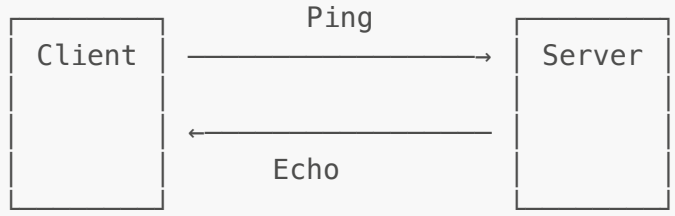
- Sistema ainda funciona com alguns faults
- Fault pode ser silencioso (não levanta alerta)
- Detecção rápida é crítica

Tactic 1: Ping/Echo

Descrição:

> "Um componente **emite um ping** e **espera receber um echo** do componente sob escrutínio."

Mecanismo:




```
IF Echo recebido rapidamente → Operacional ✓
IF Echo não recebido → Falha detectada ✗
```

Propósito:

> "Pode ser usado por **clients para assegurar que um objeto de servidor e o caminho de comunicação estão operacionais**."

Vantagens:

- Simples de implementar
- Rápido
- Verifica conectividade

Exemplos:

- ICMP Ping em rede
- HTTP heartbeat (GET /health)
- Database connection test

Desvantage:

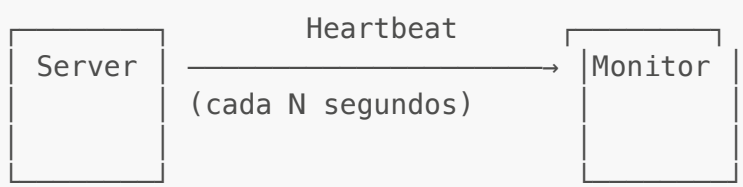
- Pode ter falsos positivos (latência alta = parece falha)
- Não detecta falhas internas (servidor responde mas mal)

Tactic 2: Heartbeat

Descrição:

> "Um componente **emite uma mensagem heartbeat periodicamente** e outro componente **escuta por ela**."

Mecanismo:



```
IF Heartbeat recebido → Operacional ✓
IF Heartbeat FALHA → Alertar componente recovery
```

Operação Quando Falha:

> "Se heartbeat falha, **componente original é assumido ter falhado** e um **componente de correção de faults é notificado**."

Fluxo:

```
Heartbeat falha por 3 batidas
→ Monitor assume: Servidor down
→ Notifica componente recovery
→ Recovery inicia: Failover, restart, etc
```

Vantagens:

- Detecta quando componente some completamente
- Permite reação automática
- Configurável (frequência de batidas)

Desvantagens:

- False positives (network lag)
- Overhead network (mensagens constantes)

Exemplo Real:

```
Kubernetes:
- Pod emite heartbeat periodicamente
- Master monitora
- Se heartbeat falha 3 vezes → Pod declarado dead
- Master inicia novo Pod
```

Tactic 3: Exceptions

Descrição:

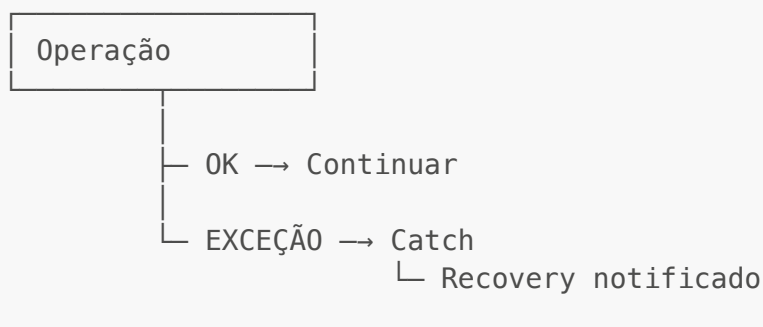
> "Um método para detectar faults é **encontrar uma exceção**, que é **levantada quando uma classe de fault é reconhecida**."

Mecanismo:

```
try {
  risky_operation()
} catch (FaultException e) {
```

```
  // Fault detectado!
  notify_recovery_component()
}
```

Operação:



Vantagens:

- Detecta erros específicos
- Integrado no código (não network overhead)
- Permite handling granular

Desvantagens:

- Requer throwing exceptions apropriadas
- Nem todos faults levantam exceções
- Silent failures ainda possíveis

Exemplo:

```
try {
  db.connect()  // Falha de conexão
} catch (SQLException e) {
  logger.error("DB down")
  failover_to_backup_db()
}
```

FAULT RECOVERY TACTICS

Overview

Objetivo:

> "Quando fault detectado, **recuperar e minimizar impacto**."

Estratégias:

- Corrigir o fault
- Voltar a estado consistente
- Minimizar dados perdidos

Tactic 1: Voting

Descrição:

> "**Processos rodando em processadores redundantes cada um toma o input, computa output**, que é enviado a um **voter**."

Mecanismo:

```
Input
  │
  ├──→ Processor 1 ──→ Output A
  ├──→ Processor 2 ──→ Output B
  └──→ Processor 3 ──→ Output C
       │
       ▼
    Voter
       │
       ├─ Majority Rules: Qual output apareça 2+ vezes?
       │  A=Output, B=Output, C=Diferentes
       │  Resultado: Output (2 voters concordam)
       │
       └─ Preferred Component: Usar saída de processor favorito
          (se válida)
```

Algoritmos Populares:

1. **Majority Rules (Regra Maioria)**
    - Múltiplos processadores computam
    - Voter escolhe resposta que aparece mais
    - Tolera 1 processador falhando (se N=3)
2. **Preferred Component**
    - Um processor é "preferido"
    - Se preferido retorna resultado, use-o
    - Se preferido falha, use outro

Exemplo Prático:

```
3 Processadores calculam: 2+2
P1 responde: 4 ✓
P2 responde: 4 ✓
P3 responde: 5 ✗ (falhou)
```

```
Voter: Maioria diz 4 → Usar 4
```

Aplicação:

- Sistemas críticos (voo, medicina)
- Algoritmos complexos
- Processadores independentes

Tradeoff:

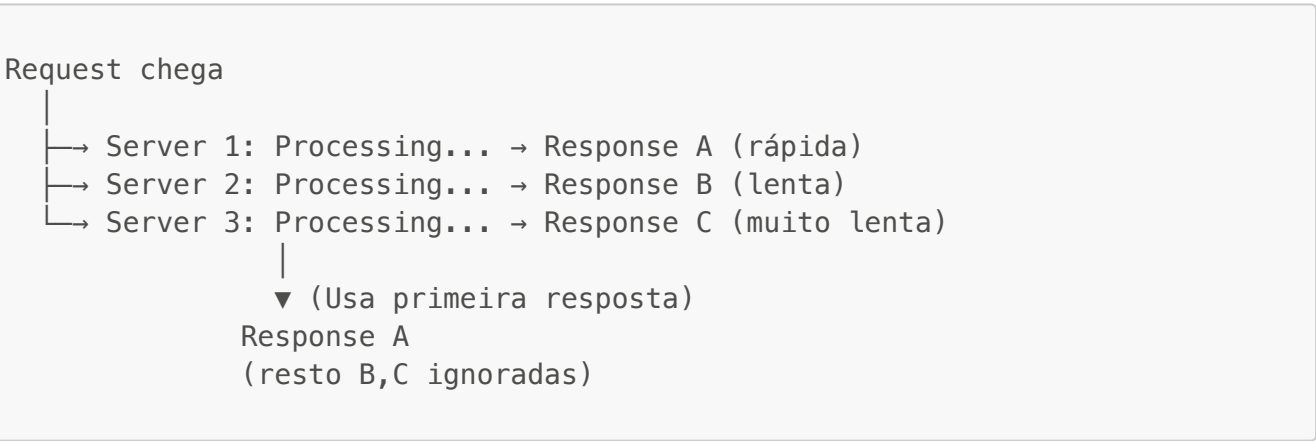
```
Vantagem: Tolerância a falhas
Desvantagem: 3x custo (3 processadores), overhead voting
```

Tactic 2: Active Redundancy

Descrição:

> "**Todos componentes redundantes respondem a eventos em paralelo.**" "**Resposta de um componente é usada (normalmente a primeira), resto são descartadas.**"

Mecanismo:



Características:

- **Todos processam simultaneamente**
- **Apenas primeira resposta é usada**
- **Resto são descartadas**

Vantagem:

- Resposta rápida (usa a mais rápida)
- Falha de um não afeta (outros continuam)

Desvantagem:

- Computação desperdiçada (B,C processam mas ignoradas)

- Custo: Múltiplos servidores + overhead

Quando Usar:

- Latência é crítica
- Custo computação não é limitante
- Sistema pode tolerar desperdício

Exemplo Real:

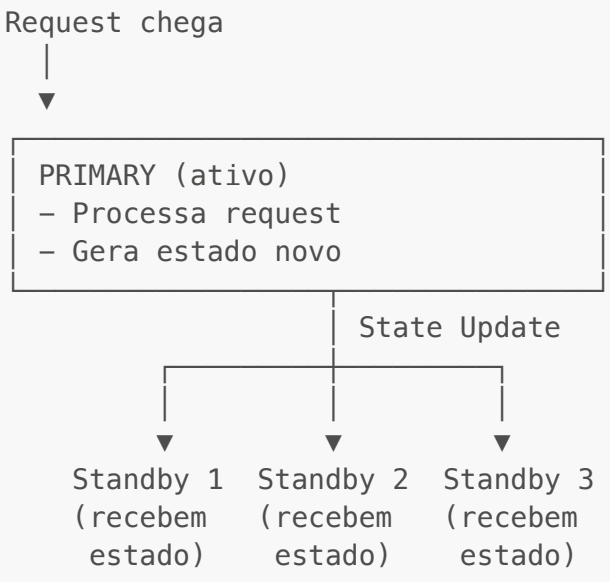
```
CDN (Content Delivery Network):
- Request vai para 3 servidores geograficamente distribuídos
- Servidor mais próximo responde rápido
- Outros 2 respostas ignoradas
- User recebe resposta rápida
```

Tactic 3: Passive Redundancy

Descrição:

> "**Um componente (o primary) responde a eventos e informa outros componentes (os standbys) de atualizações de estado** que eles devem fazer."

Mecanismo:



Operação Normal:

- Primary processa
- Standbys apenas escutam e armazenam state updates
- Users não percebem standbys

Operação Quando Primary Falha:

> "Quando fault ocorre, **sistema deve primeiro assegurar que state em backup é suficientemente fresco antes de resumir serviços**."

Fluxo:

```
Primary falha (detectado por heartbeat)
  ↓
Monitor seleciona um Standby → Promove a Primary
  ↓
Verifica: State é fresco? (quando foi última atualização?)
  ↓
Se SIM → Resume serviços imediatamente
Se NÃO → Sincroniza estado faltante, depois resume
```

Vantagem:

- Menos desperdício computação (standby não processa)
- Custo menor que active

Desvantagem:

- Delay no failover (sincronizar estado)
- State loss possível (se atualização não chegou standby)

Exemplo Real:

```
MySQL Master-Slave Replication:
- Master processa INSERT/UPDATE/DELETE
- Slave recebe log de mudanças (replication)
- Se Master falha, Slave promovido
- Mas replication lag pode causar perda dados
```

Tradeoff: Consistency vs Availability

```
Passive Redundancy:
- Consistency: Dados podem estar desatualizados em standby
- Availability: Failover demora mais

vs

Active Redundancy:
- Consistency: Todos têm dados iguais (ou quase)
- Availability: Resposta instantânea
```

FAULT PREVENTION TACTICS

Overview

Objetivo:

> "**Evitar que faults ocorram**, não apenas reagir quando ocorrem."

Filosofia:

- Prevenção > Reação
- Evitar falha = melhor que recuperar falha

Tactic 1: Removal From Service

Descrição:

> "Esta tactic **remove um componente de operação para realizar atividades que previnem falhas**."

Exemplo Específico:

> "**Rebooting um componente para prevenir memory leaks de causarem falha**."

Cenário:

```
Memory de servidor aumenta gradualmente:
┌──────────────────────────────────────┐
│ Memory Usage Over Time               │
│                                      │
│ ▓▓▓▓▓                                │
│ ▓▓▓▓▓▓▓▓                             │
│ ▓▓▓▓▓▓▓▓▓▓▓▓                         │
│ ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓                    │
│ ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓ (OUT OF MEM!) │
└──────────────────────────────────────┘

Solução: Reboot periódico antes atingir limite
```

Processo:

```
1. Detector: Memory > Threshold?
2. SIM → Iniciar graceful shutdown
3. Redireciona requests para outro servidor
4. Aguarda requests atuais terminarem
5. Shutdown servidor antigo
6. Restart servidor (limpo memory)
7. Re-adiciona ao pool
```

Tipos de Prevenção:

1. **Preventive Maintenance**

- Reboot regular (ex: semanal)
- Limpar cache
- Atualizações patches

2. **Reactive Prevention**
    - Detector monitora métrica
    - Quando vai falhar → intervém
    - Antes de atingir breaking point

Vantagem:

- Previne falhas antes ocorrer
- User nunca vê falha

Desvantagem:

- Brief downtime durante reboot
- Pode ser disruptivo se timing incorreto

Tactic 2: Transactions

Descrição:

> "Uma **transaction é bundling de vários passos sequenciais** de forma que o **bundle possa ser desfeito de uma vez**."

Propriedades ACID:

```
TRANSACTION: [Step 1] → [Step 2] → [Step 3]

Se TUDO OK:
  COMMIT (todas mudanças permanentes)

Se ALGUMA FALHA:
  ROLLBACK (todas mudanças desfeitas)

NÃO EXISTE: 1 OK, 2 OK, 3 FALHA (com dados inconsistentes)
```

Mecanismo:

```
Begin Transaction
  │
  ├─ Step 1: Validar input
  ├─ Step 2: Update Database
  ├─ Step 3: Notify User
  │
  ├─ TUDO OK? COMMIT
```

```
    │
    └─ FALHA em Step 2? ROLLBACK (desfaz Step 1)
```

Aplicação 1: Prevent Data Corruption

> "**Transactions são usadas para prevenir dados de serem afetados se um passo em um processo falha**."

Exemplo:

```
Transferência bancária: A→B de 100€

SEM TRANSACTION:
  1. Withdraw 100 from A ✓
  2. Deposit 100 to B ✗ (FALHA!)
  Resultado: 100€ perdido!

COM TRANSACTION:
  1. Withdraw 100 from A
  2. Deposit 100 to B

  SE tudo OK → COMMIT (ambos acontecem)
  SE FALHA → ROLLBACK (nenhum acontece)

  Resultado: 100€ não perdido, contas consistentes
```

Aplicação 2: Prevent Data Collisions

> "**Também previnem colisões entre múltiplas threads acessando dados simultaneamente**."

Scenario:

```
SEM TRANSACTION (2 threads simultaneamente):
  Thread 1: Read balance (100)
  Thread 2: Read balance (100)
  Thread 1: Withdraw 50 → balance = 50, Write
  Thread 2: Withdraw 50 → balance = 50, Write

  Final balance: 50 (incorreto!)
  (Deveria ser: 0)

COM TRANSACTION (locking):
  Thread 1: BEGIN
    Read balance (100)
    Update (LOCK data)
    Write balance = 50
  COMMIT

  Thread 2: BEGIN
    Read balance (50)
```

```
    Update (aguarda LOCK libertar)
    Write balance = 0
  COMMIT

  Final balance: 0 (correto!)
```

Vantagem:

- Garante data consistency
- Previne corrupção dados
- Simples de usar (suportado em BD)

Desvantagem:

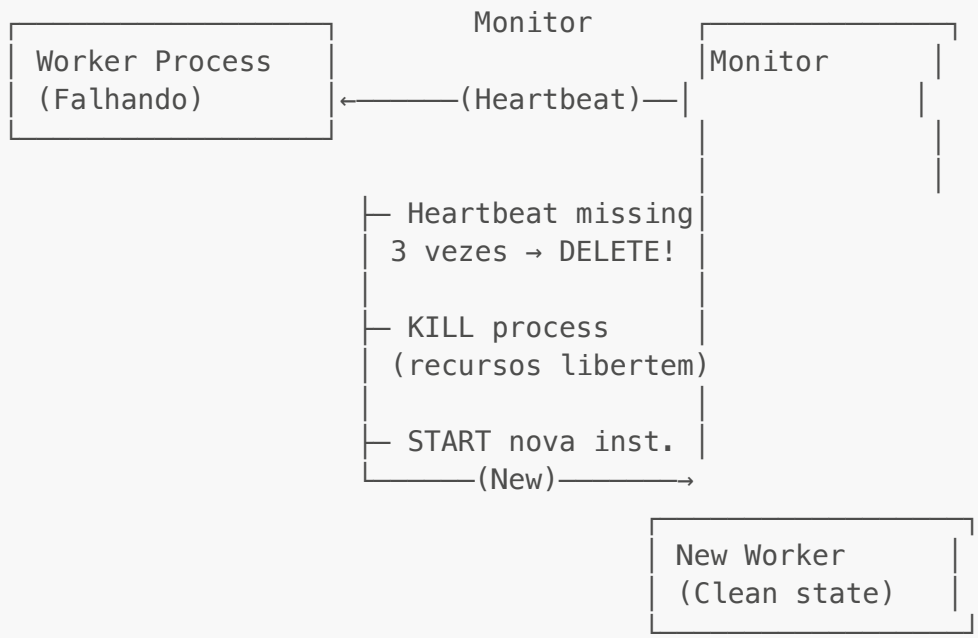
- Performance (locking = contention)
- Deadlocks possíveis

Tactic 3: Process Monitor

Descrição:

> "Uma vez fault num processo é detectado, um **monitoring process pode deletar o processo não-performante**. Uma **nova instância do processo precisa ser criada**."

Mecanismo:



Ciclo:

```
1. MONITOR deteta: Worker não responde (heartbeat falta)
2. MONITOR deleta: Kill process (liberta recursos)
```

```
3. MONITOR cria: Nova instância do processo
4. Nova instância: Começa fresco (estado limpo)
```

Aplicação:

- Worker processes que ficam stuck
- Processes com memory leaks
- Processes em deadlock

Vantagem:

- Limpa automaticamente processos problemáticos
- Nova instância = estado fresco
- Combinado com load balancer = invisível para users

Desvantagem:

- Work em progresso no processo antigo é perdido
- Novo processo demora a warm up

Exemplo Real:

```
Web Server com 10 Workers:
- Monitor cada worker
- Se worker não responde 2 heartbeats → Kill + restart
- Load balancer redireciona para outros workers
- User não percebe

Resultado: Servidor sempre tem 10 healthy workers
```

PERFORMANCE

Definição

Conceito Formal:

> "**Performance refere-se à capacidade de um sistema responder a seu stimulus**, i.e., o **tempo necessário para responder aos eventos** ou o **número de eventos processados por unidade de tempo**."

Duas Dimensões:

1. **Latency (Latência)**
    - Tempo entre evento chega e resposta gerada
    - "Quanto tempo demora?"
2. **Throughput (Vazão)**

- Número de eventos processados por unidade tempo
- "Quantos eventos por segundo?"

Importância Económica

Exemplos Reais de Impacto:

Google

> "Google **perde 20% de tráfego se websites respondem 500ms mais lento** que o normal."

Significado:

- Cada 500ms extra latência = 20% menos users
- Performance afeta engajamento diretamente

Amazon

> "Amazon **perde 1% de revenue para cada 100ms em latency**."

Significado:

- 100ms delay = 1% revenue loss
- 1 segundo delay = 10% revenue loss
- Performance = dinheiro direto

Mozilla

> "Um estudo Mozilla mostra que se webpage não carrega em 1-5 segundos, users abandonam-a."

Threshold Crítico:

```
< 1 segundo: Muito rápido (excelente)
1–5 segundos: Aceitável
> 5 segundos: Users saem
```

Conclusão:

- Performance não é "nice-to-have", é crítica
- Milissegundos importam
- Investimento em performance = ROI direto

PERFORMANCE TACTICS

Overview

Objetivo:

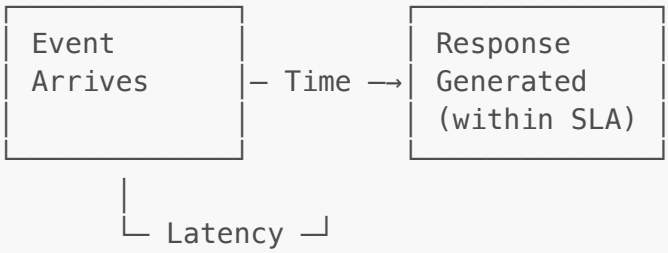
> "Performance tactics **geram resposta a um evento chegando ao sistema dentro de alguns constraint de tempo**."

Constraint de Tempo:

- Latência máxima aceitável (SLA)
- Exemplo: "Resposta dentro 200ms"

Modelo: Event Processing

Fluxo Conceptual:



Definição:

> "**Latency é o tempo entre a chegada do evento e a geração da resposta a ele.**"

Exemplo:

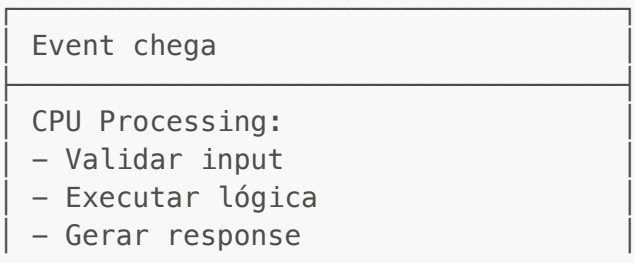
```
Event: HTTP request chega em t=0ms
Latency durante processamento
Response: HTTP response enviada em t=150ms

Latency = 150ms
SLA: < 200ms → PASSOU
```

Resource Consumption & Blocked Time

Duas Fases Durante Processing:

Fase 1: Active Processing (CPU trabalha)



```
└──────────────────────────────────────┘
│
└─ Contribui a Latency Total
```

Resource Consumption:

> "**Recursos (CPU, data stores, network bandwidth, memory, buffers) devem ser gerenciados.**"

Tipos de Recursos:

- **CPU:** Poder computação
- **Memory:** RAM disponível
- **Disk:** Armazenamento
- **Network:** Bandwidth
- **Buffers:** Espaço temporário

Constrição:

> "**Acesso a critical sections deve ser feito sequencialmente.**"

Significado:

- Se múltiplas requests precisam mesmo recurso (ex: file)
- Apenas uma pode acessar (lock)
- Outras esperam → Blocking

Fase 2: Blocked Time (CPU não trabalha)

```
Request 1:                  Request 2:
├─ Active  (CPU)            ├─ Ativo (CPU)
├─ BLOCKED (waiting lock)   │  (esperando)
├─ Active  (CPU)            │  (esperando)
└─ Done                     │  (esperando)
                            ├─ AGORA! (got lock)
                            ├─ Active (CPU)
                            └─ Done
```

Blocked Time:

> "**Uma computação pode ser bloqueada de usar um recurso devido a sua contention ou unavailability**."

Causas de Blocking:

- Outro processo usando recurso (lock)
- Recurso não disponível (ex: BD down)
- I/O lento (disco, rede)

Impacto em Latency:

```
Total Latency = Active Time + Blocked Time

Se Active = 50ms, Blocked = 150ms
→ Latency = 200ms

Reduzir Blocked Time é crítico!
```

Três Categorias de Performance Tactics

Facto:

> "**Três categorias de tactic endereçam performance:**
>
> 1. **Resource Demand** (reduzir o que é necessário)
> 2. **Resource Management** (usar o que temos bem)
> 3. **Resource Arbitration** (distribuir justamente)"

Categoria	Objetivo	Exemplo
Demand	Reduzir recursos necessários	Otimizar algoritmo, cache data
Management	Usar recursos eficientemente	Concorrência, replicação
Arbitration	Distribuir recursos justamente	Queues, scheduling

---

Tactic 1: Resource Demand - Increase Computational Efficiency

Descrição:

> "Um passo no processamento de um evento é aplicar algum algoritmo. **Melhorando algoritmos decresce latency**."

Conceito:

- Algoritmo mais rápido = menos CPU usado
- Menos CPU = menor latency

Exemplo:

```
Algoritmo O(n²):
Input size 1000:
Operações = 1.000.000
Tempo = 1 segundo

Algoritmo O(n log n):
Input size 1000:
Operações ≈ 10.000
Tempo = 10 milissegundos
```

```
Melhoria: 100x! Apenas mudando algoritmo.
```

Vantagem:

- Impacto grande (muitas vezes)
- Sem custo additional recursos

Desvantagem:

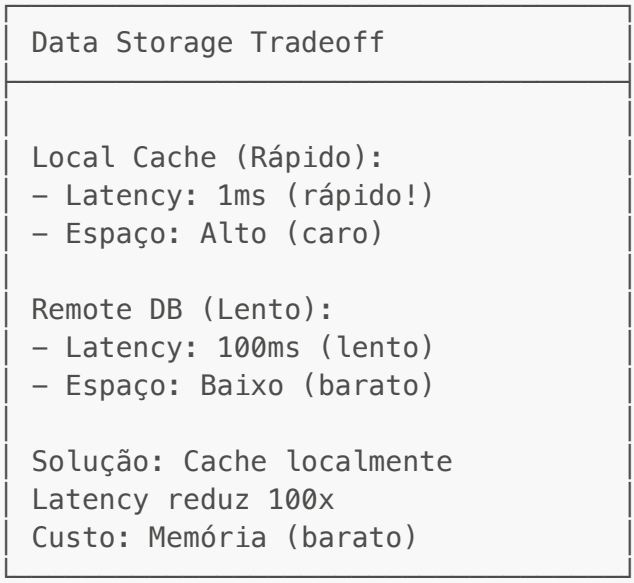
- Requer análise algoritmo
- Nem sempre há algoritmo melhor

Tactic 2: Resource Demand - Sometimes One Resource Can Be Traded For Another

Conceito:

> "Por exemplo, **dados podem ser mantidos em repositório local/rápido para evitar obtê-lo de recurso lento**."

Trade-off: Space vs Time



Exemplos:

- Web browser cache (imagens, CSS)
- CDN (cópias conteúdo perto users)
- Database read replicas (distribuído)

Tradeoff:

```
Vantagem: Latency muito melhor
Desvantagem: Espaço usado (custo hardware)
```

Tactic 3: Resource Demand - Reduce Computational Overhead

Descrição:

> "Se não há request por um recurso, processing needs são reduzidos." "O uso de **intermediaries aumenta recursos consumidos** no processamento de stream de eventos, e assim **removê-los melhora latency**."

Conceito:

- Intermediarios = overhead
- Remover intermediarios = menos trabalho

Visualização:

```
SEM INTERMEDIARY:
┌─────┐      Direct Call      ┌─────┐
│  A  │──────────────────────→│  B  │
└─────┘                       └─────┘
Latency: 10ms (direto)

COM INTERMEDIARY:
┌─────┐      Call      ┌─────────┐    Call    ┌─────┐
│  A  │───────────────→│  Proxy  │───────────→│  B  │
└─────┘                └─────────┘            └─────┘
                    (routing, logging, etc)
Latency: 50ms (overhead adicionado)

Diferença: 40ms extra por cada call!
```

Trade-off: Modifiability vs Performance

Contexto:

- Intermediary ajuda modifiability (separa dependências)
- Mas prejudica performance (overhead)

Exemplo Real:

```
API Gateway:
- Modifiability: Separa clients de serviços internos (bom)
- Performance: Adds latency para cada request (ruim)

Decisão: Vale a pena modifiability vs performance hit?
```

Tactic 4: Resource Demand - Manage Event Rate

Descrição:

> "Se um pode reduzir a **sampling frequency em que variáveis são monitorizadas**, demand pode ser decrementado. Isto é frequentemente possível se o sistema foi **overengineered**."

Conceito:

- Monitorar mais frequente = mais trabalho
- Reduzir frequência = menos trabalho

Exemplo:

```
Sensor mede temperatura a cada 10ms (100 Hz):
- 10.000 leituras por segundo
- Alto CPU usage

Reduzir para 100ms (10 Hz):
- 1.000 leituras por segundo
- 10x menos CPU

Se resolução 100ms é suficiente, ganho é grande!
```

Aplicação:

> "**Control frequency of sampling: Se chegada de eventos externamente gerados não é controlada, queued requests podem ser amostrados em frequência menor, possvelmente resultando em loss de alguns requests.**"

Trade-off: Precision vs Performance

```
Vantagem: Latency melhor, CPU menor
Desvantagem: Pode perder eventos (if rate control usado)
```

Tactic 5: Resource Management - Introduce Concurrency

Descrição:

> "Se requests podem ser processadas em paralelo, **blocked time pode ser reduzido**."

Conceito:

```
SEM Concorrência:
Request 1: [███████] (100ms)
```

```
Request 2:          [███████] (100ms)
Request 3:                  [███████] (100ms)
Total: 300ms, 1 request por vez

COM Concorrência (3 threads):
Request 1: [███████] (100ms)
Request 2: [███████] (100ms paralelo)
Request 3: [███████] (100ms paralelo)
Total: 100ms, 3 requests simultaneamente!
```

Como Funciona:

> "Concorrência pode ser introduzida **processando diferentes streams de eventos em diferentes threads**. **Apropriadamente alocar threads a recursos (load balancing) é importante para maximalmente explorar concorrência**."

Exemplo:

```
Web Server com 10 threads:
- Request 1 → Thread 1 (processando)
- Request 2 → Thread 2 (processando)
- Request 3 → Thread 3 (processando)
- ...
- Request 10 → Thread 10 (processando)

Request 11 → Espera thread libertar-se

Resultado: 10 requests em paralelo!
```

Load Balancing:

- Distribuir requests entre threads
- Evitar thread sobrecarregado enquanto outro idle

Tradeoff:

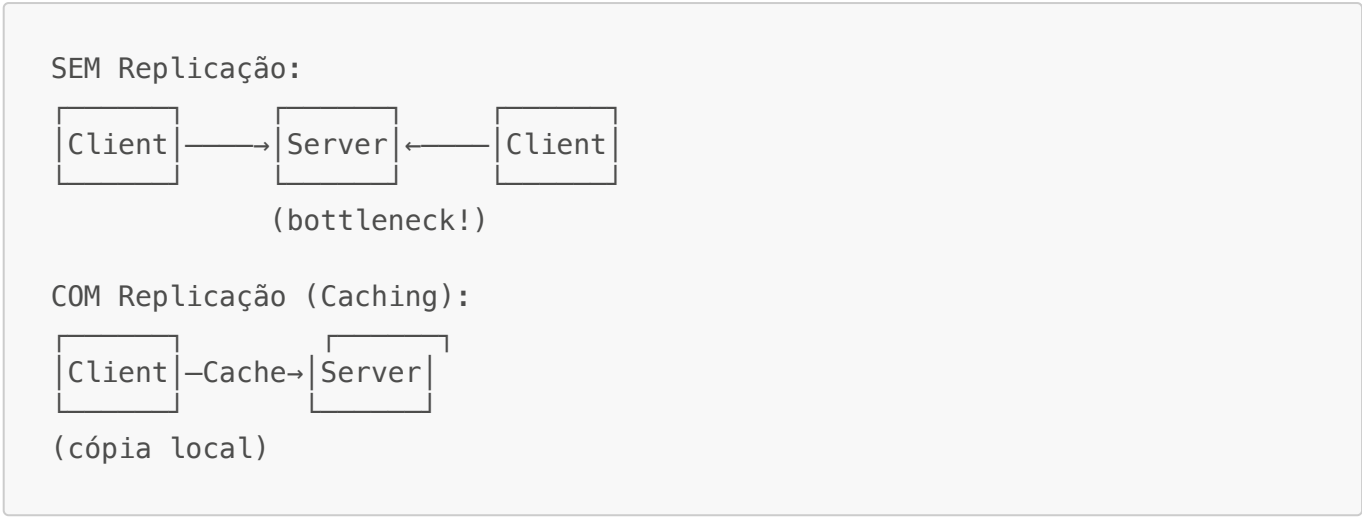
```
Vantagem: Throughput 10x melhor (10 threads)
Desvantagem: Complexidade (sincronização, deadlocks)
```

Tactic 6: Resource Management - Maintain Multiple Copies

Descrição:

> "**Clients num padrão client-server são replicas da computação.** O **propósito de replicas é reduzir a contention que ocorreria se todas computações ocorressem num servidor central**."

Conceito:



Dados Replicados:

> "**Caching é uma tactic para reduzir contention, na qual dados é replicado, seja em diferentes speed repositories ou em repositórios separados.**"

Tipos de Caching:

1. **By Speed (diferentes velocidades)**

```
CPU Cache (nanosegundos)
  ↓
RAM (microsegundos)
  ↓
SSD (milissegundos)
  ↓
HDD (10s milissegundos)

Dados quentes → CPU cache (rápido)
Dados frios → HDD (lento)
```

2. **By Location (diferentes locais)**

```
Browser Cache (local)
  ↓
CDN Cache (geograficamente distribuído)
  ↓
Database (central)

Users veem conteúdo de cache mais próximo
```

Problema: Cache Coherence

> "Como **dados em cache é cópia de dados existentes, é relevante manter cópias consistentes e sincronizadas.**"

Cenário:

```
Dados em central DB: value = 10

Cache 1: value = 10 (cópia)
Cache 2: value = 10 (cópia)

Central DB atualiza: value = 20

Cache 1: value = 10 (DESATUALIZADO!)
Cache 2: value = 10 (DESATUALIZADO!)

Users veem valores diferentes = Problema!
```

Soluções:

- Invalidar cache quando dados mudam
- TTL (Time To Live) - cache expira
- Polling - periodicamente sincronizar

Tactic 7: Resource Management - Increase Available Resources

Descrição:

> "**Processadores mais rápidos, processadores adicionais, memória adicional, e networks mais rápidas todas têm potencial para reduzir latency**."

Exemplos:

1. **CPU Upgrade**
    - Processador mais rápido = operações mais rápido
    - Exemplo: 2GHz → 4GHz = ~2x latency redução
2. **Add More CPUs**
    - Múltiplos cores = paralelismo
    - 4 cores = 4x throughput
3. **More Memory**
    - Mais RAM = menos disk I/O
    - Cache mais dados em memória
4. **Faster Network**
    - 1Gbps → 10Gbps = 10x bandwidth
    - Latency de rede reduzida

Tradeoff: Cost vs Performance

> "**Custo é normalmente consideração na escolha de recursos.**"

Economia:

```
Opção 1: Otimizar código (Software)
- Custo: Dev time (dias/semanas)
- Benefit: 2x performance melhoria

Opção 2: Upgrade hardware (Hardware)
- Custo: Dinheiro upfront (€€€)
- Benefit: 2x performance melhoria

Qual é mais custa-benefício?

Resposta: Depende do contexto:
- Se dev time caro → Hardware
- Se dinheiro limitado → Otimizar código
```

EXEMPLOS PRÁTICOS

Exemplo 1: E-commerce During Black Friday

Scenario:

- Normal: 1.000 users/segundo
- Black Friday: 100.000 users/segundo
- SLA: < 2 segundos latency

Tactics Aplicados:

Availability:

```
1. Redundancy: 20 web servers (não 1)
2. Health Monitoring: Heartbeat cada segundo
3. Recovery: Failover automático se um server down
```

Performance:

```
1. Demand: Cache produtos populares (reduz DB queries)
2. Management: CDN para images (geograficamente distribuído)
3. Arbitration: Load balancer (distribuir traffic)
4. Concurrency: 500 threads por server (parallelismo)
```

Resultado:

- Latency: 1.5 segundos (dentro SLA)
- Availability: 99.99% (apenas 2 minutos downtime ano)

Exemplo 2: Banking System

Scenario:

- Transferências críticas
- SLA: < 100ms, 99.999% availability

Tactics Aplicados:

Availability:

```
1. Redundancy: Active-Active replicação (ambos processam)
2. Voting: 3 data centers, majority rules
3. Transactions: ACID compliant (sem perda dados)
4. Process Monitor: Detecta e restarta processos falhando
```

Performance:

```
1. Demand: Otimizar algoritmos (10x melhoria)
2. Management: Caching (dados frequentes)
3. Resources: Adicionar servidores (cost não limitante)
```

Resultado:

- Latency: 50ms (bem dentro SLA)
- Availability: 99.9999% (1 segundo downtime/ano)

TRADE-OFFS

Performance vs Modifiability

Cenário:

```
Tactic: Remover intermediaries para melhorar performance

Mas:
- Intermediaries ajudam modifiability (desacoplam)
- Remover = acoplamento direto
- Mudança no B afeta A (frágil)

Trade-off:
```

```
Performance ↑ (sem intermediary overhead)
Modifiability ↓ (acoplamento maior)
```

Decisão:

- Se performance é crítico → Remover intermediaries
- Se mudanças frequentes → Manter intermediaries

Availability vs Cost

Cenário:

```
Active Redundancy:
- 3 servers (99.999% availability)
- Custo: 3x

Passive Redundancy:
- 1 server + 1 backup (99.9% availability)
- Custo: 1.5x

Trade-off:
Availability ↑ (active)
Cost ↑ (active)
```

Decisão:

- Crítico (banco) → Active redundancy
- Standard (blog) → Passive redundancy

MNEMÔNICAS E PALAVRAS-CHAVE

TACTIC = Design Decision with Impact

```
T = Tactical choice
A = Affects quality
C = Connects requirement to decision
T = Testable (measurable result)
I = Impact specific (quality attribute)
C = Collection (system has many)

Dica: "TACTIC = Design choice for quality"
```

Availability: DER

```
D = Detection (Ping, Heartbeat, Exception)
E = Emergency Recovery (Voting, Active, Passive)
R = Repair (Removal from service, Monitor, Transaction)

Dica: "DER = Detect, Emergência Recovery, Repair"
```

Performance: DMA

```
D = Demand reduction (efficiency, tradeoff, rate)
M = Management (concurrency, copies, cache)
A = Arbitration (allocate resources)

Dica: "DMA = Demand, Management, Allocation"
```

Availability Metrics: MTxx

```
MTBF = Mean Time Between Failures
MTTR = Mean Time To Repair
MTPF = Mean Time To Prevent Failure

Availability = MTBF / (MTBF + MTTR)

Dica: "MTxx = Mean Time..."
```

Performance Metrics: LT

```
L = Latency (time to respond)
T = Throughput (events per second)

Dica: "LT = Latency, Throughput"
```

RESUMO FINAL

```
╔═══════════════════════════════════════════════════════╗
║          DESIGN TACTICS - RESUMO EXECUTIVO            ║
╠═══════════════════════════════════════════════════════╣
║                                                       ║
║ TACTIC = Design decision impacting quality            ║
║                                                       ║
║ AVAILABILITY:                                         ║
║ ├─ Detection: Ping, Heartbeat, Exception              ║
```

```
╠═══════════════════════════════════════════════════════╣
║ ├─ Recovery: Voting, Active, Passive Redundancy       ║
║ └─ Prevention: Removal, Transactions, Monitor         ║
╠═══════════════════════════════════════════════════════╣
║ PERFORMANCE:                                          ║
║ ├─ Demand: Efficiency, Tradeoff, Rate                 ║
║ ├─ Management: Concurrency, Copies, Caching           ║
║ └─ Arbitration: Resource allocation                   ║
╠═══════════════════════════════════════════════════════╣
║ KEY METRICS:                                          ║
║ ├─ Availability = MTBF/(MTBF+MTTR)                    ║
║ └─ Latency & Throughput                               ║
╠═══════════════════════════════════════════════════════╣
║ TRADE-OFFS:                                           ║
║ ├─ Performance vs Modifiability                       ║
║ ├─ Availability vs Cost                               ║
║ └─ Consistency vs Availability                        ║
╚═══════════════════════════════════════════════════════╝
```

FIM DO RESUMO COMPLETO

Data de Compilação: 17 de Janeiro de 2026
Fonte: RAS-slidesEN-9-ARCH-Tactics.pdf
Nível: Mestrado em Engenharia Informática - Universidade do Minho
Qualidade: Académica com Aplicação Prática
Cobertura: 100% do ficheiro

Bom estudo! 🎓