

The background of the slide features a complex, abstract network structure composed of numerous small, glowing cyan dots (nodes) connected by thin, translucent cyan lines (edges). This pattern creates a sense of depth and connectivity, resembling a digital or scientific visualization. The overall color palette is dark, with the cyan elements providing a sharp contrast.

[beyond]

A black and white photograph showing two men in an office environment. One man, wearing a plaid shirt, is seated at a desk looking intently at a laptop screen. The other man, wearing a dark t-shirt, is standing beside him, also looking at the screen. A lamp is positioned on the desk next to the laptop. The background shows office cubicles.

PERFORMANCE DE APLICAÇÕES

CONCEITOS, DIAGNÓSTICO PROFISSIONAL E BOAS PRÁTICAS

QUEM SOU EU?



YAGO HENRIQUE FERREIRA

32 ANOS

CONSULTOR EM TECNOLOGIA PELA BEYOND SOLUÇÕES

ATUANDO NA ÁREA HÁ PRATICAMENTE 14 ANOS

GRADUADO EM SISTEMAS DE INFORMAÇÃO E PÓS GRADUADO EM ARQUITETURA DE SOFTWARE / I.A COM ÊNFASE EM MACHINE LEARNING

ONDE ME ENCONTRAR:



<https://www.linkedin.com/in/yagohferreira>



<https://www.instagram.com/yagohf>

AGENDA

PARTE 1 - TEORIA

#1 - O que é performance e como ela impacta nossa vida?

#2 – Conceitos fundamentais

#3 – Gargalos comuns

PARTE 2 - PRÁTICA

#4 – Investigando problemas

#5 – Hands on

#6 – Boas práticas

O QUE É PERFORMANCE?

O QUE É PERFORMANCE ?

"A performance de um software pode ser definida como a sua capacidade de executar tarefas de forma eficiente, dentro de um tempo de resposta aceitável para o usuário e com o uso otimizado dos recursos disponíveis."

- Ao descrevermos a performance de um sistema, talvez a pergunta não seja “O que é performance?”, mas sim “O que acontece com meu sistema quando aumento sua carga em X%?”
- Performance não é um número estático! É a história de como seu sistema se comporta sob estresse.

O problema do restaurante: Não basta o primeiro prato da noite sair rápido. A verdadeira performance é medida na hora do rush: a cozinha consegue entregar dezenas de pratos (**throughput**), mantendo a qualidade e o tempo de espera aceitável para cada cliente (**latência**)?

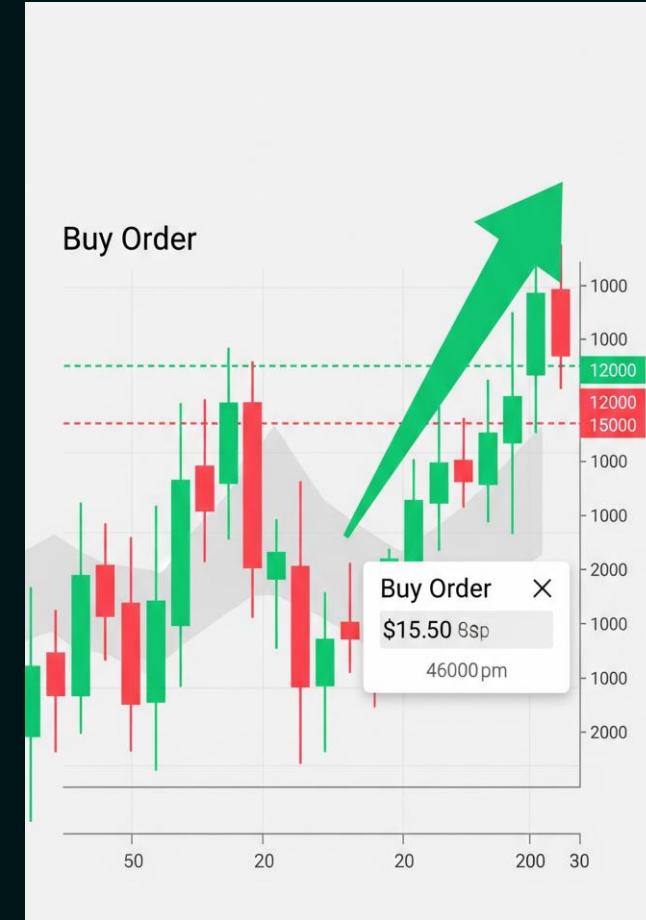


A PERFORMANCE É RELATIVA AO CONTEXTO!

Relatório gerencial – 500ms – Extremamente rápido ✓



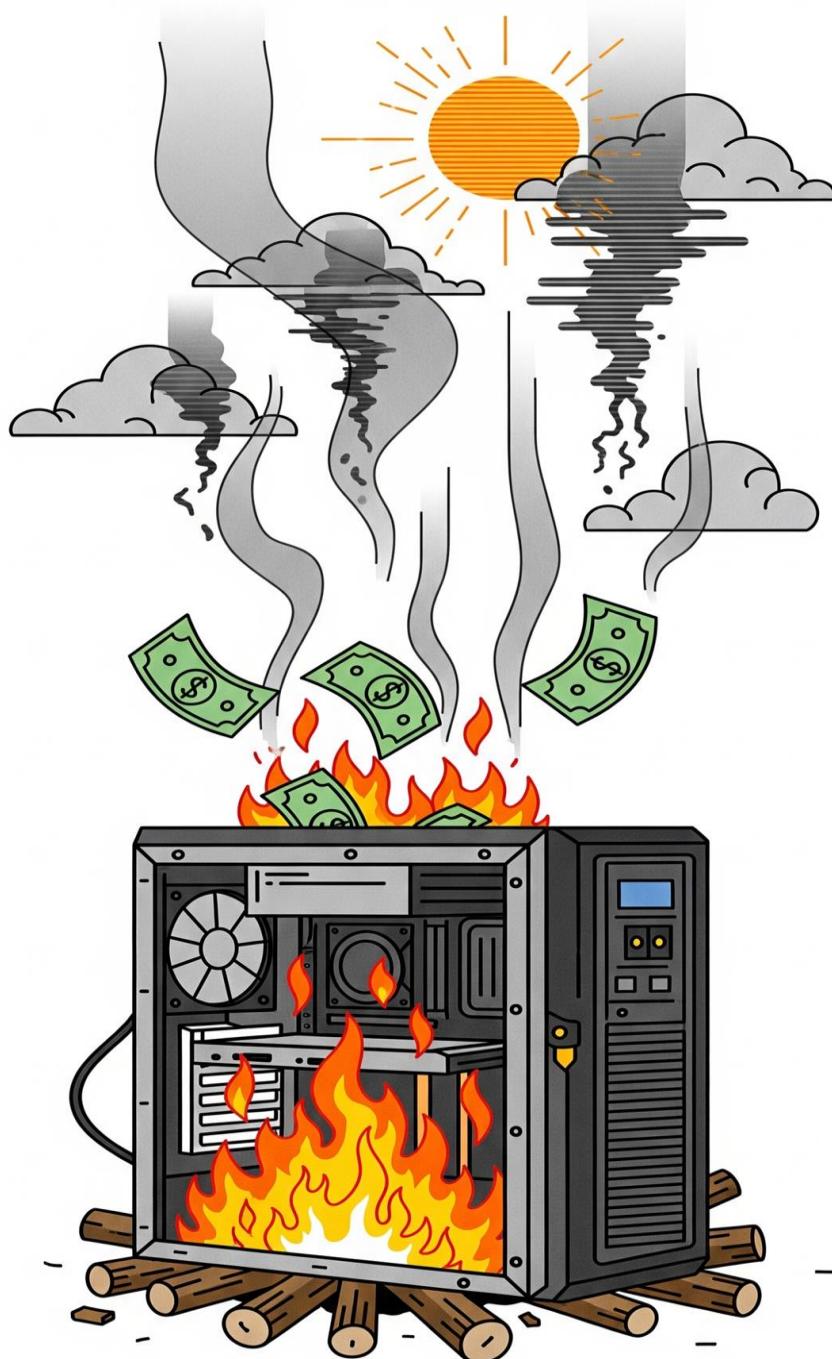
Envio de ordem (HFT) – 500 ms – Extremamente lento X



A meta de performance (SLO) é uma decisão de produto e de negócio, não apenas um capricho da engenharia.

POR QUÊ PERFORMANCE IMPORTA?

- **Custos diretos de infraestrutura:** Código ineficiente custa caro. Em um mundo de nuvem pay-as-you-go, cada ciclo de CPU e GB de memória vai para a fatura.
- **Uso de energia e sustentabilidade:** Performance é sustentabilidade. Softwares mais eficientes demandam menos dos data centers, reduzindo a pegada de carbono.
- **A experiência do usuário:** O impacto mais dramático é na experiência do usuário e, consequentemente, na receita e retenção.



O CUSTO DE UM SEGUNDO

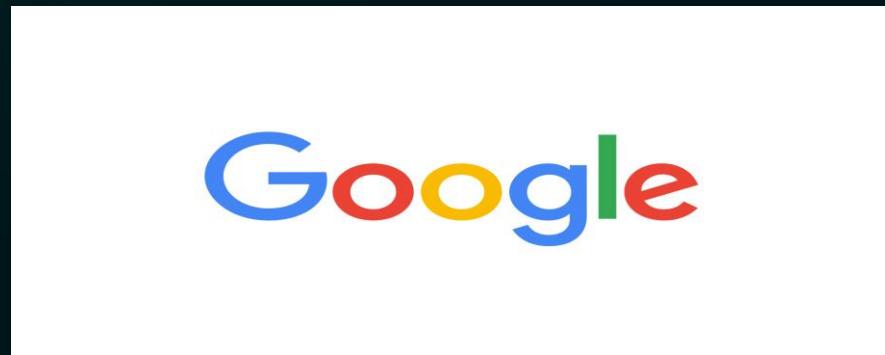
Amazon: Descobriu que cada 100ms de latência a mais custava 1% em vendas. [\[🔗\]](#)



Vodafone: Melhorou a velocidade do site em 31% e aumentou seu percentual de vendas em 8%. [\[🔗\]](#)



Google: Verificou que um atraso de 500ms na busca causava uma queda de 20% no tráfego. [\[🔗\]](#)



BBC: Descobriu que perdia 10% do total de usuários para cada segundo adicional que suas páginas demoravam para carregar. [\[🔗\]](#)



CONCEITOS FUNDAMENTAIS

[beyond]

NOÇÕES DE GRANDEZA

Operação	Tempo	Relação em escala humana	Multiplicador
Acesso ao cache L1	0.5 ns	1 segundos	1x
Acesso ao cache L2	7 ns	14 segundos	14x
Acesso ao cache L3	20 ns	40 segundos	20x
Lock/Unlock com Mutex	25 ns	50 segundos	50x
Acesso à memória RAM	100 ns	3.3 minutos	200x
Enviar 1 KB via rede (1 Gbps)	10 µs	5.6 horas	20.000x
Leitura aleatória de 4 KB do SSD	150 µs	3.5 dias	300.000x
Leitura sequencial de 1 MB da memória RAM	250 µs	5.8 dias	500.000x
Round-trip no mesmo datacenter	500 µs	1.7 semanas	1.000.000x
Leitura sequencial de 1MB do SSD	1 ms	3.3 semanas	2.000.000x
Leitura sequencial de 1MB do disco	20 ms	1.3 anos	40.000.000x
Enviar 1 KB via rede (Brasil -> EUA)	150 ms	9.5 anos	300.000.000x

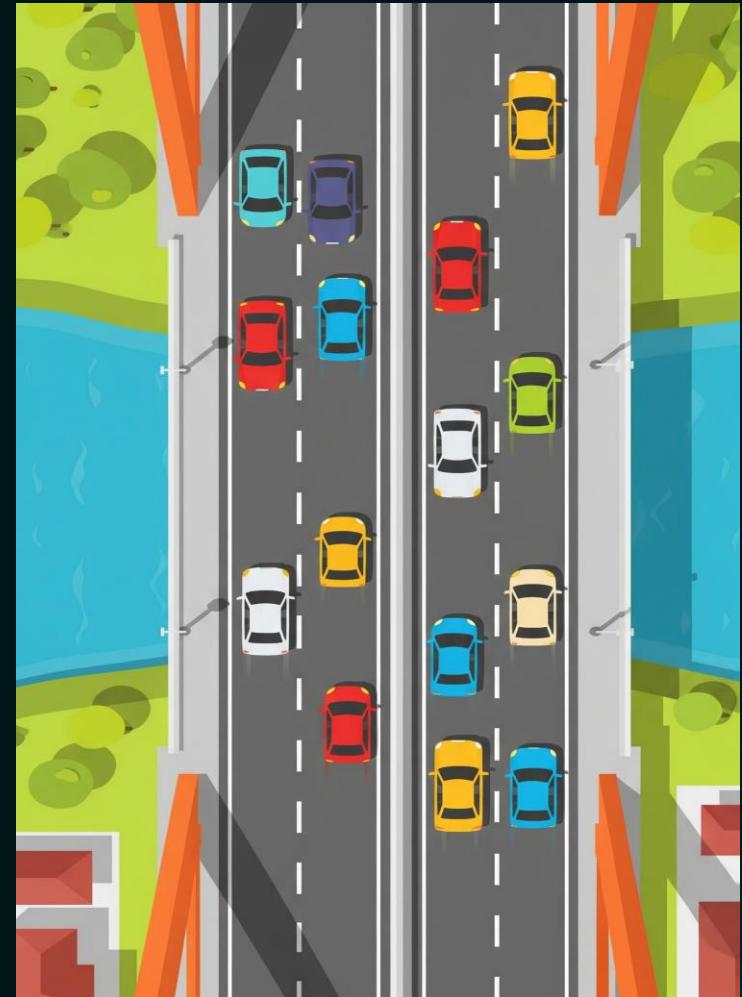
Adaptado de “[Latency Numbers Everyone Should Know](#)” (Google SRE)

LATÊNCIA X THROUGHPUT

Latência: O tempo que uma única operação leva. É uma medida de **rapidez**, geralmente mensurada em milissegundos ou segundos.

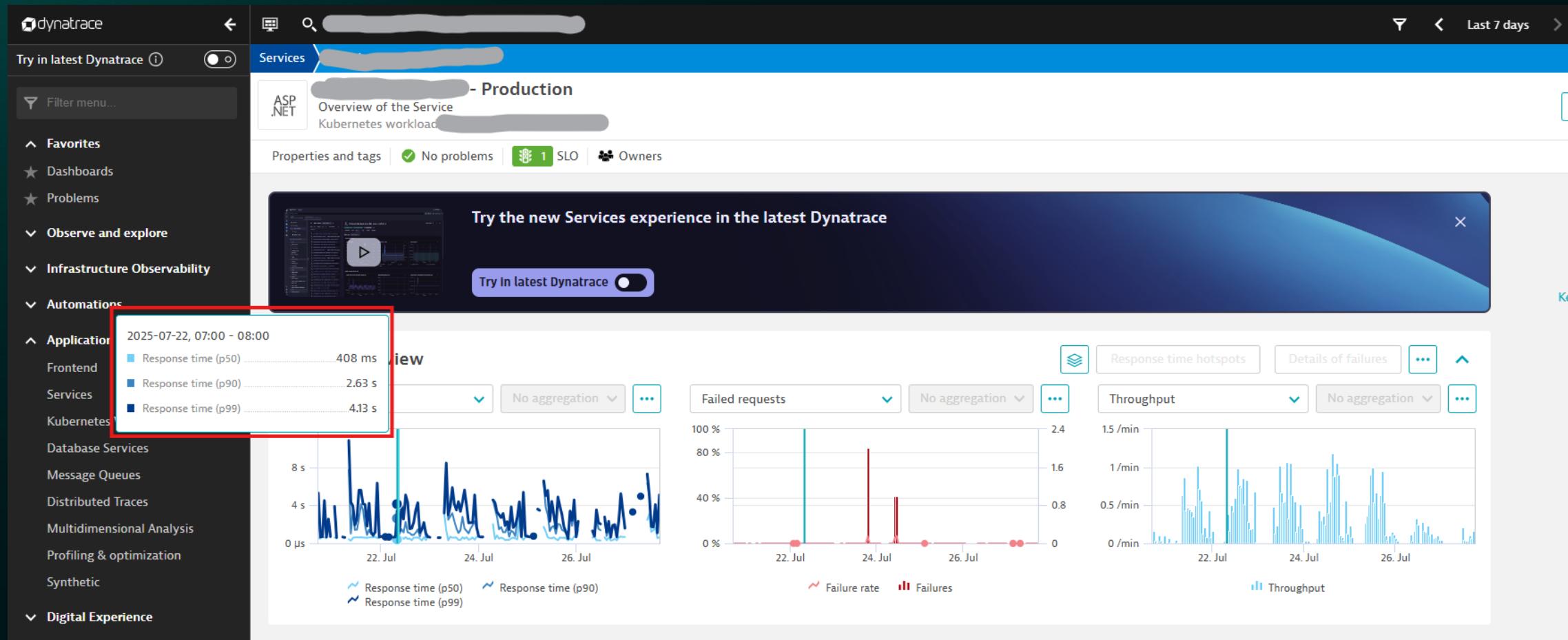
Throughput: O número de operações por unidade de tempo. É uma medida de **capacidade**, geralmente mensurada em RPS (requisições por segundo).

💡 **Importante:** Otimizar um, não garante a otimização do outro.



LATÊNCIA E PERCENTIS

Latência: A latência pode ser observada nos monitoramentos através de uma visualização de percentis.



PERCENTIS: POR QUÊ A MÉDIA MENTE?

A média esconde os outliers. Você não sente a média, **você sente a sua própria requisição**.



I/O BOUND X CPU BOUND – ONDE SUA APLICAÇÃO GASTA O TEMPO?

Toda tarefa em um computador está limitada por um recurso. Ela está "presa", esperando ou pela CPU ou por uma operação de Entrada/Saída (I/O).

CPU BOUND – Limitada por processamento

- Criptografia;
- Processamento de imagens, vídeos ou áudios;
- Cálculos complexos (como em Machine Learning);
- Compressão de arquivos;
- Serialização/deserialização de objetos;

Como otimizamos?

Algoritmos mais eficientes (Big O), código de mais baixo nível ou, em último caso, mais processadores (ou processadores mais rápidos).

I/O BOUND – Limitada pela espera de algum dado ou recurso

- Query em um banco de dados;
- Consulta a outro serviço via HTTP;
- Leitura de um arquivo do disco;
- Publicação / leitura de mensagem numa fila ou tópico;

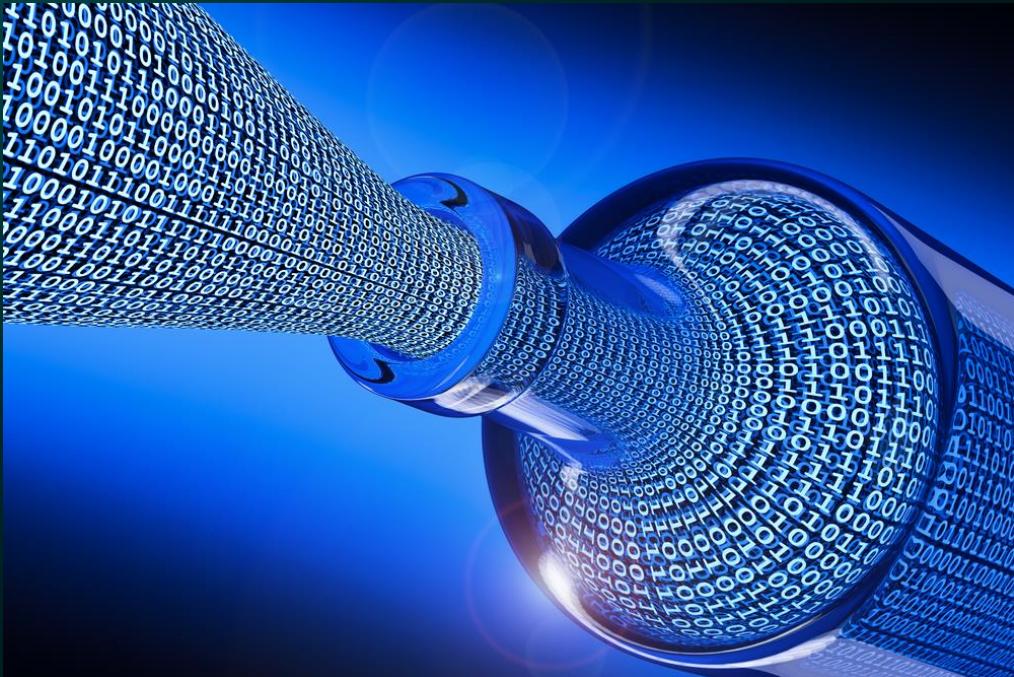
Como otimizamos?

Usar o tempo de espera de forma inteligente!
Concorrência e Paralelismo (I/O assíncrono, threads) para que a CPU faça outro trabalho enquanto espera.
Adicionar mais CPUs não resolve o gargalo de I/O.

GARGALOS COMUNS

[beyond]

O QUE É UM GARGALO?



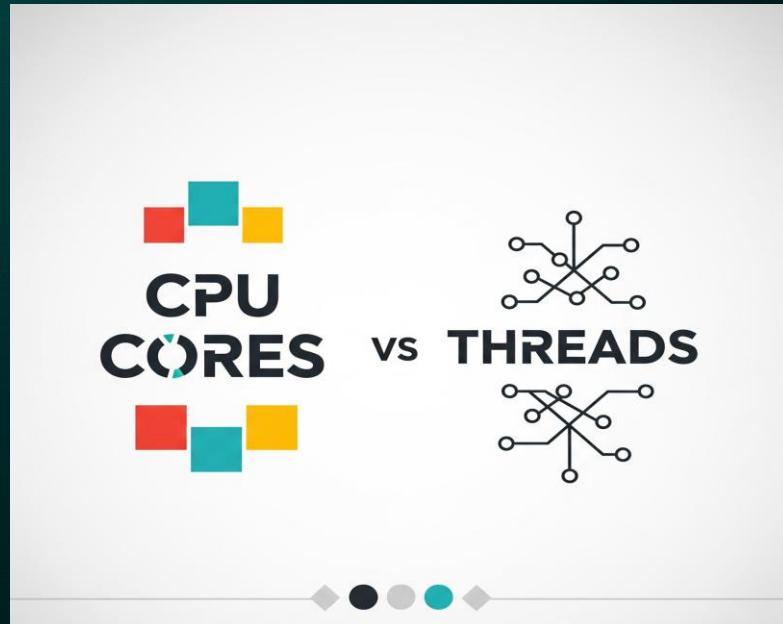
"Um sistema é tão rápido quanto o seu componente mais lento."



Otimizar qualquer outra parte que não seja o gargalo principal é um desperdício de tempo e esforço.

Se a sua query no banco de dados leva 2 segundos, não adianta otimizar um cálculo na CPU que leva 10 milissegundos.

CONCORRÊNCIA, THREADS E TROCAS DE CONTEXTO



Cores físicos e lógicos: A capacidade real de processamento do hardware. Se um processador tem 8 cores, ele pode executar 8 coisas em paralelo.

Threads (alto nível): Abstrações criadas pelas linguagens de programação, que precisam ser “encaixadas” nas threads físicas/virtuais.

A magia do S.O: A troca de contexto (Context Switch).

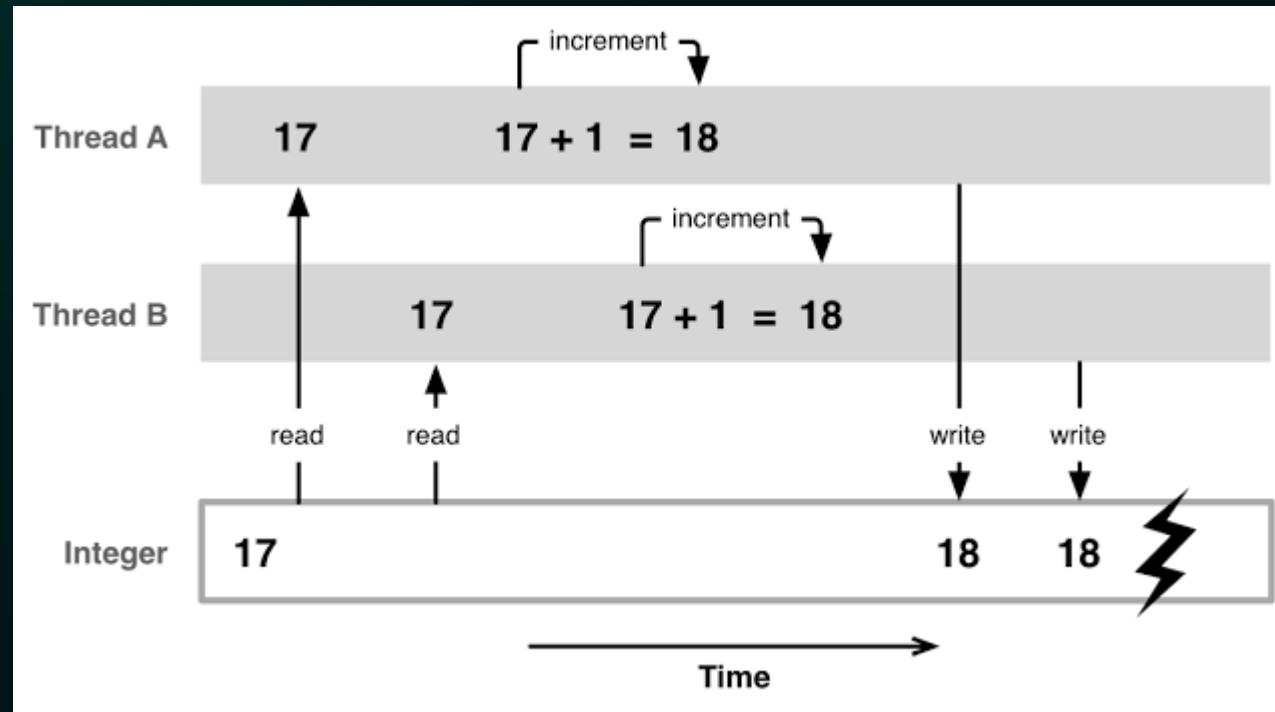
Para dar a ilusão de que centenas de threads rodam ao mesmo tempo em X cores, o S.O rapidamente pausa uma thread, salva seu estado (registradores, ponteiros), carrega o estado de outra e a retoma.

Mas este processo não é de graça!

O gargalo: Muitas threads ativas competindo por poucas CPUs levam a trocas de contexto excessivas. O sistema gasta mais tempo gerenciando as threads do que executando o trabalho útil da sua aplicação.

O DILEMA DO LOCKING

Para evitar race conditions, utilizamos locks. Em excesso, geramos contenção (filas) e nosso código assíncrono pode acabar sendo executado totalmente em série.



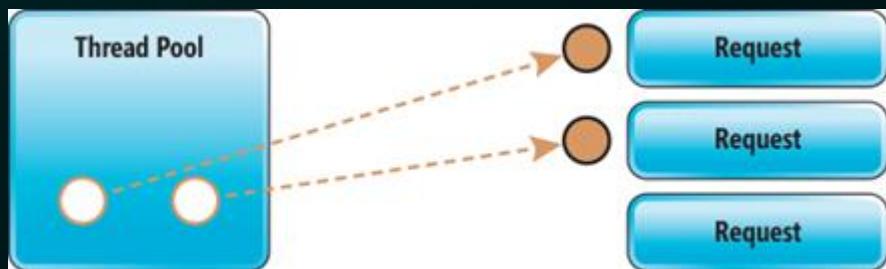
💡 Dica: O código dentro de locks deve ser pequeno e rápido.

MODELOS DE CONCORRÊNCIA - .NET X NODE.JS

.NET

Multithreading “preemptivo”: O S.O decide “pausar” (preempt) uma thread para dar vez a outra.

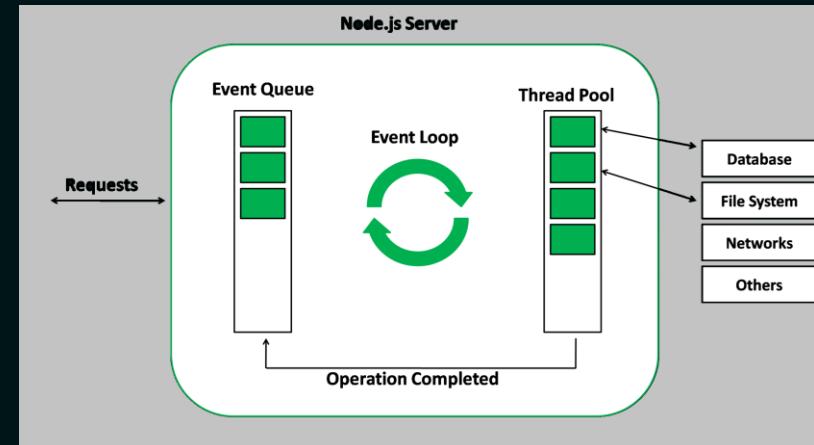
Thread pool: Um grupo de abstrações de threads esperando para atender requisições. Quando entram em I/O, são devolvidas ao pool para que outras solicitações sejam executadas.



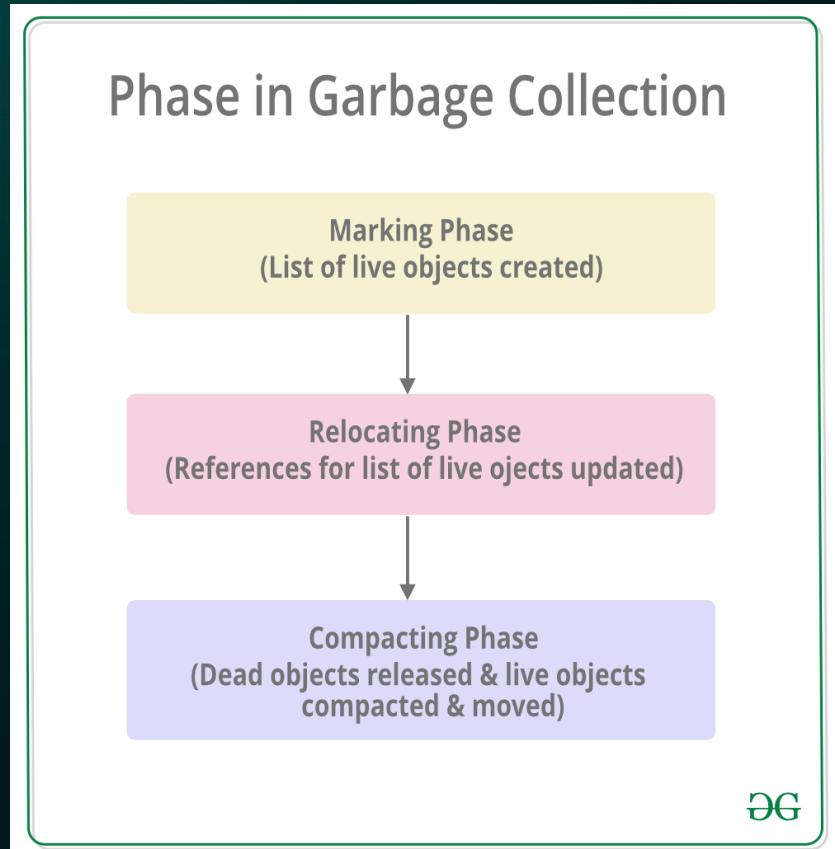
NODE.JS

Event loop “single threaded”: Uma fila de eventos é percorrida e executada, sempre na thread principal.

Thread pool: Quando o evento em questão executa uma operação de I/O, o trabalho é delegado para uma thread do thread pool (escrito em C++). Uma vez que o trabalho tenha terminado, um callback é enviado para o event loop, para ser executado no momento certo.



GARBAGE COLLECTOR (GC) – LIMPANDO A BAGUNÇA (MAS A QUE CUSTO?)



O coletor de lixo, em geral, é um processo em segundo plano que executa a liberação da memória que não está mais em uso, evitando assim os chamados “memory leaks” (vazamentos de memória).

Para isso, o GC percorre o grafo de objetos da nossa aplicação em busca de objetos “mortos” (ou seja, sem referências). Visando impedir que o estado da aplicação mude durante sua execução, o GC **pausa quase que totalmente** as rotinas da aplicação.

GARBAGE COLLECTOR – .NET X NODE.JS

.NET

“Stop the world”: Precisamos parar “tudo” para rolar a limpeza.



NODE.JS

Pausas curtas e direcionadas: Limpeza discreta em áreas específicas, sem interromper o fluxo principal.



MEMORY LEAKS



Pergunta: Em ambientes gerenciados, como .NET e Node.js, não deveríamos estar livres de memory leaks?

A resposta: Não exatamente. O GC nos alivia da tarefa de liberar memória manualmente. Mas ele não tem uma bola de cristal: **o GC só vai liberar aquilo que ele sabe que é lixo.**

MEMORY LEAKS – VILÕES COMUNS

#1 – EVENT HANDLERS QUE NUNCA SÃO REMOVIDOS

```
public class MyClass
{
    public MyClass(wifiManager wifiManager)
    {
        wifiManager.wifiSignalChanged += onWiFichanged;
    }

    private void onWiFichanged(object sender, wifiEventArgs e)
    {
        // do something
    }
}
```

Cenário: Um objeto de vida curta se inscreve em um evento de um objeto de vida longa.

O leak: Quando o objeto de vida curta deveria ser destruído, o objeto de vida longa ainda mantém uma referência a ele através do evento. Ele nunca é coletado.

Solução: Sempre que se inscrever em um evento, implemente a lógica para se "desinscrever" (ex: no Dispose de um objeto).

MEMORY LEAKS – VILÕES COMUNS

#2 – CACHING E MEMBROS ESTÁTICOS

```
public class MyClass
{
    static List<MyClass> _instances = new List<MyClass>();
    public MyClass()
    {
        _instances.Add(this);
    }
}
```

```
public class ProfilePicExtractor
{
    private Dictionary<int, byte[]> Picturecache { get; set; } =
        new Dictionary<int, byte[]>();

    public byte[] GetProfilePicByID(int id)
    {
        // A lock mechanism should be added here, but let's stay on point
        if (!PictureCache.ContainsKey(id))
        {
            var picture = GetPictureFromDatabase(id);
            PictureCache[id] = picture;
        }
        return Picturecache[id];
    }
}
```

Cenário: Você tem um Dictionary ou List estático que usa como um cache simples. Você adiciona itens a ele, mas nunca os remove.

O leak: Cada objeto adicionado a essa coleção viverá para sempre (ou enquanto a aplicação rodar), mesmo que nunca mais seja usado.

Solução: Use caches com políticas de expiração (como MemoryCache no .NET ou node-cache no Node.JS com Express) ou implemente uma lógica para limpar a coleção estática.

MEMORY LEAKS – VILÕES COMUNS

#3 – CLOSURES

```
function outer(elementCount) {
  // Create an array containing numbers from 0 to elementCount value
  let elements = Array.from(Array(elementCount).keys());

  return function inner() {
    // return a random number
    return elements[Math.floor(Math.random() * elements.length)];
  };
}

let getRandom = outer(10000);
console.log(getRandom());
console.log(getRandom());
```

```
public class MyClass
{
  private JobQueue _jobQueue;
  private int _id;

  public MyClass(JobQueue jobQueue)
  {
    _jobQueue = jobQueue;
  }

  public void Foo()
  {
    _jobQueue.EnqueueJob(() =>
    {
      Logger.Log($"Executing job with ID {_id}");
      // do stuff
    });
  }
}
```

Cenário: Uma função anônima (ou lambda) usa uma variável de um escopo maior. Essa lambda é então passada para algum lugar e vive mais que o escopo original.

O leak: A closure mantém uma referência a todo o escopo que capturou, impedindo que os objetos daquele escopo sejam coletados pelo GC.

Solução: Isolar o escopo dos dados consumidos pela closure, criar cópias de valores manipulados pela closure para evitar referências externas e descartar explicitamente objetos não mais usados.

MEMORY LEAKS – VILÕES COMUNS

#4 – THREADS QUE NUNCA TERMINAM (CUIDADO COM OS TIMERS)

```
public class MyClass
{
    public MyClass()
    {
        Timer timer = new Timer(HandleTick);
        timer.Change(TimeSpan.FromSeconds(5), TimeSpan.FromSeconds(5));
    }

    private void HandleTick(object state)
    {
        // do something
    }
}
```

```
function increment() {
    const data = [];
    let counter = 0;

    return function inner() {
        data.push(counter++); // data array is now part of the callback's scope
        console.log(counter);
        console.log(data);
    };
}

setInterval(increment(), 1000);
```

Cenário: Timers que executam processamentos e carregam referências consigo.

O leak: Enquanto uma thread viver, sua stack não será descartada. Isso inclui referências a outros objetos/funções em nossas aplicações.

Solução: Evitar o compartilhamento de estado/instâncias entre objetos acessados pelos timers e o restante da aplicação. Descartá-los de maneira elegante no shutdown da aplicação.

INVESTIGANDO PROBLEMAS

[beyond]

A MENTALIDADE DO DETETIVE



Já conhecemos os suspeitos. Agora, como encontramos o(s) culpado(s)?

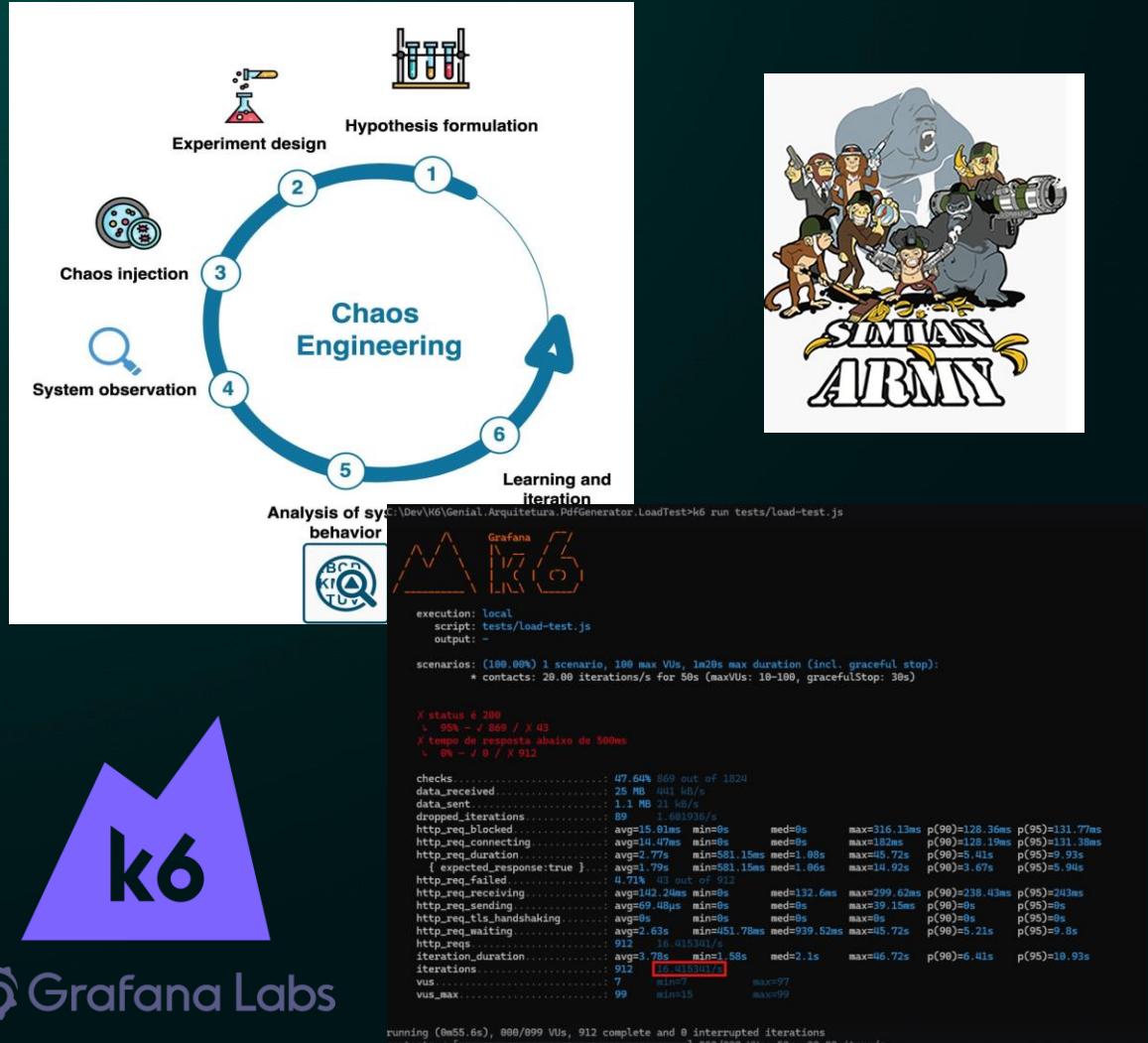
1. Escolha a ferramenta certa para o trabalho certo.
2. Não adivinhe. Meça!

NOSSO KIT DE FERRAMENTAS



1. Load tests e chaos engineering
2. APM – Application Performance Monitoring
3. Profiling

LOAD TESTING E CHAOS ENGINEERING



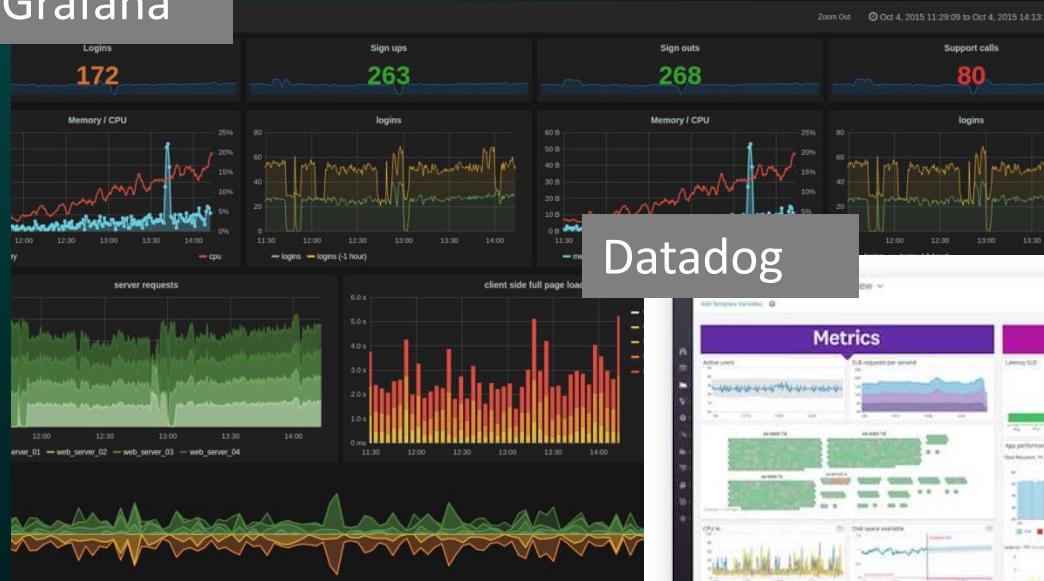
A ideia dessas técnicas é validar o comportamento das aplicações sob a carga esperada e sob condições extremas, como:

- Alta carga;
- Degradação de dependências;
- Instabilidade na infraestrutura;

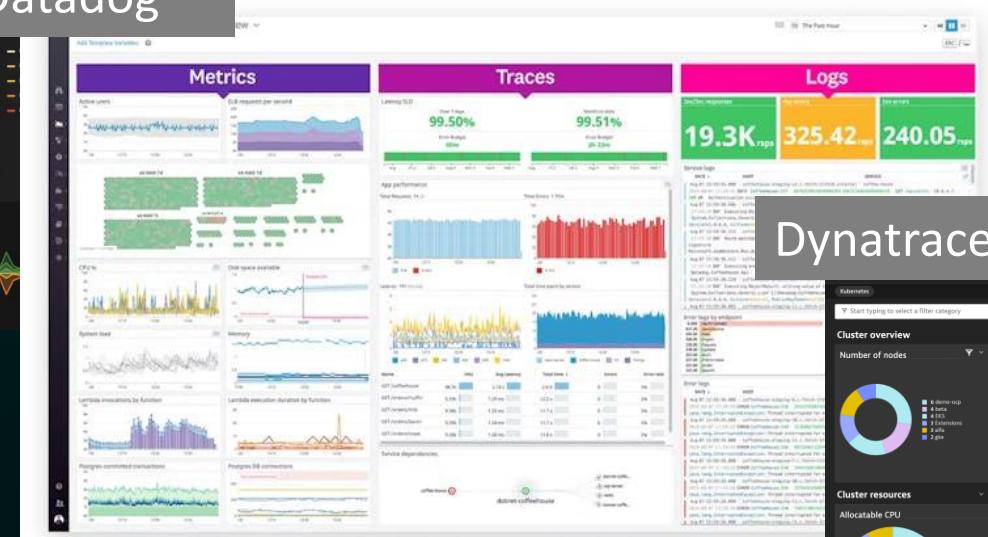


APM – APPLICATION PERFORMANCE MONITORING

Grafana



Datadog



Dynatrace



💡 Menções honrosas: NewRelic, AppDynamics e Splunk.

[beyond]

PROFILING – A LUPA SOBRE SEU CÓDIGO



O que é?

Técnica para analisar uma aplicação enquanto ela executa, medindo o uso de recursos (como uso de CPU e as alocações de memória) de cada trecho do código. As ferramentas para execução de profiling são os chamados **profilers**.

Se o APM nos mostra qual transação está lenta, os profilers mostram qual linha de código dentro daquela transação é a culpada.

PROFILING - CPU

MODOS:

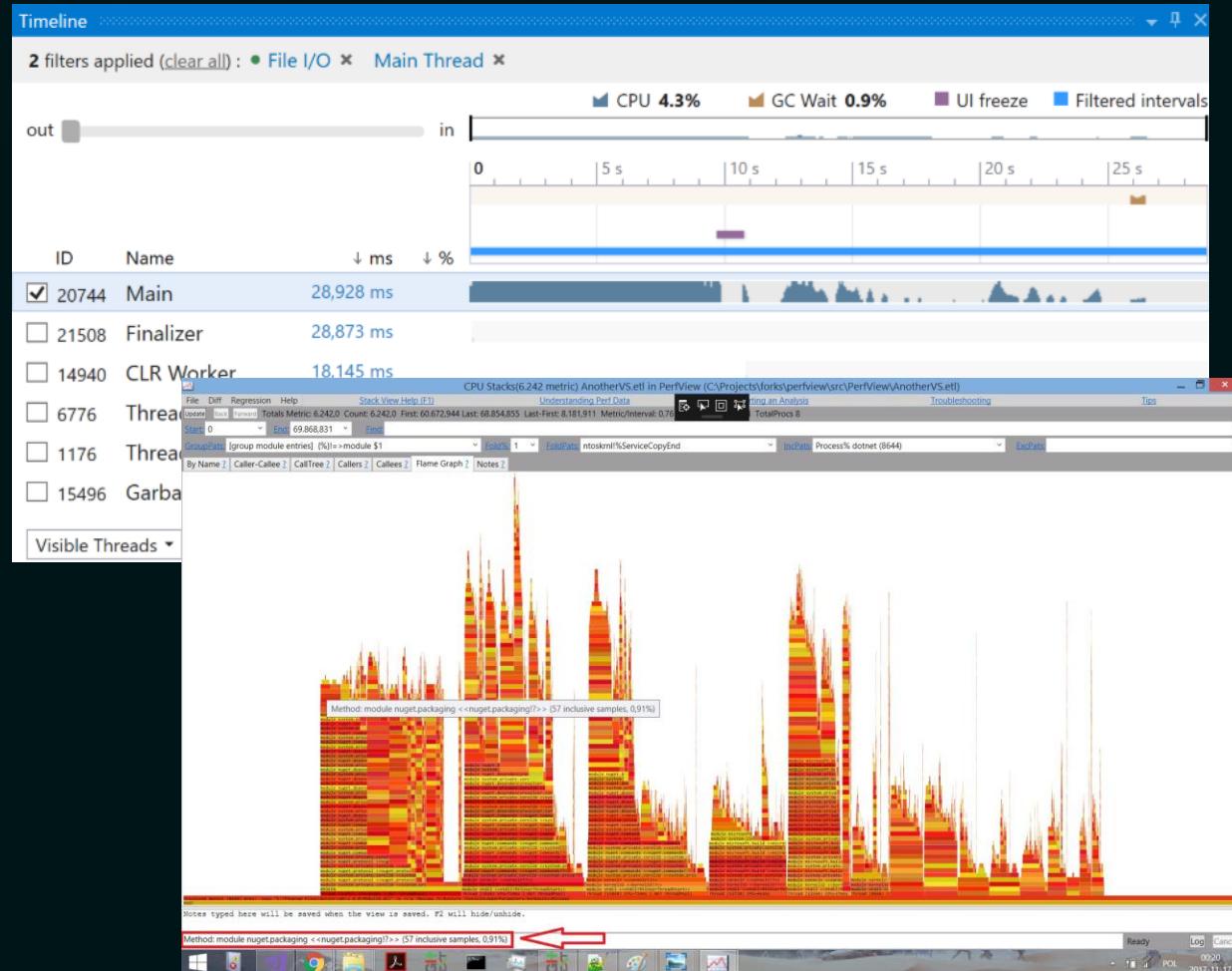
- Sampling / snapshots
- Timeline / tracing

FERRAMENTAS:

- Jetbrains Rider / dotTrace
- Perfview
- Visual Studio Diagnostic Tools / dotnet-trace
- V8 Profiler e OX (NodeJS)

VISÕES:

- Flame graphs
- Call trees
- Top-down e bottom-up



PROFILING – MEMÓRIA

MODOS:

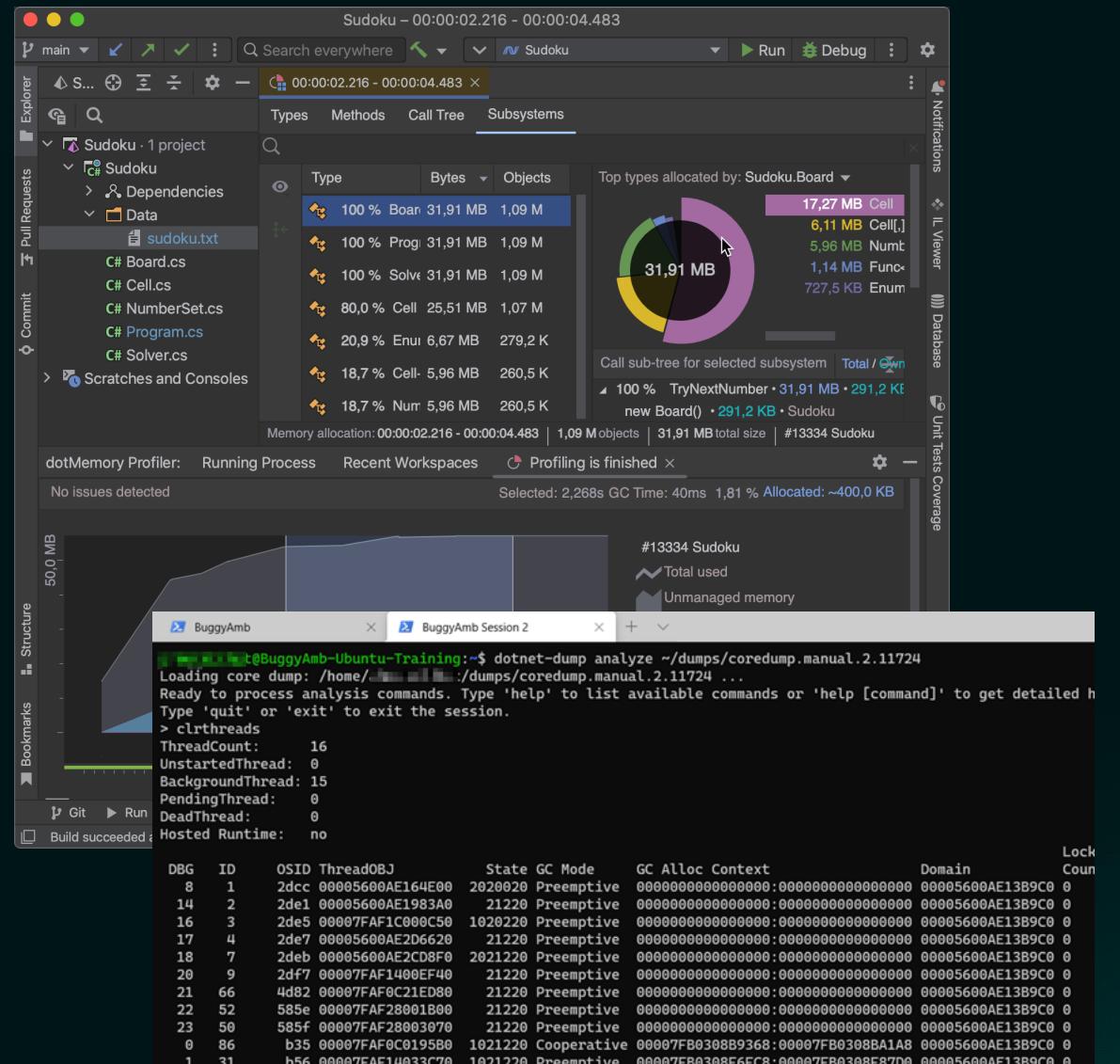
- Sampling
- Full allocations (dumps)
- Garbage Collection

FERRAMENTAS:

- Jetbrains Rider / dotMemory
- Perfview
- Visual Studio Diagnostic Tools / dotnet-dump / dotnet-gcdump

VISÕES:

- Object references count
- Memory timeline
- GC generations
- Snapshots diffing



[beyond]

HANDS ON

[beyond]

BOAS PRÁTICAS

BOAS PRÁTICAS - CÓDIGO

⚠ Atenção: esse é o seu momento de brilhar!

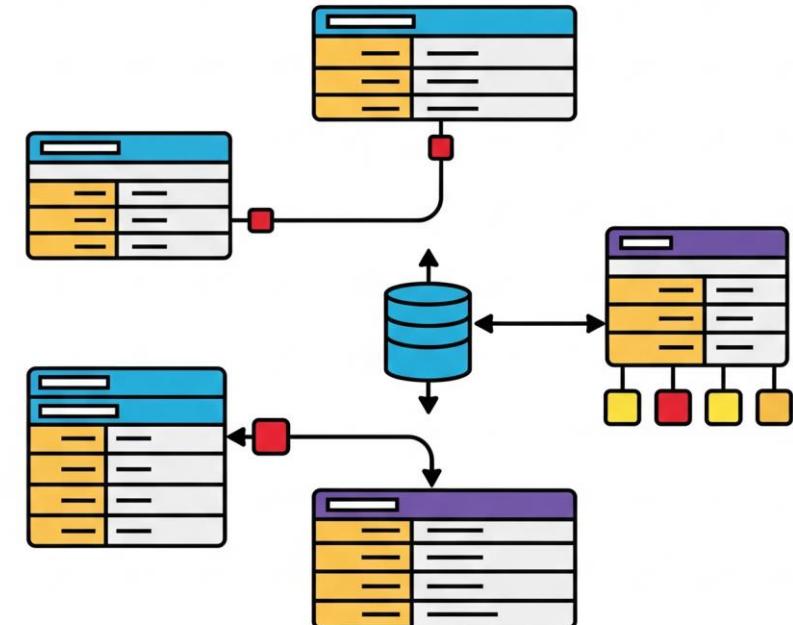
- Escolha bem seus algoritmos (Big O );
- Utilize cache sempre que possível (com TTL, claro);
- Use o paralelismo de forma inteligente;
- Evite reflection a todo custo;
- Lançar exceções custa caro. Prefira uma abordagem com result pattern;
- Utilize CancellationTokens (.NET) e AbortControllers (Node.JS) como grandes aliados;
- Nunca bloqueie código assíncrono;



BOAS PRÁTICAS – ACESSO A BANCOS DE DADOS

 Você não precisa ser um DBA. Mas precisa conhecer alguns conceitos.

- SQL x NoSQL não é uma questão de gosto;
- Escolha bem seus índices;
- Monitore seu pool de conexões;
- Prefira eager loading a lazy loading;
- Retorne do banco apenas os dados que de fato serão utilizados;
- Pagine as respostas sempre que possível;
- Cuidado com ORMs “milagrosos”;
- Replication e Sharding salvam vidas;



BOAS PRÁTICAS – REDES

⌚ Cada byte e cada milissegundo contam.

- Geografia importa: quanto mais longe, mais lento;
- Conteúdo estático sempre pode ser cacheado. Valorize a CDN!;
- Compacte seus dados, tanto a nível de aplicação quanto de transporte;
 - GZIP e Brotli;
 - XML, JSON e Protobuf;
- Protocolos mais modernos são muito mais eficientes (HTTP/3 e WebTransport);



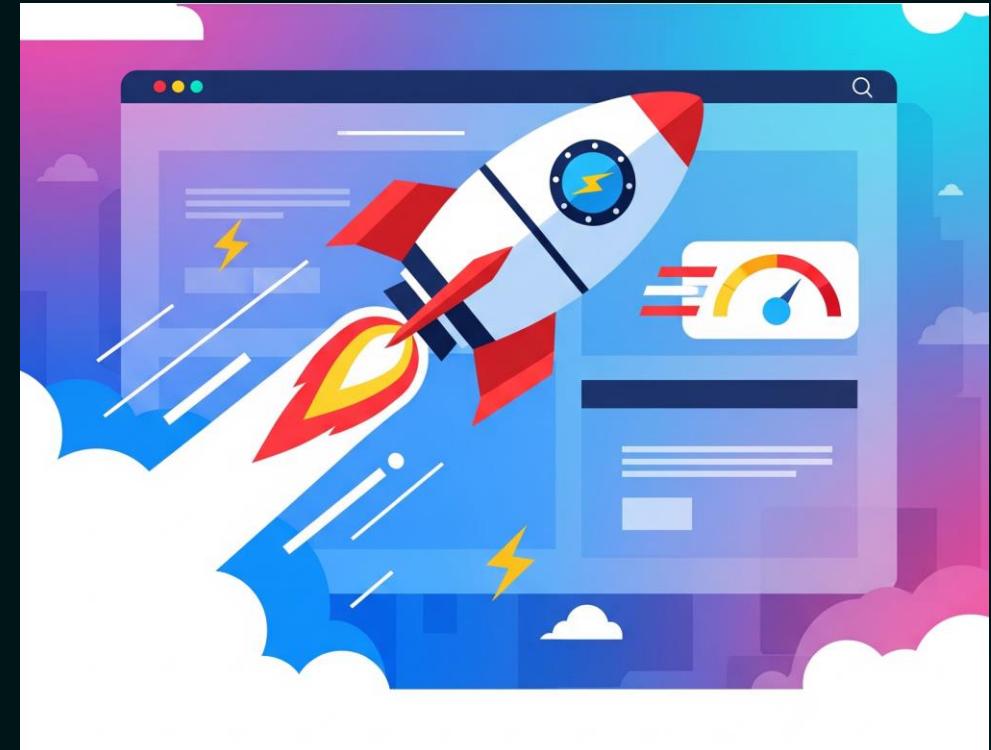
BOAS PRÁTICAS - CULTURA

Performance não é um projeto.

Performance é uma cultura.

- Zelo profissional;
- Performance como DoD – Definition of Done;
- Pare de adivinhar: meça, analise e melhore;
- Defina SLOs, SLIs e um modelo de maturidade;

“A cultura de performance não é sobre ser o mais rápido a qualquer custo. É sobre ser consistentemente rápido para encantar seus usuários e habilitar o negócio a crescer.”



ONDE APRENDER MAIS?

LIVROS:

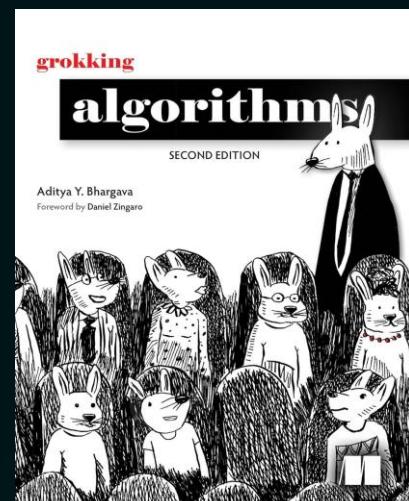
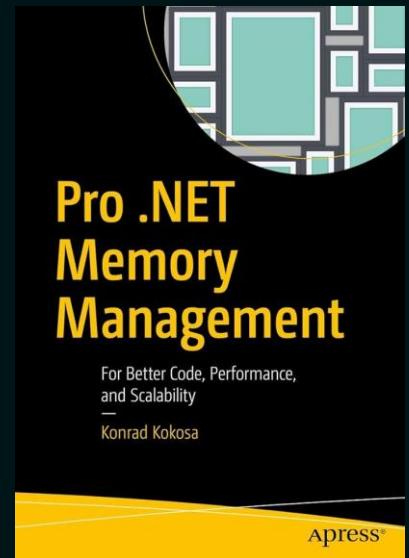
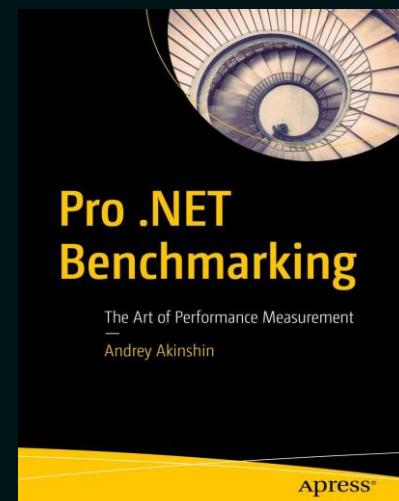
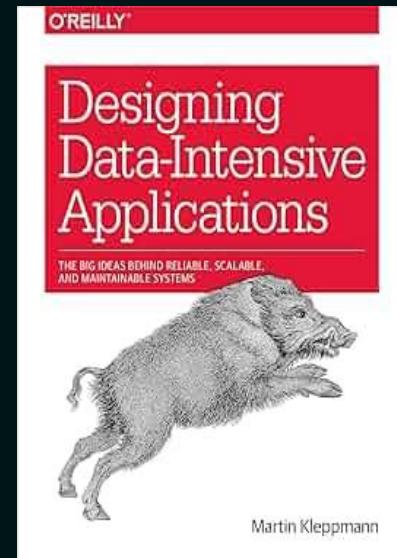
- [Designing Data-Intensive Applications \(Martin Kleppmann\)](#)
- [Pro .NET Memory Management \(Konrad Kokosa\)](#)
- [Pro .NET Benchmarking \(Andrey Akinshin\)](#)
- [Grokking Algorithms \(Aditya Y Bhargava\)](#)

VÍDEOS E BLOGS:

- [Linux Systems Performance \(Brendan Gregg\)](#)
- [Como Monitorar a Performance De Apps Node.js \(Erick Wendel\)](#)
- [Profiling and Fixing Common Performance Bottlenecks \(Jetbrains\)](#)
- [Writing High-Performance C# and .NET Code \(Steve Gordon\)](#)
- [The Latency Gambler Blog \(Medium\)](#)

DOCS OFICIAIS:

- [.NET overview of profiling tools \(Microsoft\)](#)
- [Profiling Node.JS Applications \(NodeJS.Org\)](#)



[beyond]

Q&A

[beyond]

OBRIGADO!

[beyond]