

Ficou lento, e agora?

Como garantir a performance da sua aplicação no Azure

George Luiz Bittencourt

Sobre mim



George Luiz Bittencourt

Arquiteto de soluções cloud com mais de 20 anos de experiência em infraestrutura e desenvolvimento de software.



/glzbcrt

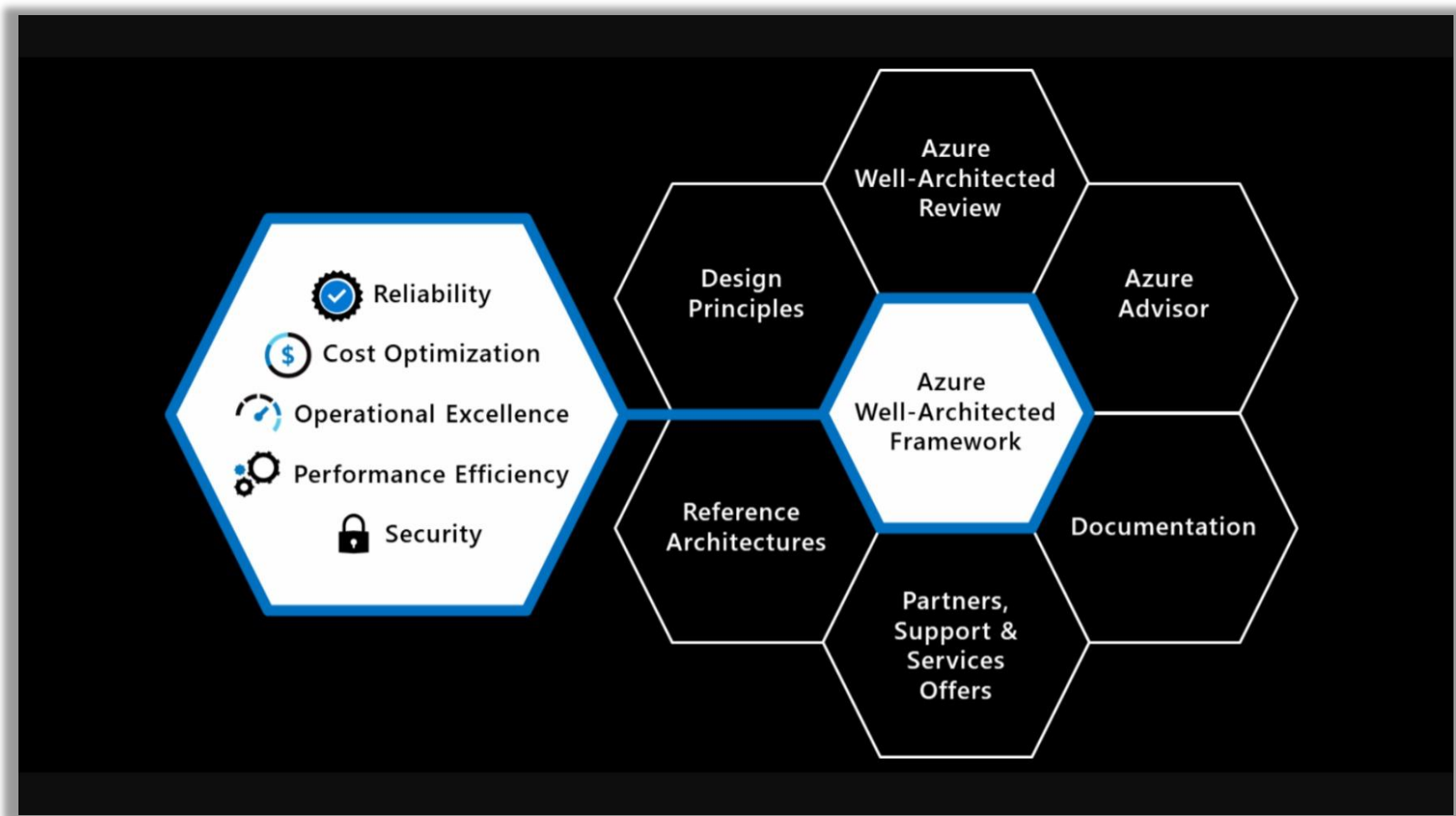


/glzbcrt



george.bittencourt@microsoft.com

Azure Well-Architected Framework



[Referência: Microsoft Azure Well-Architected Framework](#)

Azure Well-Architected Framework

*Eficiência em performance é a capacidade de **ajustar** uma arquitetura de **forma manual ou automática** para **atender** a demanda dos **usuários**.*

Azure Well-Architected Framework

Princípios

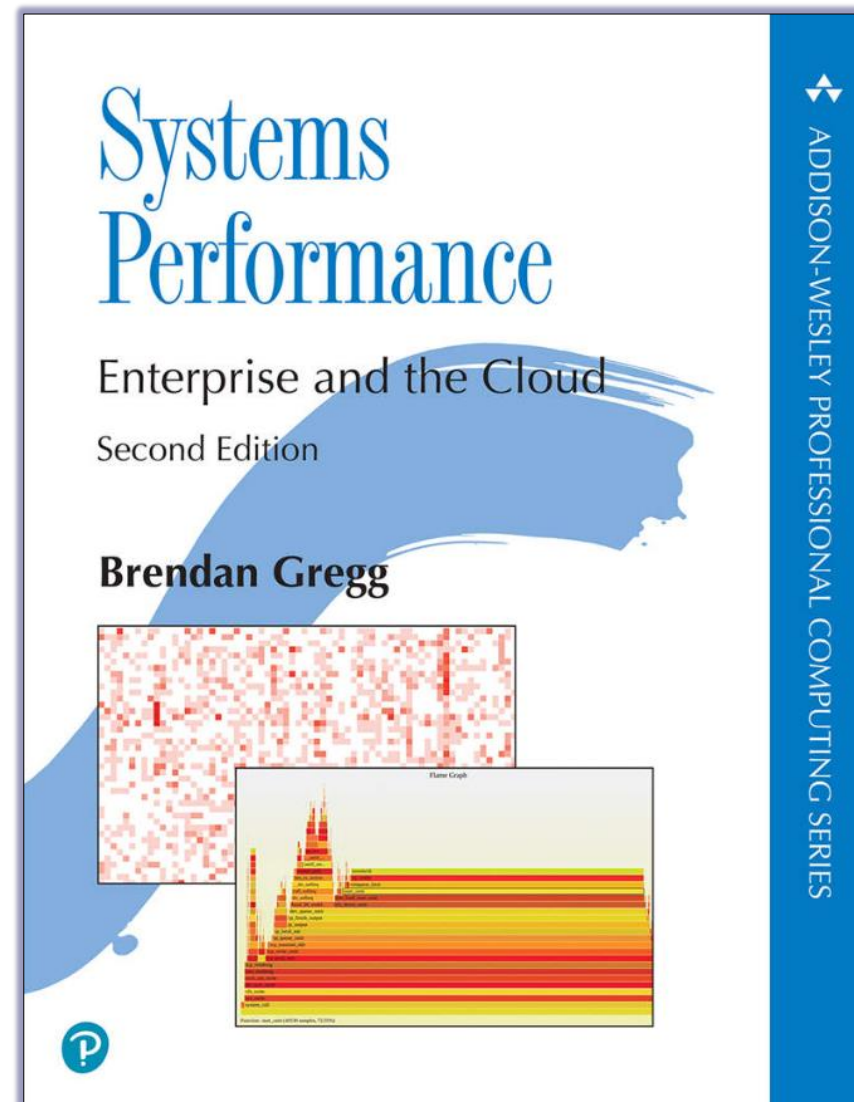
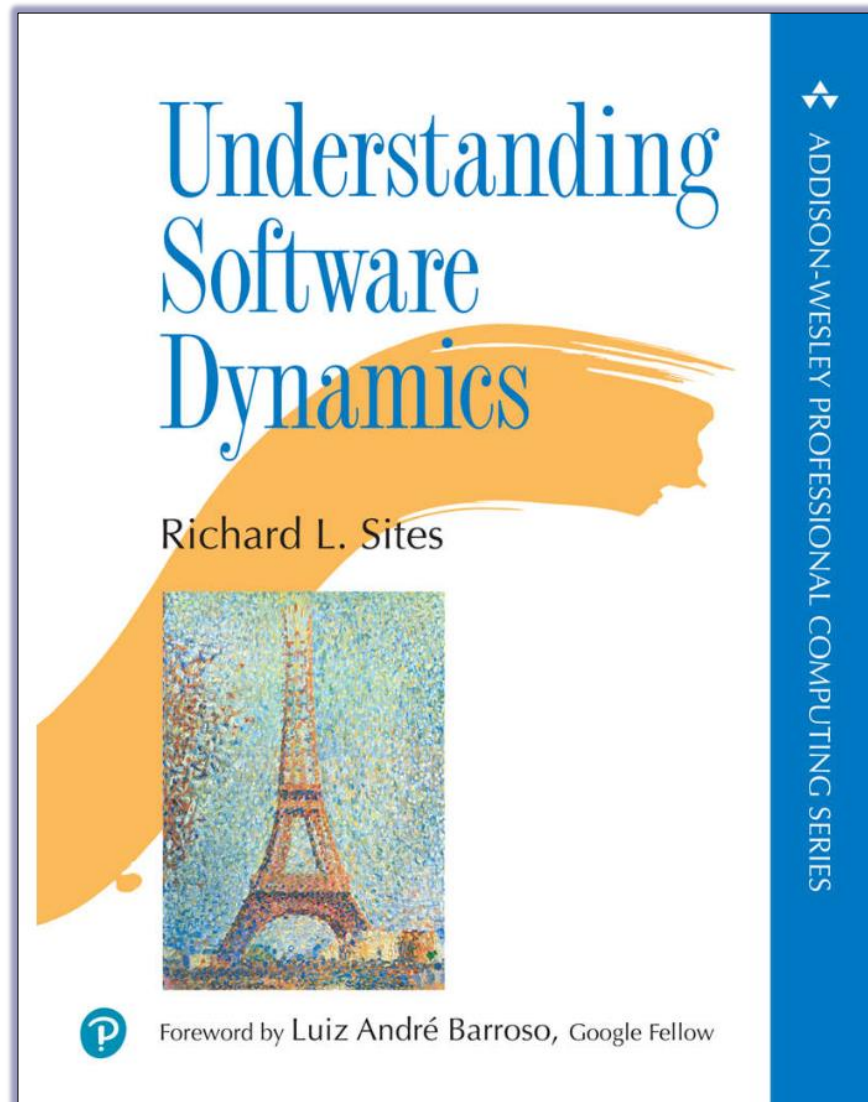
- Escalabilidade Horizontal
 - *Utilizar serviços gerenciados.*
 - *Escolher os componentes certos com os tamanhos corretos.*
- Shift-left dos testes de performance
 - *Executar testes de carga com cargas padrões e com a carga máxima esperada.*
 - *Estabelecer um baseline de performance.*
- Monitoramento contínuo em produção
 - *Avaliar de forma recorrente as alterações no uso dos componentes.*
 - *Criar alertas de monitoramento.*

Azure Well-Architected Framework

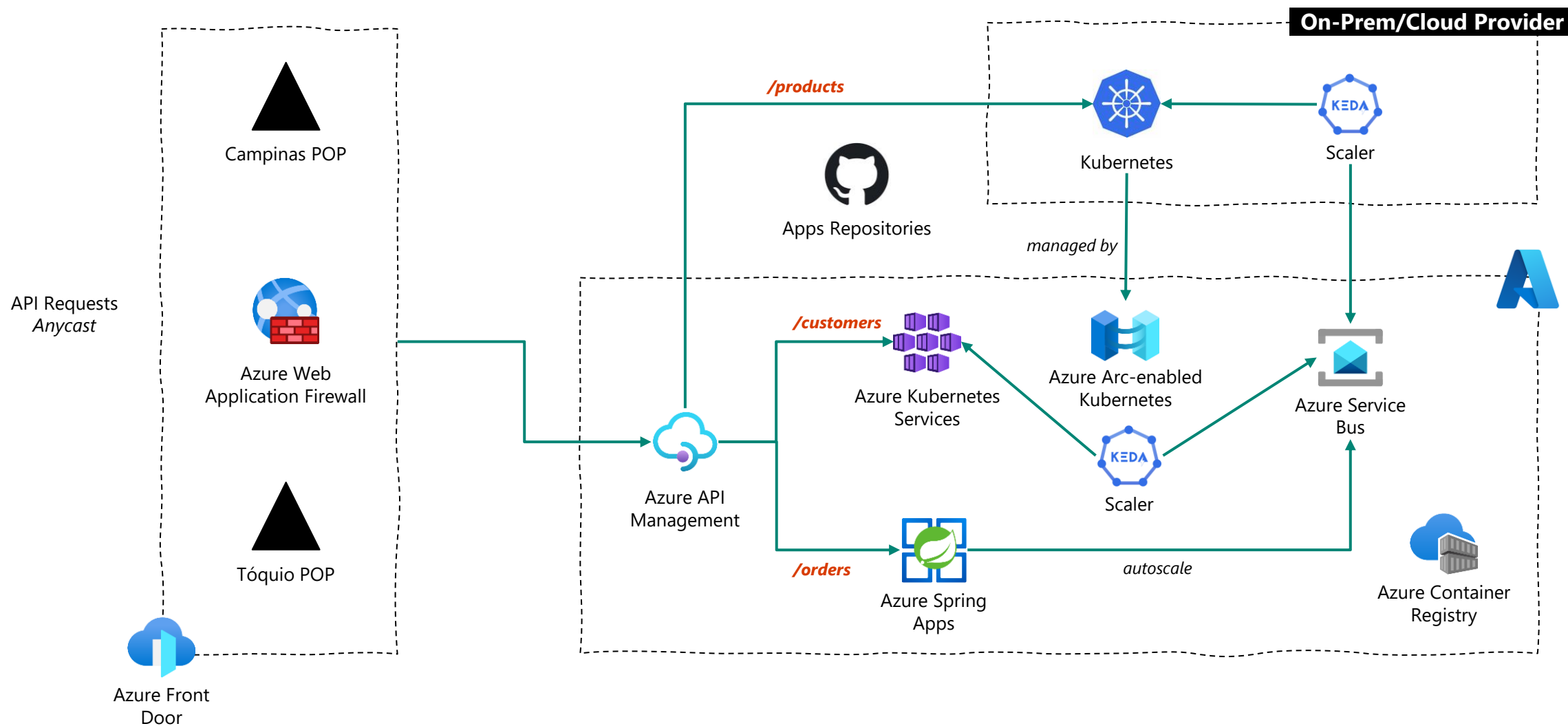
Princípios

- Escalabilidade Horizontal
 - *Utilizar serviços gerenciados.*
 - *Escolher os componentes certos com os tamanhos corretos.*
- Shift-left dos testes de performance
 - *Executar testes de carga com cargas padrões e com a carga máxima esperada.*
 - *Estabelecer um baseline de performance.*
- Monitoramento contínuo em produção
 - *Avaliar de forma recorrente as alterações no uso dos componentes.*
 - *Criar alertas de monitoramento.*

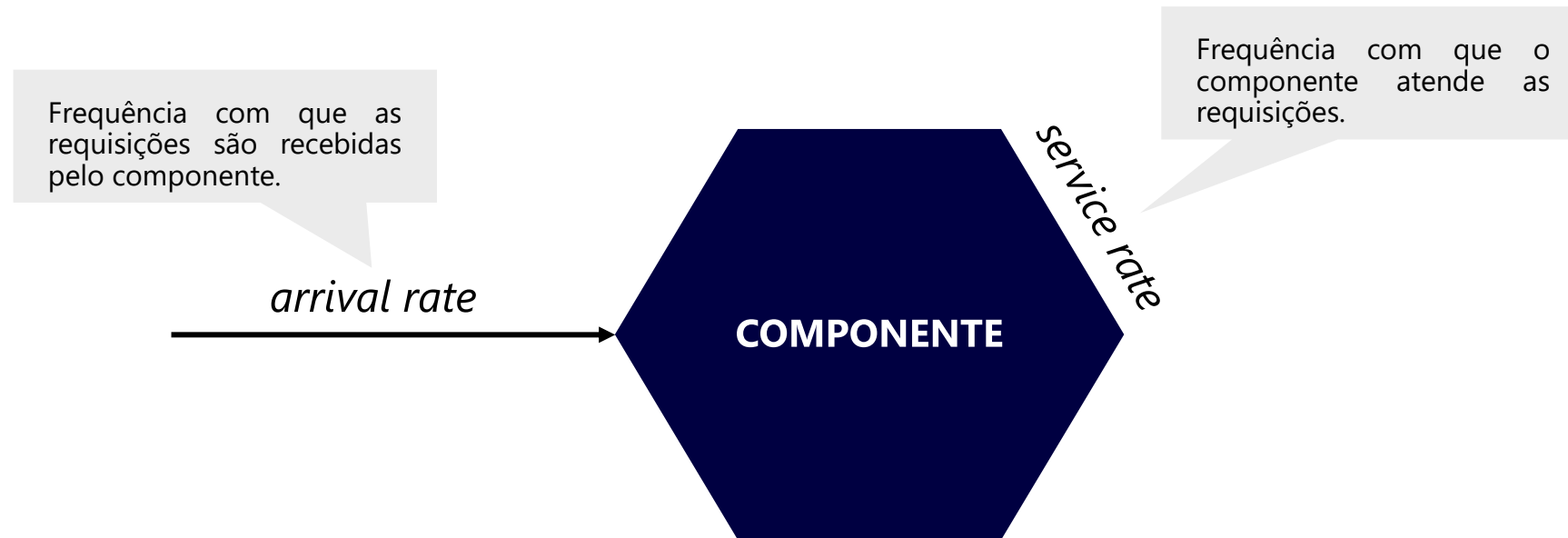
Livros Recomendados



Arquitetura de Referência



Queueing Theory



Cenário	Descrição
Arrival Rate > Service Rate	Componente menor que o necessário para atender o demanda. Filas serão formadas, a latência vai aumentar e erros/timeouts vão ser gerados.
Arrival Rate < Service Rate	Componente maior que o necessário. As requisições serão atendidas com folga, porém um custo maior será gerado.
Arrival Rate = Service Rate	É o melhor cenário, o componente está com tamanho adequado atendendo as requisições e não existe custo ocioso.

Throughput, Taxa, Vazão

- De forma simples é a relação entre dois números.
 - Quantos megabytes eu gravo por segundo em um disco. Unidade → *MB/s*
 - Quantas requisições minha API responde por segundo. Unidade → *rps*
 - Quantos bytes de memória eu preciso para processar cada requisição. Unidade → *bytes/request*
 - *Quantos bits é possível enviar por segundo em uma conexão TCP de longa distância.* Unidade → *b/s*
- O *throughput* de um componente é o seu *service rate*, dessa forma quanto maior for, mais solicitações ela poderá atender sem gerar filas, timeouts, etc.
- Vários fatores influenciam no *throughput*, como:
 - Algoritmos e estruturas de dados escolhidas pelo componente.
 - Otimizações geradas pelos compiladores.
 - Contensões geradas por *locks* para serializar execuções.
 - Complexidade de componente.

Latência e Resolução

- Latência

- É o tempo entre dois eventos.
 - Quanto tempo leva para enviar um byte do ponto A para o ponto B?
 - Quanto tempo é necessário para alocar 1 MB de memória?
 - Quanto tempo um requisição leva para atender um GET com o dado em *cache* (cache hit)? E quando o dado não está em *cache* (cache miss)?
- Em *networking* é muito utilizado e se mede o tempo de ida e volta de um pacote, conhecido como *round trip time*.
 - Quanto mais distante estamos do destino, maior é a latência. Nada é mais rápido que a velocidade de luz.
 - A rota e os roteadores também impactam na latência. Roteadores mais ocupados ou com problema tem latência maior.
- Em *browsers* é comum a métrica Time to First Byte (TTFB).

- Resolução

- É o menor valor que um sistema de medição pode medir.
 - Um sistema que mede o tempo em segundos não consegue medir um evento menor que um segundo.
 - Se coletamos o uso de CPU a cada minuto, o componente pode estar saturado em boa parte de um minuto e não teremos visibilidade.

Conceitos Básicos

Múltiplos

Múltiplo	Símbolo	Valor
Quilo	k	10^3
Mega	M	10^6
Giga	G	10^9
Tera	T	10^{12}
Peta	P	10^{15}
Exa	E	10^{18}

Sub Múltiplos

Múltiplo	Símbolo	Valor
Mili	m	10^{-3}
Micro	μ	10^{-6}
Nano	n	10^{-9}
Pico	p	10^{-12}
Femto	f	10^{-15}
Atto	a	10^{-18}

Workload Characterization

- O *workload characterization* é um método que pode ser utilizado tanto durante a fase de design quanto para resolver problemas em produção.
- Consiste em responder as 4 perguntas abaixo. Com elas é possível entender melhor a carga atual ou prevista para a aplicação.
 - **Quem** está causando a carga?
 - *Processo*
 - *Endereço IP*
 - *Usuário*
 - **Porque** a carga está sendo gerada?
 - *Qual é o stack trace gerando a carga?*
 - **Quais** são as características da carga?
 - *IOPS*
 - *throughput*
 - *memória*
 - **Como** a carga está mudando com o passar do tempo?
 - *Alguma nova funcionalidade foi liberada?*
 - *O volume de usuários está aumentando com o passar do tempo?*

USE

- O método **Utilization, Saturation, and Errors (USE)** é uma forma rápida de identificar a causa de um problema de performance.
- Consiste em avaliar três métricas dos recursos envolvidos na arquitetura.
- É importante ter um *baseline* para referência para diferenciar um comportamento normal de um anormal.
- *Utilization:*
 - *Qual é percentual de uso do componente em um intervalo?*
- *Saturation:*
 - *O componente está saturado?*
 - *Existe fila sendo gerada?*
- *Errors:*
 - *A quantidade de erros geradas.*

Load Distribution

- Raramente a carga é distribuída de maneira homogênea no período.
- Cada métrica é uma série temporal e métricas de estatística descritiva podem ser utilizadas para entender a métrica, como: média, moda, percentil, desvio padrão, etc.
- A relação de uma métrica em relação a outra pode ser avaliada com o uso de correlação. Exemplo: como a CPU se comporta com o aumento da latência de disco.

Hora	% CPU
00:00	0
01:00	100
02:00	0
03:00	100
04:00	0
05:00	100
06:00	0
07:00	100
08:00	0
09:00	100
10:00	0
11:00	100

Média: 50%
Desvio Padrão: 50%
Percentil 90%: 100%

Hora	% CPU
00:00	5
01:00	80
02:00	30
03:00	0
04:00	20
05:00	66
06:00	100
07:00	100
08:00	75
09:00	45
10:00	20
11:00	100

Média: 53%
Desvio Padrão: 36%
Percentil 90%: 100%

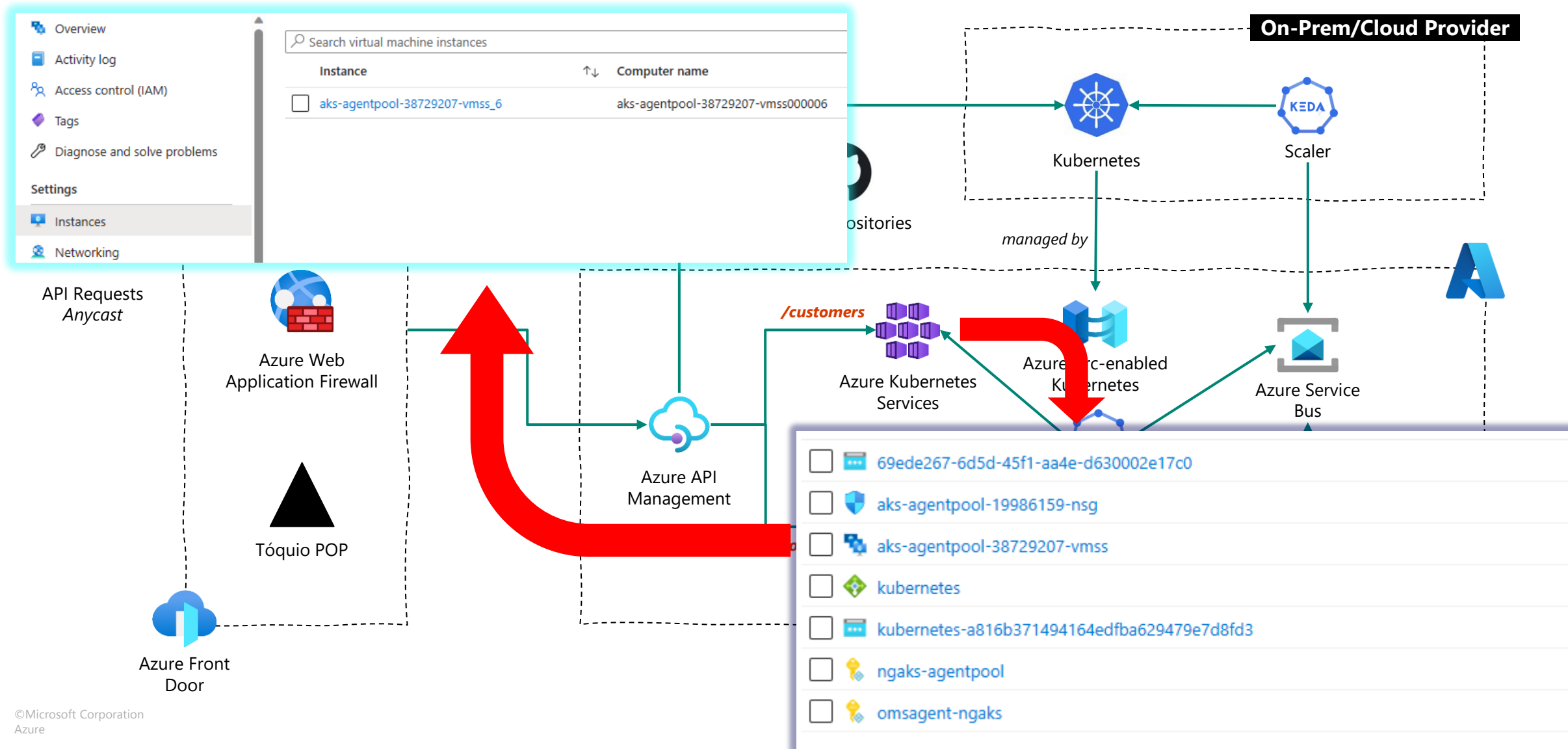
Hora	% CPU
00:00	50
01:00	50
02:00	50
03:00	50
04:00	50
05:00	50
06:00	50
07:00	50
08:00	50
09:00	50
10:00	50
11:00	50

Média: 50%
Desvio Padrão: 0%
Percentil 90%: 50%

E o código?

- Os algoritmos e estruturas de dados escolhidas tem papel fundamental no desempenho de um sistema.
 - Estruturadas de dados → arrays, linked list, map, etc.
 - Pesquisa → binary search, B tree, etc.
 - Ordenação → selection sort, quick sort, etc
- As dimensões espaço (memória e disco) e tempo (CPU) tem impacto direto nos recursos necessários.
- Em sua grande maioria os frameworks utilizadas utilizam bons algoritmos e estrutura de dados, porém casos específicos podem se beneficiar de algoritmos específicos.
- A notação *Big O* é fundamental na comparação de algoritmos.

Arquitetura de Referência



Limites

- Todo componente, seja na nuvem ou *on-premises*, possui limites.
- Componentes *on-premises* acabam sendo super estimados já que o ambiente não é elástico.
- É importante entender os limites de cada componente durante a fase de arquitetura.
- Geralmente quando o componente atinge o seu limite ocorre *throttling* e a aplicação recebe alguma forma de erro.
- Alguns limites no Azure podem ser alterados através de *tickets* de suporte.
- Referência: [Azure subscription limits and quotas](#)
- Alguns exemplos:
 - [Discos](#)
 - [Máquinas Virtuais](#)
 - [Storage Account](#)

Responsabilidade Compartilhada

CLIENTE

- *Entender os limites de cada componente.*
- *Aplicar boas práticas de engenharia para tratar os limites.*
- *Monitorar a aplicação identificando os hot-spots e implementar soluções para transpor.*

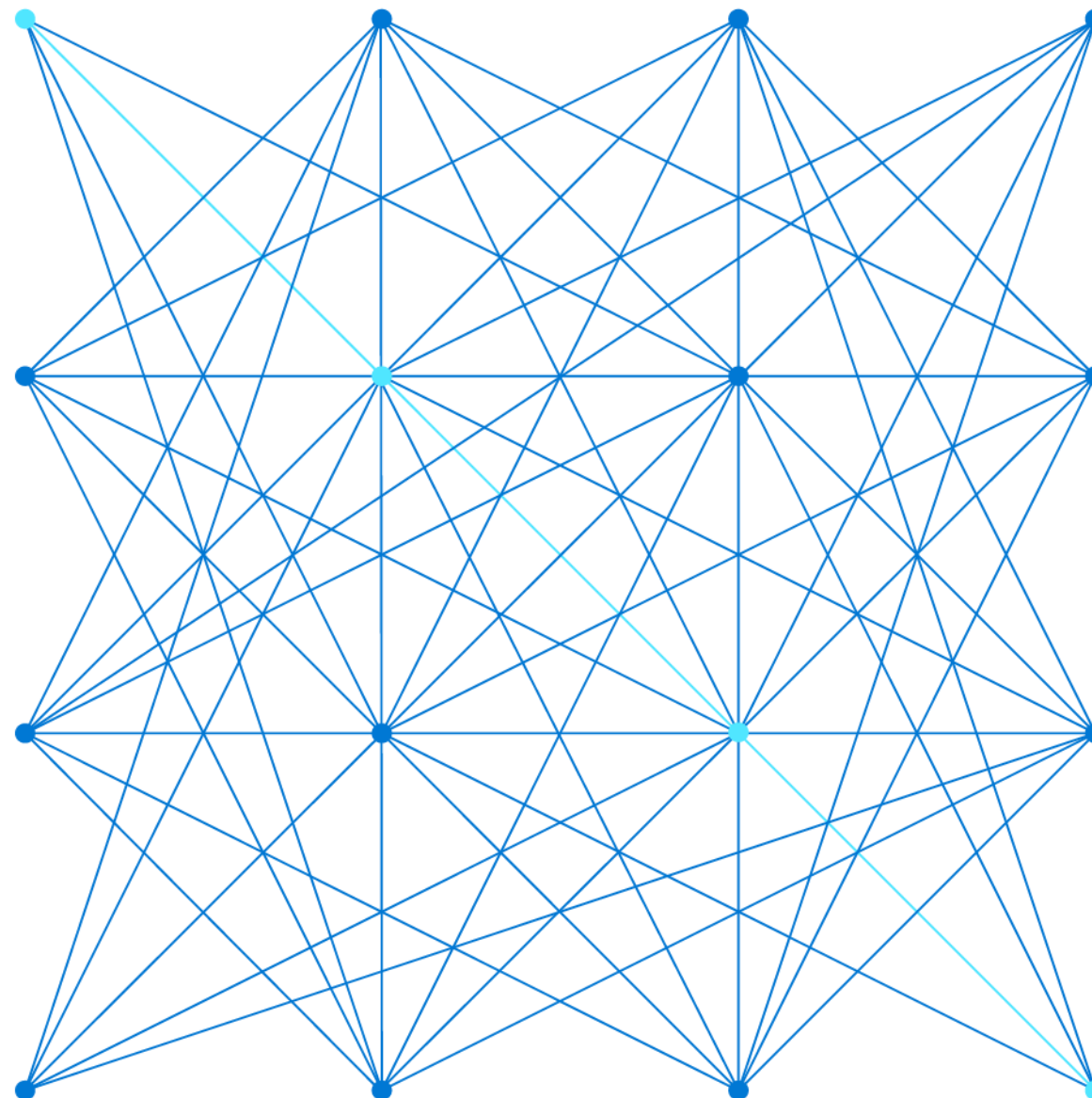
MICROSOFT

- *Disponibilizar **serviços** com performance adequada.*
- *Documentar os limites de cada serviço e seus custos.*
- *Fornecer observabilidade dos componentes.*
- *Documentar boas práticas e arquiteturas de referência.*

Benchmark Disco

- Mede a performance de um conjunto de discos com base em um *workload* sintético.
- O ideal é que *workload* sintético seja o mais representativo possível do *workload* real.
- Operações de gravação são mais onerosas que operações de leitura.
- Fatores importantes a serem considerados:
 - HDD vs SSD
 - % de gravações VS % de leituras
 - operações sequencias VS operações aleatórias
 - tamanho do bloco utilizado (4kiB, 1MiB, etc)
 - uso de cache de leitura e/ou gravação
 - multi-thread
 - queue depth
 - limites do disco e da máquina virtual
- Ferramentas:
 - [diskspd](#): ferramenta da Microsoft para Windows desenvolvida de forma *open-source*.
 - [fio](#): ferramenta *open-source* para Linux.

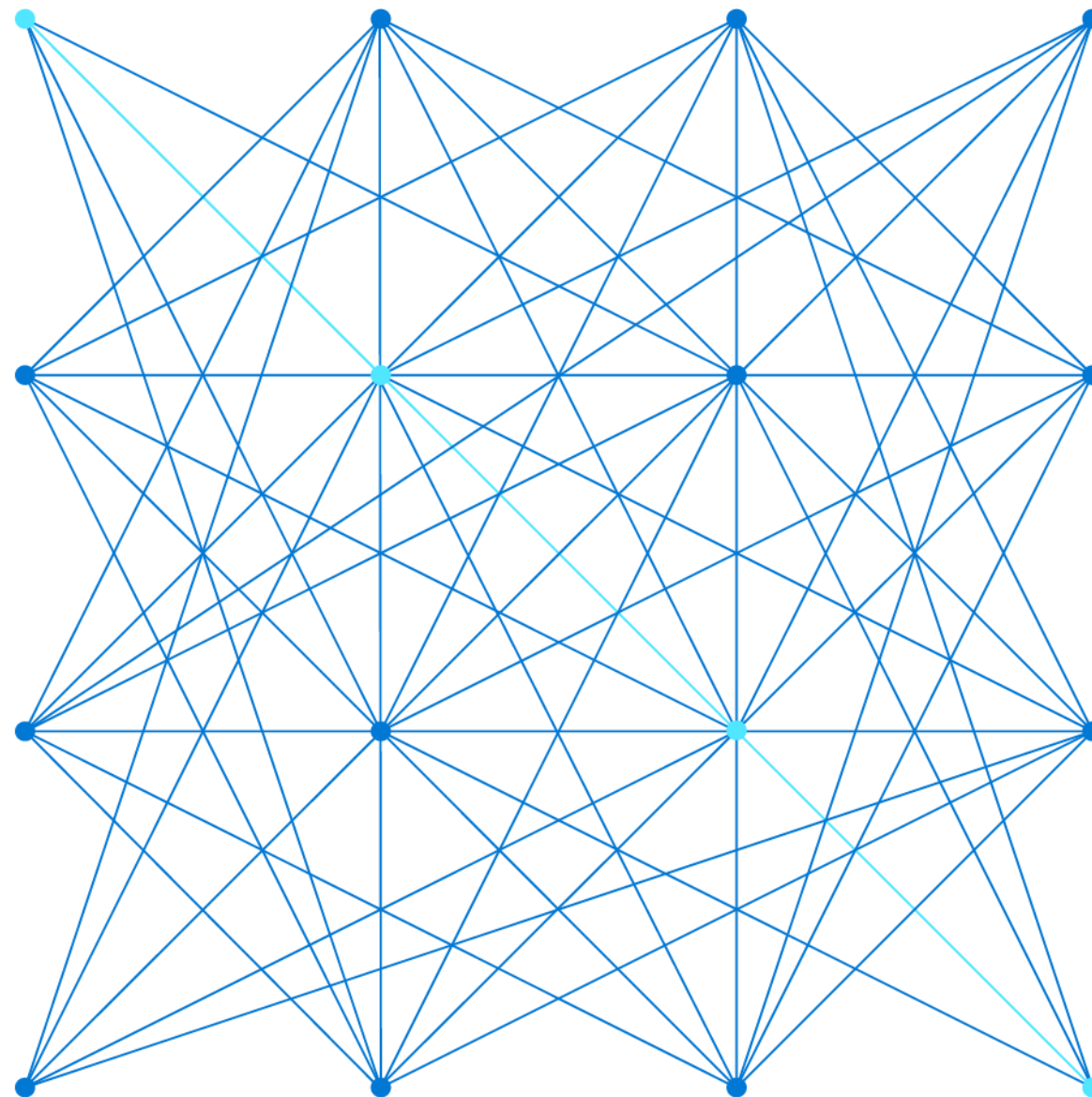
DEMO



Benchmark Rede

- Mede a performance entre dois pontos da rede.
- Pontos diferentes podem ter performance diferente.
- Conexões TCP com alta latência tem throughput baixo.
- Fatores importantes a serem considerados:
 - distância física entre os pontos
 - largura de banda disponível
 - saturação dos componentes
 - VPN, WAF, IDPS, etc
 - fragmentação
 - quantidade de conexões
 - limites da máquina virtual
- Ferramentas:
 - [iperf](#): ferramenta open-source para Windows e Linux.

DEMO



Benchmark CPU

- Processadores diferentes tem performances diferentes.
- Fatores importantes a serem considerados:
 - clock
 - quantidade de cache L1, L2, L3
 - instruções suportadas para operações específicas como criptografia, multimídia, etc.
 - hyper-threading
 - # de cores
 - NUMA vs UMA
 - memória
 - compiladores e possíveis otimizações
- Ferramentas
 - [BenchmarkDotNet](#): ferramenta *open-source* para executar benchmark de códigos .NET.

Benchmark APIs

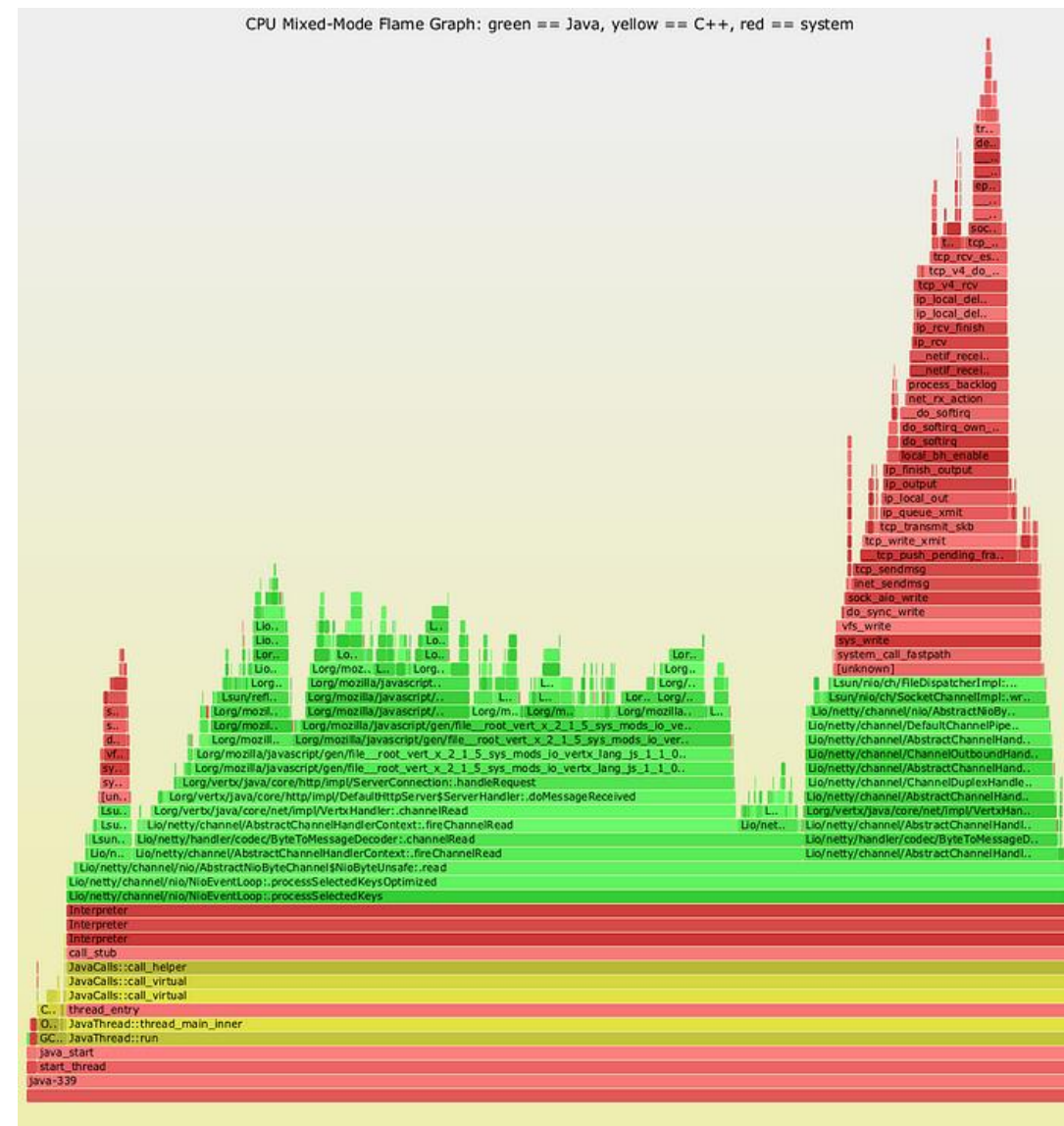
- Fatores importantes a serem considerados:
 - formato utilizado (JSON, ProtoBuf, etc)
 - chamadas únicas VS chamadas em batch
 - SSL Offloading vs SSL E2E
 - uso de cache
- Ferramentas
 - Azure Load Testing
 - K6
 - jMeter

Ferramentas

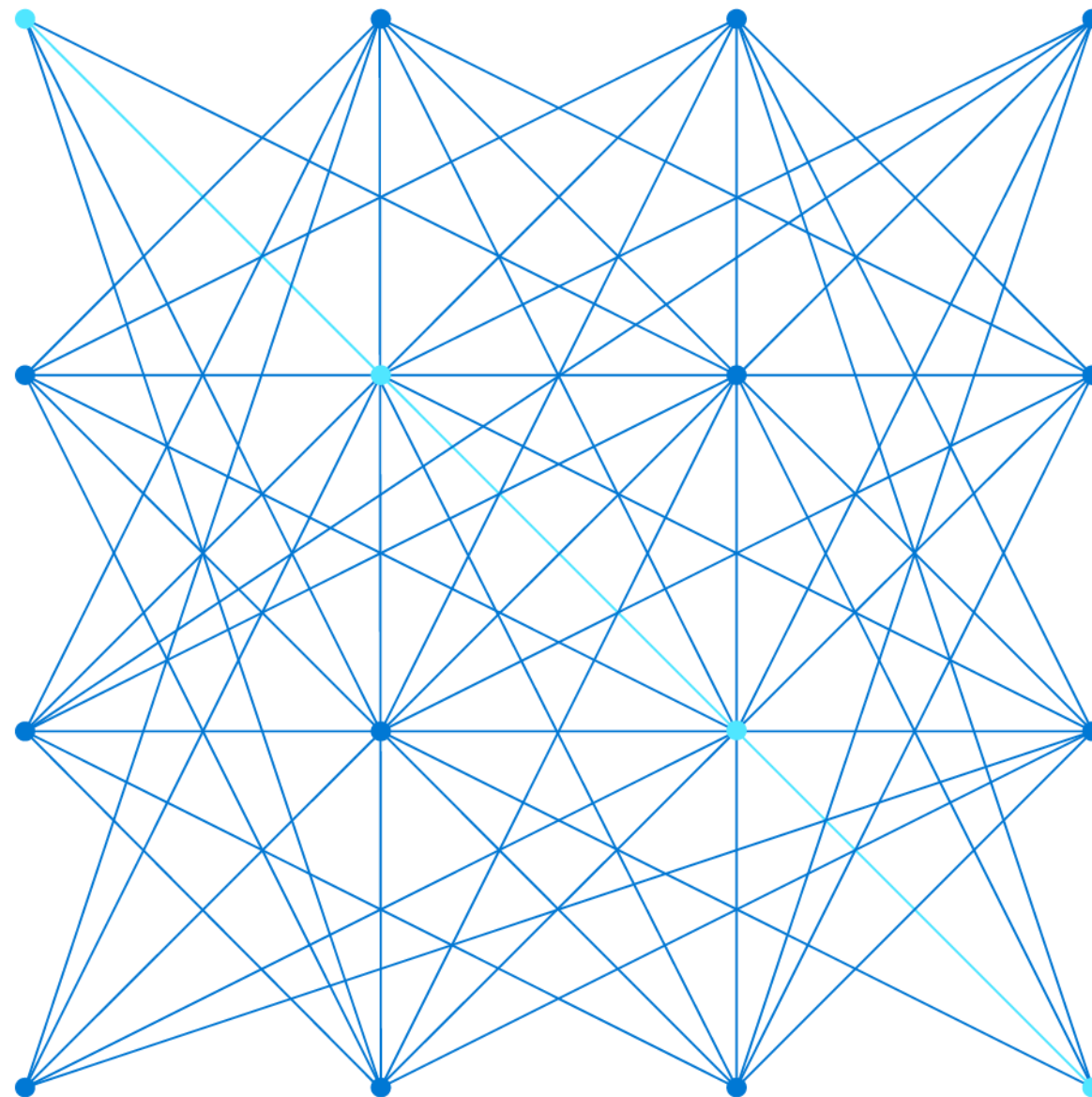
- Azure
 - Metrics disponíveis em cada recurso
 - Application Insights
 - Azure Load Testing
- Windows
 - Performance Monitor
 - Process Monitor
 - Event Tracing for Windows (ETW)
 - PerfView
- Linux
 - iostat, vmstat, lsof, strace, tcpdump, etc
 - eBPF
- Geral
 - Profilers de CPU e memória:
 - dotTrace, dotMemory
 - JProfiler

Flame Graphs

- É uma forma de visualização para entender rapidamente onde o tempo está sendo mais dispendido.
- No eixo X temos as funções e no eixo Y a profundidade da pilha de chamadas. Quanto maior a largura de uma função, mais tempo ela esteve presente.
- Referência: [Flame Graphs \(brendangregg.com\)](http://brendangregg.com)



DEMO



Checklist

- ✓ Os limites de cada componente utilizado na arquitetura são conhecidos?
- ✓ Os componentes selecionados escalam de forma horizontal?
- ✓ Foram efetuados testes de performance? Foi criado um baseline de performance?
- ✓ A carga esperada foi definida?
- ✓ Estão sendo utilizadas políticas de auto-scale?
- ✓ O código utiliza algoritmos e estrutura de dados corretas?
- ✓ Onde é possível, está sendo utilizado cache?
- ✓ Os dados estão particionados horizontalmente?

EOF



SCAN ME