25 Perguntas e Respostas sobre Aplicações e Serviços de Computação em Nuvem

Estrutura: Perguntas alinhadas com padrões dos testes (2022-2024)

1. Virtualização e Máquinas Virtuais

Pergunta 1: Comparação de Abordagens de Virtualização

Concorda com a afirmação de que "a virtualização completa (full virtualization) é sempre preferível à paravirtualização em termos de desempenho global"? Justifique a sua resposta com exemplos concretos.

Resposta: Discordo parcialmente. Embora ambas as abordagens tenham vantagens e desvantagens distintas:

Virtualização Completa (ex: VirtualBox):

- **Vantagem**: Maior portabilidade – o SO convidado não requer modificações
- **Desvantagem**: Menor desempenho, pois todas as instruções do SO convidado devem ser traduzidas para o hardware físico

Paravirtualização (ex: Xen):

- **Vantagem**: Melhor desempenho de CPU e I/O, pois o SO convidado contém "hooks" (modificações) que evitam a tradução completa
- **Desvantagem**: Requer modificações no SO convidado, prejudicando a portabilidade

Conclusão: A escolha depende do contexto. Para ambientes onde o desempenho é crítico e as modificações de SO são aceitáveis, paravirtualização é preferível. Para máxima compatibilidade e portabilidade, virtualização completa é melhor.

Pergunta 2: Hypervisors Tipo 1 vs Tipo 2

Compare os hypervisors tipo 1 (bare metal) e tipo 2 (hosted), indicando um cenário onde cada um é mais apropriado. Justifique.

Resposta:

Aspeto	Tipo 1 (Bare Metal)	Tipo 2 (Hosted)
Instalação	Diretamente no hardware	Como aplicação sobre SO
Desempenho	Máximo (sem overhead de SO)	Reduzido (SO anfitrião consome recursos)
Requisitos Hardware	Suporte específico de virtualização	Menos exigente

Aspeto	Tipo 1 (Bare Metal)	Tipo 2 (Hosted)
Exemplo	VMware ESX	VirtualBox, KVM

Cenários apropriados:

- **Tipo 1**: Centros de dados e ambientes de produção onde o desempenho máximo é crítico (ex: hospedagem de múltiplas VMs em escala)
- **Tipo 2**: Desenvolvimento local, testes e máquinas pessoais onde a flexibilidade e facilidade de instalação são prioritárias

---

Pergunta 3: Os Três Pilares da Virtualização

Indique e explique os três pilares fundamentais que justificam o uso da virtualização na cloud. Dê exemplos práticos.

Resposta:

1. **Consolidação e Eficiência**
    - Múltiplos SOs e aplicações correm no mesmo hardware físico
    - Exemplo: Um servidor físico com 4 cores pode alocar 2 cores a uma VM Linux e 2 a uma VM Windows
    - Benefício: Redução de custos de infraestrutura e melhor utilização de recursos
2. **Isolamento e Segurança**
    - Cada VM é isolada; uma VM comprometida não afeta outras
    - Exemplo: Se uma VM for atacada, as restantes continuam funcionais
    - Benefício: Segurança, desempenho dedicado e fiabilidade
3. **Abstração e Transparência**
    - A heterogeneidade do hardware é abstraída
    - Exemplo: Uma aplicação desenvolvida para x86 pode correr em qualquer máquina física x86 sem modificações
    - Benefício: Portabilidade e simplicidade para o utilizador final

---

2. Contentores e Docker

Pergunta 4: Diferença Fundamental entre Imagens e Contentores Docker

Qual é a diferença entre uma imagem Docker e um contentor Docker? Explique a relação entre ambos.

Resposta:

- **Imagem Docker**: Um ficheiro imutável (read-only) que funciona como um "molde" ou "planta". Contém tudo o que é necessário para executar uma aplicação: código, dependências, bibliotecas, variáveis de ambiente, etc.

- **Contentor Docker**: Uma instância em execução de uma Imagem. É o resultado de "executar" uma imagem. Um contentor tem estado mutável (pode ser modificado durante a execução) e pode ser criado, reiniciado ou removido.

Analogia: Se uma Imagem é um ficheiro executável (como um programa .exe), um Contentor é o programa em execução.

Exemplo prático:

```
# Criamos uma imagem (uma vez)
docker build -t minha-app:1.0 .

# Criamos 3 contentores a partir da mesma imagem
docker run -d minha-app:1.0
docker run -d minha-app:1.0
docker run -d minha-app:1.0
```

Os três contentores são independentes, embora partilhem a mesma imagem base.

Pergunta 5: Vantagens dos Contentores sobre Máquinas Virtuais

Indique duas vantagens principais dos contentores relativamente às máquinas virtuais e explique o impacto na eficiência de infraestruturas de computação em nuvem.

Resposta:

1. **Leveza e Velocidade de Arranque**
    - Contentores partilham o kernel do SO anfitrião, eliminando a necessidade de um SO completo por aplicação
    - Uma VM necessita de 50GB+ de espaço, um contentor ocupa apenas alguns MB
    - Tempo de arranque: VM (~segundos a minutos), Contentor (<1 segundo)
    - **Impacto**: Escalabilidade horizontal mais rápida em ambientes cloud. Com Kubernetes, podemos escalar aplicações automaticamente em milissegundos

2. **Isolamento com Eficiência**
    - Contentores utilizam namespaces e cgroups do Linux para isolar processos sem overhead completo de virtualização
    - Múltiplos contentores podem correr num único servidor físico
    - **Impacto**: Densidade de carga muito maior. Um servidor que suportaria 5-10 VMs pode suportar 100+ contentores, reduzindo custos de infraestrutura significativamente

---

Pergunta 6: Docker vs Máquinas Virtuais em Cenários Específicos

Para replicar o servidor aplicacional da aplicação Laravel.io, não basta apenas criar vários pods Kubernetes deste servidor. Concorda com esta afirmação? Justifique.

Resposta: Concordo plenamente. Embora o Kubernetes facilite a replicação de pods (contentores), isso é apenas parte da solução. Existem componentes que não podem ser simplesmente replicados:

Problemas que surgem:

1. **Base de Dados**: Não se pode ter múltiplas instâncias de uma base de dados escrevendo nas mesmas tabelas sem mecanismos de sincronização (replicação, clustering)
2. **Sessões de Utilizador**: Se um cliente é atendido por um pod num pedido e por outro noutro pedido, é necessário armazenamento de sessões distribuído (Redis, Memcached)
3. **Sistema de Ficheiros Compartilhado**: Uploads de ficheiros necessitam de armazenamento persistente acessível por todos os pods
4. **Estado Partilhado**: Caches, filas de mensagens, etc.

Solução completa necessária:

- Replicar o servidor aplicacional (sim, com Kubernetes)
- Replicar/shardear a base de dados
- Implementar armazenamento distribuído (Volumes Kubernetes)
- Usar serviços de armazenamento temporário (Redis para sessões)

---

3. Kubernetes e Orquestração

Pergunta 7: Mecanismos de Alta Disponibilidade em Kubernetes

Assuma um serviço composto por uma aplicação web e uma base de dados instalado num único servidor. Discuta que mecanismos da tecnologia Kubernetes podem ajudar na tarefa de garantir alta disponibilidade (HA), e se esses mecanismos são suficientes por si só. Justifique.

Resposta:

Mecanismos do Kubernetes para HA:

1. **Replicação de Pods (Deployments)**
    - Cria múltiplas réplicas do servidor web
    - Se um pod falha, o Kubernetes cria automaticamente outro
    - Exemplo: `replicas: 3` garante sempre 3 instâncias ativas
2. **Auto-reparação (Self-healing)**
    - Se um pod ou nó falha, o Kubernetes detecta e substitui automaticamente
    - Monitora a saúde dos contentores continuamente
3. **Balanceamento de Carga (Service)**
    - Um serviço Kubernetes distribui o tráfego entre as réplicas
    - Os clientes contactam o serviço, não os pods individuais
4. **Multi-nó (Cluster Kubernetes)**
    - Os pods podem ser distribuídos por vários nós físicos

    - Se um nó falha, os pods são re-agendados noutro nó

Limitações - Não são suficientes por si só:

A base de dados é o problema crítico. Kubernetes replica o servidor web (stateless), mas a base de dados é stateful. Replicar apenas o contentor da BD não garante HA porque:

- Os dados precisam estar sincronizados entre réplicas (replicação de dados)
- Precisa-se de eleição de líder em caso de falha
- Necessário armazenamento persistente com high availability próprio

Solução Completa:

1. Kubernetes para a aplicação web (replicas, auto-reparação)
2. Sistema de BD distribuído (PostgreSQL com Patroni, MySQL com Galera Cluster, etc.)
3. Armazenamento persistente de HA (ex: Kubernetes Persistent Volumes com backing de NFS HA ou cloud storage)

Pergunta 8: Replicação em Serviços Multi-camada

A complexidade da replicação de um serviço multi-camada não varia de acordo com o componente alvo (ex: servidor web, servidor aplicacional, base de dados) a replicar. Concorda ou não com esta afirmação? Justifique.

Resposta: Discordo completamente. A complexidade varia significativamente consoante o tipo de componente:

Servidor Web (Stateless) - Baixa Complexidade

- Sem estado: cada pedido é independente
- Replicação simples: criar múltiplas instâncias
- Sincronização: não necessária
- Exemplo: Nginx com 10 réplicas é trivial

Servidor Aplicacional (Potencialmente Stateless) - Complexidade Média

- Pode ter cache local ou sessões
- Necessário armazenamento distribuído para estado compartilhado
- Exemplo: Node.js com sessões em Redis

Base de Dados (Stateful) - Muito Alta Complexidade

- Todos os dados devem estar consistentes entre réplicas
- Problemas: escrita distribuída, consenso, particionamento
- Replicação síncrona vs. assíncrona (trade-offs)
- Eleição de líder em caso de falha
- Exemplo: PostgreSQL com replicação é exponencialmente mais complexo que nginx

Conclusão: Replicar é trivial para componentes stateless, mas extremamente complexo para stateful.

Pergunta 9: Elasticidade em Ambientes PaaS

Diga o que entende por elasticidade de um serviço a correr num ambiente de computação em nuvem. Em que medida é que uma arquitetura PaaS contribui para a atingir? Justifique.

Resposta:

Definição de Elasticidade: Capacidade de um sistema aumentar ou diminuir automaticamente os seus recursos (CPU, memória, instâncias) em resposta às flutuações de carga, sem intervenção manual. Objetivo: otimizar desempenho e custos.

Como PaaS contribui para Elasticidade:

1. **Abstração de Infraestrutura**
    - O programador foca-se no código, não na gestão de servidores
    - A plataforma cuida do provisionamento automático
    - Exemplo: Google App Engine – especifica-se apenas o código e a carga é gerada automaticamente
2. **Auto-scaling Automático**
    - PaaS monitora métricas (CPU, tráfego) e ajusta automaticamente
    - Exemplos: App Engine, Heroku
    - Baseado em políticas definidas (e.g., "se CPU > 70%, adicione uma instância")
3. **Billing por Uso Real**
    - Paga-se apenas pelo que se usa
    - Ao escalar up, custos aumentam; ao escalar down, custos diminuem
    - Incentivo económico para elasticidade
4. **Modelos de Dados Distribuídos**
    - PaaS fornece bases de dados escaláveis (ex: Firebase, Amazon DynamoDB)
    - Dados são particionados automaticamente

Diferença com IaaS:

- **IaaS**: Escalabilidade manual (criar VMs manualmente)
- **PaaS**: Elasticidade automática (plataforma decide quando escalar)

---

4. Armazenamento Distribuído

Pergunta 10: Separação de Dados e Metadados

Em vários sistemas de armazenamento distribuídos, a gestão de dados e metadados é feita por componentes diferentes. Explique a vantagem de considerar componentes independentes para gestão de dados e metadados. Justifique.

Resposta:

Definição:

- **Dados**: O conteúdo real dos ficheiros
- **Metadados**: Informações sobre ficheiros (localização, tamanho, permissões, proprietário)

Arquitetura Separada (Ex: HDFS)

- **Metadata Servers**: Gerem o namespace de ficheiros, permissões, localização de blocos
- **Data Nodes**: Armazenam apenas os blocos de dados

Vantagens da Separação:

1. **Escalabilidade Diferenciada**
    - Metadados são pequenos, cabem em memória
    - Dados são volumosos, requerem armazenamento de bloco
    - Exemplo: 1 terabyte de dados é apenas alguns GB de metadados
    - Podem escalar independentemente
2. **Otimização de Recursos**
    - Metadata Servers: Máquinas com muita RAM, CPUs moderadas (acesso rápido)
    - Data Nodes: Máquinas com muita capacidade de armazenamento
    - Não desperdiçar recursos
3. **Gestão de Transações Facilitada**
    - Operações de metadados são atómicas e centralizadas
    - Dados podem ser replicados assincronamente
    - Evita inconsistência (um ficheiro não pode estar em dois locais diferentes)
4. **Tolerância a Falhas**
    - Falha de um Data Node: apenas esse bloco é perdido, metadados intactos
    - Falha de Metadata Server: todo o sistema falha (necessário replicar Metadata Servers)

Exemplo Real (HDFS):

```
Cliente quer ler /user/data/file.txt
1. Contacta Metadata Server: "Onde está file.txt?"
   Resposta: Blocos [1,2,3] estão nos Data Nodes [A, B, C]
2. Contacta Data Nodes A, B, C para obter os blocos
```

Pergunta 11: Replicação vs Erasure Codes em Armazenamento

Compare os mecanismos de replicação e erasure codes para garantir a disponibilidade de dados em sistemas de armazenamento distribuído. Qual é mais apropriado em diferentes contextos?

Resposta:

Aspeto	Replicação	Erasure Codes
Conceito	Cópias exatas dos dados	Dados divididos em fragmentos + paridade
Overhead	3x (exemplo: 3 cópias)	1.5x (exemplo: k=6, m=3)
Tolerância a Falhas	2 falhas (com 3 cópias)	3 falhas (com k=6, m=3)
Latência de Leitura	Imediata	Necessita reconstrução
Latência de Escrita	Síncrona (replicação completa)	Mais rápida

Replicação é Apropriada Para:

- Dados "quentes" (acesso frequente)
- Requisitos de baixa latência
- Exemplo: Cache, base de dados ativa
- Tradeoff: Custo vs. Performance

Erasure Codes são Apropriados Para:

- Dados "frios" (acesso raro, arquivo)
- Grandes volumes onde economia de espaço é crítica
- Exemplo: Amazon Glacier, backups de longa duração
- Tradeoff: Espaço vs. Performance

Decisão Prática:

- HDFS usa replicação para dados primários (performance)
- Glacier usa erasure codes para arquivo (economia)

Pergunta 12: Técnicas de Otimização de Armazenamento

Para um sistema de armazenamento que tem como principais propósitos garantir alta disponibilidade e simultaneamente reduzir o espaço ocupado pelos dados persistidos, quais funcionalidades sugeria? Justifique.

Resposta:

Combinação Recomendada:

1. **Erasure Codes (em vez de Replicação)**
    - Reduz overhead de 3x para 1.5x
    - Mantém tolerância a 3+ falhas simultâneas
    - Justificação: Melhor relação espaço/disponibilidade
2. **Compressão**
    - Reduz tamanho dos dados em disco
    - Tipos: GZIP (bom), SNAPPY (rápido), LZMA (máxima compressão)
    - Tradeoff: CPU vs. Espaço

- Justificação: Redução imediata de espaço

3. **Deduplicação**
   - Elimina cópias redundantes de dados entre ficheiros
   - Exemplo: Se 1000 utilizadores têm a mesma foto, apenas 1 cópia no disco
   - Tradeoff: CPU para detecção vs. Espaço poupado
   - Justificação: Muito efetivo para dados altamente redundantes

4. **Tiering (Camadas)**
   - Dados quentes: Replicação em SSD (rápido)
   - Dados mornos: Erasure codes em HDD (equilibrado)
   - Dados frios: Arquivo comprimido em tape (barato)
   - Justificação: Otimizar custo-benefício conforme padrão de acesso

Solução Completa Recomendada:

```
Alta Disponibilidade + Eficiência Espaço:
→ Erasure Codes (k=8, m=4) para tolerância a falhas
+ Compressão SNAPPY (bom balanço CPU/espaço)
+ Deduplicação para ficheiros comuns
```

---

5. Provisioning e Automação

Pergunta 13: Diferença entre Provisionamento e Deployment

Defina o que entende por provisioning (provisionamento) e deployment de uma aplicação. Qual é o objetivo de cada um?

Resposta:

Provisionamento (Provisioning):

- **Definição**: A ação de fornecer ou disponibilizar recursos de infraestrutura para utilização
- **Escopo**: Servidores, armazenamento, rede, VMs, utilizadores
- **Objetivo**: Preparar a infraestrutura (hardware, SO, dependências) antes de instalar a aplicação
- **Exemplo**: Criar uma VM, instalar Python, instalar PostgreSQL, configurar firewall
- **Responsabilidade**: Tipicamente das operações (Ops)

Deployment:

- **Definição**: O processo de instalar ou atualizar uma aplicação/serviço num servidor
- **Escopo**: Código da aplicação, configuração específica da app
- **Objetivo**: Colocar o código em execução num servidor já provisionado
- **Exemplo**: Fazer `git clone`, correr migrações BD, iniciar servidor web
- **Responsabilidade**: Tipicamente do desenvolvimento (Dev)

Relação: Provisionamento → Deployment

1. Primeiro, provisiona-se a infraestrutura
2. Depois, faz-se deploy da aplicação

Na Prática DevOps: Ambos estão automatizados em código (IaC - Infrastructure as Code), unindo Dev + Ops.

Pergunta 14: Vantagens da Automação com Ansible

No guião das aulas práticas, foram criados diferentes roles para facilitar a instalação automática da aplicação Swap usando Ansible. Qual é a função dos roles num playbook Ansible? De que forma os roles contribuem para um melhor aprovisionamento de aplicações? Justifique.

Resposta:

Função dos Roles: Um role é um componente reutilizável que encapsula todas as configurações e tarefas para uma funcionalidade específica.

Estrutura de um Role:

```
roles/
└── nginx/
    ├── tasks/
    │   └── main.yml        (tarefas: instalar, configurar)
    ├── handlers/
    │   └── main.yml        (reiniciar serviço se mudou config)
    ├── templates/
    │   └── nginx.conf.j2   (ficheiro de config dinâmico)
    ├── files/
    │   └── default.html    (ficheiros estáticos)
    └── vars/
        └── main.yml        (variáveis do role)
```

Contribuições para Melhor Aprovisionamento:

1. **Reutilização e Modularidade**
   - Role `nginx` pode ser usado em múltiplos playbooks
   - Não duplicar código
   - Exemplo: Instalar Nginx em 3 projetos diferentes

2. **Organização Clara**
   - Separação de preocupações (tasks, handlers, templates)
   - Fácil de compreender e manter
   - Documentação natural da infraestrutura

3. **Composição Simples**

```
hosts: webservers
roles:
  - common          # Configuração base (SO, packages)
  - users           # Criar utilizadores
  - ssh-server      # Configurar SSH
  - nginx           # Instalar e configurar Nginx
```

- Ler este playbook é trivial
- Compreende-se exatamente o que é instalado

4. **Consistência e Repetibilidade**
   - Mesmo role garante sempre o mesmo resultado
   - Idempotência: Correr 2x tem o mesmo efeito que correr 1x
   - Infraestrutura previsível
5. **Escalabilidade**
   - Adicionar novos servidores é trivial: aplicar os mesmos roles
   - Exemplo prático no Swap: Role garante que nova instância é idêntica

---

Pergunta 15: Paradigma Imperativo vs Declarativo

Compare a abordagem imperativa (shell scripts tradicional) com a abordagem declarativa (Ansible playbooks). Quais são as implicações para a gestão de configuração em infraestruturas complexas?

Resposta:

Abordagem Imperativa (Shell Script):

```
#!/bin/sh
apt-get update
apt-get install -y nginx
cp myconfig.conf /etc/nginx/
systemctl restart nginx
```

- Define **como** fazer algo (passo a passo)
- Cada linha é um comando a executar

Problemas:

- **Não Idempotente**: Correr 2x pode falhar (ex: se package já instalado)
- **Frágil**: Se script falha a meio, estado fica inconsistente
- **Difícil Diagnosticar**: Qual passo falhou?

Abordagem Declarativa (Ansible):

```yaml
hosts: webservers
tasks:
  - name: Install Nginx
    apt:
      name: nginx
      state: present
  - name: Copy config
    template:
      src: nginx.conf.j2
      dest: /etc/nginx/nginx.conf
    notify: restart nginx

  handlers:
    - name: restart nginx
      service:
        name: nginx
        state: restarted
```

- Define **o que** queremos (estado final desejado)
- Ansible garante que o sistema atinge esse estado

Vantagens:

- **Idempotente**: Correr 2x tem o mesmo resultado
- **Robusto**: Apenas efetua mudanças se necessário
- **Diagnostico Claro**: Sabe-se exatamente qual task falhou

Implicações em Infraestruturas Complexas:

1. **Drift Configuracional**
    - Imperativo: Se alguém modifica manualmente um ficheiro, o script não consegue repor
    - Declarativo: Próxima execução repõe à configuração desejada
2. **Recuperação de Falhas**
    - Imperativo: Necessário refazer tudo
    - Declarativo: Reexecutar o playbook corrige
3. **Documentação Viva**
    - Imperativo: Código é preto-branco, pouca documentação
    - Declarativo: YAML é legível por humanos, descreve o estado desejado
4. **Versioning**
    - Imperativo: Scripts mudam, difícil rastrear versions
    - Declarativo: Playbooks em Git, histórico completo de mudanças

---

6. Monitorização e Benchmarking

Pergunta 16: A Ilusão das Médias em Monitorização

A utilização incorreta de métricas de desempenho, especialmente focando apenas em médias, pode levar a conclusões erróneas sobre a qualidade de um serviço. Concorda? Justifique com exemplos de padrões problemáticos que as médias escondem.

Resposta: Concordo completamente. As médias são enganadoras e escondem problemas críticos.

Exemplo Crítico: Três servidores web têm estas latências (em ms):

- Servidor A: [50, 50, 50, 50, 50] → Média = 50ms
- Servidor B: [10, 10, 10, 10, 200] → Média = 48ms
- Servidor C: [1, 1, 1, 1, 1] → Média = 1ms

A olhar apenas às médias, Servidor B parece melhor que A! Mas um em cada 5 pedidos é extremamente lento.

Padrões Problemáticos que as Médias Escondem:

1. **Distribuição de Cauda Longa (Long Tail)**
    - 99% dos pedidos são rápidos (20ms)
    - 1% dos pedidos são muito lentos (5000ms)
    - Média: ~70ms (enganadora)
    - **Problema Real**: Utilizadores experimentam ocasionalmente despensa gritante
    - **Causa**: Pausas de Garbage Collection, cache misses, timeouts de rede
2. **Degradação ao Longo do Tempo**
    - Primeiros 1000 pedidos: 50ms cada
    - Próximos 1000 pedidos: 100ms cada
    - Próximos 1000 pedidos: 500ms cada
    - Média: 216ms
    - **Problema Real**: Sistema degrada ao longo do tempo
    - **Causa**: Memory leak, acumulação de conexões, etc.
3. **Bimodalidade (Dois Modos)**
    - Cache hit: 5ms
    - Cache miss: 200ms
    - Média: 100ms
    - **Problema Real**: Comportamento completamente diferente consoante cache
    - **Solução**: Entender a distribuição e otimizar taxa de hits

Métricas Corretas a Usar:

1. **Percentis (P50, P95, P99)**
    - P99: Tempo para 99% dos pedidos
    - Mais representativo que média
2. **Desvio Padrão / Variância**

- Mostra a dispersão dos dados

3. **Visualizações**
    - ECDF (Empirical Cumulative Distribution Function)
    - Histogramas

Pergunta 17: Compromissos no Design de Monitores

A forma como um monitor é desenhado implica compromissos (trade-offs) importantes. Discuta o compromisso entre "observação orientada a eventos" vs "amostragem" na monitorização de um sistema. Qual escolher em diferentes contextos?

Resposta:

Observação Orientada a Eventos (Event-driven):

- A observação é despoletada sempre que um evento de interesse ocorre
- Captura todos os eventos relevantes

Amostragem (Sampling):

- A observação ocorre apenas periodicamente (ex: a cada 10ms)
- Captura uma amostra dos eventos

Compromisso: Precisão vs. Overhead de Performance

Aspeto	Orientada a Eventos	Amostragem
Precisão	100% (nenhum evento perdido)	Parcial (alguns eventos perdidos)
Overhead	Muito alto (monitor é chamado constantemente)	Baixo (apenas a intervalos)
Impacto Sistema	Degradação significativa do desempenho	Negligenciável

Exemplo Prático: Monitorizar operações de disco:

- **Event-driven**: Cada read/write dispara uma observação
    - Em alta carga: 100.000 operações/s → 100.000 observações/s
    - Overhead: 50% da CPU pode ser gasto em monitoring!
    - Problema: O próprio monitor faz o sistema mais lento
- **Amostragem**: A cada 1ms, sample 10 operações
    - Mesmo em alta carga: 1.000 observações/s
    - Overhead: <1% da CPU
    - Desvantagem: Pode-se perder picos raros (ex: uma operação de 10s)

Contextos Apropriados:

1. **Usar Event-driven:**
    - Sistemas críticos com carga baixa-média
    - Necessidade de detetar anomalias raras
    - Exemplo: Monitorizar falhas de segurança
    - Overhead aceitável
2. **Usar Amostragem:**
    - Sistemas de alta performance
    - Análise de tendências (não necessita precisão 100%)
    - Exemplo: Monitorizar latência de aplicação web em produção

Solução Prática Comum: Implementar ambos:

- **Event-driven** para eventos críticos (erros, falhas)
- **Amostragem** para métricas contínuas (CPU, memória)

Pergunta 18: Componentes de um Ciclo de Monitorização Completo

Descreva os quatro componentes essenciais de uma arquitetura de monitorização moderna. Explique como cada um contribui para a transformação de "ruído" (eventos brutos) em "sinal" (informação acionável).

Resposta:

Os Quatro Componentes (Bottom-up):

1. **Observação**
    - Recolhe eventos brutos do sistema
    - Técnicas: Observação passiva (sniffing), instrumentação, sondagem
    - **Problema**: Gera milhões de eventos/s, inúteis por si só
2. **Recolha**
    - Agrega e normaliza os eventos observados
    - Sincroniza timestamps de múltiplos sistemas
    - Oferece armazenamento temporário
    - Modelos: Push (eventos enviados ativamente) ou Pull (recolhidos periodicamente)
    - **Problema**: Ainda são dados brutos
3. **Análise**
    - Transforma dados brutos em informação
    - Armazenamento eficiente (indexação)
    - Processamento: Filtrar, consultar, sumarizar
    - Tecnologias: Elasticsearch, Prometheus, InfluxDB
    - **Resultado**: Informação estruturada

4. **Apresentação**
    - Torna a informação visível e compreensível
    - Formatos: Dashboards (tempo real), Relatórios, Alarmes
    - Ferramentas: Kibana (com Elasticsearch), Grafana
    - **Resultado**: Insights acionáveis

Transformação Ruído → Sinal (Exemplo Prático):

```
Ruído (Observação):
→ 1000 log entries por segundo
→ "User login attempt from 192.168.1.1"
→ "User login attempt from 192.168.1.1"
→ "User login attempt from 192.168.1.1"
(Padrão invisível)

↓ Recolha
(Normalização de timestamps)

↓ Análise
(Agregação): "192.168.1.1 teve 100 tentativas de login em 1 segundo"

↓ Apresentação
Sinal (Insight Acionável):
→ Alert: "Possível ataque brute-force de 192.168.1.1"
→ Ação: Bloquear IP
```

Implementação Prática (Elastic Stack):

- **Observação**: Beats (Filebeat, Metricbeat) recolhem eventos
- **Recolha**: Logstash normaliza e enriquece
- **Análise**: Elasticsearch indexa e armazena
- **Apresentação**: Kibana visualiza em dashboards

---

7. Arquiteturas Distribuídas

Pergunta 19: Padrões Fundamentais de Distribuição

Uma arquitetura distribuída pode implementar três padrões fundamentais para lidar com escalabilidade e fiabilidade. Indique e explique cada um, dando exemplos de quando cada é apropriado.

Resposta:

Padrão 1: Replicação

- **Conceito**: Múltiplas cópias idênticas da mesma funcionalidade/dados
- **Objetivo**: Disponibilidade e escalabilidade de leitura
- **Como Funciona**: Servidor Master sincroniza com Slaves
- **Exemplos**: RAID, Database replication (MySQL, PostgreSQL)

- **Quando Usar**:
  - Dados são lidos frequentemente, escritos raramente
  - Necessária baixa latência (cópias locais)
  - Exemplo: Catálogo de produtos de um e-commerce

Padrão 2: Particionamento (Sharding)

- **Conceito**: Dados/funcionalidade divididos por vários servidores
- **Objetivo**: Escalabilidade de escrita e redução de dados por servidor
- **Dois Tipos**:
  1. Horizontal (Sharding): Dividir por cliente ou por range de dados
     - Servidor A: Dados clientes A-F
     - Servidor B: Dados clientes G-M
  2. Vertical (Service-oriented): Dividir por funcionalidade
     - Servidor A: Armazenamento
     - Servidor B: Base de dados
     - Servidor C: Aplicação
- **Exemplos**: MongoDB sharding, Elasticsearch sharding
- **Quando Usar**:
  - Dados muito grandes para um servidor
  - Necessária escalabilidade de escrita
  - Exemplo: Netflix com millions de utilizadores

Padrão 3: Orientação a Serviços

- **Conceito**: Sistema dividido em múltiplos serviços independentes
- **Evolução**: Monólito → Divisão vertical → Microserviços
- **Benefícios**: Escalabilidade independente, desenvolvimento paralelo, tolerância a falhas
- **Exemplos**: Microserviços (cada um gerido por equipa diferente)
- **Quando Usar**:
  - Grande organização com múltiplos teams
  - Diferentes partes do sistema têm requisitos distintos
  - Exemplo: Netflix com serviços de Payment, Streaming, Recommendation, etc.

Combinação Comum:

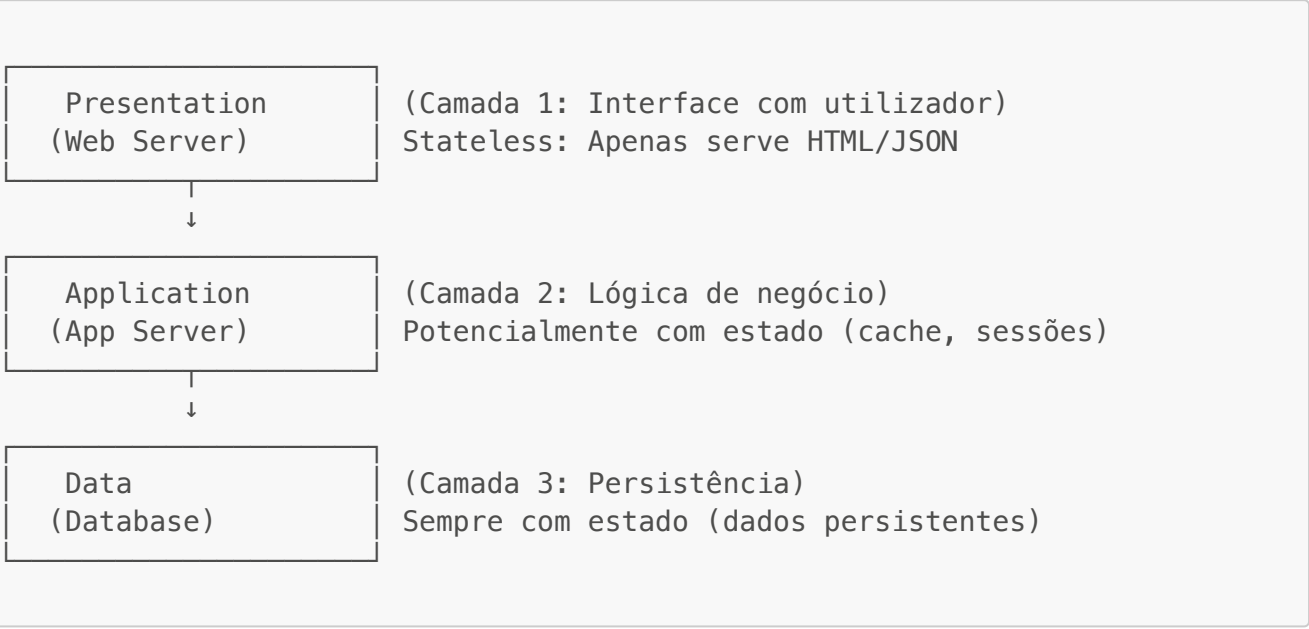
```
Aplicação Web
  ├─ Frontend (Stateless) → Replicado (10 instâncias)
  ├─ Backend Applicação (Stateless) → Replicado (5 instâncias)
  ├─ Base de Dados (Stateful) → Particionada (4 shards) + cada shard
replicado
  └─ Cache (Redis) → Replicado + Particionado
```

Pergunta 20: Arquitetura Multi-tier e Gestão de Estado

Explique a arquitetura multi-tier (3 camadas ou mais) e discuta o desafio do estado nesta arquitetura. Por que é que é mais fácil replicar computação stateless do que estado persistente?

Resposta:

Arquitetura Multi-tier Tradicional (3 Camadas):



Gestão de Estado em Cada Camada:

1. **Camada de Apresentação (Stateless)**
   - Não armazena estado
   - Cada pedido é independente
   - Replicação: Trivial (ex: 10 servidores nginx)
   - Sincronização: Não necessária
2. **Camada de Aplicação (Estado Transitório)**
   - Pode ter cache local ou sessões de utilizador
   - Replicação: Complexa
   - Solução: Armazenar sessões em Redis (camada compartilhada)
3. **Camada de Dados (Estado Persistente)**
   - Dados críticos, devem estar corretos
   - Replicação: Muito complexa

Por que é Mais Fácil Replicar Computação Stateless?

Computação Stateless (Web Server):

```
Pedido 1: GET /api/products → responde com lista de produtos
Pedido 2: GET /api/products → responde com MESMA lista
```

```
Resultado: Idêntico, não importa em qual servidor foi processado
```

- Não há memória de pedidos anteriores
- Duas cópias processam idêntico → Síncrono não necessário
- Load balancer pode distribuir aleatoriamente

Estado Persistente (Database):

```
Escrita 1: UPDATE products SET price = 100 WHERE id = 1
Escrita 2: UPDATE products SET price = 120 WHERE id = 1

Resultado: Se replicado para 3 servidores simultaneamente:
- Servidor A: price = 100
- Servidor B: price = 100
- Servidor C: price = 100
...depois
- Servidor A: price = 120
- Servidor B: price = 120
- Servidor C: price = 120

Problema: O que se um deles falha a meio? Dados inconsistentes!
```

Desafios de Replicação Stateful:

1. **Consistência Distribuída**
    - Necessário garantir que todas as réplicas veem mesmos dados
    - Teorema CAP: Não se pode ter Consistência + Disponibilidade + Tolerância a Partições
2. **Ordenação de Operações**
    - 100 clientes escrevem simultaneamente
    - Qual é a ordem correta?
    - Necessário consenso distribuído (Paxos, Raft)
3. **Conflito de Escrita**
    - Replicação 1: Cliente A modifica record X em Servidor 1
    - Replicação 2: Cliente B modifica record X em Servidor 2
    - Servidor 1 e Servidor 2 têm valores diferentes!
    - Qual é o correto? (Impossível sem comunicação)

Resumo:

- **Stateless**: Copiam, processam, esquecem → Trivial replicar
- **Stateful**: Devem estar sincronizados → Exponencialmente mais complexo

---

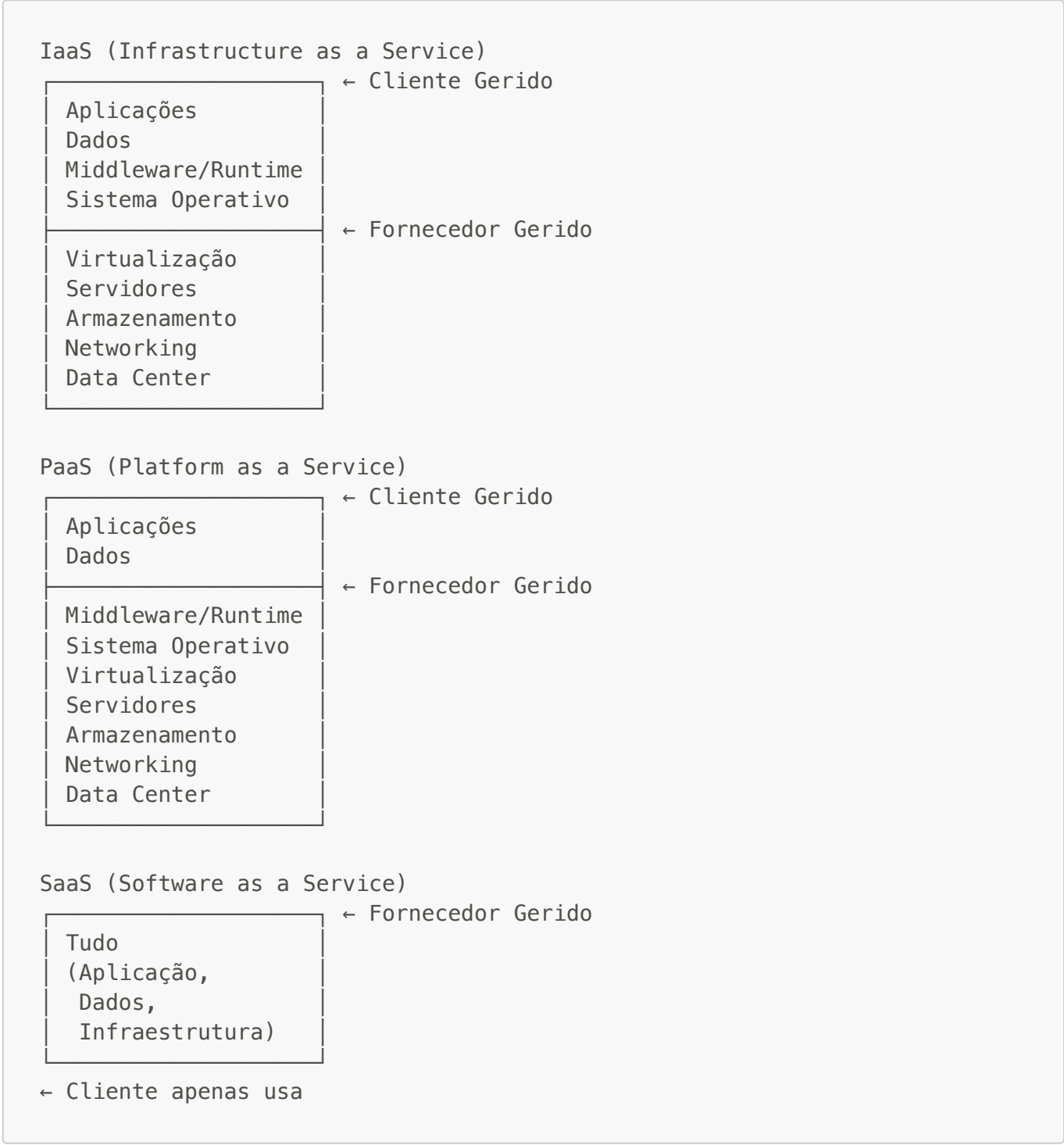
8. Modelos de Serviço Cloud

Pergunta 21: Partilha de Responsabilidades IaaS vs PaaS vs SaaS

Explique o modelo de partilha de responsabilidades entre cliente e fornecedor em cada um dos três modelos de serviço cloud (IaaS, PaaS, SaaS). Dê exemplos concretos.

Resposta:

Responsabilidades em Cada Modelo:



Exemplos Concretos:

IaaS: Amazon EC2

- **Cliente gerido**: Escolher SO (Linux/Windows), instalar aplicação, gerir backups

- **Fornecedor gerido**: Hardware, rede física, data center
- **Exemplo prático**: Aluga-se uma VM, instala-se Python + Django, faz-se deploy de aplicação
- **Controlo**: Máximo
- **Complexidade**: Máxima

PaaS: Google App Engine

- **Cliente gerido**: Escrever código Python/Java/Node.js
- **Fornecedor gerido**: Escalabilidade automática, BD gerida, deployment automático
- **Exemplo prático**: Upload de código, App Engine escala automaticamente conforme tráfego
- **Controlo**: Médio
- **Simplicidade**: Média

SaaS: Google Workspace (Gmail, Docs)

- **Cliente gerido**: Apenas usar o serviço
- **Fornecedor gerido**: Tudo (servidores, segurança, backups, updates)
- **Exemplo prático**: Aceder a Gmail.com, enviar emails
- **Controlo**: Mínimo
- **Facilidade**: Máxima

Implicações Económicas e Operacionais:

Modelo	Custo Inicial	Custo Operacional	Escalabilidade	Controlo
IaaS	Médio	Alto (hiring DevOps)	Manual	Total
PaaS	Baixo	Médio	Automática	Parcial
SaaS	Nenhum	Baixo	Fornecedor	Nenhum

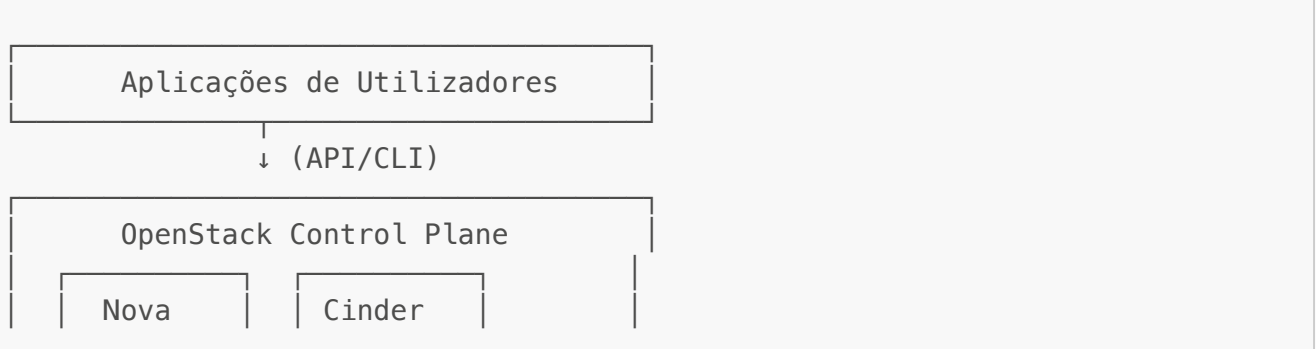
Pergunta 22: OpenStack como Solução de IaaS Privada

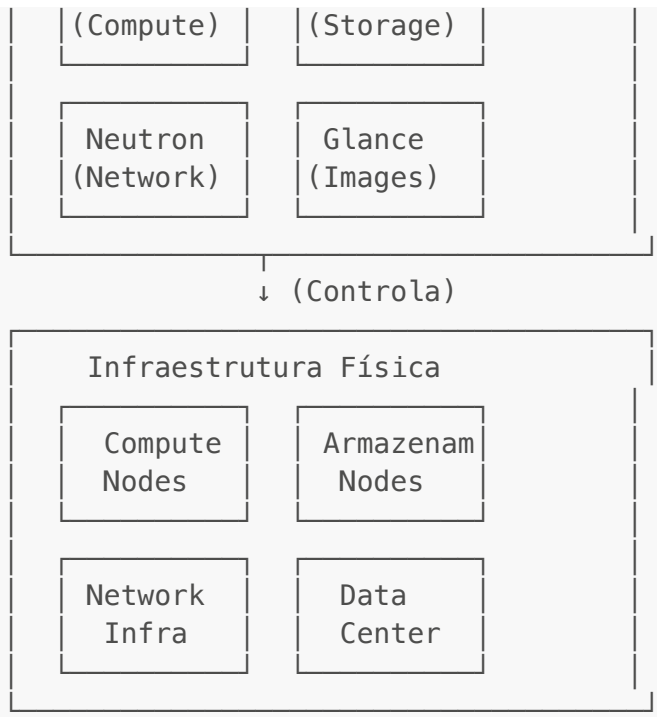
Descreva o OpenStack como exemplo de implementação de uma nuvem IaaS privada. Qual é a vantagem de uma nuvem privada relativamente a fornecedores públicos como AWS?

Resposta:

O que é OpenStack: OpenStack é um software open-source que permite criar uma nuvem IaaS privada, controlando totalmente a infraestrutura de um data center sem depender de fornecedores externos.

Arquitetura Simplificada:





Componentes Principais:

1. **Nova (Compute)**: Gerencia instâncias de computação (VMs, contentores, bare metal)
2. **Cinder (Block Storage)**: Volumes persistentes para VMs
3. **Swift (Object Storage)**: Armazenamento de objetos (similar a S3)
4. **Neutron (Networking)**: Gestão de redes virtuais
5. **Glance (Images)**: Repositório de imagens de VM

Vantagens da Nuvem Privada (OpenStack) vs Pública (AWS):

Aspeto	Nuvem Privada (OpenStack)	Nuvem Pública (AWS)
Controlo	Total (dentro da organização)	Nenhum (AWS controla)
Dados Sensíveis	Dentro de infraestrutura própria	Em servidores AWS (overseas?)
Conformidade	Mais fácil cumprir regulações (GDPR, HIPAA)	Complexo (dados fora de controlo)
Custo Inicial	Alto (investimento em hardware)	Baixo (pay-per-use)
Custo Operacional	Alto (necessário DevOps)	Baixo (AWS gerencia)
Escalabilidade	Limitada ao hardware disponível	Praticamente ilimitada
Latência	Potencialmente menor (infraestrutura local)	Depende da localização
Vendor Lock-in	Nenhum (open-source)	Forte (AWS proprietário)

Quando Usar OpenStack:

- Organizações com dados muito sensíveis (financeiros, saúde)
- Conformidade regulatória rigorosa

- Grandes volumes de dados (evitar custos de transferência)
- Necessidade de controlo total
- Já possuem data center

Quando Usar AWS:

- Startups com orçamento limitado
- Escalabilidade dinâmica imprescindível
- Não há expertise DevOps interno
- Aplicações globais (multi-região)

---

9. DevOps e Infraestrutura como Código

Pergunta 23: DevOps como Síntese de Desenvolvimento e Operações

Defina o que entende por DevOps e explique como a Infraestrutura como Código (IaC) é fundamental para alcançar os objetivos de DevOps. Dê um exemplo prático.

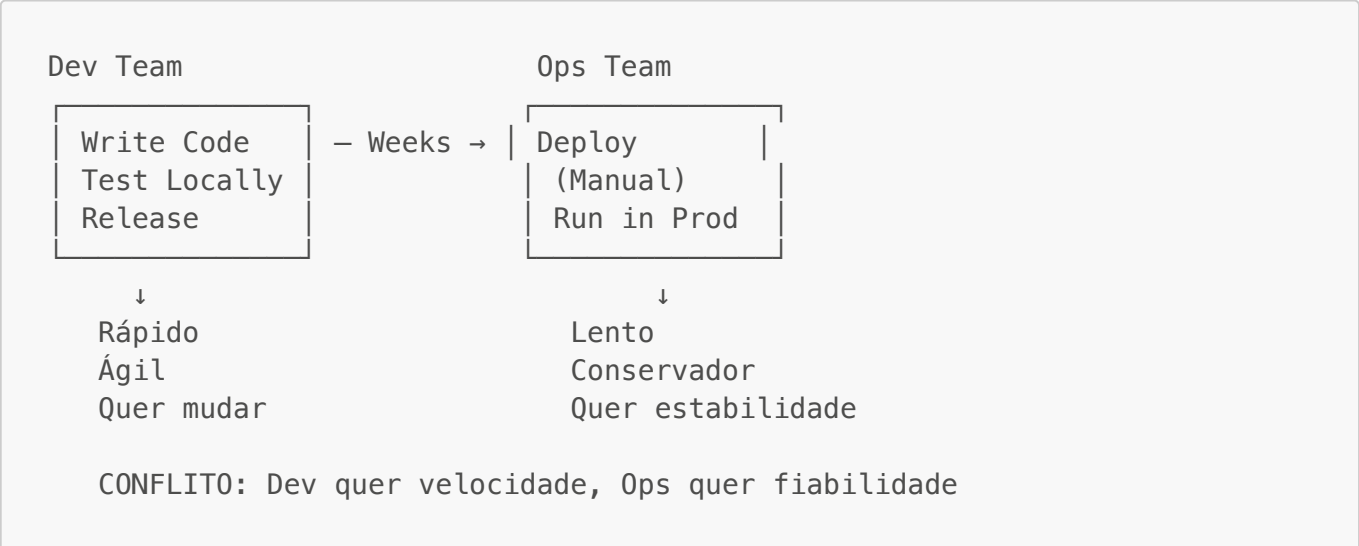
Resposta:

DevOps: Mais que Uma Ferramenta, Uma Filosofia

DevOps é a síntese cultural e técnica que une Desenvolvimento (Dev) e Operações (Ops), rompendo os silos tradicionais entre estas equipas.

Objetivo Central de DevOps: Entregar valor aos utilizadores de forma mais rápida e fiável, suportando mudanças pequenas, rápidas e frequentes.

Antes de DevOps (Modelo Tradicional):



Com DevOps (Modelo Integrado):

```
  │   │ Code      │──│ como Código   │   │
  │   └──────────┘   └──────────────┘   │
  │                                      │
  │   ┌──────────┐   ┌──────────────┐   │
  │   │ Test &   │   │ CI/CD        │   │
  │   │ Integrate│──│ Pipeline     │   │
  │   └──────────┘   └──────────────┘   │
  │                                      │
  │   ┌──────────┐   ┌──────────────┐   │
  │   │ Deploy   │   │ Monitorização│   │
  │   │ & Run    │──│ & Feedback   │   │
  │   └──────────┘   └──────────────┘   │
  │                                      │
  └──────────────────────────────────────┘
     Rápido, Fiável, Iterativo
```

Infraestrutura como Código (IaC): A Ponte Técnica

Antes de IaC, provisionamento era manual e ad-hoc:

```
Ops Engineer 1: "Criar um servidor Ubuntu com Nginx"
Ops Engineer 2: Pede instruções verbalmente
Resultado: Dois servidores ligeiramente diferentes (configuration drift)
```

Com IaC, infraestrutura é definida em código versionado:

```
# Terraform (IaC)
resource "aws_instance" "web_server" {
  ami           = "ami-ubuntu-20.04"
  instance_type = "t2.micro"

  provisioner "remote-exec" {
    inline = [
      "sudo apt update",
      "sudo apt install -y nginx",
      "sudo systemctl start nginx"
    ]
  }
}
```

Benefícios de IaC para DevOps:

1. **Reprodutibilidade**
    - Mesmo código = Mesmo resultado
    - Dev e Prod são idênticos
    - Fim do "mas funciona na minha máquina"
2. **Versionamento**
    - Infraestrutura está em Git

- Histórico completo de mudanças
- Possível fazer rollback se algo falha

3. **Automatização Completa**
   - `git push` → CI/CD pipeline
   - Testes automáticos
   - Deploy automático para staging/prod
4. **Documentação Viva**
   - O código é a documentação
   - Ler o IaC código é ver exatamente qual é a infraestrutura

Exemplo Prático Completo:

Cenário: Startup com aplicação Node.js precisa escalar de 1 para 10 servidores

Sem DevOps/IaC (Tradicional):

1. Engenheiro criar manualmente 9 novos servidores
2. Instalar Node.js, npm, dependências (pode falhar, variações)
3. Configurar load balancer (manual)
4. Documentação? Não existe...
5. Tempo: 3-5 dias
6. Taxa de erro: Alta

Com DevOps/IaC:

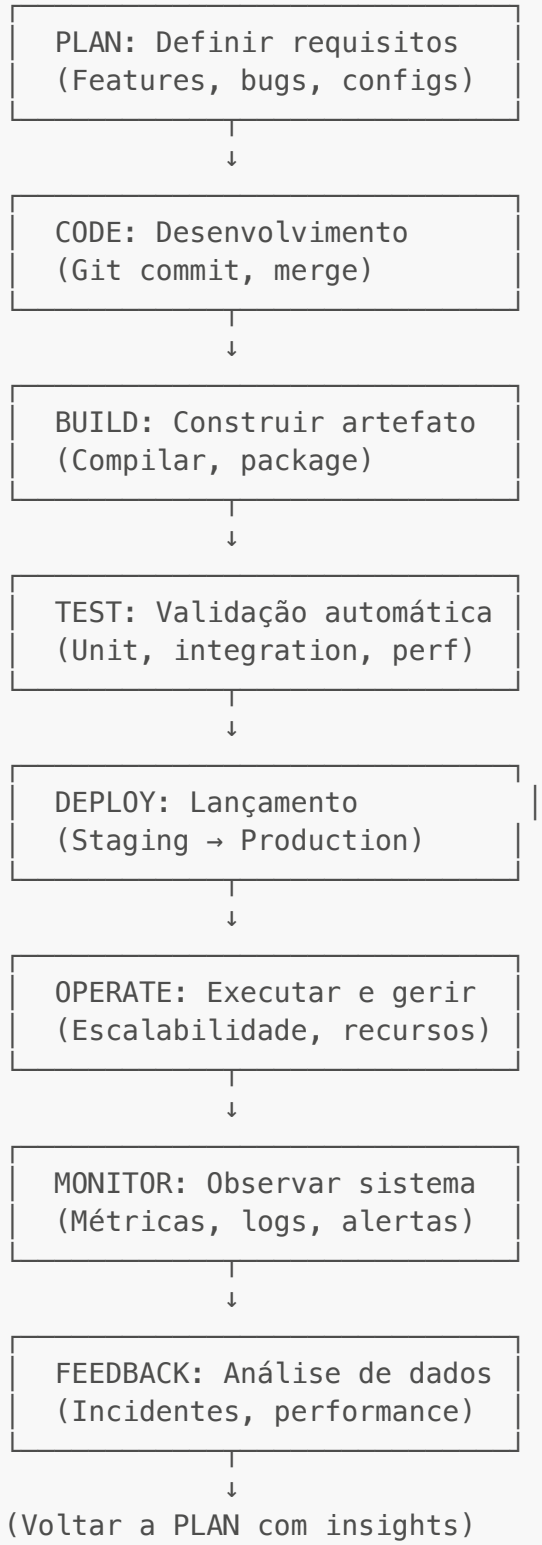
1. Engenheiro modifica código IaC: `instance_count = 10`
2. Faz `git push`
3. CI/CD pipeline:
   - Valida código IaC
   - Cria 9 novos servidores
   - Instala software (automaticamente)
   - Testa connectivity
   - Atualiza load balancer (automaticamente)
4. Monitoring detecta status de todos
5. Tempo: 5-10 minutos
6. Taxa de erro: Praticamente zero
7. Infraestrutura documentada em Git (histórico completo)

Pergunta 24: Ciclo de Feedback em DevOps

Descreva o ciclo iterativo de DevOps (Plan → Code → Build → Test → Deploy → Operate → Monitor → Feedback → Plan...). Como é que cada fase contribui para alcançar maior velocidade e fiabilidade?

Resposta:

O Ciclo Contínuo DevOps:



Cada Fase Contribui Para:

PLAN:

- **Objetivo**: Definir claramente o que se vai fazer
- **Benefício de Velocidade**: Evitar retrabalho
- **Benefício de Fiabilidade**: Compreender requisitos reduz bugs

CODE:

- **Objetivo**: Implementar a feature/fix
- **Benefício de Velocidade**: Developers trabalham em paralelo em branches
- **Benefício de Fiabilidade**: Code review, pair programming

BUILD:

- **Objetivo**: Criar artefato executável (ex: Docker image)
- **Benefício de Velocidade**: Automático, <5 minutos
- **Benefício de Fiabilidade**: Mesmo artefato em todos os ambientes (Dev/Staging/Prod)

TEST:

- **Objetivo**: Validar antes de ir à produção
- **Benefício de Velocidade**: Feedback automático em minutos
- **Benefício de Fiabilidade**: Testes unitários, integração, performance previnem bugs

DEPLOY:

- **Objetivo**: Colocar em produção
- **Benefício de Velocidade**: Automático (blue-green deploy, canary deploy)
- **Benefício de Fiabilidade**: Rollback automático se algo falha

OPERATE:

- **Objetivo**: Manter sistema em produção
- **Benefício de Velocidade**: Auto-scaling, orquestração automática
- **Benefício de Fiabilidade**: Infraestrutura como código garante consistência

MONITOR:

- **Objetivo**: Observar comportamento em tempo real
- **Benefício de Velocidade**: Detetar problemas rápido
- **Benefício de Fiabilidade**: Alertas automáticos, correlação de eventos

FEEDBACK:

- **Objetivo**: Aprender com dados reais de produção
- **Benefício de Velocidade**: Priorizir próximas features baseado em dados
- **Benefício de Fiabilidade**: Incidentes informam melhorias arquiteturais

Exemplo Prático: Deploy de Feature em 1 Hora

```
10:00 – PLAN: Product owner define: "Botão 'Like' em posts"
10:05 – CODE: Dev cria branch, implementa feature
10:15 – BUILD: CI/CD constrói Docker image (automático)
10:18 – TEST: Testes unitários & integração passam (automático)
10:20 – DEPLOY: Deploy em staging para QA testar
10:30 – OPERATE: Escalabilidade avaliada (Kubernetes auto-scales)
10:35 – MONITOR: Métricas baseline capturadas
10:40 – DEPLOY: Deploy em produção (blue-green deploy)
10:45 – MONITOR: Dashboard mostra performance em tempo real
10:55 – FEEDBACK: 50 likes em 10 minutos! Feature é sucesso
```

```
(Sem DevOps, isto levaria 1–2 semanas)
```

---

10. Benchmarking e Performance

Pergunta 25: Latência vs Throughput - O Compromisso Fundamental

Explique a relação entre latência e throughput sob carga crescente. Por que é que aumentar throughput não diminui necessariamente latência, e vice-versa? Dê um exemplo de otimização que melhora throughput mas piora latência.

Resposta:

Definições Básicas:

- **Latência (L)**: Tempo entre pedido e resposta (ex: 50ms)
- **Throughput (T)**: Número de pedidos processados por segundo (ex: 1000 req/s)

Relação Intuitiva Errada: Muitos pensam: L = 1 / T (relação inversa simples)

- Se T = 1000 req/s, então L = 1ms?
- Isto é **FALSO**

Relação Real (Verdadeira) sob Carga Crescente:

Existem 3 fases:

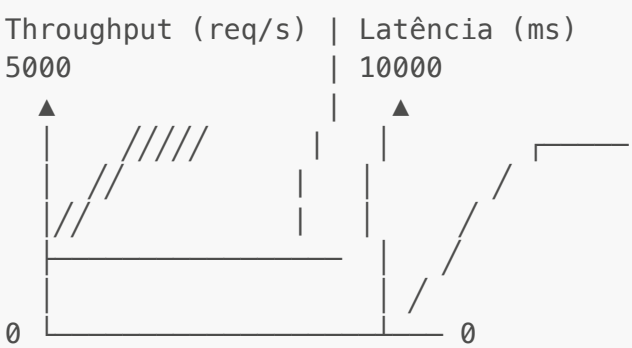
```
Fase 1: Idle/Vazio (Carga Baixa)
┌──────────────────────────────────┐
│ Throughput: Cresce linearmente   │
│ Latência: Permanece baixa (~50ms)
│ Razão: Sistema tem capacidade    │
│        de sobra, sem contenção   │
└──────────────────────────────────┘
   0 → 100 utilizadores

Fase 2: Near Capacity (Carga Média)
┌──────────────────────────────────┐
│ Throughput: Aproxima-se do máximo
│ Latência: Começa a subir         │
│ Razão: Filas começam a formar    │
│        (Little's Law)            │
└──────────────────────────────────┘
   100 → 500 utilizadores

Fase 3: Overload (Carga Alta)
┌──────────────────────────────────┐
│ Throughput: Estagna ou cai       │
│ Latência: Dispara exponencialmente
│ Razão: Sistema saturado, muitas  │
```

```
│        filas, contenção CPU     │
└──────────────────────────────────┘
   500+ utilizadores
```

Gráfico Real:



Little's Law (Lei de Little):

```
Latência = (Tamanho Fila + Tempo Serviço) / Throughput

À medida que carga sobe, fila cresce.
Fila cresce exponencialmente em overload.
Portanto, Latência cresce exponencialmente.
```

Exemplo Prático: Batching

Otimização: Agrupar 100 pedidos e processar como 1 batch

Sem Batching:

- 100 pedidos individuais: 100 × 10ms = 1000ms total
- Throughput: 100 req/s
- Latência individual: 10ms

Com Batching:

- 1 batch de 100: Processamento paralelo = 50ms
- Throughput: 2000 req/s (20x melhor!)
- Latência individual: 50ms (5x pior!)

```
      Throughput (req/s)  │  Latência (ms)
                          │
```

```
Sem Batch: 100          │  Sem Batch: 10
Com Batch: 2000         │  Com Batch: 50

✓ 20x melhor throughput │  ✗ 5x pior latência
✗ Pior para clients     │  ✓ Melhor para sistema
individuais             │
```

Trade-offs Reais:

Otimização	Throughput	Latência	Quando Usar
Batching	↑↑	↓↓	Sistemas de batch (MapReduce)
Caching	↑	↓	Dados repetidos
Compressão	↓	↑	Banda limitada
Multithreading	↑	↓	I/O bound
Lazy evaluation	↑	↓	Dados não necessários

Conclusão: Latência e Throughput **não são inversamente proporcionais**. Sob carga:

- Fase 1: Ambas melhoram
- Fase 2: Compromisso (throughput sobe, latência sobe lentamente)
- Fase 3: Tradeoff direto (aumentar throughput piora latência)

A melhor estratégia é **permanecer na Fase 1-2** (antes de saturação) através de:

- Auto-scaling (adicionar servidores)
- Otimização (cache, batching consoante requisito)
- Monitorização (detetar antes de atingir Fase 3)

FIM DO DOCUMENTO

Total de Perguntas e Respostas: 25 Tópicos Cobertos: 10 áreas principais Alinhamento com Testes: 2022-2024