Resumo do PDF: EAI – Enterprise Application Integration

✅ Conceito de EAI

Enterprise Application Integration (EAI) é a prática de integrar sistemas empresariais distintos para garantir:

- Pessoas certas
- Processos certos
- Tecnologia certa
- Dados certos
- No momento certo
- Com protocolos de segurança adequados
- Usando recursos certos

Objetivos: **Conectar**, **Combinar** e **Otimizar** sistemas e dados.

❓ Porquê utilizar EAI?

- Evitar problemas de integração ad hoc e ponto-a-ponto.
- Adaptação rápida a ambientes de negócio em constante mudança (ex: microserviços).
- Aumentar o sucesso e valor do negócio.
- Melhor visibilidade em tempo real.
- Redução de custos.
- Aumento de produtividade.

📍 Onde aplicar EAI?

Integração é relevante em:

- Sistemas legados
- Aplicações isoladas
- Fusões e aquisições

- SaaS
- Cadeias de abastecimento
- Cloud privada/pública
- CRM, ERP
- IoT
- Aplicações móveis
- Sistemas analíticos

⚙️ **Como implementar EAI?**

1. Identificar lacunas entre o estado atual e futuro.
2. Criar um plano de integração (blueprint).
3. Mapear transações e dependências.
4. Usar tecnologias:
    - Web services (SOAP, REST)
    - Troca de dados por ficheiros (CSV, XML)
5. Garantir segurança (certificados, credenciais, whitelist IPs).
6. Definir mecanismos de monitorização e manutenção.

📈 **Desafios de Negócio**

- Alinhamento de requisitos
- Ferramentas tecnológicas
- Inovação
- Participação das pessoas
- Investimento financeiro
- Capacidade técnica
- Formação

🔧 **Desafios Técnicos**

- Latência
- Segurança dos dados
- Integração de novos endpoints
- Debug/troubleshooting
- Uso de REST
- Grande volume de dados (IoT)
- Segurança imatura do IoT
- Uso imprevisível e conectividade em mobile
- Operações 24/7

Resumo do PDF: Service Oriented Architectures (SOA)

📌 Conceito de SOA

Service-Oriented Architecture (SOA) é uma arquitetura onde **serviços** são a principal forma de representar lógica de negócio, focando na **conectividade** e **reutilização** de componentes de software em ambientes distribuídos.

⚙️ Motivação para SOA

- A funcionalidade isolada já não é o foco; **conectividade** entre recursos é o novo valor.
- SOA surge como resposta à necessidade de um novo modelo de programação para lidar com múltiplos recursos distribuídos.
- Exemplo marcante: **Amazon** (Jeff Bezos) obrigou todas as equipas a expor funcionalidades exclusivamente através de interfaces de serviço.

🧩 O que é um Serviço?

- Conjunto de funcionalidades + políticas de uso.
- Acesso via **interfaces bem definidas** e com **restrições/políticas controladas**.

- Difere de um simples componente ou API pela aplicação dos princípios de **orientação a serviços**.

📄 Políticas de Serviço

Controlam:

- Quem pode usar
- Quando e como
- Tipo de resposta
- Custo envolvido

Isto permite diferentes **modelos de negócio**, como:

- Modelos fechados (telecomunicações)
- Modelos abertos (web)

🤝 Federação

- Estabelecimento de relações de confiança entre entidades.
- Pode ser **estática** (pré-definida) ou **dinâmica** (real-time).
- Exemplo: roaming com billing pré e pós-pago.

🔄 De Componentes para Serviços

Transição exige:

- Interfaces cliente
- Acoplamento fraco (via mensagens)
- Stateless
- Granularidade média a alta
- Independência de contexto

🔌 Interfaces de Serviço

- Não proprietárias

- Polimórficas e adaptáveis
- Múltiplos consumidores simultâneos
- Tolerantes a mudanças internas

💡 Intent e Offer

- Consumidor expressa uma **"intent"**
- Provedor oferece um **"offer"**
- Mediação pode ocorrer para casar intents com offers (tipo matchmaking).

🔄 Ciclo de Vida dos Serviços

1. Análise orientada a serviços (identificação de *service candidates*)
2. Design e desenvolvimento com foco em objetivos estratégicos
3. Cada serviço tem contexto funcional próprio

🧱 Componentes de Coordenação

- **Service Composition**: combinação de serviços para automatizar uma tarefa.
- **Orchestration**: processo centralizado, modela e controla os fluxos.
- **Choreography**: processo distribuído com múltiplos participantes.

🧭 Princípios da Orientação a Serviços

1. **Loose Coupling** – baixo acoplamento
2. **Service Contract** – contrato de comunicação formal
3. **Autonomy** – lógica encapsulada
4. **Abstraction** – esconde a complexidade interna
5. **Reusability** – promovido desde a concepção
6. **Composability** – serviços combináveis
7. **Statelessness** – sem dependência de estado

8. **Discoverability** – facilmente localizáveis

🚀 Características principais

- Baseado em **standards abertos**
- Promove **interoperabilidade**, **extensibilidade** e **descoberta**
- Suporta **agilidade organizacional** e diversidade de fornecedores
- Ideal para ambientes como **telecomunicações e empresas globais**

🧠 Ponto-chave

SOA **não é apenas um renome de API ou componentes**. É um paradigma **verdadeiramente novo**, adaptável, agnóstico à tecnologia, e centrado em reutilização e conectividade.

Resumo do PDF: API

🔧 Boas Práticas na Criação de APIs RESTful

1. **Base URL Simples**
    - Use substantivos, **não verbos**:
      ✅ /products, /products/12345
      ❌ /getAllProducts
2. **HTTP Methods Adequados**
    - GET – buscar recurso(s)
    - POST – criar recurso
    - PUT/PATCH – atualizar recurso
    - DELETE – apagar recurso
3. **Use Plurais**
    - Evita ambiguidades:
      ✅ /products
      ❌ /product/all

4. **Parâmetros de Query**
    - Para filtros e pesquisa:
      ✅ /products?name=ABC
      ❌ /getProductsByName
5. **Códigos HTTP Corretos**
    - 200 OK
    - 201 Created
    - 400 Bad Request
    - 401/403 Unauthorized/Forbidden
    - 404 Not Found
    - 500 Internal Server Error
    - 502 Bad Gateway
6. **Versionamento**
    - Sempre incluir na URL:
      ✅ /v1/products, /v2/products
      ❌ /v1.2/products
7. **Paginação**
    - Uso de limit e offset:
      /products?limit=25&offset=50
8. **Formato de Resposta**
    - Preferência: **JSON**
    - XML apenas se necessário para sistemas legados
9. **Mensagens de Erro Detalhadas**
    - Inclua code, message e, idealmente, um link de ajuda.
    - Exemplo (Facebook):
    - {
    - "error": {
    - "message": "Some of the aliases you requested do not exist: products",
    - "type": "OAuthException",

- "code": 803
- }
- }

10. **OpenAPI**
    - Padronização de design de APIs
    - Ferramentas como **Swagger** ajudam a documentar e validar

---

🔎 **GraphQL (Facebook, 2015)**

- **Um único endpoint** para todas as queries
- Linguagem de consulta baseada em tipos e campos
- Define o **schema** da API com o SDL (Schema Definition Language)
- Retorna sempre um objeto JSON com os dados pedidos
- Suporta:
    - **Queries** (leitura)
    - **Mutations** (escrita)
    - **Subscriptions** (tempo real)

Exemplo de Query:

```
query {
 human(id: "1000") {
   name
   height
 }
}
```

Resultado JSON:

```
{
 "data": {
   "human": {
     "name": "Luke Skywalker",
```

```
    "height": 1.72
  }
 }
}
```

⚔️ REST vs GraphQL – Comparação

Critério	REST	GraphQL
Data Fetching	Vários endpoints	Uma query com dados precisos
Over/Underfetching	Comum	Evitado
Flexibilidade	Limitada	Alta
Tipagem	N/A	Schema com SDL
Analytics	Limitado	Granular, baseado nas queries

🔄 WebSockets

- Protocolo IETF e API W3C (RFC 6455)
- Conexão **bidirecional e persistente**
- Evita overhead de HTTP e polling contínuo
- Ideal para aplicações **em tempo real** (ex: chat, notificações)
- Necessita definição de protocolo a nível de aplicação

Resumo do PDF: Micro Serviços

🔧 Arquitetura de Microserviços

1. **Conceito**
    - Arquitetura baseada em serviços pequenos, modulares e independentes.
    - Cada serviço tem uma responsabilidade única e comunica via APIs bem definidas.

2. **Arquitetura Monolítica vs Microserviços**
    - **Monolítica**:
        - Simples de desenvolver, testar, e escalar horizontalmente.
        - Dificuldades de manutenção, atualização e adoção de novas tecnologias.
    - **Microserviços**:
        - Melhor para CI/CD, equipas distribuídas, flexibilidade tecnológica.
        - Mais complexidade, testes difíceis, maior uso de recursos.
3. **Princípios Essenciais**
    - Escalabilidade, disponibilidade, resiliência.
    - Autonomia, governança descentralizada.
    - Isolamento de falhas, entrega contínua (DevOps).

🧩 **Padrões de Design**

1. **Decomposição**
    - Por capacidade de negócio (Ex: Orders, Customers).
    - Por subdomínio (Core, Supporting).
    - Por transação (2PC: prepare + commit/rollback).
    - **Strangler Pattern**: migrar de monólito gradualmente.
    - **Bulkhead Pattern**: isolar componentes para evitar propagação de falhas.
    - **Sidecar Pattern**: serviços acoplados ao processo principal.
2. **Integração**
    - **API Gateway**: ponto único de entrada, roteamento, segurança.
    - **Aggregator**: agrega dados de múltiplos serviços.
    - **Proxy / Gateway Routing**: variantes mais leves do gateway.
    - **Chained Microservice**: orquestração sequencial.
    - **Branch Pattern**: mistura de aggregator + chained.

- **Client-Side UI Composition**: composição no lado do cliente.

🛡️ **API Gateway – Motivações**

1. **Problemas sem Gateway**
   - Acoplamento forte entre cliente e serviços.
   - Muitos round-trips, maior latência.
   - Segurança exposta: todos os serviços ficam públicos.
   - Código duplicado para autenticação, logs, etc.
2. **Benefícios com Gateway**
   - Roteamento reverso (proxy) único para clientes.
   - Agregação de requisições.
   - Tratamento centralizado de:
     - Autenticação/autorização
     - Caching, retries, circuit breaker
     - Rate limiting, load balancing
     - Logging, tracing
     - Transformação de headers/claims/IP

🗂️ **Padrões de Base de Dados**

1. Database per Service
2. Shared Database
3. Brownfield Solution
4. **CQRS** (Command Query Responsibility Segregation): separar leitura de escrita.
5. **Event Sourcing**: modelo baseado em eventos.
6. **Saga Pattern**: coordenação de transações via eventos.

📊 **Observabilidade**

- Log Aggregation
- Performance Metrics
- Distributed Tracing
- Health Checks

⚙️ **Cross-Cutting Concerns**

- External Configuration
- Service Discovery
- Circuit Breaker
- Blue-Green Deployment

Resumo do PDF: Message Oriented Middlewares

🔧 **Enterprise Application Integration (EAI)**

1. **Objetivo**
    - Integrar aplicações de uma empresa para maximizar a sua utilidade.
    - Lidar com sistemas independentes que precisam comunicar.
2. **Tipos de Integração**
    - **Data Integration**: unificação de bases de dados.
    - **Process Integration**: coordenação a nível de processos.
3. **Tipos de Interação**
    - **Síncrona**: client/server com bloqueio.
    - **Assíncrona**: interação desacoplada.

🔄 **Interação Síncrona – Desvantagens**

- Ambas as partes precisam estar online.
- O cliente é bloqueado à espera da resposta.
- Problemas de escalabilidade e overhead.

- Risco de falha em cascata se chamadas falharem.

✅ **Soluções**:

- Pools de conexões
- Threads por conexão
- Alocação dinâmica de conexões

📨 **Fila de Mensagens (Reliable Queuing)**

- Permite interação assíncrona e modular.
- Ideal para distribuição sofisticada (multicast, replicação, etc).
- Mais natural para interações complexas.

📦 **Sistemas de Filas**

- Comunicação via filas persistentes (mesmo após falhas).
- Suporte a redes/sistemas heterogéneos.
- Promovem tolerância a falhas e flexibilidade.

📡 **Padrões com Filas**

1. **1-to-1**: comunicação com resiliência a falhas.
2. **1-to-many (multicast)**: útil para subscrições.
3. **Many-to-1**: concentração para priorização e ordenação.
4. **Many-to-many**: serviços replicados para muitos clientes.

🟡 Trade-off: maior flexibilidade vs. menor performance.

🔔 **Publish/Subscribe**

- Baseado em eventos:
    - Serviços **publicam** eventos.
    - Clientes **subscritos** recebem notificações via filas.
- O subscritor não precisa conhecer o publicador diretamente.

📌 Exemplo:

- Cliente subscreve a um tipo de mensagem.
- Quando publicada, o sistema entrega a todos os subscritores.

🧠 **Message Brokers**

- Adicionam lógica às filas e mensagens.
- Permitem definir **regras e processamento** para as mensagens.
- Maior complexidade e menor visibilidade global do sistema.

📌 Suportam dois modelos:

1. **Publish/Subscribe**: múltiplos produtores e subscritores.
2. **Point-to-Point**: produtor envia para fila, consumidor lê.

📦 **Exemplos de Brokers**:

- Apache Kafka
- Eclipse Mosquitto
- Java Message Service (JMS)

🔁 Em alguns casos, usa-se modelo **peer-to-peer** para baixa latência.

🌐 **Message Brokers vs. REST APIs**

Critério	**REST API**	**Message Broker**
Protocolo	HTTP (síncrono)	Mensagens (assíncrono)
Comunicação	Requisição/Resposta	Desacoplada, via tópicos/filas
Tolerância a falhas	Baixa (bloqueia se falhar)	Alta (mensagens persistem)
Escalabilidade	Limitada	Alta (pub/sub facilita)
Estado	Não mantido	Brokers rastreiam estado dos consumidores

Resumo do PDF: ESB

🔧 Enterprise Service Bus (ESB)

1. **Definição**
    - Arquitetura middleware que oferece serviços fundamentais para arquiteturas complexas.
    - Utilizado para implementar SOA (Service-Oriented Architecture).
    - Apresenta uma interface única e consistente para utilizadores e serviços.
2. **Objetivos**
    - Distribuir informação rapidamente pela empresa.
    - Abstrair diferenças entre plataformas, arquiteturas e protocolos.
    - Garantir entrega de mensagens mesmo com falhas.
    - Roteamento, registo e enriquecimento de mensagens sem reescrever aplicações.
    - Implementação incremental sem alterações totais.

⚙️ Funcionalidades Principais (Capabilities)

1. **Routing**
    - Direcionamento de pedidos baseado em critérios:
        - Estático (determinístico)
        - Baseado em conteúdo
        - Baseado em políticas
        - Regras complexas
2. **Message Transformation**
    - Converter estrutura e formato de mensagens conforme necessário.
3. **Message Enhancement**
    - Adição/modificação da informação da mensagem.

- Exemplos: conversão de datas, preenchimento de dados, normalização.

4. **Protocol Transformation**
   - Converter protocolo de entrada (ex: SOAP/JMS) para outro (ex: IIOP).
   - Ex.: XML/HTTP → RMI/IIOP
5. **Service Mapping**
   - Traduz serviço de negócio para implementação real.
   - Contém: nome do serviço, protocolo, info de ligação, timeout, failover.
6. **Message Processing**
   - Gestão de estado e sincronização das mensagens.
7. **Process Choreography**
   - Coordenação de vários serviços para completar um processo de negócio.
8. **Service Orchestration**
   - Coordenação de serviços implementados para um pedido composto.
9. **Transaction Management**
   - Execução de transações distribuídas como unidade de trabalho única.
10. **Security**
    - Proteção dos serviços:
      - Autenticação
      - Autorização
      - Auditoria
      - Controlo de acessos
      - Integração com gestores de segurança (não responsabilidade direta do ESB)

🧩 **Componentes do ESB**

1. **Mediator**
    - Responsável por: routing, transformação, comunicação, orquestração, segurança, etc.
2. **Service Registry**
    - Catálogo de serviços com mapeamento e descoberta (ex: UDDI).
3. **Choreographer**
    - Responsável por processamento de mensagens, transações e coreografia de processos.
4. **Rules Engine**
    - Lida com regras de routing, transformação e enriquecimento de mensagens.

Resumo do PDF: IAM (Identity, Authentication & Authorization)

🔐 **Identity Management**

1. **Provisionamento**
    - Adição de novos utilizadores aos diretórios dos sistemas e aplicações, tanto internos (empresa) como externos (parceiros).
2. **Gestão de Senhas**
    - Uso de um único conjunto de credenciais para acesso.
    - Permite que o utilizador administre suas próprias senhas e dados.
3. **Controlo de Acesso**
    - Aplica políticas de segurança por grupos.
    - Ex: impedir alteração do próprio cargo sem aprovação.

📄 **SAML (Security Assertion Markup Language)**

• Padrão aprovado pela OASIS (2005).
• Permite a partilha de informações de autenticação entre servidores.
• Baseado em **XML**, usa **SOAP**, **XMLDSig**, **XMLEnc**, e requer **SSL**.
• Não autentica diretamente; fornece **assertions** (declarações sobre identidade

ou permissões).
• É **agnóstico a fornecedores** e utilizado para **Single Sign-On (SSO)**.

Termos importantes:

- **Assertion:** declaração sobre um sujeito (ex: autenticado às 12h00).
- **Identity Provider (IDP):** entidade que gera as assertions.
- **Service Provider (SP):** consome e confia nas assertions.

Tipos de Assertions:

- Autenticação
- Atributos (ex: permissões)
- Decisões de autorização

Comunicação: protocolo simples de pedido/resposta via SSL.

🛡️ **XACML (eXtensible Access Control Markup Language)**
• Também aprovado pela OASIS (2005).
• Define políticas de acesso (quem pode fazer o quê).
• Totalmente XML, agnóstico a fornecedores.
• Usa **algoritmos de combinação de regras**:

- Ex: *Deny overrides* ou *Permit overrides*.

Componentes:

- **Policy Language:** define requisitos de acesso.
- **Request/Response Language:** permite perguntas do tipo "utilizador X pode aceder ao recurso Y?"
- Respostas possíveis: *permit, deny, indeterminate, not applicable*.

🌐 **OpenID**
• Iniciativa desde 2005, para autenticação de utilizadores web.
• Usado por grandes empresas (Google, IBM, PayPal, etc.).
• Menos "pesado" que o SAML, mas perdeu tração para o OAuth.
• **Altamente escalável**, sem necessidade de acordos prévios.

🔑 OAuth (e OAuth2)

• Padrão aberto para autenticação segura em APIs.

OAuth (RFC 5849 – 2010)

- Criado pelo Twitter (2006).
- **Token-based**: utilizador recebe token único para aceder aos dados.
- Não é expansível nem "Enterprise-ready".

Passos:

1. Obtenção do token de requisição
2. Autorização do utilizador
3. Troca do token por token de acesso
4. Acesso ao recurso protegido via:
    - Cabeçalho HTTP Authorization
    - POST
    - Query string

OAuth2 (RFC 6749 – 2012)

- **Novo protocolo**, não é compatível com OAuth original.
- Valida provedores via **HTTPS**.
- **Melhorias:**
    - Escopos de autorização (ex: só leitura)
    - Permite revogação de permissões previamente dadas

Resumo do PDF: Services in Telecom

📡 Transformação das Telecomunicações

- **Internet**: distribuída e aberta
- **Telecom tradicional**: monolítica e fechada

- As **Next Generation Networks (NGN)** procuram unir os dois mundos: infraestrutura unificada com abertura e flexibilidade.

🧱 Modelo 3GPP/TISPAN de Telecomunicações

- Três camadas principais:
    1. Tecnologias de acesso (GSM, CATV, PSTN, etc.)
    2. Transporte (com controlo de sessões, segurança, faturação)
    3. Serviços e Aplicações (servidores de conteúdos)

🧠 IMS (IP Multimedia Subsystem)

- **Fundamento arquitetural dos serviços de telecomunicações IP.**
- Garante: **QoS, faturação, segurança, integração de serviços.**
- Adotado por 3GPP, ITU, ETSI.
- Baseado em:
    - Sinalização **SIP**
    - Dados centralizados de assinantes
    - Segurança de fronteira
    - QoS fim a fim

Objetivo: "Always Best Connected"

- Funciona com qualquer dispositivo, tecnologia de acesso ou localização.
- Permite uma **rede única**, **controle de sessão comum**, **experiência de utilizador consistente**.

🧩 Componentes da IMS

- **Core IMS**: independente do acesso
    - **CSCF (Call Session Control Function)**
        - P-CSCF: primeiro ponto de contacto SIP
        - S-CSCF: regista e gere sessões

        - I-CSCF: gestão interdomínios
    - **HSS (Home Subscriber Server)**
        - Base de dados central de utilizadores e perfis
        - Suporte a autenticação, autorização, mobilidade e provisionamento

---

📇 Identidade na IMS

- Baseada em **SIP URIs** (ex: sip:utilizador@operadora.pt)
- **IMPI (identidade privada)**: usada na autenticação
- **IMPU (identidade pública)**: usada na comunicação
- Cada IMPI pode ter vários IMPUs (ex: profissional, pessoal, etc.)

🎛️ Serviços na IMS

- Os serviços residem nos **Application Servers (AS)**, externos à IMS.
- AS usam SIP para comunicar com S-CSCF.
- Atuam como proxy SIP ou User Agent.
- **Filter Criteria** no perfil do utilizador determinam quando os AS são acionados:
    - Critérios por domínio, tempo, localização, etc.
    - Flexibilidade para direcionar sessões a diferentes AS.

⚙️ Desafios das Operadoras

- Infraestruturas heterogéneas, criadas em diferentes épocas.
- Aplicações isoladas (ex: faturação, atendimento, etc.)
- Alta **complexidade, dependências, manutenção difícil.**
- Forte **acoplamento entre serviços e infraestrutura de rede.**

🚀 SDP (Service Delivery Platform)

- Plataforma para **criação e entrega rápida de serviços**.
- Define um **modo padronizado** de desenvolver e aceder a sistemas.
- Permite:
    - Redução do tempo de desenvolvimento
    - Reutilização de competências e ferramentas
    - Menor risco em novos projetos
- Exemplo:
    - Antes: cada serviço acede sistemas de forma própria
    - Com SDP: acesso padronizado → mudanças não afetam os serviços

---

Resumo do PDF: Integrated Virtualization Approaches

📦 **Container Virtualization**

- **Virtualização ao nível do SO**: não virtualiza hardware completo, apenas serviços do sistema operativo.
- **Container**: ambiente isolado para execução de processos.
- **Vantagens**:
    - **Isolamento**: cada container vê apenas os seus próprios recursos.
    - **Controlo de recursos**: CPU, RAM, I/O, etc.
    - **Portabilidade**: mesmo ambiente replicado em vários hosts.

🔧 **Tecnologias base (Linux)**

- **Namespaces**: isolamento de recursos (PID, NET, MNT, IPC, etc.)
- **Cgroups**: controlo de recursos por grupo de processos
- **chroot, AppArmor, SELinux, seccomp**: segurança e isolamento extra

---

🛠️ **Como funciona um container**

1. Criar namespaces e cgroups
2. Criar imagem do sistema de ficheiros

3. Usar chroot para entrar no container
4. Iniciar o processo (PID 1 no container)

🔌 **Rede em containers**

- Containers não veem interfaces reais do host
- Solução: par de interfaces virtuais (veth) ligadas a bridges

---

🖥️ **Container vs Máquina Virtual (VM)**

Característica	VM	Container
Sistema operativo	Diferente por VM	Compartilhado
Boot	Lento (sistema completo)	Rápido (instância do app)
RAM/CPU	Pré-reservado	Sob demanda
Overhead	Alto	Quase zero
Escalabilidade	~10 VMs/core	~100 containers/core

💡 Container = processo isolado, não simulação completa de hardware

---

⚙️ **Ferramentas**

- **LXC**, **Docker**, **Singularity**, **Kubernetes**
- **OCI (Open Container Initiative)**: padrões para portabilidade e compatibilidade

---

🐳 **Docker**

- Evolução do LXC, open source desde 2013
- Conceitos:
    - **Imagem**: dados do container (app, libs, config)
    - **Container**: instância em execução
    - **Engine**: software que executa containers
    - **Registry**: armazém de imagens (ex: Docker Hub)

    - **Control Plane**: gestão de containers

🔁 **Workflow Docker**

$ docker search nginx

$ docker pull nginx

$ docker image ls

$ docker run -d --name frontend-server nginx

📜 **Dockerfile**

- Lista comandos para criar o container
- Define libs, comandos, entrypoint
- Imagens são compostas por **camadas** (overlayfs)

📦 **Persistência**

- Camada de escrita não persiste por defeito
- Usar:
    - **Bind mounts**: caminho do host montado no container
    - **Volumes**: geridos pelo Docker (não apagados com container)

🌐 **Redes Docker**

- Tipos: bridge, host, none, overlay, macvlan
- Portas devem ser explicitamente expostas:

$ docker run -d -p 8000:80 nginx

🔐 **Dados sensíveis**

- Não incluir diretamente em imagens
- Melhor: usar **Docker Secrets**

📦 **Docker Compose**

- Arquivo docker-compose.yml descreve múltiplos containers como um serviço completo
- Pode incluir volumes, variáveis, secrets, etc.

☸️ **Kubernetes (K8s)**

- Plataforma de **orquestração de containers** (Docker/Containerd)
- Oferece:
    - **Auto-healing**, **balanceamento de carga**, **escalabilidade**, **gestão declarativa**
- Gere:
    - Containers, volumes, redes, secrets, configuração

🧱 **Componentes Principais**

- **kube-apiserver**: API REST
- **etcd**: armazenamento de estado do cluster
- **kube-controller-manager**: garante estado desejado
- **kube-scheduler**: decide onde instanciar pods
- **kube-proxy**: regras de rede e encaminhamento
- **kubelet**: garante que containers estão a correr no nó

🧩 **Objetos Kubernetes**

- **Pod**: grupo de containers, unidade básica
- **Deployment**: define como e quantas instâncias de um pod devem existir
- **StatefulSet**: igual ao deployment, mas com dados persistentes e identificadores únicos (ex: bases de dados)
- **Service**: IP estático e nome DNS para acesso interno aos pods
- **Ingress**: expõe serviços ao exterior (via proxy como Traefik)
- **Job**: execução pontual de uma tarefa
- **ConfigMap**: dados de configuração (não sensíveis)
- **Secret**: dados sensíveis (ex: tokens, passwords, certificados)

Resumo do PDF: Cloud Computing

☁️ **Tendências Atuais**

1. **Explosão de Dados**

- Petabytes são comuns.
- Milhões de núcleos de processamento.
- Uso crescente de sistemas virtualizados.

2. **Infraestruturas em Nuvem**
   - Data centers consolidados.
   - Arquitetura baseada em multicore/manycore.
   - Hardware COTS (comercial, barato).
   - Falhas são comuns, mas invisíveis para o utilizador.
3. **Tecnologias Relacionadas**
   - Sistemas de ficheiros distribuídos.
   - Frameworks como MapReduce.
   - Linguagens de alto nível para aplicações paralelas.

📊 Infraestrutura de Serviços de Informação

1. **Escala**
   - EUA: 38 milhões de servidores físicos.
   - Grande crescimento esperado.
   - Alto desperdício de capacidade (subutilização).
2. **Custo e Consumo**
   - Data centers são caros e não ecológicos.
   - PUE (Power Usage Effectiveness): 1.2–3.
   - Ex: Google >400MW, Microsoft >160MW.
3. **Problemas**
   - **Dimensionamento Difícil**: pico vs. média → desperdício ou insatisfação.
   - **Escalabilidade**: difícil escalar para cima ou para baixo.
   - **Alto Custo**: hardware + manutenção.
   - **Alta Disponibilidade**: exigência de uptimes quase perfeitos (ex: 99.9999%).

📈 Estudos de Caso

- **Animoto**: Escalou de 40 para 3500 instâncias em 3 dias após uma promoção no Facebook.
- **Novartis**: 87.000 núcleos em EC2 Spot → 39 anos de química computacional em 9h por $4.232.

🌐 Computação em Nuvem

1. **Modelo**
    - Serviços sob demanda via internet.
    - Escalabilidade horizontal sem atrasos.
    - Transparência da complexidade para o utilizador final.
2. **Vantagens**
    - Pay-per-use.
    - Escalabilidade instantânea.
    - Segurança e confiabilidade.
    - APIs acessíveis.
3. **Tipos de Serviço**
    - **IaaS (Infraestrutura como Serviço)**: ex. Amazon EC2, Google Cloud.
        - Acesso completo à infraestrutura.
    - **PaaS (Plataforma como Serviço)**: ex. Heroku, Google App Engine.
        - Foco no desenvolvimento, sem gestão de infraestrutura.
    - **SaaS (Software como Serviço)**: ex. Google Apps, Salesforce.
        - Acesso via browser, sem instalação local.

🏷️ Tipos de Nuvem

- **Nuvem Pública**: Ex. AWS, Azure.
- **Nuvem Privada**: Para uso interno de uma organização.

- **Nuvem Comunitária**: Compartilhada entre organizações com interesses comuns.

🔧 Outros “...as a Service”

- Network, Storage, AI, Energy Storage, Security...
