Revisão arquitetura de sistemas

Aula 1:

Conceitos Fundamentais

Sistema é um conjunto de componentes que interagem para atingir um objetivo comum. A arquitetura de sistemas é a maneira como os componentes de um sistema se organizam e se comunicam.

- Computação centralizada: todos os recursos estão em um único local, como um mainframe.

- Computação distribuída: os recursos estão espalhados em vários nós que se comunicam através de uma rede. Um nó é uma unidade autônoma em um sistema distribuído, e a rede é a infraestrutura que permite a comunicação entre eles, graças a protocolos de comunicação.

- Escalabilidade: capacidade de um sistema crescer em desempenho ou volume,

- Transparência: ocultar a complexidade do sistema para o usuário, de forma que as mudanças internas não afetem a interface interna

- Tolerância a falhas: capacidade de um sistema continuar funcionando mesmo se algumas de suas partes falharem. Isso é ideal para sistemas robustos e descentralizados

- Latência: o tempo entre uma resposta e a requisição

- Concorrência: múltiplas operações disputando um único recurso

- Sincronização: componentes se coordenam para garantir a consistência

- Consistência: manter um estado global coerente, com diferentes modelos como forte, eventual, casual e sequencial

    - Forte: mais rigoroso, garante que qualquer operação de leitura sempre retorne o valor mais recente, como se tivesse apenas um nó no sistema. É ideal para transações financeiras, mas pode prejudicar a escalabilidade e disponibilidade

    - Eventual: modelo mais flexível, garante que se não houver novas atualizações, todas as replicas eventualmente terão o mesmo valor, ou seja, pequenas inconsistências temporárias são aceitáveis. É a escolha ideal pra sistemas que precisam ser altamente escaláveis e disponíveis, como as redes sociais

    - Sequencial: Toas as operações em todos os nós sejam vistas na mesma ordem sequencial, como se houvessem sido executadas em

um único lugar, mas não em tempo real, isso é útil para aplicações que a ordem das operações é importante

- Causal: modelo mais fraco, garante que as operações com uma relação de causa e efeito sejam vistas na mesma ordem, mas aquelas que não estão relacionadas podem ser vistas em ordens diferentes. É um bom equilíbrio entre consistência e desemprenho sendo útil em sistemas de chat e fóruns online

Limites da computação

Limites físicos: impostos pelas leis da natureza, como matéria e energia

- Tamanho dos transistores: A miniaturização aproxima-se de um ponto onde efeitos quânticos, como tunelamento, interferem no comportamento dos circuitos

- Dissipação de calor: Processadores mais rápidos consomem mais energia e geram calor, o que pode causar falhas. A velocidade da luz também impõe latência mínima para a comunicação entre componentes, o que é um grande desafio para sistemas grandes.

Limites teóricos: restrição da lógica e da matemática

- Problemas não computáveis: não podem ser resolvidos por nenhum algoritmo, independente do poder do processador.

- Complexidade computacional: alguns problemas são teoricamente solúveis, mas exigem tanto tempo e memória que se tornam impraticáveis, como os problemas da classe NP

Limites práticos: Restrições ligadas á construção, manutenção e evolução de sistemas no mundo real, como custo e complexidade de engenharia.

- Monolitismo: Sistemas construídos como um bloco único, tendem a se tornar rígidos, dificultando a evolução, os testes e a manutenção

- Escalabilidade e manutenção limitadas: muitos sistemas não são projetados para o crescimento, o que leva a degradação de desempenho e custos de manutenção insustentáveis

- Custo de infraestrutura: manter servidores locais e fazer upgrades constantes pode ser economicamente inviável

Sistemas Distribuídos

Um sistema distribuído é um conjunto de computadores independentes que trabalham em conjunto para um objetivo comum, parecendo uma única unidade para o usuário. Ele permite escalabilidade, redundância/resiliência

(continua funcionando mesmo se partes falharem), geodistribuição (nós podem ser posicionados em locais diferentes para reduzir a latência para usuários em diversas partes do mundo), compartilhamento de recursos.

Aula 2:

Revisão histórica: a era centralizada

Entre 195 e 1970 foi a era dos mainframes e terminais burros. Os mainframes eram computadores grandes e. caros que centralizavam todo o processamento e armazenamento de dados, acessíveis por meio de terminais simples que não tinham capacidade de processamento. Essa arquitetura tinha limitações:

- Ponto único de falha
- Escalabilidade limitada ao poder do hardware central
- Custos de aquisição e manutenção muito altos

A evolução seguiu nos anos de 1970 e 1980, com os minicomputadores e as primeiras redes locais (LANs). Embora mais baratos, esses sistemas ainda dependiam de um servidor central, que podia se tornar um gargalo.

Período	Marco Histórico	Características
Década de 1950–60	**Mainframes e terminais burros**	Computação centralizada, alto custo, processamento batch.
Anos 1970	**Arquitetura Cliente-Servidor**	Servidor central com múltiplos clientes conectados.
Anos 1980	**Redes Locais (LANs)**	Computadores autônomos compartilhando recursos via rede.
Anos 1990	**Internet e sistemas corporativos distribuídos**	Integração de sistemas geograficamente dispersos.
Anos 2000	**Computação em Grade e Clusters**	Alto desempenho com nós interligados; surgimento de HPC acessível.
Anos 2010	**Microsserviços e Cloud Computing**	Arquiteturas distribuídas elásticas, escaláveis e tolerantes a falhas.
Anos 2020+	**Edge Computing e IoT**	Processamento próximo à origem dos dados, integração massiva de dispositivos.

Sistemas Distribuídos: é apresentado como uma coleção de computadores independentes que, para os usuários, se comportam como um sistema único e coerente. Diferente dos modelos anteriores, os componentes dos sistemas distribuídos se comunicam apenas trocando mensagens

Critério	Centralizado	Distribuído
Arquitetura	Um ponto central de processamento e dados (servidor/mainframe)	Vários nós autônomos coordenados por rede
Escalabilidade	**Vertical** (mais CPU/RAM no mesmo nó)	**Horizontal** (adicionar nós/instâncias)
Disponibilidade	Ponto único de falha (SPOF)	Redundância/failover; elimina SPOF bem projetado
Latência	Baixa em LAN/local	Variável; depende de rede/geo (CDN, edge ajudam)
Consistência	Forte (ACID "natural")	Variável; forte/eventual/causal (CAP entra em cena)

Critério	Centralizado	Distribuído
Tolerância a falhas	Limitada; recovery central	Alta: replicação, quorum, reeleição de líderes
Segurança	Perímetro único, controle central	Superfície de ataque ampliada; exige zero-trust
Operação/DevOps	Mais simples de operar	Mais complexidade (observabilidade, orquestração)
Ciclo de mudanças	Rápido no início; degrada com o tempo (acoplado)	Autonomia por serviço; coordenação inter-times
Custos	Previsíveis; CAPEX alto	OPEX variável; custo por uso; risco de "bill shock"
Dados	Sem sharding/replicação complexa	Particionamento, replicação, consenso, Sagas
Casos típicos	Core bancário legado, ERP local, SIS acadêmico pequeno	Streaming, e-commerce global, IoT, redes sociais

Lei de amdahl: usada para encontrar a maxima melhora esperada para um sistema em geral quando apenas uma única parte dele é melhorada. É constantemente usado para prever o maximo speedup teórico, usando múltiplos processadores.

A Lei de Amdahl mostra que a parte sequencial do código é um fator limitante para o ganho de desempenho. Por mais que você aumente o número de processadores, o *speedup* total nunca será maior do que o inverso da parte sequencial. Em outras palavras, mesmo em um sistema com processamento infinito, o desempenho geral será limitado pela parte do programa que não pode ser paralelizada.

Centralizado se:

- Escopo e público são locais e estáveis.
- Consistência forte é mandatória e latência ultra-baixa no site.
- Time e orçamento enxutos; requisitos mudam pouco.
- Dependências externas e integração são limitadas.

Distribuído se:

- Usuários/geografia dispersos; picos imprevisíveis.
- Alta disponibilidade e continuidade de negócio são críticos.
- Há múltiplas equipes entregando features de forma independente.
- Integrações extensas (APIs externas, IoT, analytics em tempo real).

- Uma CDN, ou Rede de Entrega de Conteúdo (Content Delivery Network), é uma rede de servidores distribuídos geograficamente que acelera a entrega de conteúdo da web. A ideia principal é aproximar o conteúdo do usuário final.

Aula 3:

Processo: é uma instancia de um programa em execução, ele tem seu próprio espaço de memória, registradores de CPU e contexto de execução. Os processos são isolados uns dos outros, o que significa que a falha de um não afeta os demais.

Thread: é a menor unidade de processamento que pode ser agregada pelo sistema operacional e existe dentro de um processo. Threads compartilham o mesmo espaço de memória ei recursos do processo, mas tem seu próprio contador de programa, registradores e pilha. A criação de threads é mais barata do que a de processos.

Concorrência: é a execução alternada de múltiplas tarefas que dá a impressão de simultaneidade (como em uma CPU de um único núcleo)

Paralelismo: execução real de tarefas ao mesmo tempo (como em CPUs com múltiplos núcleos)

Em sistemas distribuídos, a concorrência local (com threads) é combinada com o paralelismo global (com processos em máquinas diferentes)

O papel de Threads e processos em sistemas distribuídos:

- Processo representa uma unidade de execução independente em diferentes máquinas, comunicando-se através da rede

- Thread são usadas dentro de um processo para lidar com múltiplas requisições simultâneas. Elas permitem um melhor aproveitamento de CPUs multicore e reduzem a latência em operações de I/O.

Característica	Processo	Thread
Memória	Espaço de endereçamento próprio	Compartilha memória com outras threads do processo
Isolamento	Total	Parcial
Custo de criação	Alto	Baixo
Comunicação	IPC (pipes, sockets, memória compartilhada)	Direta (variáveis compartilhadas)
Falha	Não afeta outros processos	Pode derrubar o processo inteiro

A comunicação entre processos em ambientes distribuídos é conhecida como IPC (Inter-Process Communication), é essencial e pode acontecer através de vários mecanismos:

- Pipes

- Filas de mensagens, que permitem a troca assíncrona de dados, como em sistemas middleware

- Memória compartilhada, simulada em ambientes distribuídos

- RCP (Remote Procedure Call), que permite a chamada de funções em máquinas remotas como se fossem locais

Modelos de propagação e desafios:

- Thread-per-connection: Cria uma nova thread para cada cliente que se conecta. É simples, mas não escala bem em alto volume de conexões devido ao alto consumo de memória.

- Thread pool: conjunto fixo de threads antecipadamente. Novas conecoes são atendidas por threads disponíveis no pool, sendo mais eficiente para alto volume de conexões

- Race condicions: quando múltiplas threads acessam e modificam recursos compartilhados sem coordenação, levando a resultados imprevisíveis

- Deadlock: acontece quando um conjunto de threads fica bloqueado esperando recursos que nunca serão liberados, gerando um grande impasse.

- Starvation: ocorre quando um ou mais processos nunca recebem recursos da CPU, pois os outros os monopolizam.

- Overhead de contexto: A troca de contexto entre threads/processos gera um custo elevado, o que afeta a escalabilidade em ambientes com muitos processos. Isso levou o surgimento de modelos como thread pools e event loops.

Em um sistema distribuído existem técnicas para implementar a tolerância a falhas. Essas são heartbeats para detectar a falha e replicação de dados para garantir a consistência.

Aula 4:

Socket é uma interface de software que permite a comunicação entre processos, estejam eles na mesma máquina ou em máquinas diferentes conectadas por uma rede, as aplicações conseguem se comunicar para trocar dados. Sockets foram introduzidas no BDS Unix nos anos de 1980 e são a base de grande parte da internet. Elas podem operar sobre protocolos como TCP ou UDP, localizados na camada de transporte do modelo TCP/IP. São essenciais para funcionamento de navegadores, e-mails, jogos online e outros sistemas que trocam dados pela rede.

Exemplos de uso:

- Pool de threads para atender requisições de múltiplos clientes de forma concorrente e escalável.

- O cliente é a parte que inicia a comunicação. Ele abre um socket TCP, se conecta ao servidor, envia uma mensagem e aguarda a resposta. Em seguida, ele imprime a resposta completa do servidor e a mensagem recebida.

A combinação de sockets com threads é fundamental para construir servidores eficientes, principalmente aqueles que precisam lidar com muitos clientes ao mesmo tempo. sockets criam os canais de comunicação para a rede e threads permitem que um único processo utilize esses canais de forma simultânea e eficiente, tornando os servidores capazes de lidar com a carga de múltiplos clientes sem lentidão ou falhas.

Aula 5:

A comunicação em sistemas distribuídos é muito importante para possibilitar a coordenação entre processos, consistência de dados, tolerância a falhas e escalabilidade. Os principais são Remote Procedure Call (RPC), Mensageria e Streams de dados.

A comunicação pode ser:

- Síncrona: O cliente envia uma mensagem e aguarda a resposta antes de continuar a execução. A vantagem é a simplicidade de programação, mas pode causar bloqueios e perda de desempenho se a rede for lenta. Um exemplo é RCP.

- Assíncrona: O cliente envia mensagem e continua sua execução sem esperar resposta imediata. Isso oferece maior escalabilidade e tolerância a falhas, mas exige uma programação mais complexa para lidar com call-backs (é uma função que é passada como argumento para outra função, quando a tarefa assíncrona é concluída, a função que a executou "chama de volta" a função call-back, passando o resultado) ou pooling (Cliente verifia periodicamente o status da operação para checar se ela foi concluída). Um exemplo é envio de mensagem para uma fila como RabbitMQ ou Kafka

A comunicação em relação a orientação a conexão pode ser:

- Orientada a conexão: requer estabelecimento de conexão confiável antes da transmissão de dados, garante a entrega e ordem dos pacotes, mas apresenta um overhead maior (TCP é um exemplo).

#obs: *overhead* é a carga extra que um sistema tem para gerenciar seus recursos e operações, e um bom design de sistema busca minimizar esse custo para melhorar o desempenho geral.

- Sem conexão: as mensagens são enviadas em um canal dedicado, é mais rápido, tem baixa latência, ideal para streaming de áudio/vídeo, mas não oferece garantias de entrega ou ordem

Modelos de comunicação:

- Remote procedure Call (RPC): permite que um processo em uma máquina invoque uma função de outra máquina como se fosse uma chamada local, a complexidade de rede é avstraida para o programador.

    - Como funciona: Um stub (cópia local da função) no lado do cliente empacota os parâmetros (*marshalling*) e os envia pela rede.
    O stub no lado do servidor recebe, desempacota (*unmarshalling*) e invoca a função real.

    - Vantagens: simplicidade de uso e abstração de rede. Para o programador, chamar uma função remota parece tão simples quanto chamar uma função local, já que a complexidade de sockets, pacotes e protocolos de rede fica oculta.

    - Desvantagens: Alto acoplamento, pois cliente e servidor precisam concordar sobre a interface da função. Além disso, mascara falhas de rede como se fossem falhas locais, criando a ilusão que chamadas remotas são tão confiáveis quanto as locais.

- Mensageria (Message-Oriented Middleware): modelo de comunicação assíncrona onde o remetente (produtor) e o destinatário (consumidor) são desacoplados no tempo. As mensagens são enviadas através de um

intermediário, o message broker (intermediário de comunicação), que as armazena em filas ou tópicos.

- Ponto-a-ponto (Fila): Cada mensagem é consumida por um único consumidor, é ideal para balanceamento de carga, pois a fila distribui as mensagens entre os consumidores

- Publish/Subscribe (Tópico): Uma mensagem pode ser entregue a vários consumidores que estejam inseridos no tópico.

- Vantagens: Maior resiliência, escalabilidade e flexibilidade na integração de sistemas, pois o remetente não precisa se preocupar com a disponibilidade do destinatário.

Característica	Ponto-a-Ponto (Queue)	Publish/Subscribe (Topic)
Consumidor	1 por mensagem	Vários possíveis
Uso típico	Balanceamento de carga	Notificações, eventos
Persistência	Mensagem some após consumo	Pode ser reprocessada por vários
Exemplo	Fila de pedidos	Alerta de novo conteúdo

- Stream de dados:

- Modelo de fluxo contínuo de dados, onde os eventos são produzidos e consumidos em tempo real, funciona como um log distribuído (funciona como um registro imutável e ordenado de eventos que ocorrem em um sistema distribuído), onde os eventos ficam registrados e podem ser lidos em tempo real ou reprocessados posteriormente

-Diferença da mensageria: na mensageria, uma mensagem é consumida e geralmente removida, em streams, as mensagens permanecem no log imutável, permitindo o reprocessamento do histórico

Mensageria (Filas/Tópicos)	**Streams (Logs Distribuídos)**
Mensagem consumida geralmente é removida.	Mensagens ficam gravadas como log imutável.
Comunicação **assíncrona**, mas ainda focada em entrega “uma vez” (ou broadcast).	Comunicação contínua, com possibilidade de **replay** de eventos passados.
Uso típico: desacoplamento entre sistemas (ex.: fila de pedidos, notificações).	Uso típico: análise em tempo real + histórico (ex.: métricas, logs de acesso).

Características:

1. Persistência longa
2. Leitura em paralelo
3. Reprocessamento
4. Escalabilidade horizontal

Característica	RPC	Mensageria	Streams
Estilo de comunicação	Síncrono	Assíncrono	Contínuo
Acoplamento	Alto	Médio	Baixo
Uso típico	Chamadas diretas	Integração de sistemas	Processamento em tempo real
Exemplo moderno	gRPC	RabbitMQ	Kafka

Aula 7:

Tolerância a falha é a capacidade de um sistema de continuar operando corretamente mesmo quando uma ou mais de suas partes falham. Em sistemas distribuídos falhas são eventos esperados devido ao grande número de componentes, nós, processos e redes envolvidos.

Tipo de falha:

- Falha de Crash: Quando um processo ou servidor para de responder abruptamente. Pode ser causada por um desligamento inesperado de um nó, um processo que trava ou a queda de energia de um datacenter. Boas práticas para lidar com isso incluem:

    1. redundância de servidores
    2. Health checks e monitoramento: para detectar rapidamente quando um processo para de responder
    3. Failover automático: redireciona requisições para um instancia saudável sem intervenção manual

- Falha de omissão: Ocorre quando uma mensagem não chega ao seu destino. Isso pode acontecer por perda de pacotes na rede ou por buffers cheios que precisam descartar mensagens. Soluções incluem uso de protocolos confiáveis como TCP e o design de operações para serem idempotentes, de forma que o resultado não mude mesmo se a mensagem for processada mais de uma vez.

- Falhas temporais: ocorrem quando a resposta de um processo ou servidor chega fora de tempo esperado, resultando em timeouts. Para lidar com isso:

    1. Timeouts bem definidos: para evitar que o cliente espere indefinidamente por uma resposta

2. Retry com backoff exponencial: esperar intervalos progressivamente maiores entre as tentativas, evitando sobrecarregar um servidor já lento

3. Circuir breaker: um padrão que "abre o disjuntor" para parar de enviar requisições a um serviço que está constantemente falhando

- Falhas bizantinas: ocorrem quando um componente do sistema age de forma imprevisível, enviando mensagens incorretas ou contraditórias. É mais difícil de detectar do que uma falha de crash. O seu nome é uma referência ao Problema dos generais bizantinos. Estratégias para lidar com esse tipo de falha inclui uso de protocolos de consenso bizantino (BTF), redundância com votação, criptografia e um design de arquitetura de confiança zero

Com isso, percebemos a importância da redundância por meio da replicação de processos, que cria cópias extras de componentes críticos, do failover automático para redirecionar requisições em caso de falha e o uso de checkpoints e logs.

Aula 8:

Recuperação de erros: Técnicas que permitem que um sistema volte a um estado estável ou aceitável após falha, com o objetivo de restaurar a operação. Isso é visto em sistemas operacionais que matam um processo travado, em bancos de dados que usam logs de transações para garantir consistência e em sistemas de cloud computing com failover automático.

Resiliência: propriedade sistêmica que permite a um sistema absorver falhas (de hardware, software, rede ou humana) e continuar prestando um serviço com impacto mínimo para o usuário.

Enquanto a recuperação de erros foca em restaurar após falha, a resiliência foca em manter o funcionamento durante a falha.

Características de um sistema resiliente:

1. Continuidade de serviço: aplicação não para, mesmo que partes do sistema falhem
2. Degradação graciosa: o sistema reduz funcionalidades, mas não interrompe totalmente o serviço
3. Elasticidade: capacidade de se adaptar automaticamente a cargas maiores ou falhas, usando escalabilidade horizontal
4. Autonomia: detecta e corrige falhas se intervenção manual imediata

Estratégias de recuperação de erros:

1. Retries com backoff exponencial: em vez de desistir de uma requisição que falhou, o cliente repete automaticamente. Para evitar a tempestade de retries que pode sobrecarregar o servidor, a prática recomenta esperar intervalos progressivamente maiores entre as tentativas, uma técnica conhecida como backoff exponencial, e a adicionar uma variação aleatória (jitter) para que os clientes não tentem novamente no mesmo instante.
2. Checkpoint e log: salvar periodicamente o estado da aplicação (checkpoint) e registrar as operações em um log para que, em caso de falha, o sistema possa se recuperar sem perder todo o progresso. Checkpoint fornece um estado consistente, enquanto o log permite reprocessar eventos que ocorreram depois do último ponto salvo. Bancos de dados usam logs como o WAL (Write-Ahead log) para garantir a durabilidade e a consistência após um crash
3. Failover Ativo-Passivo: para alta disponibilidade, um servidor primário (ativo) atende a todas as requisições, enquanto um servidor secundário (passivo) permanece em espera. Se o primeiro falhar, o tráfego é redirecionado para o secundário, geralmente por um balanceador de carga, o que minimiza a interrupção para o usuário.
4. Fallbacks: oferece uma resposta alternativa quando o serviço principal falha. Ele é útil para dependências instáveis ou funcionalidades não criticas. Suas funcionalidades incluem retornar dados de um cache local, oferecer uma versão simplificada do serviço ou usar padrões como o circuit breaker (Um padrão de projeto que protege um serviço de sobrecarga ao interromper as chamadas para dependências que falham repetidamente, evitando o colapso do sistema)

Propriedades sistêmicas e observabilidade:

1. Redundância e replicação: ter copias extras de componentes e dados em diferentes nós para que a falha de um não comprometa a operação
2. Elasticidade: capacidade de ajustar os recursos dinamicamente para se ajustar a alta demanda e falhas
3. Circuit breakers

Observabilidade é a capacidade de entender o comportamento interno de um sistema a partir dos sinais que ele emite. Ela se baseia em três pilares: métricas (valores numéricos como CPU e latência), logs (registros detalhados de eventos) e tracing (o rastreamento de uma requisição por múltiplos serviços).

Aula 9:

Introdução a segurança distribuída: não é apenas sobre proteger um ponto de entrada central, é uma propriedade emergente que depende da iteração segura entre múltiplos nós e redes potencialmente inseguras. O dilema da confiança fala sobre como garantir que todos os nós se comportem corretamente, já que cada um pode ser um ponto de falha ou de entrada para ataques, isso leva a necessidade de arquitetura Zero Trust, onde nenhum componente é confiável por padrão.

Os principais desafios são:

1. Superfície de ataque ampliada: enquanto sistemas monolíticos tem um único ponto central para proteger, sistemas distribuídos tem múltiplos nos que podem ser alvo de ataques
2. Autenticação e autorização: é um desafio gerenciar as identidades e permissões em centenas de serviços diferentes, tem algumas soluções modernas como OAuth 2.0, OpenID Connect e JSON Web Tokens (JWT) para autenticação federada.
3. Comunicação segura: dados trafegam constantemente entre os nos. Para evitar a intercepção o uso de TLS/HTTPS é essencial, pois ele criptografa os dados e garante a autenticidade do servidor, isso protege contra ataques como MITM.
4. Gerenciamento de segredos: credenciais como tokens e chaves de API nunca devem ser armazenadas em um código fonte, repositórios Git, planilhas ou e-mails. A solução é usar cofres de segredos como HashiCorp Vault ou Azure Key Vault, que centralizam o armazenamento e permitem rotação automática de chaves.
5. Disponibilidade sob ataque: ataques DDoS (Distributed denial of service) carregam o sistema com requisições, tornando-o indisponível. As defesas incluem o uso de rate limiting para impor limites de requisições, balanceadores de carga inteligentes para distribui o tráfego e CDNs para absorver picos de tráfego malicioso.

Estratégias de segurança:

1. Defesa em camadas: a ideia de não confiar em um único mecanismo de segurança e construir múltiplas barreiras de proteção. Essas camadas podem incluir autenticação forte, criptografia em trânsito e em repouso, firewalls e monitoramento.
2. Princípio do menos privilégio: garante que cada usuário ou serviço tenha apenas permissões estritamente necessárias para realizar sua função, limitando o dano em caso de comprometimento
3. Auditoria e observabilidade: capacidade de registrar e rastrear o que acontece no sistema é crucial para detectar anomalias e investigar incidientes. O uso de um SIEM (Security information and event manegement), como o elastic SIEM, permite centralizar, correlacionar e analisar logs de toda a infraestrutura

4. Testes e validações contínuas: a segurança deve ser continuamente validada, não apenas em auditorias anuais, as abordagens incluem:
    a. Testes de penetração (Pentests): simulam ataques para encontrar falhas
    b. Varreduras de vulnerabilidade: ferramentas automatizadas que chevam software desatualizado
    c. Chaos engineering: injeta falhas de segurança para testar a resiliência do sistema
    d. DevSecOps: integra a segurança em todas as fases do ciclo de desenvolvimento desde a codificação até a implementação.