Solução do Problema 1: Automação de Ambientes Operacionais

A seguir, apresento um script em Python que automatiza o processo de gerenciamento de backups. O código foi desenvolvido para ser robusto e configurável, cumprindo todos os requisitos solicitados: listar arquivos, remover os que possuem data de criação superior a 3 dias, copiar os mais recentes e gerar os logs correspondentes.

```
from __future__ import annotations

import argparse
import os
import shutil
import sys
from datetime import datetime, timedelta
from pathlib import Path
from typing import List, Tuple, Dict

DEFAULT_SOURCE = Path(os.environ.get("BACKUP_SOURCE",
"/home/valcann/backupsFrom"))
DEFAULT_DEST = Path(os.environ.get("BACKUP_DEST",
"/home/valcann/backupsTo"))
DEFAULT_LOG_DIR = Path(os.environ.get("BACKUP_LOGDIR",
"/home/valcann/"))
DEFAULT_DAYS = int(os.environ.get("BACKUP_DAYS", "3"))

# Nomes de log
FROM_LOG_NAME = "backupsFrom.log"
TO_LOG_NAME = "backupsTo.log"

# Formato de data/hora nos logs
TIME_FORMAT = "%Y-%m-%d %H:%M:%S"


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser(description="Gerencia backups
conforme política de retenção.")
    parser.add_argument("--source", type=Path, default=DEFAULT_SOURCE,
help="Diretório origem dos arquivos (backupsFrom)")
    parser.add_argument("--dest", type=Path, default=DEFAULT_DEST,
help="Diretório destino para cópia (backupsTo)")
```

```
    parser.add_argument("--log-dir", type=Path,
default=DEFAULT_LOG_DIR, help="Diretório onde ficam os logs")
    parser.add_argument("--days", type=int, default=DEFAULT_DAYS,
help="Quantidade de dias limite (<= mantém, > remove)")
    parser.add_argument("--time-field", choices=["ctime", "mtime"],
default="ctime", help="Campo de tempo usado para critério de idade")
    parser.add_argument("--dry-run", action="store_true", help="Simula
sem remover/copiar arquivos")
    parser.add_argument("--append-logs", action="store_true",
help="Acrescenta aos logs em vez de sobrescrever")
    return parser.parse_args()


def ensure_dirs(*paths: Path) -> None:
    for p in paths:
        p.mkdir(parents=True, exist_ok=True)


def collect_file_info(path: Path) -> Dict[str, str]:
    st = path.stat()
    creation_time = datetime.fromtimestamp(st.st_ctime)
    mod_time = datetime.fromtimestamp(st.st_mtime)
    return {
        "path": str(path),
        "name": path.name,
        "size": str(st.st_size),
        "ctime": creation_time.strftime(TIME_FORMAT),
        "mtime": mod_time.strftime(TIME_FORMAT),
        "st_ctime": st.st_ctime,
        "st_mtime": st.st_mtime,
    }


def list_files(directory: Path) -> List[Path]:
    files: List[Path] = []
    for entry in os.scandir(directory):
        if entry.is_file():
            files.append(Path(entry.path))
    return files


def format_listing_line(info: Dict[str, str]) -> str:
    return (f"Nome: {info['path']}, Tamanho: {info['size']} bytes, "
```

```
            f"Criação: {info['ctime']}, Modificação: {info['mtime']}")


def write_listing_log(files: List[Path], log_path: Path, mode: str) ->
None:
    now_str = datetime.now().strftime(TIME_FORMAT)
    with log_path.open(mode, encoding="utf-8") as f:
        f.write(f"Log de arquivos em {log_path.parent} gerado em
{now_str}\n\n")
        count = 0
        for file in files:
            try:
                info = collect_file_info(file)
                f.write(format_listing_line(info) + "\n")
                count += 1
            except Exception as e:  # pragma: no cover (falhas raras de
FS)
                f.write(f"ERRO lendo '{file}': {e}\n")
        f.write(f"\nTotal listado: {count}\n")


def classify_files(files: List[Path], days: int, time_field: str) ->
Tuple[List[Path], List[Path]]:
    threshold = datetime.now() - timedelta(days=days)
    recent: List[Path] = []
    old: List[Path] = []
    for f in files:
        try:
            st = f.stat()
            ref_ts = st.st_ctime if time_field == "ctime" else
st.st_mtime
            file_time = datetime.fromtimestamp(ref_ts)
            if file_time < threshold:
                old.append(f)
            else:
                recent.append(f)
        except Exception:
            recent.append(f)
    return recent, old


def remove_files(files: List[Path], dry_run: bool) -> Tuple[int,
List[str]]:
```

```
    removed = 0
    errors: List[str] = []
    for f in files:
        try:
            if dry_run:
                continue
            f.unlink(missing_ok=True)
            removed += 1
        except Exception as e:
            errors.append(f"Erro removendo {f}: {e}")
    return removed, errors


def copy_files(files: List[Path], dest: Path, dry_run: bool) ->
Tuple[int, List[Dict[str, str]], List[str]]:
    copied_infos: List[Dict[str, str]] = []
    errors: List[str] = []
    copied = 0
    for f in files:
        try:
            if not dry_run:
                shutil.copy2(f, dest)
            info = collect_file_info(f)
            copied_infos.append(info)
            copied += 1
        except Exception as e:
            errors.append(f"Erro copiando {f}: {e}")
    return copied, copied_infos, errors


def write_copy_log(dest_log: Path, mode: str, copied_infos:
List[Dict[str, str]], summary: Dict[str, int], dry_run: bool) -> None:
    now_str = datetime.now().strftime(TIME_FORMAT)
    with dest_log.open(mode, encoding="utf-8") as f:
        f.write(f"Log de cópia gerado em {now_str}\n\n")
        if not copied_infos:
            f.write("Nenhum arquivo recente para copiar.\n")
        else:
            for info in copied_infos:
                f.write("Copiado: " + format_listing_line(info) + ("
(dry-run)" if dry_run else "") + "\n")
        f.write("\nResumo:\n")
        for k, v in summary.items():
```

```
            f.write(f"{k}: {v}\n")


def manage_backups(source: Path, dest: Path, log_dir: Path, days: int,
time_field: str, dry_run: bool, append_logs: bool) -> int:
    if source.resolve() == dest.resolve():
        print("ERRO: Diretório de origem e destino são o mesmo.")
        return 2

    ensure_dirs(source, dest, log_dir)

    from_log = log_dir / FROM_LOG_NAME
    to_log = log_dir / TO_LOG_NAME
    file_mode = "a" if append_logs else "w"

    print(f"Iniciando gerenciamento. SOURCE={source} DEST={dest}
DIAS={days} TIME_FIELD={time_field} DRY_RUN={dry_run}")

    # 1 & 2: Listar + log
    try:
        files = list_files(source)
        write_listing_log(files, from_log, file_mode)
        print(f"Log de listagem salvo em {from_log}")
    except Exception as e:
        print(f"Falha ao listar: {e}")
        return 1

    # Classificar
    recent, old = classify_files(files, days, time_field)

    # Remover antigos
    removed_count, remove_errors = remove_files(old, dry_run)
    for err in remove_errors:
        print(err)

    # Copiar recentes
    copied_count, copied_infos, copy_errors = copy_files(recent, dest,
dry_run)
    for err in copy_errors:
        print(err)

    summary = {
        "Arquivos_listados": len(files),
```

```
            "Antigos_para_remover": len(old),
            "Removidos": removed_count,
            "Recentes_para_copiar": len(recent),
            "Copiados": copied_count,
            "Dry_run": 1 if dry_run else 0,
            "Erros_remocao": len(remove_errors),
            "Erros_copia": len(copy_errors),
        }

        # Log de cópia (5)
        write_copy_log(to_log, file_mode, copied_infos, summary, dry_run)
        print(f"Log de cópia salvo em {to_log}")

        if remove_errors or copy_errors:
            print("Concluído com avisos/erros.")
            return 3

        print("Processo finalizado com sucesso!")
        return 0


def main() -> None:
    args = parse_args()
    exit_code = manage_backups(
        source=args.source,
        dest=args.dest,
        log_dir=args.log_dir,
        days=args.days,
        time_field=args.time_field,
        dry_run=args.dry_run,
        append_logs=args.append_logs,
    )
    sys.exit(exit_code)


if __name__ == "__main__":
    main()
```

Solução do Problema 2: Monitoramento e Performance

A análise a seguir apresenta um plano detalhado para diagnosticar e resolver um problema complexo de lentidão em uma aplicação web, conforme solicitado. A abordagem segue a estrutura de Problema > Causa > Solução e é projetada para identificar gargalos que não são aparentes em um monitoramento superficial de infraestrutura.

Problema

Um cliente relata lentidão contínua em uma de suas principais aplicações web. A infraestrutura é composta por 4 servidores de aplicação e 2 servidores de banco de dados em um esquema de replicação (um nó de escrita e um de leitura). Uma análise preliminar dos indicadores de monitoramento, como uso de CPU e memória, não revelou qualquer sinal de sobrecarga ou pico de utilização nos servidores. Apesar da aparente saúde da infraestrutura, a aplicação continua lenta para os usuários finais.

Causa (Plano de Diagnóstico)

O fato de os recursos de infraestrutura estarem normalizados sugere que a causa raiz da lentidão é um gargalo de performance mais sutil. Proponho um plano de diagnóstico sistemático e end-to-end, investigando o sistema em quatro camadas críticas:

1. Camada de Frontend (Experiência Real do Usuário - RUM)

A investigação deve começar na ponta, onde a lentidão é percebida. O objetivo é quantificar a experiência do usuário.

- **O que investigar:** Em vez de suposições, mediremos a performance real no navegador. Analisaremos métricas de Core Web Vitals (LCP, FID, CLS) para entender se a lentidão vem do carregamento de ativos pesados (imagens, JS), da renderização da página ou de scripts ineficientes.
- **Ferramentas:** Google Lighthouse, DevTools do navegador e, idealmente, uma ferramenta de RUM (Real User Monitoring) para coletar métricas de performance de usuários reais.

2. Camada de Aplicação (Backend)

Esta é a camada mais provável para gargalos lógicos. Uma análise profunda do comportamento da aplicação é essencial.

- **O que investigar:**
    - **Latência de Cauda Longa (p95/p99):** A latência média pode mascarar problemas. Analisar os percentis 95 e 99 revela a experiência dos usuários

mais afetados, que é onde os problemas de performance se tornam mais visíveis.
    - **Queries N+1:** Usando Distributed Tracing (Rastreamento Distribuído), podemos identificar padrões ineficientes de acesso a dados, como uma query que é executada repetidamente dentro de um loop, quando poderia ser substituída por uma única consulta mais eficiente.
- **Ferramentas:** Implementação de uma solução de APM (Application Performance Monitoring) como Datadog, New Relic ou uma stack open-source com OpenTelemetry e Jaeger.

3. Camada de Banco de Dados

Mesmo com baixo uso de CPU, o banco de dados pode ser a fonte da lentidão por razões lógicas.

- **O que investigar:**
    - **Pool de Conexões:** Um esgotamento do pool de conexões com o banco de dados fará com que a aplicação espere por uma conexão disponível, causando lentidão geral sem sobrecarregar o banco.
    - **Locks e Waits:** Analisar a existência de transações que geram bloqueios (locks) em tabelas ou registros por tempo excessivo, criando uma "fila" de operações.
    - **Plano de Execução de Queries:** Usar EXPLAIN ANALYZE nas consultas mais frequentes para garantir que elas estejam utilizando índices e não realizando varreduras completas e ineficientes na tabela (full table scans).
- **Ferramentas:** Logs de Slow Query do banco, painéis de monitoramento do banco e o comando EXPLAIN ANALYZE.

4. Camada de Dependências Externas

Aplicações modernas raramente operam isoladamente. A lentidão pode ser causada por serviços de terceiros.

- **O que investigar:** O tempo de resposta de APIs externas (ex: gateways de pagamento, serviços de autenticação, etc.). Um timeout mal configurado ou uma falha em um desses serviços pode paralisar as requisições na nossa aplicação.
- **Ferramentas:** As mesmas ferramentas de APM e Distributed Tracing são perfeitas para visualizar o tempo gasto em chamadas externas e identificar qual dependência está lenta.

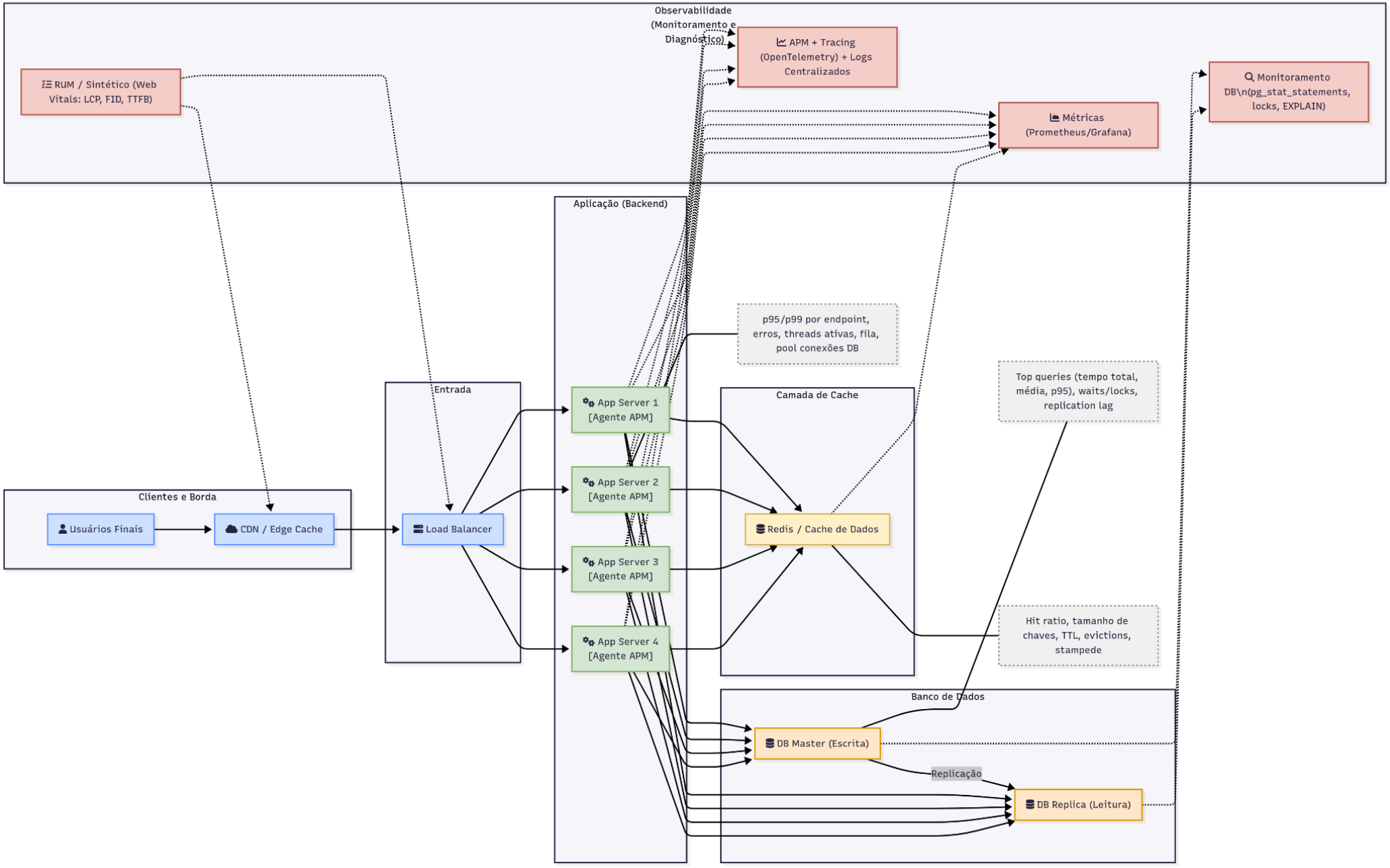
Solução

A solução será direcionada pela causa raiz identificada no diagnóstico. As ações podem incluir:

- **Causa no Frontend:** Otimizar imagens e scripts e implementar uma CDN (Content Delivery Network) para acelerar a entrega de ativos estáticos.
- **Causa na Aplicação:**
    - Refatorar o código para eliminar gargalos, como o problema de N+1 queries.
    - Implementar um Cache em Memória (ex: Redis) para armazenar resultados de operações custosas e reduzir a carga no banco de dados.
    - Para dependências externas, implementar o padrão Circuit Breaker, que impede que a aplicação continue tentando chamar um serviço que está falhando, melhorando a resiliência do sistema.
- **Causa no Banco de Dados:**
    - Otimizar ou criar índices para acelerar as consultas lentas.
    - Ajustar o tamanho do pool de conexões na aplicação para atender à demanda sem gerar filas.

Diagrama de Arquitetura da Solução

Arquitetura de Monitoramento e Performance



Solução do Problema 3: Aplicações e Desenvolvimento de Software

Esta análise detalha as ações e ferramentas para automatizar o processo de build e deploy de uma aplicação, conforme solicitado, transformando um fluxo manual em um pipeline de CI/CD (Continuous Integration/Continuous Deployment) moderno, ágil e seguro. A solução segue a estrutura de Problema > Causa > Solução.

Problema

A empresa possui uma aplicação com backend em Node.JS e frontend em React cujo processo de lançamento de novas versões é inteiramente manual. Para cada atualização, a equipe precisa primeiro empacotar manualmente todos os componentes de frontend e backend para então realizar o deploy no ambiente de homologação. Após um ciclo de uma semana de validações, um novo empacotamento manual é realizado antes da atualização do ambiente de produção. Este fluxo é lento, propenso a erros humanos e consome um tempo valioso da equipe de desenvolvimento.

Causa

A causa raiz do problema é a ausência total de um pipeline de automação (CI/CD) e de uma cultura DevOps. Essa carência resulta em várias ineficiências e riscos críticos:

- **Processo Manual e Sujeito a Erros:** A intervenção humana em cada etapa pode introduzir inconsistências e falhas que são difíceis de rastrear e depurar.
- **Ciclo de Feedback Lento:** O intervalo de uma semana entre o deploy de homologação e produção atrasa a entrega de valor aos clientes e a capacidade de corrigir bugs rapidamente.
- **Falta de Padronização:** A ausência de um processo repetível pode levar a diferenças de configuração entre os ambientes de homologação e produção, resultando em comportamentos inesperados da aplicação.

Solução

A solução é a implementação de um ecossistema de entrega de software automatizado, que abrange desde a submissão do código até o deploy em produção. A estratégia é dividida em cinco pilares fundamentais para garantir qualidade, velocidade e segurança.

1. Fundação: Pipeline de CI/CD e Containerização

- **Ação:** Centralizar o código em um repositório Git e criar um pipeline automatizado com uma ferramenta como GitHub Actions ou GitLab CI. Para garantir consistência

entre todos os ambientes, vamos containerizar a aplicação com Docker, criando imagens separadas para o frontend (React servido por Nginx) e o backend (Node.js).

- **Resultado:** Cada alteração no código dispara automaticamente o build, os testes e a criação de um artefato (imagem Docker) versionado e imutável, pronto para deploy.

2. Qualidade Automatizada e Contínua

- **Ação:** Integrar diretamente no pipeline etapas de verificação de qualidade. Isso inclui testes unitários e de integração automatizados, análise estática de código com linters (ESLint) e verificação de dependências vulneráveis.
- **Resultado:** O pipeline atua como um “guardião da qualidade’, bloqueando automaticamente qualquer código que não atenda aos critérios mínimos antes de ser implantado.

3. Consistência de Ambientes com Infraestrutura como Código (IaC)

- **Ação:** Para eliminar o problema de "funciona na minha máquina", os ambientes de homologação e produção serão provisionados e gerenciados via Infraestrutura como Código com ferramentas como Terraform ou Ansible.
- **Resultado:** Os ambientes se tornam idênticos e reprodutíveis, eliminando bugs que só aparecem em produção devido a diferenças de configuração.

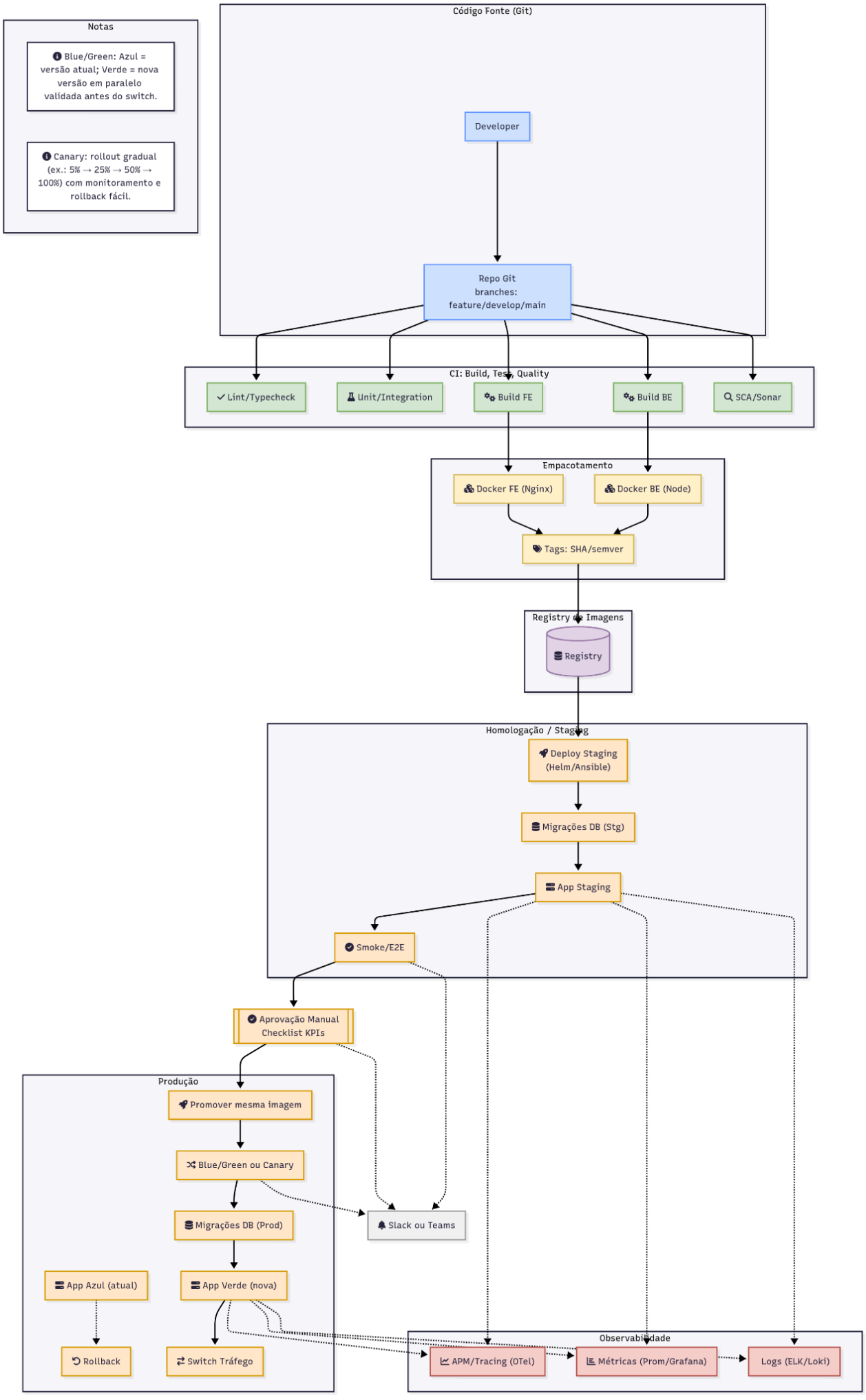
4. Deploys Seguros e Confiáveis

- **Ação:** O deploy em homologação será totalmente automático após o sucesso do CI. O deploy em produção exigirá uma aprovação manual para manter o controle de negócios. Utilizaremos uma estratégia de deploy segura como Blue-Green para permitir atualizações sem tempo de inatividade e com capacidade de rollback instantâneo.
- **Resultado:** O risco de um deploy impactar negativamente os usuários é drasticamente reduzido.

5. Gerenciamento Automatizado do Banco de Dados

- **Ação:** As migrações de esquema do banco de dados serão gerenciadas com uma ferramenta de migração (ex: Prisma Migrate, TypeORM) e executadas como uma etapa automatizada e idempotente dentro do pipeline de deploy.
- **Resultado:** O banco de dados evolui de forma sincronizada e segura com a aplicação, evitando falhas de deploy.

Diagrama de Arquitetura da Solução


Notas
Blue/Green: Azul = versão atual; Verde = nova versão em paralelo validada antes do switch.
Canary: rollout gradual (ex.: 5% → 25% → 50% → 100%) com monitoramento e rollback fácil.
Código Fonte (Git)
Developer
Repo Git
branches:
feature/develop/main
CI: Build, Test, Quality
Lint/Typecheck
Unit/Integration
Build FE
Build BE
SCA/Sonar
Empacotamento
Docker FE (Nginx)
Docker BE (Node)
Tags: SHA/semver
Registry de Imagens
Registry
Homologação / Staging
Deploy Staging
(Helm/Ansible)
Migrações DB (Stg)
App Staging
Smoke/E2E
Aprovação Manual
Checklist KPIs
Produção
Promover mesma imagem
Blue/Green ou Canary
Migrações DB (Prod)
Slack ou Teams
App Azul (atual)
App Verde (nova)
Rollback
Switch Tráfego
Observabilidade
APM/Tracing (OTel)
Métricas (Prom/Grafana)
Logs (ELK/Loki)