Templates de CI/CD

Um Passo Antes da Plataforma: Padronização como Fundação para Engenharia Escalável

Milton Jesus
Platform Engineer

Quem sou Eu

Me chamo **Milton** e atuo como **Platform Engineer** na **Bemobi** uma fintech global**,** sou chapter lead da **CNCF Salvador** e um dos organizadores dos Meetups que acontecem nessa comunidade.

Atualmente estou trabalhando ativamente na construção/refatoramento de uma IDP usando várias ferramentas do ecossistema CNCF , com o Crossplane, Keda, External Secrets e muitas outras.

“Alguns escrevem poemas, eu escrevo YAML’s “ ❤️

O Problema

Cada time cria o próprio pipeline. Um usa Node 18, o outro usa 16. Um faz scan de segurança, o outro esquece. Um deploy funciona… o outro não !

Estágio 1

Pipelines Manuais

Cada equipe cria e mantém seus próprios pipelines de forma independente

↓

Estágio 2

Templates Padronizados

Implementação de templates reutilizáveis que padronizam processos e estabelecem governança

↓

Estágio 3

Catálogo Self-Service

Desenvolvedores descobrem e consomem templates de forma autônoma

↓

Estágio 4

IDP Completa com GitOps

Plataforma de engenharia interna que abstrai complexidades de infraestrutura

O Caos

Pipelines cada vez mais complexos

Pressão por velocidade

Ausência de governança

Pouca preocupação com segurança

Repetição Infinita

Cada novo projeto criava-se criação pipelines do zero, consumindo dias até semanas de engenharia.

Retrabalho Constante

Ausência de padrões forçava equipes a **revisitar repetidamente** os mesmos erros.

Falta de Comunicação.

Diferentes equipes implementam soluções divergentes para **problemas idênticos**.

Solução ??

Templates de Pipeline

E Por que usar Templates ?

Reuso

Encapsula melhores práticas reutilizáveis automaticamente

Governança

Políticas aplicadas consistentemente sem intervenção manual

Aceleração de Entrega

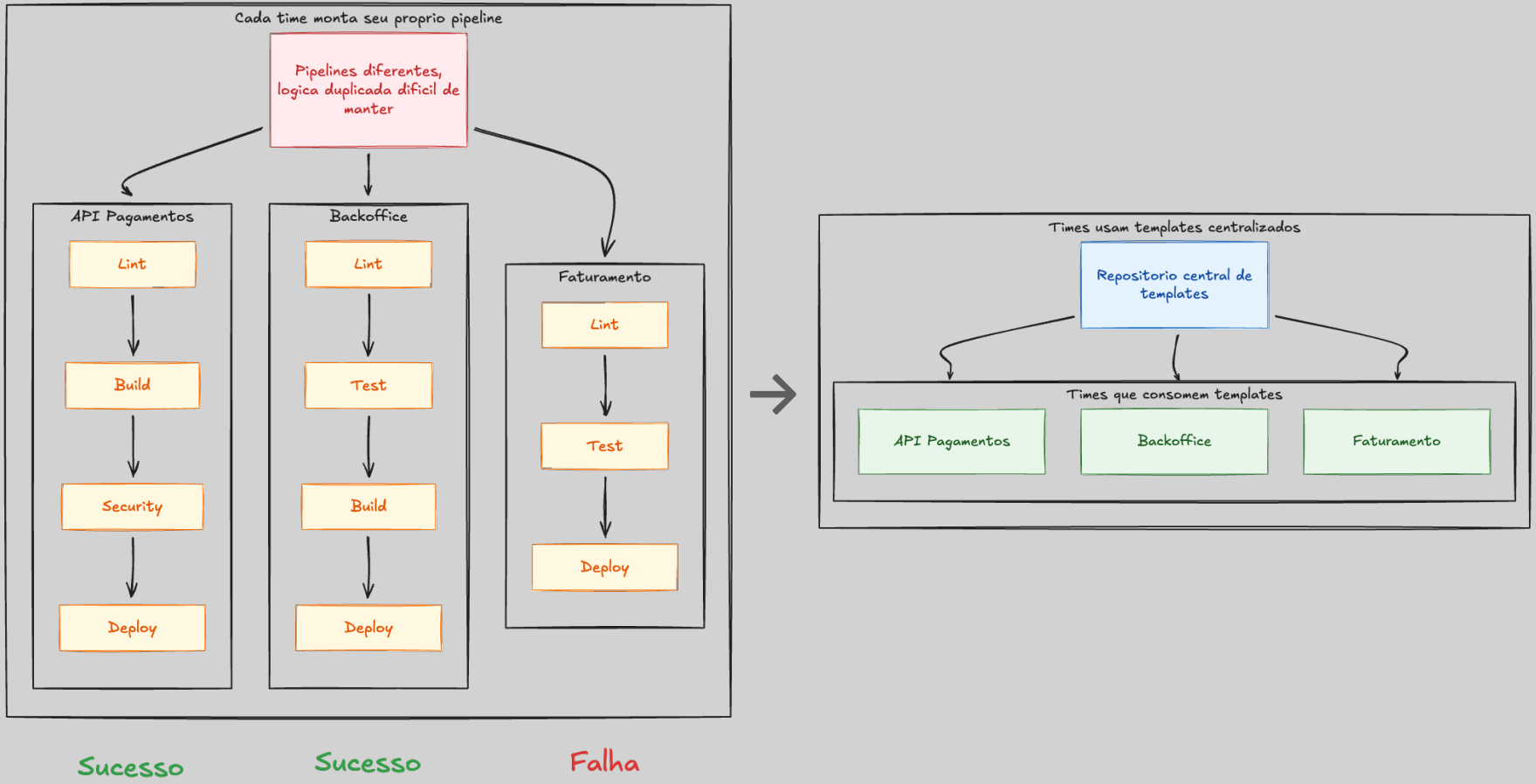
Novos projetos iniciam com pipelines funcionais imediatamente

Redução de Riscos

Minimiza erros humanos e superfície de ataque

Consistência

Todos os projetos seguem os mesmos padrões de qualidade


Cada time monta seu proprio pipeline
Pipelines diferentes, logica duplicada dificil de manter
API Pagamentos
Lint
Build
Security
Deploy
Backoffice
Lint
Test
Build
Deploy
Faturamento
Lint
Test
Deploy
Sucesso
Sucesso
Falha
Times usam templates centralizados
Repositorio central de templates
Times que consomem templates
API Pagamentos
Backoffice
Faturamento

```yaml
stages:
  - lint
  - test
  - build
  - security
  - docker
  - deploy

variables:
  NODE_VERSION: “18"
  DOCKER_IMAGE: registry.example.com/app-web
  K8S_NAMESPACE: web

lint:
  stage: lint
  image: node:${NODE_VERSION}
  script:
    - echo "Instalando dependências"
    - npm ci
    - echo "Executando lint"
    - npm run lint
  only:
    - main
    - merge_requests

test_unit:
  stage: test
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - npm run test:unit
  artifacts:
    paths:
      - reports/unit.xml
    when: always
  only:
    - main
    - merge_requests

test_integration:
  stage: test
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - npm run test:integration
  artifacts:
    paths:
      - reports/integration.xml
    when: always
  only:
    - main
    - merge_requests

build_frontend:
  stage: build
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - npm run build
  artifacts:
    paths:
      - dist
  only:
    - main
    - merge_requests

security_scan_deps:
  stage: security
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - echo "Executando audit"
    - npm audit --production
  allow_failure: true
  only:
    - main
    - merge_requests

security_scan_container:
  stage: security
  image: aquasec/trivy:latest
  script:
    - trivy fs . --severity CRITICAL,HIGH --exit-code 0
  allow_failure: true
  only:
    - main
    - merge_requests

docker_build:
  stage: docker
  image: docker:24
  services:
    - docker:24-dind
  script:
    - echo "Fazendo login no registry"
    - docker login -u "$CI_REGISTRY_USER" -p
"$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"

    - echo "Buildando imagem"
    - docker build -t "$DOCKER_IMAGE:$CI_COMMIT_SHA" .

    - echo "Aplicando tag latest"
    - docker tag "$DOCKER_IMAGE:$CI_COMMIT_SHA"
"$DOCKER_IMAGE:latest"

    - echo "Enviando imagem"
    - docker push "$DOCKER_IMAGE:$CI_COMMIT_SHA"
    - docker push "$DOCKER_IMAGE:latest"
  only:
    - main

deploy_k8s:
  stage: deploy
  image: bitnami/kubectl:1.29
  script:
    - echo "Configurando kubectl"
    - kubectl config set-cluster cluster --server="$K8S_API"
    - kubectl config set-credentials gitlab --token="$K8S_TOKEN"
    - kubectl config set-context ctx --cluster=cluster --
user=gitlab --namespace="$K8S_NAMESPACE"
    - kubectl config use-context ctx

    - echo "Aplicando nova imagem"
    - kubectl set image deployment/app-web app-
web=“$DOCKER_IMAGE:$CI_COMMIT_SHA"
    - echo "Verificando rollout"
    - kubectl rollout status deployment/app-web
  environment:
    name: production
  only:
    - main
```

```yaml
stages: # Stages definidos aqui, mas lógica repetida em cada job
  - lint
  - test
  - build
  - security
  - docker
  - deploy

variables:
  NODE_VERSION: “18" # Versão de runtime acoplada ao pipeline
  DOCKER_IMAGE: registry.example.com/app-web
  K8S_NAMESPACE: web

lint:
  stage: lint
  image: node:${NODE_VERSION}
  script:
    - echo "Instalando dependências"
    - npm ci
    - echo "Executando lint"
    - npm run lint
  only:
    - main
    - merge_requests

test_unit:
  stage: test
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - npm run test:unit
  artifacts:
    paths:
      - reports/unit.xml
    when: always
  only:
    - main
    - merge_requests

test_integration:
  stage: test
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - npm run test:integration
  artifacts:
    paths:
      - reports/integration.xml
    when: always
  only:
    - main
    - merge_requests

build_frontend:
  stage: build
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - npm run build
  artifacts:
    paths:
      - dist
  only:
    - main
    - merge_requests

security_scan_deps:
  stage: security
  image: node:${NODE_VERSION}
  script:
    - npm ci
    - echo "Executando audit"
    - npm audit --production
  allow_failure: true
  only:
    - main
    - merge_requests

security_scan_container:
  stage: security
  image: aquasec/trivy:latest # Ferramenta e versão de segurança definidas no app
  script:
    - trivy fs . --severity CRITICAL,HIGH --exit-code 0
  allow_failure: true
  only:
    - main
    - merge_requests

docker_build:
  stage: docker
  image: docker:24
  services:
    - docker:24-dind
  script:
    - echo "Fazendo login no registry"
    - docker login -u "$CI_REGISTRY_USER" -p "$CI_REGISTRY_PASSWORD" "$CI_REGISTRY"

    - echo "Buildando imagem"
    - docker build -t "$DOCKER_IMAGE:$CI_COMMIT_SHA" . # Cada projeto inventa seu próprio padrão de build/tag

    - echo "Aplicando tag latest"
    - docker tag "$DOCKER_IMAGE:$CI_COMMIT_SHA" "$DOCKER_IMAGE:latest"

    - echo "Enviando imagem"
    - docker push "$DOCKER_IMAGE:$CI_COMMIT_SHA"
    - docker push "$DOCKER_IMAGE:latest"
  only:
    - main

deploy_k8s:
  stage: deploy
  image: bitnami/kubectl:1.29
  script:
    - echo "Configurando kubectl"
    - kubectl config set-cluster cluster --server="$K8S_API"
    - kubectl config set-credentials gitlab --token="$K8S_TOKEN"
    - kubectl config set-context ctx --cluster=cluster --user=gitlab --namespace="$K8S_NAMESPACE"
    - kubectl config use-context ctx

    - echo "Aplicando nova imagem"
    - kubectl set image deployment/app-web app-web=“$DOCKER_IMAGE:$CI_COMMIT_SHA" # Deploy imperativo, frágil e específico demais

    - echo "Verificando rollout"
    - kubectl rollout status deployment/app-web
  environment:
    name: production
  only:
    - main
```

Estrutura proposta repositório de templates

```
ci-templates/
├── README.md
├── CHANGELOG.md
├── .gitlab-ci.yml              # <-- define os stages globais
├── templates/
│   ├── ci/                     # <-- templates de Continuous Integration
│   │   ├── node_ci.yml         # lint, test_unit, test_integration, build
│   │   └── go_ci.yml           #
│   │
│   ├── security/               # <-- templates de segurança
│   │   ├── node_security.yml   # npm audit, scans de dependências
│   │   └── container_security.yml   # ferramentas de sca, trivy e etc.
│   │
│   ├── shared/                 # <-- templates utilizados por qualquer stack
│   │   ├── docker_build.yml    # docker build + docker push
│   │   ├── docker_push.yml     # (opcional)
│   │   ├── cache.yml           # (opcional)
│   │   └── notify_slack.yml    # (opcional)
│   │
│   └── cd/                     # <-- templates para deploy / delivery
│       └── deploy_k8s.yml
├── versions/                   # <-- versionamento das releases dos templates
│   ├── v1.0.0/
│   │   ├── manifest.json
│   │   └── notes.md
│   ├── v1.1.0/
│   │   ├── manifest.json
│   │   └── notes.md
│   └── v2.0.0/
│       ├── manifest.json
│       └── notes.md
└── tests/                      # <-- projetos mock para validar templates
    ├── node_test_project/
    │   └── .gitlab-ci.yml      # pipeline que usa os templates para testes
    └── go_test_project/
        └── .gitlab-ci.yml
```

```yaml
include:
  - project: "platform/ci-templates"             # Repositório central mantido pelo time de Plataforma
    ref: "v1.0.0"                                # Versão dos templates (imutável, versionada)
    file:                                        # Templates específicos usados por este serviço
      - "/templates/ci/node_ci.yml"              # Templates de Lint, Test e Build para Node
      - "/templates/security/node_security.yml"  # Templates de auditoria e segurança
      - "/templates/shared/docker_build.yml"     # Template padronizado de build/push Docker
      - "/templates/cd/deploy_k8s.yml"           # Template padronizado de deploy em Kubernetes

# ---------------------------------------------------------------
# VARIABLES - Variáveis específicas da aplicação
# ---------------------------------------------------------------
variables:
  NODE_VERSION: "18"                           # Versão do runtime usada pelos templates
  DOCKER_IMAGE: "registry.example.com/app-web" # Imagem final do serviço
  K8S_NAMESPACE: "web"                         # Namespace onde o deploy será aplicado
  K8S_API: $K8S_API_URL                        # Endpoint do cluster Kubernetes (via env)
  K8S_TOKEN: $K8S_GITLAB_TOKEN                 # Token usado pelos templates de deploy
  HELM_CHART: "./chart"                        # Caminho do chart Helm (se usado no template)
  HELM_RELEASE: "app-web"                      # Nome do release no cluster

# ---------------------------------------------------------------
# JOBS - Cada job apenas herda um template, O serviço NÃO implementa lógica de CI/CD aqui, Toda a lógica real vive nos templates da plataforma.
# ---------------------------------------------------------------

lint:
  extends: .node_lint                        # Usa o template de Lint para Node
test_unit:
  extends: .node_test_unit                   # Testes unitários padronizados
test_integration:
  extends: .node_test_integration            # Testes de integração padronizados
build_frontend:
  extends: .node_build                       # Build padronizado (npm ci, build, artifacts)
security_scan_deps:
  extends: .node_security_deps               # Auditoria de dependências (npm audit)
security_scan_container:
  extends: .container_security_scan          # Scan de imagem com ferramenta corporativa
docker_build:
  extends: .docker_build_and_push            # Build e push de imagem Docker padronizado
deploy_k8s:
  extends: .deploy_k8s                       # Deploy em Kubernetes com padrão corporativo
```

Obstáculos

Desafios		Como superamos
Resistência Cultural Desenvolvedores preferem manter controle total sobre suas configurações	→	**Demonstração de Valor** Mostre ROI claro através de casos de uso reais e métricas de impacto
Falta de Documentação Templates sem documentação clara tornam-se barreiras em vez de facilitadores	→	**Documentação Abrangente** Exemplos práticos, guias de troubleshooting e FAQ detalhado
Templates Rígidos Falta de flexibilidade leva à criação de soluções paralelas	→	**Flexibilidade Controlada** Permita customização adequada mantendo padrões essenciais
Ausência de Ownership Templates sem responsáveis rapidamente tornam-se obsoletos	→	**Ownership Claro** Defina responsáveis pela manutenção e evolução dos templates

Objetivos Alcançados

~70%

Menos tempo criando novos pipelines. Equipes agora focam em lógica de negócio, não em configuração.

∞

Vulnerabilidades evitadas através de padrões de segurança incorporados. Scanning automático em todos os builds.

96%

Governança centralizada. Políticas aplicadas consistentemente sem intervenção manual em cada projeto.

↓

Redução de retrabalho e dedicação exclusiva

Aprendizados

Recomendações práticas para implementar templates com sucesso

Comece com 2 Templates

Não tente padronizar tudo de uma vez. Escolha **2 templates iniciais** que resolvam os maiores problemas.

Documente Tudo

Documentação é **tão importante quanto o código**. Exemplos, guias e troubleshooting são essenciais.

Construa a Plataforma Sobre Essa Base

Templates bem estruturados são o **alicerce de uma IDP**. Não pule este passo.

Evolua com Governança

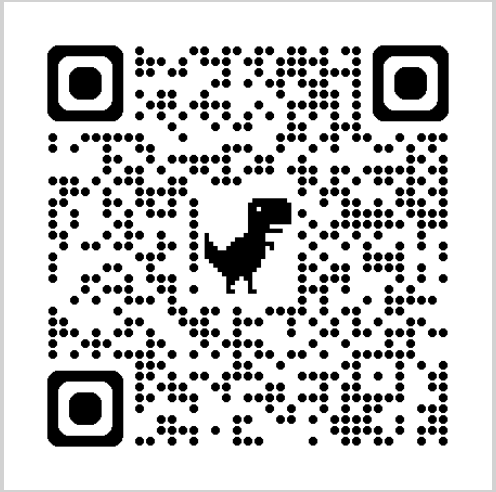
Incremente templates gradualmente, **sempre com governança** e feedback de usuários.

Versione Adequadamente

Use **versionamento semântico**. Permita que equipes adotem versões de forma controlada.

Adote a Colaboração

Incentive a colaboração entre times na evolução dos templates, estratégias como Inner Source funcionam muito bem.



Obrigado!