



GUIDE PRATIQUE · SÉCURITÉ APPLICATIVE

DevSecOps

Le guide complet

Principes, architecture, flux, performance, sécurité et mise en place

Frédéric PEREZ

Édition 2026

Sommaire

01	Introduction : qu'est-ce que le DevSecOps ?	3
02	Principes fondamentaux	5
03	Architecture de référence	7
04	Flux détaillé du pipeline sécurisé	9
05	Outillage par catégorie	13
06	Sécuriser la chaîne d'approvisionnement logicielle	15
07	Performance : sécurité sans ralentir la livraison	17
08	Prérequis	19
09	Mise en place : feuille de route	21
10	Exemples de pipelines	24
11	Gouvernance, maturité et pièges	28
12	Conclusion et références	30

Ce guide explique comment intégrer la sécurité à chaque étape de la livraison logicielle : de l'IDE à la production, avec schémas d'architecture et de flux, choix d'outils, impact sur la performance, prérequis, feuille de route et exemples de pipelines prêts à adapter.

Introduction : qu'est-ce que le DevSecOps ?

Le DevSecOps intègre la sécurité dans chaque étape du cycle de livraison logicielle, au lieu de la traiter comme un contrôle final avant la mise en production. La sécurité devient une propriété du pipeline : automatisée, mesurée et partagée entre développeurs, opérations et équipe sécurité.

■ Le problème résolu

Dans un modèle classique, l'audit de sécurité ou le pentest arrive après le développement. Une faille découverte à ce stade coûte cher : il faut rouvrir le code, re-tester, re-déployer, parfois retarder une release. Avec des déploiements quotidiens, ce modèle devient un goulet d'étranglement ou, pire, il est contourné.

■ Le principe du « shift-left »

On déplace les contrôles vers la gauche de la chaîne, c'est-à-dire au plus près de l'écriture du code : dans l'IDE, au commit, à la pull request. Plus une vulnérabilité est détectée tôt, plus elle est simple et peu coûteuse à corriger.

Le DevSecOps ajoute aussi un « shift-right » : surveillance, détection et réponse en production, car tout ne se voit pas avant le déploiement.

■ DevOps et DevSecOps

Dimension	DevOps	DevSecOps
Objectif	Livrer vite et fiable	Livrer vite, fiable et sûr
Place de la sécurité	Contrôle final, équipe séparée	Intégrée à chaque étape, responsabilité partagée

Dimension	DevOps	DevSecOps
Contrôles	Tests fonctionnels, qualité	+ SAST, SCA, secrets, IaC, conteneurs, DAST, signature
Mise en production	Tests verts	Tests verts + politiques de sécurité respectées (quality gates)
Production	Monitoring de disponibilité	+ détection de menaces, protection runtime, réponse à incident
Culture	Dev + Ops	Dev + Sec + Ops, « security champions » dans les équipes

Principes fondamentaux

Sept principes structurent une démarche DevSecOps. Ils sont indépendants des outils : un pipeline outillé sans ces principes reste un empilement de scanners ignorés.

- 1. Shift-left et shift-right.** Détecter tôt (IDE, commit, PR) et surveiller en continu en production.
- 2. Automatisation par défaut.** Tout contrôle répétable est exécuté par le pipeline, pas par un humain. L'humain intervient sur l'exception et l'arbitrage.
- 3. Security as Code.** Politiques, règles de scan, configurations d'infrastructure et exceptions sont versionnées dans Git, revues en PR et auditable.
- 4. Moindre privilège partout.** Pour les humains, les pipelines, les comptes de service et les workloads. Identités fédérées (OIDC) plutôt que secrets statiques.
- 5. Feedback rapide et actionnable.** Un résultat de sécurité doit arriver au développeur en minutes, dans son outil (PR, IDE), avec la correction proposée.
- 6. Quality gates basées sur le risque.** On bloque sur ce qui compte (critique exploitable, secret exposé), on signale le reste. Bloquer tout conduit au contournement.
- 7. Traçabilité de bout en bout.** Du commit signé à l'artefact déployé : on sait qui a modifié quoi, ce qui a été testé, et ce qui tourne en production (SBOM, attestations).

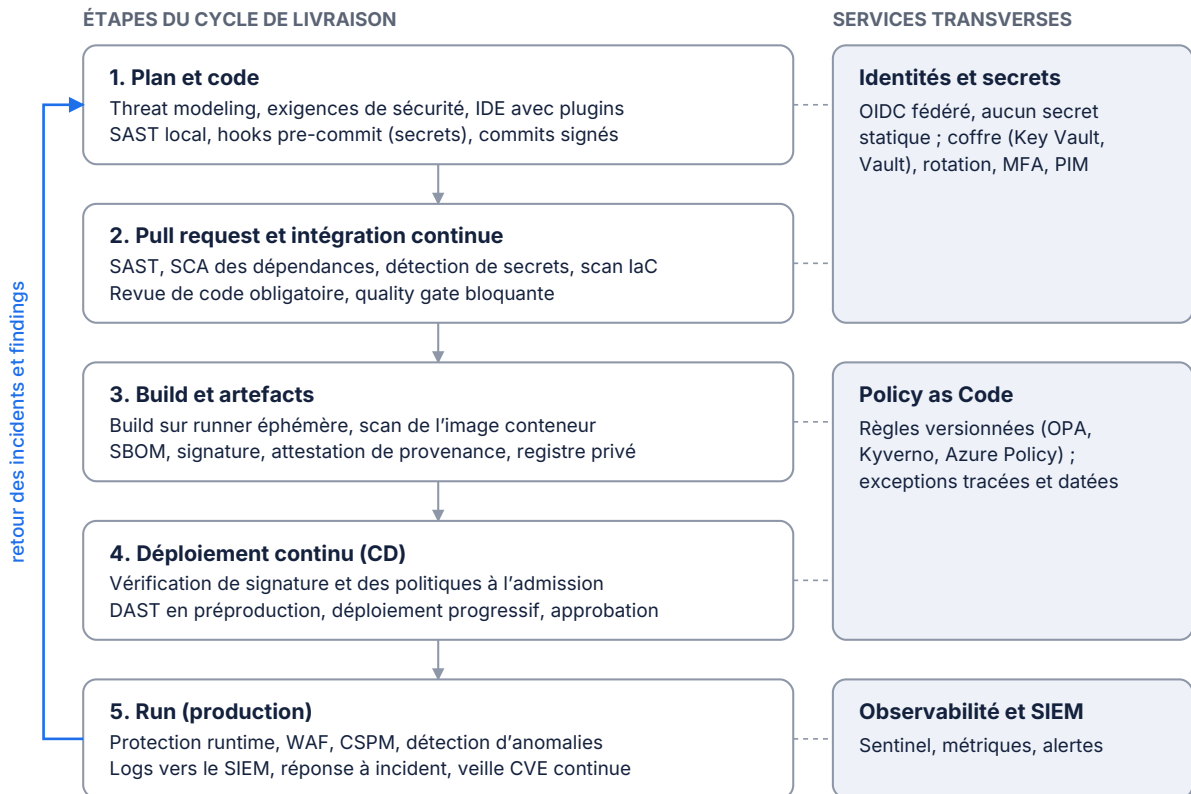
■ Modèle de responsabilité partagée

Acteur	Responsabilités principales
Développeurs	Code sécurisé, correction des findings, revue de code, gestion des dépendances
Security champions	Relais sécurité dans chaque équipe, triage de premier niveau, diffusion des bonnes pratiques
Équipe sécurité (AppSec)	Politiques, choix et réglage des outils, threat modeling, formation, gestion des exceptions

Acteur	Responsabilités principales
Plateforme / DevOps	Pipeline durci, runners isolés, templates réutilisables, gestion des secrets, registres
Opérations / SOC	Détection runtime, réponse à incident, retour d'expérience vers les équipes
Management	Priorisation, budget de remédiation, acceptation formelle des risques résiduels

Architecture de référence

L'architecture DevSecOps s'organise en cinq étapes séquentielles, appuyées par trois services transverses qui s'appliquent à toutes les étapes.



Architecture de référence : 5 étapes, 3 services transverses

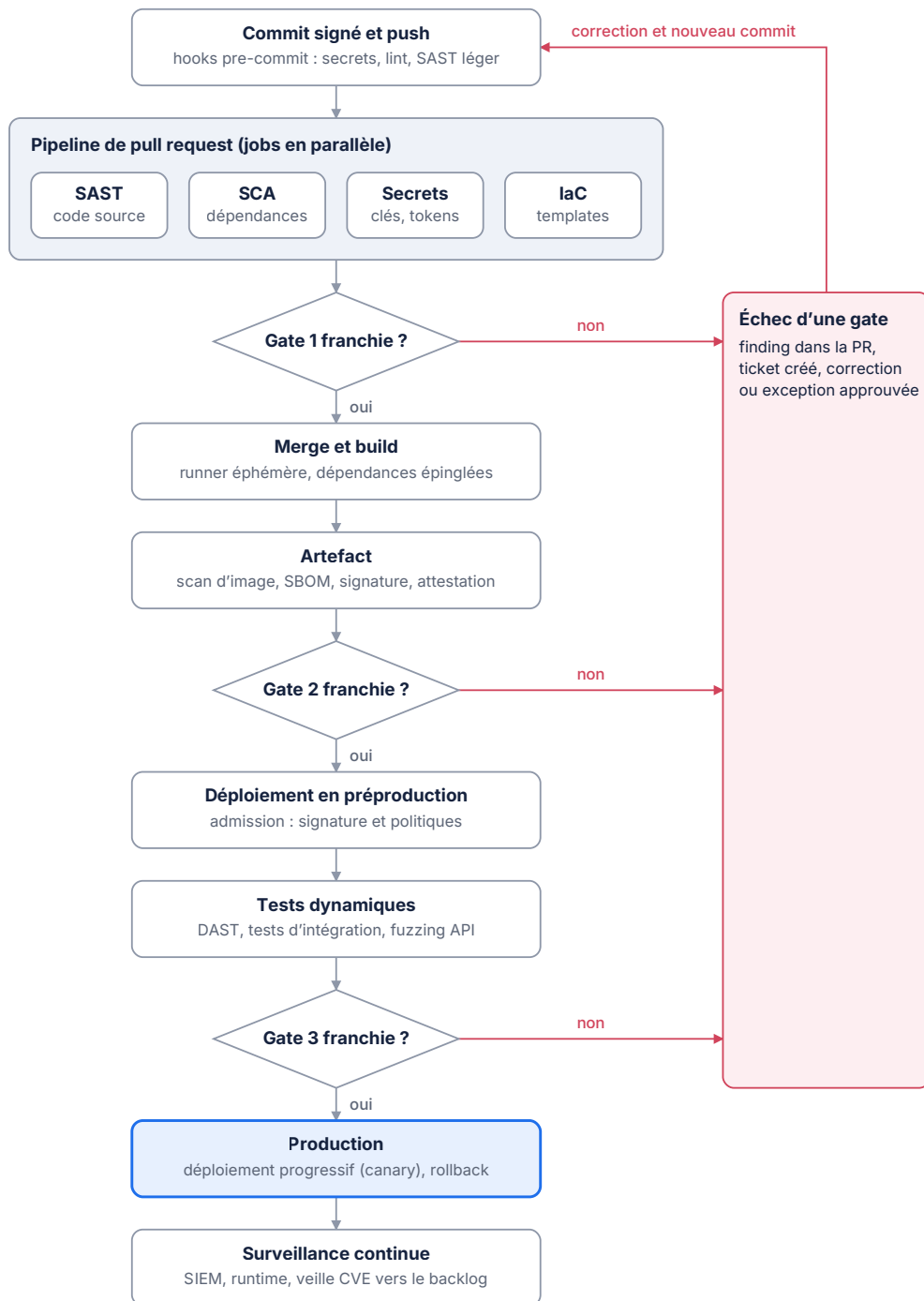
Chaque étape ajoute ses contrôles ; la boucle de gauche renvoie les incidents et vulnérabilités de production vers le backlog et le threat model.

■ Les composants clés

- **Dépôt de code (SCM)** : GitHub, GitLab, Azure Repos. Branches protégées, revue obligatoire, commits signés, CODEOWNERS.
- **Orchestrateur CI/CD** : GitHub Actions, GitLab CI, Azure Pipelines, Jenkins. Pipelines définis en YAML, templates centralisés.
- **Runners** : éphémères (détruits après chaque job), isolés par projet, sans accès réseau inutile.
- **Registre d'artefacts** : ACR, Artifactory, Harbor, GitHub Packages. Images signées, immuables, scannées en continu.
- **Coffre de secrets** : Azure Key Vault, HashiCorp Vault, AWS Secrets Manager. Aucun secret dans le code ni dans les variables en clair.
- **Moteur de politiques** : OPA/Gatekeeper, Kyverno, Azure Policy. Décide ce qui a le droit d'être déployé.
- **Gestion des vulnérabilités (ASPM)** : DefectDojo, Defender for Cloud, Snyk. Agrège, dédoublonne et priorise les findings de tous les outils.
- **Observabilité / SIEM** : Microsoft Sentinel, Splunk, Elastic. Corrèle les événements du pipeline et de la production.

Flux détaillé du pipeline sécurisé

Un changement de code traverse trois quality gates avant d'atteindre la production ; chaque échec renvoie un résultat actionnable au développeur dans la pull request.



Flux du pipeline sécurisé : 3 quality gates

Les contrôles de la PR tournent en parallèle pour limiter la durée ; une exception n'est acceptée que si elle est approuvée, justifiée et datée.

■ Ce que vérifie chaque gate

Gate	Moment	Bloque si	Signale seulement
Gate 1 — PR	Avant merge	Secret détecté ; vulnérabilité critique ou haute introduite par la PR ; laC exposant une ressource publiquement ; revue non approuvée	Findings moyens et faibles, dette existante
Gate 2 — Artefact	Après build	CVE critique avec correctif disponible ; image de base non autorisée ; SBOM ou signature absente	CVE sans correctif disponible
Gate 3 — Préprod	Avant production	Vulnérabilité DAST haute confirmée ; politique d'admission violée ; approbation manquante pour un changement sensible	Alertes DAST informatives

■ Déroulé étape par étape

- 1. Poste du développeur.** L'IDE signale les failles en temps réel (plugin SAST, linter sécurité). Un hook pre-commit bloque les secrets avant qu'ils quittent la machine. Le commit est signé (GPG, SSH ou Sigstore gitsign).
- 2. Pull request.** Le pipeline lance en parallèle SAST, SCA, détection de secrets et scan laC. Les résultats sont publiés en annotations sur les lignes concernées. Seuls les findings introduits par la PR (diff) sont bloquants.
- 3. Merge et build.** Le merge n'est possible que sur branche protégée, avec revue approuvée et gate 1 verte. Le build s'exécute sur un runner éphémère, avec dépendances verrouillées (lockfile, digest).
- 4. Artefact.** L'image est scannée, un SBOM est généré (CycloneDX ou SPDX), l'artefact est signé et une attestation de provenance est produite, puis poussée dans un registre privé.
- 5. Préproduction.** Le contrôleur d'admission vérifie signature, provenance et politiques (pas de conteneur root, ressources limitées, image autorisée). Le DAST et les tests d'API tournent sur l'environnement déployé.
- 6. Production.** Déploiement progressif (canary, blue/green) avec rollback automatique sur alerte. Les changements sensibles passent par une approbation humaine.

7. Surveillance. Les logs d'exécution et du pipeline partent vers le SIEM. Une nouvelle CVE publiée sur un composant du SBOM déclenche une alerte, même sans nouveau déploiement.

Outillage par catégorie

Chaque catégorie d'outil couvre un type de risque différent ; aucune ne suffit seule. Commencer par secrets + SCA + SAST donne le meilleur rapport effort / réduction de risque.

Catégorie	Ce qui est analysé	Risques couverts	Étape	Exemples d'outils
Détection de secrets	Code, historique Git, logs de build	Clés API, mots de passe, tokens exposés	IDE, pre-commit, PR	Gitleaks, TruffleHog, GitHub Secret Scanning
SAST	Code source	Injectons SQL, XSS, désérialisation, crypto faible	IDE, PR	Semgrep, SonarQube, CodeQL, Checkmarx
SCA	Dépendances et licences	CVE connues dans les librairies, licences incompatibles	PR, build, continu	Dependabot, OWASP Dependency-Check, Snyk, Trivy
Scan IaC	Terraform, Bicep, ARM, Kubernetes, Helm	Stockage public, ports ouverts, absence de chiffrement, droits excessifs	PR	Checkov, Trivy, KICS, Defender for DevOps
Scan de conteneurs	Images et couches	CVE OS et runtime, image de base obsolète, exécution en root	Build, registre	Trivy, Gype, Defender for Containers
SBOM	Inventaire des composants	Visibilité sur ce qui est déployé, réponse rapide aux nouvelles CVE	Build	Syft, CycloneDX, SPDX tools
Signature et provenance	Artefacts, images	Substitution ou altération d'artefact	Build, admission	Sigstore Cosign, Notation, attestations SLSA
DAST	Application en exécution	Faibles exploitables à l'exécution, mauvaise configuration HTTP	Préproduction	OWASP ZAP, Burp Suite Enterprise

Catégorie	Ce qui est analysé	Risques couverts	Étape	Exemples d'outils
Tests d'API / fuzzing	Endpoints REST, GraphQL	Contrôles d'accès cassés (BOLA), validation d'entrée	Préproduction	ZAP API scan, RESTler, Schemathesis
Admission / policy	Manifests au déploiement	Déploiement non conforme	CD	OPA Gatekeeper, Kyverno, Azure Policy
Runtime / CSPM / CNAPP	Workloads et cloud en production	Comportement anormal, dérive de configuration	Production	Falco, Defender for Cloud, Wiz, Prisma Cloud
ASPM / agrégation	Résultats de tous les outils	Bruit, doublons, absence de priorisation	Transverse	DefectDojo, Defender for Cloud DevOps security

Règle de sélection. Préférer peu d'outils bien réglés, intégrés à la PR et agrégés dans une seule console, plutôt qu'une collection de scanners dont personne ne lit les rapports. Dans un environnement Microsoft (Azure DevOps ou GitHub + Azure), Defender for Cloud DevOps security centralise une grande partie de ces résultats.

Sécuriser la chaîne d'approvisionnement logicielle

Le pipeline lui-même est une cible : compromettre un runner, une dépendance ou un compte de CI permet d'injecter du code dans tout ce qui est livré. Les attaques de type SolarWinds, Codecov ou les paquets npm/PyPI malveillants exploitent précisément ce maillon.

■ Menaces principales

- **Dépendance malveillante ou compromise** : typosquatting, prise de contrôle d'un paquet légitime, confusion de dépendances (dependency confusion) entre registre interne et public.
- **Pipeline compromis** : action ou plugin tiers malveillant, runner persistant réutilisé, secret de CI volé, workflow modifiable depuis un fork.
- **Artefact altéré** : image remplacée dans le registre entre le build et le déploiement.
- **Identité de CI sur-privilegiée** : un compte de service avec des droits Owner sur l'abonnement devient une porte d'entrée vers toute la production.

■ Contre-mesures

Contre-mesure	Mise en œuvre concrète
Épingler les dépendances	Lockfiles commités, actions tierces référencées par SHA de commit et non par tag, images par digest
Registre proxy interne	Toutes les dépendances passent par un miroir (Artifactory, Azure Artifacts) avec scopes réservés contre la confusion de dépendances
Identité fédérée OIDC	Le pipeline obtient un jeton court-terme auprès d'Entra ID ou du cloud ; aucun secret client stocké

Contre-mesure	Mise en œuvre concrète
Moindre privilège du pipeline	Une identité par environnement, rôle limité au groupe de ressources, pas de droits Entra ID sauf besoin justifié
Runners éphémères et isolés	Un runner neuf par job, sortie réseau sur liste blanche, pas de runner auto-hébergé partagé entre projets publics et privés
SBOM systématique	Généré à chaque build, stocké avec l'artefact, réinterrogé en continu contre les nouvelles CVE
Signature + vérification	Signature Cosign / Notation au build, vérification obligatoire à l'admission (Kyverno, Gatekeeper, Ratify)
Attestations de provenance	Preuve de quel code, quel pipeline et quel builder ont produit l'artefact
Protection des workflows	Fichiers de pipeline sous CODEOWNERS, approbation requise pour les PR externes, jeton de CI en lecture seule par défaut

■ Le cadre SLSA

SLSA (Supply-chain Levels for Software Artifacts, prononcé « salsa ») définit des niveaux progressifs de garantie sur la façon dont un artefact a été construit.

Niveau (piste Build)	Exigence résumée
Build L1	La provenance existe : on documente comment l'artefact a été construit
Build L2	Build sur une plateforme hébergée, provenance signée par cette plateforme
Build L3	Plateforme de build durcie : builds isolés entre eux, secrets de signature inaccessibles au code du build

Viser L2 est réaliste pour la plupart des organisations avec GitHub Actions ou Azure Pipelines ; L3 demande des builders isolés dédiés.

Performance : sécurité sans ralentir la livraison

Un contrôle de sécurité trop lent ou trop bruyant finit désactivé ou contourné. La performance du pipeline est donc une exigence de sécurité à part entière.

■ Où se situe le coût

Contrôle	Coût relatif	Où le placer	Levier d'optimisation
Détection de secrets	Très faible	Pre-commit + chaque PR	Scanner uniquement le diff ; historique complet en tâche planifiée
SAST	Faible à élevé selon le moteur	Chaque PR	Analyse incrémentale des fichiers modifiés ; scan complet nocturne
SCA	Faible	Chaque PR + continu	Ne relancer que si le lockfile change ; surveillance continue côté registre
Scan IaC	Faible	PR touchant l'IaC	Filtres de chemins pour ne lancer que si nécessaire
Scan d'image	Moyen	Build	Cache de la base de vulnérabilités, images de base minimales (distroless, Alpine)
DAST	Élevé	Préproduction	Scan ciblé sur les endpoints modifiés ; scan complet planifié
Fuzzing	Très élevé	Planifié, hors chemin critique	Campagnes nocturnes ou hebdomadaires

■ Bonnes pratiques d'optimisation

- **Paralléliser** les jobs indépendants (SAST, SCA, secrets, IaC) au lieu de les enchaîner.
- **Scanner le diff** en PR et réserver les scans complets aux exécutions planifiées.

- **Mettre en cache** dépendances, bases de vulnérabilités et couches d'image.
- **Séparer bloquant et informatif** : seuls les contrôles rapides et fiables sont sur le chemin critique ; les contrôles longs tournent en asynchrone et ouvrent des tickets.
- **Réduire les faux positifs** : un outil mal réglé coûte plus en temps humain qu'en temps machine. Suppressions versionnées, règles adaptées au langage et au framework.
- **Fixer un budget de temps** : par exemple, un objectif interne de retour de sécurité en PR sous une dizaine de minutes, à ajuster selon le contexte.

■ Mesurer : métriques DORA + métriques de sécurité

Les quatre métriques DORA mesurent la performance de livraison ; elles doivent rester stables ou s'améliorer quand on ajoute la sécurité.

Famille	Métrique	Ce qu'elle révèle
DORA	Fréquence de déploiement	La sécurité ne freine pas la cadence
DORA	Délai de mise en production (lead time)	Le pipeline sécurisé reste rapide
DORA	Taux d'échec des changements	Qualité et sécurité se renforcent
DORA	Temps de restauration	La capacité de rollback et de correction
Sécurité	Délai moyen de correction par criticité	L'efficacité réelle de la remédiation
Sécurité	Âge et volume des vulnérabilités ouvertes	La dette de sécurité accumulée
Sécurité	Taux de faux positifs	La confiance des développeurs dans les outils
Sécurité	Durée des étapes de sécurité	Le coût en temps de chaque contrôle
Sécurité	Exceptions actives et expirées	La discipline de gestion du risque
Sécurité	Couverture (% de dépôts équipés)	L'étendue du déploiement

Prérequis

Sans socle DevOps minimal, le DevSecOps n'a rien sur quoi s'appuyer : il faut d'abord du code versionné, des pipelines automatisés et une infrastructure décrite en code.

■ Prérequis organisationnels

- ☐ Sponsor de direction (CTO, RSSI) qui arbitre entre vitesse et risque et finance la remédiation.
- ☐ Politique de sécurité applicative écrite : niveaux de criticité, délais de correction attendus (SLA), processus d'exception.
- ☐ Rôles définis : propriétaire de la plateforme CI/CD, équipe AppSec, un security champion par équipe produit.
- ☐ Accord explicite sur ce qui bloque un déploiement et qui peut accorder une dérogation.
- ☐ Temps de remédiation réservé dans les sprints (par exemple une part fixe de la capacité).

■ Prérequis techniques

- ☐ Tout le code dans un SCM centralisé (GitHub, GitLab, Azure DevOps) avec branches protégées et revue obligatoire.
- ☐ Pipelines CI/CD définis en YAML et versionnés, idéalement à partir de templates partagés.
- ☐ Infrastructure as Code (Terraform, Bicep), pas de modifications manuelles en production.
- ☐ Environnements séparés (dev, préprod, prod) avec identités et secrets distincts.
- ☐ Coffre de secrets opérationnel (Key Vault, Vault) et identité fédérée OIDC pour les pipelines.
- ☐ Registre d'artefacts privé.
- ☐ Plateforme de logs centralisée (SIEM) capable d'ingérer les journaux du SCM et du CI/CD.

☐ Inventaire des applications et de leur criticité métier.

■ Prérequis de compétences

Profil	Compétences attendues
Développeurs	OWASP Top 10, OWASP API Top 10, codage sécurisé, lecture d'un rapport SAST/SCA
Security champions	Threat modeling (STRIDE), triage des findings, notions de CVSS et EPSS
Ingénieurs plateforme	Durcissement CI/CD, OIDC, Kubernetes et politiques d'admission, gestion des secrets
Équipe AppSec	Réglage des outils, analyse de risque, pentest applicatif, supply chain (SLSA, SBOM)
SOC	Détection sur les journaux CI/CD et cloud, réponse à incident sur la chaîne de livraison

Mise en place : feuille de route

On déploie le DevSecOps progressivement, en commençant par des contrôles informatifs sur quelques équipes pilotes avant de rendre les gates bloquantes pour tous.



Feuille de route : 5 phases et leurs critères de passage

Les durées sont des ordres de grandeur à adapter à la taille du parc applicatif ; ne passer à la phase suivante que lorsque le critère est atteint.

■ Points d'attention par phase

- **Phase 0.** Le threat modeling des applications critiques commence ici : il oriente le choix des règles et des contrôles prioritaires.

- **Phase 1.** Tout est en mode « audit » : on mesure le volume de findings et le taux de faux positifs sans bloquer personne. C'est la phase où l'on gagne ou perd la confiance des développeurs.
- **Phase 2.** Le blocage ne s'applique qu'aux nouveaux findings (diff) ; la dette existante est traitée dans un backlog dédié avec des échéances.
- **Phase 3.** La vérification des signatures commence en mode audit à l'admission, puis passe en mode refus une fois tous les pipelines signés.
- **Phase 4.** Les résultats de pentests et d'incidents réels alimentent de nouvelles règles automatisées.

Exemples de pipelines

Les exemples ci-dessous implémentent la gate 1 (PR) et la gate 2 (artefact). Ce sont des points de départ : vérifier les versions des actions et tâches au moment de l'implémentation, et les épingler par SHA en production.

■ GitHub Actions — pull request (gate 1)

```
name: devsecops-pr
on:
  pull_request:
    branches: [main]

permissions:
  contents: read
  security-events: write
  pull-requests: write

jobs:
  secrets:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - uses: gitleaks/gitleaks-action@v2
        env:
          GITHUB_TOKEN: ${ secrets.GITHUB_TOKEN }

  sast:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: github/codeql-action/init@v3
        with:
          languages: csharp
      - uses: github/codeql-action/autobuild@v3
      - uses: github/codeql-action/analyze@v3

  sca:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/dependency-review-action@v4
        with:
          fail-on-severity: high

  iac:
    runs-on: ubuntu-latest
```

```

steps:
  - uses: actions/checkout@v4
  - uses: bridgecrewio/checkov-action@v12
  with:
    directory: infra/
    soft_fail: false

```

Les quatre jobs tournent en parallèle. Pour que la gate soit réellement bloquante, déclarer ces jobs comme « required status checks » dans la protection de la branche `main`.

■ GitHub Actions — build, scan, SBOM, signature (gate 2)

```

name: build-release
on:
  push:
    branches: [main]

permissions:
  contents: read
  packages: write
  id-token: write # OIDC : signature keyless + Azure sans secret

env:
  IMAGE: ghcr.io/mon-org/mon-app:${{ github.sha }}

jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Build
        run: docker build -t "$IMAGE" .

      - name: Scan de l'image
        uses: aquasecurity/trivy-action@0.28.0
        with:
          image-ref: ${{ env.IMAGE }}
          severity: CRITICAL,HIGH
          ignore-unfixed: true
          exit-code: '1'

      - name: Push
        run: |
          echo "${{ secrets.GITHUB_TOKEN }}" | docker login ghcr.io \
            -u ${{ github.actor }} --password-stdin
          docker push "$IMAGE"
          D=$(docker inspect --format='{{index .RepoDigests 0}}' "$IMAGE")
          echo "DIGEST=$D" >> "$GITHUB_ENV"

      - name: SBOM CycloneDX
        uses: anchore/sbom-action@v0
        with:
          image: ${{ env.IMAGE }}
          format: cyclonedx-json

```

```

- uses: sigstore/cosign-installer@v3

- name: Signature keyless (Sigstore)
  run: cosign sign --yes "$DIGEST"

- name: Connexion Azure par OIDC
  uses: azure/login@v2
  with:
    client-id: ${vars.AZURE_CLIENT_ID}
    tenant-id: ${vars.AZURE_TENANT_ID}
    subscription-id: ${vars.AZURE_SUBSCRIPTION_ID}

```

La connexion Azure utilise un credential fédéré déclaré sur l'application Entra ID : aucun secret client n'est stocké dans GitHub. La signature porte sur le digest, jamais sur le tag, car un tag peut être déplacé.

■ Azure DevOps — PR avec Microsoft Security DevOps

```

trigger: none
pr:
  branches:
    include: [main]

pool:
  vmImage: ubuntu-latest

stages:
- stage: Securite
  jobs:
  - job: Scans
    steps:
    - checkout: self
      fetchDepth: 0

    - task: MicrosoftSecurityDevOps@1
      displayName: SAST, IaC et conteneurs (MSDO)
      inputs:
        categories: 'code,IaC,containers'
        break: true

    - script: |
        docker run --rm -v $(Build.SourcesDirectory):/src \
          zricethezav/gitleaks:latest \
          detect --source /src --exit-code 1
      displayName: Détection de secrets (Gitleaks)

    - script: |
        docker run --rm -v $(Build.SourcesDirectory):/src \
          aquasec/trivy:latest fs --scanners vuln \
          --severity CRITICAL,HIGH --exit-code 1 /src
      displayName: SCA (Trivy)

```

Côté Azure DevOps, la gate est imposée par une « build validation policy » sur la branche `main`, et les déploiements utilisent une service connection en « workload identity federation » plutôt qu'un secret de principal de service. Les résultats MSDO remontent dans Defender for Cloud (DevOps security) lorsque le connecteur Azure DevOps y est configuré.

■ Kubernetes (Kyverno) — vérifier la signature à l'admission

```
apiVersion: kyverno.io/v1
kind: ClusterPolicy
metadata:
  name: verifier-signature-images
spec:
  validationFailureAction: Audit # Enforce une fois tout signé
  webhookTimeoutSeconds: 30
  rules:
  - name: images-signeas
    match:
      any:
      - resources:
          kinds: [Pod]
    verifyImages:
      - imageReferences:
          - "ghcr.io/mon-org/*"
        attestors:
          - entries:
              - keyless:
                  subject: "https://github.com/mon-org/*"
                  issuer: "https://token.actions.githubusercontent.com"
```

Gouvernance, maturité et pièges

La gouvernance DevSecOps fixe qui décide, sur quels critères, et comment le risque résiduel est accepté de façon traçable.

■ Vulnérabilités et exceptions

- **Priorisation par le risque réel** et non par le seul score CVSS : exploitabilité (EPSS, présence dans le catalogue CISA KEV), exposition (Internet ou interne), criticité métier de l'application, accessibilité du code vulnérable.
- **SLA de correction par criticité**, définis dans la politique AppSec et suivis dans l'ASPM.
- **Exceptions** : justification écrite, compensation éventuelle, approbateur identifié, date d'expiration obligatoire. Une exception expirée redevient bloquante.
- **Séparation des rôles** : celui qui demande une exception n'est pas celui qui l'approuve.
- **Audit** : journaux du SCM et du CI/CD conservés et envoyés au SIEM (modifications de protection de branche, d'identités de pipeline, de politiques).

■ Modèle de maturité

Plusieurs référentiels existent : OWASP SAMM, OWASP DSOMM (DevSecOps Maturity Model), BSIMM. Une lecture simplifiée en quatre niveaux :

Niveau	Caractéristiques
1 — Initial	Scans manuels ou ponctuels, pentest annuel, sécurité perçue comme un frein
2 — Outillé	Secrets, SCA et SAST automatisés en PR, résultats en mode informatif, quelques équipes couvertes
3 — Intégré	Gates bloquantes basées sur le risque, templates généralisés, SBOM et signature, métriques suivies

Niveau	Caractéristiques
4 — Optimisé	Supply chain vérifiée de bout en bout (SLSA), threat modeling systématique, amélioration pilotée par les métriques et les incidents

■ Pièges courants

Piège	Conséquence	Parade
Tout bloquer dès le premier jour	Révolte des équipes, contournements, outils désactivés	Mode audit d'abord, blocage progressif sur le diff
Empiler les outils	Doublons, bruit, rapports ignorés	Peu d'outils, bien réglés, agrégés dans une console
Ignorer les faux positifs	Perte de confiance dans les résultats	Réglage continu, suppressions versionnées et revues
Sécuriser l'appli mais pas le pipeline	Le pipeline devient le vecteur d'attaque	OIDC, runners éphémères, actions épinglées, CODEOWNERS
Secrets dans les variables de CI	Fuite via logs, forks ou runners compromis	Coffre de secrets + identité fédérée
Pas de temps de remédiation	La dette de sécurité s'accumule	Capacité réservée dans les sprints, SLA suivis
DevSecOps vu comme un projet outil	Adoption superficielle	Security champions, formation, objectifs partagés
Oublier la production	Nouvelles CVE non détectées sur l'existant	SBOM réinterrogé en continu, protection runtime

Conclusion et références

Le DevSecOps réussi n'est pas celui qui a le plus d'outils, mais celui où la sécurité est devenue une étape normale, rapide et fiable du travail quotidien des développeurs.

Trois leviers font la différence : des contrôles automatisés au bon endroit du pipeline, un pipeline lui-même durci, et une gouvernance qui arbitre le risque de façon explicite.

Par où commencer demain : activer la détection de secrets avec blocage au push, activer la SCA sur les dépôts critiques, remplacer les secrets de pipeline par une identité fédérée OIDC.

■ Références

- OWASP DevSecOps Guideline et OWASP DSOMM (DevSecOps Maturity Model)
- OWASP SAMM (Software Assurance Maturity Model)
- OWASP Top 10 et OWASP API Security Top 10
- NIST SP 800-218 — Secure Software Development Framework (SSDF)
- SLSA — Supply-chain Levels for Software Artifacts (slsa.dev)
- CIS Software Supply Chain Security Guide
- OpenSSF Scorecard et projet Sigstore
- Microsoft Security Development Lifecycle (SDL) et documentation Defender for Cloud DevOps security
- DORA — State of DevOps Report

À retenir

1

Shift-left et shift-right

Détecter au plus tôt dans l'IDE et la PR, surveiller en continu en production.

2

Des gates basées sur le risque

Bloquer ce qui compte, signaler le reste, en commençant en mode audit.

3

Un pipeline lui-même durci

OIDC, runners éphémères, dépendances épinglées, artefacts signés et vérifiés.

4

Mesurer vitesse et sécurité

Métriques DORA stables, délais de correction et faux positifs suivis.

5

Une gouvernance explicite

Exceptions justifiées, approuvées et datées ; security champions dans chaque équipe.