PETELET VIRGIL

BTS SIO SLAM

VEILLE TECHNOLOGIQUE

La méthodologie DevOps

Table des matières

1. Introduction 3
2. Contexte actuel de GSB : l'état des lieux 3
2.1 Architecture et stack technologique 3
2.2 Architecture logique et flux métier 4
2.3 Processus actuel de développement et déploiement 5
2.4 Les Problèmes Identifiés 5
3. Problématique et enjeux stratégiques 6
3.1 Formulation de la Problématique 6
3.2 Les enjeux métier 6
3.3 Ligne directrice et enjeu principal 7
4. Historique et Évolution de DevOps 7
4.1 L'ère pré-DevOps (Avant 2010) 7
4.2 L'ère Agile (2010-2015) 8
4.4 L'Ère Moderne : Kubernetes et observabilité (2020-Présent) 9
5. Les principes clés de DevOps appliqués à GSB 9
5.1 CI/CD : Le pipeline automatisé 9
5.2 Monitoring et alertes en temps réel 10
6. Les avantages concrets pour GSB 11
6.1 Déploiements rapides et fiables 11
6.2 Réduction drastique des bugs en production 12
6.3 Collaboration Développeurs ↔ Administrateurs 12
6.4 Moins de stress et meilleure qualité de vie 13
6.5 Scalabilité et adaptabilité 13
7. Les Défis et Inconvénients de DevOps 13
7.1 Courbe d'apprentissage très importante 13
7.2 Complexité initiale 14
7.3 Coût d'infrastructure 14
7.4 Sécurité renforcée mais complexe 14
7.5 Maintenance constante 15
8. Étude d'impact : Coûts et ROI détaillés 15
8.1 Investissement initial (Année 1) 15
8.3 Bénéfices quantifiables 15
9. Conclusion 16

1. Introduction

DevOps est bien plus qu'un terme à la mode. C'est une véritable transformation culturelle et technique qui redéfinit la manière dont les organisations construisent, testent, déploient et maintiennent les applications logicielles. Pour GSB, notre application de gestion des notes de frais destinée aux visiteurs médicaux, DevOps représente une opportunité stratégique majeure.

Actuellement, GSB fonctionne correctement d'un point de vue fonctionnel, mais elle souffre de plusieurs problèmes structurels typiques des applications développées selon une approche "en cascade" :

- **Les déploiements sont lents et manuels** : à chaque mise à jour, quelqu'un doit se connecter au serveur, faire un git pull, redémarrer Apache, et espérer que rien ne casse. Ce processus peut prendre 2-3 heures et est sujet à des erreurs humaines.
- **Les tests ne sont pas systématiques** : Les tests sont faits manuellement ce qui signifie que des bugs arrivent souvent en production.
- **Il n'existe pas de supervision automatique** : Personne ne sait que l'application fonctionne mal jusqu'à ce qu'un utilisateur appelle pour se plaindre.
- **Les équipes travaillent en silos** : Les développeurs, les administrateurs système et l'équipe support ne communiquent pas vraiment. Quand un problème apparaît, c'est une situation de "blâme" plutôt que de collaboration.
- **Les cycles de livraison sont longs** : Une nouvelle fonctionnalité peut prendre des semaines ou des mois entre sa validation et son déploiement en production.

DevOps propose une solution à ces problèmes : **automatiser au maximum, permettre aux équipes de collaborer étroitement, et créer un pipeline continu où le code passe de la machine du développeur à la production de manière fiable et rapide**.

Dans cette veille, nous allons explorer en détail comment DevOps pourrait transformer GSB, les bénéfices concrets attendus, les défis à relever, les coûts impliqués, et un plan d'action réaliste pour mettre en place DevOps progressivement et efficacement.

2. Contexte actuel de GSB : l'état des lieux

2.1 Architecture et stack technologique

GSB est une application complète qui s'étend sur trois différentes distributions différentes :

Backend Web :

- Framework : Symfony 8.0.3
- Langage : PHP 8.4.16
- Serveur web : Apache 2.4.66

- Système d'exploitation : Debian 13.3

Frontend Web :

- Moteur de template : Twig
- Langages : HTML5, CSS3, JavaScript
- Framework CSS : Bootstrap 5
- Communication serveur : via des routes REST définies dans Symfony

Application Mobile :

- Plateforme : Android natif
- Langage : Java
- Bibliothèques clés : Retrofit 2.11.0 (client HTTP), Gson 2.11.0 (sérialisation JSON), Android Studio IDE

Base de Données :

- Système : MariaDB
- ORM : Doctrine 2.x avec migrations versionnées
- Versioning : les migrations sont versionnées dans Git pour reproducibilité

Authentification :

- Intégration : Active Directory / LDAP
- Avantage : synchronisation avec les annuaires d'entreprise existants
- Sécurité : authentification centralisée, gestion des groupes/permissions via AD

Infrastructure de Mailing :

- Service : Mailtrap (service de test/staging pour les emails)
- Cas d'usage : confirmations, réinitialisations de mot de passe, notifications

2.2 Architecture logique et flux métier

GSB suit un modèle trois-couches classique :

1. **Couche Présentation** (Twig + HTML/CSS) : Les interfaces utilisateurs pour les trois rôles (Visiteur, Comptable, Admin)
2. **Couche Métier** (Symfony + PHP) : La logique applicative, les règles métier, les calculs complexes
3. **Couche Données** (MariaDB + Doctrine) : Persistance des données, rapports

Les trois rôles utilisateurs et leurs workflows :

- **Visiteur médical** : Se connecte via AD → Saisit ses dépenses (restaurant, transport, hôtel) → Calcul automatique de l'indemnité kilométrique → Soumet les notes → Reçoit un email de confirmation
- **Comptable** : Reçoit les notes → Les valide une par une → Marque les notes "acceptées" ou "rejetées" → Génère des rapports de remboursement
- **Admin** : Gère les utilisateurs → Configure les règles de validation → Accède aux statistiques d'utilisation

2.3 Processus actuel de développement et déploiement

Environnement de Développement :

- Chaque développeur utilise sa propre machine
- VirtualBox est utilisé pour tester en environnement Linux local
- Code versionnisé sur GitLab

Environnement de Test :

- Une VM dédiée sur laquelle on teste les nouvelles versions
- Tests manuels par une seule personne
- Pas de suite de tests automatisée complète

Environnement de Production :

- Un serveur physique ou une VM sur infrastructure d'entreprise
- Accès restreint (SSH) pour les administrateurs seuls
- Déploiement manuel : git pull, redémarrage d'Apache
- Risque : le code testé en dev peut différer du code en prod

Gestion des Versions :

- Versions taggées sur Git (v1.0, v1.1, etc.)
- Documentation sommaire des changements
- Pas de changelog automatisé

2.4 Les Problèmes Identifiés

Problème 1 : Déploiements longs et risqués

- Un déploiement complet prend 1-2 heures
- Beaucoup d'étapes manuelles = beaucoup de points de défaillance
- Impossible de déployer rapidement un bug fix critique

- Quand quelque chose casse, on perd du temps à identifier le problème

Problème 2 : Absence de tests automatisés

- Les tests sont manuels et incomplets
- Pas de garantie que la nouvelle version ne casse pas les fonctionnalités existantes
- Les bugs de régression arrivent souvent en production
- Le temps de test augmente avec la taille de l'appli

Problème 3 : Pas de supervision

- On ne sait pas l'état réel de l'application en production
- Les administrateurs doivent vérifier manuellement que ça fonctionne
- Les alertes n'existent pas : on découvre les problèmes via les utilisateurs
- Pas de données sur la performance, les erreurs, ou la capacité

Problème 4 : Silos organisationnels

- Les développeurs livrent du code en fin de journée
- Pas de communication fluide entre les équipes
- En cas de problème, c'est "ton problème" vs "mon problème" au lieu de "notre problème"

Problème 5 : Documentation décentralisée et incomplète

- La configuration du serveur n'est documentée que dans la tête de l'admin
- Si l'admin part, c'est le chaos
- Reproducibilité quasi impossible

3. Problématique et enjeux stratégiques

3.1 Formulation de la Problématique

Question centrale : Comment optimiser le cycle complet de développement, de test et de déploiement de GSB pour :

- Livrer des mises à jour plus rapides (passer de 1-2 par mois à 1-2 par jour) ?
- Réduire drastiquement le nombre de bugs arrivant en production ?
- Permettre aux équipes de travail ensemble plutôt qu'en silos ?
- Assurer une disponibilité maximale et une performance prévisible ?
- Faciliter la maintenance à long terme et la scalabilité ?

3.2 Les enjeux métier

Pour les utilisateurs :

- Accès à de nouvelles fonctionnalités rapidement
- Moins d'interruptions de service
- Meilleure expérience utilisateur grâce à l'optimisation continue

Pour l'organisation GSB :

- ROI amélioré : moins de temps perdu sur les déploiements = plus de temps sur les features
- Réduction des incidents critiques = réduction des coûts d'intervention
- Meilleure réactivité aux demandes métier

Pour les équipes techniques :

- Moins de stress et de travail manuel répétitif
- Plus de temps pour innovation et amélioration technique
- Meilleure collaboration et communication

3.3 Ligne directrice et enjeu principal

L'enjeu principal est de **passer d'une approche "projet" (où on livre une grosse version tous les 3-6 mois) à une approche "produit" (où on livre continuellement de petites améliorations)**. Pour cela, il faut :

1. **Automatiser tout ce qui peut l'être** (tests, déploiements, monitoring)
2. **Créer une culture de collaboration** entre développement et exploitation
3. **Instrumenter l'application** pour voir en temps réel ce qui se passe
4. **Mesurer et s'améliorer continuellement**

C'est là que DevOps intervient : c'est l'approche qui permet de répondre à tous ces enjeux simultanément.

4. Historique et Évolution de DevOps

4.1 L'ère pré-DevOps (Avant 2010)

Contexte :

- Les applications étaient développées selon le modèle "Waterfall" strict
- Une phase de développement pouvait durer 6-12 mois
- Une fois le code terminé, il fallait plusieurs semaines pour le tester
- Le déploiement était un événement majeur, souvent réalisé le soir ou le week-end

Processus typique :

1. Les développeurs codaient pendant des mois sans aucun feedback de production
2. À la fin, on "versait" le code en test
3. Les testeurs passaient des semaines à chercher des bugs
4. On corrigeait les bugs découverts
5. Finalement, après d'innombrables révisions, on déployait en production
6. Les opérationnels s'occupaient du serveur, de la maintenance, des backups

Problèmes identifiés :

- Communication quasi nulle entre dev et ops
- Déploiements catastrophiques (des heures d'interruption de service)
- Découverte de bugs critiques seulement après le déploiement
- Impossibilité de corriger rapidement : 3 mois pour une correction mineure

Exemple historique : La plupart des applications bancaires et d'assurance fonctionnaient encore sur ce modèle. Un bug découvert pouvait immobiliser une application pendant des jours.

4.2 L'ère Agile (2010-2015)

Contexte :

- Agile commence à émerger et à se populariser
- Les entreprises réalisent que le Waterfall est trop rigide
- Les cycles d'itération passent de 12 mois à 2-3 semaines

Améliorations apportées :

- Développement en sprints de 2-3 semaines
- Livraisons plus fréquentes (1-2 par mois possible)
- Feedback utilisateur plus rapide
- Tests plus systématiques (mais encore manuels)

Limitations de cette période :

- Les déploiements restaient complexes et manuels
- Les tests étaient partiellement automatisés
- Les opérationnels n'étaient pas impliqués dans le processus Agile
- Pas de monitoring automatisé

4.3 L'ère DevOps : naissance et décollage (2015-2020)

Catalyst : Docker. L'apparition de Docker révolutionne tout.

Docker, c'est : Un conteneur léger qui empaquète une application avec toutes ses dépendances (code, runtime, bibliothèques, configuration). Avantage majeur : "ça marche sur ma machine" devient enfin vrai en production.

Changements clés :

- **CI/CD Pipelines** : Automatisation complète du test et du déploiement
- **Infrastructure as Code** : Les serveurs et configurations sont décrites en code (Terraform, Ansible)
- **Monitoring centralisé** : ELK Stack, Prometheus, Grafana apparaissent
- **Collaboration** : Dev et Ops travaillent ensemble sur les mêmes outils

Impact : Les entreprises passent de 1-2 déploiements par mois à 1-2 par jour ou par heure.

4.4 L'Ère Moderne : Kubernetes et observabilité (2020-Présent)

Kubernetes : Orchestration des conteneurs à grande échelle. Permet de :

- Gérer des milliers de conteneurs automatiquement
- Redémarrer automatiquement les services qui crash
- Faire de la scalabilité automatique (ajouter des serveurs quand la charge augmente)
- Faire du rolling deployment (déployer sans downtime)

DevSecOps : Intégration de la sécurité dans le pipeline (scans de vulnérabilités, tests de pénétration, etc.)

Observabilité : Aller au-delà du monitoring. Pouvoir tracer chaque requête, comprendre exactement ce qui se passe en production.

État actuel : Les meilleures organisations livrent en production plusieurs dizaines de fois par jour, avec une fiabilité et une performance extraordinaire.

Exemple : Netflix livre plusieurs centaines de déploiements par jour. Comment ? Parce que tout est automatisé et monitorsé. Si quelque chose casse, ça se détecte immédiatement et on fait un rollback en secondes.

5. Les principes clés de DevOps appliqués à GSB

5.1 CI/CD : Le pipeline automatisé

C'est l'épine dorsale de DevOps.

CI = Intégration Continue :
Chaque fois qu'un développeur pousse du code sur Git, une série de vérifications s'exécutent automatiquement :

1. **Compilation** : Vérifie que le code PHP/Symfony compile sans erreur
2. **Linting** : Vérifie que le code suit les conventions (indentation, nommage, etc.)
3. **Tests unitaires** : Lance tous les tests unitaires pour vérifier la logique métier
4. **Tests fonctionnels** : Teste les scénarios complets (saisir une note, la valider, etc.)
5. **Tests de sécurité** : Cherche les failles connues (injection SQL, XSS, etc.)
6. **Couverture de code** : Mesure le pourcentage du code testé
7. **Analyse de qualité** : SonarQube détecte les smells de code, les duplications, etc.

Résultat :
Si tous ces tests passent, on a confiance que le code est bon. Si un test échoue, le développeur est alerté immédiatement et peut corriger.

CD = Déploiement Continu :
Une fois que les tests CI passent :

1. **Build Docker** : Crée une image Docker de l'application
2. **Push en registry** : L'image est poussée sur un registre (Docker Hub, GitLab Registry, etc.)
3. **Déploiement en staging** : La nouvelle version est déployée en environnement de test
4. **Tests d'intégration** : Tests qui vérifient que tout fonctionne dans cet environnement
5. **Validation manuelle** : Quelqu'un teste manuellement les fonctionnalités clés
6. **Déploiement en production** : Si la validation passe, déploiement automatique en production
7. **Monitoring** : Les alertes surveillent que tout fonctionne bien

Avantages pour GSB :

- Les bug fixes arrivent en production en 20 min au lieu de 3 jours
- Les nouvelles features peuvent être testées en production immédiatement
- Pas de "surprise" : chaque déploiement a été testé automatiquement
- Rollback rapide si quelque chose casse

5.2 Monitoring et alertes en temps réel

Actuellement, GSB n'a **aucune supervision**. On ignore complètement comment ça va en production.

Avec DevOps, on met en place du monitoring continu sur plusieurs dimensions :

Métriques d'Application :

- Nombre de requêtes par seconde
- Temps de réponse moyen (et percentiles)
- Taux d'erreurs HTTP (500, 404, etc.)
- Nombre de sessions actives
- Nombre de notes de frais saisies par heure

Métriques d'Infrastructure :

- Utilisation CPU
- Utilisation mémoire
- Utilisation disque
- Bande passante réseau
- Santé des disques

Métriques de Base de Données :

- Nombre de connexions actives
- Temps de réponse des requêtes
- Nombre de requêtes lentes
- Taille de la base de données
- Cache hit rate

Outils utilisés :

Prometheus : Collecte les métriques

- Scrape régulièrement (toutes les 10 secondes) l'application, le serveur et la base
- Stocke les données historiques (typiquement 15 jours)
- Langage de requête puissant pour analyser les métriques

Grafana : Affiche les métriques visuellement

- Tableaux de bord avec graphiques en temps réel
- Alertes configurables basées sur les métriques

6. Les avantages concrets pour GSB

6.1 Déploiements rapides et fiables

Situation actuelle :

- 1 déploiement = 2-3 heures
- 1-2 déploiements par mois
- À chaque déploiement : stress, risque d'erreur, downtime possible

Avec DevOps :

- 1 déploiement = 15-20 minutes
- 10-20 déploiements par jour possible
- Chaque déploiement est testé automatiquement, donc peu de risque

Cas concret : Un bug critique est découvert en production (ex: calcul d'indemnité kilométrique incorrect).

- Sans DevOps : correction, test manuel (2h), déploiement (1h), vérification = 4-5 heures d'attente
- Avec DevOps : correction, test auto (2 min), déploiement auto (5 min) = 7-10 minutes

Impact métier : Les utilisateurs reçoivent des corrections 30x plus vite. Ça réduit énormément les impacts des bugs critiques.

6.2 Réduction drastique des bugs en production

Pourquoi ça marche :

- Tests automatisés à chaque commit
- Déploiements isolés et testés
- Rollback rapide si quelque chose casse

Statistiques typiques :

- Avant DevOps : 1 incident critique par mois
- Après DevOps : 1 incident critique tous les 6 mois (voir moins)

Pour GSB :

- Calcul des frais : tests automatisés garantissent que la logique métier n'est jamais cassée
- Authentification : tests de sécurité automatisés
- Génération de rapports : tests de régression automatisés

6.3 Collaboration Développeurs ↔ Administrateurs

Avant :

- Les devs font du code et le balance à l'admin
- L'admin fait le déploiement et c'est "à lui de se débrouiller"
- En cas de problème, c'est le "blâme game"

Après :

- Les devs et les admins partagent les mêmes outils (Git, pipeline, monitoring)
- Tout le monde voit l'état de l'application
- Les problèmes sont "nos problèmes" pas "ton problème"

Impact :

- Meilleure communication
- Résolution plus rapide des problèmes
- Moins de tension entre équipes

6.4 Moins de stress et meilleure qualité de vie

Avant :

- Tout le monde stressé, une fausse manip et c'est catastrophe
- Après le déploiement, monitoring manuel jusqu'à 2h du matin

Après :

- Tout est automatique, personne n'est stressé
- Monitoring automatique, pas de surveillance manuelle
- Rollback en cas de problème

6.5 Scalabilité et adaptabilité

Quand GSB grandit et qu'on a plus d'utilisateurs :

- Fin de mois : rush, beaucoup de saisies simultanées
- Avant : on croise les doigts et prie que le serveur tienne
- Après : Kubernetes ajoute automatiquement des instances pour absorber la charge

7. Les Défis et Inconvénients de DevOps

7.1 Courbe d'apprentissage très importante

DevOps c'est vaste. Les équipes doivent apprendre :

- Docker et la conteneurisation
- Kubernetes

- Pipelines CI/CD (Jenkins, GitLab CI, GitHub Actions)
- Monitoring (Prometheus, Grafana)
- Infrastructure as Code (Terraform, Ansible)
- Agile et méthodologie des sprints
- Gestion Git avancée

Impact : 2-3 mois d'apprentissage sérieux pour que tout le monde soit productif.

7.2 Complexité initiale

Mettre en place DevOps, c'est complexe. Il y a beaucoup de pièces qui doivent fonctionner ensemble.

Risques :

- Si on fait mal, on peut tout casser
- Les pipelines peuvent être fragiles au début
- Les outils peuvent interférer les uns avec les autres

Mitigation : Commencer petit, avec les éléments les plus critiques, puis expand progressivement.

7.3 Coût d'infrastructure

Docker et Kubernetes demandent des ressources.

Pour GSB, estimation :

- Serveur de build (CI/CD) : 1 VM (4 CPU, 8GB RAM)
- Serveur de test : 1 VM (2 CPU, 4GB RAM)
- Serveur de production (containérisé) : 2-3 VMs pour la redondance
- Stockage pour les images Docker : ~100GB
- Monitoring (Prometheus, Grafana) : 1 VM (2 CPU, 4GB RAM)

Coûts mensuels (cloud): ~800-1200 €/mois pour l'infra (vs. peut-être 300-400€ avant)

7.4 Sécurité renforcée mais complexe

Plus il y a d'automatisation, plus il faut de sécurité.

Risques :

- Si quelqu'un pousse du code malveillant, l'automatisation le déploiera partout
- Les images Docker peuvent contenir des vulnérabilités
- Les certificats, clés d'API, mots de passe doivent être gérés très soigneusement

Mitigation :

- Code reviews systématiques
- Scans de vulnérabilités automatisés
- Gestionnaire de secrets (Vault, etc.)
- Logs d'audit de tous les déploiements

7.5 Maintenance constante

Les outils DevOps évoluent rapidement.

Effort :

- Mises à jour régulières de Docker, Kubernetes, etc.
- Patches de sécurité à appliquer
- Monitoring des meilleures pratiques qui changent

Effort estimé : 10-15% du temps de l'équipe ops en continu.

8. Étude d'impact : Coûts et ROI détaillés

8.1 Investissement initial (Année 1)

Catégorie	Détail	Coût
Formation	3 jours pour chaque dev/admin	8,000 €
Outils et Licences	Jenkins, SonarQube, Artifactory, etc.	3,000 €
Infrastructure supplémentaire	VMs build/test/monitoring	8,000 €
Consulting/Accompagnement	Aide externe pour mise en place	10,000 €
Temps interne	300 heures dev/admin à 50€/h	15,000 €
Temps de mise en place	Code reviews, tests, IaC	12,000 €
Documentation	Guides, runbooks, etc.	2,000 €
Buffer (15% imprévus)	-	6,750 €
TOTAL ANNÉE 1		**64,750 €**

8.3 Bénéfices quantifiables

Avant DevOps (sans changement) :

- Déploiement manuel : 1 personne x 3 heures = 150 € d'effort humain
- 1 déploiement par mois = 1,800 € / an

- Incidents en prod : 2 par mois x 2h de reparation = ~4,800 € / an
- Total : ~6,600 €/an en inefficacité

Après DevOps :

- Déploiement automatique : coûts d'infra + maintenance
- 20 déploiements par mois x 0€ d'effort manuel = 0 €
- Incidents : peut-être 0.5 par mois (déploiements plus fiables) = ~1,200 € / an
- Total direct : ~1,200 €/an

Gain direct : 6,600 - 1,200 = 5,400 € / an

Bénéfices indirects :

- **Productivité dev** : Les devs ne passent plus 5h/mois sur les déploiements, mais plutôt sur des features. Gain : 40h/an x 50€ = 2,000 €
- **Time-to-market** : Nouvelles features sortent 10x plus vite. Ça peut générer de la valeur métier (plus de fonctionnalités, plus tôt). Valeur : 10,000 - 50,000 € dépendant du secteur
- **Réduction des bugs** : 80% moins de bugs en prod (si on retire les bugs de déploiement). Coûts évités : ~4,000 €/an
- **Satisfaction utilisateurs** : Moins de downtime, features plus rapides = rétention utilisateurs accrue. Valeur : 5,000 - 20,000 €
- **Moral des équipes** : Moins de stress = moins de turnover. Remplacement d'une personne = 50,000€+. Économies : peut-être 10,000 €/an

Bénéfices indirects totaux : ~31,000 - 100,000 € / an

9. Conclusion

DevOps est une méthodologie continu d'amélioration. Pour GSB, l'adoption de DevOps transformerait radicalement la manière dont l'équipe développe, teste et déploie l'application.

Ce qu'on gagne :

- Déploiements 10x plus rapides (3h → 20 min)
- Bugs 4x moins nombreux en production
- Équipes collaborant plutôt qu'en silos
- Infrastructure reproductible et versionée
- Visibilité totale sur l'état de l'application
- Capability de scaler quand la charge augmente

Ce que ça coûte :

- Investissement initial : ~65k€
- Coûts récurrents : ~25k€/an
- Effort d'apprentissage : 2-3 mois
- Risques de transition à gérer

Le verdict :
Le ROI est clairement positif. Même dans un scénario conservateur, on retrouve l'investissement en moins de 2 ans. Et les bénéfices qualitatifs (moral des équipes, satisfaction utilisateurs, capacité d'innovation) sont probablement plus importants que les bénéfices quantifiés.