Maintenance et évolution

Plan et objectifs

- Défis de l’étape de maintenance
- Concept de dette technique
- Surveillance
- Observabilité
- Évolution d’application
- Gestion de changements (données et API)
- Migrations

Maintenance

La maintenance est l’étape du processus de développement logiciel où on modifie une application après qu’elle ait été mise en service.

Habituellement, la maintenance n’implique pas de changement majeur à l’architecture du système. Elle consiste plutôt en une évolution graduelle réalisée en modifiant ou en ajoutant des composants.

Les activités de maintenance sont:

- Correction de défauts
- Adaptation de l’application à un environnement d’opération en évolution
- Ajout ou ajustement de fonctionnalités

Les *lois* de l'évolution de logiciel

Ensemble de lois (observations) qui régissent l'évolution de logiciel selon Lehman:

- **Changement continu** - un système doit être continuellement adapté, sinon il devient progressivement moins satisfaisant.
- **Complexité croissante** - au fur et à mesure qu'un système évolue, sa complexité augmente, à moins que l'on ne travaille à la maintenir ou à la réduire.
- **Croissance continue** - le contenu fonctionnel d'un système doit être continuellement augmenté pour maintenir la satisfaction de l'utilisateur pendant sa durée de vie.
- **Qualité déclinante** - la qualité d'un système semblera décliner s'il n'est pas rigoureusement entretenu et adapté aux changements de l'environnement opérationnel.

Maintenance et coûts

Les coûts de maintenance de logiciel sont habituellement plus élevés que les coûts de construction (facteur x2 à x100).

Plusieurs facteurs influencent ces coûts:

- la durée de vie de l’application
- la stabilité de l’équipe
- les compétences de l’équipe
- la pile technologique et son évolution

Dette technique

La dette technique peut être accrue lors d'un codage non optimal. ***Une conception logicielle négligée induit des coûts futurs****, les intérêts, à rembourser sous forme de temps de développement supplémentaire, et des bugs de plus en plus fréquents. La dette technique doit être remboursée rapidement pour éviter l'accumulation de ces intérêts, d'où l'analogie avec le concept de dette financière.*

- https://fr.wikipedia.org/wiki/Dette_technique

Il arrive souvent que les gens moins techniques ne comprennent pas les implications d’une solution rapide sous-optimale pour ajouter une fonctionnalité ou corriger un défaut dans une application.

Bien qu’elle ne soit pas chiffrée, le concept de dette technique est une métaphore qui exprime bien l'impact de différentes décisions techniques.

Dette technique

Une dette technique peut être intentionnelle ou non.

Il est possible que la décision de prendre un raccourci pour l'implémentation d'une fonctionnalité (en connaissant les répercussions) soit une bonne décision d'un point de vue affaires.

Il est important de bien identifier et comprendre les impacts à court et long terme de la dette technique accrue.

Il est aussi possible qu'une dette technique soit mise en évidence lors de l' évolution de l'application. Par exemple, les développeurs n'étaient pas au courant d'une faiblesse de leur design avant qu'une modification le démontre.

Encore une fois, il est important de noter le problème et de prioriser sa solution en fonction des autres priorités du projet.

Mauvaises odeurs - Réusinage de code (*refactoring*)

Pour tenter de garder les coûts de maintenance raisonnables, il est important de constamment améliorer la base de code (et le système en général).

Il est difficile de justifier un réusinage sans ajout de valeur. Lors de correction de défauts ou lors de l'ajout de nouvelles fonctionnalités, il convient de laisser le code plus propre qu'il ne l'était au départ.

Quelques indicateurs qu'un réusinage est nécessaire:

- duplication de code
- haute complexité
- fort couplage
- un changement demande une opération chirurgicale

Surveillance

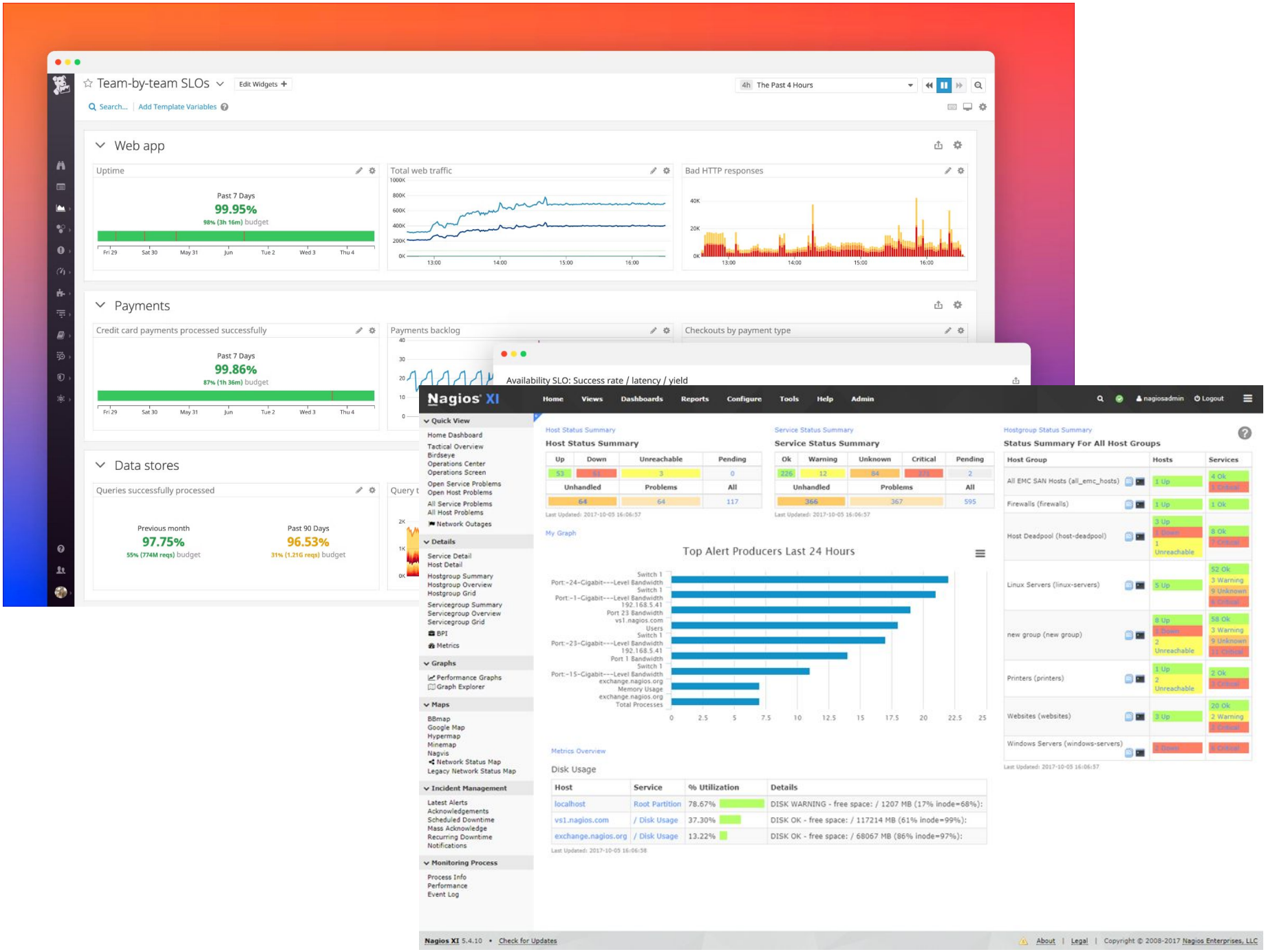
Une grande partie des activités de maintenance consiste à surveiller l'application afin de détecter les problèmes.

La surveillance repose sur des outils spécialisés qui amassent des données sur le système (bande passante, cpu, mémoire, etc) et sur l'application (requêtes par secondes, taille de la heap, etc).

L'objectif de la surveillance est de s'assurer que le système est opérationnel et de détecter rapidement les défaillances.

Pour ce faire, on construit souvent un tableau de bord qui donne une vue globale de l'état du système et on définit des alertes lorsque certaines mesures dépassent un seuil prédéterminé.

Exemples d'outils de surveillance: New Relic, Datadog, Nagios, etc


Team-by-team SLOs
Edit Widgets
Search...
Add Template Variables
4h The Past 4 Hours
Web app
Uptime
Past 7 Days
99.95%
98% (3h 16m) budget
Total web traffic
Bad HTTP responses
Payments
Credit card payments processed successfully
Past 7 Days
99.86%
87% (1h 36m) budget
Payments backlog
Checkouts by payment type
Data stores
Queries successfully processed
Previous month
97.75%
55% (774M reqs) budget
Past 90 Days
96.53%
31% (1.21G reqs) budget
Availability SLO: Success rate / latency / yield
Nagios XI
Home
Views
Dashboards
Reports
Configure
Tools
Help
Admin
nagiosadmin
Logout
Quick View
Home Dashboard
Tactical Overview
Birdseye
Operations Center
Operations Screen
Open Service Problems
Open Host Problems
All Service Problems
All Host Problems
Network Outages
Details
Service Detail
Host Detail
Hostgroup Summary
Hostgroup Overview
Hostgroup Grid
Servicegroup Summary
Servicegroup Overview
Servicegroup Grid
BPI
Metrics
Graphs
Performance Graphs
Graph Explorer
Maps
BBmap
Google Map
Hypermap
Minemap
Nagvis
Network Status Map
Legacy Network Status Map
Incident Management
Latest Alerts
Acknowledgements
Scheduled Downtime
Mass Acknowledge
Recurring Downtime
Notifications
Monitoring Process
Process Info
Performance
Event Log
Host Status Summary
Host Status Summary
Up
Down
Unreachable
Pending
Unhandled
Problems
All
Service Status Summary
Service Status Summary
Ok
Warning
Unknown
Critical
Pending
Unhandled
Problems
All
Hostgroup Status Summary
Status Summary For All Host Groups
Host Group
Hosts
Services
All EMC SAN Hosts (all_emc_hosts)
Firewalls (firewalls)
Host Deadpool (host-deadpool)
Linux Servers (linux-servers)
new group (new group)
Printers (printers)
Websites (websites)
Windows Servers (windows-servers)
My Graph
Top Alert Producers Last 24 Hours
Metrics Overview
Disk Usage
Host
Service
% Utilization
Details
localhost
Root Partition
78.67%
DISK WARNING - free space: / 1207 MB (17% inode=68%):
vs1.nagios.com
/ Disk Usage
37.30%
DISK OK - free space: / 117214 MB (61% inode=99%):
exchange.nagios.org
/ Disk Usage
13.22%
DISK OK - free space: / 68067 MB (86% inode=97%):
Nagios XI 5.4.10 • Check for Updates
About | Legal | Copyright © 2008-2017 Nagios Enterprises, LLC

Surveillance d’une application web

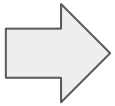
Il est utile de définir un *endpoint* qui permet à différents systèmes de déterminer l’état de santé d’un service.

Par exemple, on peut ajouter un *endpoint* /*health* à l’API de notre service qui retourne 200 OK si tout va bien. Si le *endpoint* ne répond pas dans un temps prescrit, le service est considéré comme mort.

Ceci peut déclencher des actions automatiques comme un rebalancement de la charge, un redémarrage automatique, le déploiement d’une nouvelle instance du service, etc.

Exemple Spring Boot Actuator:

http://127.0.0.1:8080/actuator/health



```
// HTTP 200 OK
{
   "status": "UP"
}
```

Métriques

Savoir qu'un service est fonctionnel ou non est une bonne première étape pour surveiller un système.

Il est encore plus intéressant d'en connaître davantage sur l'état du service.

Pour se faire, il est possible d'exposer des métriques qui seront publiées pour être stockées et consultées.

Il existe différentes approches:

- le service publie ses métriques à interval régulier
- le service expose ses métriques et un système externe vient le lire à intervalle régulier

Les types de métriques

Compteur: Le nombre d'événements (ex. nombre d'appels). La valeur du compteur augmente toujours (sauf au démarrage).

Jauge: Une mesure instantanée (ex. taille de la heap). La valeur de la jauge peut augmenter ou diminuer.

Histogramme: Une répartition des différentes valeurs enregistrée avec calcul de percentiles (ex. taille des messages).

Chronomètre: Mesure un temps d'exécution et le nombre d'appels (ex. temps de réponse).

Le **taux** est souvent calculé par l'agrégateur de métriques en fonction d'un intervalle de temps spécifié.

Par exemple, le taux de requêtes par seconde est calculé à partir d'un **compteur** de requêtes en divisant la variation du compteur pour un intervalle de temps donné (ex. 10s). Donc si le compteur a augmenté de 100 en 10s, le taux est de 10 requêtes par seconde.

Les signaux d’or

Selon le guide de l’ingénierie de la fiabilité des sites de Google (https://sre.google/), il faut observer au minimum les 4 signaux suivant:

- le temps de réponse (séparer succès et échecs)
- la charge
- le taux d’erreur
- la saturation - % de l’utilisation maximale du système

Alertes

Pour s’assurer de détecter et de réagir rapidement lorsqu’une défaillance survient, il est souhaitable de définir une règle automatique qui déclenche une alerte lorsqu’une condition particulière survient (habituellement une métrique qui dépasse un seuil).

L’objectif de l’alerte est d’avertir l’équipe responsable que le système est dans un état qui demande une intervention.

Pour éviter la fatigue (qui conduit à ignorer les alertes), il est important que:

- la ou les actions à prendre pour chaque alerte soient définies
- une alerte doit requérir une action *intelligente* (sinon automatiser la réponse)
- une alerte doit signaler un nouveau problème (pas d’alerte redondante)

Observabilité

L’observabilité est une caractéristique d’un système qui permet d’inférer son état interne à partir des données qu’elle expose.

Les 3 piliers de l’observabilité:

- journaux
- métriques
- traces

La combinaison des trois éléments permet souvent de diagnostiquer les défaillances.

Il est important, lorsque l’on bâtit un système, de penser à **l’observabilité** et de ne pas hésiter à ajouter des métriques et des événements dans les journaux.

Évolution de système

Une application peut évoluer de différentes façons à chacune des nouvelles versions.

Est-ce que la nouvelle version change le modèle de données?

Est-ce que la nouvelle version change l’API d’un service?

Est-ce que la nouvelle version altère le comportement d’un composant utilisé par d’autres composants?

Lorsqu'une application est en service et qu’on désire la modifier, il faut penser aux conséquences des modifications sur les données et les composants existants.

Il faut réfléchir à une stratégie de déploiement qui assure qu’à tout moment, l’application sera fonctionnelle.

Déploiement sans temps mort

Lorsque l'on met à jour une application web (backend) deux options sont possibles pour ne pas occasionner de temps mort (soit pour le système dans son ensemble ou pour un service en particulier):

- Déploiement Bleu/Vert: La nouvelle version est déployée sur un nouvel environnement (bleu) et ensuite le trafic est transféré de l'environnement vert à l'environnement bleu. Cette option demande de doubler (au moins temporairement) les ressources du système, mais elle a l'avantage qu'un retour en arrière est simple et rapide.
- Déploiement graduel: Les instances sont mises à jour graduellement (ex. une à une). Cette option demande d'avoir une instance supplémentaire (redondance) et deux versions d'un même service seront actives en même temps.

Rétrocompatibilité

La rétrocompatibilité assure qu'une nouvelle version est compatible avec les anciennes versions.

Par exemple, si on ajoute un champ optionnel dans le modèle de données, cette modification est rétrocompatible, car les anciennes versions ne vont ni lire ni écrire ce champ et vont donc continuer de fonctionner sans la nouvelle fonctionnalité.

Par contre, supprimer un champ qui était obligatoire n'est pas rétrocompatible, car les anciennes versions qui liront les nouvelles données ne sauront pas comment gérer le cas où la valeur qui était obligatoire n'est pas présente.

Rétrocompatibilité

Comme il est impossible de tout mettre à jour instantanément (application monopage, services, etc), la séquence de déploiement doit être une suite de changements rétrocompatibles.

Comment faire pour effectuer une modification a priori non rétrocompatible sans attendre une nouvelle version majeure?

Exemple:

Comment ajouter un champ obligatoire dans le modèle de données partagé entre plusieurs composants (ex. application monopage, services, base de données)?

À la fin du processus de mise à jour, tous les enregistrements (passés et futurs) devront contenir une valeur valide.

Pensez en termes de *lecture* et d'*écriture*.

Rétrocompatibilité - Exemple

Comment ajouter un champ obligatoire dans le modèle de données partagé entre plusieurs composants (ex. application monopage, services, base de données)?

À la fin du processus de mise à jour, tous les enregistrements (passés et futurs) devront contenir une valeur valide.

1. On déploie une version pour tous les composants où:
   a. en écriture, on écrit le nouveau champ (il est considéré obligatoire)
   b. en lecture, on lit le nouveau champ, mais il est considéré optionnel
2. On migre les données existantes pour ajouter une valeur pour le nouveau champ obligatoire pour tous les enregistrements.
3. À partir de ce moment, tous les enregistrements passés et futurs auront le champ.
4. On déploie une deuxième version où le champ est obligatoire en lecture (simplification du code) et dans la base de données.

Évolution du modèle de données

Si une fonctionnalité demande de modifier le modèle de données, il faut faire attention à la rétrocompatibilité. On suppose ici qu'un lecteur ignore un champ qu'il ne connaît pas.

Changement	**Rétrocompatible**
Suppression d'un champ obligatoire	Non - le rendre optionnel en lecture d'abord
Suppression d'un champ optionnel (ou avec valeur par défaut)	Oui - ignoré en écriture, la valeur par défaut sera utilisée dans l'intérim
Ajout d'un champ obligatoire	Non - le considérer optionnel en lecture d'abord
Ajout d'un champ optionnel	**Oui - on ajoute graduellement en écriture et lecture**
Modification du type d'un champ	Non - requiert plusieurs versions intermédiaires
Changer le nom d'un champ	Non - requiert plusieurs versions intermédiaires

Évolution d’API

De la même manière, lorsque l’on modifie un API, il faut valider que le changement est rétrocompatible.

Changement	**Rétrocompatible**
Suppression d’un *endpoint*	Non - les utilisateurs doivent être mise à jour d’abord
Suppression d’un paramètre	Non - les utilisateurs doivent être mise à jour d’abord
Ajout d’un *endpoint*	**Oui**
Ajout d’un paramètre optionnel	**Oui**
Ajout d’un paramètre obligatoire	Non - on le met optional d’abord jusqu’à ce que tous les utilisateurs soient mis à jour
Changer le URL d’un *endpoint*	Non - il faut dupliquer temporairement et mettre à jour les utilisateurs
Changer le nom d’un paramètre	Non - il faut dupliquer temporairement et mettre à jour les utilisateurs

Indicateurs de fonctionnalité

Les indicateurs de fonctionnalité (*feature flags*) permettent d'activer ou de désactiver une fonctionnalité de l'application sans changement de code (et possiblement sans redémarrage) via une configuration.

Ceci permet d'intégrer des fonctionnalités incomplètes ou risquées graduellement dans la base de code (l'indicateur est inactif). Puis, lorsque l'on désire activer la fonctionnalité, on active l'indicateur sans avoir à déployer de nouvelles versions. Le retour arrière consiste simplement à remettre l'indicateur inactif.

Migration de données

Selon le type de base de données utilisé, différentes options sont possibles pour faire évoluer les données et le schéma (si applicable).

Dans une base de données avec schéma (e.g base de données relationnelle), il faut faire évoluer les données et la définition du schéma. **Chaque évolution devrait être contenue dans un script et versionée.** Le script devrait contenir les modifications du schéma et les opérations à effectuer sur les données existantes. La création d'un script de retour-arrière associé à chacune des migrations est encouragé.

Si possible, les scripts devraient être idempotents, c'est-à-dire que si un même script est exécuté plusieurs fois le résultat reste le même. Ceci implique souvent de vérifier si l'opération à déjà été effectuée avant de l'exécuter.

Plusieurs outils permettent d'automatiser les migrations de données (ex. Flyway). Ces outils maintiennent une table qui sauvegarde les migrations qui ont déjà été appliquées et applique automatiquement les suivantes.

Migration de données

Dans le cas d’une base de données sans schéma, seulement les données sont à modifier.

Dans ce cas, un script idempotent dans un outils de gestion de version est une bonne option.

Dans tous les cas, il faut s’assurer de bien tester les migrations (développement, tests, mise en scène) et de sauvegarder de la base de données avant d’appliquer la migration de données dans un environnement critique.

Migration de système

La migration de système fait aussi partie de l'évolution d'une application web.

Parfois une dépendance atteint une limite qui freine l'évolution du système. Il faut alors migrer vers une autre technologie.

Souvent, cette migration implique non seulement des modifications de code pour s'intégrer avec la nouvelle technologie, mais aussi la migration de données d'une technologie (ancienne) vers une autre (nouvelle).

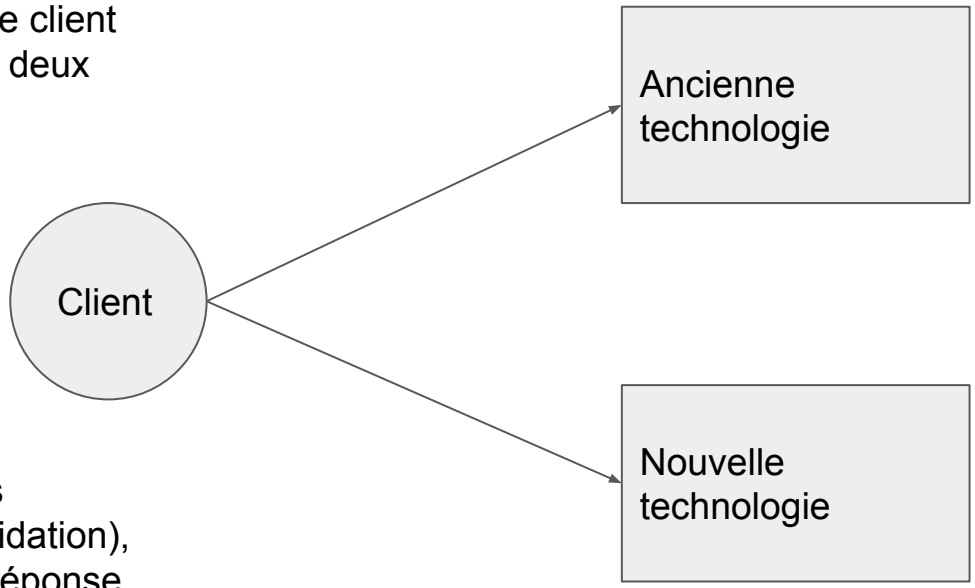
Si la quantité de données est non négligeable, ce processus peut prendre beaucoup temps et il faut aussi considérer que de nouvelles écritures seront effectuées durant la migration.

S'il n'est pas possible de planifier un temps de maintenance pour effectuer la migration, une stratégie de lecture et d'écriture parallèle peut être intéressante. Cette stratégie peut aussi permettre de diminuer les risques liés à l'utilisation de nouvelles technologies.

Stratégie de lecture et d’écriture parallèle

1

Lors de chaque opération de lecture ou d’écriture, le client envoie la requête aux deux technologies.



2

Le client compare les réponses reçues (validation), mais n’utilise que la réponse de la technologie indiquée par un indicateur de fonctionnalité.

3

Ceci permet de tester la validité, les performances et la robustesse de la nouvelle technologie en contexte de production en limitant les risques.

4

Lorsque toutes les données ont été migrées et que la nouvelle technologie a fait ses preuves, on modifie l’indicateur de fonctionnalité pour utiliser les réponses de la nouvelle technologie.