Les micro-services,

Séduisants mais difficiles

11 Mars 2022

claude@lemarson.com
https://www.lemarson.com

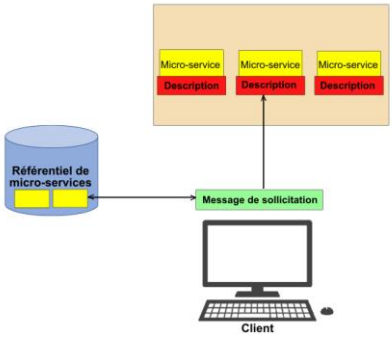
Sommaire

Les micro-services

Le marché quadruple de 2018 à 2022 (8 G$)
A rapprocher du marché global du développement logiciel entre 200 et 240 G$ selon les sources (entre 4 et 5 %).

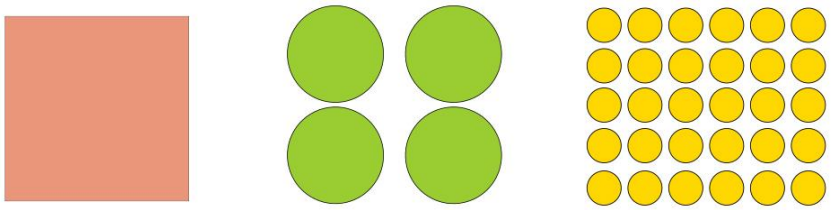
Micro-service : définition

- Un micro-service est un service applicatif de "petite" taille, qui implémente un nombre réduit de fonctions.
- Le terme a été proposé par Peter Rodgers en 2005.
- Ils s'intègrent dans une architecture MSA, distribuée ou non, avec un "core" qui sollicite l'exécution de services via des messages, transportés par des protocoles standards.
- Les services sont exposés dans une bibliothèque de codes prêts à l'emploi.
- Un service peut être métier ou technique (différence avec les SOA).
- L'intégration peut être synchrone ou asynchrone.



- Encapsulation des services : la logique est dissimulée aux appelants.
- Le seul point d'entrée est l'interface : description des traitements, contraintes et données.
- Faible couplage pour réduire les dépendances.
- Contrats de description et découverte dynamique.
- Réutilisation à volonté.
- Autonomie des services ("stateless"), qui ne doivent pas dépendre du contexte d'usage.

SOA non, MSA oui



- D'abord une question de granularité.
- Pas de distinction entre métier et technique.
- Une entreprise peut s'appuyer sur des milliers de micro-services.
- Les grands concepts de bus, d'interfaces et d'orchestration sont les mêmes.
- Machine virtuelle contre container.
- Mais…
  - Le terme micro est relatif, quel est le juste milieu, risque de tomber dans les nano-services.
  - La granularité des SOA pose problème, mais le nombre de micro-services en pose un autre.
  - Comment être informé qu'un micro-service existe : le problème est amplifié par rapport aux SOA.
  - Le codage devient une recherche permanente des micro-services.
  - Difficulté pour maintenir l'indépendance et l'autonomie des micro-services : augmentation de la complexité.

Objectifs et bonnes pratiques du projet

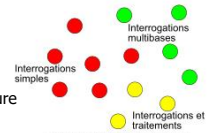
Les objectifs

- Scalability (évolutivité)
- Disponibilité
- Réactivité ("résilience")
- Flexibilité de l'architecture
- Indépendance des services et autonomie
- Gouvernance décentralisée
- Isolation des dysfonctionnements
- Livraison continue via DevOps
- Réutilisation d'au moins 50 % des services
- Réduction des coûts sur le long terme
- Utiliser les meilleurs langages et API en fonction de la nature de chacun des services
- Maintenance plus simple
- Adéquation aux contraintes réglementaires plus facile à établir

Avoir une perception globale du projet
d'urbanisation via les micro-services.
Veiller à ne pas fonctionner en silos et imposer les règles de modélisation et de réutilisation des services existants.

Prendre son temps.
L'urbanisation d'un TI applicatif peut prendre des années, voire des dizaines d'années. Ce n'est jamais un "one shot". Le projet doit s'inscrire dans une politique globale d'urbanisation.

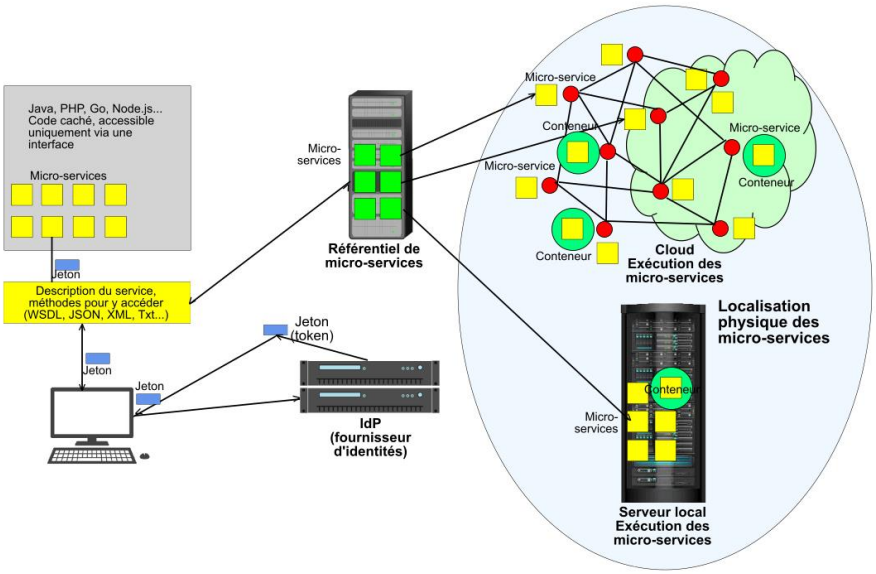
Passer par une phase de modélisation.
Utiliser un langage de modélisation, UML, BPMN, etc, pour décrire les services et leurs interactions.



Définir des groupes d'usages.
Interrogations simples ou multi bases, interrogations et traitements légers de mise à jour, traitements lourds...

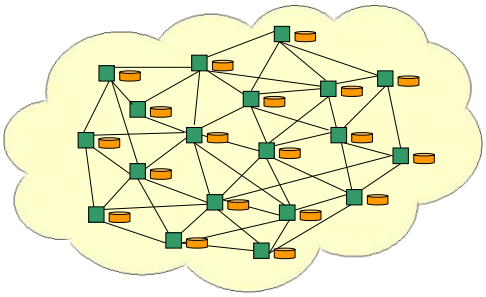
Langage d'orchestration et run-time d'exécution.
Le code des services est caché. Mais l'orchestration est modélisée via un langage type BPEL (XML), dont l'exécution fait appel à un "run time" dédié.

Micro-services : architecture de mise en oeuvre



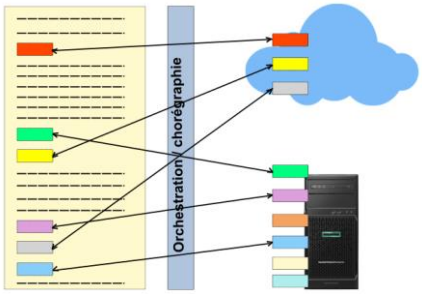
Une autre manière d'imaginer les applications

- Construire un repository de services.
- Modéliser la cinématique de l'application à l'aide d'un langage formel : BPEL (XML), utilisé pour les Web Services ou un autre.
- Tenir compte du fait que les services vont être orchestrés, mais qu'ils ne sont pas nécessairement prêts à l'exécution (il faut les acheminer sur un serveur disponible)
- Il faut prévoir :
  - La découverte des micro-services, via le repository (service normalement natif).
  - Le traitement de l'asynchronisme.
  - La gestion des exceptions, d'autant plus complexe que l'on est dans un mode distribué.
  - La gestion des files d'attentes aux micro-services, selon les modes LIFO (Last In First Out) ou FIFO (First In First Out).
  - Le traitement des formats de messages : XML, JSON, texte, CSV…
  - La prise en charge du chiffrement (confidentialité) et identité des micro-services.
  - La gestion de la dégradation des applications, chaque micro-service pouvant envoyer un état de comportement (heartbeating) au repository pour l'informer de sa situation.
  - La répartition de charge, qui est une conséquence du point précédent, l'idéal étant de disposer d'une API dédiée qui règlera ce problème.
  - La tolérance aux fautes, indispensable pour certains micro-services, pour qui on prévoira des procédures de repli prioritaires.
  - L'enregistrement des évènements de l'application dans un fichier log.



L'enchaînement des micro-services

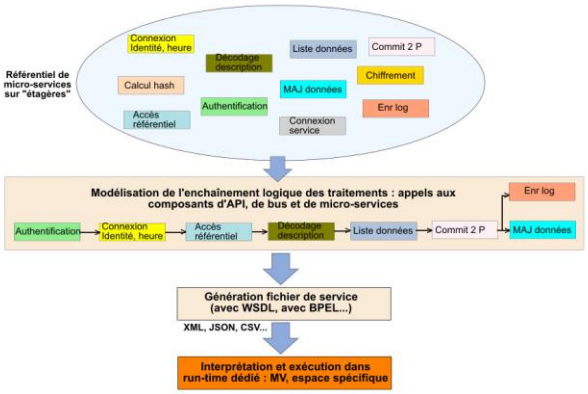
- Même limité en périmètre, un micro-service nécessite de faire un choix d'architecture : 2 manières de l'envisager.
- Le mode chorégraphie induit d'embarquer dans chaque service, les éléments de code pour traiter les fonctions techniques et métiers de base :
  - Interroger le référentiel pour choisir le bon service.
  - S'informer auprès des fichiers de description (référentiel) pour savoir comment s'interfacer avec le service.
  - Se connecter au service, en passant par les interfaces réseaux standard.
  - Coder les sollicitations asynchrones et synchrones.
  - Embarquer les fonctions de confidentialité et d'intégrité.
  - … et de nombreuses fonctions qui alourdissent considérablement l'écriture.
- Le mode orchestration suppose que le service s'adresse à une API de bus pour effectuer toutes les opérations techniques et métiers.
  - Tout dépend de la taille et du nombre de fonctions à traiter. Au-delà de 2 fonctions, il faut passer par le bus (le Gartner préconise 100 …).
  - Prendre son temps pour choisir l'API bus et l'imposer à tous les développeurs. Aucun dérogation ne sera acceptée.



La modélisation de l'orchestration

L'exemple de BPEL (XML)

- Le modèle BPEL d'un processus métier est un processus en 4 étapes, chacune d'elles étant symbolisée par un document XML:
  - Inventaire de tous les composants impliqués dans le processus métier.
  - Définition de l'interface WSDL du processus : sa présentation vis-à-vis de l'extérieur.
  - Définition les liens partenaires, c'est-à-dire l'interaction entre le processus BPEL les services auxquels il a accès et les clients du service BPEL.
  - Création du processus métier BPEL, avec la description des variables et l'enchaînement dynamique des opérations.
- Le processus est composé de tâches qui peuvent être primitives ou de structure.
  - Primitives :
    - **<invoke>** pour appeler un service.
    - **<assign>** pour manipuler les variables.
    - **<throw>** pour gérer les exceptions.
    - **<terminate>** pour conclure un service.
  - Structure : exemples
    - **<sequence>** pour définir l'enchaînement ordonné de services.
    - **<flow>** pour définir un ensemble de services pouvant s'exécuter en parallèle.
    - **<switch>** pour implémenter des branchements conditionnels.
    - **<while>** pour les boucles d'enchaînement de services.



Les problèmes spécifiques

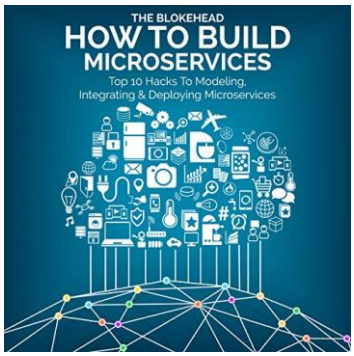
Partir de l'existant

- Il faut s'appuyer sur l'existant
- Dans 95 % des cas, il existe déjà un SI : on ne part pas d'une page blanche
- Certaines fonctions existent déjà, complètes ou partielles
- Rares sont les entreprises qui font l'effort d'inventaire en amont (7 % des responsables de TI ne savent pas quelles sont les applications dont ils disposent et si toutes ont encore des clients… Ils ne font pas d'analyse sérieuse…)
- Deux solutions
  - On les redéveloppe indépendamment
  - On les isole et on les enveloppe de manière à ce qu'elles soient accessibles par un code externe via une interface
- Problème : il est très rare que les responsables de TI aient une approche globale d'un projet de micro-services
- Approche limitée à un périmètre, qui vient se juxtaposer aux autres
  - On ne se préoccupe pas de sa cohérence avec l'existant
  - D'où, incompatibilités et redondance

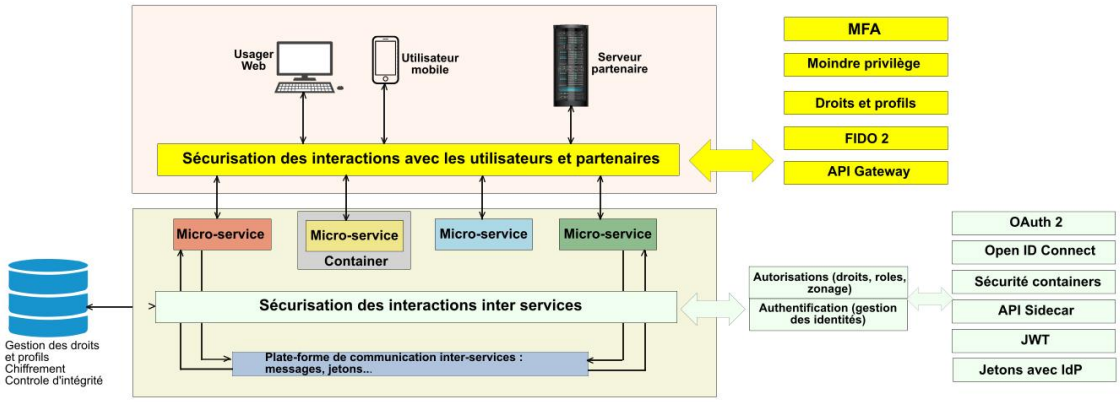
Le découpage en services et modélisation

Le vrai problème des micro-services

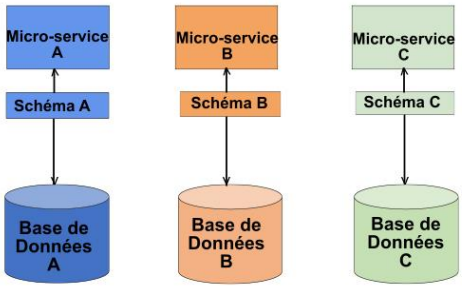
- Il n'y a pas de méthode particulière, seulement des bonnes pratiques.
- Expérience métier recherchée…
- Il faut définir des groupes d'usage, 3 ou 4 : il est impossible de décliner une stratégie pour chaque service : sécurité, disponibilité, risques, confidentialité…
- Adéquation du périmètre fonctionnel : il faut limiter les responsabilités (une seule fonction, fonctions regroupées qui ont un lien fonctionnel)
  - Concept de "cohésion forte" : il ne faut pas associer des fonctions qui n'ont pas de lien logique ou métier.
- Les services doivent être faiblement couplés. Ils ne dépendent pas d'autres services, mais peuvent communiquer avec eux (principe du "srtateless").
  - Réduire au maximum la dépendance, liée à la granularité du service : plus le service est petit, moins il génèrera de dépendances avec d'autres services.
  - Eviter de partager systématiquement les API.
  - Eviter le partage des bases de données … mais le schéma "une base : un micro-service" est impossible à appliquer.
  - Réduire les communications synchrones, qui bloquent le réseau.
  - Réduire le partage des infrastructures : réseaux…
  - Eviter les API qui dépendent d'une implémentation ou d'une technologie propriétaires.



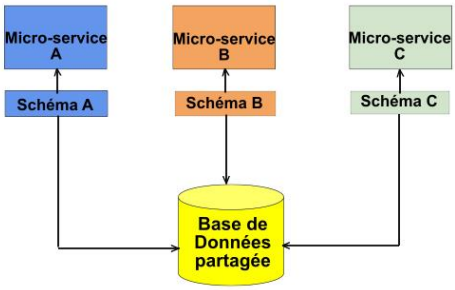
Les problèmes spécifiques : Sécurité



Les problèmes spécifiques : L'accès aux données



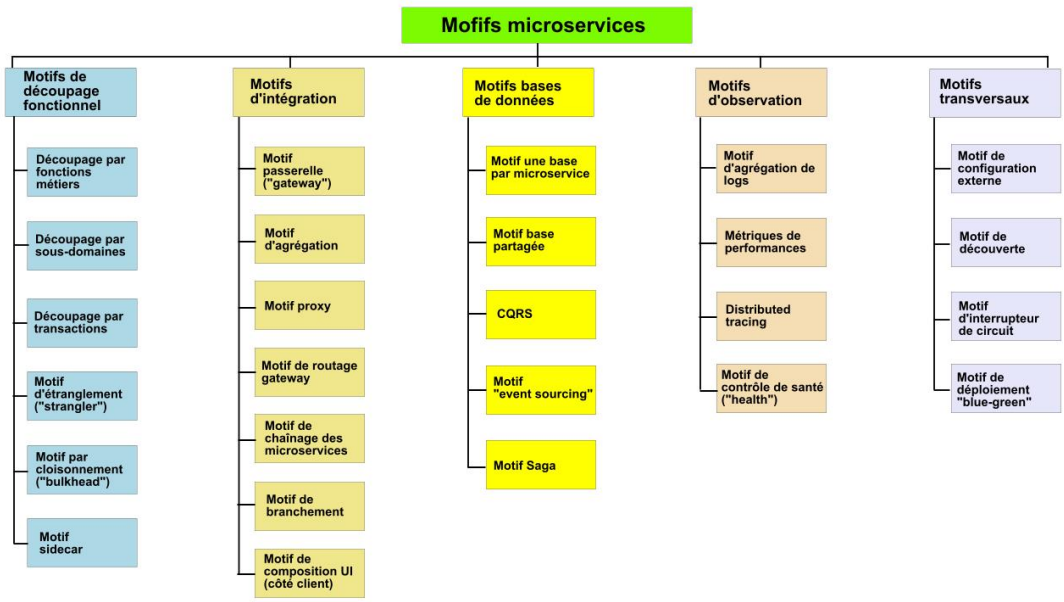
Une base de données par micro-service



Une base de données partagée entre plusieurs micro-services avec un seul schéma d'accès ou un schéma par micro-service

- Deux architectures principales :
- Une base par micro-service
  - Les changements apportés à une base n'impactent pas les autres.
  - L'application est plus réactive ("resilience").
  - On peut implémenter des technologies différentes : SQL; NoSQL…
  - Le "scaling" est plus facile.
- Une base unique partagée
  - "Scaling" difficile.
  - Les changements apportés à une base impactent plusieurs micro-services : perte d'indépendance et couplage plus élevé.
  - Il y a convergence des accès physiques qui peuvent dégrader les performances.
  - Les sauvegardes et récupération sont plus simples.
- Plusieurs motifs sont dédiés : SAGA, CQRS, Event Sourcing, Composition

Les patterns (motifs de conception)



Les plates-formes de développement

Les acteurs clés des micro-services

- IBM, Microsoft, Oracle, Salesforce, Mulesoft, Software AG, CA Technologies, Syntel, Datrawire

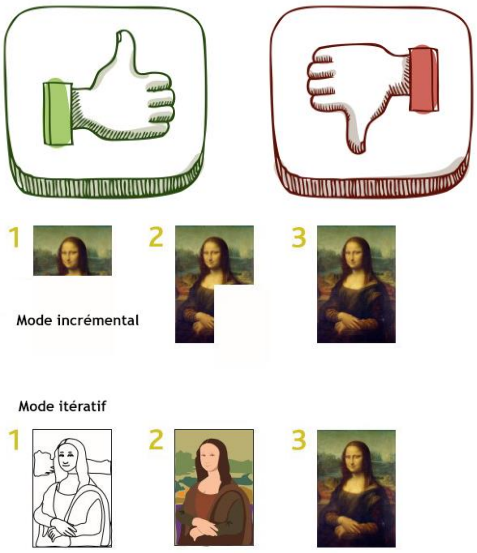
Avantages et inconvénients

Avantages (espérés)

- Support des architectures incrémentales.
- Bénéficient de tous les avantages du développement moderne : Cloud, containers, DevOps, Serverless.
- Scaling horizontal.
- Modularité et "remplaçabilité" (?).
- Très forte réduction des temps de mise à jour et maintenance.
- Diversité des technologies à l'intérieur d'une MSA
- Indépendance lié au "sans état" : un micro-service peut communiquer avec un autre, mais il ne doit pas en être dépendant.
- Indépendance du déploiement.
- Résilience (réactivité) et disponibilité : si un micro-service tombe, on peut faire en sorte que l'ensemble ne s'écroule pas.

Inconvénients

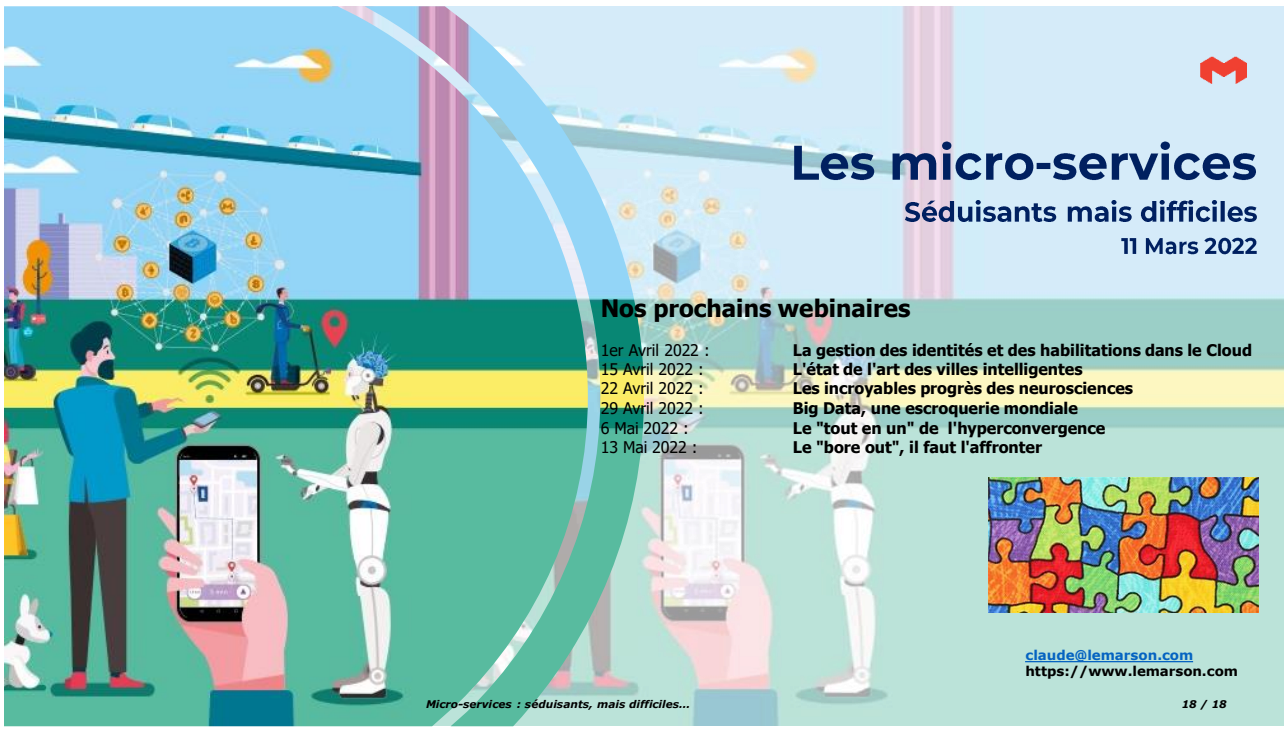
- Globalement la conception d'une MSA, même limitée, est plus difficile que pour une architecture monolithique.
- Un grand nombre de participants : API, containers, bases de données, micro-services…
- La complexité est transférée du code vers l'infrastructure.
- Quel que soit le protocole de sollicitation, RPC, gRPX, Soap… les appels prolifèrent, ce qui peut s'avérer incompatible avec le réseau.
- La gestion globale de la sécurité est plus compliquée.
- On a généralement plusieurs référentiels de micro-services et on a du mal à s'y retrouver.
- On ne peut pas se lancer dans les micro-services sans se faire aider par des mécanismes de mise en production DevOps et CI/CD. Manuellement c'est impossible.
- La mise au point est plus complexe, du fait des micro-services répartis dans une architecture distribuée.
- La consistance des données est plus difficile à obtenir, on n'est plus sous la protection du bouclier ACID.
- Généralement très coûteux sur le long terme.



Les risques d'urbaniser de manière inconsidérée

- Les micro-services ne sont pas une fatalité.
- Les risques sont élevés de s'obstiner sur une architecture qui ne s'imposerait pas.
- Le système d'information applicatif est-il suffisamment étendu pour nécessiter une urbanisation à base de micro-services ?
- Les transactions ont-elles besoin de communiquer avec d'autres micro-services ?
- Si oui, ce n'est pas sans danger :
  - Les blocages peuvent entraîner une grande complexité.
  - Certains protocoles (REST), sont sans état et ne dépassent pas les limites d'une transaction.
  - Le 2PC ("Commit à 2 Phases") est très handicapant pour les hautes performances (le motif SAGA que l'on utilise en substitution, ajoute aussi une forme de complexité.
- La distribution des applications est coûteuse, qui nécessite de passer par DevOps.
- Certaines applications sont tellement intégrées (anciennes) que le découplage peut s'avérer catastrophique.
- On manque souvent d'expérience : l'urbanisme par micro-services est un mix de métier (utilisateurs) et technique (TI).
- Les tests de bout en bout sont difficiles à réaliser.
- Avec le temps, les référentiels sont redondants et foisonnants.
- Financièrement, les micro-services peuvent s'avérer des gouffres.

On est pas obligé de se précipiter sur cette architecture sous prétexte que LeMarson et le Gartner en parlent…



Les micro-services

Séduisants mais difficiles

11 Mars 2022

Nos prochains webinaires

1er Avril 2022 :	**La gestion des identités et des habilitations dans le Cloud**
15 Avril 2022 :	**L'état de l'art des villes intelligentes**
22 Avril 2022 :	**Les incroyables progrès des neurosciences**
29 Avril 2022 :	**Big Data, une escroquerie mondiale**
6 Mai 2022 :	**Le "tout en un" de l'hyperconvergence**
13 Mai 2022 :	**Le "bore out", il faut l'affronter**

claude@lemarson.com
https://www.lemarson.com