Architecture applicative - Études de cas

Virginie Galtier

7 avril 2026

Ce dossier regroupe des études de cas utilisées dans le cadre de l'enseignement « architectures applicatives ». Elles ont vocation à illustrer et compléter les concepts présentés en cours, à nuancer la dichotomie théorique grâce aux situations réelles, et à amener des questionnements généralisables à d'autres propositions architecturales.

Table des matières

1 Passer aux microservices

1.1 Un exemple de transition aux microservices qui « explose » chez Segment

1.1.1 Document

Le document proposé est un résumé de l'exposé intitulé « *To Microservices and Back Again* » présenté par Alexandra Noonan à la conférence Qcon Plus du 26 mai 2020 à Londres.

La vidéo est visible sur YouTube : `https://www.youtube.com/watch?v=hIFeaeZ9_AI`

Choisir de passer à une architecture microservices est une tentation à laquelle presque toutes les équipes finissent par succomber, quelle que soit la taille de l'entreprise ou du produit : on trouve énormément de contenus qui vous mettent en garde sur les compromis, mais on a vite l'impression que la productivité décline parce qu'un monolithe ne vous donne pas l'isolation des pannes et la modularité dont vous rêvez, et on se dit que les microservices vont être la réponse ; pourtant, si on les implémente de travers, ou si on les utilise comme un pansement sans corriger les défauts de fond du système, on finit par ne plus pouvoir développer de nouvelles fonctionnalités produit parce qu'on se noie dans la complexité. Chez Segment, nous avons décidé de casser notre monolithe en microservices environ un an après le lancement, mais cela n'a pas réglé certains défauts structurels de notre système, et après environ trois ans à continuer d'empiler des microservices nous étions submergés ; en 2017, nous avons pris du recul et décidé de revenir à un monolithe, parce que les compromis associés aux microservices créaient plus de problèmes qu'ils n'en résolvaient, et parce que ce retour au monolithe nous a donné la possibilité de corriger des défauts fondamentaux. Mon objectif aujourd'hui est que vous compreniez ce que ces compromis impliquent vraiment, pour pouvoir décider de ce qui est juste pour votre équipe.

Segment, pour donner du contexte, c'est un pipeline de données : nous ingérons des centaines de milliers d'événements par seconde, des payloads JSON qui contiennent des informations simples sur des utilisateurs et leurs actions ; ces événements sont générés par le logiciel construit par nos clients, et comme beaucoup d'entreprises, nos clients veulent comprendre comment leurs utilisateurs se servent de leur produit ; il existe des outils comme Google Analytics, Mixpanel, Salesforce, etc., mais le problème, c'est que pour chaque outil, vous devez ajouter du code à plusieurs endroits de votre produit pour envoyer les données à l'API correspondante, et au fil du temps vous finissez avec une sorte de maillage de sources et d'outils, avec des données incohérentes d'un outil à l'autre, un risque de fuite de données sensibles, et une

difficulté permanente dès que vous voulez tester un nouvel outil. Segment vise à simplifier cela en fournissant une API unique : vous instrumentez une fois, et nous nous chargeons de relayer les événements vers les outils finaux que vous choisissez. Dans l'exposé, je me concentre sur ces outils finaux, que nous appelons des « destinations », et sur la façon dont nous leur acheminons les événements.

En 2013, au lancement, nous sommes partis sur un monolithe parce que nous avions besoin du faible coût opérationnel d'un monolithe : nous étions essentiellement les fondateurs, et nous voulions l'infrastructure la plus simple pour itérer ; l'architecture de départ était : une API unique qui ingère les événements et les pousse dans une file de messages distribuée, puis un unique agent (*worker*) monolithique en bout de chaîne qui consomme la file ; pour chaque événement, il regarde la configuration gérée par le client pour savoir quelles destinations sont activées, il transforme l'événement au format attendu par chaque destination, il envoie des requêtes sur Internet vers les APIs des destinations, il attend les réponses, et il passe à la suite. Les requêtes vers les destinations échouent souvent, donc nous avons distingué les erreurs *non réessayables*, celles dont on sait qu'elles ne seront jamais acceptées (parce que les identifiants sont invalides, ou qu'un champ requis est manquant par exemple), et les erreurs *réessayables*, c'est-à-dire celles qui pourraient passer plus tard sans modifier l'événement ; notre mécanisme de re-essai (*retry*) consistait à remettre l'événement dans la même file « avec tout le monde », et à réessayer un certain nombre de fois (typiquement 2 à 10) avant d'abandonner. C'est là que nous nous sommes heurtés à un problème majeur : le *head-of-line blocking*[1], c'est-à-dire le fait qu'une file premier-entré-premier-sorti est bloquée par ce qui est en tête. Si vous regardez la file, vous trouvez des nouveaux événements mélangés à des événements en seconde chance (*retry*), pour tous les clients et toutes les destinations. Quand une destination comme Salesforce a une panne temporaire, tous les événements destinés à Salesforce échouent, retournent dans la file, et créent un afflux massif. Nous avions de l'*auto-scaling* pour ajouter des agents de traitement, mais cette inondation pouvait dépasser notre capacité à monter en charge assez vite, ce qui entraînait des retards non pas seulement pour Salesforce, mais pour toutes les destinations et tous les clients. Or nos clients dépendent de la fraîcheur des données, et nous ne pouvions accepter ces délais nulle part dans la chaîne. Après environ un an de production, l'*overhead* opérationnel du monolithe était faible, mais l'absence d'isolation entre destinations nous coûtait trop : environ 10 % des requêtes vers destinations échouaient en « *réessayable* », et un client qui n'utilisait même pas Salesforce pouvait être impacté par une panne de Salesforce !

C'est ce problème qui nous a conduits, en 2014, à passer aux microser-

1. effet de file d'attente où, dans une file FIFO, un élément en tête (lent, bloqué ou en échec) retarde tout ce qui suit, même si le reste pourrait être traité rapidement

vices, parce que l'isolation « environnementale » vient naturellement quand on découpe. Notre nouvelle architecture a été : l'API ingère les événements et les transmet à un routeur ; le routeur récupère les réglages du client pour savoir quelles destinations cibler, puis il duplique l'événement et l'envoie vers une file par destination ; au bout de chaque file, un agent spécifique à la destination traite les messages. Ça nous a aidés à ce moment-là pour plusieurs raisons : d'abord, nous voulions éliminer l'effet « tout le monde souffre » du *head-of-line blocking*, et avec une file par destination, si une destination a des problèmes, seule sa file se remplit et les autres continuent ; ensuite, nous pouvions ajouter des destinations très vite, ce qui était essentiel dans une phase où les demandes commerciales arrivaient sans arrêt : « un gros client veut Marketo, vous ne le supportez pas », et nous pouvions répondre en ajoutant une file et un agent Marketo, sans mettre en péril les autres destinations. Dans ce travail, la complexité est surtout dans la compréhension des APIs de destinations et de leurs cas limites, parce que chaque destination attend un format spécifique, certaines transformations sont triviales, d'autres très compliquées, certaines destinations exigent même du XML, et en plus les réponses ne sont pas toujours « propres » : je peux recevoir un `HTTP 200`, « *success : true* », et découvrir dans le contenu une erreur « *not found* », donc il faut du code sur mesure pour interpréter correctement succès/échec et décider de re-essayer ou non. Un autre bénéfice que nous avons eu avec cette architecture, c'est une visibilité opérationnelle plus simple : quand tout est séparé, on identifie plus facilement quelle destination est responsable d'une fuite mémoire, et la profondeur de file devient un signal très direct. Cette visibilité est certes possible dans un monolithe, mais qu'elle ne vient pas « gratuitement », immédiatement, il faut investir pour l'obtenir.

Au début, nous avions gardé tous les services dans un unique dépôt (*mono-repo*), chaque destination étant dans son sous-répertoire avec son code et ses tests ; mais ça créait des frictions du type « je change Salesforce et les tests Marketo cassent », et je dois réparer Marketo pour livrer Salesforce[2]. Pour retrouver une forme d'isolation, nous avons éclaté les destinations en un dépôt par destination... mais nous avons découvert plus tard que c'était une fausse bonne idée, un avantage illusoire qui allait nous revenir en pleine figure. En 2015, nous n'étions qu'une dizaine d'ingénieurs et ça allait encore : l'isolation était bonne, la modularité semblait meilleure, la visibilité était confortable, et l'*overhead* opérationnel n'était pas encore un problème parce que nous n'avions qu'une vingtaine de destinations. Puis, en 2016, le produit a pris de la traction, nous sommes entrés en hyper-croissance, les demandes d'ajout de destinations se sont accélérées, et en 2017 nous avions ajouté plus de 50 destinations supplémentaires : cela voulait dire plus de 50

2. Quand tout le code vit dans un seul dépôt, l'état global des tests peut bloquer un changement local si les tests ne sont pas bien isolés ou si la politique est « tout doit être vert ».

services, files et dépôts en plus, et une explosion de la maintenance, parce que chaque destination a ses transformations, ses conventions, ses réponses « bizarres », et en plus notre événement Segment accepte beaucoup de variantes (par exemple des champs pouvant être en camelCase, snake_case, ou décomposés en plusieurs attributs), ce qui entraîne une duplication de logique dans le code de chaque destination. Pour réduire cette duplication, nous avons écrit des bibliothèques partagées : par exemple, plutôt que de réimplémenter partout l'extraction du nom d'un utilisateur, je peux appeler `event.name` et la bibliothèque gère les différents cas ; ça a rendu le code plus uniforme et plus facile à maintenir... jusqu'au jour où il fallait corriger un bug dans une de ces bibliothèques. Et là, le coût réel de notre architecture est apparu : publier une nouvelle version d'une bibliothèque, c'est une chose, mais la déployer partout en est une autre ; notre infrastructure était sur AWS, gérée via Terraform, et même si, quand tout va bien, on peut sortir un changement dans un service en une heure, à l'échelle de dizaines puis de plus de cent destinations, une correction minuscule pouvait devenir une semaine de travail, ou un effort collectif pénible [3]. Résultat : soit nous arrêtions d'améliorer les bibliothèques partagées même quand il le fallait, soit nous faisions des mises à jour « au cas par cas », et les versions ont commencé à diverger entre les dépôts, ce qui a détruit le bénéfice d'uniformité : les destinations se sont retrouvées avec des versions différentes des mêmes bibliothèques, et la complexité a explosé. En parallèle, nous avons été frappés par un autre problème : certaines destinations traitaient très peu de trafic, jusqu'au jour où un gros client décidait de les activer, ce qui inondait la file et déclenchait des alertes ; je me retrouvais à devoir faire passer à l'échelle en urgence ces petites destinations parce que la montée en charge dépassait notre capacité de réaction, et cela arrivait fréquemment. Nous avions une règle d'*auto-scaling* commune à tous les services, mais chaque destination a un profil CPU/mémoire différent, donc il n'y avait pas une configuration magique qui marche pour toutes ; sur-provisionner un pool minimum coûte cher, et avoir des règles par destination ajouterait encore de la complexité alors que nous étions déjà submergés. Avec la cadence d'ajout (environ trois destinations par mois), l'*overhead opérationnel* augmentait linéairement avec chaque nouvelle destination, et sans croissance proportionnelle de l'équipe, la productivité a chuté : la gestion de cette infrastructure est devenue une taxe énorme, nous perdions du sommeil, nous devenions « insensibles » aux alertes, et l'organisation le voyait. Pire, l'*overhead* a fini par bloquer le développement produit : par exemple, une demande très classique était de donner aux clients une visibilité sur la réussite des envois vers les destinations, parce que notre produit était une boîte noire ; dans notre monde microservices, une

3. Pour que la correction soit effective en production, il faut, pour chaque destination : mettre à jour la version de la bibliothèque utilisée (changer la dépendance), relancer les tests, construire/déployer le service...

approche aurait été de construire un mécanisme commun de métriques ou de suivi que toutes les destinations utiliseraient, mais nous savions que cela impliquerait de déployer et tester massivement, donc toute nouvelle fonctionnalité ressemblait à « ajouter encore de la complexité » ; et surtout, nous avons fini par admettre que les microservices n'avaient pas résolu le problème structurel de *head-of-line blocking* : il existait toujours, simplement « déplacé » au niveau de chaque destination, puisque les files d'une destination mélangeaient encore événements frais et événements en re-essai, et que la limitation de débit d'un client pouvait provoquer des retards pour tous les clients utilisant cette destination. L'architecture idéale pour supprimer réellement cet effet aurait été une file et un agent de traitement par client et par destination, mais cela nous aurait menés à des dizaines de milliers de files et de services, irréaliste. En 2017, avec plus de 140 services/files/repos, nous avons atteint le point de rupture ; nous avons fait venir un ingénieur très senior, et ce qu'il a dessiné, c'était un bateau avec un énorme trou et trois ingénieurs en train d'écoper : c'est exactement ce que ça faisait.

Nous avons donc décidé de revenir à un monolithe, en identifiant l'*overhead* opérationnel comme cause racine, mais sans revenir naïvement à l'architecture initiale, parce que remettre « un agent de traitement monolithique » au-dessus de multiples files aurait été une complexité qu'on ne voulait pas, et revenir à une file unique nous aurait remis dans le même piège qu'au lancement. C'est là qu'est née l'idée de Centrifuge : Centrifuge remplace nos files et s'occupe d'acheminer les événements vers un unique agent monolithique, en se comportant comme s'il existait conceptuellement une file par client par destination, mais sans exposer la complexité externe correspondante ; Centrifuge a enfin résolu le *head-of-line blocking* « une bonne fois pour toutes », et l'équipe *destinations* (une douzaine de personnes) s'est dédiée à construire Centrifuge et cet agent. En même temps, puisque nous revenions à un seul service, nous voulions revenir à un dépôt unique, ce que nous avons appelé la « *great mono-gration* » : nous avons réparti les destinations entre nous et commencé à les rapatrier ; et nous avons profité du mouvement pour corriger deux choses : d'abord, remettre tout le monde sur les mêmes versions de dépendances, parce qu'à ce moment-là nous avions environ 120 dépendances uniques et nous voulions une version unique pour les dépendances partagées ; ensuite, reconstruire une stratégie de tests viable, parce que notre dette venait en grande partie de tests qui faisaient de vraies requêtes HTTP vers les APIs des destinations, avec des identifiants de test invalides qui faisaient échouer des tests, et des APIs parfois lentes : à l'échelle de 140 destinations, une suite de tests pouvait durer une heure et c'était inacceptable. Nous avons donc construit *traffic recorder* : à la première exécution, il enregistre requêtes et réponses dans des fichiers, et aux exécutions suivantes il rejoue ces échanges localement au lieu d'appeler l'API distante ; les fichiers sont versionnés avec le code, ce qui rend les tests plus déterministes et résilients ; je me souviens du moment où j'ai lancé les tests de toutes les destinations

et où ça a pris des millisecondes alors qu'avant c'était des minutes, et ça a rendu le dépôt unique praticable. Nous avons terminé la *mono-gration* à l'été 2017, puis nous avons migré progressivement vers Centrifuge et vers l'agent monolithique, avec une migration complétée début 2018 ; et globalement, les gains ont été énormes : la mise à l'échelle est devenue plus simple parce que le grand réservoir d'agents mélange naturellement des destinations gourmandes en CPU et en mémoire, ce qui absorbe mieux les pics ; nous ne nous faisions plus réveiller à cause de petites destinations qui explosent soudainement en charge ; l'*overhead* opérationnel n'augmentait plus à chaque nouvelle destination ; la productivité a bondi parce qu'une modification dans une bibliothèque partagée ne nécessitait plus le déploiement de 140 services : un seul déploiement fait en quelques minutes ; les versions des dépendances étaient unifiées, et les tests étaient rapides, donc nous pouvions à nouveau livrer des produits. Évidemment, ce n'était pas « tout rose » : nous avons payé des compromis, notamment la perte de visibilité et d'isolation « gratuites » qu'on avait avec des microservices ; quelques mois après le retour au monolithe, nous avons eu des crashs mémoire, et alors qu'avant nous aurions su immédiatement quelle destination était coupable, nous avons dû reconstruire des outils d'observation (Sysdig [4], heap dumps [5]) pour identifier le composant responsable, ce qui a pris du temps ; et l'isolation environnementale est devenue un vrai sujet, parce que des exceptions non interceptées peuvent faire tomber l'agent de traitement, et combinées avec les re-essais, cela peut créer des effets en cascade, une pression accrue, et une diminution des ressources disponibles dans le réservoir si l'infrastructure ne redémarre pas assez vite ; face à cela, notre réaction instinctive a souvent été « on devrait redécouper en microservices », mais nous avons appris que ça ne traite pas la cause racine, ça réduit seulement l'étendue, et nous avions déjà été brûlés par l'*overhead* ; nous avons donc plutôt construit un *wrapper* jouant le rôle de middleware entre l'application et l'infrastructure, capable d'intercepter certaines erreurs, de redémarrer proprement, d'empêcher le réservoir de s'effondrer, et de produire des métriques et alertes pour ensuite corriger le code.

La leçon que je veux laisser, c'est qu'il n'y a pas de solution miracle : tout est affaire de compromis et de compréhension de la cause racine ; si nous avions démarré en microservices dès 2013, il est possible que l'*overhead* nous aurait empêchés de décoller, puis nous avons eu besoin d'isolation à un moment où nous n'avions ni les ressources ni le temps de construire la vraie correction (Centrifuge) ; plus tard, quand nous avons eu les ressources, les microservices étaient devenus un frein majeur parce que nous manquions

4. Outil d'observabilité « bas niveau » pour systèmes Linux. Il permet de capturer et analyser ce que fait un processus en observant les appels système : ouvertures de fichiers, réseau, allocations, etc.

5. Snapshot de la mémoire (heap) de Node.js à un instant t, enregistré dans un fichier pour analyse.

d'outillage et que l'*overhead* tuait la productivité ; nous sommes revenus au monolithe, mais avec une stratégie explicite pour chaque compromis (tests, isolation, visibilité), et avec une solution qui cible réellement le défaut structurel de la file ; et si vous ne retenez qu'une chose, c'est qu'aucune mode architecturale ne remplace l'analyse lucide de votre problème et le fait de choisir, à un instant donné, ce qui est supportable à long terme pour votre équipe.

1.1.2 Pistes de lecture

Objectif de lecture : être capable d'expliquer pourquoi Segment passe aux microservices, pourquoi ça marche un temps, pourquoi ça s'effondre, et pourquoi / comment ils reviennent au monolithe (et ce qu'ils gagnent / perdent).

Déclencheur de la migration Identifiez le problème initial du monolithe : quel mécanisme crée des retards « pour tout le monde » quand une seule destination a un incident ?

Architecture avec les microservices : principe et bénéfices attendus Qu'est-ce qui motive explicitement le passage aux microservices ? Dans la version microservices, quel est le rôle du *routeur*, et en quoi « une file + un agent par destination » change le comportement du système ? Quels bénéfices concrets cherchent-ils ?

Causes de l'explosion de la charge Qu'est-ce qui, avec la croissance du nombre de destinations, fait exploser la charge d'exploitation et la complexité du code ? Relevez les signaux de « point de bascule ».

Bibliothèques partagées : de solution à problème Pourquoi les bibliothèques partagées, censées réduire la duplication, finissent-elles par aggraver le problème dans leur organisation (déploiement, versions divergentes) ?

Files FIFO Dans une architecture de traitement par files, comment le découpage en microservices affecte-t-il le risque de « *head-of-line blocking* » ? Pourquoi l'isolation « idéale » au niveau client × destination (une file et un agent dédiés) éliminerait le blocage mais devient impraticable à grande échelle ?

Retour au monolithe Pourquoi décident-ils de revenir au monolithe : quel coût devient dominant, au point d'écraser tous les bénéfices ?

Apport de Centrifuge Qu'est-ce que Centrifuge apporte, et quel problème structurel est-il censé régler dans la chaîne ?

Tests Qu'est-ce que la « mono-gration » change sur les dépendances et sur les tests ? Pourquoi « traffic recorder » est un élément clé pour rendre le retour au dépôt unique viable ?

Le monolithe en mieux Après retour au monolithe, quels problèmes réapparaissent et quels mécanismes les architectes de Segment mettent-ils en place pour compenser ?

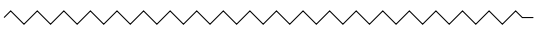
Synthèse Quelles leçons générales tirent-ils sur le fait que « microservices vs monolithe » n'est pas une solution miracle mais un arbitrage à faire selon les causes racines, l'outillage et la taille de l'équipe ?

1.2 Un exemple de transition aux microservices réussie chez GitHub

1.2.1 Document

Le document proposé est un résumé de l'exposé intitulé « From Monolith to Microservices » présenté par Sha Ma à la conférence Qcon Plus du 9 avril 2021.

La vidéo est visible sur YouTube : `https://www.youtube.com/watch?v=Sr0OE-AzAJU`



Je m'appelle Sha Ma. Je suis Vice-Présidente Software Engineering chez GitHub, responsable de la plateforme cœur et des produits de l'écosystème. Avant GitHub, j'étais VP engineering chez SendGrid et j'ai fait partie de l'équipe de direction ayant conduit l'entreprise à l'introduction en bourse en 2017. Je vais présenter le parcours récent de GitHub vers une architecture microservices.

GitHub a été fondé en 2008 pour faciliter l'hébergement et le partage de code. Les fondateurs étaient très impliqués dans l'open source et dans la communauté Ruby, et l'architecture de GitHub est donc historiquement ancrée dans Ruby on Rails. Au fil des années, nous avons recruté d'excellents développeurs Ruby et nous avons continuellement optimisé et fait évoluer ce codebase [6]. Aujourd'hui, GitHub compte plus de 50 millions de développeurs,

6. ensemble du code source d'un logiciel, organisé en fichiers et modules, avec ses dépendances

plus de 80 millions de pull requests fusionnées[7] par an, et plus de 100 millions de repositories[8] sur tous les continents. Cette trajectoire montre qu'un monolithe peut aller très loin : un codebase vieux de plus de 12 ans, des « trains de déploiement » coordonnés[9] avec plusieurs déploiements par jour, une plateforme très sollicitée servant plus d'un milliard d'appels API par jour, et une interface utilisateur performante orientée efficacité. En interne, GitHub a connu une forte phase de croissance au cours des 18 derniers mois. Avec plus de 2000 employés, nous avons plus que doublé le nombre d'ingénieurs contribuant au code. Nous avons grandi à la fois organiquement et par acquisitions (Semmle, npm, Dependabot, Pull Panda). GitHub est aussi une entreprise très distribuée : avant la pandémie, plus de 70 % des employés étaient hors du siège de San Francisco. Nous collaborons sur six continents, dans tous les fuseaux horaires.

Avec plus de 1000 développeurs internes, avec des compétences diverses et un large éventail de technologies, il est devenu clair que nous devions repenser notre façon de développer. Exiger que tout le monde apprenne Ruby avant d'être productif, et faire contribuer tout le monde dans un même codebase monolithique, n'est plus la voie la plus efficace pour continuer à grandir. On peut invoquer la loi de Conway : une organisation qui conçoit un système produit un design dont la structure reflète la structure de communication de l'organisation. Cela marche aussi dans l'autre sens : un environnement monolithique conduit à de grandes réunions avec de nombreuses parties prenantes et à des processus de décision plus complexes, à cause de la logique imbriquée et des données partagées qui affectent toutes les équipes. Cela nous a conduit à nous demander : est-ce le moment de migrer d'un monolithe Ruby on Rails vers une architecture microservices ? Et si oui, comment le faire ? Les deux architectures ont des avantages. Dans un monolithe, il est plus simple d'être opérationnel rapidement, sans se battre avec des dépendances distribuées. Un nouvel employé peut lancer GitHub localement en quelques heures. Il y a aussi une simplicité au niveau du code : pas besoin d'ajouter de logique pour gérer les timeouts ou dégrader proprement face à la latence réseau et aux pannes. Enfin, parce que tout le monde travaille sur une pile technologique partagée et a une familiarité avec la même base de code, il est plus facile de déplacer les gens et les équipes pour travailler sur différentes fonctionnalités et pousser vers une priorisation plus globale des fonctionnalités. Mais avec la croissance récente de GitHub, plusieurs avantages des microservices deviennent plus attirants : créer des équipes « feature teams » avec une responsabilité claire, instaurer des frontières fonctionnelles via des contrats d'API bien définis, et permettre aux équipes de choisir la pile technologique

7. demandes de modification du code qui ont été acceptées puis intégrées après revue
8. espace de stockage et de gestion d'un projet de code (fichiers + historique Git)
9. déploiements regroupés et planifiés à intervalles réguliers, où plusieurs changements (de plusieurs équipes) montent ensemble en production via un même pipeline et une fenêtre de mise en prod commune

(« stack ») la plus adaptée tant que les contrats sont respectés. Des services plus petits rendent le code plus lisible, l'onboarding plus rapide, et le diagnostic/dépannage plus simple. Un développeur n'a plus besoin de comprendre tout l'intérieur d'un gros monolithe pour être productif. Et surtout, chaque service peut être scalé séparément selon ses besoins. Avant d'engager cette transition, nous avons passé du temps sur le « pourquoi » et sur nos objectifs. C'est un changement culturel important qui demande beaucoup d'efforts. Il faut être intentionnel et clarifier les problèmes que l'on cherche réellement à résoudre. À GitHub, notre motivation principale est d'activer une grande partie des développeurs arrivés récemment (plus de la moitié de nos devs ont rejoint GitHub dans les 18 derniers mois) pour qu'ils puissent être productifs en dehors du monolithe. Notre objectif est donc la mise en capacité, pas le remplacement. En conséquence, nous acceptons que GitHub sera hybride (monolithe + microservices) pendant longtemps ; il est donc essentiel de continuer à maintenir et améliorer le monolithe.

Une bonne architecture commence par la modularité. La première étape pour découper un monolithe consiste à penser la séparation du code et des données par fonctionnalités. On peut faire cela dans le monolithe avant de séparer physiquement en microservices. C'est une bonne pratique pour rendre le codebase plus maîtrisable. Nous recommandons de commencer par les données et de surveiller de près les modes d'accès. L'objectif est que chaque service possède ses données et contrôle l'accès à celles-ci, et que l'accès aux données se fasse uniquement via des API clairement définies. Nous avons observé de nombreux cas où des équipes extraient la logique applicative, mais continuent à dépendre d'appels vers une base de données partagée dans le monolithe. Cela mène à un monolithe distribué : on cumule les complexités des microservices sans obtenir les bénéfices (comme la capacité à déployer indépendamment un sous-ensemble de fonctionnalités). Réussir la séparation des données est une pierre angulaire de la migration. Voici comment nous l'abordons chez GitHub : En premier lieu nous identifions des frontières fonctionnelles dans les schémas de base de données existants, puis nous regroupons les tables par frontières (par exemple : tout ce qui concerne les repositories, les utilisateurs, les projets). Ces groupes sont appelés des schema domains et sont décrits dans un fichier de définition YAML qui devient une source de vérité. Ce fichier doit être mis à jour à chaque ajout/suppression de tables. Un test linter [10] aide à rappeler aux développeurs cette exigence de mise à jour. Ensuite nous identifions un champ commun qui relie les informations d'un groupe fonctionnel. Par exemple, pour le domaine « repository » (issues, pull requests, review comments, etc.), la clé est le repo_id. Le fait de structurer la base par domaines fonctionnels prépare la séparation future des données sur différents serveurs et clusters. Avant de séparer, nous

10. test automatisé qui exécute un outil de vérification statique pour détecter des écarts à des règles

devons corriger les requêtes existantes qui traversent des frontières de domaines, pour éviter de casser le produit quand la séparation aura lieu. Nous avons donc implémenté un query watcher dans le monolithe pour détecter et alerter lorsqu'une requête traverse plusieurs domaines. Nous réécrivons alors ces requêtes en plusieurs requêtes respectant les frontières, et nous effectuons les jointures nécessaires au niveau de la couche applicative. Enfin, une fois les groupes fonctionnels isolés, nous pouvons aller plus loin en répartissant les données en plusieurs partitions correspondant à des groupes de clients, de sorte que chaque groupe soit servi par un sous-ensemble de bases et de serveurs. À l'échelle GitHub, un domaine fonctionnel peut devenir très grand ; la partition key permet de définir des plages (par exemple répartir les utilisateurs sur différents systèmes de stockage selon des plages numériques). D'autres groupements peuvent être plus logiques selon les caractéristiques des données (régions, taille, etc.). La tenantisation [11] permet de limiter l'étendue des dégâts d'un incident : une panne de stockage n'affecte qu'un sous-ensemble de clients au lieu d'impacter tout le monde.

Après la séparation des données, nous devons préparer l'extraction de services depuis le monolithe. Un point essentiel : la direction des dépendances doit toujours aller de l'intérieur du monolithe vers l'extérieur, et pas l'inverse, sinon on retombe dans le monolithe distribué. Par conséquent quand on extrait, on commence par des services cœur, puis on progresse vers des fonctionnalités. Ensuite, il faut identifier les « forces de gravité » qui maintiennent les développeurs dans le monolithe : des outils partagés, construits au fil du temps, qui rendent le développement dans le monolithe très pratique. Par exemple chez GitHub, les feature flags donnent aux développeurs un contrôle fin sur l'exposition d'une fonctionnalité (de « staff shipped » à bêta puis production). Pour déplacer cette gravité, il faut rendre ce type de ressources partagées disponible aussi en dehors du monolithe. Enfin il faut supprimer les anciens chemins de code une fois les nouveaux services en place ; il faut comprendre qui appelle le service et planifier le basculement de 100 % du trafic, sinon on reste coincé à maintenir deux implémentations ; chez GitHub, nous utilisons un outil open source appelé Scientist pour exécuter et comparer l'ancien et le nouveau chemin en parallèle pendant le déploiement progressif d'une nouvelle version ou fonctionnalité en production. Les premiers « core services » que nous avons choisi d'extraire sont l'authentification et l'autorisation. L'authentification est complexe car tout en dépend, avec beaucoup de logique partagée entre le site web et les opérations Git. Si github.com est down, l'accès aux systèmes Git peut l'être aussi, et les opérations pull/push ne fonctionnent plus, même en ligne de commande. Extraire ces fondations permet à des fonctions principales de rester dispo-

11. organisation d'un système "multi-clients" où plusieurs clients (tenants) partagent la même application, mais avec une séparation logique (et parfois physique) de leurs données et ressources

nibles sans dépendre du monolithe. L'autorisation a été plus directe : elle a déjà été réécrite comme un service externe. Le monolithe Rails communique avec lui via Twirp, un framework de communication inter-services proche de gRPC[12], ce qui respecte la dépendance « intérieur → extérieur ».

Passer aux microservices nécessite des changements opérationnels importants pour éviter une explosion des coûts d'exploitation. Nous recommandons d'actualiser le monitoring : passer de métriques d'appels « fonctionnels » à des métriques réseau et de contrats d'interface ; d'aller vers un CI/CD plus automatisé, fiable, et mutualisé entre services ; d'utiliser la conteneurisation pour supporter plusieurs langages et stacks ; de créer des templates de workflows réutilisables. Chez GitHub, nous avons créé une plateforme runtime self-service : « microservices in a box ». L'objectif est de réduire l'overhead opérationnel des équipes. La plateforme fournit des templates Kubernetes prêts à l'emploi, la configuration Ingress (load balancing), l'acheminement automatique des logs vers Splunk, et l'intégration à notre processus de déploiement interne. Cela simplifie fortement l'expérimentation et la mise en place de nouveaux services. À partir d'un certain point, toute nouvelle fonctionnalité devrait être créée en dehors du monolithe sous forme de microservice. Ensuite, il faut choisir quelques fonctionnalités simples à sortir : celles avec peu de dépendances et peu de logique partagée. Chez GitHub, nous avons commencé par la livraison des webhooks et le surlignage syntaxique. L'objectif est d'apprendre, de repérer les patterns, et d'identifier les manques avant d'attaquer des composants plus complexes. Il est utile d'observer ce qui change et se déploie souvent ensemble : cela révèle des parties plus fortement couplées et peut aider à définir des groupements naturels pour des livraisons indépendantes. Se baser sur la valeur produit/business[13] et ce dont une équipe peut raisonnablement être responsable de bout en bout peut également servir de guide. Attention toutefois : découper trop fin peut ajouter de la complexité et de l'overhead (plus de clés de déploiement, plus de charge d'astreinte, et des points de fragilité dus à la faible mutualisation de la connaissance).

Passer du monolithe aux microservices est un changement de paradigme majeur. Les processus de dev et le codebase deviennent très différents. Pour terminer nous allons rapidement couvrir les communications de service à service et la conception pour les pannes, qui sont tous deux des concepts importants dans le développement de microservices. Il existe deux façons dont les services communiquent les uns avec les autres, de manière synchrone et asynchrone. Avec les communications synchrones, le client envoie une requête

12. cadre de communication entre services (Remote Procedure Call) utilisant HTTP/2 et des interfaces typées (souvent décrites en Protocol Buffers), pour appeler des fonctions à distance de manière performante et standardisée

13. importance d'une fonctionnalité ou d'un composant pour le produit et l'activité (impact mesurable sur l'usage, le chiffre d'affaires, le risque, la conformité, la performance ou le time-to-market), justifiant des priorités et des investissements dédiés

et attend une réponse du serveur. Avec les communications asynchrones, le client envoie un message sans attendre de réponse, et chaque message peut être traité par plusieurs récepteurs. Chez GitHub, nous utilisons Twirp pour les communications synchrones entre le monolithe et des services cœur externes (comme l'autorisation). Mais lorsque de plus en plus de services sortent du monolithe, le synchrone devient rapidement inefficace et crée du couplage. Une meilleure approche est de bâtir un pipeline d'événements partagée pour broker des messages entre producteurs et consommateurs (c'est l'architecture utilisée à SendGrid). Parce que les services ne sont plus hébergés sur un seul serveur, il est important de tenir compte de la latence et des scénarios de défaillance lors de la communication sur le réseau. Une logique de réessai simple avec une fréquence de réessai clairement définie et un nombre maximum de réessais peut être suffisant pour gérer la plupart des problèmes de réseau temporaires. Un circuit breaker peut également être ajouté pour se protéger quand un service est en échec (après un certain nombre de tentatives infructueuses, le disjoncteur s'ouvrira et n'autorisera pas d'autres requêtes à passer jusqu'à ce que le service soit récupéré). Définissez une temporisation pour que votre service n'attende pas indéfiniment qu'un service externe réponde. Essayez de dégrader de manière contrôlée (messages conviviaux, retour au dernier état connu dans le cache...). Il faut toujours garder en tête l'expérience utilisateur et ce qui fait sens business.

1.2.2 Pistes de lecture

Objectif de lecture : être capable d'expliquer pourquoi GitHub bouge, comment ils s'y prennent (méthode), et ce qu'ils payent en échange.

Déclencheurs de la migration Identifiez les déclencheurs de la migration (en quoi la croissance rend le monolithe moins efficace ?).

Forces du monolithe Quelles sont les forces du monolithe reconnues explicitement ?

Bénéfices attendus des microservices Quels sont les bénéfices espérés des microservices ? En particulier : (a) équipes « feature teams » avec responsabilité claire/ownership, (b) frontières fonctionnelles via des contrats d'API bien définis. Pourquoi est-ce utile ?

Mise en capacité hybride ou remplacement Quel est l'objectif affiché : remplacement du monolithe ou mise en capacité avec un système hybride ? Quelles conséquences ?

Séparation des données Quelles sont les étapes de la séparation des données ?

Extraction des services Quels sont les conseils pour extraire les services ?

1.3 Questions ouvertes

Indicateurs de dépassement de capacité de l'équipe Quelles métriques "prédictives" indiquent qu'une architecture distribuée est en train de dépasser la capacité de l'équipe (point de bascule) avant l'arrêt du delivery ? Proposez 5 signaux concrets (pensez technique (ops), organisation, et qualité).

Tests Comment obtenir une confiance élevée sur des intégrations externes sans dépendre du réseau, de credentials de partenaires, et de comportements non déterministes ? Quelles combinaisons mettriez-vous en place ?

Indicateurs de réussite de la transition Une motivation pour sortir d'un monolithe peut être de permettre à de nouveaux développeurs de livrer et d'opérer des fonctionnalités plus facilement. Quels indicateurs montreraient que cette « mise en capacité » est atteinte ?

Respect de la séparation des données Pour préparer la séparation des données, certaines organisations maintiennent une cartographie explicite des « domaines de données » (quelles tables appartiennent à quel domaine) et automatisent sa mise à jour via des contrôles en intégration. Selon vous, ce type de mécanisme est-il suffisant pour éviter les contournements ? Pourquoi ?

Jointures au niveau applicatif Quand on interdit les requêtes SQL qui font une jointure entre deux domaines de données différents, on peut être amené à remplacer une grosse requête unique par plusieurs petites requêtes, puis « recoller » les résultats dans le code applicatif. À votre avis, quelles conséquences cela peut avoir ?

Sortir authentification et autorisation Dans une application, les fonctions d'authentification (vérifier l'identité) et d'autorisation (vérifier les droits) sont centrales : quasiment tout le système les utilise. Imaginons qu'on décide de les sortir du monolithe pour en faire des services séparés. Quels nouveaux risques cela peut-il créer pour le système et pour les utilisateurs ? Quelles contre-mesures techniques et organisationnelles peut-on mettre en place pour limiter ces risques ?

Que basculer en asynchrone Dans un système microservices, quelles interactions doivent rester synchrones (appel + réponse immédiate) et lesquelles doivent passer en asynchrone (messages/événements) ? Quels signaux vous feraient décider de basculer davantage vers une architecture événementielle ?

Template de création de services Une entreprise crée une plateforme interne qui permet à n'importe quelle équipe de lancer un microservice « clé en main » : modèles prêts, déploiement automatisé, configuration réseau, logs, monitoring... bref, un microservice en quelques clics. Quels problèmes secondaires cette facilité et cette standardisation peuvent-elles créer, même si l'intention est bonne ?

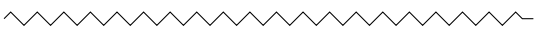
Changements d'organisation Quels changements d'organisation (responsabilités, astreinte, gouvernance) deviennent nécessaires quand on passe à des services ?

2 Sortir du mainframe

2.1 La solution de *Software Defined Mainframe* de LzLabs

2.1.1 Document

Le document proposé est une synthèse d'informations issues du site web de LzLabs et du white paper en collaboration avec CIO Magazine « Mainframe modernisation, An increasingly appealing business case and real-life case studies » paru en 2024.



Dans un monde où les cycles d'innovation raccourcissent et où des acteurs plus agiles viennent challenger des positions établies, votre portefeuille applicatif mainframe peut être à la fois votre actif le plus précieux et votre principal frein. Précieux, parce qu'il porte des décennies de savoir-faire métier, de règles, d'exceptions et de processus critiques, souvent éprouvés en production. Frein, parce que faire évoluer ces applications, les intégrer à des services modernes, ou simplement accélérer la livraison de nouvelles fonctionnalités devient de plus en plus coûteux, long et risqué. À cela s'ajoutent des pressions très concrètes : des coûts d'exploitation et de licences qui pèsent sur les budgets, des fenêtres de changement limitées parce que l'activité ne peut pas s'arrêter, et une raréfaction des compétences capables d'intervenir rapidement et sereinement sur ces environnements. Dans ce contexte, moderniser n'est pas « tout réécrire » : moderniser, c'est reprendre le contrôle, réduire le risque, et rendre possible une évolution continue, avec des résultats visibles bien avant la fin d'un programme pluriannuel.

Chez LzLabs, notre objectif est de vous redonner cette maîtrise en construisant un chemin de modernisation progressif, prédictible et à faible risque, sans sacrifier ce qui fait la valeur de vos systèmes : leur logique métier, leur fiabilité, leurs exigences de sécurité, et la continuité des opérations. Le cœur de l'approche repose sur le LzLabs Software Defined Mainframe (SDM), conçu pour faciliter une transition « sans rupture » depuis votre mainframe vers une infrastructure moderne, qu'elle soit on-premise, cloud privé, cloud public, ou hybride. Concrètement, vous décidez du rythme et de la séquence : quelles applications migrer en priorité, quelles données déplacer en premier, quels systèmes garder provisoirement en place, et quels composants moderniser plus tard. Cette capacité à prioriser et à séquencer est essentielle, parce qu'elle vous permet d'aligner la modernisation sur vos enjeux métier : commencer par ce qui coûte le plus, ce qui bloque le plus l'innovation, ce qui nécessite le plus d'élasticité, ou ce qui doit s'intégrer rapidement avec de nouveaux canaux digitaux.

La difficulté majeure des modernisations traditionnelles vient souvent de l'interconnexion : une application n'est jamais « seule », elle parle à d'autres

programmes, d'autres bases, d'autres fichiers, d'autres outils. Dès que vous modifiez un composant, l'effet domino commence, et ce qui paraissait simple devient une refonte globale, parfois impossible faute de documentation complète, faute de code source disponible, ou faute des experts historiques. C'est précisément là que le SDM change la dynamique : il vous permet d'avancer par étapes sans imposer une refonte générale dès le départ, en conservant l'interopérabilité entre ce qui a déjà été transféré vers une infrastructure moderne et ce qui reste encore sur l'environnement historique. Autrement dit, vous pouvez « libérer » progressivement des parties de votre patrimoine applicatif, sans casser le reste, et sans immobiliser votre organisation dans un chantier tout-ou-rien.

Cette approche progressive se décline en scénarios très simples à comprendre, y compris pour des publics non spécialistes. Vous pouvez commencer par la donnée : déplacer une base de données vers le SDM et la rendre accessible en temps réel à vos outils de reporting et de business intelligence. C'est souvent un premier pas très efficace, parce qu'il crée rapidement de la valeur : vos équipes peuvent exploiter des informations auparavant difficiles d'accès, accélérer la prise de décision, et connecter la donnée legacy à des usages modernes (tableaux de bord, analyses, machine learning) sans attendre une migration applicative complète. Vous pouvez ensuite passer aux applications, une par une : transférer une application legacy vers le SDM, la laisser fonctionner de manière stable, puis la connecter au « monde moderne » via des API et des outils d'intégration, tout en maintenant ses échanges avec les composants qui n'ont pas encore bougé. Enfin, vous élargissez progressivement : vous ajoutez d'autres applications, vous optimisez, vous améliorez les flux, vous modernisez là où cela apporte un bénéfice clair, et vous installez une dynamique d'amélioration continue plutôt qu'un programme ponctuel qui épuise l'organisation.

Un avantage déterminant de cette trajectoire est qu'elle protège votre investissement dans la propriété intellectuelle existante. Vos applications mainframe ne sont pas seulement du code : elles encapsulent des règles métier et des comportements parfois très spécifiques, qui ont été validés par l'usage, par les audits, par les incidents résolus au fil des années, et par l'expérience accumulée. Repartir de zéro ou réécrire massivement, c'est prendre le risque de perdre cette richesse, d'introduire des écarts fonctionnels difficiles à détecter, et de s'engager dans des efforts de test interminables. Le SDM vise à éviter ce piège : vous migrez vos workloads vers des environnements ouverts sans exiger une refonte immédiate, et vous choisissez ensuite, avec calme et méthode, ce que vous souhaitez optimiser, refactorer, découper, remplacer, ou retirer. Vous réduisez ainsi le risque de paralysie projet, de dépassements budgétaires et de « perte de temps » liée à des chantiers trop vastes, tout en gardant la liberté de moderniser réellement, progressivement, et au bon endroit.

La modernisation n'est pas seulement un sujet de plateforme ; c'est aussi

un sujet de performance, de sécurité, de fiabilité et d'exploitation. Une transition vers une infrastructure moderne n'a de valeur que si elle respecte les exigences de production et de conformité que votre mainframe a historiquement satisfaites. Le SDM est pensé pour permettre cette continuité : vous migrez sans compromettre la robustesse opérationnelle, vous maintenez un contrôle d'accès et une gouvernance compatibles avec votre cadre de conformité, et vous conservez la capacité à intégrer les politiques de sécurité legacy avec les mécanismes modernes, ce qui est indispensable dans un SI hybride. L'objectif n'est pas de choisir entre « sécurité mainframe » et « agilité cloud », mais d'obtenir les deux, en contrôlant les transitions et en sécurisant les interfaces.

Pour accélérer l'innovation, il ne suffit pas de déplacer des applications : il faut aussi moderniser la façon de livrer. C'est pourquoi l'approche SDM s'inscrit naturellement dans une trajectoire où les pratiques DevOps, l'automatisation, et des cycles de déploiement plus fluides deviennent accessibles sur des workloads historiquement contraints par des processus lourds. Dans un environnement ouvert, vous pouvez industrialiser les chaînes de build/test/deploy, réduire les délais de mise en production, et surtout réduire le coût marginal du changement. Cette transformation des pratiques est un accélérateur majeur : elle permet de livrer plus souvent, plus sûrement, et avec un meilleur retour utilisateur, ce qui est exactement ce que les organisations recherchent lorsqu'elles parlent d'agilité.

Sur le plan produit, le SDM s'appuie sur des briques qui correspondent aux grands usages mainframe, afin de parler un langage simple et concret. De nombreuses organisations s'appuient sur des bases relationnelles critiques (souvent DB2) ; avec LzRelational™, l'objectif est de permettre une trajectoire vers un SGBD open source comme PostgreSQL tout en conservant la structure relationnelle attendue par les applications. Beaucoup d'organisations dépendent encore de traitements batch, souvent nocturnes, qui restent essentiels pour la production, la consolidation, la facturation, ou les clôtures ; LzBatch™ adresse cette réalité en offrant un cadre de prise en charge adapté à ces traitements. Et dans la plupart des entreprises, les systèmes transactionnels en ligne (OLTP) sont au cœur des opérations quotidiennes ; LzOnline™ s'inscrit dans cette dimension en tenant compte des choix modernes d'infrastructure x86 et des modèles de déploiement cloud. L'intérêt de ce découpage est simple : vous ne partez pas sur une promesse abstraite, vous avancez sur des composants concrets, alignés sur vos besoins réels, et vous bâtissez progressivement un socle modernisé.

En termes de bénéfices, la modernisation doit produire un retour sur investissement visible et crédible. Un premier axe est la réduction des coûts mainframe, notamment en diminuant la dépendance à du matériel et à des modèles de licences legacy, ce qui vous permet de réallouer des ressources vers des initiatives de croissance et d'innovation. Un deuxième axe est l'élargissement des capacités : en environnement cloud ou hybride, vous gagnez

en flexibilité, vous pouvez dimensionner plus efficacement, et vous évitez que des contraintes d'infrastructure deviennent des limites de business. Un troisième axe est la simplification du chemin : éviter une refonte complète, éviter des changements de formats de données, et éviter de devoir « tout casser » pour « tout refaire » réduit la friction et le risque, et permet d'avancer tout en maintenant la production. Un quatrième axe, souvent sous-estimé, est la valorisation de la donnée : rendre la donnée legacy accessible aux outils modernes d'analyse et d'IA accélère la création de valeur, parce que vous pouvez enfin exploiter pleinement l'historique et le temps réel, sans attendre que toute l'application soit transformée. Un cinquième axe est l'évolution des compétences : basculer vers des environnements ouverts facilite la formation, le recrutement, et la mobilité interne, et vous aide à réduire le risque lié à la pénurie de profils spécialisés.

La clé, dans la réalité, est d'éviter les trajectoires qui enferment. Une modernisation réussie est rarement linéaire : vos priorités peuvent évoluer, vos contraintes peuvent changer, des acquisitions peuvent modifier le périmètre, un cadre réglementaire peut se durcir, une plateforme cloud peut s'imposer, une autre peut être privilégiée, et de nouvelles technologies peuvent rendre certaines options plus attractives. C'est pourquoi l'approche incrémentale est si importante : elle vous permet d'obtenir des gains dès les premières étapes, de réduire progressivement la dépendance aux éléments les plus coûteux et les plus rigides, et de garder une liberté d'action. Vous ne vous engagez pas dans un pari unique où toute la valeur est à la fin ; vous construisez un chemin où chaque étape est utile, contrôlée, et intégrée dans une trajectoire globale.

Enfin, moderniser avec succès suppose un accompagnement et une exécution rigoureuse. La technologie seule ne suffit pas : il faut une démarche de transformation « consultative », capable de sécuriser les étapes, de faciliter le transfert de connaissances, de soutenir les équipes pendant la transition, et d'installer des pratiques qui rendent la modernisation durable. C'est cette combinaison qui permet de moderniser avec confiance et continuité : une plateforme conçue pour l'hybride et l'interopérabilité, une migration incrémentale pilotée par vos priorités, une valorisation rapide de la donnée et des applications, une exploitation alignée sur les exigences de sécurité et de conformité, et des pratiques modernes pour accélérer la livraison. Le résultat recherché est clair : retrouver de l'agilité sans perdre la fiabilité, réduire les coûts sans perdre la valeur, et transformer un patrimoine legacy complexe en un actif modernisé, connecté et prêt pour l'innovation.

2.1.2 Contentieux IBM vs LzLabs

IBM reprochait à LzLabs d'avoir utilisé l'accès d'une filiale UK (Winsopia, considérée par IBM comme une « coquille » constituée pour acquérir auprès de lui une licence mainframe) à un mainframe/licences IBM pour

désassembler / reverse-engineerer des éléments logiciels IBM à plusieurs niveaux de sa pile mainframe (OS, middlewares et compilateurs), et s'en servir pour construire la plateforme Software Defined Mainframe (SDM).

Défense de LzLabs : la Software Directive de 2009 garantit, au Royaume-Uni, des droits à l'observation, à l'étude et au test de programmes informatiques.

Décision de justice (High Court, Technology and Construction Court, 10 mars 2025) : la juge a conclu que Winsopia avait violé les termes du contrat/licence IBM, et que LzLabs avaient provoqué ces violations.

2.1.3 Pistes de lecture

Objectif de lecture : être capable d'expliquer pourquoi le mainframe freine l'agilité, comment LzLabs propose une trajectoire de modernisation incrémentale via le *Software Defined Mainframe* (SDM), et quels bénéfices/compromis cette approche implique (risque, coûts, dépendances, organisation).

Motivations de la modernisation Quelles motivations sont mises en avant pour justifier la modernisation d'une architecture basée sur un mainframe ? Lesquelles vous semblent les plus déterminantes si votre contrainte n°1 est la continuité de service et le risque ? Et si c'est plutôt la compétitivité / time-to-market ?

Une transition douce Quels éléments font du SDM un mécanisme de transition qui transforme une modernisation mainframe en série de petites migrations interopérables, plutôt qu'en une refonte totale à haut risque ?

Séquence recommandée Quelle est la séquence recommandée et pourquoi ?

Protection de l'essentiel Qu'est-ce qui est présenté comme « protégé » et quelles implications cela a sur le coût de la transformation ?

2.2 La solution de transformation par Google

2.2.1 Document

Le document proposé reprend le contenu d'une vidéo de Google Cloud Next OnAir de Travis Webb publiée en 2021, enrichi d'informations provenant d'un article de Travis Webb sur developpers.googleblog.com de février 2021 (*Mainframe modernization antipatterns*) et mis à jour avec la brochure « *Google Cloud Mainframe Modernization* » de janvier 2026.

Je suis Travis Webb, architecte solutions entreprise chez Google. Merci de me rejoindre sur Google Cloud Next OnAir. J'accompagne nos clients pour résoudre leurs problèmes IT les plus importants et les plus complexes, et aujourd'hui je vais parler de la modernisation du mainframe, une offre de Google pour aider les organisations à faire évoluer leurs applications historiques vers des plateformes plus agiles. C'est une présentation d'introduction : je vais commencer par une vue d'ensemble de l'écosystème mainframe, expliquer la place du mainframe dans le monde actuel, puis passer en revue des approches courantes, souvent avec de mauvais résultats, que les entreprises ont tentées pour en sortir, avant de présenter la façon dont Google adresse ce sujet.

D'abord, un aperçu rapide du marché : le mainframe représente un marché important qui inclut non seulement le coût des machines, mais aussi les frais d'usage, les licences et l'ensemble des logiciels spécifiques. La grande majorité des grandes entreprises s'appuient encore sur des mainframes pour exécuter des charges métiers critiques, et beaucoup de ces applications se sont construites sur des décennies. Cet usage dépasse les tout premiers acteurs : il existe encore des milliers de systèmes en production, et chacun représente une opportunité de transformation stratégique, parce qu'au moment où les charges « standard » migrent rapidement vers le cloud, les charges mainframe ont nettement moins bougé. Pourquoi ? Il faut se souvenir que pendant une grande partie de l'histoire de l'informatique, l'idée d'exécuter des processus métiers vraiment critiques sur des serveurs « banals » ou des PC aurait été jugée fantaisiste ; le mainframe était la plateforme « sérieuse ». Et le mainframe résiste encore aujourd'hui à la concurrence pour plusieurs raisons structurelles. Premièrement, le mainframe IBM dispose d'une chaîne ininterrompue de rétrocompatibilité remontant aux années 1960 : cette durabilité a permis d'accumuler une base installée massive et des revenus très stables pour les fournisseurs, et même si le matériel de l'époque a disparu, une partie du code écrit il y a des décennies est toujours en service. Deuxièmement, dans les décisions d'achat IT, les dirigeants scrutent fortement les coûts d'entrée... mais sous-estiment souvent les coûts de sortie : quitter une plateforme mainframe est historiquement coûteux, risqué, et pour de nombreuses entreprises la migration n'a longtemps pas semblé économiquement viable. Le caractère propriétaire de l'écosystème renforce ces coûts de sortie : de nombreuses applications dépendent de technologies et d'outils indisponibles hors mainframe. Cette combinaison laisse peu d'alternatives simples : si les clients ne peuvent pas facilement partir, pourquoi un fournisseur investirait-il agressivement dans l'innovation ? Même si le mainframe s'est beaucoup amélioré au XXe siècle, ses avantages relatifs se sont érodés, et beaucoup de clients se retrouvent aujourd'hui avec un double handicap : technologique (accès plus lent aux innovations modernes) et économique (coûts élevés). On n'a plus « besoin » d'un mainframe pour la plupart des traitements intensifs, mais quand on en a un, on reste souvent coincé à le financer. Ces désavantages sont

largement reconnus, mais les options concrètes de sortie ont longtemps manqué, et en parallèle l'écosystème vieillit : moins de nouveaux développeurs y entrent, et les fournisseurs ont tendance à réduire l'effort d'innovation. Autour de l'an 2000, les entreprises ont accéléré l'usage de serveurs standards pour des charges métiers, et Google a été à l'avant-garde des innovations d'infrastructure de centres de données qui rendent possible, à grande échelle, ce que le mainframe faisait dans un autre paradigme. Le cloud public démocratise l'innovation IT à un rythme très rapide, parce que des entreprises veulent exploiter des technologies que leurs solutions historiques ne livrent pas assez vite. L'IT « legacy » est chère, et le mainframe est souvent en tête de liste ; et si un coût élevé pouvait se justifier par une forte agilité, on constate souvent l'inverse : un écosystème statique, de plus en plus coûteux, qui rend l'innovation plus difficile. Les attentes clients augmentent, le rythme de l'innovation augmente, mais beaucoup d'organisations font face à une pénurie chronique de compétences mainframe, ce qui met en risque des décennies d'investissement : beaucoup disent clairement « nous voulons en sortir ».

Le problème, c'est que migrer ou moderniser des workloads mainframe reste complexe et difficile, même dans de bonnes conditions ; et si on évite certains « anti-patterns », on augmente fortement les chances de succès, que la cible soit un cloud public, des machines virtuelles on-premise, ou autre. On voit revenir trois approches « classiques » que les entreprises tentent seules. La première est la réécriture massive (« big bang rewrite ») : réécrire et ré-architecturer manuellement tout le patrimoine dans un langage moderne en espérant repartir sur des bases propres. L'intention paraît rationnelle (innover, moderniser, transférer la connaissance des experts vers de nouveaux ingénieurs), mais des pièges se cachent : les organisations résistent mal à la tentation de « re-faire mieux » et de changer la logique métier en cours de route, c'est le fameux second system effect, et surtout, obtenir un comportement strictement équivalent à l'existant est un effort de longue traîne systématiquement sous-estimé. Très vite, le projet se retrouve à un point de décision : soit on s'acharne à reproduire exactement l'historique (plus long que prévu), soit on change la logique métier (ce qui impose de modifier des processus et des systèmes aval qui dépendaient parfois de comportements « idiosyncratiques »). Et un facteur aggravant majeur apparaît dans la vraie vie : pendant la réécriture, le système de production continue d'évoluer ; on tombe alors dans la double maintenance (il faut implémenter et tester les évolutions sur l'ancien et sur le nouveau), ce qui décale le planning à répétition et fait exploser le risque et le budget ; au final, même lorsque l'entreprise a la ténacité d'aller au bout, le ROI devient souvent peu convaincant au regard du coût brut et des retards.

La deuxième approche est l'émulation / lift-and-shift : déplacer « tel quel » en faisant tourner l'environnement mainframe sur du matériel standard via une couche d'émulation (en pratique, cela implique d'émuler sur

x86 et de recompiler avec les outils fournis par le vendeur d'émulation). Sur le papier, cela semble être la voie la plus rapide vers le cloud, notamment lors d'un renouvellement matériel, et l'on peut penser y retrouver une forme d'élasticité. Mais c'est là que le piège se referme : à l'échelle d'une machine, on n'atteint pas la performance « cloud-scale », mais « à poids égal », le mainframe reste un matériel excellent ; or la plupart des problèmes viennent de la couche écosystème et de la structure de coûts, pas du silicium. Avec l'émulation, on perd souvent le seul atout net, le matériel mainframe performant, tout en conservant ce qui retient (dépendances propriétaires, rigidité, coûts), en y ajoutant une dépendance supplémentaire à la couche d'émulation. Le résultat est que l'on encourt beaucoup des risques d'une migration, pour peu de bénéfices réels en agilité et en innovation, et l'on peut même se retrouver avec un nouveau verrouillage (licences et support de l'émulateur), pour des gains économiques souvent plus faibles qu'espéré une fois additionnés les coûts cloud et ceux de l'émulation.

La troisième approche est la modernisation sur place (« in-place modernization ») : améliorer la qualité, la maintenabilité et les pratiques de développement tout en restant sur mainframe, parfois en adoptant des outils plus modernes et même des briques d'orchestration pour une expérience partiellement « cloud-like ». Cela peut sembler moins risqué (moins de changement de plateforme, pas de bascule majeure), mais cela ne résout ni les problèmes de structure de coûts ni ceux de rareté des talents, et garde souvent l'organisation enfermée dans un écosystème dont les tendances sont défavorables (compétences qui se raréfient, plateforme qui s'isole, consolidation des fournisseurs).

Chez Google, nous avons une approche différente : au lieu de demander aux clients de « tout refaire » ou de déplacer sans moderniser, l'objectif est d'industrialiser la trajectoire pour réduire le risque, limiter la durée et maximiser les bénéfices cloud. Historiquement, nous avons présenté cela sous la forme d'un pipeline clair (souvent résumé comme G4) : analyser le code et les dépendances pour comprendre la logique métier, convertir progressivement le code legacy vers une base moderne fondée sur des standards ouverts afin de l'exécuter nativement dans le cloud, mesurer et piloter la complexité et la performance, puis valider / vérifier / tester rigoureusement pour garantir l'équivalence fonctionnelle et réduire le risque, en migrant par vagues plutôt qu'en une seule fois, et en parallélisant l'effort comme une « factory » de migration ; un point important est de viser une sortie propre sans verrouillage contractuel sur le code produit.

L'offre s'est ensuite structurée et modernisée : l'idée G4 a été industrialisée en produits et enrichie par Gemini et l'IA générative. Concrètement, la phase « analyze » se retrouve dans Mainframe Assessment avec le Mainframe Assessment Tool (MAT), qui aide à analyser le patrimoine, expliquer le code, résumer la logique applicative, produire de la documentation et des spécifications, identifier les dépendances et même générer des cas de test.

La transformation de code se décline en Application Code Transformation avec deux axes : Mainframe Code Rewrite, intégré à Gemini Code Assist pour générer du code moderne fonctionnellement équivalent à partir du code mainframe, et Mainframe Refactor, qui automatise refactoring et conversion pour exécuter des applications modernisées sur des environnements cloud comme Google Kubernetes Engine (GKE) ou Cloud Run.

Un volet complémentaire existe aussi, Mainframe Augmentation, via un Mainframe Connector : l'idée est de désenclaver des données mainframe jusque-là « siloées » pour les exploiter en analytique et en IA / ML dans Google Cloud tout en conservant temporairement les applications, ce qui peut réduire la consommation (par exemple en MIPS) et accélérer l'agilité, sans exiger une migration totale immédiate. Enfin, pour sécuriser la transformation, l'offre met en avant un mécanisme de test et de dé-risquage explicite, Dual Run, qui rejoue des événements réels issus de la production mainframe sur l'application modernisée dans le cloud et compare les résultats entre les deux systèmes afin de certifier la correction de la logique métier.

Pour résumer : le mainframe est resté en retrait du mouvement cloud pour des raisons historiques et structurelles, les trois approches « instinctives » échouent souvent pour des raisons prévisibles, et une trajectoire plus fiable consiste à passer par un pipeline industrialisé, puis par une offre produit moderne, renforcée par Gemini, qui combine assessment, transformation progressive, accès aux innovations cloud et validation robuste, tout en gardant le cap sur la réduction des coûts et des risques et l'augmentation de l'agilité et de l'innovation.

2.2.2 Pistes de lecture

Objectif de lecture : être capable d'expliquer pourquoi les charges mainframe sont difficiles à déplacer, pourquoi certaines stratégies échouent (anti-patterns), comment Google structure une trajectoire de migration « industrialisée », et quels bénéfices/compromis cette approche implique (coûts, risques, dépendances, effort de validation).

Difficultés à sortir du mainframe Quels sont, selon le document, les facteurs qui rendent la sortie du mainframe difficile ? Parmi eux, lesquels sont techniques, économiques, organisationnels et humains ?

Anti-patterns de migration Quels sont les trois anti-patterns présentés et, pour chacun, quel est le mécanisme principal qui conduit à l'échec ou à une valeur limitée ?

Priorité de la solution Google Quel problème la solution Google cherche-t-elle à résoudre en priorité : coût, risque, vitesse, pénurie de compétences,

accès à l'innovation cloud, ou un mix ? Quels indices dans le texte permettent de trancher ?

Pipeline G4 Comment résumer le pipeline « G4 » en 4–5 étapes et à quel moment le risque est-il censé être « absorbé » ?

Promesse technique vs méthode Quels éléments relèvent d'une promesse technique (ce que la solution « fait ») et quels éléments relèvent d'une promesse de gouvernance/méthode (comment la transformation est conduite et « dé-risquée ») ?

2.3 Questions ouvertes

Objectif : ouvrir la discussion au-delà des promesses produits, en identifiant les décisions « terrain » qui conditionnent la réussite d'une sortie de mainframe (organisation, risques, preuves, exploitation).

— Comment transformer les équipes (dev/ops) sans casser la production ?

Angles : modèle d'ownership (produit/plateforme), montée en compétences par lots réels, binômes legacy/open-systems, runbooks et capitalisation, trajectoire DevSecOps/SRE.

— En période hybride, qui est responsable de quoi (conformité, run, incidents) ?

Angles : matrice RACI fournisseur / intégrateur / client, responsabilités « de la plateforme » vs « dans la plateforme », engagements SLA/SLO, limites contractuelles.

— Quelles preuves « auditables » faut-il produire, et comment les industrialiser ?

Angles : logs d'accès/IAM, traçabilité des changements (CI/CD), preuves de tests, scans vulnérabilités, configuration baseline, rapports de restauration/DR, rétention et intégrité des journaux.

— Comment pilote-t-on la trajectoire et qui tranche en cas de conflit délai vs conformité ?

Angles : comités (steering risque vs architecture), critères de bascule (gates), registre d'exceptions et mesures compensatoires, règle d'arbitrage lorsque le risque réglementaire/sécurité est en jeu.

— Comment prouver l'équivalence fonctionnelle pendant et après migration ?

Angles : stratégie de tests (régression, intégration, batch), données de test, tolérances acceptables, comparaison en conditions réelles (replay/dual run), gestion des comportements historiques implicites.

— Quelle stratégie de résilience adoptez-vous (RPO/RTO) et comment la vérifiez-vous ?

Angles : classification par criticité (tiers), architecture multi-zone, sauvegardes chiffrées et testées, exercices réguliers de reprise, rapports et postmortems.

— Comment concevoir l'observabilité en hybride pour diagnostiquer vite ?

Angles : logs structurés + métriques + traces, propagation d'identifiants de transaction entre legacy et modernisé, SLO par parcours critique, dashboards/runbooks standard.

— Quels coûts cachés apparaissent dans l'hybride (TCO) ?

Angles : double exploitation, réseau/egress, licences/outils, coûts d'audit, coûts de test et de données, coûts de support/astreinte, coûts de « cohabitation prolongée ».

— Quelles dépendances résiduelles risquent de bloquer la sortie complète ?

Angles : interfaces, ordonnancement batch, produits tiers, formats/protocoles, dépendance à des compétences rares, dette « fonctionnelle » (cas limites), dépendances organisationnelles.

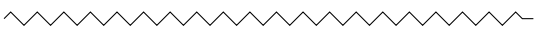
— Risque de verrouillage : comment préserver la réversibilité ?

Angles : standards, portabilité des artefacts, propriété des pipelines/outils, stratégie de sortie (exit plan) dès le départ, clauses contractuelles, tests de réversibilité (au moins sur un périmètre).

3 Choisir entre deux architectures

3.1 Document

Le document proposé est une adaptation de l'article de recherche *2-tier vs. 3-tier Architectures for Data Processing Software* présenté par Dmitriy Dorofeev et Sergey Shestakov en 2018 à l'ICAIT (3rd International Conference on Applications in Information Technology).



Introduction Les traitements analytiques (qui interrogent, croisent, agrègent et explorent de grands volumes de données pour produire des indicateurs, des tableaux de bord ou des visualisations) exigent souvent l'exécution d'un grand nombre de requêtes vers la base de données pour satisfaire une seule interaction utilisateur. C'est particulièrement vrai dans le cas de la visualisation analytique. La préparation d'un simple tableau de bord peut nécessiter de nombreuses requêtes et de multiples interactions avec les sources de données. Même si l'on exécute ces requêtes en parallèle, le problème n'est pas entièrement résolu, car il faut ensuite synchroniser, agréger et mettre en forme les résultats. Une architecture 3-tier permet bien de séparer la présentation, la logique applicative et la gestion des données, mais elle impose aussi de nombreux allers-retours entre le serveur applicatif et la base. Dans une application analytique, où la valeur principale du système réside moins dans des règles métier complexes que dans la capacité à interroger rapidement de gros volumes de données et à renvoyer des résultats adaptés à l'affichage, cette circulation entre couches peut devenir un coût important.

Cette problématique se retrouve dans les contextes où la croissance très rapide des volumes de données, provenant par exemple de l'Internet des objets ou des réseaux sociaux, déplace une partie de la valeur des systèmes depuis la seule logique métier vers la donnée elle-même, et oblige à repenser des architectures conçues pour des contextes moins massifs. Dans cette tendance plus large dite « data-centric », une partie significative du traitement est placée au plus près des données. Plusieurs évolutions techniques vont dans ce sens, par exemple le développement de solutions de calcul distribué, de traitements en fog computing, ou encore l'enrichissement des SGBD relationnels qui ont progressé à la fois sur leurs capacités de requêtage et sur leurs fonctions en mémoire ; les moteurs relationnels modernes offrent désormais des extensions procédurales de plus en plus riches autour de SQL, suffisamment puissantes pour implémenter directement dans la base toute la logique métier nécessaire. Cela ouvre de nouvelles possibilités pour les architectures *data-centric*, en particulier dans le cas de traitements intensifs en données comme les applications analytiques.

Ces considérations amènent à une question simple : est-il préférable de conserver un serveur applicatif séparé entre le client et la base de données,

ou bien peut-on servir directement les clients depuis la base, avec un serveur REST embarqué dans celle-ci ? C'est cette question que nous voulons explorer à partir de deux implémentations réelles d'une même plateforme analytique commerciale, Luxms BI.

Implémentation de référence de l'architecture 3-tier Notre implémentation 3-tier comprend une application cliente dans le navigateur, développée en HTML5 et JavaScript, un serveur applicatif développé en Java 8, et un serveur de base de données PostgreSQL 9.6. Devant le serveur applicatif, nous plaçons Nginx, qui joue le rôle de proxy HTTP et assure également des fonctions de sécurité et de répartition de charge.

Le flux de traitement d'une requête se déroule de la manière suivante. Le client web construit une requête HTTP au format JSON pour demander des données. Cette requête est transmise telle quelle par le proxy HTTP au serveur applicatif. Le serveur applicatif analyse alors la requête reçue et construit, à partir de celle-ci, une ou plusieurs requêtes SQL adaptées. La couche JDBC, intégrée au serveur applicatif, convertit ensuite ces requêtes SQL vers le protocole utilisé pour dialoguer avec la base de données. PostgreSQL exécute les requêtes, lit les données, puis les résultats remontent dans le sens inverse : la couche JDBC récupère les données, le serveur applicatif les transforme en JSON, le proxy relaie la réponse, et le client reçoit finalement les données demandées. Une seule interaction utilisateur ne correspond pas nécessairement à une seule requête SQL. Le serveur applicatif peut devoir lancer plusieurs requêtes, agréger leurs résultats, puis assembler une réponse cohérente pour le client.

Dans le cas d'applications analytiques, les échanges entre le serveur applicatif et la base de données apparaissent donc comme un point potentiellement critique de la chaîne de traitement.

Implémentation de référence de l'architecture 2-tier centrée sur les données Notre implémentation 2-tier *data-centric* repose sur le même client web, développé en HTML5 et JavaScript, mais la logique applicative n'est plus portée par un serveur Java séparé. Elle est implantée directement dans PostgreSQL 9.6, sous la forme d'un serveur applicatif *in-database* écrit en PL/pgSQL, hébergé dans le même moteur que le schéma de données. Devant cet ensemble, Nginx joue le rôle de proxy HTTP pour recevoir les requêtes et servir les fichiers statiques, tandis que HAProxy est utilisé pour la répartition de charge et la mise en file des requêtes entrantes vers le serveur applicatif en base.

Le flux de traitement d'une requête se déroule de la manière suivante. Le client web construit une requête HTTP au format JSON pour demander des données ; le format est identique à celui utilisé côté client dans l'architecture 3-tier. Le proxy HTTP transmet cette requête au serveur applicatif implanté

dans la base, en appelant une procédure stockée. Le serveur applicatif en base analyse la requête et l'oriente vers la procédure PL/pgSQL appropriée ; plusieurs requêtes SQL peuvent alors être exécutées par la base, selon la logique métier et les détails de la requête. Comme dans l'architecture 3-tier décrite dans la section précédente, une fois les requêtes terminées, les données repartent ensuite vers le client dans l'ordre inverse.

Les fonctionnalités de Nginx peuvent être étendues à l'aide du langage Lua ; nous avons donc mis en place une passerelle entre HTTP et les procédures stockées, ce qui a nécessité environ 200 lignes de code Lua.

Le serveur applicatif implanté dans la base est écrit en PL/pgSQL et prend en charge le routage HTTP, l'authentification, l'autorisation, les opérations CRUD génériques sur les tables, ainsi que des points d'entrée d'API spécifiques lorsque les opérations CRUD ne suffisent pas. Les requêtes SQL et les procédures PL/pgSQL ne peuvent pas s'exécuter en parallèle à l'intérieur d'une même session cliente, ce qui constitue un inconvénient important par rapport à l'implémentation 3-tier fondée sur Java. L'ensemble du code du serveur applicatif représente environ 6000 lignes de PL/pgSQL. Il s'agit d'une implémentation assez verbeuse de la logique applicative, car on ne dispose pas ici des modules tiers habituels de l'écosystème Java. La seule partie consacrée au routage des requêtes HTTP représente environ 800 lignes de code.

Comparaison des cycles de développement logiciel en architecture 2-tier et 3-tier Historiquement, le déplacement de popularité des architectures 2-tier vers les architectures 3-tier a été fortement influencé par la meilleure qualité des outils disponibles pour le cycle de développement logiciel. Il est donc important de réexaminer ce point et d'évaluer l'état actuel des outils que l'on peut mobiliser aujourd'hui. Dans cette section, nous présentons les avantages et les inconvénients des outils disponibles pour les différentes phases du cycle de développement logiciel (*Software Development Life Cycle*, ou SDLC) dans nos deux implémentations de référence.

Développement L'architecture 3-tier fondée sur Java dispose d'un IDE complet et riche en fonctionnalités, ainsi que d'outils de débogage et de profilage très aboutis.

L'architecture 2-tier centrée sur les données et fondée sur PL/pgSQL ne propose pas d'environnement de développement aussi complet ; le débogage reste toutefois possible avec des outils comme OmniDB ou pgAdmin, et le profilage peut être réalisé au moyen de l'extension PL Profiler ou de la vue `pg_stat_statements`.

Java est un langage compilé et impose de reconstruire l'application complète avant de pouvoir l'exécuter dans la JVM. PL/pgSQL permet au contraire de modifier le code fonction par fonction, ce qui est sensiblement plus rapide

qu'avec Java.

Tests Pour les tests, l'architecture 3-tier fondée sur Java dispose d'un large choix de frameworks et d'outils, adaptés à des besoins très variés. L'architecture 2-tier centrée sur les données et fondée sur PL/pgSQL suppose davantage que le développeur prenne lui-même en charge la mise en place de l'environnement de test, et seuls quelques frameworks sont disponibles.

Déploiement L'architecture 3-tier fondée sur Java offre un bon support pour le téléversement de fichiers JAR et le redémarrage de l'application ; dans certains cas, un rechargement à chaud est même pris en charge par les conteneurs de servlets Java.

L'architecture 2-tier centrée sur les données et fondée sur PL/pgSQL dispose, elle aussi, d'un support correct, grâce à des scripts transactionnels regroupant des instructions `CREATE OR REPLACE FUNCTION` (c'est-à-dire des instructions SQL qui permettent de redéfinir une fonction existante sans reconstruire ni redéployer toute l'application). Cela se rapproche d'un rechargement à chaud, mais suppose un certain outillage et une certaine discipline, car il faut gérer proprement l'ordre des modifications, les dépendances entre fonctions et la compatibilité avec le schéma de données.

Exploitation En exécution, l'architecture 3-tier classique nécessite un processus ou serveur supplémentaire, avec en plus des réglages spécifiques de la JVM à ajuster. L'architecture 2-tier centrée sur les données s'exécute directement dans le processus de la base de données et ne requiert donc pas de matériel supplémentaire.

Pour la supervision, l'architecture 3-tier classique peut s'appuyer sur un large choix d'outils compatibles JMX (*Java Management Extensions*, mécanisme standard de supervision et d'administration des applications Java et de la JVM), tandis que l'architecture 2-tier centrée sur les données peut être surveillée à l'aide des outils standards de la base.

Maintenance et support Dans une architecture 3-tier, le schéma de données et la logique applicative évoluent dans deux composants distincts. Toute modification importante du modèle de données peut donc obliger à faire évoluer en parallèle les requêtes SQL, les objets manipulés par le serveur applicatif, les transformations de données et parfois même les interfaces exposées au client. Il faut alors synchroniser plusieurs versions de plusieurs composants, ce qui rend les migrations plus délicates et plus risquées.

À l'inverse, dans une architecture 2-tier centrée sur les données, la logique applicative étant beaucoup plus proche du schéma, il est plus facile de faire évoluer ensemble structures de données et traitements, à condition d'imposer une discipline stricte sur les changements.

Performances Les mesures de performance ont été réalisées avec l'outil open source Locust. Ce générateur de charge tournait sur une machine virtuelle dédiée, tandis que les deux versions de Luxms BI, 2-tier et 3-tier, tournaient sur une autre machine virtuelle dédiée, et s'appuyaient toutes deux sur PostgreSQL 9.6.5. Afin d'évaluer différents aspects des implémentations côté serveur, plusieurs niveaux de charge ont été générés.

Charge légère Des charges légères ont été utilisées pour mesurer le débit maximal possible au moyen de requêtes aléatoires spécialement conçues. L'idée était de produire de très petites réponses, tout en obligeant malgré tout le serveur à exécuter une certaine activité SQL. La charge a été générée par 4000 utilisateurs Locust. Au total, 200000 requêtes légères ont été mesurées.

Dans cette configuration, la solution 2-tier a montré une latence trois fois plus faible et a traité environ deux fois plus de requêtes par seconde que la solution 3-tier. La distribution des temps de réponse sous charge légère montre que l'architecture 2-tier reste plus rapide sur l'ensemble des percentiles mesurés, et pas seulement en moyenne.

Par ailleurs, l'implémentation 3-tier soumise à une charge de 5000 utilisateurs Locust a provoqué un grand nombre de timeouts, alors que l'implémentation 2-tier continuait à bien se comporter.

Nous en concluons que, dans notre environnement de test, 5000 utilisateurs simultanés représentent la limite supérieure de l'implémentation 3-tier.

Charge moyenne Des charges moyennes ont été exécutées sur un jeu de données réaliste, comprenant environ 90 millions d'enregistrements dans la table de faits (la table principale contenant les événements ou mesures à analyser). Pour préparer ces réponses, le serveur devait consulter trois tables de dimensions (des tables descriptives permettant d'analyser les faits selon différents axes, comme le temps, un produit ou une catégorie) et parcourir de manière aléatoire la table de faits. L'exécution de requêtes aléatoires sur ce jeu de données produisait des réponses JSON pouvant atteindre 120 KB. Une requête SQL typique sur la table de faits prenait environ 100 ms côté base de données, en exécution directe ; cette valeur constituait donc un minimum absolu pour le temps de réponse du serveur applicatif. La charge a été générée par 400 utilisateurs Locust.

Les résultats montrent que l'architecture 2-tier conserve un avantage, mais plus modéré que dans le scénario précédent : les temps de réponse et le débit restaient meilleurs, avec un gain global évalué à environ 15%.

Charge élevée La charge élevée repose sur les mêmes requêtes que la charge moyenne, mais avec un nombre plus important d'utilisateurs simultanés, jusqu'à atteindre la limite du serveur. Avec 800 utilisateurs Locust,

l'architecture 3-tier classique n'a pu mener à bien que 1875 requêtes, toutes les autres ayant échoué. L'architecture 2-tier n'a enregistré que 19 erreurs de timeout sur `10000` requêtes.

Quand on passe de la charge moyenne à la charge élevée, les temps de réponse du 2-tier sont à peu près multipliés par deux. Le débit en revanche varie très peu.

3.2 Pistes de lecture

Objectif de lecture : comprendre pourquoi, dans certains contextes, un choix d'architecture qui paraît moins « naturel » au regard des pratiques dominantes peut redevenir pertinent, et analyser cette décision en mobilisant plusieurs critères de comparaison, au-delà des seules performances.

Contexte / motivation

- Quel problème architectural les auteurs cherchent-ils à résoudre dans le contexte des applications analytiques ?
- Pourquoi les traitements analytiques mettent-ils particulièrement sous tension les échanges entre serveur applicatif et base de données ?
- En quoi l'approche *data-centric* modifie-t-elle la manière de répartir les responsabilités entre les couches du système ?

Critères d'évaluation Au-delà des performances, quels autres critères les auteurs comparent-ils et pourquoi ?

Résultats des comparaisons Dressez un tableau résumant le positionnement de chacune des 2 architectures pour chacun des critères considérés. Identifiez quels critères semblent favoriser le 2-tier, et quels critères continuent à favoriser le 3-tier.

Comportement sous charge élevée Que se passe-t-il sous charge élevée pour les deux architectures ? Que peut-on en déduire sur leur comportement quand on s'approche de la saturation ?

Ouverture Dans quelle mesure les conclusions de cet article sont-elles transférables à une autre application ?

3.3 Questions ouvertes

Objectif : ouvrir la discussion au-delà du cas étudié, en amenant à réfléchir plus généralement à la place de la logique applicative, aux compromis d'architecture et aux conditions de transférabilité des conclusions.

— Peut-on rapprocher la logique des données sans aller jusqu'à un véritable serveur applicatif *in-database* ? Quelles solutions intermédiaires imaginer ?
— Dans quelle mesure les progrès des SGBD relationnels remettent-ils en cause certaines habitudes architecturales héritées des années 2000 ?
— Une architecture plus performante mais moins bien outillée est-elle réellement préférable sur le long terme ?
— Quels risques apparaissent lorsqu'une part croissante de la logique applicative est écrite dans des procédures stockées ?
— Comment comparer deux architectures quand l'une est meilleure pour les performances et l'autre meilleure pour l'outillage, les tests ou l'organisation des équipes ?
— Les choix d'architecture doivent-ils être guidés d'abord par la charge actuelle, ou par la trajectoire d'évolution attendue du système ?
— Dans un système moderne distribué, la vraie frontière architecturale passe-t-elle encore entre client, serveur applicatif et base, ou faut-il raisonner autrement ?

4 Passer à l'échelle

4.1 Document

Le document proposé reprend un podcast de Meta de décembre 2015 : *Under the hood : Broadcasting live video to millions* de Federico Larumbe et Abhishek Mathur, enrichi de quelques informations données à InfoQ.

Facebook a d'abord déployé la vidéo live via Facebook Mentions, une application destinée aux personnalités publiques vérifiées, puis a commencé à élargir progressivement la fonctionnalité à un petit pourcentage d'utilisateurs, au départ aux États-Unis sur iPhone. Cette montée en charge a été un exercice d'ingénierie « à l'échelle Internet » avec deux objectifs distincts mais complémentaires. D'un côté, il fallait encaisser des pics de trafic extrêmes provoqués par des figures publiques capables de déclencher, sans préavis, l'arrivée simultanée de centaines de milliers à plus d'un million de spectateurs, comme cela a pu se produire lors d'un live de Vin Diesel. De l'autre, en ouvrant le live à des utilisateurs non célèbres, l'enjeu devenait aussi de réduire fortement la latence afin de rendre les diffusions réellement interactives, proches d'une conversation.

Le premier problème à résoudre est celui que l'on appelle le thundering herd. Aucun système à très grande échelle n'aime les variations irrégulières et les pics soudains, et un live est précisément le scénario où des masses de personnes demandent les mêmes objets au même moment. Quand une diffusion démarre, des notifications sont envoyées aux abonnés et amis, et la vidéo apparaît aussi dans le fil d'actualité. Si l'émetteur a des millions de followers, une foule de clients peut se connecter simultanément, et si ces clients déclenchent au même instant des cache misses sur les mêmes segments, la charge peut « piétiner » les couches d'infrastructure et remonter jusqu'au cœur du système, provoquant lag, pertes et déconnexions. La meilleure façon d'éviter la ruée est de ne jamais la laisser franchir les portes, autrement dit d'empêcher les clients de frapper directement les serveurs qui gèrent le stream.

Pour cela, Facebook s'appuie sur une architecture en couches de cache. Une vidéo live est découpée en segments HLS[14] de trois secondes, et le lecteur qui affiche la diffusion demande ces segments séquentiellement. Une requête de segment arrive d'abord sur un proxy HTTP dans un edge data center, qui vérifie si le segment se trouve déjà dans un edge cache proche de l'utilisateur. Le contenu présent dans cet edge cache correspond à du contenu déjà demandé auparavant. Si le segment est en cache, il est renvoyé

14. HLS (HTTP Live Streaming) est un protocole de streaming adaptatif basé sur HTTP : la vidéo est découpée en petits segments (ici 3 s) que le lecteur télécharge successivement, en pouvant adapter le débit/qualité selon le réseau.

directement. Si le segment n'est pas en cache, c'est un cache miss et le proxy émet alors une requête HTTP vers le cache d'origine, l'origin cache, qui constitue une seconde couche de cache avec une architecture similaire. Les serveurs d'origin cache peuvent recevoir des requêtes de plusieurs edge caches pour différents segments d'une même diffusion. Si le segment n'est pas non plus dans l'origin cache, il faut alors aller le demander au serveur qui gère cette diffusion, le serveur de traitement du flux live, qui retourne la réponse HTTP contenant le segment. Une fois le segment obtenu, il est mis en cache dans chaque couche au passage, d'abord dans l'origin cache puis dans l'edge cache, et la réponse est servie au client, de sorte que les clients suivants récupèrent le segment plus vite et au plus près d'eux. Avec ce schéma, plus de 98 % des segments sont typiquement déjà disponibles dans un edge cache proche de l'utilisateur et l'origine ne reçoit qu'une fraction des requêtes.

Cette protection par caches fonctionne bien, sauf qu'à l'échelle de Facebook il restait une fuite non négligeable : environ 1,8 % des requêtes parvenaient malgré tout à dépasser l'edge cache. À première vue, 1,8 % semble faible, mais avec un million de viewers cela représente encore des milliers à des dizaines de milliers de requêtes qui remontent vers l'origin cache puis le serveur live, ce qui suffit à créer un goulot d'étranglement. La difficulté spécifique du live est que, contrairement à la vidéo non-live où les spectateurs regardent à des moments différents et où l'on peut anticiper progressivement la montée en charge quand un contenu devient viral, les spectateurs de live demandent au même instant les mêmes segments, souvent sans avertissement. Dans ce contexte, si un segment n'est pas encore en cache, tous les clients peuvent provoquer simultanément un cache miss, et sans mécanisme de protection, toutes ces requêtes remonteraient en cascade vers l'origin cache et jusqu'au serveur de stream, concentrant une charge massive sur un seul point.

Pour éliminer ce risque, Facebook a appliqué une technique appelée request coalescing, c'est-à-dire la fusion ou l'agrégation des requêtes identiques. Concrètement, lorsqu'un segment demandé n'est pas dans l'edge cache, l'edge ne laisse passer qu'une seule requête pour aller chercher le segment, et place les requêtes suivantes pour ce même segment dans une file d'attente. Quand la réponse HTTP arrive, le segment est stocké dans l'edge cache, puis toutes les requêtes en attente sont servies immédiatement depuis l'edge comme des cache hits. Le même principe est appliqué au niveau de l'origin cache pour gérer les requêtes provenant de multiples edge caches, car un même objet peut être demandé simultanément par exemple depuis un edge cache à Chicago et un edge cache à Miami. Les chiffres rapportés illustrent l'impact : la quantité de requêtes remontant de l'edge vers l'origin peut être réduite de 98,2 %, passant par exemple d'environ 200 000 requêtes vers l'origin à environ 3 600, et le serveur de traitement live est lui aussi protégé puisque l'origin coalesce à son tour et n'émet plus qu'une requête par segment vidéo.

Même avec ce mécanisme, un second problème apparaît lors des très gros

lives : la charge sur les serveurs d'edge cache eux-mêmes. Un edge cache typique peut gérer environ 200 000 requêtes « proprement », mais si chaque segment est servi par un edge cache unique, alors une fois le segment ramené et mis en cache, ce serveur doit servir potentiellement un volume énorme de clients, ce qui peut le transformer en goulot d'étranglement. La réponse consiste à répliquer les segments sur plusieurs edge caches dès qu'une diffusion dépasse environ 200 000 spectateurs, en ajoutant typiquement un serveur d'edge cache supplémentaire par tranche de 200 000 viewers. Cette réplication distribue la charge, réduit le pic absorbé par chaque machine et améliore la stabilité de l'expérience de visionnage.

Après la phase « Mentions », qui était surtout un exercice de tenue à l'échelle et d'absorption de pics, l'ouverture du live à davantage de personnes a déplacé le centre de gravité vers un autre critère dominant : la latence. Les utilisateurs non célébrités diffusent plus souvent à de petits groupes qui interagissent, et une latence trop élevée rend les échanges désagréables. Pour descendre la latence des diffusions à quelques secondes, de l'ordre de deux à trois secondes, Facebook a choisi d'activer une lecture via RTMP [15], en plus de l'approche HLS. RTMP est un protocole de streaming qui maintient une connexion TCP persistante entre le lecteur et le serveur pendant toute la diffusion. Contrairement à HLS, qui repose sur HTTP et un modèle pull où le lecteur demande chaque segment, RTMP adopte un modèle push où le serveur envoie en continu les données audio et vidéo au client. Le client peut toujours émettre des commandes comme pause et reprise quand l'utilisateur le demande ou quand le lecteur n'est pas visible. En RTMP, la diffusion est séparée en deux flux, un flux vidéo et un flux audio, découpés en chunks d'environ 4 KB, lesquels peuvent être multiplexés dans la même connexion TCP, c'est-à-dire intercalés. À un débit vidéo de 500 kbps, un chunk représente environ 64 ms de média, ce qui, comparé à des segments HLS de trois secondes, permet un streaming plus « lisse » à travers l'ensemble des composants. Le broadcaster peut envoyer des données dès qu'il a encodé 64 ms de vidéo, le serveur de transcodage peut traiter ce chunk et produire plusieurs versions du flux à différents débits (et donc différentes qualités/résolutions) pour s'adapter aux conditions réseau des spectateurs, puis le chunk est relayé à travers des proxys jusqu'au lecteur. La combinaison du modèle push et de petits chunks réduit le décalage entre le broadcaster et le viewer d'un facteur d'environ cinq, et rend l'expérience plus interactive. La plupart des produits de live s'appuient sur HLS parce que c'est basé sur HTTP et facile à intégrer avec des CDN3 existants, mais Facebook a décidé d'implémenter RTMP pour viser une meilleure expérience live, en modifiant notamment un module de type nginx-rtmp et en développant un proxy RTMP. Les leçons tirées des premiers déploiements à grande échelle (notamment avec Mentions, côté HLS) ont aussi servi à concevoir l'architecture RTMP pour qu'elle tienne

15. Real-Time Messaging Protocol

la charge, y compris vers des millions de broadcasters.

4.2 Pistes de lecture

Sources de saturation Dans ce récit, qu'est-ce qui fait « exploser » le système ? Quelles sont, dans d'autres systèmes, d'autres sources classiques de saturation ? Qu'est-ce que ça change dans votre façon de concevoir ?

Attention aux petits ! 1,8 % de requêtes qui « fuient » l'edge cache, cela vous a-t-il paru spontanément « grave » ? Quelle leçon générale en tirez-vous ?

Scaler vs protéger Si vous deviez résumer l'architecture en une intention, ce serait quoi : « scaler le backend » ou « protéger le backend » ? Comment, dans un autre SI, choisiriez-vous entre ces 2 options ?

Contrôle de concurrence Le request coalescing ressemble à quoi, au fond : un cache, une file d'attente, un verrou, un mécanisme d'admission control ? Quelle idée générale de contrôle de concurrence vous semble réutilisable dans d'autres contextes ?

Observer le déplacement de goulot Après avoir protégé l'origin, un nouveau goulot apparaît à l'edge. Qu'est-ce que ça vous dit sur la nature des optimisations à grande échelle (résolution de goulots « en cascade ») ? Comment vous organiseriez l'observabilité pour voir ce déplacement de goulot en temps réel ?

Ni meilleur ni pire, différent RTMP et HLS ne sont pas présentés comme « meilleur vs pire » mais comme adaptés à des objectifs différents. Quels compromis produit/tech sont implicitement acceptés quand on passe de HLS à RTMP ? Dans un autre SI, quel serait l'équivalent d'un choix de protocole qui « change le modèle » ?

Passage à l'échelle Pourquoi Facebook n'a pas « juste » ajouté plus de serveurs au live stream server ? Qu'est-ce que cela suggère sur les limites du scale-up/scale-out naïf et sur l'intérêt des mécanismes de régulation en amont (proxies, caches, gating) ?

Trajectoire Le texte décrit une trajectoire : d'abord Mentions (figures publiques), puis ouverture plus large, puis focus latence. Qu'est-ce que ça vous apprend sur la manière de faire évoluer une architecture en production sans big-bang ?