* Le principal problème des applications monolithiques sont qu’elles centralisent tous les besoins et qu’elles sont réalisées dans une seule technologie. Il est intéressant d’exploiter les capacités de différents langages pour des buts spécifiques

Ou

Un autre effet pervers de l’architecture monolithique est de rendre une application dépendante à certaines technologies, obligatoirement compatibles entre elles. Choisir Java limitera les évolutions futures aux technologies compatibles avec la JVM, C# aux technologies Microsoft, etc. Si un Framework utilisé par l’application devient obsolète, la migration vers un nouveau peut nécessiter la réécriture complète de l’application avec les risques que cela comporte.

* Quelles que soient les modifications réellement effectuées, pose divers problèmes. Premièrement, le temps de déploiement s’allonge, tendant à limiter leur nombre. En effet, la taille de l’application augmentant, le container prend plus de temps pour effectuer les démarrages applicatifs. En conséquence, le déploiement continu devient plus complexe, ce qui engendre une augmentation des risques associés à chaque déploiement et une perte de productivité.
* Si en début de projet, la simplicité du modèle monolithique favorise grandement la compréhension du code, ceci devient de moins en moins vrai au fur et à mesure que le volume de code augmente, et ce, même avec une organisation rigoureuse de ce dernier. Un nouveau venu mettra davantage de temps à être pleinement opérationnel. Parce qu’il n’y a pas de frontières fortes entre les modules d’une application monolithique, elles auront tendance à s’effacer, affaiblissant la modularité, complexifiant les relations de dépendance. Le code devient de moins en moins lisible et testable, la productivité et la qualité baisse et enfin la dette technique s’accumule.
* Ou
* Chaque développeur doit avoir l’application au complet dans son environnement de développement et doit comprendre la majeure partie de l’application
* Lorsque la charge d’une application augmente, la scalabilité de l’application devient un point critique. Une application monolithique n’ayant qu’une capacité assez limitée (car celle-ci ne se fait que dans une dimension), on ne peut avoir que des copies de l’application entière. Or les différents composants de l’application n’ont vraisemblablement pas les mêmes besoins en ressource. Certains vont faire un usage intensif du processeur, d’autres de la mémoire, d’autres des entrées / sorties. Scaler une application monolithique peut entraîner une augmentation de la consommation des ressources conséquentes de manière inutile.
* **Évolutivité et fiabilité :** Plus le temps passe, plus les nouvelles fonctionnalités métier deviennent complexes, et plus les différentes briques ont d’interactions. On a beau organiser le code en couches et en composants, il y a toujours des cas particuliers et des rustines qui rendent les choses plus floues. Au-delà d’un certain seuil, il devient impossible d’avoir en tête un modèle global du projet. Même avec une base de tests solide, la multiplication des effets de bord de chaque action rend le système moins fiable, et il devient alors plus difficile d’ajouter proprement des nouvelles fonctionnalités et d’effectuer des refactorings.

#### Scalabilité horizontale

Améliorer la scalabilité d’un système peut demander de modifier des éléments structurants du projet.

Plus un projet est gros, plus ces interventions deviennent coûteuses et risquées.

Le risque est alors de se retrouver avec un système qu’il est impossible de faire évoluer pour un nouveau cas d’usage.

#### Innovation technologique

Pour capitaliser les investissements et faciliter la gestion des personnes, il est normal de vouloir avoir une cohérence entre les différents projets d’une entreprise : même manière de travailler, mêmes langages de programmation, mêmes outils.

Chaque projet est invité à suivre des choix transverses, et peut s’en écarter en fonction de ses spécificités, à condition de le justifier.

Pour les gros projets, la même tension a lieu à l’intérieur même des projets : pour éviter la fragmentation, chaque évolution technique doit pouvoir être propagée à l’intégralité du code.

Avec le temps, les modifications deviennent donc plus coûteuses, et il est plus difficile d’introduire de nouveaux outils pour des besoins spécifiques.

#### Innovation métier

Pour répondre aux nouveaux besoins métier, il faut être capable de ménager une zone d’innovation à l’intérieur des projets.

Car si certaines nouveautés sont mises en œuvre par de nouveaux projets, la plupart se font sur des projets existants.

Or plus un projet est gros, plus il est critique pour l’entreprise, moins on va prendre de risques de le modifier pour tester de nouveaux produits ou de nouveaux marchés, et petit à petit les enjeux de stabilité vont prendre le pas sur la capacité d’innovation.

* Les modifications sur l’application peuvent impacter d’autres modules
* Pour assurer la montée en charge, il faut une montée à l’échelle horizontale en déployant l’application au complet sur plusieurs serveurs
* À chaque modification apportée à l’application, le cycle de développement est long (définition du fonctionnel, développement, tests, déploiement)
* Au fil du temps, il est de plus en plus difficile de garder l’architecture modulaire prévue à l’origine
* Chaque développeur doit avoir l’application au complet dans son environnement de développement et doit comprendre la majeure partie de l’application