



Annexe : Monolithe vs Microservices

Défaire les idées reçues

🎯 **Objectif** : Comprendre que monolithe ≠ mauvais code et que l'architecture interne (hexagonale) est orthogonale au mode de déploiement.

Slides :

1. Définitions précises
2. Confusion fréquente
3. Matrice qualité × déploiement
4. Quand choisir quoi ?
5. Migration progressive



Définitions



III Monolithe

Définition : Application déployée comme une seule unité

Caractéristiques :

- 1 processus
- 1 base de code
- 1 déploiement
- Communication in-process (appels de fonction)

Exemples :

- Application Django/Rails classique
- Backend FastAPI complet



Microservices

Définition : Application décomposée en **services autonomes déployables indépendamment**

Caractéristiques :

- N processus
- N bases de code
- N déploiements
- Communication réseau (HTTP, RPC, messages)

Exemples :

- Netflix (~800 services)
- Amazon, Uber
- Architecture event-driven

Architecture vs Déploiement

Confusion fréquente :

"Monolithe = code spaghetti

Microservices = bonne architecture"

✗ FAUX !

Réalité :

- **Monolithe** = mode de **déploiement**
- **Architecture** = qualité **structurelle** du code

Ce sont deux dimensions orthogonales :

	Bien structuré	Mal structuré
Mono	<input checked="" type="checkbox"/> Monolithe	<input checked="" type="checkbox"/> Big Ball

🔍 Détails de la matrice

✓ Monolithe modulaire (BIEN)

Déploiement : 1 app

Structure interne : Hexagonale

```
src/
└── tickets/
    ├── domain/
    ├── ports/
    └── adapters/
└── users/
    ├── domain/
    ├── ports/
    └── adapters/
└── notifications/
    ├── domain/
    ├── ports/
    └── adapters/
```

→ Modules découplés

✗ Big Ball of Mud (MAUVAIS)

Déploiement : 1 app

Structure interne : Chaos

```
src/
├── models.py      (500 lignes)
├── views.py       (800 lignes)
├── utils.py        (1200 lignes)
├── helpers.py     (600 lignes)
└── services.py    (2000 lignes)
```

→ Tout dépend de tout

→ Impossible à tester

→ Changement = risque maximal

Problème : Pas l'architecture, juste du mauvais code



Microservices bien conçus (BIEN)

Déploiement : N services

Chaque service : Hexagonal

Service Tickets:

```
├── domain/  
├── ports/  
└── adapters/
```

Service Users:

```
├── domain/  
├── ports/  
└── adapters/
```

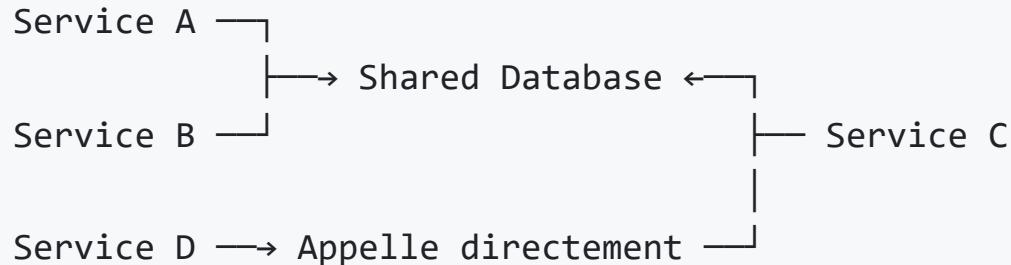
- Bounded contexts clairs
- APIs bien définies
- Autonomie complète

Exemples réels : Netflix, Amazon (après des années d'évolution)

✗ Distributed Monolith (LE PIRE)

Déploiement : N services

Couplage : Maximal



- Complexité des microservices
- Couplage du monolithe
- Pire des deux mondes

Problème : Architecture distribuée sans découplage



Comparaison détaillée

Critère	Monolithe modulaire	Microservices	Distributed Monolith
Complexité déploiement	✓ Faible	⚠ Élevée	✗ Élevée
Complexité code	✓ Faible	⚠ Moyenne	✗ Très élevée
Testabilité	✓ Excellente	✓ Excellente	✗ Difficile
Performance	✓ In-process	⚠ Réseau	✗ Réseau + couplage
Scaling	⚠ Vertical	✓ Horizontal	⚠ Compliqué
Maintenance	✓ Simple	⚠ Distribuée	✗ Cauchemar

Classement : 1 Monolithe modulaire → 2 Microservices bien conçus → 3 À éviter

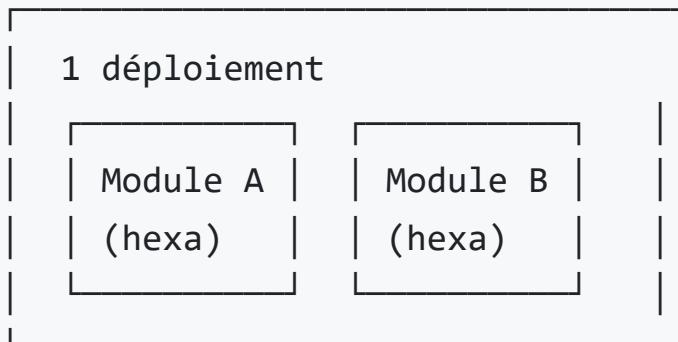
La vraie question

Pas "Monolith ou Microservices ?"

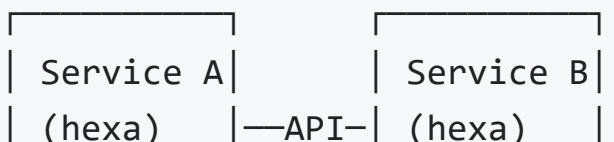
Mais "Mon code est-il bien structuré ?"

Architecture hexagonale fonctionne dans les deux cas :

Monolith hexagonal :



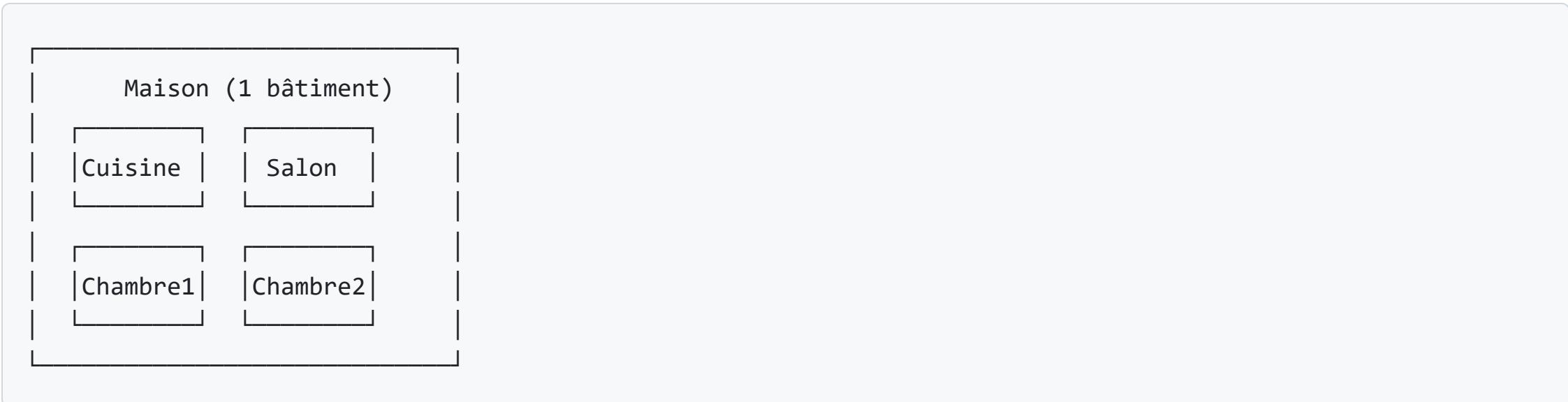
Microservices hexagonaux :





Métaphore : Maison vs Village

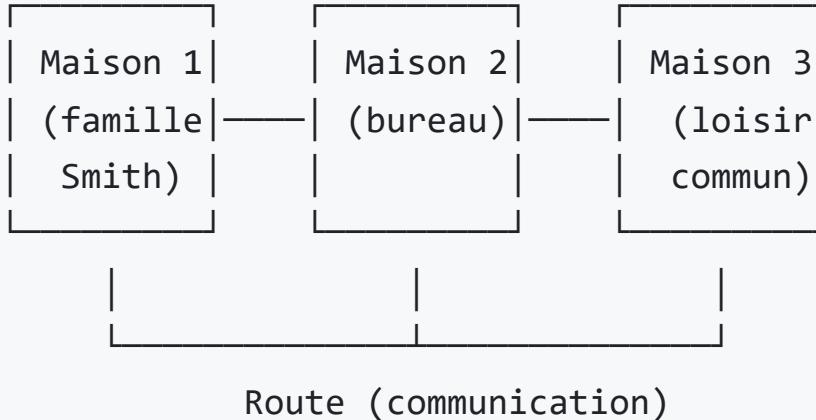
Monolithe modulaire = Maison bien organisée



Avantages :

- Tout sous un même toit
- Facile de passer d'une pièce à l'autre
- Un seul système de chauffage/électricité

Microservices = Village bien planifié



Avantages :

- Chaque maison indépendante
- Pas d'impact si une maison en travaux
- Peut ajouter maisons sans limite

Inconvénients :

- Besoin d'infrastructure (routes)

Distributed Monolith = Village sans plan d'urbanisme

- 🏡 Maisons entassées
- 🚧 Routes qui vont partout
- ⚡ Électricité partagée entre toutes
- 💧 Un seul puit d'eau commun

- Complexité du village
- Dépendances de la maison unique
- Pire scénario

Message clé :

Un monolithe bien conçu bat des microservices mal conçus 99% du temps.

Quand choisir quoi ?

Monolithe (hexagonal) si :

- Équipe < 10 développeurs
- Trafic < 1000 req/s
- Besoin de rapidité de développement
- Domaine métier encore flou
- Budget/compétences DevOps limitées
- Startup / MVP / Projet étudiant ➡ VOUS

Exemples : Shopify (600+ dev, monolithe), GitHub, Basecamp



Microservices si :

- Équipe > 50 développeurs
- Scaling indépendant nécessaire (certaines features × 100)
- Technologies hétérogènes (Python + Java + Go)
- Déploiements indépendants critiques
- Organisation en équipes produit autonomes
- Budget DevOps conséquent

Exemples : Netflix, Amazon, Uber

⚠️ Signaux d'alarme (ne PAS faire microservices)

- ✗ "Microservices c'est moderne, on devrait faire ça"
- ✗ "Pour apprendre" (apprenez sur un side project, pas en prod)
- ✗ "Notre monolithe est mal organisé" (refactorez-le d'abord !)
- ✗ "Pour scaler" (scale le monolithe d'abord, c'est plus simple)
- ✗ Équipe < 10 personnes
- ✗ Pas de compétences DevOps solides

Règle d'or :

Ne faites PAS de microservices tant que le monolithe ne vous fait pas mal.



Évolution progressive

Étape 1 : Monolithe modulaire (NOW)

1 déploiement

Structure hexagonale :

- └ tickets/ (module)
- └ users/ (module)
- └ notifications/ (module)

- Rapide à développer
- Facile à déployer
- Bien structuré

Votre projet BUT = ICI

Étape 2 : Identifier les bounded contexts

Au bout de 6 mois / 1 an :

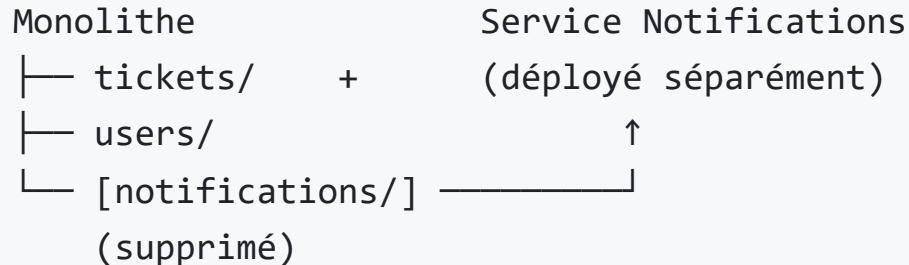
- Quel module change souvent ?
- Quel module a besoin de scale ?
- Quelle équipe pourrait être autonome ?

Exemple :

- Notifications → beaucoup de trafic
- Tickets → change rarement
- Users → critique

Étape 3 : Extraction progressive

Étape 3.1 : Extraire 1 service (le plus évident)



- Communication via API HTTP
- Le reste reste monolith

Pattern : Strangler Fig (étrangler progressivement le monolithe)

Étape 4 : Continuer si ça marche

Si l'extraction a apporté de la valeur :

- Extraire un 2e service
- Puis un 3e
- Etc.

Si l'extraction a été coûteuse sans bénéfice :

- STOP
- Rester monolithe modulaire

Message clé :

Microservices = destination possible, pas point de départ



Exemple concret : Ticketing

Aujourd'hui (Monolithe hexagonal)

```
1 app FastAPI
```

```
src/
└── tickets/
    ├── domain/
    ├── ports/
    └── adapters/
└── users/
└── ...
└── notifications/
└── ...
```

→ Simple, testable, maintenable 

Domain (si besoin)

3 microservices

Service Tickets (Python)

Service Users (Python)

Service Notifications (Go) ← Besoin de perf

Communication : API REST + Events

- Scaling indépendant ✓
- Technologies adaptées ✓
- Mais : complexité × 3 !

Quand ? Quand le monolithe ne suffit plus (rarement en BUT)



Coût réel des microservices

Infrastructure

Monolithe : 1 serveur

Microservices : N serveurs + orchestration

Outils nécessaires :

- Kubernetes / Docker Swarm
- Service discovery (Consul)
- API Gateway (Kong, Traefik)
- Monitoring distribué (Prometheus)
- Logging centralisé (ELK)
- Tracing distribué (Jaeger)
- Message broker (RabbitMQ, Kafka)

Coût : × 5 à × 10

Complexité opérationnelle

Monolithe :

- 1 déploiement
- 1 base de données
- 1 log à consulter
- 1 debugger

Microservices :

- N déploiements (coordination)
- N bases de données (transactions distribuées)
- N logs (tracer une requête = cauchemar)
- Debugging distribué (bonne chance)

Temps DevOps : × 3 à × 5

Développement

Monolithe :

- Tester en local : npm start
- Changer 2 modules : 1 commit
- Onboarding : 1 jour

Microservices :

- Tester en local : Docker Compose (15 services)
- Changer 2 services : 2 commits, 2 déploiements
- Onboarding : 1 semaine

Vélocité initiale : $\div 2$



Cas d'école : Shopify

Chiffres

- **1.7 million** de boutiques
- **600+** développeurs
- **\$5.6 milliards** de revenu annuel

Architecture

Monolithe Rails (toujours aujourd'hui)

Pourquoi ?

- Bien structuré (modulaire)
- Scale verticalement (big servers)
- Équipe productive



Pour vous (BUT2)

Objectif : Maîtriser le monolithe modulaire

Ce que vous allez apprendre :

- Architecture hexagonale (structure interne)
- Découplage (modules indépendants)
- Testabilité (sans dépendances)
- Évolutivité (changer infrastructure sans toucher métier)

Ce que vous NE ferez PAS (et c'est normal) :

- Déploiement distribué
- Service discovery
- Transactions distribuées

Progression naturelle

BUT2 (20h) : Monolithe hexagonal

↓

BUT3 (projet) : Monolithe modulaire avancé

↓

Stage / Alternance : Monolithe en production

↓

M1 : Introduction microservices (théorie)

↓

M2 : Microservices (si projet adapté)

↓

Emploi : 80% monolithes, 20% microservices

Réalité du marché :

- La majorité des entreprises : monolithes bien structurés
- GAFAM + Licornes : microservices
- Startups : monolithe puis migration progressive



Checklist : Votre monolithe est bon si

- [] Modules découplés (un module ignore les autres)
- [] Domaine métier pur (zéro import technique)
- [] Tests unitaires rapides (< 1s)
- [] Changement d'infrastructure facile (swap adapter)
- [] Nouveau développeur productif en < 1 semaine
- [] Déploiement simple (1 commande)
- [] Moins de 100k lignes (au-delà, envisager split)

Si 6-7 ✓ → Excellent monolithe, gardez-le !

? Questions fréquentes

Q : Monolithe = legacy ?

R : Non. Legacy = code mal structuré (peut être monolithe OU microservices).

Q : Microservices = scalable ?

R : Pas automatiquement. Un monolithe peut scaler (Shopify, GitHub).

Q : Combien de lignes avant de splitter ?

R : Pas une question de lignes, mais de douleur. Si ça marche bien, gardez.

Q : Netflix a des microservices, on devrait faire pareil ?

R : Netflix a 800 services ET 2000 développeurs. Vous avez 4 dev. Contexte différent.



Fin de l'annexe

À retenir :

1. Monolithe ≠ Mauvais

→ Bien structuré (hexagonale) = excellent choix

2. Microservices ≠ Solution magique

→ Complexité × 5, bénéfices si gros scale

3. Architecture > Mode de déploiement

→ Hexagonale fonctionne pour les deux

4. Commencez monolithe

→ Splittez SEULEMENT si douleur

5. Votre projet BUT = Monolithe hexagonal

→ C'est le bon choix 