

Atelier Swift Avancé

3 Jours d'Immersion

Swift 6 · SwiftUI & Liquid Glass · Apple Intelligence

Programme des 3 Jours

Jour 1 : Swift 6 & Concurrency Avancée

Jour 2 : SwiftUI Moderne & Liquid Glass

Jour 3 : Apple Intelligence & AI pour Développeurs

Jour 1 : Swift 6 & Concurrency

Matin (9h-12h30)

- Swift 6 : Nouveautés et migration
- `async/await` fondamentaux
- Structured Concurrency & TaskGroup

Après-midi (14h-17h30)

- Actors & Thread Safety
- @MainActor pour UI
- Sendable & Data Race Safety
- TP : Backend du projet fil rouge

Jour 2 : SwiftUI Moderne

Matin (9h-12h30)

- SwiftUI : Cycle de vie & Performance
- Property wrappers (@Observable, @State, etc.)
- Animations & Transitions avancées
- Matched Geometry Effect

Jour 2 : SwiftUI Moderne

Après-midi (14h-17h30)

- Design System : Liquid Glass
- Custom components & modifiers
- Layout & composition
- TP : UI magnifique du projet
- Visite surprise d'un guest à 16h30

Jour 3 : Apple Intelligence

Matin (9h-12h30)

- **Apple Intelligence** : Vue d'ensemble
- App Intents (Siri, Shortcuts, Spotlight)
- Writing Tools integration
- Apple Foundation Models (on-device)
- Image Playground & Visual Intelligence

Jour 3 : Apple Intelligence

Après-midi (14h-17h30)

- AI pour Développeurs : Xcode 16
- Swift Assist & Code Completion
- Tahoe : Analytics & Code Review AI
- TP : Intégration AI complète
- Démo finale & Conclusion

Horaires

9h30 - 10h : Accueil et revue des exercices de la veille

10 - 11h30 : Session théorique 1

11h00 - 11h15 : ☕ Pause

11h15 - 12h30 : Session théorique 2

12h30 - 14h00 : 🍴 Déjeuner

14h00 - 15h15 : Session théorique 3

15h45 - 17h15 : Travaux Pratiques

Projet Fil Rouge : 2 Apps au Choix

Vous choisirez l'une de ces deux applications :

Option A : **Mindful** - Journal Personnel Intelligent

Option B : **MeetMind** - Assistant de Réunions

Les deux apps couvrent **exactement les mêmes concepts techniques** mais dans des contextes différents.

Option A : Mindful

Journal Personnel Intelligent

Features

-  Journal intime avec rich text
-  Photos & souvenirs
-  Mood tracking & sentiment analysis
-  Auto-tagging intelligent
-  iCloud sync multi-device
-  Stats & insights sur votre bien-être

Mindful : Architecture Technique

Jour 1 - Concurrency

- **JournalStorage** actor (CRUD + cache)
- **iCloudSyncManager** avec conflict resolution
- **PhotoProcessor TaskGroup** (compression parallèle)
- Export multi-format (PDF/Markdown/JSON)

Jour 2 - SwiftUI

- Timeline avec Glass cards
- Editor avec Writing Tools
- Mood selector animé
- Stats dashboard

Jour 3 - AI

- Sentiment analysis temps réel
- Auto-tagging (NLP on-device)
- App Intents : "Hey Siri, add journal entry"
- Smart prompts & insights

Option B : MeetMind



Assistant de Réunions Intelligent

Features

-  Prise de notes temps réel
-  Extraction d'action items automatique
-  Gestion des participants
-  Calendar integration
-  Partage multi-canal (Email/Slack)
-  Métriques de productivité

MeetMind : Architecture Technique

Jour 1 - Concurrency

- **MeetingStorage** actor (CRUD + cache)
- **CalendarSyncManager** avec scheduling
- **ShareService TaskGroup** (Email/Slack/PDF parallèle)
- Audio processing (optionnel)

Jour 2 - SwiftUI

- Meeting timeline avec Glass cards
- Notes editor avec auto-save
- Meeting type selector
- Analytics dashboard

Jour 3 - AI

- Action items extraction automatique
- Meeting tone analysis
- App Intents : "Create meeting note"
- Smart follow-up suggestions

Équivalence Technique Garantie

Pattern	Mindful	MeetMind
Actor isolation	JournalStorage	MeetingStorage
Sync complexe	iCloud multi-device	Calendar sync
TaskGroup	Photos processing	Multi-channel share
Writing Tools	Journal editor	Notes editor
NLP on-device	Sentiment analysis	Action items extraction
App Intents	"Add entry"	"Create meeting"

Format de Travail

Sessions Théoriques

- Tous ensemble (10 participants)
- Slides, démos live, discussions
- Questions/réponses interactives

Travaux Pratiques

- 2 groupes selon l'app choisie
- ~5 personnes par groupe
- Blank page starter

Démos Quotidiennes

- Fin de chaque jour : show & tell
- Partage d'astuces entre groupes
- Feedback constructif

Choix de Votre App

Vous choisirez après ma présentation des deux options.

Quelques questions pour vous guider :

- Préférez-vous développer une app **personnelle** ou **professionnelle** ?
- Êtes-vous plus attiré par le **bien-être/créatif** ou la **productivité** ?
- Journal intime ou notes de meetings : quoi vous parle plus ?

Important : Même niveau technique, ce n'est qu'une question de préférence !

Qui suis-je ?



Clément S.



34 ans



Développeur  depuis 19 ans

Expérience :

- ex-Airbnb, ex-Dropbox (San Francisco)
- Aujourd'hui fondateur de **BnC Consulting** : Alma, Credit Agricole, HSBC, ...

Pourquoi cet atelier ?

- Passion pour l'écosystème  et ses nouveautés
- Envie de partager les best practices modernes
- Envie que de belles boîtes techs aient de super appli.

LinkedIn : /in/csauvage

Twitter : @clementsauvage

Github : @csauvage

Instagram : @pyrographik

Philosophie

“Il n'est pas de bonne pédagogie qui ne commence par éveiller le désir d'apprendre.”

- François de Closets

Mon approche pédagogique

Principes

- Théorie juste nécessaire → Pratique maximale
- Code production-ready → Pas de toy examples
- Learning by doing → Vous codez, j'accompagne
- Peer learning → On apprend les uns des autres

Avant l'atelier

- ①, Repos GitHub avec code starter
- ①, Slides en PDF

Pendant l'atelier

- ①, Code examples en live
- ①, Documentation de référence

Après l'atelier

- ① Cheatsheets (Concurrency, SwiftUI, AI)
- ① Projet complet avec solutions
- ① Ressources pour approfondir
- ① Enregistrements (sans garantie)

Prérequis Techniques

Hardware

- Mac avec suffisamment de RAM (16GB / 32 GB recommandé)
- iPhone ou iPad pour tests (simulateur OK)

Logiciels

- macOS Tahoe (26.0) ou plus récent
- Xcode 26+ (beta si nécessaire)
- Compte développeur Apple (gratuit suffit)

Connaissances

- Swift intermédiaire (3 ans d'XP idéal)
- SwiftUI bases (Views, State, modifiers)

À vous !



Tour de table (2-3 min par personne) :

1. Votre prénom
2. Votre expérience iOS (années, types d'apps)
3. Votre motivation pour cet atelier
4. Un défi technique récent que vous avez rencontré
5. Mindful ou MeetMind ? (première impression)

Questions avant de commencer ?

-  Setup technique ?
-  Horaires & pauses ?
-  Apps Mindful vs MeetMind ?
-  Matériel & ressources ?
-  Autre chose ?

Let's Go!

Objectif de ces 3 jours :

- Maîtriser Swift 6,
- Créer des UIs modernes,
- Intégrer l'IA dans vos apps.

À la fin, vous aurez :

- ①, Une app complète production-ready
- ①, Maîtrise des patterns de concurrence Swift 6
- ①, Compétences SwiftUI / Liquid Glass avancées
- ①, Intégration Apple Intelligence



Checklist Rapide

Avant de démarrer, vérifiez que vous avez :

- Xcode 26+ installé et fonctionnel
- Clone du repo GitHub starter
- Accès WiFi stable
- Chargeur branché (longues sessions !)
- Bloc-notes pour prendre des notes
- Questions préparées (si vous en aviez)

On commence dans 5 minutes ! ⏱

Swift 6 & Concurrency

Les Fondamentaux Modernes

Atelier Développeurs iOS pour Le Bon Coin

Agenda (3h)

- Module 1 (30min) : Introduction & Problématique
- Module 2 (50min) : async/await Fondamentaux
- Module 3 (40min) : Structured Concurrency
- Module 4 (40min) : Actors & Thread Safety
- Module 5 (20min) : Sendable & Data Race Safety

Module I



Introduction & Problématique

Timeline Swift

- Swift 5.0 (2019) → Stable ABI
- Swift 5.5 (2021) → `async/await` introduit
- Swift 5.7 (2022) → Actors matures
- Swift 6.0 (2024) → Complete Concurrency Checking

Objectif Principal de Swift

6

Éliminer les data races au compile-time

Exemple : Le Problème

```
// Swift 5.x - Compile mais peut crasher
class Counter {
    var value = 0

    func increment() {
        value += 1 // ! Potentiel data race
    }
}

let counter = Counter()
DispatchQueue.concurrentPerform(iterations: 1000) { _ in
    counter.increment() // 💥
}
```

Swift 6 : Ne Compile Pas

```
// Swift 6 - Error au compile-time
class Counter {
    var value = 0

    func increment() {
        value += 1
    }
}

let counter = Counter()
DispatchQueue.concurrentPerform(iterations: 1000) { _ in
    counter.increment()
    // ✗ Error: "Counter is not Sendable"
}
```

Les 3 Piliers de Swift 6

1. **async/await** : Syntaxe moderne pour code asynchrone
2. **Actors** : Isolation automatique pour thread-safety
3. **Sendable** : Garanties compile-time sur le partage de données

Avant Swift 5.5 : Completion Blocks & Callbacks

```
// L'enfer des callbacks
func fetchUserProfile(completion: @escaping (Data?, Error?) -> Void) {
    fetchFromAPI { data, error in
        guard let data = data else {
            completion(nil, error)
            return
        }
        processImage(data) { image in
            updateUI(image)
            completion(data, nil)
        }
    }
}

// Appel :
fetchUserProfile { data, error in
    print("Résultat prêt !")
}
```

The moumObjective-C : Blocks et Pyramide du Destin

```
// Objective-C style (2015)
[self fetchFromAPI:^{
    if (data) {
        [self processImage:data completion:^(UIImage *image) {
            [self updateUI:image];
            NSLog(@"%@", @"Fini !");
        }];
    }
}];
```

Pourquoi la Concurrency ?

Apps Modernes = Multiples Tâches Simultanées

App Moderne :

- └ Fetch data depuis API (réseau)
- └ Process images (CPU)
- └ Update UI (main thread)
- └ Save to database (I/O)
- └ Sync avec iCloud (réseau)

Tout ça doit se passer SANS bloquer l'UI !

Exemple : Mauvaise Approche

```
// ✗ MAUVAIS - Bloque l'UI  
func loadUserProfile() {  
    let data = fetchFromAPI() // ⏳ 2 secondes  
    let image = processImage(data) // ⏳ 1 seconde  
    updateUI(image) // ⏳ instant  
  
    // Total: 3 secondes de freeze ! 😱  
}
```

Objectif : Non-Blocking

```
// ✅ BON - UI reste fluide
func loadUserProfile() async {
    let data = await fetchFromAPI()
    let image = await processImage(data)
    await updateUI(image)

    // L'utilisateur peut continuer d'utiliser l'app !
}
```

L'Évolution : Callback Hell

```
// 🤯 Pyramid of Doom
func fetchUser(id: String, completion: @escaping (User?, Error?) -> Void) {
    URLSession.shared.dataTask(with: url) { data, response, error in
        guard let data = data else {
            completion(nil, error)
            return
        }

        parseJSON(data) { user, error in
            guard let user = user else {
                completion(nil, error)
                return
            }

            fetchAvatar(user.avatarURL) { image, error in
                var updatedUser = user
                updatedUser.avatar = image
                completion(updatedUser, nil)
            }
        }
    }.resume()
}
```

L'Évolution : `async/await`

```
// Clair et lisible
func fetchUser(id: String) async throws -> User {
    let (data, _) = try await URLSession.shared.data(from: url)
    var user = try JSONDecoder().decode(User.self, from: data)
    user.avatar = try await fetchAvatar(user.avatarURL)
    return user
}

// Utilisation simple :
let user = try await fetchUser(id: "123")
updateUI(user)
```

Module 2

async/await - Les Fondamentaux

Déclarer une Fonction Asynchrone

```
// Fonction asynchrone simple
func fetchData() async -> Data {
    // Code asynchrone
}

// Fonction asynchrone qui peut throw
func fetchData() async throws -> Data {
    // Code asynchrone qui peut échouer
}

// Fonction synchrone (normale)
func processData(_ data: Data) -> String {
    // Code synchrone
}
```

Appeler une Fonction Asynchrone

```
// ✗ Erreur - await manquant
func loadData() {
    let data = fetchData() // Compile Error!
}
```

```
// ✓ Correct - avec await
func loadData() async {
    let data = await fetchData()
    print(data)
}
```

```
// ✓ Avec error handling
func loadData() async {
    do {
        let data = try await fetchData()
        print(data)
    } catch {
        print("Error: \(error)")
    }
}
```

Règle : async Propagation

```
// Si une fonction appelle du code async,  
// elle doit être async  
func processAndSave() async throws {  
    let data = try await fetchData() // async  
    let processed = process(data) // sync  
    try await save(processed) // async  
}  
  
// ✗ Ne compile pas si processAndSave n'est pas async
```

Suspension Points

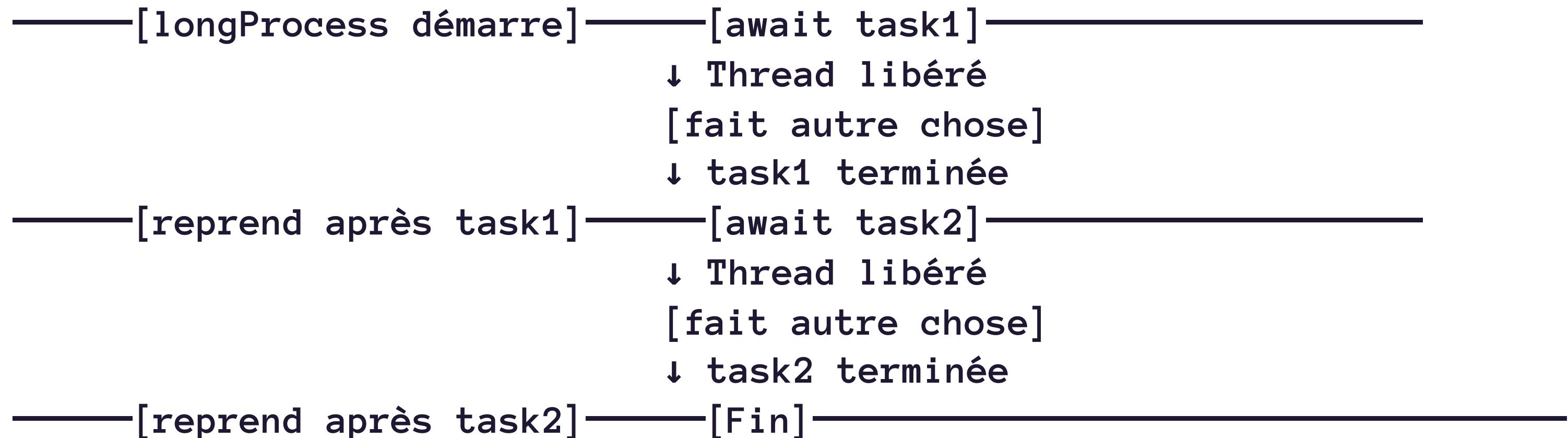
```
func longProcess() async {
    print("1. Début")

    await task1() // ● Suspension point
    print("2. Task1 terminée")

    await task2() // ● Suspension point
    print("3. Task2 terminée")

    print("4. Fin")
}
```

Visualisation : Timeline du Thread



Exemple Concret

```
func downloadAndProcess() async {
    print("Début téléchargement")

    // Suspension : thread libéré pendant le réseau
    let data = await download() // 2 secondes

    print("Téléchargement terminé, processing...")

    // Suspension : thread libéré pendant le processing
    let result = await process(data) // 1 seconde

    print("Tout terminé !")
}
```

async let : Parallélisme Simple

Exécution Séquentielle (Lente)

```
func fetchAllData() async throws -> (User, [Post], [Comment]) {  
    let user = try await fetchUser() // ⏳ 1 sec  
    let posts = try await fetchPosts() // ⏳ 1 sec  
    let comments = try await fetchComments() // ⏳ 1 sec  
  
    return (user, posts, comments)  
    // Total: 3 secondes 😰  
}
```

async let : Exécution Parallèle (Rapide)

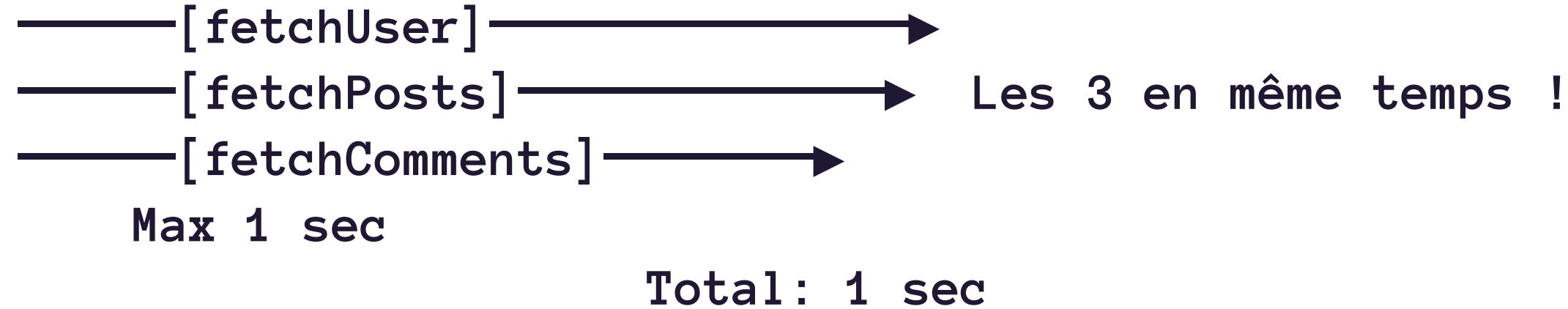
```
func fetchAllData() async throws -> (User, [Post], [Comment]) {  
    // Démarrer les 3 en parallèle  
    async let user = fetchUser()  
    async let posts = fetchPosts()  
    async let comments = fetchComments()  
  
    // Attendre que tout soit fini  
    return try await (user, posts, comments)  
    // Total: ~1 seconde (le plus lent) 🚀  
}
```

Comment ça Marche ?

Sans `async let` (séquentiel) :



Avec `async let` (parallèle) :



Règles Importantes : `async let`

```
func example() async throws {
    // 1. Déclarer avec async let
    async let result1 = fetch1()
    async let result2 = fetch2()

    // 2. DOIT await avant la fin du scope
    let (r1, r2) = try await (result1, result2)

    // 3. Si on quitte le scope sans await, erreur !
}
```

✗ Erreur Courante

```
// ✗ Erreur - async let non awaited  
func wrong() async {  
    async let data = fetch()  
    return // Compile Error!  
}
```

```
// ✓ Correct  
func correct() async {  
    async let data = fetch()  
    let result = await data // OK  
    return  
}
```

Exemple Pratique : Image Thumbnails

```
func generateThumbnails(_ images: [UIImage]) async -> [UIImage] {  
    // Générer 4 thumbnails en parallèle  
    async let thumb1 = resize(images[0])  
    async let thumb2 = resize(images[1])  
    async let thumb3 = resize(images[2])  
    async let thumb4 = resize(images[3])  
  
    return await [thumb1, thumb2, thumb3, thumb4]  
    // 4x plus rapide que séquentiel !  
}
```

Gestion d'Erreurs : try await

```
func fetchAndParse() async throws -> User {
    // Les deux peuvent throw
    let data = try await fetchData()
    let user = try parseJSON(data)
    return user
}

// Utilisation
do {
    let user = try await fetchAndParse()
    print("Success: \(user.name)")
} catch let error as NetworkError {
    print("Network error: \(error)")
} catch let error as ParseError {
    print("Parse error: \(error)")
} catch {
    print("Unknown error: \(error)")
}
```

Erreurs Partielles avec `async let`

```
func fetchMultiple() async throws -> (User?, [Post]?) {  
    async let user = fetchUser()                      // Peut throw  
    async let posts = fetchPosts()                     // Peut throw  
  
    // Si UNE des deux throw, TOUT throw  
    return try await (user, posts)  
}
```

Alternative : Gérer Erreurs Individuellement

```
func fetchMultipleSafe() async -> (User?, [Post]?) {
    async let userResult = Result { try await fetchUser() }
    async let postsResult = Result { try await fetchPosts() }

    let (user, posts) = await (userResult, postsResult)

    return (try? user.get(), try? posts.get())
}
```

Pattern : Retry avec async/await

```
func fetchWithRetry(maxAttempts: Int = 3) async throws -> Data {
    var lastError: Error?

    for attempt in 1...maxAttempts {
        do {
            return try await fetchData()
        } catch {
            lastError = error
            print("Attempt \(attempt) failed: \(error)")

            if attempt < maxAttempts {
                // Exponential backoff
                try await Task.sleep(for: .seconds(attempt))
            }
        }
    }

    throw lastError ?? FetchError.unknown
}
```

Module 3



Structured Concurrency & Task Management

Problème : Unstructured Concurrency

```
// ✗ Dangereux - Tasks "orphelines"
func badExample() {
    Task {
        await longRunningOperation1()
    }

    Task {
        await longRunningOperation2()
    }

    return // Function termine, mais les Tasks continuent !
}
```

Problèmes Unstructured

- Tasks peuvent outlive leur parent
- Pas de gestion des erreurs centralisée
- Impossible d'annuler facilement
- Memory leaks potentiels
- Comportement non-déterministe

Solution : Structured Concurrency

```
// ✓ Safe – Toutes les tasks sont gérées
func goodExample() async throws {
    try await withThrowingTaskGroup(of: Void.self) { group in
        group.addTask { await longRunningOperation1() }
        group.addTask { await longRunningOperation2() }

        // Attendre que TOUTES les tasks soient finies
        try await group.waitForAll()
    }
    // Ici, GARANTIE que tout est terminé
}
```

Avantages Structured Concurrency

- ① Lifetime contrôlé
- ① Erreurs propagées automatiquement
- ① Annulation en cascade
- ① Pas de leaks
- ① Prévisible et testable

withTaskGroup - Sans Erreurs

```
func processImages(_ images: [UIImage]) async -> [ProcessedImage] {
    await withTaskGroup(of: ProcessedImage.self) { group in
        // Ajouter une task pour chaque image
        for image in images {
            group.addTask {
                await processImage(image)
            }
        }

        // Collecter tous les résultats
        var results: [ProcessedImage] = []
        for await result in group {
            results.append(result)
        }
        return results
    }
}
```

withThrowingTaskGroup - Avec Erreurs

```
func fetchAllArticles(ids: [String]) async throws -> [Article] {
    try await withThrowingTaskGroup(of: Article.self) { group in
        for id in ids {
            group.addTask {
                try await fetchArticle(id: id)
            }
        }
    }

    // Si UNE task throw, tout le groupe throw
    var articles: [Article] = []
    for try await article in group {
        articles.append(article)
    }
    return articles
}
```

Gestion d'Erreurs Partielles

```
func fetchWithPartialFailure(ids: [String]) async -> [Article] {
    await withTaskGroup(of: Article?.self) { group in
        for id in ids {
            group.addTask {
                // Capturer l'erreur, retourner nil
                try? await fetchArticle(id: id)
            }
        }

        var articles: [Article] = []
        for await article in group {
            if let article = article {
                articles.append(article)
            }
        }
        return articles
    }
    // Résultat: articles qui ont réussi, ignore les erreurs
}
```

Pattern : Limitation de Concurrence

```
func processWithLimit(
    items: [Item],
    maxConcurrent: Int
) async -> [Result] {
    await withTaskGroup(of: Result.self) { group in
        var results: [Result] = []
        var iterator = items.makeIterator()
        var activeCount = 0

        // Lancer les premières tasks
        while activeCount < maxConcurrent,
            let item = iterator.next() {
            group.addTask { await process(item) }
            activeCount += 1
        }

        // Pour chaque résultat, lancer une nouvelle task
        for await result in group {
            results.append(result)
            if let nextItem = iterator.next() {
                group.addTask { await process(nextItem) }
            }
        }
        return results
    }
}
```

Task Lifecycle & Cancellation

```
// Créer une task
let task = Task {
    await longRunningOperation()
}

// Attendre le résultat
let result = await task.value

// Annuler
task.cancel()

// Vérifier si annulée
if Task.isCancelled {
    return // Sortir tôt
}
```

Cooperative Cancellation

```
func processData(_ items: [Item]) async throws {
    for item in items {
        // Vérifier cancellation régulièrement
        try Task.checkCancellation()

        await process(item)
    }
}

// Alternative
func processData2(_ items: [Item]) async {
    for item in items {
        if Task.isCancelled {
            print("Task cancelled, stopping...")
            return
        }
        await process(item)
    }
}
```

Pattern : Debouncing

```
class SearchViewModel: ObservableObject {
    @Published var searchText = ""
    private var searchTask: Task<Void, Never>?

    func onSearchTextChanged() {
        // Annuler recherche précédente
        searchTask?.cancel()

        // Créer nouvelle recherche avec délai
        searchTask = Task {
            try? await Task.sleep(for: .milliseconds(300))

            guard !Task.isCancelled else { return }

            await performSearch(searchText)
        }
    }
}
```

Task Priority

```
// Priorités disponibles
Task(priority: .high) {
    await criticalOperation()
}

Task(priority: .medium) {
    await normalOperation()
}

Task(priority: .low) {
    await backgroundOperation()
}

Task(priority: .background) {
    await heavyComputation()
}
```

Module 4



Actors & Thread Safety

Le Problème : Data Races

```
// ✖ DANGEREUX - Data race !
class Counter {
    var value = 0

    func increment() {
        value += 1 // Pas thread-safe !
    }
}

let counter = Counter()

// Lancer 1000 incrément en parallèle
await withTaskGroup(of: Void.self) { group in
    for _ in 0..<1000 {
        group.addTask {
            counter.increment()
        }
    }
}

print(counter.value)
// Devrait être 1000, mais... 987, 992, aléatoire ! 💥
```

Visualisation du Data Race

Thread 1:

1. Lit value (0)
2. Calcule $0 + 1 = 1$
3. Écrit value = 1

Thread 2:

1. Lit value (0)
2. Calcule $0 + 1 = 1$
3. Écrit value = 1

Résultat final: value = 1 (devrait être 2!)

Solutions Traditionnelles

```
// Option 1: Locks (verbeux, dangereux)
class Counter {
    private var value = 0
    private let lock = NSLock()

    func increment() {
        lock.lock()
        value += 1
        lock.unlock() // Oubli = deadlock !
    }
}

// Option 2: DispatchQueue (mieux mais verbeux)
class Counter {
    private var value = 0
    private let queue = DispatchQueue(label: "counter")

    func increment() {
        queue.sync {
            value += 1
        }
    }
}
```

Solution : Actors

```
// ✓ SAFE - Actor garantit thread-safety
actor Counter {
    var value = 0

    func increment() {
        value += 1 // Thread-safe automatiquement !
    }

    func getValue() -> Int {
        value
    }
}

let counter = Counter()

// Utilisation avec await
await withTaskGroup(of: Void.self) { group in
    for _ in 0..<1000 {
        group.addTask {
            await counter.increment()
        }
    }
}

print(await counter.getValue()) // TOUJOURS 1000 ! ✓
```

Comment ça Marche ?

Actor = Mailbox Pattern

```
Thread 1 → [increment] → Actor Counter  
Thread 2 → [increment] → Actor Counter  
Thread 3 → [getValue]   → value = 0
```

Timeline:

1. Thread 1 envoie "increment" → Actor traite → value = 1
2. Thread 2 envoie "increment" → Actor traite → value = 2
3. Thread 3 envoie "getValue" → Actor traite → retourne 2

Pas de conflit possible !

Règles des Actors

```
actor DataStore {
    // 1. Propriétés privées (isolation)
    private var data: [String: Data] = [:]

    // 2. Méthodes accessibles avec await
    func save(key: String, value: Data) {
        data[key] = value // Thread-safe
    }

    func load(key: String) -> Data? {
        data[key] // Thread-safe
    }

    // 3. nonisolated pour méthodes synchrones
    nonisolated func description() -> String {
        "DataStore actor"
        // Pas d'accès aux propriétés
    }
}

// Utilisation
let store = DataStore()
await store.save(key: "user", value: userData)
let data = await store.load(key: "user")
```

Actor Reentrancy !

```
actor DownloadManager {
    private var cache: [URL: Data] = [:]

    func download(url: URL) async throws -> Data {
        // Si déjà en cache
        if let cached = cache[url] {
            return cached
        }

        // Télécharger (suspension point!)
        let data = try await URLSession.shared.data(from: url).0

        // ⚠ ATTENTION : L'actor peut avoir été accédé
        // pendant le download !
        // cache pourrait avoir changé !

        cache[url] = data
        return data
    }
}
```

Solution : Re-vérifier après await

```
actor DownloadManager {
    private var cache: [URL: Data] = [:]

    func download(url: URL) async throws -> Data {
        if let cached = cache[url] {
            return cached
        }

        let data = try await URLSession.shared.data(from: url).0

        // Re-vérifier le cache (peut-être ajouté entre temps)
        if let cached = cache[url] {
            return cached
        }

        cache[url] = data
        return data
    }
}
```

@MainActor : UI Thread Safety

Le Problème UI

```
// ✗ ERREUR - UI update hors main thread !
func loadData() async {
    let data = await fetchFromAPI()

    // Cette ligne peut s'exécuter sur n'importe quel thread
    label.text = data // ⚡ Crash ou comportement bizarre !
}
```

Solution : @MainActor

```
// Méthode 1 : Marquer la classe entière
@MainActor
class ViewModel: ObservableObject {
    @Published var text = ""
    @Published var isLoading = false

    // Toutes ces méthodes sont garanties sur main thread
    func loadData() async {
        isLoading = true

        let data = await fetchFromAPI()

        text = data // Automatiquement sur main thread !
        isLoading = false
    }
}
```

@MainActor sur Méthode

```
class Service {
    @MainActor
    func updateUI(data: String) {
        label.text = data // Garanti main thread
    }

    func processData() async {
        let data = await fetch()
        await updateUI(data: data) // Bascule sur main thread
    }
}
```

MainActor.run

```
func example() async {
    let data = await fetchData()

    // Forcer exécution sur main thread
    await MainActor.run {
        label.text = data
        progressBar.isHidden = true
        reloadButton.isEnabled = true
    }
}
```

@MainActor avec SwiftUI

```
// SwiftUI Views sont automatiquement @MainActor
struct ContentView: View {
    @StateObject var viewModel = ViewModel() // @MainActor

    var body: some View {
        Text(viewModel.text)
            .task {
                // Démarre sur main actor
                await viewModel.loadData()
                // Retourne sur main actor
            }
    }
}

// Observable macro (iOS 17+) avec @MainActor
@Observable
@MainActor
class ViewModel {
    var text = ""

    func load() async {
        text = await fetchData() // Safe
    }
}
```

nonisolated

```
@MainActor
class ViewModel {
    var data = ""

    // Cette méthode DOIT être sur main thread
    func updateUI() {
        // Access data OK
    }

    // Cette méthode peut être appelée de n'importe où
    nonisolated func heavyComputation() -> Int {
        // ✗ Ne peut PAS accéder à data
        // ✓ Peut faire du calcul pur
        return (1...1000).reduce(0, +)
    }
}
```

Module 5

Sendable & Data Race Safety

Qu'est-ce que Sendable ?

"Ce type peut être partagé entre threads sans danger"

```
// Types Sendable par défaut
let number: Int = 42           // ✓ Sendable (value type)
let text: String = "Hello"     // ✓ Sendable (value type)
let array: [Int] = [1, 2, 3]    // ✓ Sendable (éléments Sendable)

// Types NON-Sendable par défaut
class User {                  // ✗ Pas Sendable
    var name: String          // (reference type mutable)
}
```

Conformance Automatique

```
// Struct avec propriétés Sendable → Sendable automatiquement
struct Article: Sendable { // ✓ OK
    let id: UUID
    let title: String
    let content: String
}

// Struct avec propriété non-Sendable → Erreur
struct Post: Sendable { // ✗ Erreur
    let title: String
    var author: User           // User n'est pas Sendable
}
```

Rendre un Type Sendable

```
// Option 1 : Struct immuable
struct Point: Sendable {
    let x: Int
    let y: Int
}

// Option 2 : Class immuable (final + let)
final class Configuration: Sendable {
    let apiKey: String
    let baseURL: URL

    init(apiKey: String, baseURL: URL) {
        self.apiKey = apiKey
        self.baseURL = baseURL
    }
}

// Option 3 : Actor (toujours Sendable)
actor Cache: Sendable { // Implicit
    private var data: [String: Data] = [:]
```

@unchecked Sendable !

```
// Utilisez SEULEMENT si vous garantissez
// thread-safety manuellement
class LegacyCache: @unchecked Sendable {
    private var data: [String: Data] = [:]
    private let lock = NSLock()

    func save(key: String, value: Data) {
        lock.lock()
        data[key] = value
        lock.unlock()
    }
}
```

Closures Sendable

```
// ✗ Erreur – capture de variable mutable
func example() {
    var count = 0

    Task {
        count += 1 // ✗ Erreur: mutation de capture non-Sendable
    }
}

// ✓ Solution 1 : Capturer la valeur
func example() {
    var count = 0
    let capturedCount = count

    Task {
        print(capturedCount) // OK
    }
}

// ✓ Solution 2 : Actor
actor Counter {
    var count = 0
    func increment() { count += 1 }
}

func example() {
    let counter = Counter()
    Task {
        await counter.increment() // OK
    }
}
```

@Sendable Closures

```
// Fonction qui prend une closure @Sendable
func performAsync(
    operation: @Sendable () async -> Void
) async {
    await operation()
}

// ✅ OK - closure ne capture rien
performAsync {
    await someWork()
}

// ❌ Erreur - capture non-Sendable
var mutableState = State()
performAsync {
    mutableState.update() // Erreur
}

// ✅ OK - capture Sendable
let immutableValue = 42
performAsync {
    print(immutableValue) // OK
}
```

Exemples Pratiques

```
// URLSession - retourne Sendable
func fetch() async throws -> Data {
    let (data, _) = try await URLSession.shared.data(from: url)
    return data // Data est Sendable ✅
}

// TaskGroup avec Sendable
func processItems(_ items: [Item]) async -> [Result] {
    await withTaskGroup(of: Result.self) { group in
        // Result doit être Sendable
        for item in items {
            group.addTask {
                return process(item)
            }
        }
    }

    var results: [Result] = []
    for await result in group {
        results.append(result)
    }
    return results
}
```

Models Typiques

```
// Pour API responses
struct User: Codable, Sendable {
    let id: UUID
    let name: String
    let email: String
}

// Pour SwiftUI
struct Article: Identifiable, Sendable {
    let id: UUID
    let title: String
    let content: String
}
```

Récapitulatif : Les 5 Concepts Clés

1. **async/await** : Syntaxe moderne pour code asynchrone
2. **Structured Concurrency** : TaskGroup pour parallélisme contrôlé
3. **Actors** : Thread-safety automatique
4. **@MainActor** : UI updates garantis sur main thread
5. **Sendable** : Compile-time safety pour partage entre threads

Avant → Après

```
// AVANT (Swift 5 – Callback Hell)
func fetchUser(completion: @escaping (User?, Error?) -> Void) {
    DispatchQueue.global().async {
        URLSession.shared.dataTask(with: url) { data, _, error in
            guard let data = data else {
                DispatchQueue.main.async {
                    completion(nil, error)
                }
                return
            }
            let user = try? JSONDecoder().decode(User.self, from: data)
            DispatchQueue.main.async {
                completion(user, nil)
            }
        }.resume()
    }
}
```

Avant → Après

```
// APRÈS (Swift 6 - async/await)
func fetchUser() async throws -> User {
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode(User.self, from: data)
}
```

Checklist Migration Swift 6

- ①, Remplacer completion handlers par async/await
- ①, Convertir DispatchQueue en actors
- ①, Marquer ViewModels avec @MainActor
- ①, Rendre models Sendable
- ①, Activer Complete Concurrency Checking
- ①, Corriger warnings progressivement
- ①, Tests de non-régression

Ressources

- Documentation : docs.swift.org/swift-book/.../concurrency
- WWDC Sessions :
 - WWDC21: Meet async/await in Swift
 - WWDC21: Protect mutable state with Swift actors
 - WWDC22: Eliminate data races using Swift Concurrency
 - WWDC23: Beyond the basics of structured concurrency

Swift Evolution Proposals

- SE-0296 : `async/await`
- SE-0304 : Structured Concurrency
- SE-0306 : Actors
- SE-0302 : Sendable and `@Sendable` closures
- SE-0414 : Region based isolation

Questions Fréquentes

Q: Dois-je migrer tout mon code vers `async/await` ?

R: Non, migration progressive. Nouveaux features d'abord, puis refacto.

Q: `async/await` remplace complètement GCD ?

R: Pour la plupart des cas oui, mais GCD reste pour cas spécifiques.

Q: Les actors ont-ils un coût performance ?

R: Minimal. Comparable à un `dispatch_async`.

Q: Puis-je utiliser `async/await` avec Combine ?

R: Oui ! `.values` convertit un Publisher en AsyncSequence.

Merci !



Questions ?



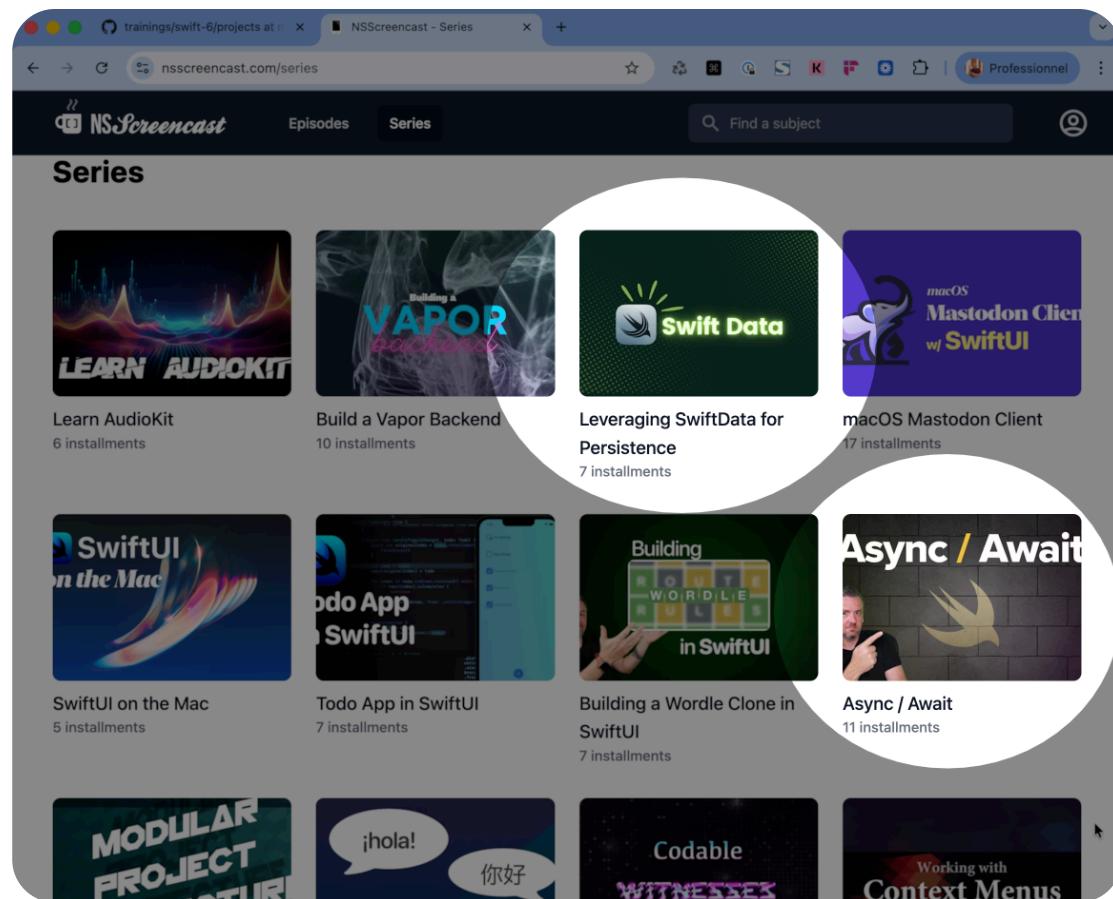
Prochaine étape : TPs pratiques 

SwiftUI Moderne & Liquid Glass

Design iOS 26 & Performance

Jour 2 - Atelier Swift Avancé

Ressource utile



NSScreencast.com

Au menu du chef

Partie 1 (40min) : Cycle de Vie SwiftUI

Partie 2 (50min) : Patterns de Performance

Partie 3 (45min) : Animations & Transitions

Partie 4 (55min) : Liquid Glass iOS 26

Après-midi : TP - Implémentation de votre UI

16h00 / 16h30 : Fireside Chat avec un special guest

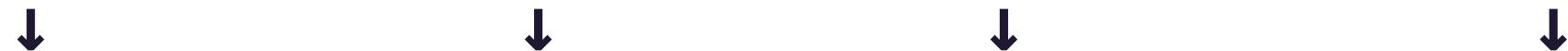
Partie I



Cycle de Vie SwiftUI

Comment SwiftUI Fonctionne

View Struct → body → View Tree → UIView/NSView



Concepts clés :

- Views = descriptions, pas des objets
- Body recalculé à chaque changement
- Diffing intelligent du View Tree
- Updates minimaux du vrai UI

View Identity & Lifecycle

```
struct ContentView: View {  
    @State private var count = 0  
  
    var body: some View {  
        VStack {  
            Text("Count: \(count)")  
                .id(count) // Force nouvelle identity  
  
            Button("Increment") {  
                count += 1  
            }  
        }  
    }  
}
```

Identity détermine :

- Si une View est "la même" entre updates
- Quand l'état est préservé ou reset
- Ordre des animations

@State, le property wrapper fondamental

```
struct CounterView: View {  
    @State private var count = 0  
  
    var body: some View {  
        Button("Count: \(count)") {  
            count += 1 // Déclenche re-render  
        }  
    }  
}
```

Règles :

- **@State** : source de vérité locale à la View
- **private** toujours (encapsulation)
- Value types uniquement (Int, String, struct)
- SwiftUI gère le storage

@Binding pour communication parent-enfant

```
struct ParentView: View {  
    @State private var isOn = false  
  
    var body: some View {  
        ToggleView(isOn: $isOn) // $ = Binding  
    }  
}  
  
struct ToggleView: View {  
    @Binding var isOn: Bool  
  
    var body: some View {  
        Toggle("Switch", isOn: $isOn)  
    }  
}
```

@Binding : référence bidirectionnelle à un **@State**

- Lecture ET écriture
- Pas de stockage (pointe vers **@State** parent)

Property Wrappers : @Observable (iOS 17+)

```
import Observation

@Observable
class ViewModel {
    var name = ""
    var age = 0

    func updateName(_ newName: String) {
        name = newName
    }
}

struct ContentView: View {
    @State var viewModel = ViewModel()

    var body: some View {
        TextField("Name", text: $viewModel.name)
    }
}
```

@Observable vs ObservableObject

```
// Ancien (iOS 13–16)
class OldViewModel: ObservableObject {
    @Published var name = ""
    @Published var age = 0
}
// Usage : @StateObject, @ObservedObject

// Nouveau (iOS 17+)
@Observable
class NewViewModel {
    var name = ""
    var age = 0
}
// Usage : @State, @Bindable
```

Avantages

@Observable :

- Syntaxe plus simple
- Performance améliorée (tracking granulaire)
- Marche avec optionals et collections

@Environment pour valeurs injectées dans la hiérarchie

```
struct ContentView: View {
    @Environment(\.colorScheme) var colorScheme
    @Environment(\.dismiss) var dismiss

    var body: some View {
        Text("Mode: \(colorScheme == .dark ? "Dark" : "Light")")
        Button("Close") {
            dismiss()
        }
    }
}

// Custom environment
private struct ThemeKey: EnvironmentKey {
    static let defaultValue = Theme.default
}

extension EnvironmentValues {
    var theme: Theme {
        get { self[ThemeKey.self] }
        set { self[ThemeKey.self] = newValue }
    }
}
```

⚠️ Les Environment Values se propagent vers le bas de la hiérarchie uniquement

Custom Environment Values

```
// 1. Définir une EnvironmentKey
private struct ThemeKey: EnvironmentKey {
    static let defaultValue = AppTheme.default
}

// 2. Étendre EnvironmentValues
extension EnvironmentValues {
    var appTheme: AppTheme {
        get { self[ThemeKey.self] }
        set { self[ThemeKey.self] = newValue }
    }
}

// 3. Injecter dans la hiérarchie
ContentView()
    .environment(\.appTheme, .premium)

// 4. Utiliser dans n'importe quelle View
struct MyView: View {
    @Environment(\.appTheme) var theme

    var body: some View {
        Text("Hello").foregroundStyle(theme.primaryColor)
    }
}
```

View Updates : Dependencies¹

```
struct ProfileView: View {  
    let user: User  
    @State private var isExpanded = false  
  
    var body: some View {  
        VStack {  
            Text(user.name) // Dépend de user  
            if isExpanded { // Dépend de isExpanded  
                Text(user.bio)  
            }  
        }  
    }  
}
```

- Dependency tracking :**
- SwiftUI détecte automatiquement les dépendances
 - View se re-render si une dépendance change
 - Parties non-affectées ne se recalculent pas

¹ WWDC23 "Demystify SwiftUI performance"

Partie 2



Patterns de Performance

Métriques de performance fondamentales

Les 3 métriques clés :

1. **Hangs** : Délai dans l'apparition d'une View
2. **Hitches** : Frames perdus pendant animation/scroll
3. **Memory** : Utilisation mémoire excessive

Objectif :

- 60 fps = 16.7ms par frame
- 120 fps (ProMotion) = 8.3ms par frame

Outils :

- Instruments (Time Profiler, SwiftUI)
- Xcode Debug Navigator
- Metal System Trace

Anti-Pattern : Expensive Body

```
// ✗ MAUVAIS - Calcul lourd dans body
struct SlowView: View {
    @State private var data: [Item]

    var body: some View {
        let processedData = data
            .filter { $0.isActive }
            .sorted { $0.name < $1.name }
            .map { processItem($0) } // Lourd !

        List(processedData) { item in
            Text(item.name)
        }
    }
}
```

Problème : Ce calcul se refait à CHAQUE re-render !

Solution : Computed Properties

```
// ✓ BON - Calcul en computed property
struct FastView: View {
    @State private var data: [Item]

    private var processedData: [ProcessedItem] {
        data
            .filter { $0.isActive }
            .sorted { $0.name < $1.name }
            .map { processItem($0) }
    }

    var body: some View {
        List(processedData) { item in
            Text(item.name)
        }
    }
}
```

Avantage : Même résultat, mais SwiftUI peut mieux optimiser.

Pattern : ViewModels avec @Observable

```
@Observable
class ListViewModel {
    private(set) var processedData: [ProcessedItem] = []

    var rawData: [Item] = [] {
        didSet {
            updateProcessedData()
        }
    }

    private func updateProcessedData() {
        processedData = rawData
            .filter { $0.isActive }
            .sorted { $0.name < $1.name }
            .map { processItem($0) }
    }
}

struct FastView: View {
    @State private var viewModel = ListViewModel()

    var body: some View {
        List(viewModel.processedData) { item in
            Text(item.name)
        }
    }
}
```

LazyVStack vs VStack

Règle : Utilisez Lazy* pour listes longues (>50 items)

```
// VStack - Tous les enfants créés immédiatement
VStack {
    ForEach(1...1000, id: \.self) { i in
        HeavyView(number: i) // 1000 Views créées !
    }
}

// LazyVStack - Crées seulement quand visibles
ScrollView {
    LazyVStack {
        ForEach(1...1000, id: \.self) { i in
            HeavyView(number: i) // Seulement ~20 à la fois
        }
    }
}
```

Identifiable & id()

```
struct Item: Identifiable {
    let id: UUID
    var name: String
}

// ✅ BON - id stable
List(items) { item in
    Text(item.name)
}

// ❌ MAUVAIS - id instable
List(items) { item in
    Text(item.name)
        .id(UUID()) // Nouveau UUID à chaque render !
}
```

Règle d'or : L'id doit être stable et unique

- Stable = ne change pas entre renders
- Unique = différent pour chaque item

Pattern : Extracting Subviews

```
// ✗ MAUVAIS - Vue monolithique
struct HeavyListView: View {
    @State private var items: [Item]

    var body: some View {
        List(items) { item in
            HStack {
                AsyncImage(url: item.imageURL)
                    .frame(width: 50, height: 50)
                VStack(alignment: .leading) {
                    Text(item.name).font(.headline)
                    Text(item.description).font(.caption)
                }
                Spacer()
                Button("Details") { /* ... */ }
            }
        }
    }
}
```

Pattern : Extracted Subviews

```
// ✓ BON - Subviews extraites
struct OptimizedListView: View {
    @State private var items: [Item]

    var body: some View {
        List(items) { item in
            ItemRow(item: item)
        }
    }
}

struct ItemRow: View {
    let item: Item

    var body: some View {
        HStack {
            AsyncImage(url: item.imageUrl)
                .frame(width: 50, height: 50)
            VStack(alignment: .leading) {
                Text(item.name).font(.headline)
                Text(item.description).font(.caption)
            }
            Spacer()
            Button("Details") { /* ... */ }
        }
    }
}
```

@ViewBuilder : Custom Containers

```
struct Card<Content: View>: View {  
    let title: String  
    @ViewBuilder let content: () -> Content  
  
    var body: some View {  
        VStack(alignment: .leading) {  
            Text(title)  
                .font(.headline)  
            Divider()  
            content()  
        }  
        .padding()  
        .background(.background)  
        .cornerRadius(12)  
    }  
}
```

```
Card(title: "Profile") {  
    Text("Name: John")  
    Text("Age: 30")  
    Button("Edit") {}  
}
```

Partie 3

Animations & Transitions

Animation Basics

```
struct AnimatedView: View {  
    @State private var isExpanded = false  
  
    var body: some View {  
        VStack {  
            Rectangle()  
                .fill(.blue)  
                .frame(width: isExpanded ? 300 : 100,  
                       height: isExpanded ? 300 : 100)  
            .animation(.spring(response: 0.6), value: isExpanded)  
  
            Button("Toggle") {  
                isExpanded.toggle()  
            }  
        }  
    }  
}
```

Deux approches :

1. **.animation()** modifier
2. **withAnimation { }** closure

Animation Curves

```
// Linear  
.animation(.linear(duration: 1.0))  
  
// Ease in/out  
.animation(.easeInOut(duration: 0.5))  
  
// Spring (physique réaliste)  
.animation(.spring(response: 0.6, dampingFraction: 0.7))  
  
// Custom  
.animation(.timingCurve(0.2, 0.8, 0.2, 1.0, duration: 0.5))  
  
// Interpolating Spring (iOS 17+)  
.animation(.smooth(duration: 0.5))
```

Spring recommandé
pour la plupart des cas

- Feels naturel
- Self-terminating (pas de durée fixe)
- Response = "bounciness",
damping = "friction"

Implicit vs Explicit Animations

```
struct AnimationDemo: View {  
    @State private var scale: CGFloat = 1.0  
  
    var body: some View {  
        VStack {  
            // Implicit - animation attachée à la View  
            Circle()  
                .scaleEffect(scale)  
                .animation(.spring(), value: scale)  
  
            // Explicit - animation dans l'action  
            Button("Animate") {  
                withAnimation(.spring()) {  
                    scale = scale == 1.0 ? 1.5 : 1.0  
                }  
            }  
        }  
    }  
}
```

Différence :

- Implicit : anime automatiquement tout changement de value
- Explicit : contrôle fin de QUAND animer

Transitions

```
struct TransitionDemo: View {  
    @State private var show = false  
  
    var body: some View {  
        VStack {  
            if show {  
                Text("Hello!")  
                    .transition(.scale.combined(with: .opacity))  
            }  
  
            Button("Toggle") {  
                withAnimation {  
                    show.toggle()  
                }  
            }  
        }  
    }  
}
```

Transitions disponibles :

- **.opacity, .scale, .slide, .move(edge:)**
- **.asymmetric(insertion: removal:)**
- Combiner avec **.combined(with:)**

Custom Transitions

```
extension AnyTransition {
    static var slideAndFade: AnyTransition {
        .asymmetric(
            insertion: .move(edge: .trailing)
                .combined(with: .opacity),
            removal: .move(edge: .leading)
                .combined(with: .opacity)
        )
    }
}

// Usage
if show {
    DetailView()
        .transition(.slideAndFade)
}
```

Matched Geometry Effect

```
struct MatchedGeometryDemo: View {
    @State private var showDetail = false
    @Namespace private var animation

    var body: some View {
        if !showDetail {
            Circle()
                .fill(.blue)
                .frame(width: 100, height: 100)
                .matchedGeometryEffect(id: "circle", in: animation)
                .onTapGesture {
                    withAnimation(.spring()) {
                        showDetail = true
                    }
                }
        } else {
            Circle()
                .fill(.blue)
                .frame(width: 300, height: 300)
                .matchedGeometryEffect(id: "circle", in: animation)
                .onTapGesture {
                    withAnimation(.spring()) {
                        showDetail = false
                    }
                }
        }
    }
}
```

Phase Animator (iOS 17+)

```
struct PhaseAnimationDemo: View {  
    var body: some View {  
        PhaseAnimator([false, true]) { phase in  
            Circle()  
                .fill(.blue)  
                .scaleEffect(phase ? 1.5 : 1.0)  
                .opacity(phase ? 0.5 : 1.0)  
        } animation: { phase in  
            .easeInOut(duration: 1.0)  
        }  
    }  
}
```

PhaseAnimator = séquence d'états animés

- Boucle automatiquement à travers les phases
- Différentes animations par phase
- Parfait pour loading, pulsing, etc.

Keyframe Animator (iOS 17+)

```
struct KeyframeDemo: View {
    @State private var trigger = false

    var body: some View {
        Circle()
            .fill(.blue)
            .frame(width: 100, height: 100)
            .keyframeAnimator(
                initialValue: AnimationValues(),
                trigger: trigger
            ) { content, value in
                content
                    .scaleEffect(value.scale)
                    .rotationEffect(value.rotation)
            } keyframes: { _ in
                KeyframeTrack(\.scale) {
                    SpringKeyframe(1.5, duration: 0.3)
                    SpringKeyframe(1.0, duration: 0.3)
                }
                KeyframeTrack(\.rotation) {
                    LinearKeyframe(.degrees(45), duration: 0.3)
                    LinearKeyframe(.degrees(0), duration: 0.3)
                }
            }
    }
}

struct AnimationValues {
    var scale = 1.0
    var rotation = Angle.zero
}
```

Partie 4



Liquid Glass (iOS 26+)

Qu'est-ce que Liquid Glass ?

Un nouveau design system unifié :

- iOS 26, iPadOS 26, macOS Tahoe 26
- watchOS 26, (~ tvOS 26, visionOS 2+)

Principe : "Digital meta-material" qui :

- Bend la lumière dynamiquement (lensing)
- Répond au touch avec élasticité
- S'adapte au contenu sous-jacent
- Crée de la profondeur sans obscurcir

Pas juste un blur : Optical effects réalistes

- Refraction aux edges
- Highlights qui bougent avec le device
- Shadows adaptatives selon le fond

Liquid Glass : Regular vs Clear

Two Variants / Regular (défaut) :

- Adapte automatiquement light/dark
- Fonctionne à toute taille
- Effets visuels complets
- Use case : 90% des cas

⚠ Ne jamais mélanger les deux dans une app

Liquid Glass : Regular vs Clear

Two Variants / Clear :

- Transparence permanente élevée
- Nécessite dimming layer
- Pour contenus media-rich
- Use case : Photos, vidéos

⚠ Ne jamais mélanger les deux dans une app

Regular (défaut) :

- Adapte automatiquement light/dark
- Fonctionne à toute taille
- Effets visuels complets
- Use case : 90% des cas

Two Variants / Clear :

- Transparence permanente élevée
- Nécessite dimming layer
- Pour contenus media-rich
- Use case : Photos, vidéos

glassEffect() Modifier

```
// Basic
Text("Hello, Glass!")
    .padding()
    .glassEffect()

// Avec variante
.glassEffect(.regular)
.glassEffect(.clear)

// Avec shape
.glassEffect(.regular, in: .capsule)
.glassEffect(.regular, in: .rect(cornerRadius: 20))

// Avec tint
.glassEffect(.regular.tint(.blue))

// Avec interactivité
.glassEffect(.regular.tint(.blue).interactive())
```

Interactive = répond au touch avec flexing

GlassEffectContainer

```
import SwiftUI

GlassEffectContainer(spacing: 40.0) {
    HStack(spacing: 40.0) {
        Image(systemName: "pencil")
            .frame(width: 80, height: 80)
            .glassEffect()

        Image(systemName: "eraser")
            .frame(width: 80, height: 80)
            .glassEffect()

        Image(systemName: "paintbrush")
            .frame(width: 80, height: 80)
            .glassEffect()
    }
}
```

Pourquoi ? Glass ne peut pas échantillonner glass

- Container groupe les éléments
- Partage les ressources de sampling
- Permet morphing entre éléments

Glass Morphing

```
struct MorphingDemo: View {
    @State private var showDetail = false
    @Namespace private var namespace

    var body: some View {
        if !showDetail {
            Button("Show More") {
                withAnimation(.smooth) {
                    showDetail = true
                }
            }
            .glassEffect()
            .glassEffectID("button", in: namespace)
        } else {
            VStack {
                Text("Details")
                Button("Hide") {
                    withAnimation(.smooth) {
                        showDetail = false
                    }
                }
            }
            .padding()
            .glassEffect()
            .glassEffectID("button", in: namespace)
        }
    }
}
```

Button Styles Glass

```
// Glass button (secondary)
Button("Action") {
    // action
}
.buttonStyle(.glass)

// Glass Prominent (primary)
Button("Primary") {
    // action
}
.buttonStyle(.glassProminent)

// Avec tint
Button("Colored") {
    // action
}
.buttonStyle(.glass)
.tint(.blue)
```

Nouveaux styles :

- .glass = translucent, pour actions secondaires
- .glassProminent = opaque, pour primary actions

NavigationSplitView avec Glass

```
NavigationSplitView {  
    List(items) { item in  
        NavigationLink(item.name, value: item)  
    }  
    .backgroundExtensionEffect()  
} detail: {  
    DetailView()  
}
```

Nouveautés iOS 26 :

- Sidebar flotte au-dessus du content (glass)
- `.backgroundExtensionEffect()` étend le contenu sous la sidebar
- Content se mirror et blur en dehors du safe area

TabView avec Glass²

```
TabView(selection: $selectedTab) {  
    HomeView()  
    .tabItem {  
        Label("Home", systemImage: "house")  
    }  
  
    SearchView()  
    .tabItem {  
        Label("Search", systemImage: "magnifyingglass")  
    }  
}  
.tabBarMinimizeBehavior(.onScrollDown)  
.tabViewBottomAccessory {  
    NowPlayingView()  
}
```

Nouveautés :

- Tab bar flotte (glass)
- **.tabBarMinimizeBehavior()**
cache sur scroll
- **.tabViewBottomAccessory()**
pour contenu persistant

² WWDC25 "Build a SwiftUI app with the new design" montre tout ça en action avec l'app Landmarks.

Sheets avec Glass

```
struct SheetDemo: View {
    @State private var showSheet = false

    var body: some View {
        Button("Show Sheet") {
            showSheet = true
        }
        .sheet(isPresented: $showSheet) {
            VStack {
                Text("Sheet Content")
                Spacer()
            }
            .presentationDetents([.medium, .large])
            // ⚠ Pas de .presentationBackground()
            // Le glass s'applique automatiquement
        }
    }
}
```

Changement : Sheets ont glass automatiquement

- Retirez `.presentationBackground()`
- Edges s'incurvent aux petits detents
- Morphing depuis button possible

UIKit : UIGlassEffect

```
import UIKit

let effectView = UIVisualEffectView()
let glassEffect = UIGlassEffect()

// Configuration
glassEffect.isInteractive = true
glassEffect.tintColor = .systemBlue

// Application
UIView.animate(withDuration: 0.3) {
    effectView.effect = glassEffect
}

// Variants
let regularGlass = UIGlassEffect()
let clearGlass = UIGlassEffect.clear()
```

Support UIKit complet :

- **UIGlassEffect** pour l'effet
- **UIVisualEffectView** comme container
- Animation avec **UIView.animate**

Design Principles : Où Utiliser Glass

Règle : Glass sépare navigation du contenu

Navigation Layer uniquement :

- Toolbars
- Tab bars
- Sidebars
- Sheets
- Menus
- Floating buttons

Pas sur Content Layer :

- Table views / Lists
- Cards de contenu
- Images
- Text areas

Design Principles : Tinting

```
// ✓ BON - Tint avec signification
Button("Checkout") {
    // checkout action
}
.buttonStyle(.glassProminent)
.tint(.red) // Rouge = attention, action importante

// ✗ MAUVAIS - Tint décoratif partout
VStack {
    Button("Action 1").tint(.blue)
    Button("Action 2").tint(.green)
    Button("Action 3").tint(.purple)
    Button("Action 4").tint(.orange)
}
```

Tinting rules :

- Avec signification, pas décoration
- Un ou deux tints max par écran
- Tints s'adaptent au background (light/dark)

Performance : GPU Rendering

Liquid Glass utilise
Metal shaders :

- Gaussian blur GPU-accéléré
- Fragment shaders en parallèle
- 60-120 fps maintenable

Device requirements :

- iOS 26 nécessite A13+ (iPhone 11+)
- A17 Pro / A18 : performance optimale
- iPhone 12-13 : peut avoir lag

Optimizations :

- Max 16 shapes par GlassEffectContainer
- Glass thickness < 20px recommandé
- Glass statique > glass mobile

Performance : Battery Impact

Tests réels :

- 8-12% CPU overhead vs interfaces statiques
- Impact batterie minimal sur A17+ / A18
- Dark mode + glass = meilleur sur OLED

iOS 26.1 Toggle :

- Settings → Display & Brightness → Liquid Glass
- Clear (défaut, plus transparent)
- Tinted (plus opaque, meilleur contraste)

Profiling tools :

- Metal System Trace (Instruments)
- Metal Debugger (Xcode)
- Target < 8.3ms par frame (120fps)

Accessibility : Adaptation Automatique

```
@Environment(\.accessibilityReduceTransparency)  
var reduceTransparency  
  
if reduceTransparency {  
    // Alternative UI  
}
```

Settings respectés automatiquement :

- Reduce Transparency → glass plus opaque
- Increase Contrast → black/white avec borders
- Reduce Motion → moins d'élasticité

Developer responsibilities :

- Maintenir 4.5:1 contrast ratio (WCAG)
- Utiliser SF Symbols (VoiceOver OK)
- Labels texte en plus des icônes

Migration : Automatic Adoption³

Apps SwiftUI iOS 18+ :

1. Recompile avec Xcode 16 (iOS 26 SDK)
2. C'est tout ! Glass appliqué automatiquement

Cleanup nécessaire :

```
swift
// Retirer ces modifiers obsolètes
.toolbarBackground(.visible, for: .navigationBar)
.presentationBackground(.ultraThinMaterial)
.background(.regularMaterial)
```

Opt-out temporaire (⚠⚠⚠⚠ expire iOS 27

⚠⚠⚠⚠) :

```
xml
<!-- Info.plist -->
<key>UIDesignRequiresCompatibility</key>
<true/>
```

³ WWDC25 Session 208 "Showcase: Apps integrating Liquid Glass" avec retours d'expérience real-world de plusieurs grandes apps.

Migration : Timeline Réelle

Safari team (Apple) :

- Estimation : 1.5 semaines
- Réalité : 1.5 jours
- Raison : Utilisaient déjà composants natifs

American Airlines :

- Migration rapide grâce à SwiftUI natif
- Pas de UIKit custom à migrer

Recommendation :

- Migrez progressivement
- Testez sur devices réels (pas que simulateur)
- Utilisez les bêta iOS !

Custom Glass Effects

```
struct CustomGlassCard<Content: View>: View {
    @ViewBuilder let content: Content

    var body: some View {
        content
            .padding()
            .frame(maxWidth: .infinity)
            .glassEffect(.regular.tint(.blue).interactive())
            .cornerRadius(20)
            .shadow(color: .black.opacity(0.1),
                    radius: 10, y: 5)
    }
}

// Usage
CustomGlassCard {
    VStack(alignment: .leading) {
        Text("Title").font(.headline)
        Text("Content")
    }
}
```

Liquid Glass : Best Practices

DO :

- Utiliser pour navigation uniquement
- GlassEffectContainer pour éléments proches
- Tint avec signification
- Tester sur devices réels
- Respecter Regular vs Clear

DON'T :

- Glass sur contenu
- Mélanger Regular et Clear
- Trop de tints
- Glass sur glass
- Oublier accessibility

Ressources Officielles

Documentation Apple :

- Liquid Glass Overview
- Adopting Liquid Glass
- Applying Liquid Glass to custom views
- Human Interface Guidelines

WWDC 2025 Sessions :

- Session 219 : Meet Liquid Glass
- Session 323 : Build SwiftUI app with new design
- Session 284 : Build UIKit app with new design
- Session 356 : Get to know the new design system
- Session 208 : Apps integrating Liquid Glass

Sample Code :

- Landmarks project (updated for iOS 26)

Récapitulatif Jour 2

Ce qu'on a couvert :

Cycle de Vie : @State, @Binding, @Observable, @Environment

Performance : Body optimization, LazyVStack, Subviews

Animations : Springs, Transitions, Matched Geometry

Liquid Glass : glassEffect(), GlassEffectContainer, Regular vs Clear

Prochaine étape :

14h-16h00 : TP

16h-16h30 : Fireside Chat with...

16h30-18h00 : TP

Questions ?

Liens Rapides

SwiftUI : developer.apple.com/swiftui

Performance : WWDC23 Session 10160

Animations : WWDC23 "Animate with springs"

Liquid Glass : WWDC25 Session 219

Fireside Chat

Dans 5 minutes

Guest : 

Sujet : All things 

Préparez vos questions !

Apple Intelligence & IA

iOS 18 & iOS 26

Jour 3 - Atelier Swift Avancé

Agenda du Jour (3h-3h30)

- Partie 1 : Architecture Apple Intelligence
- Partie 2 : Foundation Models & On-Device AI
- Partie 3 : App Intents & Siri Integration
- Pause (15min) ☕
- Partie 4 : Writing Tools & Text APIs
- Partie 5 : Outils IA pour Développeurs
- Conclusion & Questions

Partie I



Architecture Apple Intelligence

Qu'est-ce qu'Apple Intelligence ?

Définition technique :

- Suite de modèles d'IA générative on-device (AFM iOS 26+)
- Intégration système profonde (iOS 18+, iOS 26)
- Architecture multi-niveaux pour confidentialité

Pas juste un feature :

- C'est une capability du système
- Comme UIKit, AVFoundation, Core ML
- Disponible pour toutes les apps via APIs

Changement de paradigme :

- L'app devient "proactive" et "conversationnelle"
- L'utilisateur peut parler naturellement à l'app
- L'app suggère des actions contextuelles

Concepts Fondamentaux : LLM & RLM

LLM (Large Language Model) :

- Modèle entraîné sur texte massif
- Génère du texte de manière probabiliste
- Exemples : GPT-4, Claude, Llama

RLM (Reasoning Language Model) :

- LLM optimisé pour le raisonnement
- Meilleur sur tâches logiques/complexes
- Exemples : GPT-4o, Claude Sonnet 4.5

Paramètres :

- Poids du modèle (ex: 3B, 7B, 70B)
- Plus de paramètres = plus capable
- Mais aussi plus lent et gourmand
- Apple : ~3B paramètres (2-bit quantized)

Tokens :

- Unité de traitement (≈ 0.75 mot)
- "Bonjour" = 1 token, "Hello world" = 2 tokens
- Vitesse : tokens/sec (ex: 30 t/s iPhone 15 Pro)

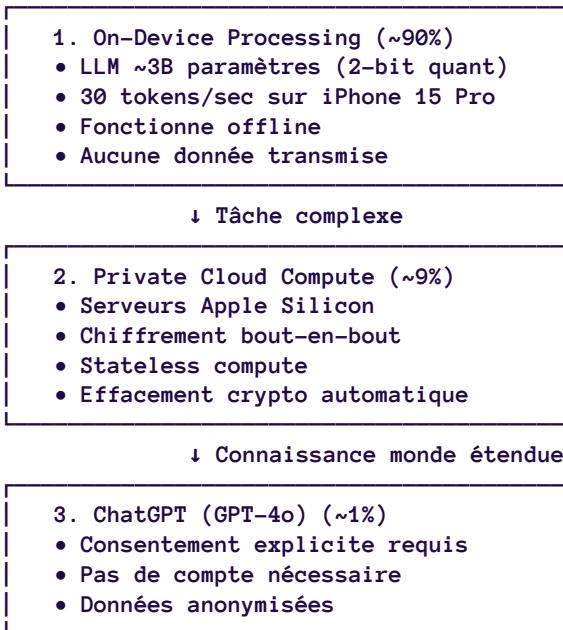
Context Window :

- Nombre max de tokens en mémoire
- Exemples : 8K, 32K, 128K, 200K
- Plus grand = plus de contexte
- Apple on-device : ~4.096 tokens

Quantization :

- Réduction précision (32-bit \rightarrow 2-bit)
- Divise taille par 16x
- Légère perte qualité, gain énorme performance

Apple Foundation Model : Architecture à 3 Niveaux¹



¹ <https://machinelearning.apple.com/research/introducing-apple-foundation-models>

On-Device : Capacités & Limites

Excellences du modèle on-device :

- Résumé de texte (jusqu'à ~4000 tokens)
- Extraction d'informations structurées
- Classification et tagging de contenu
- Génération de texte court/moyen
- Traduction entre langues courantes

Limitations connues :

- Raisonnement mathématique complexe
- Génération de code sophistiqué
- Connaissance factuelle étendue
- Contextes très longs (>8K tokens)
- Tâches multimodales complexes

Pour développeurs :

- Testez toujours on-device d'abord
- Escaladez à PCC uniquement si nécessaire

Private Cloud Compute : Garanties²

Architecture de sécurité unique :

1. Stateless Computation

- Aucune donnée stockée, jamais
- Mémoire effacée cryptographiquement après traitement

2. Enforceable Guarantees

- Garanties techniques, pas juste policies
- Vérifiables indépendamment

1. No Privileged Runtime Access

- Même les employés Apple ne peuvent pas accéder
- Pas de backdoor possible

2. Non-Targetability

- Impossible de cibler un utilisateur spécifique
- Routing aléatoire entre serveurs

1. Verifiable Transparency

- Code source inspectable par chercheurs
- Bug bounty jusqu'à \$1M

² Apple extends its privacy leadership...

ChatGPT Integration

Déclenchement automatique :

- Requêtes nécessitant connaissance monde étendue
- Système détecte automatiquement
- Popup de consentement utilisateur

Contrôle utilisateur :

Settings → Apple Intelligence → ChatGPT Extension

- Peut être désactivé globalement
- Consentement à chaque utilisation
- Pas de compte ChatGPT requis

Pour les dév :

- Pas d'API directe ChatGPT dans vos apps
- Utilisez Foundation Models framework
- Le système route automatiquement si nécessaire

Partie 2

Foundation Models : Vue d'ensemble

Disponibilité :

- iOS 26, iPadOS 26, macOS Tahoe 26
- visionOS 26

Capacités principales :

- Génération de texte libre
- Génération structurée (@Generable)
- Streaming en temps réel
- Tool calling (fonctions externes)

Requirements :

- iPhone 15 Pro / 16 / 16 Pro
- iPad M1+, Mac Apple Silicon
- 8GB RAM minimum

Gratuit : Aucun coût API, tout on-device

Génération de Texte Basique

```
import FoundationModels

func generateText(prompt: String) async throws -> String {
    let session = LanguageModelSession()
    let response = try await session.respond(to: prompt)
    return response.content
}

// Utilisation
let explanation = try await generateText(
    prompt: "Explique les optionals Swift en 3 phrases simples"
)
print(explanation)
```

Simple mais puissant :

- Génération synchrone ou async
- Contexte automatiquement géré
- Température et max_tokens configurables

Génération Structurée avec @Generable³

```
import FoundationModels

@Generable
struct MeetingSummary: Equatable {
    @Guide(description: "Sujet principal de la réunion")
    let subject: String

    @Guide(description: "Participants présents")
    let participants: [String]

    @Guide(description: "Points principaux discutés")
    let keyPoints: [String]

    @Guide(description: "Décisions prises")
    let decisions: [String]

    @Guide(description: "Actions à suivre avec responsables")
    let actionItems: [String]

    @Guide(description: "Date et sujet de la prochaine réunion si mentionnée")
    let nextMeeting: String?
}
```

³ Deep dive into the Foundation Models framework: WWDC 2025 Session 301

Génération Structurée avec @Generable

```
func analyzeMeetingNotes(notes: String) async throws -> MeetingSummary {  
    let session = LanguageModelSession()  
    let response = try await session.respond(  
        to: "Analyse ces notes de réunion: \(notes)",  
        generating: MeetingSummary.self  
    )  
    return response.content  
}
```

@Generable : Types Supportés

Value types simples :

```
@Generable
struct SimpleData {
    let name: String
    let age: Int
    let score: Double
    let isActive: Bool
}
```

Collections :

```
@Generable
struct ComplexData {
    let items: [String]
    let scores: [Int]
    let metadata: [String: String]
}
```

Nested structures :

```
@Generable
struct Author {
    let name: String
    let bio: String
}

@Generable
struct Book {
    let title: String
    let author: Author // Nested!
    let chapters: [String]
}
```

Streaming pour UX Réactive

```
import FoundationModels
import SwiftUI

struct StreamingView: View {
    @State private var partialSummary: ArticleSummary?
    @State private var isGenerating = false

    func generateSummary(text: String) async {
        isGenerating = true
        defer { isGenerating = false }

        let session = LanguageModelSession()
        let stream = session.streamResponse(
            to: "Résume cet article: \(text)",
            generating: ArticleSummary.self
        )

        for try await partial in stream {
            // partial contient des propriétés progressivement remplies
            partialSummary = partial
        }
    }
}
```

```
var body: some View {
    VStack {
        if let summary = partialSummary {
            Text(summary.title ?? "Génération...")
            Text(summary.mainPoints?.joined(separator: "\n") ?? "")
        }
    }
}
```

Tool Calling : Étendre les Capacités

```
import FoundationModels

struct WeatherTool: Tool {
    let name = "get_weather"
    let description = "Obtient la météo actuelle pour une ville"

    @Generable
    struct Arguments {
        @Guide(description: "Nom de la ville")
        let city: String
    }

    func call(arguments: Arguments) async throws -> ToolOutput {
        // Appel API météo réelle
        let weather = try await WeatherAPI.fetch(city: arguments.city)
        return ToolOutput("Météo à \(arguments.city): \(weather.temp)°C, \(weather.condition)")
    }
}

func chatWithTools(message: String) async throws -> String {
    let session = LanguageModelSession(tools: [WeatherTool()])
    let response = try await session.respond(to: message)
    return response.content
}

// Utilisation
let answer = try await chatWithTools(message: "Quel temps fait-il à Paris ?")
// Le modèle détecte qu'il doit appeler WeatherTool, le fait automatiquement
// Puis génère une réponse avec les données réelles
```

Vérifiez la disponibilité d'AFM

```
import FoundationModels

func checkAvailability() {
    switch SystemLanguageModel.default.availability {
        case .available:
            print("✅ Foundation Models disponible")

        case .unavailable(let reason):
            switch reason {
                case .appleIntelligenceNotEnabled:
                    print("⚠️ Apple Intelligence désactivé dans Settings")
                    // Montrer UI pour guider l'utilisateur

                case .deviceNotEligible:
                    print("❌ Appareil non compatible (iPhone 15 Pro+ requis)")
                    // Fallback vers alternative

                case .modelNotReady:
                    print("⏳ Modèle pas encore téléchargé")
                    // Afficher message "En cours de téléchargement"

                @unknown default:
                    print("❓ Raison inconnue")
            }
    }
}
```

Partie 3

App Intents & Siri Integration

App Intents : Pourquoi ?

Avant App Intents :

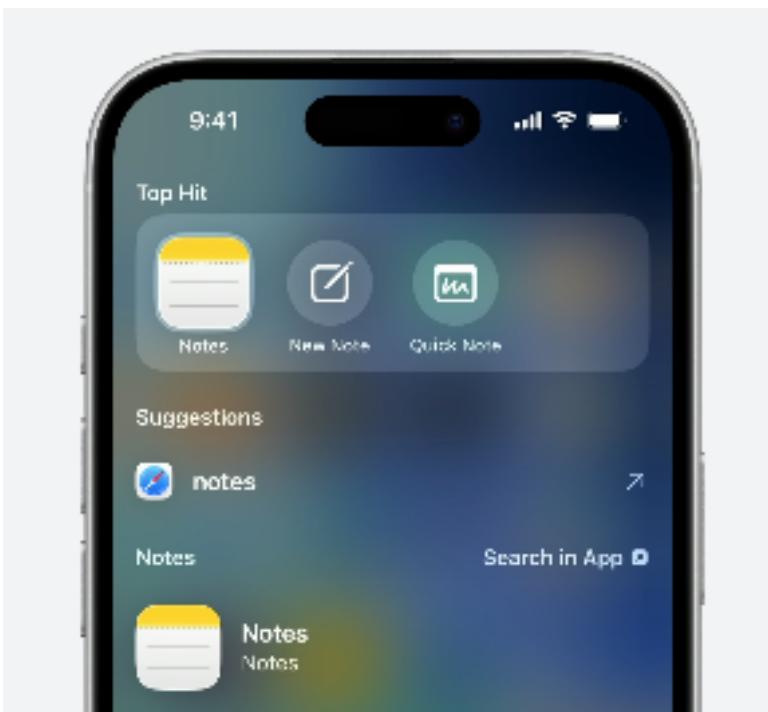
Utilisateur → Ouvre l'app → Navigation → Action

Avec App Intents :

Utilisateur → "Crée une note dans MyApp" → Fait !

Surfaces d'exposition :

-  Siri (vocal et texte)
-  Spotlight Search
-  Shortcuts app
-  Control Center
-  Action Button
-  Proactive
-  Apple Intelligence



Structure d'un App Intent

```
import AppIntents

struct CreateTaskIntent: AppIntent {
    // 1. Métdonnées
    static var title: LocalizedStringResource = "Créer une tâche"
    static var description = IntentDescription("Crée une nouvelle tâche")
    static var openAppWhenRun: Bool = false // Ne lance pas l'app

    // 2. Paramètres
    @Parameter(title: "Titre de la tâche")
    var taskTitle: String

    @Parameter(title: "Liste", default: TaskList.inbox)
    var list: TaskList

    @Parameter(title: "Date limite")
    var dueDate: Date?

    // 3. Phrase suggérée pour Siri
    static var parameterSummary: some ParameterSummary {
        Summary("Créer \$(taskTitle) dans \$(list)")
    }

    // 4. Exécution
    func perform() async throws -> some IntentResult {
        try await TaskManager.shared.createTask(
            title: taskTitle,
            list: list,
            dueDate: dueDate
        )
        return .result(dialog: "Tâche créée !")
    }
}
```

App Entity : Objets Métier

```
struct TaskListQuery: EntityQuery {
    func entities(for identifiers: [UUID]) async throws -> [TaskList] {
        TaskDataManager.shared.lists
            .filter { identifiers.contains($0.id) }
            .map { TaskList(id: $0.id, name: $0.name, color: $0.color) }
    }

    func suggestedEntities() async throws -> [TaskList] {
        TaskDataManager.shared.lists
            .map { TaskList(id: $0.id, name: $0.name, color: $0.color) }
    }
}
```

App Shortcuts Provider

```
import AppIntents

struct MyAppShortcutsProvider: AppShortcutsProvider {
    @AppShortcutsBuilder
    static var appShortcuts: [AppShortcut] {
        AppShortcut(
            intent: CreateTaskIntent(),
            phrases: [
                "Créer une tâche dans \(.applicationName)",
                "Ajouter à faire dans \(.applicationName)",
                "Create task in \(.applicationName)",
                "Add todo in \(.applicationName)"
            ],
            shortTitle: "Nouvelle tâche",
            systemImageName: "plus.circle"
        )
        AppShortcut(
            intent: ShowTodayTasksIntent(),
            phrases: [
                "Mes tâches du jour dans \(.applicationName)",
                "What's on my list in \(.applicationName)"
            ],
            shortTitle: "Tâches aujourd'hui",
            systemImageName: "calendar"
        )
    }
}
```

Activation :

```
swift
// Dans votre app delegate ou scene delegate
MyAppShortcutsProvider.updateAppShortcutParameters()
```

Assistant Schemas (iOS 18+)

```
import AppIntents

@AssistantIntent(schema: .system.search)
struct SearchPhotosIntent: AppIntent {
    static var title: LocalizedStringResource = "Rechercher photos"

    @Parameter(title: "Critères de recherche")
    var criteria: String

    func perform() async throws -> some IntentResult {
        let results = await PhotoManager.shared.search(query: criteria)

        // Retourner résultats pour Apple Intelligence
        return .result(
            value: results,
            dialog: "J'ai trouvé \(results.count) photos"
        )
    }
}
```

Assistant Schemas disponibles :

- **.system.search** - Recherche de contenu
- **.system.open** - Ouvrir un élément
- **.system.settings** - Paramètres app
- **.photos.search** - Recherche photos spécifique
- **.messages.send** - Envoyer messages

Partie 4

Writing Tools & Text APIs

Writing Tools : Intégration Automatique

```
import SwiftUI

struct EditorView: View {
    @State private var text = ""

    var body: some View {
        VStack {
            // ✅ Writing Tools activé par défaut
            TextEditor(text: $text)
                .frame(height: 200)

            // ⚠️ Désactiver si nécessaire
            TextField("Sans outils", text: $text)
                .writingToolsBehavior(.disabled)

            // 📁 Mode limité (copier/partager uniquement)
            TextEditor(text: $text)
                .writingToolsBehavior(.limited)

            // 🖊️ Comportement complet (default)
            TextEditor(text: $text)
                .writingToolsBehavior(.complete)
        }
    }
}
```

Actions disponibles pour l'utilisateur :

- Rewrite (Professional / Concise / Friendly)
- Proofread (Grammaire & orthographe)
- Summarize (Key Points / Paragraphe)

Writing Tools : Delegates UIKit

```
import UIKit

class MyTextViewController: UIViewController, UITextViewDelegate {
    let textView = UITextView()

    override func viewDidLoad() {
        super.viewDidLoad()
        textView.delegate = self
    }

    // Appelé quand Writing Tools commence
    func textViewWritingToolsWillBegin(_ textView: UITextView) {
        print("⚠️ Writing Tools commence")

        // Désactiver sync cloud temporairement
        CloudSyncManager.shared.pauseSync()

        // Désactiver auto-save
        autoSaveTimer?.invalidate()
    }

    // Appelé quand Writing Tools termine
    func textViewWritingToolsDidEnd(_ textView: UITextView) {
        print("✅ Writing Tools terminé")

        // Réactiver sync
        CloudSyncManager.shared.resumeSync()

        // Réactiver auto-save
        scheduleAutoSave()
    }

    // Optionnel : Ignorer certains ranges
    func textView(
        _ textView: UITextView,
        writingToolsIgnoredRangesIn range: NSRange
    ) -> [NSValue] {
        // Ignorer les code blocks, markdown syntax, etc.
        return findCodeBlockRanges(in: textView.text)
    }
}
```

Natural Language Framework

```
import NaturalLanguage

// 1. Extraction d'entités nommées
func extractEntities(from text: String) -> [(text: String, type: NLTag)] {
    var entities: [(String, NLTag)] = []
    let tagger = NLTagger(tagSchemes: [.nameType])
    tagger.string = text

    let tags: [NLTag] = [.personalName, .placeName, .organizationName]

    tagger.enumerateTags(
        in: text.startIndex..
```

```
// Exemple
let text = "Clément SAUVAGE remplace Tim COOK à la tête d'Apple à Cupertino pour Apple"
let entities = extractEntities(from: text)
// [("Clément SAUVAGE", .personalName), ("Tim COOK", .personalName), ("Cupertino", .placeName), ("Apple", .organizationName)]
```

Résumé Automatique

```
import NaturalLanguage

func summarize(text: String, sentenceCount: Int = 3) -> String {
    let tagger = NLTagger(tagSchemes: [.tokenType])
    tagger.string = text

    var sentences: [String] = []
    tagger.enumerateTags(
        in: text.startIndex..
```

Partie 5



Outils IA pour Développeurs

Xcode Predictive Code Completion

Status :

- Disponible : Xcode 16.0+ (Predictive Completion)
- Swift Assist : Annoncé WWDC24, disponible (réellement) dans Xcode 26

Activation :

Xcode → Settings → Components
→ Predictive Code Completion Model
→ Download & Enable

Caractéristiques :

- Modèle ML ~2GB on-device
- Apple Silicon Mac avec 16GB RAM minimum
- Completion contextuelle basée sur projet
- Génération depuis commentaires

```
// Create a function to fetch user data from API
// Le model génère automatiquement :
func fetchData() async throws -> User {
    let url = URL(string: "https://api.example.com/user")!
    let (data, _) = try await URLSession.shared.data(from: url)
    return try JSONDecoder().decode(User.self, from: data)
}
```

ChatGPT pour Développement Swift

Cas d'usage excellents :

1. Génération de code boilerplate

Prompt: "Create a SwiftData model for a Book with title, author, publicationDate, and rating. Include sample data."

2. Décodage JSON vers Codable

Prompt: "Convert this JSON to Swift Codable struct:
{ "user": { "name": "John", "age": 30, "email": "..." } }

1. Génération de tests unitaires

Prompt: "Generate XCTest unit tests for this ViewModel:
[paste code]"

2. Debugging et explication

Prompt: "Explain why this code crashes: [paste crash log]"
[.column]

3. Optimisation performance

Prompt: "Optimize this SwiftUI View for better performance: [paste code]"

Prompt Engineering pour Swift

Template efficace :

[PERSONA]

Act as a senior iOS developer with 10+ years Swift experience.

[CONTEXT]

I'm building a SwiftUI app using MVVM architecture, SwiftData for persistence, and async/await for networking.

[TASK]

Create a ViewModel for fetching and displaying a list of products from this API endpoint: <https://api.example.com/products>

[CONSTRAINTS]

- Use `@Observable` (iOS 17+)
- Include error handling with custom errors
- Add loading states
- Use Swift 6 strict concurrency

[FORMAT]

Provide complete code with documentation comments.

Claude Code : Coding Agentique

Qu'est-ce que c'est ? :

- Outil de coding agentique terminal-native
- Lancé par Anthropic en mai 2025
- Fenêtre contexte 200K tokens

Capabilities :

- Construire features complètes from scratch
- Debugger automatiquement avec execution
- Lire codebases existantes (context aware)
- Intégration MCP (Google Drive, Jira, etc.)
- Refactoring intelligent multi-fichiers

Installation :

```
brew install claude  
claude login
```

Usage :

```
cd MyiOSProject  
claude "Add SwiftData persistence to this app"
```

LM Studio : LLMs Locaux GUI

Installation :

```
bash  
brew install lm-studio
```

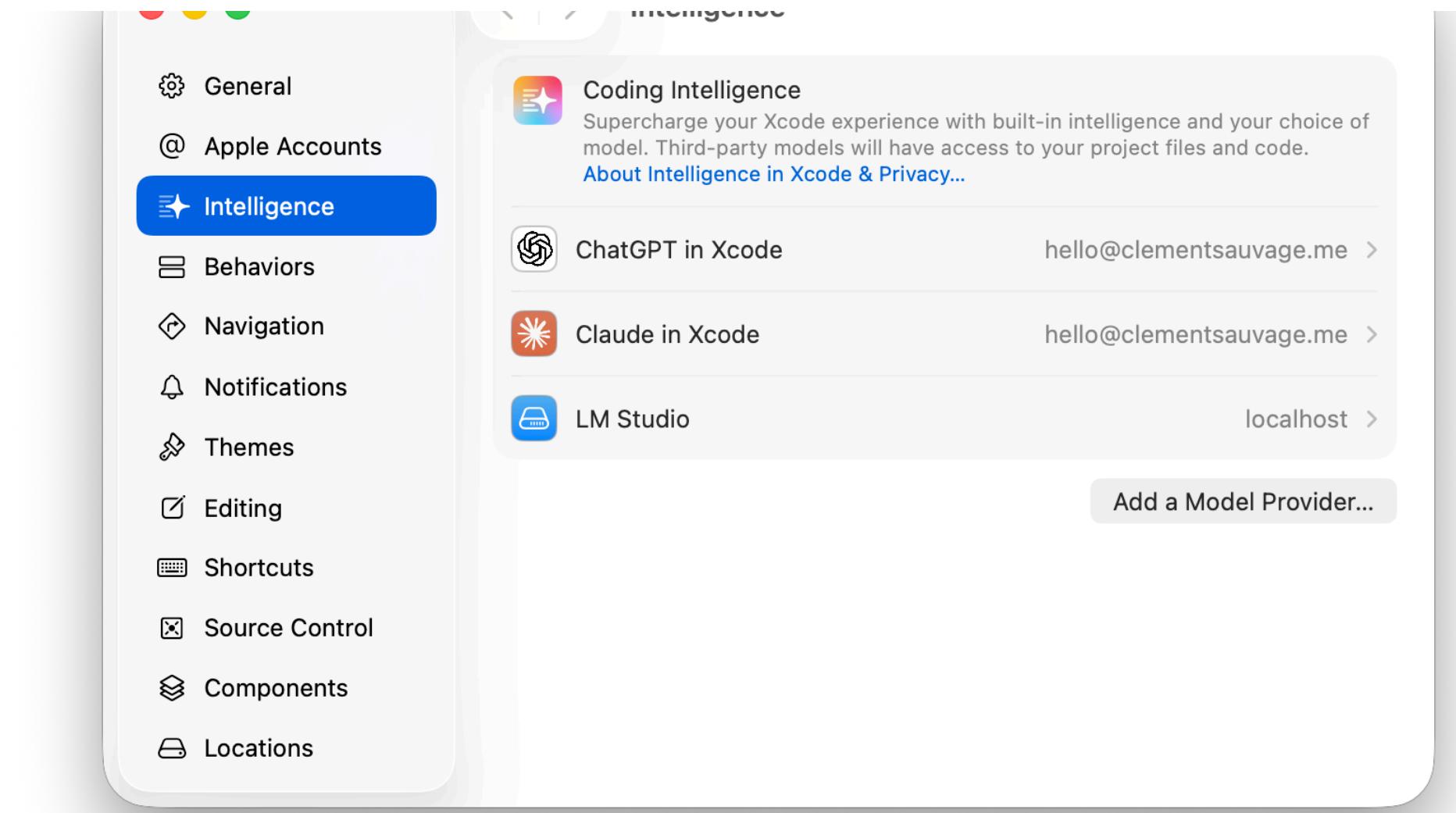
Features :

- Interface graphique intuitive
- Découverte de modèles (HuggingFace)
- Serveur API local (compatible OpenAI)
- Inference GPU-accelerated (Metal)
- Gratuit

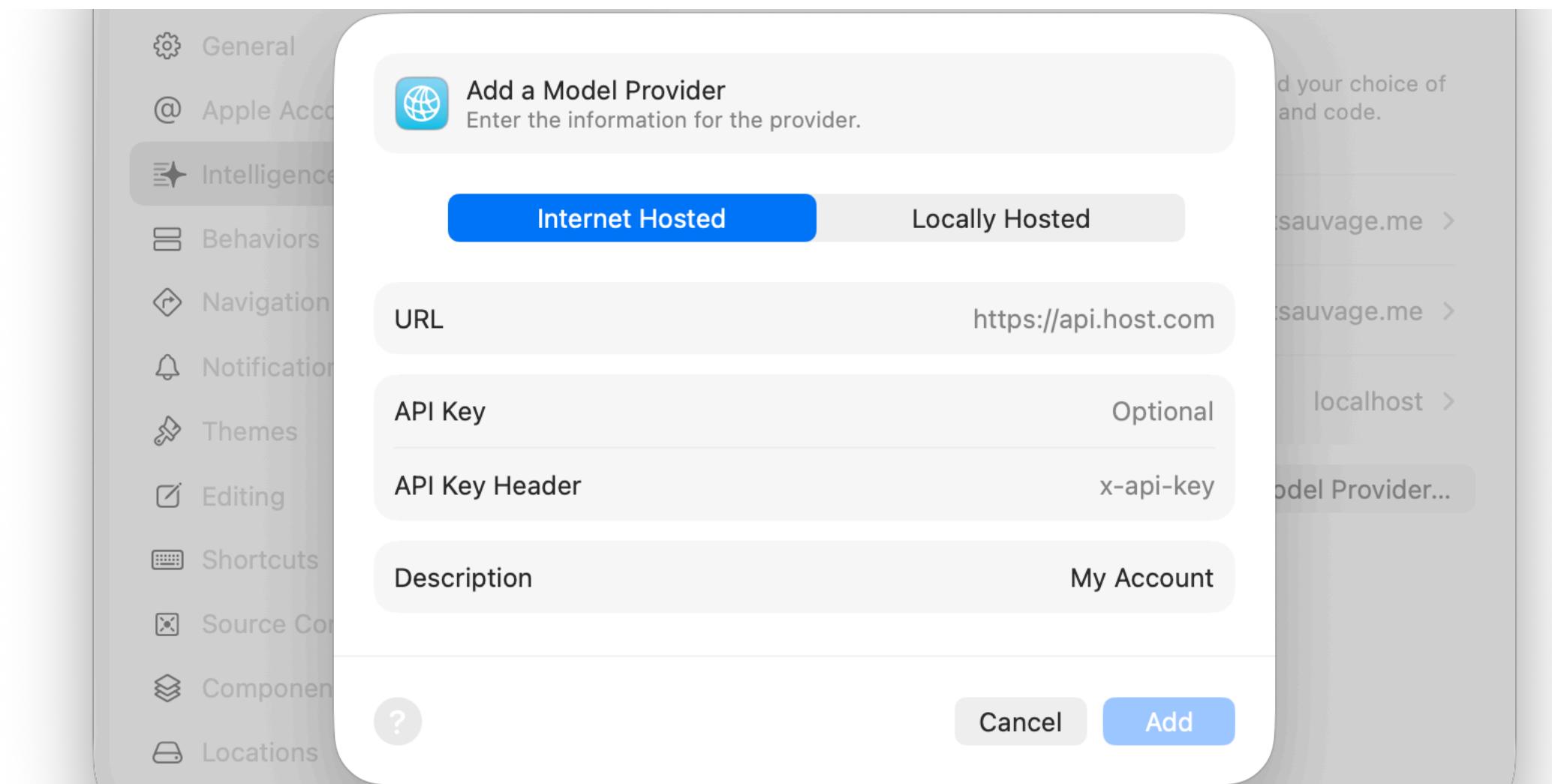
API Usage :

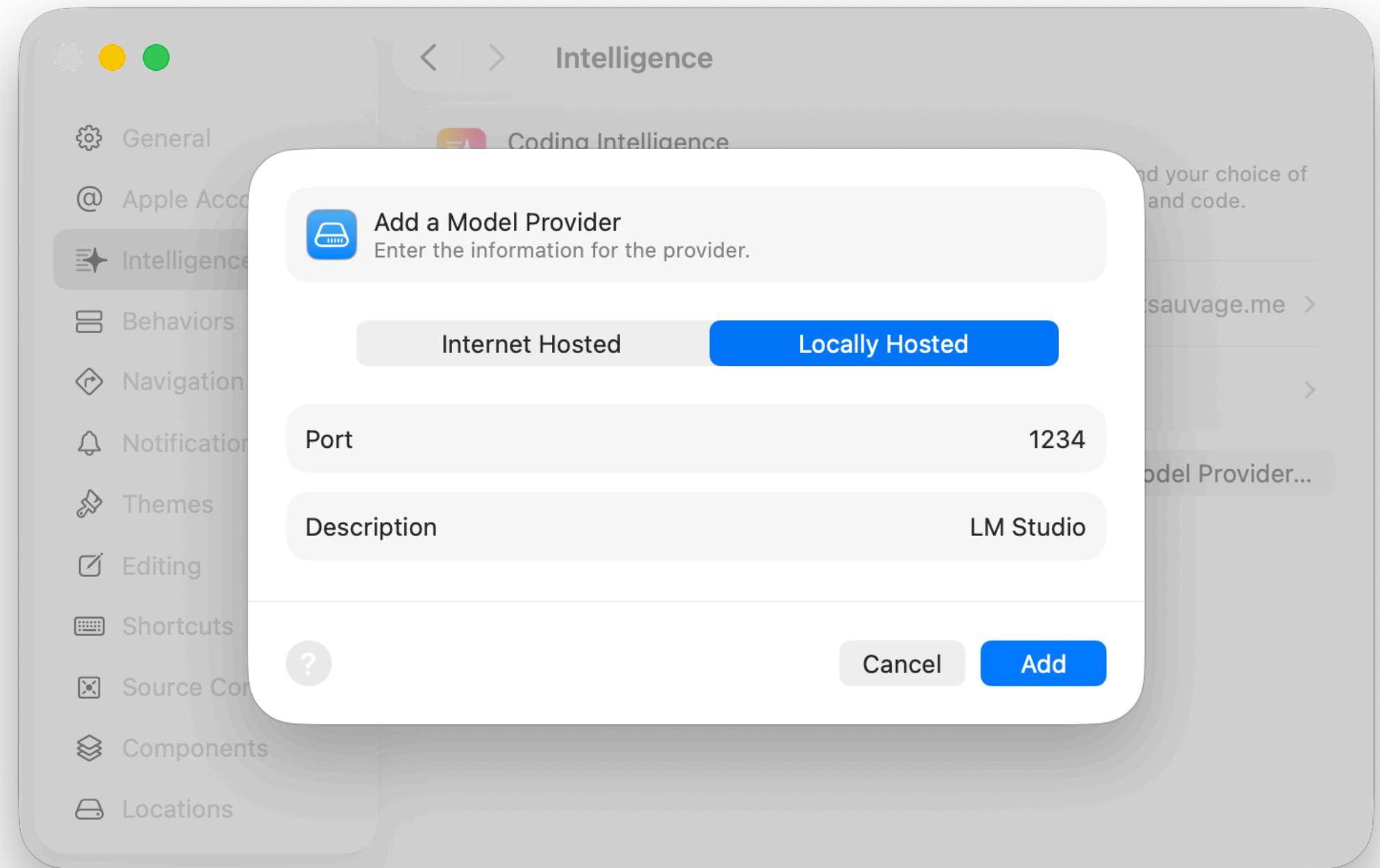
```
bash  
curl http://localhost:1234/v1/chat/completions  
\  
-H "Content-Type: application/json" \  
-d '{"messages": [{"role": "user",  
"content": "Explain Swift actors"}]}'
```

Utilisation dans Xcode



Utilisation dans Xcode





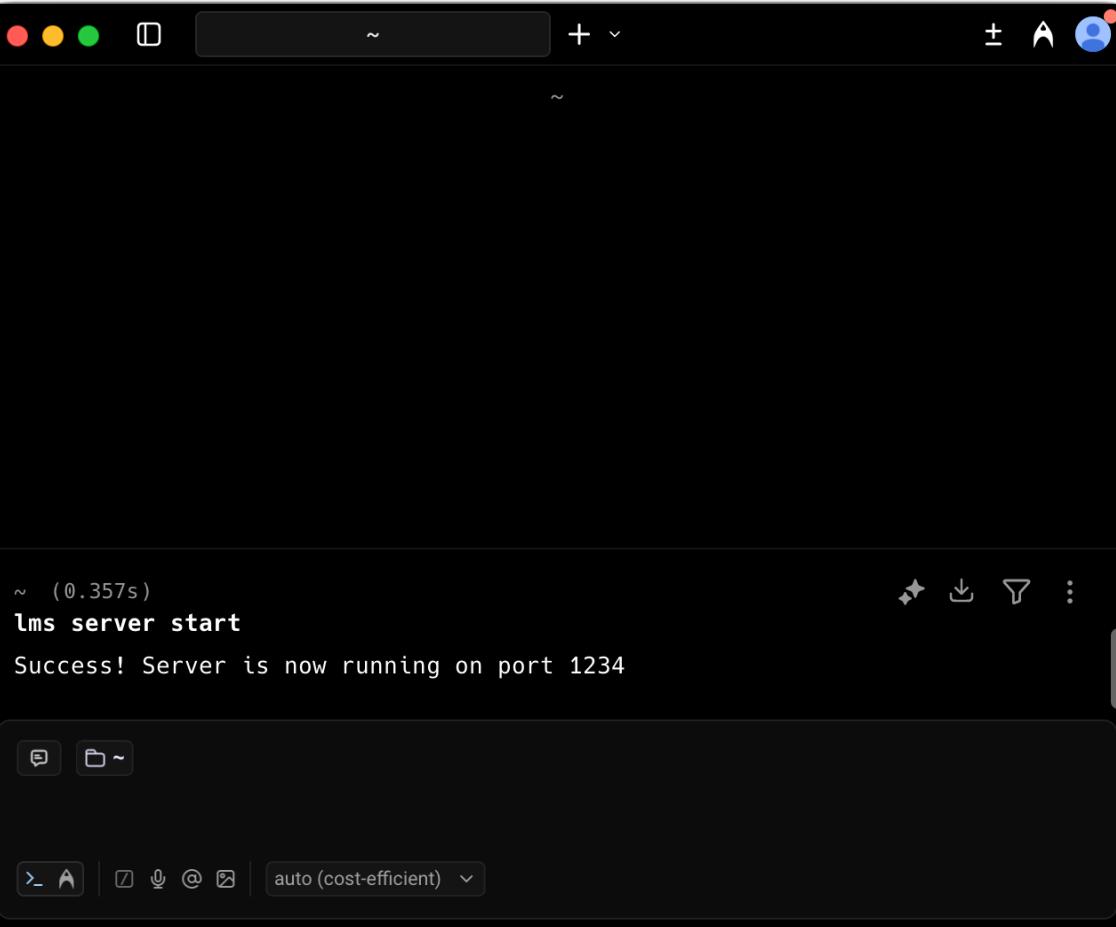


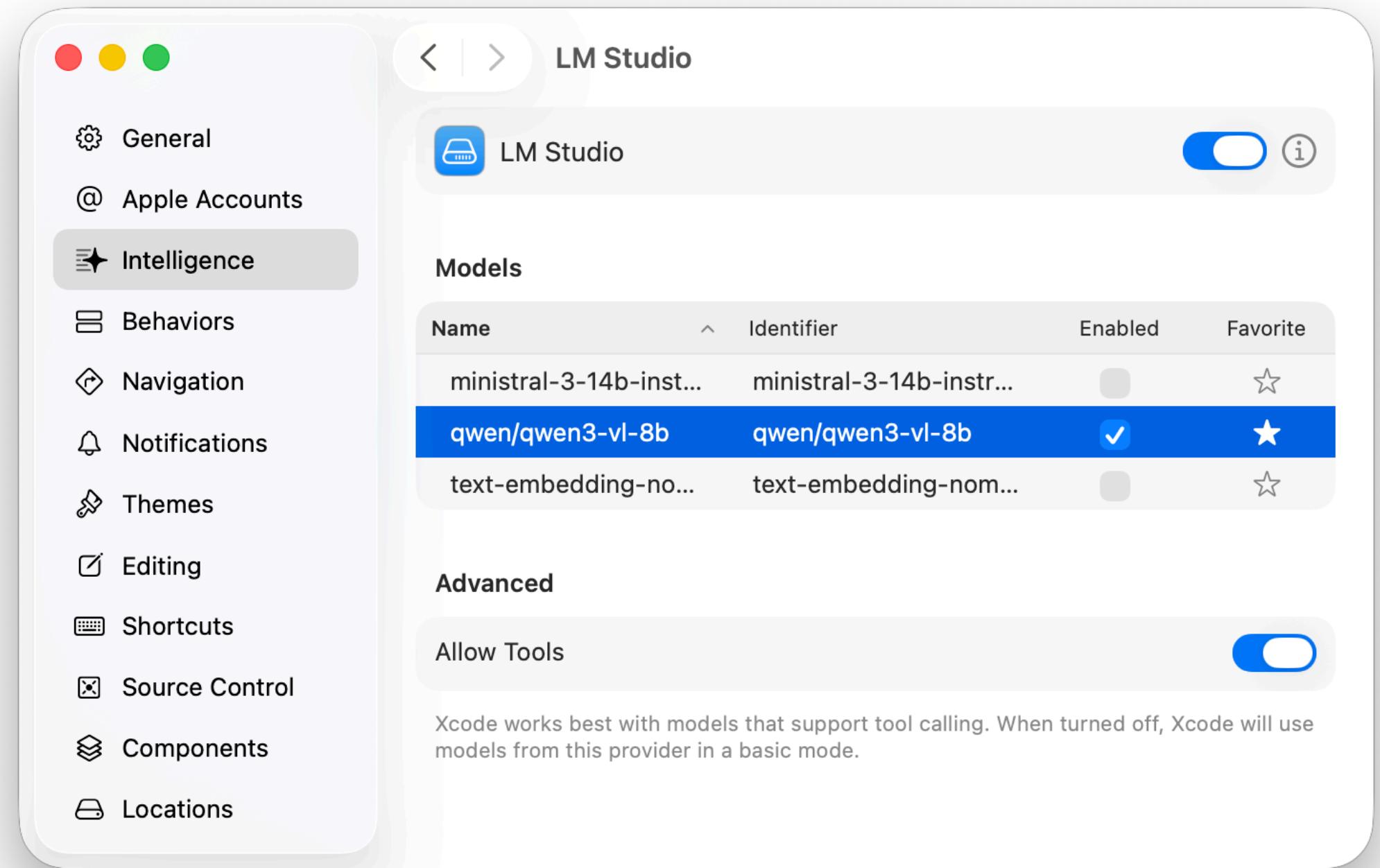
Unable to load models for LM Studio

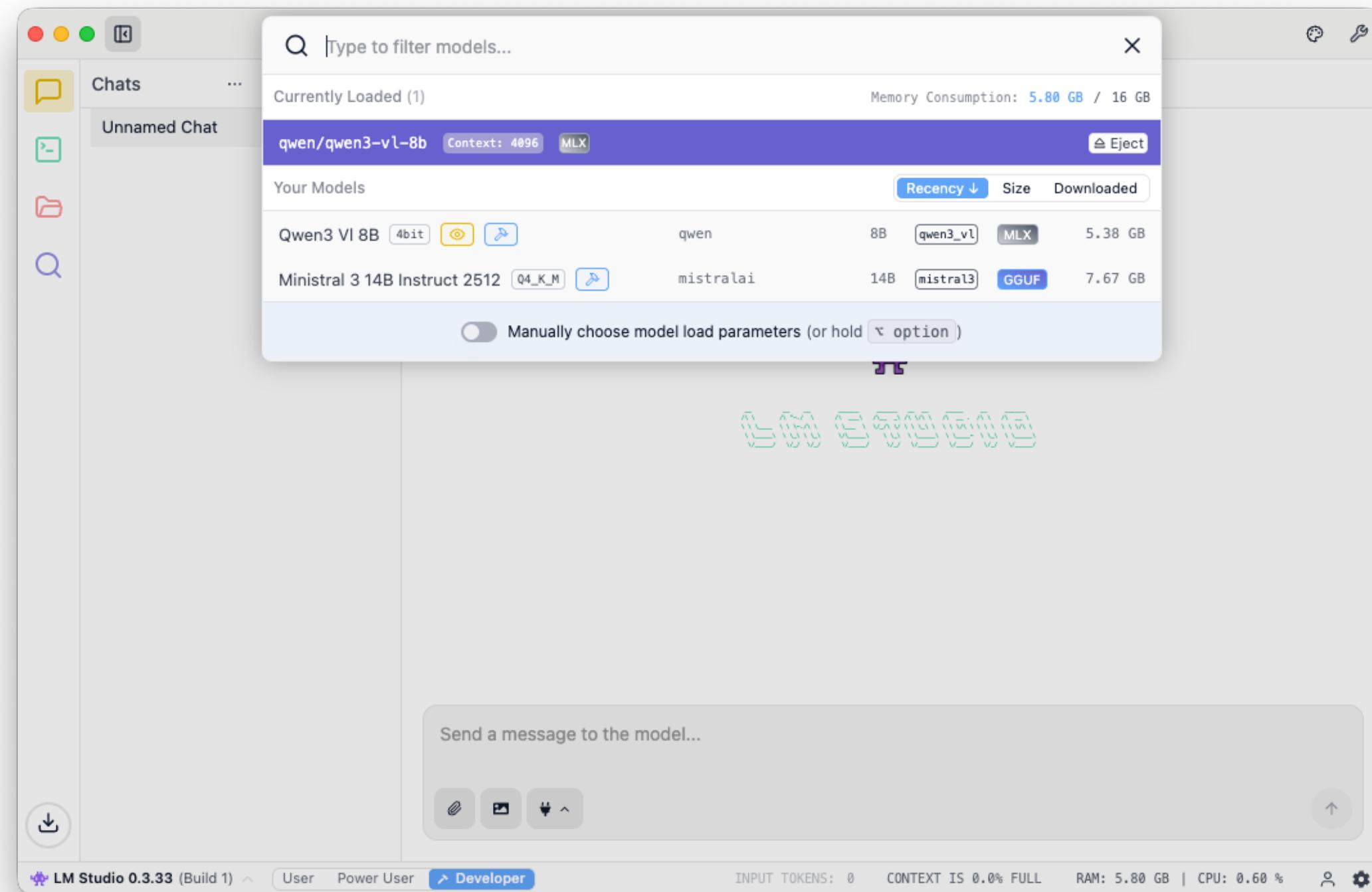
Check your internet connection and try again.

Refresh

```
lms server start  
# Puis quand on a fini  
lms server stop
```







GGUF vs MLX : Comprendre la Différence

GGUF :

- Format universel basé sur llama.cpp
- Compatible macOS / Windows / Linux
- Très léger grâce aux quantizations (Q2-Q8)
- Excellentes perfs CPU, bonnes perfs GPU
- Large écosystème, très stable dans LM Studio

Limites GGUF :

- Perte de qualité possible selon la quantization
- Pas optimisé à 100% pour Apple Silicon
- Conçu uniquement pour l'inférence

MLX :

- Le format Apple Silicon
- Framework créé par Apple
- Exploite GPU + Neural Engine
- Performances supérieures sur Mac
- Idéal pour les modèles 7B-13B

Limites MLX :

- Fonctionne uniquement sur Mac
- Moins de quantizations disponibles
- Demande plus de VRAM

Quand utiliser GGUF ? :

- Multi-plateforme (Windows/Linux)
- Modèles très quantifiés (Q2-Q4)
- Écosystème large et stable
- Besoin de compatibilité maximale

Quand utiliser MLX ? :

- Développement exclusif Mac
- Modèles 7B-13B non quantifiés
- Besoin de performances maximales
- Exploitation Neural Engine

Ollama : LLMs en CLI

Installation :

```
brew install ollama
```

```
brew services start ollama
```

Commandes essentielles :

```
# Lister modèles disponibles  
ollama list
```

```
# Télécharger un modèle  
ollama pull qwen/qwen3-v1-8b
```

```
# Exécuter interactivement  
ollama run qwen/qwen3-v1-8b
```

```
# One-shot  
ollama run qwen/qwen3-v1-8b "Explain Swift Sendable protocol"
```

```
# API serveur (localhost:11434)  
curl http://localhost:11434/api/generate -d '{  
    "model": "qwen3-v1-8b",  
    "prompt": "Write a SwiftUI login view"  
}'
```

Cursor IDE pour iOS Development

Qu'est-ce que Cursor ? :

- IDE basé sur VS Code
- IA native intégrée (GPT-5.1, Claude Sonnet 4.5)
- Autocomplete AI-powered
- Chat contextuel
- Edit via prompts

Raccourcis essentiels :

- Tab : Accepter autocomplete
- Cmd+K : Éditer sélection via prompt
- Cmd+L : Ouvrir chat

The screenshot shows the homepage of the XcodeBuildMCP website. At the top, there's a navigation bar with links for Features, Installation, Usage Examples, Contributing, and GitHub. Below the navigation is a banner with the text "AI-Powered Xcode Automation". The banner features a logo with a hammer and gear icon, followed by the text "XcodeBuild MCP". Below the banner are buttons for "Get Started" and "View on GitHub". Underneath, there's a section titled "Powerful Xcode Integration" with the subtext "Everything you need to integrate Xcode workflows with AI assistants and automation tools." At the bottom, there are three large buttons with icons: a lightning bolt, a greater than sign, and a shield.

The screenshot shows the GitHub repository page for XcodeBuildMCP. The repository has 2938 stars, 135 forks, and is version v1.14.1. The main tab is selected, showing 19 branches and 55 tags. The commit history lists 676 commits from the user cameroncooke, starting with "Ignore derived data" (commit 015c14d) and ending with "Create LICENSE" (commit 4ce793). The commits are organized into several folders like .claude/agents, .cursor, .github, .vscode, build-plugins, docs, example_projects, scripts, src, .axe-version, .cursrrules, .gitignore, .prettierignore, .prettierrc.js, AGENTS.md, CHANGELOG.md, CLAUDE.md, CODE_OF_CONDUCT.md, Dockerfile, and LICENSE. On the right side, there's an "About" section with details about the project, including its purpose (integrating AI assistants with Xcode), links to its website and GitHub pages, and various GitHub statistics like 2.9k stars, 13 watching, and 135 forks. There are also sections for Releases (with a latest release at v1.14.1), Sponsor this project, and Contributors.

Best Practices : IA au Service du Dev

Règle d'or :

"L'IA accélère un senior, elle ne remplace pas un senior."

DO :

- Review tout code généré par IA
- Comprendre avant d'utiliser
- Tester extensively les suggestions
- Utiliser IA pour boilerplate et tests
- Apprendre des explications IA

DON'T :

- Copier-coller sans comprendre
- Faire confiance aveuglément
- Ignorer les warnings du compilateur
- Utiliser IA pour crypto/sécurité critique
- Remplacer code review humain

Conclusion

Apple Intelligence & IA pour Développeurs

Messages Clés

Apple Intelligence :

- Architecture à 3 niveaux (on-device / PCC / ChatGPT)
- Foundation Models framework = accès direct au LLM
- Privacy-first avec garanties vérifiables

App Intents :

- "Anything your app does should be an App Intent"
- Exposition dans Siri, Spotlight, Shortcuts, etc.
- Proactive suggestions by AI

Writing Tools & Text APIs :

- Intégration automatique dans text fields
- Natural Language framework pour analyse
- Résumé et extraction structurée

Outils IA Dev :

- ChatGPT / Claude Code pour génération
- LM Studio / Ollama pour LLMs locaux
- Cursor IDE pour workflow augmenté

L'IA n'est pas Magique

"L'IA est un super assistant, pas un remplaçant."

Réalités :

- L'IA fait des erreurs (hallucinations)
- Elle nécessite supervision humaine
- Elle excelle sur patterns connus
- Elle struggle sur nouveautés

Rôle du développeur :

- Architecte et reviewer
- Validator et testeur
- Curator de qualité
- Gardien de la sécurité

Ressources pour Aller Plus Loin

Documentation Apple :

- developer.apple.com/apple-intelligence/
- developer.apple.com/documentation/FoundationModels
- developer.apple.com/documentation/appintents
- security.apple.com/documentation/private-cloud-compute

WWDC Sessions :

- "Meet the Foundation Models framework" (WWDC25)
- "Bring your app to Siri" (WWDC24)
- "Get started with Writing Tools" (WWDC24)
- "Get to know App Intents" (WWDC25)

Outils & Communities :

- lmstudio.ai (LLMs locaux GUI)
- ollama.com (LLMs CLI)
- cursor.com (IDE AI-powered)
- anthropic.com/cl Claude-code

Merci ! Questions ?

Prochaines étapes :

- Intégrez App Intents dans vos apps
- Testez Foundation Models
- Adoptez les outils IA dans votre workflow
- Partagez vos apprentissages !

Restons en contact :

- Twitter/X : [@clementsauvage](https://twitter.com/clementsauvage)
- GitHub : github.com/csauvage

Au travail !