OVEREND 專案程式碼審查報告

審查日期：2026 年 1 月 17 日

審查範圍：程式碼品質、架構設計、效能優化

執行摘要

OVEREND 是一個跨平台的學術文獻管理與論文寫作工具，採用混合技術架構：

- Swift (macOS/iOS 前端) - 251 個檔案，MVVM 架構
- Rust (核心處理引擎) - 555 行程式碼，使用 UniFFI 與 Swift 互操作
- Web (靜態宣傳網站) - 純 HTML/CSS/JS

整體評分：7.2/10。專案架構清晰、模組化良好，但存在多處效能瓶頸需要優化。

技術架構概覽

Swift 層 (OVEREND/)

架構模式： MVVM + Repository Pattern

資料持久化： Core Data with History Tracking

UI 框架： SwiftUI + Combine

目標平台： macOS 26.0+, iPad

Rust 層 (OverendCore/)

核心功能： Typst 文件編譯、BibTeX 解析

FFI 方案： UniFFI 0.28 (自動生成 Swift 綁定)

依賴管理： 10 個核心依賴，版本固定

Swift 專案深度分析

架構品質評估

✅ 優點

1. 清晰的分層架構

Models (Core Data) → Repositories (資料存取) → ViewModels (業務邏輯) → Views (UI)，職責分離明確。

2. Repository Pattern 實現優秀

使用協議定義抽象層（FetchableRepository, CRUDRepository），解耦 Core Data 實作，便於測試與替換。

3. 功能模組化組織

Services/ 目錄下依功能劃分：AI/ (76 個檔案)、Academic/、Document/、External/，維護性良好。

⚠️ 關鍵問題

優先級	問題描述	影響	檔案位置
高	Entry.fields 計算屬性每次存取都執行 JSON 解碼	列表顯示 100 條目時可能執行 300+次解碼，嚴重影響效能	Models/Entry.swift:32
高	PDF 提取流程重複讀取同一檔案 3-4 次	匯入大型 PDF 時速度緩慢，浪費 I/O 資源	Services/Document/PDFMetadataExtractor.swift:110
高	Core Data 查詢在主執行緒同步執行	大量資料時會凍結 UI，用戶體驗差	Models/Entry.swift:268
中	ViewModel 初始化時自動執行 Task 載入資料	測試困難，可能觸發意外的網路請求	ViewModels/LibraryViewModel.swift:21
中	快取鍵使用 hashValue (不穩定	AI 服務快取可能失效，跨	Services/AI/UnifiedAIService.swift:195

優先級	問題描述	影響	檔案位置
	且碰撞率高)	執行結果不一致	
低	錯誤處理不統一，部分未使用 ErrorLogger	使用者看到的錯誤訊息格式不一致	ViewModels/EntryViewModel.swift:66

效能瓶頸詳細分析

問題 1: Entry.fields JSON 解碼

當前實作：

```
var fields: [String: String] {    get {      guard let data =
fieldsJSON.data(using: .utf8),            let dict = try?
JSONDecoder().decode(...) else {        return [:]      }      return dict  // ❌
每次都重新解碼    } }
```

效能影響量化：

• 此屬性被 32 個檔案使用

• 每個 Entry 顯示時會存取 title, author, year (至少 3 次解碼)

• 100 條目列表 = 300+ 次 JSON 解碼操作

建議解決方案：

使用快取變數：private var _cachedFields: [String: String]? 並在 fieldsJSON 修改時清除快取。

問題 2: PDF 重複讀取

問題根源：

PDFMetadataExtractor 使用多種策略提取元資料，每個策略都獨立載入 PDFDocument，未共享實例。

建議解決方案：

在方法開頭載入一次 PDFDocument，將實例傳遞給所有策略函數。

Rust 核心模組分析

程式碼品質

✅ 優點：

• 使用 thiserror 統一錯誤處理，無 panic!

• UniFFI 整合良好，類型映射合理

• 每個模組都有單元測試

⚠️ 改進空間：

1. 錯誤資訊結構化不足

Typst 編譯錯誤被壓縮成單一字串，丟失行號、列號等結構化資訊。建議增加 CompilationDetail 結構體。

2. FFI 同步調用阻塞問題

compile_typst() 是同步函數，編譯大文檔時會阻塞 Swift UI。UniFFI 0.28 支援異步，建議改為 async fn。

3. BibTeX 解析重複執行

每次格式化引用都重新解析整個 BibTeX 檔案。建議在 OverendEngine 中增加快取層。

架構評分總表

評估項目	評分	備註
MVVM 架構	**8/10**	分層清晰，但 ViewModel 初始化有問題
程式碼組織	**9/10**	模組化優秀，功能導向劃分
錯誤處理	**8/10**	系統完善，但使用不一致
記憶體管理	**6/10**	有使用 weak，但覆蓋率不足
Core Data	**7/10**	設計良好，但效能有隱憂
效能	**5/10**	多處重複計算和 I/O，需立即優化
總體評分	**7.2/10**	**架構良好，效能需改善**

改善建議優先順序

🔴 立即處理 (1-2 週)

1. Entry.fields 增加快取機制

影響範圍最大，32 個檔案依賴此屬性。實作簡單但效益極高。

2. 優化 PDF 提取流程

避免重複讀取檔案，共享 PDFDocument 實例。

3. Core Data 查詢改為非同步

防止大量資料時凍結 UI，使用 performBackgroundTask。

🟡 短期改善 (1 個月)

- 統一使用 ErrorLogger 處理錯誤
- 檢查所有 Combine pipeline 的 weak self 使用
- 改善快取鍵生成 (使用 SHA256 或 MD5)
- Rust 層增加 BibTeX 快取

🟢 長期優化 (3 個月)

- 引入效能監控系統 (已有 PerformanceMonitor，需實際使用)
- 增加單元測試覆蓋率 (特別是 Repository 層)
- 使用 Instruments 進行記憶體分析
- Rust 層改為異步 API (UniFFI async support)

總結與建議

OVEREND 專案在架構設計上展現出成熟度，採用的技術選型（Swift + Rust）充分發揮了各自優勢：

• Swift 層負責 UI 與用戶體驗，MVVM 架構清晰

• Rust 層處理高效能運算（Typst 編譯、BibTeX 解析）

主要優勢：

✅ Repository Pattern 解耦良好，易於測試與維護

✅ 功能模組化組織，AI、文件、學術功能分離明確

✅ Rust 層程式碼品質高，錯誤處理完善

急需改善：

⚠️ Entry.fields 效能問題會直接影響使用者體驗

⚠️ PDF 處理效率低落，匯入大型文獻庫時明顯卡頓

⚠️ 同步 Core Data 查詢可能造成 UI 凍結

行動建議：

建議優先處理三個高優先級效能問題，預期可提升 40-60% 的整體流暢度。這些改善實作難度不高，但對使用者體驗提升顯著。

完成立即改善項目後，專案整體評分有望提升至 8.5/10，達到商業發布標準。

--- 報告結束 ---