長篇教材自動化排版與轉換系統深度技術調研報告：從 CSS Paged Media 到 韌性 AI 架構的完整實踐

1. 導論與問題空間分析

在當前的數位教育與出版領域，將原本結構鬆散的長篇內容（如課程講義、技術手冊、教科書）轉換為符合專業出版標準的格式（PDF 與 Word），是一個兼具高度技術挑戰與業務價值的課題。隨著生成式 AI 技術的普及，內容生產的速度大幅提升，然而「內容生成」與「專業呈現」之間仍存在巨大的斷層。傳統的網頁列印（Browser Print）無法滿足教材對於頁眉、頁腳、雙頁排版（Spread Layout）、註腳（Footnotes）以及交叉引用（Cross-references）的嚴苛要求；而生成式 AI 在處理數十萬字的長篇教材時，面臨著上下文遺忘（Context Rot）、幻覺（Hallucination）以及處理中斷（Processing Interruption）等穩定性問題。

本報告旨在針對「開發一個能將長篇教材轉換為專業教材排版 PDF 與 Word 的網頁程式」這一需求，進行詳盡的技術調研與架構設計。我們將深入探討開源社群中的成熟排版框架，分析其優劣與適用場景，並提出一套基於現代分散式架構的 AI 處理方案，以解決大檔案處理中的中斷與一致性問題。

1.1 專業教材排版的特殊性

與一般的商業報告或發票不同，教材排版具有其獨特的複雜性。首先是結構的階層性，教材通常包含章（Chapter）、節（Section）、小節（Subsection）等多層級結構，這要求排版引擎能夠自動生成準確的多層級目錄（Table of Contents, TOC）[1]。其次是輔助閱讀元素的豐富性，包括邊註（Marginalia）、腳註（Footnotes）、以及大量的圖表與代碼塊。特別是在長文檔中，如何確保跨頁內容的連貫性，例如避免標題孤立於頁面底部（Orphans/Widows控制），以及如何處理複雜的表格跨頁斷裂，都是技術選型的關鍵考量點 [2]。

1.2 長篇文檔處理的技術瓶頸

在技術實作層面，長篇教材（通常超過 100 頁或 5 萬字）的處理面臨兩大核心瓶頸：

1. 渲染引擎的內存與性能限制：傳統的基於 DOM 的渲染庫（如 React-PDF 或基於 Canvas 的方案）在處理數千個 DOM 節點時，極易導致瀏覽器崩潰或內存溢出（OOM）。例如，Puppeteer 在渲染包含大量圖片或複雜 CSS 佈局的長頁面時，內存佔用可能呈指數級增長 [3]。
2. **AI** 處理的時效性與穩定性：當使用 LLM 對長篇教材進行潤色、翻譯或格式化時，單次 API 請求往往無法容納整個文檔的上下文（即使有 1M Context Window）。此外，HTTP 請求的超時限制（通常為 30-60 秒）與 AI 處理所需的數分鐘甚至數小時形成了根本性的衝突。如果沒有強健的狀態管理與斷點續傳機制，任何網絡波動或服務端錯誤都將導致昂貴的計算資源浪費 [4]。

本報告將依次解構上述問題，從排版引擎的選型對比，到 Word 文檔生成的特殊挑戰，再到基於非同步隊列與檢查點機制的 AI 架構設計，最後匯總為一份可執行的開發者文件草案。

---

2. 大文件專業排版框架深度調研

在開源社群中，將標記語言（Markdown/HTML）轉換為高品質 PDF 的技術路線主要分為三類：基於瀏覽器（Headless Browser）的渲染、專用排版引擎（Dedicated Engines）以及前端組件庫。針對教材類文檔的專業需求，我們重點評估了 **Paged.js**、**WeasyPrint**、**React-PDF** 以及相關輔助工具。

2.1 CSS Paged Media 標準與瀏覽器渲染方案

W3C 定義的 **CSS Paged Media Module Level 3** 是實現網頁轉印刷品的核心標準。它引入了 @page 規則，允許開發者定義頁面尺寸、邊距、頁眉頁腳區域（Margin Boxes）以及分頁策略 [6]。然而，主流瀏覽器（Chrome, Firefox, Safari）對這一標準的支援度極低，僅支援基本的 @page 尺寸設置，缺乏對頁眉頁腳內容（Generated Content in Margin Boxes）的原生支援 [8]。

2.1.1 Paged.js：瀏覽器端的排版 Polyfill

Paged.js 是目前開源社群中最受矚目的解決方案之一。作為一個 JavaScript 庫，它在瀏覽器端「填補」了瀏覽器對 CSS Paged Media 標準支援的空白。

- 工作原理：Paged.js 不依賴瀏覽器原生的分頁機制，而是將 HTML 內容解析為一系列碎片（Fragments），然後根據定義好的 CSS 網格模型，將這些碎片動態填充到一個個分頁容器中。這種機制使得它能夠精確控制每一個元素的分割位置 [9]。
- 教材排版優勢：
  - 完全支援 **CSS Paged Media**：它支援 @top-center、@bottom-right 等 16 個邊距盒子，能夠輕鬆實現「章節標題在左頁眉，頁碼在右頁腳」的複雜佈局 [11]。
  - 交叉引用與計數器：透過 target-counter() 函數，Paged.js 可以自動計算並渲染「參見第 X 頁」這樣的交叉引用，這對於教材中的圖表索引至關重要 [12]。
  - 預覽即所得：由於運行在瀏覽器中，開發者可以直接使用 Chrome DevTools 調試排版樣式，極大提升了開發效率 [13]。
- 性能挑戰：對於數百頁的文檔，Paged.js 在客戶端進行內容分割（Fragmentation）會消耗大量的 CPU 資源，導致頁面卡頓。在生產環境中，通常配合 Headless Chrome（如 Puppeteer）在伺服器端運行，生成 PDF 後再傳送給用戶 [9]。

2.1.2 Puppeteer 與 Headless Chrome 的角色

Puppeteer 是控制 Headless Chrome 的 Node.js 庫。雖然它本身只提供基礎的 page.pdf() 功能，但結合 Paged.js，它成為了最強大的渲染後端 [14]。

- 優勢：支援所有現代 Web 特性（Flexbox, Grid, SVG, Canvas），這意味著教材中的動態圖表（如使用 D3.js 生成的統計圖）可以被完美渲染為向量 PDF，這是 Python 方案（如

WeasyPrint）無法做到的 [15]。

- 內存洩漏風險：研究指出，Puppeteer 在處理大文件時，每個瀏覽器實例可能佔用 500MB 以上內存。若不妥善管理頁面關閉與瀏覽器重啟，極易導致伺服器崩潰。對於大批量教材生成，必須採用連接池（Browser Pool）模式 [3]。

2.2 專用排版引擎：WeasyPrint

WeasyPrint 是一個基於 Python 的視覺渲染引擎，它不使用 WebKit 或 Blink，而是自行解析 HTML 和 CSS 並繪製 PDF [16]。

- 技術特性：WeasyPrint 對 CSS Paged Media 的支援度極高，甚至優於 Paged.js 在某些邊緣情況下的表現。它能夠生成符合 PDF/A 標準的文檔，適合長期歸檔 [18]。
- 與 **Paged.js** 的對比：
  - 註腳（**Footnotes**）支援：WeasyPrint 較早實現了 CSS Fragment 規範中的註腳功能（float: footnote），能夠自動將註釋排版在頁面底部，這在學術教材中是剛需 [19]。相比之下，Paged.js 對註腳的支援仍在完善中，有時需要額外的 JS 腳本輔助。
  - **JavaScript** 限制：WeasyPrint 不支援執行 **JavaScript**。這意味著所有內容必須是靜態的 HTML/CSS。如果教材中包含動態生成的數學公式（如 MathJax）或圖表，必須先在服務端將其預渲染為 SVG 或圖片，這增加了管線的複雜度 [21]。
  - 性能瓶頸：在處理包含大量複雜表格（Tables）的長文檔時，WeasyPrint 的渲染速度明顯慢於 Headless Chrome，且內存佔用同樣可觀。基準測試顯示，處理 1000 行以上的表格可能導致內存激增至 2GB 以上 [22]。

2.3 其他方案評估：React-PDF 與 Pandoc

- **React-PDF**：這是一個用於在瀏覽器或 Node 端生成 PDF 的 React 渲染器。它不是將 HTML 轉為 PDF，而是提供了一套專屬的組件（<View>, <Text>）。雖然它在生成簡單票據時非常高效，但缺乏對複雜排版邏輯（如自動分頁、孤行控制、斷字）的支援，且無法復用現有的 CSS 生態，不推薦用於專業教材開發 [15]。
- **Pandoc (PDF 路徑)**：Pandoc 通常透過 LaTeX 引擎生成 PDF。雖然其排版品質極高（尤其是數學公式），但 LaTeX 的學習曲線極陡峭，且自定義樣式極其困難。對於現代 Web 開發者而言，基於 HTML/CSS 的方案（Paged.js/WeasyPrint）在可維護性上遠勝於 LaTeX [25]。

2.4 綜合評估與選型建議

特性維度	**Paged.js + Puppeteer**	**WeasyPrint**	**React-PDF**
排版標準	CSS Paged Media (JS 模擬)	CSS Paged Media (原生)	自定義組件
JavaScript 支援	完整支援 (含圖表	不支援	JS 環境但非 DOM

	庫)		
註腳/目錄	強 (需配置腳本)	強 (原生 CSS 支援)	弱 (需手寫邏輯)
調試難度	低 (Chrome DevTools)	中 (生成文件查看)	中
性能/內存	高消耗 (Chromium)	中高消耗 (Python)	低消耗
適用場景	富媒體教材、動態圖表	純文本文檔、Python 棧	簡單報告

結論：針對「長篇教材」且包含潛在的圖表與豐富樣式需求，**Paged.js + Puppeteer** 是目前最靈活且生態最成熟的選擇。雖然需注意內存管理，但其對現代 CSS 的支援和可調試性使其成為開發首選。

---

3. Word (Docx) 生成的技術挑戰與解決方案

與 PDF 的固定版式不同，Word 文檔是流式（Flow-based）的。開發長篇教材轉換程式時，Word 生成的難點不在於內容本身，而在於樣式系統的映射與動態域（**Fields**）的更新。

3.1 核心轉換引擎：Pandoc 的深度應用

Pandoc 是目前最權威的文檔轉換工具，尤其擅長將 Markdown 轉換為 Docx。其核心優勢在於其抽象語法樹（AST）能很好地保留文檔結構（章節、列表、引用）[26]。

3.1.1 Reference Doc 機制：解決樣式痛點

直接轉換的 Docx 文件通常樣式簡陋。Pandoc 提供了 --reference-doc 參數，允許開發者提供一個「樣式模板」文件。

- 機制：Pandoc 不會複製模板的內容，而是讀取其樣式定義（Styles.xml）。開發者可以在 Word 中預先設計好「Heading 1」的字體顏色、段落間距、邊框等，保存為 reference.docx。
- 應用：在生成教材時，系統應動態調用 Pandoc：
  pandoc input.md -o output.docx --reference-doc=template.docx
  這樣生成的文檔將自動繼承模板中的所有專業排版樣式，無需編寫複雜的 XML 操作代碼 25。

3.2 目錄與頁碼的自動更新難題

這是所有服務端 Word 生成方案的「阿基里斯之踵」。Pandoc 可以生成帶有目錄標記（TOC Field

code）的文檔，但它無法計算頁碼，因為 Word 文檔的分頁是動態的，只有在 Word 渲染引擎打開文檔時才能確定頁碼 [28]。

3.2.1 解決方案 A：Headless LibreOffice (推薦)

利用 LibreOffice 的命令行模式在服務端「打開」並保存文檔，觸發索引更新。

- 實作：使用 libreoffice --headless --convert-to pdf 或透過 UNO 接口調用宏（Macro）。
- 優勢：開源免費，Linux 友好。
- 劣勢：安裝包巨大，且 LibreOffice 的渲染與 MS Word 存在細微差異，可能導致頁碼偏差 [30]。

3.2.2 解決方案 B：Win32 COM Automation (Windows Server)

在 Windows 伺服器上安裝 MS Word，透過 Python 的 win32com 庫調用 Word 應用程序。

- 代碼示例：
  ```
  Python
  word = win32com.client.DispatchEx("Word.Application")
  doc = word.Documents.Open(file_path)
  doc.TablesOfContents(1).Update()
  doc.Save()
  ```

- 評估：這是唯一能保證 100% 正確頁碼的方案，但維護 Windows 伺服器成本高，且併發能力差 [32]。

3.2.3 解決方案 C：docx.js / python-docx (純代碼生成)

- **docx.js**：Node.js 庫，支援宣告式地創建文檔。雖然功能強大，但對於長篇文檔，手動構建數萬個段落對象會導致代碼極其臃腫且難以維護。它更適合作為 Pandoc 生成後的「後處理（Post-processing）」工具，例如用來插入特定的浮水印或元數據 [33]。
- **python-docx**：同樣適合作為後處理工具。雖然它無法更新 TOC，但可用於調整分節符（Section Breaks）以實現不同的頁眉頁腳設置 [35]。

結論：建議採用 **Pandoc + Reference Doc** 進行主體生成，並在 Linux 環境下利用 **Headless LibreOffice** 進行目錄刷新，這是目前性價比最高的自動化路徑。

---

4. 解決大檔案導致 AI 處理中斷的韌性架構

當教材長度達到數十萬字時，依賴單次 LLM 調用進行潤色或格式化是不可行的。這不僅涉及 API 成本與速率限制（Rate Limits），更核心的是系統的穩定性與上下文的有效性。

4.1 LLM 的「上下文腐爛」與分塊策略

雖然 Gemini 1.5 Pro 等模型支援 1M Token 的上下文，但研究表明，隨著輸入長度的增加，模型對中間內容的檢索能力會下降（Lost-in-the-Middle Phenomenon），且指令遵循能力（如「保持語氣

一致」）會變弱 [37]。

4.1.1 策略：Map-Reduce 與 Hierarchical Summarization

針對長篇教材，應採用分而治之的策略：

1. **Chunking**（分塊）：按章節（Chapter）或語義段落切分文檔。對於教材，章節是最自然的分割點。
2. **Map**（映射處理）：並行處理每個章節。例如，並行發送 10 個請求對 10 個章節進行「Markdown 語法標準化」。
3. **Reduce**（歸約合併）：將處理後的章節合併，再進行一次全局的一致性檢查（如術語統一）[39]。

4.2 非同步任務隊列架構

為了解決 HTTP 超時與處理中斷問題，必須將「請求」與「處理」解耦，採用 非同步任務隊列（**Asynchronous Task Queues**）。

4.2.1 技術選型：BullMQ vs Celery

- **BullMQ (Node.js)**：基於 Redis 的輕量級隊列。
  - 優勢：與 NestJS/Express 生態整合極佳，支援延遲任務、優先級隊列，且對 TypeScript 友好 [41]。
  - 適用：如果後端主體是 Node.js，BullMQ 是首選。
- **Celery (Python)**：Python 生態的工業級標準。
  - 優勢：功能極其豐富，支援多種 Broker（RabbitMQ, Redis），與 LangChain/LlamaIndex 等 Python AI 庫無縫銜接 [43]。
  - 適用：考慮到大多數強大的 AI 處理庫（如 PyTorch, HuggingFace）都在 Python，採用 Celery Worker 處理 AI 任務是更合理的選擇。

4.2.2 架構設計：生產者-消費者模式

1. **Web Server**：接收用戶上傳的文件，進行初步分塊，將每個分塊封裝為一個 Task，推入 Redis 隊列，並立即返回 Job ID 給前端。
2. **Worker Cluster**：後台 Worker 從隊列中搶佔任務。
   - 若任務失敗（如 LLM API 報錯），Worker 根據指數退避（Exponential Backoff）策略自動重試。
   - 利用 Redis 存儲每個分塊的處理進度（Progress），前端透過 WebSocket 或輪詢（Polling）獲取實時進度條 [45]。

4.3 持久化執行與檢查點機制 (Checkpointing)

僅有隊列是不夠的。如果一個長達 1 小時的流程在第 55 分鐘崩潰，重頭開始是不可接受的。我們需要引入**持久化執行（Durable Execution）**概念。

4.3.1 LangGraph 與 Checkpointer

LangChain 推出的 **LangGraph** 框架專為有狀態的 AI Agent 設計。它引入了 Checkpointer 機制

（通常基於 Postgres 或 SQLite）。

- 機制：在圖（Graph）的每一個節點執行完畢後，系統會自動將當前的 State（包括已生成的文本、變量狀態）序列化並寫入資料庫。
- 恢復（**Resumability**）：當流程中斷並重啟時，系統讀取 Thread ID 對應的最後一個 Checkpoint，直接從斷點處繼續執行，而非重新開始。這對於長篇教材處理至關重要，能節省巨大的 API Token 成本 [47]。

4.3.2 Temporal.io 的角色

對於更複雜的業務流程（如涉及人工審核步驟），**Temporal.io** 提供了更底層的「代碼即工作流」保證。它記錄每一步的執行歷史（Event History），確保無論伺服器重啟多少次，工作流都能準確恢復。雖然學習曲線較陡，但對於企業級應用是最佳實踐 [49]。

4.4 術語一致性與 RAG 注入

在分塊處理中，最大的風險是不同章節對同一專有名詞的翻譯不一致。解決方案是引入 **RAG**（檢索增強生成）與 術語庫（**Glossary**）。

- 術語提取：在處理前，先用輕量模型掃描全文，提取高頻術語，生成 glossary.json。
- 上下文注入：在處理每個分塊（Chunk）時，將該術語表作為 System Prompt 的一部分注入：「請遵循以下術語翻譯對照表...」。這確保了即便並行處理，風格和用詞也能保持一致 [37]。

---

5. 開發者文件草案 (Developer Documentation Draft)

標題：EduFormat Engine - 長篇教材自動化排版系統開發指南

版本: 0.1.0 (Draft)
目標讀者: 後端工程師、AI 工程師

5.1 系統架構概覽 (System Architecture)

本系統採用 事件驅動微服務架構，旨在處理高併發、長耗時的文檔轉換任務。

5.1.1 核心組件

1. **API Gateway (Node.js/NestJS)**: 處理文件上傳、任務調度、鑑權。
2. **Task Broker (Redis)**: 負責消息隊列與短期狀態存儲。
3. **AI Worker (Python/Celery)**: 執行 LangGraph 工作流，進行內容清洗與結構化。
4. **Render Worker (Node.js/Puppeteer)**: 運行 Paged.js 生成 PDF。
5. **Docx Worker (Pandoc/LibreOffice)**: 處理 Word 轉換與目錄更新。
6. **Persistent DB (PostgreSQL)**: 存儲任務元數據、LangGraph Checkpoints。

5.1.2 數據流 (Data Flow)

1. 用戶上傳原始 Markdown/Docx。
2. API 服務計算文件 Hash，檢查緩存；若無，上傳至 S3，並創建 Job 記錄。

3. **Pre-processing**: 文件被切分為 Chunks (每章一個)。
4. **AI Processing**:
    - Workers 並行領取 Chunks。
    - 調用 LLM 進行格式修復（添加 Admonitions, LaTeX 公式標準化）。
    - *Checkpointing*: 每個 Chunk 完成後，寫入 PG 數據庫。
5. **Merging**: 所有 Chunks 狀態為 COMPLETED 後，觸發 MergeTask。
6. **Rendering**:
    - 合成 Master HTML (含 Paged.js 腳本)。
    - Puppeteer 列印為 PDF。
    - Pandoc 轉換為 Docx -> LibreOffice 更新目錄。
7. **Callback**: 通過 WebSocket 通知前端下載。

5.2 API 規範 (API Specification)

5.2.1 提交轉換任務

POST /api/v1/jobs

JSON

```
{
  "input_url": "s3://bucket/raw/math_book.md",
  "config": {
    "target_formats": ["pdf", "docx"],
    "style_theme": "academic_v2",
    "ai_enhancement": true
  }
}
```

Response:

JSON

```
{
  "job_id": "550e8400-e29b-41d4-a716-446655440000",
  "queue_position": 2
}
```

5.2.2 查詢任務狀態 (支援長輪詢)

GET /api/v1/jobs/:job_id

JSON

```
{
  "status": "processing",
  "progress_percentage": 65,
  "current_stage": "rendering_pdf",
  "stages": {
    "ai_processing": "completed",
    "docx_generation": "pending"
  },
  "error": null
}
```

5.2.3 任務恢復 (手動觸發)

POST /api/v1/jobs/:job_id/resume
用途：當任務因不可抗力失敗（狀態為 failed），調用此接口可讀取 Checkpoint 並從失敗的步驟繼續。

5.3 數據庫模型 (Database Schema)

Table Name	Description	Key Fields
jobs	總任務記錄	id, user_id, status, s3_paths
chunks	分塊任務狀態	id, job_id, sequence, content_hash, status
checkpoints	LangGraph 狀態	thread_id (關聯 job_id), checkpoint_data (BLOB)
artifacts	生成結果	job_id, file_type (pdf/docx), s3_url

5.4 排版模板開發指南 (Templating Guide)

5.4.1 PDF (Paged.js CSS)

CSS

```css
/* 定義 A4 紙張與邊距 */
@page {
  size: A4;
  margin: 20mm;
  bleed: 3mm; /* 出血位 */

  /* 左頁頁眉：顯示書名 */
  @top-left {
    content: string(book-title);
  }
  /* 右頁頁眉：顯示當前章節名 */
  @top-right {
    content: string(chapter-title);
  }
  /* 頁腳：頁碼 */
  @bottom-center {
    content: counter(page);
  }
}

/* 提取章節標題到變量 */
h1.chapter-title {
  string-set: chapter-title content();
  break-before: right; /* 章節總是從右頁開始 */
}

/* 交叉引用樣式 */
a.cross-ref::after {
  /* 自動生成：(參見第 5 頁) */
  content: " (參見第 " target-counter(attr(href url), page) " 頁)";
  font-style: italic;
}
```

5.4.2 Word (Pandoc Reference)

1. 準備模板：在 Word 中創建 reference.docx。
2. 樣式定義：

    - 修改 Heading 1 為所需字體（如黑體 16pt）。
    - 定義 TOC Heading 樣式以控制目錄標題外觀。
3. 轉換命令封裝：
   Bash
   ```
   pandoc source.md \
    --output output.docx \
    --reference-doc=styles/academic.docx \
    --toc \
    --toc-depth=3
   ```

5.5 AI 處理韌性實作 (AI Resilience Implementation)

5.5.1 LangGraph 檢查點配置 (Python)

Python

```
from langgraph.checkpoint.postgres import PostgresSaver
from langgraph.graph import StateGraph

# 初始化 DB 連接與 Checkpointer
pool = psycopg2.connect(...)
checkpointer = PostgresSaver(pool)

# 定義工作流圖
workflow = StateGraph(State)
workflow.add_node("standardize_markdown", standardize_node)
workflow.add_node("translate_terms", translate_node)
workflow.set_entry_point("standardize_markdown")

# 編譯時啟用 Checkpoint
app = workflow.compile(checkpointer=checkpointer)

# 執行任務（帶有 thread_id）
def process_job(job_id, content):
    config = {"configurable": {"thread_id": job_id}}
    # 若此 job_id 曾執行過，將自動加載上次狀態
    for event in app.stream({"content": content}, config=config):
        update_progress(job_id, event)
```

5.5.2 錯誤處理策略

1. **Retryable Errors** (如 429 Too Many Requests): 使用指數退避策略重試（1s, 2s, 4s...）。
2. **Context Overflow**: 若檢測到 Token 超限，自動觸發 split_chunk 邏輯，將當前 Chunk 再拆分為 Sub-chunks。
3. **Validation Failure**: 若 AI 輸出的 JSON 格式錯誤，使用 OutputFixingParser 進行自我修正，或標記為人工介入。

---

6. 結論與未來展望

本調研確認了開發專業級長篇教材轉換系統的可行性。透過結合 **Paged.js** 的強大排版能力、**Pandoc** 的文檔轉換生態，以及基於 **LangGraph/Celery** 的韌性 AI 架構，我們能夠有效解決大檔案處理中的性能瓶頸與穩定性問題。

核心建議：

1. 擁抱 **Web** 標準：優先選擇基於 CSS Paged Media 的渲染方案，這保證了排版的靈活性與未來的可維護性。
2. 重視數據一致性：在 AI 介入的環節，必須引入 Checkpoint 機制與術語庫（Glossary），這是保證長篇文檔品質的關鍵。
3. 混合渲染路徑：針對 PDF 與 Word 採用不同的技術棧（Puppeteer vs Pandoc），不要試圖用一種工具解決所有問題。

未來，隨著瀏覽器對 Paged Media 標準支援的提升，以及 AI Agent 在長上下文推理能力的增強，該系統將能進一步實現「語義級排版」，即 AI 自動根據內容語氣調整排版風格，實現真正的智能出版。

(Report End)

引用的著作

1. Ask HN: Where are the good Markdown to PDF tools (that meet these requirements)? | Hacker News, 檢索日期：1月 15, 2026, https://news.ycombinator.com/item?id=43231964
2. datalab-to/marker: Convert PDF to markdown + JSON quickly with high accuracy - GitHub, 檢索日期：1月 15, 2026, https://github.com/datalab-to/marker
3. Puppeteer Isn't Meant for PDFs — Here's Why | by Anup Singh - Medium, 檢索日期：1月 15, 2026, https://medium.com/@onu.khatri/puppeteer-isnt-meant-for-pdfs-here-s-why-1e3a4419263f
4. Successfully Deploying a Task Queue - Prefect, 檢索日期：1月 15, 2026, https://www.prefect.io/blog/successfully-deploying-task-queue
5. State Management Patterns for Long-Running AI Agents: Redis vs StatefulSets vs External Databases - DEV Community, 檢索日期：1月 15, 2026,

https://dev.to/inboryn_99399f96579fcd705/state-management-patterns-for-long-running-ai-agents-redis-vs-statefulsets-vs-external-databases-39c5
6. CSS paged media - MDN Web Docs - Mozilla, 検索日期：1月 15, 2026, https://developer.mozilla.org/en-US/docs/Web/CSS/Guides/Paged_media
7. CSS Generated Content for Paged Media Module - W3C, 検索日期：1月 15, 2026, https://www.w3.org/TR/css-gcpm-3/
8. Chrome “Print to PDF” and headless --print-to-pdf aren't the same! - André.Arko.net, 検索日期：1月 15, 2026, https://andre.arko.net/2025/05/25/chrome-headless-print-to-pdf/
9. Paged.js - a free and open source JavaScript library that paginates content in the browser to create PDF output from any HTML content. This means you can design works for print (eg. books) using HTML and CSS - Reddit, 検索日期：1月 15, 2026, https://www.reddit.com/r/javascript/comments/f5syqi/pagedjs_a_free_and_open_source_javascript_library/
10. How to use Paged.js, 検索日期：1月 15, 2026, https://pagedjs.org/en/documentation/
11. Generated Content - Paged.js —, 検索日期：1月 15, 2026, https://pagedjs.org/en/documentation/6-generated-content/
12. Cross References - Paged.js —, 検索日期：1月 15, 2026, https://pagedjs.org/en/documentation/-cross-references/
13. Introduction to CSS for Paged Media - Tony Graham - YouTube, 検索日期：1月 15, 2026, https://www.youtube.com/watch?v=Q48PSfvtvBw
14. How to convert HTML to PDF: 10 best tools compared - Nutrient iOS, 検索日期：1月 15, 2026, https://www.nutrient.io/blog/top-ten-ways-to-convert-html-to-pdf/
15. Best JavaScript PDF libraries 2025: A complete guide to viewers, generators, and enterprise solutions - Nutrient iOS, 検索日期：1月 15, 2026, https://www.nutrient.io/blog/javascript-pdf-libraries/
16. WeasyPrint, 検索日期：1月 15, 2026, https://weasyprint.org/
17. Creating a PDF from HTML – Formatting documents with CSS Paged Media - parson AG, 検索日期：1月 15, 2026, https://www.parson-europe.com/en/knowledge-base/html-to-pdf-with-css-paged-media
18. Common Use Cases - WeasyPrint 67.0 documentation - CourtBouillon, 検索日期：1月 15, 2026, https://doc.courtbouillon.org/weasyprint/stable/common_use_cases.html
19. Footnotes | PrintCSS, 検索日期：1月 15, 2026, https://printcss.net/articles/footnotes
20. PrintCSS: Footnotes - by Andreas Zettl - Medium, 検索日期：1月 15, 2026, https://medium.com/printcss/printcss-footnotes-9bb67fb2064b
21. 1.5M PDFs in 25 Minutes - Hacker News, 検索日期：1月 15, 2026, https://news.ycombinator.com/item?id=39379690
22. WeasyPrint consuming a lot of memory when rendering tables with 5000 rows #1104, 検索日期：1月 15, 2026, https://github.com/Kozea/WeasyPrint/issues/1104
23. Performance issue on long documents · Issue #578 · Kozea/WeasyPrint - GitHub, 検索日期：1月 15, 2026, https://github.com/Kozea/WeasyPrint/issues/578

24. HTML to PDF with performances : r/webdev - Reddit, 検索日期：1月 15, 2026, https://www.reddit.com/r/webdev/comments/18zm20r/html_to_pdf_with_performances/
25. Markdown to docx, including complex template - Stack Overflow, 検索日期：1月 15, 2026, https://stackoverflow.com/questions/14249811/markdown-to-docx-including-complex-template
26. Pandoc User's Guide, 検索日期：1月 15, 2026, https://pandoc.org/MANUAL.html
27. Pandoc Markdown to Docx with Cover Page and TOC in separated pages - Stack Overflow, 検索日期：1月 15, 2026, https://stackoverflow.com/questions/52890081/pandoc-markdown-to-docx-with-cover-page-and-toc-in-separated-pages
28. Libraries to generate .docx files? : r/learnpython - Reddit, 検索日期：1月 15, 2026, https://www.reddit.com/r/learnpython/comments/1h8b38j/libraries_to_generate_docx_files/
29. Add Table of Contents, List of Figures and List of Tables · Issue #723 · python-openxml/python-docx - GitHub, 検索日期：1月 15, 2026, https://github.com/python-openxml/python-docx/issues/723
30. Updating table of contents TOC with command line - English - Ask LibreOffice, 検索日期：1月 15, 2026, https://ask.libreoffice.org/t/updating-table-of-contents-toc-with-command-line/56010
31. Automatically update table of contents · Issue #1207 · python-openxml/python-docx - GitHub, 検索日期：1月 15, 2026, https://github.com/python-openxml/python-docx/issues/1207
32. Update the TOC (table of content) of MS Word .docx documents with Python - Stack Overflow, 検索日期：1月 15, 2026, https://stackoverflow.com/questions/32992457/update-the-toc-table-of-content-of-ms-word-docx-documents-with-python
33. Docx, 検索日期：1月 15, 2026, https://docx.js.org/
34. Modifying "style" of .docx TOC using javascript (docx-js) - Stack Overflow, 検索日期：1月 15, 2026, https://stackoverflow.com/questions/64383132/modifying-style-of-docx-toc-using-javascript-docx-js
35. Working with Documents - python-docx - Read the Docs, 検索日期：1月 15, 2026, https://python-docx.readthedocs.io/en/latest/user/documents.html
36. Working with Sections — python-docx 1.2.0 documentation, 検索日期：1月 15, 2026, https://python-docx.readthedocs.io/en/latest/user/sections.html
37. Context Rot: How Increasing Input Tokens Impacts LLM Performance | Chroma Research, 検索日期：1月 15, 2026, https://research.trychroma.com/context-rot
38. Why Gemini 1.5 (and other large context models) are bullish for RAG - Medium, 検索日期：1月 15, 2026, https://medium.com/enterprise-rag/why-gemini-1-5-and-other-large-context-models-are-bullish-for-rag-ce3218930bb4
39. Summarization techniques, iterative refinement and map-reduce for document

workflows | Google Cloud Blog, 検索日期：1月 15, 2026, https://cloud.google.com/blog/products/ai-machine-learning/long-document-summarization-with-workflows-and-gemini-models
40. [2502.00977] Context-Aware Hierarchical Merging for Long Document Summarization, 検索日期：1月 15, 2026, https://arxiv.org/abs/2502.00977
41. BullMQ - Background Jobs processing and message queue for NodeJS | BullMQ, 検索日期：1月 15, 2026, https://bullmq.io/
42. How to Handle Heavy Background Jobs in Next.js using BullMQ & Redis - YouTube, 検索日期：1月 15, 2026, https://www.youtube.com/watch?v=zpkj9Z-JWKQ
43. Scheduling Background Tasks in Python with Celery and RabbitMQ | AppSignal Blog, 検索日期：1月 15, 2026, https://blog.appsignal.com/2025/08/27/scheduling-background-tasks-in-python-with-celery-and-rabbitmq.html
44. Celery and Background Tasks. Using FastAPI with long running tasks | by Hitoruna | Medium, 検索日期：1月 15, 2026, https://medium.com/@hitorunajp/celery-and-background-tasks-aebb234cae5d
45. Tame Long Running Tasks with Web Sockets! - Serverless DNA, 検索日期：1月 15, 2026, https://serverlessdna.com/strands/websockets/async-task-runner
46. Implementing reactive progress tracking for AWS Step Functions | AWS Compute Blog, 検索日期：1月 15, 2026, https://aws.amazon.com/blogs/compute/implementing-reactive-progress-tracking-for-aws-step-functions/
47. Debugging Non-Deterministic LLM Agents: Implementing Checkpoint-Based State Replay with LangGraph Time Travel - DEV Community, 検索日期：1月 15, 2026, https://dev.to/sreeni5018/debugging-non-deterministic-llm-agents-implementing-checkpoint-based-state-replay-with-langgraph-5171
48. Auto resuming challenges in langgraph - LangChain Forum, 検索日期：1月 15, 2026, https://forum.langchain.com/t/auto-resuming-challenges-in-langgraph/1657
49. Temporal - Pydantic AI, 検索日期：1月 15, 2026, https://ai.pydantic.dev/durable_execution/temporal/
50. Managing very long-running Workflows with Temporal, 検索日期：1月 15, 2026, https://temporal.io/blog/very-long-running-workflows
51. Boosting machine translation with AI-powered terminology features - ACL Anthology, 検索日期：1月 15, 2026, https://aclanthology.org/2024.eamt-2.13.pdf