Report p.1

Mounir Samite 2026

Table of Contents

1 Problem analysis

The problem presents a computational task that is inherently parallelizable due to the independent nature of processing individual pdf files. The core challenge involves recursively traversing a directory structure, identifying pdf files, extracting text content, and searching for a specific word while maintaining accurate counts and providing real-time progress updates through a gui.

1.1 Decomposition strategy

The problem can be decomposed along two primary dimensions: task decomposition and data decomposition. Task decomposition involves breaking down the workflow into distinct operations:

- directory traversal
- file identification
- pdf text extraction
- word matching
- result aggregation

While data decomposition focuses on partitioning the set of pdf files into chunks that can be processed concurrently. The initial sequential implementation revealed that pdf files constitute independent units of work, as analyzing one file does not depend on the results of analyzing another, making them ideal candidates for parallel processing.

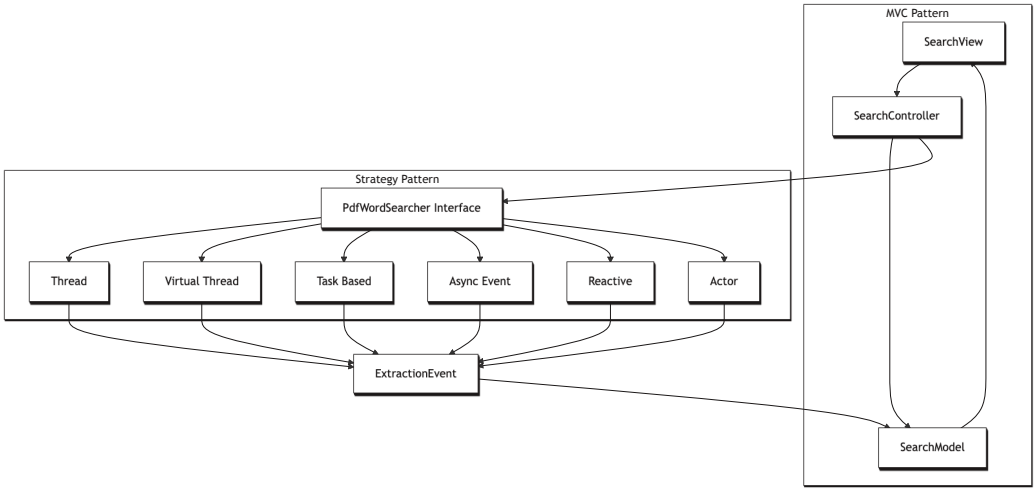
1.2 Coordination and synchronization challenges

Despite the high degree of independence, several coordination points require careful synchronization. The shared counters (total files analyzed, pdfs found, pdfs containing the target word) represent critical sections that must be protected against race conditions when multiple threads update them concurrently. Additionally, the gui updates must be coordinated to ensure consistent and accurate display of progress without overwhelming the event dispatch thread. The start, stop, suspend, resume controls introduce additional complexity, requiring mechanisms to pause and resume worker threads cooperatively while maintaining system state consistency.

2 Solution strategy and architecture

2.1 Highlevel architecture

The project adopts a modular architecture that combines the Model-View-Controller pattern with the Strategy pattern to accommodate six distinct concurrency approaches while maintaining code reusability and separation of concerns.



2.1.1 Core architecture components

The application's structure consists of three primary MVC components that remain consistent across all concurrency implementations:

- **SearchModel**: Encapsulates the application state, including counters for analyzed files, pdf files found, and matches containing the target word. This model serves as the single source of truth and notifies observers of state changes
- **SearchView**: Provides the graphical user interface with input fields for directory path and search word, control buttons (start/stop/suspend/resume), and output boxes displaying real-time progress
- **SearchController**: Mediates between the view and model, handling user interactions and delegating work to the appropriate concurrency strategy implementation

The ModelObserver interface implements the Observer pattern, enabling the view to react to model state changes without tight coupling, ensuring that gui updates remain

hronized with the underlying computational progress.

2.1.2 Strategy pattern

The core of this architecture lies in the strategies package, which encapsulates each concurrency approach as an interchangeable implementation of the PdfWordSearcher interface. This design decision allows the controller to remain agnostic to the underlying concurrency mechanism while supporting six fundamentally different approaches:

- **thread**: Implements traditional multithreaded approach using custom monitors and thread pools
- **virtual threads**: Leverages Java virtual threads for lightweight concurrency
- **task based**: Utilizes Java Executors and Fork/Join framework with task decomposition
- **async event**: Employs Vertx event-loop architecture for asynchronous processing
- **reactive programming**: Applies RxJava reactive streams for data flow management
- **actors**: Uses Akka actor model for message-passing concurrency

Each strategy package contains specialized components tailored to its concurrency model while adhering to a common interface, ensuring seamless swapping between implementations.

2.1.3 Event system

The events package defines a unified event model (ExtractionEvent and ExtractionEventType) that enables consistent communication between Model, Controller and View. This abstraction allows different strategies to report progress using the same event types regardless of their internal implementation details.

2.1.4 Supporting infrastructure

The project structure includes dedicated components for testing and validation:

- generator: Contains scripts and sample pdfs for creating test datasets with varying file counts and directory depths
- pdfs: Houses multiple test scenarios ranging from small sets (3-10 files) to large-scale tests (50,000+ files) with both flat and recursive directory structures

- jpf-workspace: Provides Java PathFinder integration for formal verification of concurrent properties in the thread based implementation

This organizational structure ensures that each concurrency strategy can be developed, tested, and analyzed independently while sharing common infrastructure components, facilitating comparative performance analysis across all six approaches.

2.2 Thread based approach

The thread based implementation leverages traditional multithreading principles with custom monitor synchronization to achieve optimal CPU utilization while maintaining strict control over concurrency mechanisms.

2.2.1 Thread pool size strategy

The solution dynamically determines the optimal number of worker threads based on available CPU resources using the formula Nthreads = Ncpu + 1, where Ncpu is obtained via `Runtime.getRuntime().availableProcessors()`. This approach follows the established heuristic that a system with N processors achieves optimal utilization with N+1 threads.

The ThreadPoolSearch class divides the list of pdf files into equal-sized chunks using a step-based partitioning strategy: step = numFiles / Nthreads. Each Worker thread receives a contiguous range defined by start and end indices, processing files independently without inter-worker communication. The final worker handles any remaining files due to integer division, ensuring all pdfs are processed.

2.2.3 Monitor based Coordination

The custom monitor class implements a synchronization mechanism using ReentrantLock and Condition variables from java.util.concurrent.locks, adhering to the assignment constraint of using only low-level concurrent utilities for monitor implementation. The monitor maintains two critical pieces of state:

- count: Accumulates the total number of pdfs containing the target word across all workers.
- numFiles: Tracks remaining unprocessed files, decremented as workers complete their chunks.

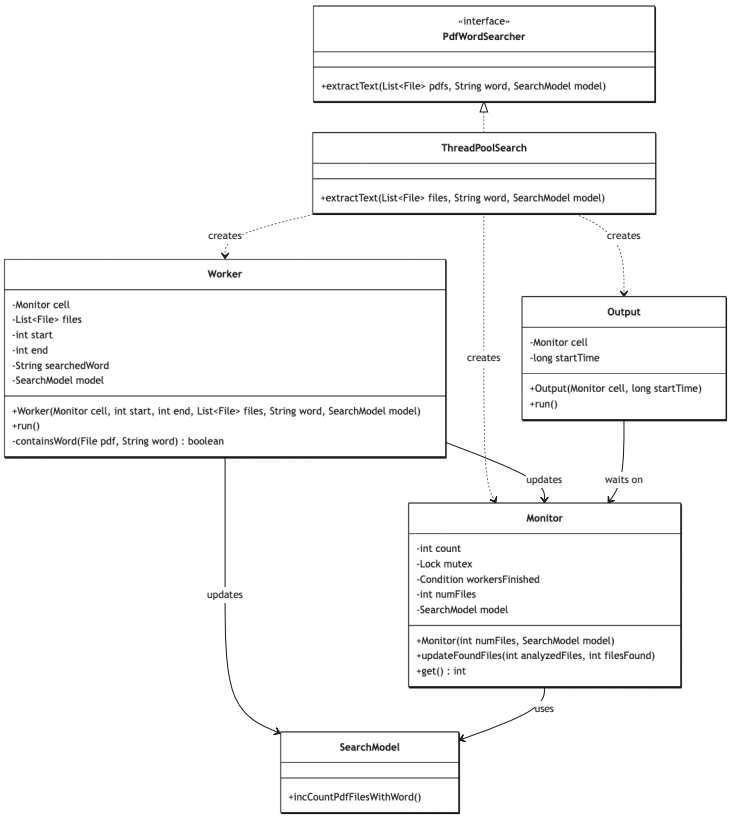
Workers call `updateFoundFiles(analyzedFiles, filesFound)` upon completing their assigned chunk, which atomically updates both counters under mutex protection. When the last worker finishes (numFiles == 0), the monitor signals the waiting Output thread via the workersFinished condition variable.

2.2.4 Worker thread implementation

Each worker thread `extends Thread` and processes its assigned file range by iterating through pdf files, extracting text using Apache pdfBox's pdfTextStripper, and performing string matching. The worker immediately updates the shared SearchModel when a match is found via `model.incCountPdfFilesWithWord()`, enabling real-time updates. After processing all assigned files, the worker reports its results to the monitor, decoupling individual progress updates from final aggregation.

2.2.5 Output thread and results collection

The Output thread implements a separate waiting mechanism that blocks on the monitor's get() method until all workers signal completion. This design separates result collection from computation, allowing the main thread to remain responsive while workers execute in parallel. The output thread also measures total execution time from job start to completion, providing performance metrics.


«interface»
PdfWordSearcher
+extractText(List<File> pdfs, String word, SearchModel model)
ThreadPoolSearch
+extractText(List<File> files, String word, SearchModel model)
creates
creates
creates
Worker
-Monitor cell
-List<File> files
-int start
-int end
-String searchedWord
-SearchModel model
+Worker(Monitor cell, int start, int end, List<File> files, String word, SearchModel model)
+run()
-containsWord(File pdf, String word) : boolean
Output
-Monitor cell
-long startTime
+Output(Monitor cell, long startTime)
+run()
updates
waits on
Monitor
-int count
-Lock mutex
-Condition workersFinished
-int numFiles
-SearchModel model
+Monitor(int numFiles, SearchModel model)
+updateFoundFiles(int analyzedFiles, int filesFound)
+get() : int
updates
uses
SearchModel
+incCountPdfFilesWithWord()

2.3 Virtual Threads

The virtual thread implementation adopts a virtual thread per file strategy where each pdf is processed by its own virtual thread.

2.3.1 Implementation Structure

The solution creates a **monitor** to coordinate all virtual threads, initialized with the total number of files to process. A **virtual thread executor** manages the lifecycle of all worker threads automatically through Java's resource management. Additionally, a separate **output thread** is created but held in reserve until all worker threads begin their execution.

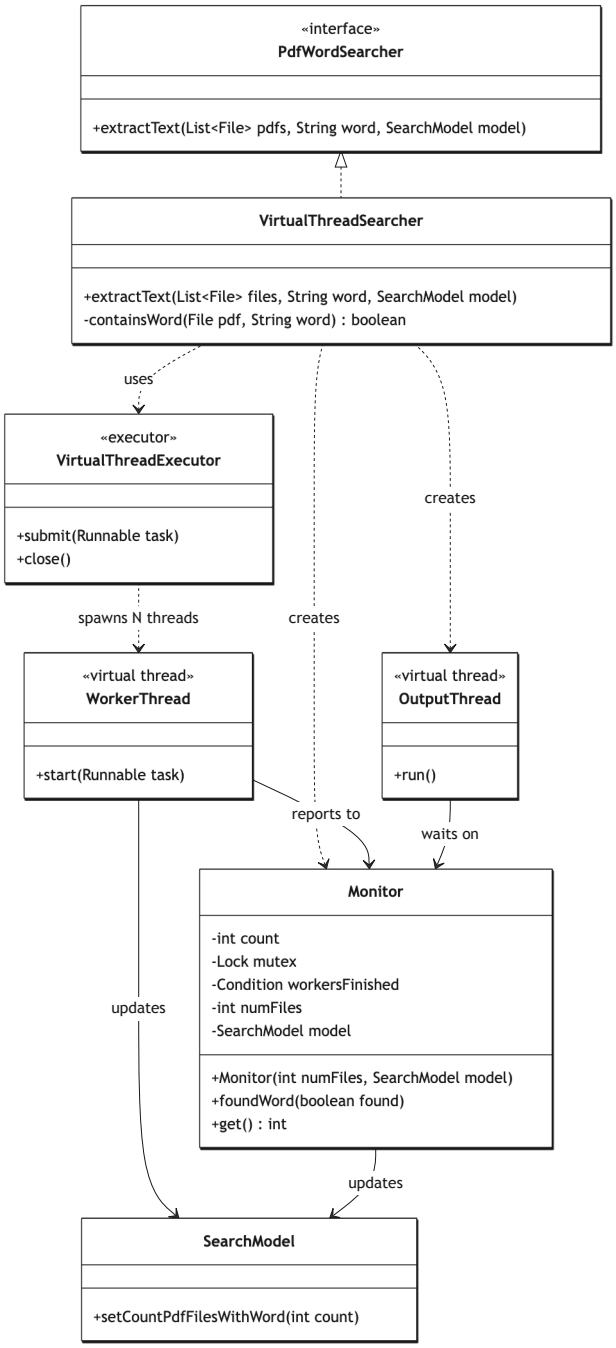
The processing logic iterates through the entire list of pdf files and submits each one as an independent task to the executor. For each file, a new virtual thread is spawned with a unique identifier for debugging purposes. Each virtual thread independently loads its assigned pdf, extracts the text content, searches for the target word, and reports whether a match was found to the shared monitor. Once all tasks are submitted, the output thread begins execution and the main program waits for it to complete before returning

2.3.2 Monitor synchronization

The monitor uses explicit lock and condition variable mechanisms rather than traditional synchronized blocks, which is essential to prevent virtual threads from being pinned to their carrier threads during blocking operations. When each thread completes processing its file, it calls a method that updates the match counter if the word was found, decrements the remaining work counter, and signals when all files have been processed. The output thread blocks on the monitor until it receives the completion signal, then retrieves and returns the final count.

During the computation, every time a new file contains the word, the model is updated.

This architecture exploits the lightweight nature of virtual threads to achieve massive parallelism, creating as many virtual threads as there are pdf files, without overwhelming system resources.


«interface»
PdfWordSearcher
+extractText(List<File> pdfs, String word, SearchModel model)
VirtualThreadSearcher
+extractText(List<File> files, String word, SearchModel model)
-containsWord(File pdf, String word) : boolean
uses
creates
creates
«executor»
VirtualThreadExecutor
+submit(Runnable task)
+close()
spawns N threads
«virtual thread»
WorkerThread
+start(Runnable task)
«virtual thread»
OutputThread
+run()
reports to
waits on
Monitor
-int count
-Lock mutex
-Condition workersFinished
-int numFiles
-SearchModel model
+Monitor(int numFiles, SearchModel model)
+foundWord(boolean found)
+get() : int
updates
updates
SearchModel
+setCountPdfFilesWithWord(int count)

2.4 Task based approach

The task-based implementation uses Java's ForkJoin framework to decompose the problem hierarchically, mirroring the recursive structure of the directory tree itself.

2.4.1 Hierarchical task decomposition

The solution begins by constructing a complete representation of the directory structure before processing begins. The tree representation recursively captures subdirectories and pdf files at each level, creating a hierarchical data structure that mirrors the file system. This upfront construction enables the ForkJoin framework to understand the complete workload structure and optimize task distribution.

The tree builder traverses each directory, classifying entries as either subdirectories (which are recursively processed) or pdf files (which are collected as leaf nodes). This separation allows the framework to spawn different task types: directory scanning tasks for structural traversal and word search tasks for actual file processing.

2.4.2 ForkJoin execution model

The coordinator creates a ForkJoin pool and submits the root in the directory task, which initiates the recursive decomposition. Each directory scanning task examines its assigned directory node and performs a two phase forking strategy.

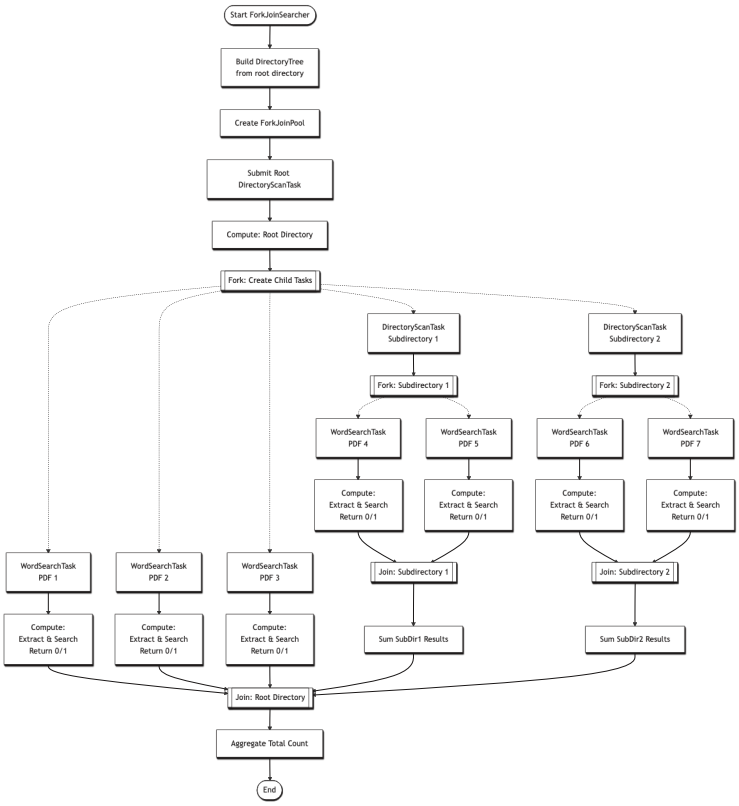
In the first phase, the task creates and forks child directory tasks for each subdirectory, enabling parallel exploration of the directory tree. In the second phase, it creates and forks word search tasks for each pdf file in the current directory. All forked tasks are collected in a list for subsequent joining.

After forking all subtasks, the directory task enters a joining phase where it waits for each child task to complete and accumulates their results. This fork-join pattern creates a recursive decomposition where work is divided (forked) down the tree and results are aggregated (joined) back up.

2.4.3 Leaf (word search) task process

Word search tasks represent the atomic units of computation in this approach. Each task receives a single pdf file reference, extracts its text content, searches for the target word,

and returns either 1 for a match or 0 for no match. When a match is found, the task immediately updates the shared model to provide constant UI feedback



2.5 Async event approach

The async event implementation uses the Vert.x framework, which provides an event loop architecture for non-blocking asynchronous processing.

2.5.1 Verticle based architecture

The solution creates a custom verticle, which serves as the deployable unit of concurrent execution in Vert.x. The main searcher class configures a Vert.x instance with a worker pool sized to match available CPU cores, then deploys the pdf search verticle onto this runtime. The verticle encapsulates the entire search logic within its methods, receiving the list of pdf files, target word, and model reference through its constructor. When the verticle starts, it initiates the asynchronous processing pipeline.

2.5.2 Event loop and worker pool

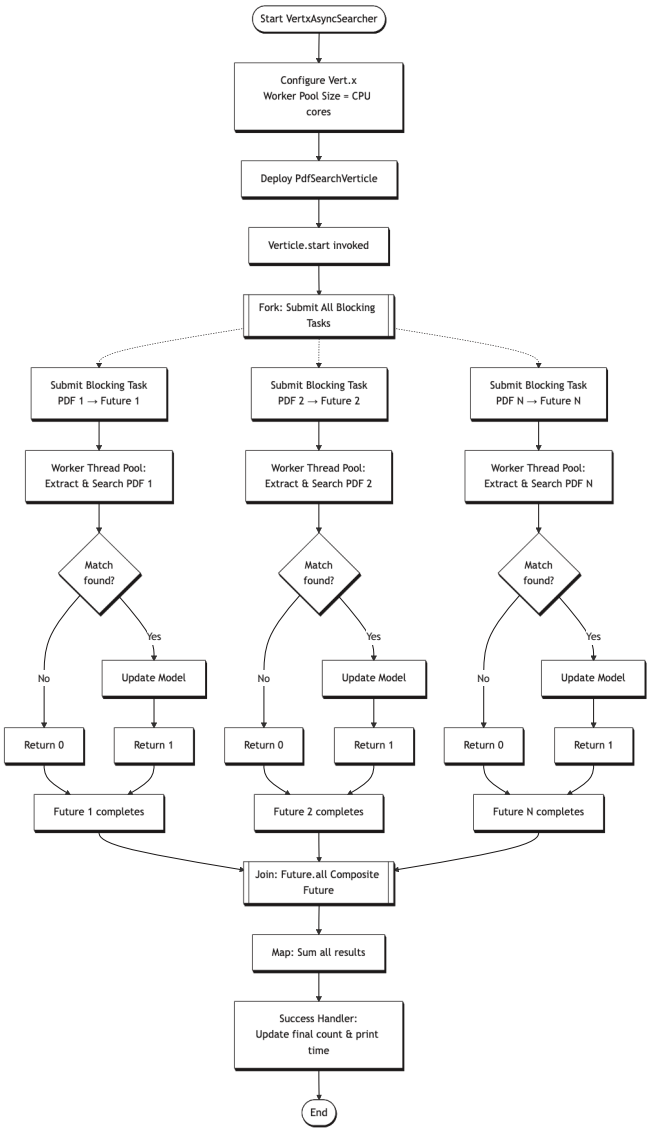
For each pdf file, the verticle submits a blocking task that extracts text and searches for the target word. These tasks execute on worker threads from the pool, preventing the event loop from blocking. Each blocking operation immediately returns a Future object representing the eventual completion of that task.

2.5.3 Future Composition and aggregation

All individual file processing futures are collected into a list as they are created. The framework then composes these futures using a composite future that completes only when all individual futures have finished. When the composite future completes, a mapping operation iterates through all results, summing the match counts and updating the model with the final total. This composition is declarative, with success handlers attached to process results when they become available.

2.5.4 Non blocking result handling

The result handling follows a callback based pattern where success handlers are registered to execute when futures complete. This approach ensures the event loop remains free to handle other events while waiting for blocking operations to finish. Timing information is captured at the start of processing and compared against completion time to measure total execution duration.


Start VertxAsyncSearcher
Configure Vert.x Worker Pool Size = CPU cores
Deploy PdfSearchVerticle
Verticle.start invoked
Fork: Submit All Blocking Tasks
Submit Blocking Task PDF 1 → Future 1
Submit Blocking Task PDF 2 → Future 2
Submit Blocking Task PDF N → Future N
Worker Thread Pool: Extract & Search PDF 1
Worker Thread Pool: Extract & Search PDF 2
Worker Thread Pool: Extract & Search PDF N
Match found?
Match found?
Match found?
No
Yes
No
Yes
No
Yes
Update Model
Update Model
Update Model
Return 0
Return 1
Return 0
Return 1
Return 0
Return 1
Future 1 completes
Future 2 completes
Future N completes
Join: Future.all Composite Future
Map: Sum all results
Success Handler: Update final count & print time
End

2.6 Reactive programming approach

The reactive programming implementation uses RxJava to model pdf processing as a data stream.

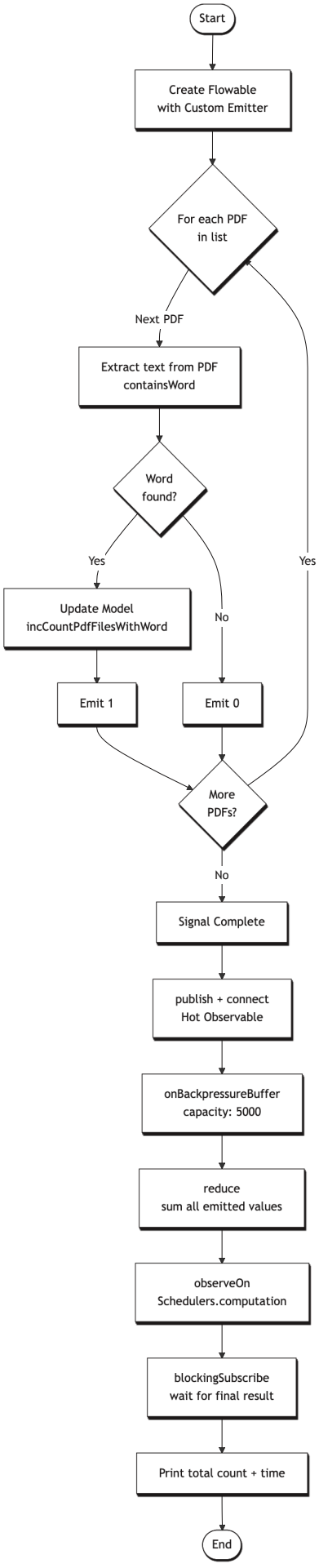
2.6.1 Hot observable stream creation

The solution creates a hot observable stream by first defining a flowable source that emits processing results. The flowable is constructed using a custom emitter that iterates through all pdf files sequentially, processing each one and emitting integer values: 1 for files containing the target word, 0 otherwise. The emitter handles the complete lifecycle: it processes all files in a loop, emits results as they become available, signals completion when all files are processed, and propagates any errors encountered during execution. A **buffer backpressure strategy** is configured to handle situations where the consumer cannot keep up with emitted items. After creating the flowable, it is converted to a hot observable using the publish operator and immediately connected.

2.6.2 flow control

The main processing pipeline applies a sequence of reactive operators to transform the stream. First, a backpressure buffer with a capacity of 5,000 items protects against overflow when the producer emits faster than the consumer can process. The reduce operator aggregates all emitted values into a single sum, effectively counting total matches across all pdfs. This reduction happens asynchronously as values flow through the stream. The pipeline concludes with a blocking subscription that waits for the final reduced value, executing success and error handlers when the stream completes.

Unlike the final aggregation performed by the reduce operator, the model is updated immediately within the emitter loop whenever a match is found. This update strategy provides real-time UI feedback through incremental model updates while the reduce operator handles final result computation.


Start
Create Flowable
with Custom Emitter
For each PDF
in list
Next PDF
Extract text from PDF
containsWord
Word
found?
Yes
No
Update Model
incCountPdfFilesWithWord
Emit 1
Emit 0
More
PDFs?
Yes
No
Signal Complete
publish + connect
Hot Observable
onBackpressureBuffer
capacity: 5000
reduce
sum all emitted values
observeOn
Schedulers.computation
blockingSubscribe
wait for final result
Print total count + time
End

2.7 Actor approach

The actor based implementation uses the Akka framework to model pdf processing as independent actors communicating through asynchronous message passing.

2.7.1 Actors

The solution creates an Akka actor system named `PdfCounter` that serves as the runtime environment for all actors. Two actor types are instantiated: a single analyzer actor that processes all pdf files and maintains the match count, and a requester actor that retrieves the final result and terminates the system.

The analyzer actor is created first and receives its reference, which serves as its mailbox address for incoming messages. After all processing messages are sent, the requester actor is created and sent a query message to retrieve the accumulated count.

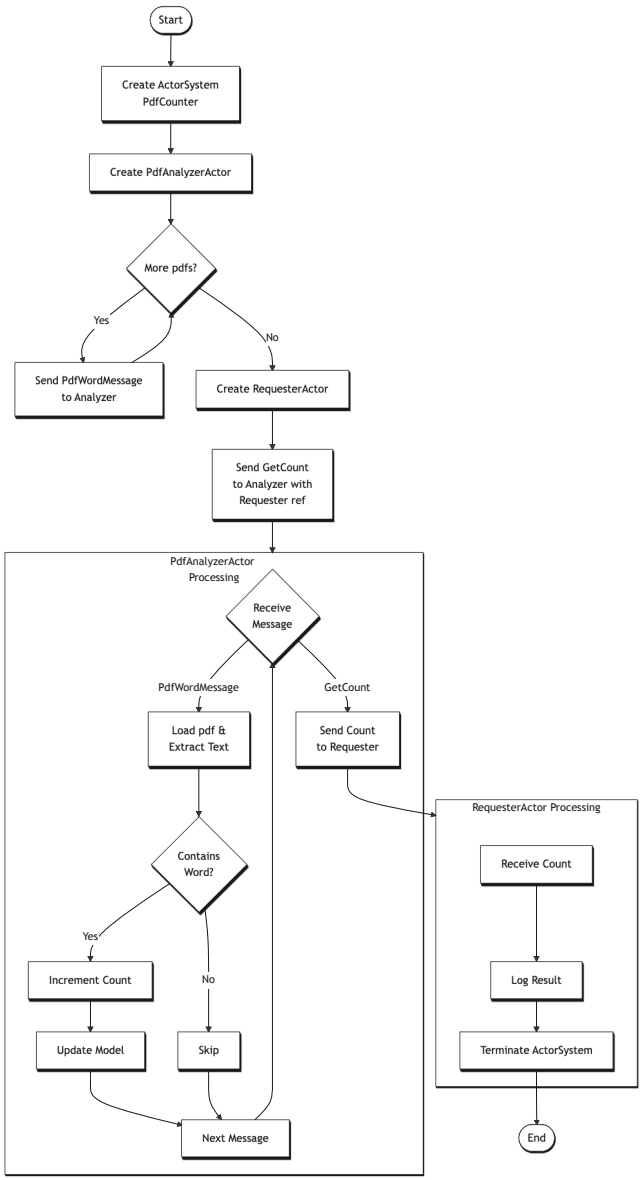
2.7.2 Messages

The implementation defines two message types as immutable data classes. The processing message encapsulates a pdf file reference, the search word, and the model to update, bundling all necessary information for a single file analysis task. The query message is a simple marker class that signals the actor should respond with its current count.

Messages are sent using the tell method, which provides asynchronous delivery without blocking the sender. For processing messages, no sender reference is provided since no response is expected. For the query message, the requester actor's reference is provided so the analyzer knows where to send the final count.

2.7.3 Actor behavior

The analyzer actor defines its behavior through a receive builder that pattern-matches incoming messages. When it receives a processing message, it executes the pdf text extraction, searches for the word, increments its internal count if a match is found, and updates the model. When it receives a query message, it responds by sending its accumulated count back to the sender using the sender reference automatically captured from the incoming message context.


Start
Create ActorSystem PdfCounter
Create PdfAnalyzerActor
More pdfs?
Yes
No
Send PdfWordMessage to Analyzer
Create RequesterActor
Send GetCount to Analyzer with Requester ref
PdfAnalyzerActor Processing
Receive Message
PdfWordMessage
GetCount
Load pdf & Extract Text
Send Count to Requester
Contains Word?
Yes
No
Increment Count
Update Model
Skip
Next Message
RequesterActor Processing
Receive Count
Log Result
Terminate ActorSystem
End

3 Performance and correctness

3.1 Benchmark methodology

The benchmarks evaluate the six different strategies for pdf word search using a consistent testing framework. Each benchmark follows this protocol:

3.1.1 Setup

- Warmup: before the execution on directories with less than 1000 files to minimize JIT compilation effects
- Iterations: 7 executions per test to calculate average performance (time)
- Baseline: sequential single threaded approach for speedup calculation
- Hardware: benchmarks were run in MacBook M3 pro (12 cores)

3.1.2 Testing folders

- Small: 3 with word out of 10 pdfs
- Medium: 50 with word out of 100 pdfs(flat and recursive)
- Medium: 0 with word out of 1000 pdfs
- Large: 1001 with word out of 10000 pdfs (flat and recurisive)
- Largest: 10000 with word out of 50000 pdfs

3.2 Results

Strategy	**Average Speedup**	**Performance Category**
ForkJoin	5.50	Best
Virtual Threads	3.18	Strong
Thread Pool	2.98	Good

Reactive (RxJava)	1.23	Minimal improvement
Async Event (Vert.x)	1.08	No benefits
Actor (Akka)	0.83	Slower than sequential

Table 3.1

3.2.1 Findings

- **ForkJoin**: delivered the best performance (5.50 average speedup) with exceptional results on recursive directory structures, achieving 22.71 speedup on one large recursive directory test, probably because the discovery strategy was different and more efficient from all the other approaches.
- **Virtual Threads**: excelled on small datasets (6.33 speedup on 3-10 files) due to low thread creation overhead, but performance degraded on larger datasets (1.91-2.71)
- **Thread Pool**: provided stable, predictable performance (2.5-3.7) across all dataset sizes, making it reliable for production use.
- **Reactive, Async Event, Actor**: performed poorly probably due to high framework initialization overhead that overwhelmed the actual work. The Actor model was actually slower than sequential execution on small datasets.

3.3 Model Checking

To verify the correctness of the concurrent implementations, model checking was applied to the Thread Pool strategy using Java PathFinder.

To use java PathFinder was used the lightweight environment provided, which includes a preconfigured JPF installation for JDK 8. The ThreadPool strategy was compiled with JPF's custom classpath and executed with Precise race detector listener to check for data races.

3.3.1 Results

The verification completed successfully with no race conditions, deadlocks, or synchronization errors detected in the ThreadPool strategy. This confirms that the Monitor based synchronization pattern and worker thread coordination are correctly implemented,

ensuring thread safe updates to shared state (file counts, pdf matches) across all possible execution scenarios. Model checking provided formal verification that the concurrent design is correct, complementing the performance benchmarks by ensuring reliability alongside efficiency.