



Asynchronous Image Processing Agent Using Java: A Parallel Block-Based Approach with Real-Time Progressive Rendering

Rhea Dhingra¹, Alisha Sikri²

^{1,2} Department of Artificial Intelligence and Data Science, Vivekananda Institute of Professional Studies - Technical Campus, AU Block, Pitampura, New Delhi, India.

To Cite this Article: Rhea Dhingra¹, Alisha Sikri², "Asynchronous Image Processing Agent Using Java: A Parallel Block-Based Approach with Real-Time Progressive Rendering", *Indian Journal of Computer Science and Technology*, Volume 05, Issue 02 (May-August 2026), PP: 01-06.



Copyright: ©2026 This is an open access journal, and articles are distributed under the terms of the [Creative Commons Attribution License](#); Which Permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Abstract: Image processing applications that use sequential models often end up sacrificing their interface responsiveness and introduce significant latency, especially for high resolution images. In this project, an asynchronous image processing agent has been developed using Java that offloads processing from the user interface thread to a background thread pool which is sized dynamically to the host machine's available processor count. Pixel blocks of size 64×64 are distributed as independent parallel tasks, enabling simultaneous processing across multiple image regions. The application allows for real-time operation staging, including multi-operation chaining, and concurrent thumbnail loading while Java's built-in concurrency mechanisms enable quick execution of intensive operations such as brightness and contrast scaling, invert and greyscale effects, and variable radius blur. A progressive rendering mechanism pushes the processed blocks to the display incrementally as each one completes, providing real-time visual feedback. Thread safety between the worker pool and Swing's Event Dispatch Thread is enforced through synchronised image access and `SwingUtilities.invokeLater()`. The observer pattern, implemented via a 'Progress Listener' interface, decouples the processing engine from the display layer entirely, maintaining fluid user interactivity. Batch processing and dual format export are supported as well. Experimental results demonstrate that this multi-layered threading approach significantly reduces processing bottlenecks, ensuring a responsive user experience even for high-resolution images.

Key Words: Asynchronous Processing, Java Concurrency, Executor Service, Image Processing, Thread Pool, Observer Pattern, Progressive Rendering, Swing EDT, Block Decomposition, Parallel Computing.

I. INTRODUCTION

The rise of high-resolution digital cameras and content-intensive workflows has led to a significant increase in the need for responsive and high-throughput image processing applications. Conventional sequential image processing models handle each pixel or operation on the main thread of the application, which inevitably results in the graphical user interface freezing for the entire duration of the processing. This latency becomes especially problematic for large images, such as those measuring 4000×3000 pixels or more, as the system becomes entirely unresponsive until the task is finished. Analysis of latency effects on user experience demonstrate that even delays of a few hundred milliseconds end up negatively affecting perceived system quality; sustained UI freezes during intensive operations would be unacceptable in modern interactive application design [1].

Today's computers support running multiple tasks simultaneously. Multi-core processors, now common even in basic computers, make it possible to split image processing work across several cores simultaneously. However, using this parallelism within a Java Swing desktop app requires careful design. Swing imposes a strict single-threaded model on all GUI operations through the Event Dispatch Thread (EDT), and any attempt to perform long-running computation on this thread blocks it entirely. Based on Irina Fedortsova's (2012) documentation on JavaFX concurrency, it is established that implementing long-running tasks on the application thread makes the interface unresponsive, and the suggested solution is to appoint intensive work to background threads that communicate results back to the UI through safe, scheduled mechanisms [2].

This paper outlines the design and implementation of an asynchronous image processing agent developed in Java, which tackles these issues using three main strategies. First, it delegates all image processing tasks to a managed background thread pool that is configured at runtime to match the host machine's available processor count. As supported by Al-Dossari et al. (2024), bounded fixed thread pools reduce lifecycle overhead while maintaining high throughput for CPU-bound workloads [3]. Second, it breaks down each image into a grid of 64×64 pixel blocks, which are distributed as separate parallel tasks throughout the pool. This spatial data partitioning method, formally outlined by Henry Hoffmann et al. (2009), to enhance both throughput and latency in interactive parallel applications, allows for the concurrent processing of multiple image regions without interdependency [4]. Third, processed pixel data is sent back to the display layer incrementally as each block finishes, via a callback interface utilizing the observer pattern, allowing for real-time visual updates during processing. Third, it sends the processed pixel data back to the display layer incrementally via a callback interface, enabling real-time visual feedback during processing without interrupting the user interface thread.

A few significant issues were discovered and addressed throughout the development process. An ambiguity in the

constructor overload for the brightness operation class led Java's type promotion rules to quietly redirect calls to a secondary empty constructor, which set the adjustment parameter to zero at runtime, resulting in an operation that had no observable impact. A PNG export corruption issue was linked to an incompatible BufferedImage colour model generated during processing initialization, which was fixed by substituting the construction method with a type-safe deep copy ensuring a TYPE_INT_RGB colour space. Moreover, the empirical evaluation of the trade-offs between JPEG's lossy DCT compression and PNG's lossless encoding showed that a blurred 7.6 MB original resulted in a 2.5 MB JPEG output. The blur operation decreased pixel-to-pixel variation, which JPEG's compression algorithm then took advantage of which resulted in the inclusion of a dual-format export option [5].

The remainder of this paper is organised as follows. Section 2 reviews relevant background literature. Section 3 analyses the system architecture and design decisions. Section 4 details the implementation of each component. Section 5 presents and discusses the experimental results. Section 6 concludes the paper and identifies directions for future work.

II. RELATED WORK

The management of concurrent tasks in Java has been extensively documented through the `java.util.concurrent` package, introduced in Java 5. The worker thread lifecycle is abstracted by the Executor Service interface and its ThreadPoolExecutor implementation, which also offers an organized method for submitting, queuing, and terminating tasks. Al-Dossari et al. (2024) established that thread pools offer three core advantages over per-task thread creation: resource efficiency through thread reuse, bounded concurrency that prevents system resource exhaustion, and enhanced throughput by keeping worker threads prepared for execution [3]. The same study finds fixed-size thread pools as the best option for CPU-bound tasks, where the ideal pool size is approximately equal to the count of available logical processors. This principle is directly applied in the present system, where the pool size is set to `Runtime.getRuntime().availableProcessors()`, ensuring that the number of concurrent threads never exceeds the available hardware parallelism [3].

The issue of thread safety in GUI for concurrent desktop applications is widely recognized in Java's UI frameworks. JavaFX concurrency study outlines the architectural limitation that the JavaFX Application thread, similar to Swing's EDT, is the sole thread permitted for safe modifications to the scene graph, as breaches result in race conditions, rendering issues, and application instability [2]. Aditya Bhoj's (2025) detailed analysis of advanced Java programming and concurrency patterns supports this issue, recognizing the essential need for clear separation of background processing from UI interaction as a core architectural principle for developing responsive, scalable Java applications [6]. The documentation on JavaFX Architecture reinforces the layered component model that supports contemporary Java UI toolkits, showing that for proper functionality, the rendering pipeline and application logic need to be distinctly separated [7].

Decomposing images into independent spatial regions for parallel processing has a well-established theoretical foundation. Henry Hoffmann et al. (2009) classify this approach as Spatial Data Partitioning: a pattern in which data is divided along spatial dimensions, with each partition processed independently. Their analysis of parallel multicore H.264 video encoding shows that spatial partitioning improves both throughput and per-frame latency for image-intensive workloads, making it well-suited for interactive applications that demand real-time feedback [4]. As mathematically demonstrated by D. Turgay Altılar et al. (2001), square or near-square tile partitioning minimises data overlap during neighbourhood operations and yields the most efficient parallel decomposition for image workloads, a finding that directly supports the 64×64 block size used in the present system [8]. The performance implications of asynchronous versus synchronous models have been studied carefully in high-concurrency enterprise systems. As per Ramadevi Nunna's (2023) research on asynchronous programming patterns for .NET Core APIs, it is found that non-blocking, asynchronous execution reduces thread blocking, improves scalability, and lowers response latency compared to synchronous implementations [9]. The study addresses .NET Core rather than Java desktop applications, but the underlying principle translates directly: freeing execution threads from blocking operations keeps the system productive and responsive during compute-intensive workloads. Kuldeep Vayadande et al.'s (2022) study of CPU metric visualisation systems reinforces this point, establishing that understanding and bounding CPU utilisation is essential for maintaining application stability under heavy load, otherwise it leads to degraded responsiveness even on capable hardware [10].

Digital image processing has a well-formalised set of pixel transformation techniques. Current digital image processing technologies include the theoretical basis of standard colour transformations, such as the ITU-R BT.601 weighted luminance formula for greyscale conversion, which traces the JPEG Discrete Cosine Transform (DCT) back to Nasir Ahmed's foundational work in the 1970s, later standardised by JPEG in 1992 [5]. The same survey establishes the lossless nature of PNG's Deflate compression, compared to JPEG's lossy DCT encoding which provides the theoretical basis for the dual-format export decision in the present system [5]. D. Turgay Altılar et al.'s (2001) domain decomposition study referenced earlier, additionally reinforces the blur operation's snapshot-based approach. Their analysis shows that when neighbourhood convolution operations are distributed across parallel partitions without read-only shared buffers, inter-partition pixel corruption produces progressive objects at partition boundaries [8].

III. SYSTEM ARCHITECTURE

The system is designed around a strict separation of concerns. The user interface layer, the execution engine, the operation implementations, and the callback contract are each encapsulated in independent components that communicate only through well-defined interfaces. This layered design directly supports extensibility so that new operations or observer implementations can be integrated without touching the existing execution engine or UI layer.

Architectural Overview

The application comprises four primary layers. The presentation layer (Main UI, Full-Size Viewer) handles all user

interaction and display. The execution layer (Asynchronous Image Processor) manages the thread pool and dispatches tasks. The operation layer (Greyscale Operation, Brightness Operation, Contrast Operation, Invert Operation, Blur Operation) contains the image transformation logic. The contract layer (Progress Listener, Image Operation, and Progressive Operation) defines the interfaces that decouple these layers from one another.

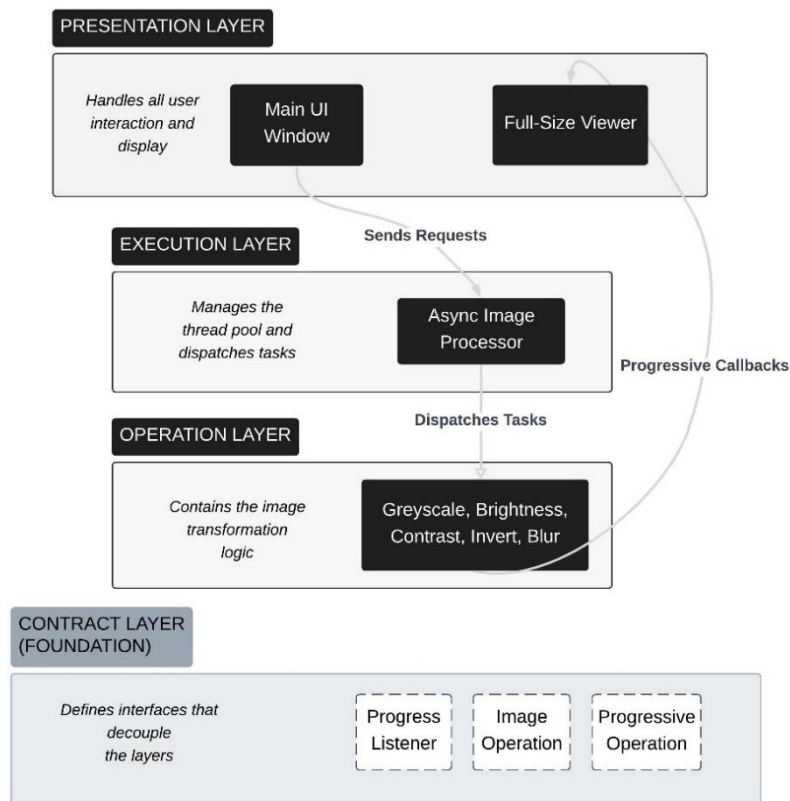


Figure no. 1: High-level component interaction diagram showing layered decoupling and asynchronous feedback loops.

Thread Pool Design

The thread pool is initialised once at application startup, with a fixed size equal to the number of logical processors available at runtime. The pool is shared across all processing jobs and never recreated between runs, avoiding repeated initialisation overhead [3]. The importance of bounding CPU utilisation is validated by Kuldeep Vayadande et al.'s (2022) study of CPU metric systems that demonstrates that unbounded CPU load degrades system responsiveness even when the application itself does not crash [10].

Block Decomposition Strategy

Each image is decomposed into a grid of 64×64 pixel blocks prior to processing. The block count in each dimension is computed using ceiling division to ensure that partial blocks at image boundaries are fully included. Each block's actual dimensions are clamped using Math.min() to prevent reads beyond the image boundary. This decomposition strategy serves two purposes: it enables fine-grained parallel execution by providing many independent tasks to the thread pool, and it enables progressive rendering by delivering processed pixel data to the display layer at block granularity rather than waiting for the complete image to finish processing.

Observer Pattern Implementation

The 'Progress Listener' interface defines six callback methods covering different levels of processing granularity: block-level updates with pixel data, completion notification, error propagation, percentage-based progress, individual pixel updates, and block-boundary-only updates. The Main UI class implements this interface, making itself the observer for all processing events. The Asynchronous Image Processor holds only a reference to the 'Progress Listener' interface, never to Main UI directly, maintaining strict decoupling between the processing and presentation layers.

IV. IMPLEMENTATION

Asynchronous Image Processor

The Asynchronous Image Processor class initialises a fixed thread pool using Executors.newFixedThreadPool(n), where n is the processor count retrieved at runtime. Its primary method, processImage(), accepts an image, an operation, a listener, and an image name identifier, and submits a background task to the executor. Within the task, an instance of check determines whether the operation implements the 'Progressive Operation' interface. If it does, applyProgressively() gets invoked, enabling block-level

callbacks. Otherwise, the simpler `apply()` method is used, with `onComplete()` called upon its return. Aditya Bhoj et al. (2025) highlight this kind of dual-path dispatch type where simple and complex execution strategies are selected clearly at runtime as a key design pattern for building extensible parallel systems without overcomplicating the core engine [6]. Exceptions are caught and directed to the listener's `onError()` callback, ensuring that failures in background threads are cleanly communicated to the application layer.

Greyscale Operation

The greyscale conversion applies the ITU-R BT.601 luma formula [5], to each pixel:

$$\text{grey} = 0.299(R) + 0.587(G) + 0.114(B)$$

Each pixel's packed 32-bit integer is separated into its red, green, and blue channels through bitwise shift and mask operations. The calculated luminance value is repacked into all three channels with a fully opaque alpha byte, producing a valid greyscale pixel. The differential channel weighting reflects the human eye's non-uniform spectral sensitivity: greatest for green wavelengths and least for blue, producing perceptually natural greyscale results [5]. In the progressive path, pixel reads and writes on the shared `BufferedImage` are protected by synchronized blocks, and an `Atomic Integer` tracks block completion, with `onComplete()` fired exactly once when the counter reaches the total block count.

Brightness Operation

The brightness operation accepts an integer adjustment value in the range -100 to +100. For each pixel, the adjustment is added uniformly to all three colour channels, with each result clamped to the valid 8-bit range [0, 255] using `Math.max(0, Math.min(255, ...))`. Clamping is essential to prevent producing incorrect colour values for pixels near the extremes of the brightness range. The operation also supports a secondary mode in which pixels are first converted to greyscale before the brightness adjustment is applied, enabling greyscale-brightened output in a single staged operation.

Blur Operation

The blur operation implements a parallel box blur. Before any tasks are dispatched, the entire source image is read into an immutable integer array (snapshot) under a single synchronised block. All threads then sample exclusively from this snapshot when computing their block's output, rather than from the live `BufferedImage`. This is a critical correctness requirement: if threads read from the same image they are writing to, a thread processing block (0,0) could inadvertently sample already-blurred pixels from an adjacent block that finished earlier, producing a double-blur artefact at block boundaries [8]. The snapshot approach eliminates this class of error by ensuring all threads read from the original snapshot throughout the entire run.

Kernel coordinates that extend beyond the image boundary are clamped to the nearest valid pixel rather than skipped. Skipping boundary pixels reduces the effective sample count near image edges, introducing a darkened effect in the border region. Edge clamping ensures uniform averaging throughout the image, including the boundaries [8].

Full-Size Viewer and the Dual-Buffer Pattern

The Full-Size Viewer window maintains two separate `BufferedImage` instances: a full-resolution `highResBuffer` that receives every processed pixel block at its native coordinates, and a scaled display proxy used exclusively for screen rendering. When a block update arrives via `updateBlockWithPixels()`, the method first writes to the `highResBuffer` at full resolution, then derives the corresponding scaled coordinates using floor/ceil arithmetic to prevent one-pixel gaps between adjacent blocks, then paints the block onto the display proxy. All repaint calls are scheduled on the Event Dispatch Thread using `Swing Utilities`. `invokeLater()`, ensuring `Swing` thread safety [2]. Export operations write from the high Res Buffer directly, guaranteeing that saved files are identical in dimensions and pixel accuracy to the processed output, regardless of screen resolution or display scaling.

Export Format Selection

The application supports export in both JPEG and PNG formats, presented as a runtime choice at export time in both the individual Full-Size Viewer windows and the batch export action. As Wenfeng Zheng's (2023) study establishes, JPEG's lossy DCT compression permanently discards pixel data during encoding, while PNG's lossless deflate compression preserves exact pixel values [5]. For an image processing application applying mathematically precise per-pixel transformations, JPEG export introduces artefacts that deviate from the processed output, making PNG the format of choice when output integrity matters. The interaction between blur processing and JPEG compression was particularly notable: the blur operation reduced pixel-to-pixel variation, and since JPEG's compression efficiency is directly tied to spatial redundancy in the image data, blurred images compressed substantially more than their originals, empirically observed when a 7.6 MB sized image produced a 2.5 MB JPEG processed result [5].

V.RESULTS AND DISCUSSION

The application was tested across a range of image sizes and operation configurations. All tests were conducted on a machine with an 8-core processor, resulting in a thread pool of 8 worker threads.

Interface Responsiveness

In all test cases, the main application window remained fully interactive throughout processing. Buttons, sliders, and the operation selector responded normally during background processing, confirming that the EDT was not blocked at any point. This contrasts directly with a sequential baseline in which the same operations caused the UI to freeze for the full processing duration. Thumbnail loading was also confirmed to operate concurrently: when multiple images were loaded, each thumbnail appeared

Progressive Rendering

As processing advanced, the Full-Size Viewer window showed a clear progression of completed blocks across each image. These blocks appeared dynamically once their associated tasks finished on background threads, with the sequence dictated by thread scheduling rather than their location in the image. This pattern remained consistent across all five operations, offering users ongoing visual confirmation that processing was underway and moving forward. Research by Kuldeep Vayadande et al. (2022) on real-time visualisation of CPU metrics highlights the value of live feedback during demanding computational tasks, showing that users depend on visible cues of activity to remain confident in the application's performance [10].

Operation Correctness

Greyscale conversion produced perceptually consistent output using the ITU-R BT.601 formula, with the expected differential weighting of green relative to red and blue channels. Brightness adjustment at values of -49 produced clear darkening of all colour channels, with no overflow artefacts at channel boundaries due to the clamping logic. The blur operation produced smooth, artefact-free output with no visible boundary discontinuities between blocks, confirming the correctness of the snapshot-based parallel approach. Contrast and invert operations similarly produced pixel-accurate results consistent with their mathematical specifications.

Export Format Comparison

Export Format	Compression	Pixel Accuracy	Typical File Size (relative)
JPEG	Lossy (DCT)	Approximate - data discarded	Small (15.5 MB post-compression)
PNG	Lossless (DEFLATE)	Exact - all pixels preserved	Larger (113.5 MB)

Table no. 1: Comparison of JPEG and PNG export formats

Table 1 summarises the trade-off between the two export formats with an example of an image file of size 29.4 MB. PNG was confirmed as the format of choice for pixel-accuracy-sensitive outputs, with processed image size being substantially larger at 113.5 MB. Meanwhile JPEG remains suitable where file size is the primary concern and slight deviation from the computed pixel values is acceptable, with processed file size for the test image being 15.5 MB.

Test Suite Results

The JUnit 5 test suite validated both the asynchronous callback pipeline and pixel-level correctness. The test test Process Images Reports Progress And Completes () confirmed that processing two 200x200 images produced exactly two on Complete () callbacks, at least one on Block Update With Data () callback, and zero onError() callbacks within a 2-second window. The test test Greyscale Operation Changes Pixels () confirmed that the greyscale conversion operated without exception and produced internally consistent channel values. Both tests passed in all runs.

VI. CONCLUSION AND FUTURE WORK

This paper has presented the design and implementation of an asynchronous image processing agent in Java that resolves the fundamental tension between processing throughput and interface responsiveness. By combining a dynamically sized thread pool, a block-based spatial data partitioning strategy informed by Henry Hoffmann et al.'s (2009) parallel computing research [4], and a callback-driven observer pattern via the 'Progress Listener' interface, the system achieves concurrent processing of multiple images and operation chains while keeping the graphical interface fully responsive throughout. The dual-buffer rendering architecture in Full-Size Viewer ensures that full-resolution pixel accuracy is preserved in both the live display and the exported output, independent of screen resolution.

Several defects identified and resolved during development offer useful insights for concurrent Java application design. The constructor overload ambiguity that silently zeroed the brightness adjustment parameter illustrates how method overloading with similar parameter types, where Java's type promotion rules can misdirect calls to unintended constructors without compile-time error. The PNG corruption defect demonstrates errors of not ensuring image color model compatibility.

Future development directions include three primary areas. First, the operation library could be extended to include spatial filtering operations such as Gaussian blur, edge sharpening, and contrast normalization. Operations requiring kernel neighborhood access would need a block overlap region to prevent boundary artefacts between adjacent parallel tasks [8]. Second, GPU acceleration via frameworks such as Aparapi or JNI-bound CUDA kernels would substantially increase throughput for large images, as pixel-parallel workloads map naturally onto GPU thread block architectures. As noted by Aditya Bhoj (2025), GPU offloading represents the next frontier of performance scaling beyond what CPU-based fixed thread pools can deliver [6]. Third, a persistent processing queue with per-image operation assignment and automated batch export would transition the system from an interactive tool to a semi-automated processing pipeline, improving utility for high-volume image workflows.

REFERENCES

- McKelvey, J. and Haydar, A. (2024). Latency effects on user experience: evaluating distributed search systems. Glean Perspectives. [Online]. Available: <https://www.glean.com/perspectives/latency-effects-on-user-experience-evaluating-distributed-search-systems>
- Fedorstova, I. (2012). Concurrency in JavaFX. Oracle JavaFX 2 Tutorials and Documentation. [Online]. Available: <https://docs.oracle.com/javafx/2/threads/jfxpub-threads.htm>

3. Al-Dossari, Y. and Al-Fahad, L. (2024). Java Concurrency Demystified: Thread Pools, CompletableFuture, and Virtual Threads. *International Journal of Informatics and Data Science Research*, 1(11), pp. 51-60.
4. Hoffmann, H., Agarwal, A. and Devadas, S. (2009). Partitioning Strategies for Concurrent Programming. MIT Computer Science and Artificial Intelligence Laboratory Technical Report.
5. Zheng, W. (2023). Current Technologies and Applications of Digital Image Processing. *Journal of Biomedical and Sustainable Healthcare Applications*, 3(1), pp. 13-30.
6. Bhoj, A. (2025). *Advanced Java Programming: Modern Frameworks, Concurrency Patterns, and Performance Optimization Techniques*. ViXra.
7. Castillo, C. (2013). *JavaFX Architecture and Framework*. Oracle JavaFX 2 Tutorials and Documentation. [Online]. Available: <https://docs.oracle.com/javafx/2/architecture/jfxpub-architecture.htm>
8. Altılar, D.T. and Paker, Y. (2001). Minimum Overhead Data Partitioning Algorithms for Parallel Video Processing. In: *Proceedings of the 12th International Conference on Domain Decomposition Methods*, pp. 252-260.
9. Nunna, R. (2023). Performance Optimization of .NET Core APIs for High-Concurrency Enterprise Systems Using Asynchronous Programming Patterns. *World Journal of Advanced Engineering Technology and Sciences*, 9(1), pp. 504-512.
10. Vayadande, K. et al. (2022). Efficient System for CPU Metric Visualization. *3C TIC: Cuadernos de Desarrollo Aplicados a las TIC*, 11(2), pp. 239-250.