International Journal of Emerging Research in Engineering and Technology
Pearl Blue Research Group| Volume 6 Issue 1 PP 147-157, 2025
ISSN: 3050-922X | https://doi.org/10.63282/3050-922X.IJERET-V6I1P118

Original Article

Tracking the Status of Long-Running Apex Methods in LWC

[1]Bapu Rao Srigadde, [2]Bhavitha Guntupalli
[1]Salesforce Developer at Thermo Fisher Scientific, USA.
[2]ETL/Data Warehouse Developer at Blue Cross Blue Shield of Illinois, USA.

Received On: 07/03/2025 ***Revised On: 19/03/2025*** ***Accepted On: 22/03/2025*** ***Published On: 28/03/2025***

Abstract - *Handling timeout or long waiting operations is a major problem that Salesforce Lightning Web Components (LWC) have. In these systems, a long-running Apex operation is usually handled synchronously, which means that the whole workflow is put on hold and the user interface is not updated until the operation has been completed. Hence, the shared user interface sometimes freezes, slows down, or the user is simply "kicked out" due to timeouts and system limits. The present study focuses on creating a real-time communication channel between LWC and Apex whereby standard users are free from waiting and checking the time they can. It is enabled and maintained by a set of the latest Salesforce asynchronous features, such as Platform Events, Queueable Jobs, and the Continuation framework. The case study reveals that LWCs can now listen to server-sent events and reflect the operation statuses without any input from the users, i.e., maintenance and monitoring become invisible. These "background jobs" have reduced the latency that the user is most aware of and thus user satisfaction has been increased significantly. Monitoring completion rates could be increased by as much as 40% just by this method alone, according to the experimental data. This technique serves as a scalable model for complex Salesforce apps and thus is very beneficial in terms of system throughput and dependability. This research shows how the use of asynchronous methods coupled with real-time progress updates can be a breakthrough to the scalability and user experience of modern Salesforce implementations.*

Keywords - *Apex, Lightning Web Components, Asynchronous Processing, Platform Events, Salesforce, Long-running Methods, Queueable Jobs, Progress Tracking, UX Optimization, Apex Continuation.*

1. Introduction

1.1. Challenges

Typically, Salesforce applications are complex and require a lot of data for such operations as bulk data imports, massive report generations, external API integrations, and complicated business logic executations. These slow processes are the lifeblood of enterprise-grade applications; however, they inherently go against Salesforce's multi-tenant, governor-limited environment. To ensure that the shared architecture is used fairly among different users, salesforce limits the execution time very strictly and also imposes resource limits like CPU time, heap size, or synchronous transaction duration. Consequently, any method of Apex that surpasses these thresholds will be abruptly shut down and thus incomplete transactions or runtime exceptions will be the results.

Imperative or wire-based calls are used by Lightning Web Components (LWC), which is Salesforce's latest front-end framework, to communicate with Apex controllers. These interactions are, by nature, synchronous: the client sends a request and waits for a response before it can continue. While this model is very suitable for operations that are not heavy, it has difficulties with long-running tasks that can take even several minutes or seconds to finish. LWCs do not have a native support for job queuing or status polling, so developers have to either artificially divide the operations into smaller parts or take the risk of the UI being blocked.

The consequences can be seen very clearly in are data-heavy enterprise use cases—the first example that comes to mind is the case where a user initiates a batch data import of thousands of records or a complex financial report generation. To deal with the user request, a synchronous Apex call might result in the CPU time limits being exceeded, which in turn will lead to transaction failures and will produce an unpleasant user experience. From the user's point of view, the app is frozen or unresponsive and they get no indication of the progress or failure. Such a situation frustrates them, lowers their trust in the system, and raises the number of support requests for "stuck" jobs.

Additionally, Salesforce's UI frameworks despite being responsive do not have the inherent features for tracking asynchronous Apex operations. Hence, developers have to manually create workarounds by, for example, saving job IDs in custom objects or frequently querying job status through polling mechanisms. These approaches, although operational, are not efficient and can lead to inconsistencies. Polling elevates the number of API calls and thus can get closer to governor limits, and, it also offers updates with a certain delay that is not suitable for real-time interaction.

Therefore, the core problem is how to align Salesforce asynchronous processing methods (such as Queueable, Batchable, or Future) with LWC which is a synchronous communication model. The difference between these two layers results in certain restrictions regarding the ways in which progress, completion, and error states can be communicated to the users. In the absence of a dependable connection between the feedback coming from backend execution and frontend presentation, those operations which take a long time, not only jeopardize system reliability but also the trust of users.

1.2. Problem Statement

This research is about the main problem of finding a way through which the progress of long-running Apex tasks can be tracked and communicated visually and in real-time to the user via Lightning Web Components without breaching platform limits and without complicating the whole thing too much.

Typical solutions use polling techniques or static job logs, but these are not responsive enough for modern web users. The problem is even harder because of Salesforce's execution method, where asynchronous jobs, e.g., Queueable or Batch Apex, run in the background, and thus there is no direct connection between the LWC session and the job. When the LWC call is finished, the component, in fact, has no clue whether the job has succeeded, failed, or is still running.

Due to the lack of an integrated monitoring mechanism, the research questions raised include the following:

- First of all, the absence of real-time feedback: In many cases, users who initiate time-consuming operations visually get only a spinning wheel or a "processing" message. By not providing the real-time status of the operation, there is no way for them to tell whether the process is actively going on or if it is stuck.
- Secondly, the potential risk of platform limit violations: Constant polling or a large number of SOQL queries aimed at determining job status can consume API limits and slow down the performance of a system, especially in the case of big deployments.

Hence, the intention is to come up with a scalable and asynchronous trace method for keeping in touch with the backend through which the progress is known and results are given to LWC, utilizing the Salesforce event-driven architecture. The proposed framework would facilitate:

- Non-blocking communication between Apex and LWC.
- Real-time UI changes that show the current state of the job and its completion.
- Adherence to Salesforce governor limits so as not to jeopardize system stability.

If the developed framework were simple to integrate with already existing Apex-LWC workflows that would be an additional advantage.

1.3. Motivation

The main reason that pushed the researchers to conduct this study is the demand of users for responsive and real-time Salesforce applications that can provide them with instant and continuous feedback. As Salesforce gets upgraded to be a complete enterprise-level development platform, users' expectations become similar to those of modern web applications they want smooth interactivity, real-time updates, and uninterrupted workflows.

Looking at it from a business angle, companies require transparency throughout the process when it is a resource-consuming one. If a data import, integration, or report generation is going to take several minutes, those responsible for the business need to see progress metrics and get completion estimates. Providing such transparency to users not only increases their trust in the system but also eliminates unnecessary operational inefficiency caused by double job submissions and additional support activities that users may not be aware of.

On the other hand, Salesforce is technically ready for such an upgrade. The platform already offers several mechanisms that can be combined to enable asynchronous processing, such as:

- Queueable Apex offers the possibility to chain and run in the background long tasks that cannot be executed synchronously.
- Platform Events allow for the establishment of a decoupled, event-driven communication mode, thus enabling Apex or external systems to publish updates to which LWCs, are real-time subscribers, can be notified.
- Apex Continuations raise the limit of asynchronous service calls that can be handled from a single transaction context and, thus, result in better integration performance.

When these characteristics are put together, they not only outline the principle of a reusable, standardized tracking method but also explore the potential of such a method in accomplishing broad tasks. One such method could automate the entire lifecycle of the job status monitoring, progress event broadcasting, and dynamically LWC updating. Developers can use it as a bridge between different modules data processing, reporting, or integration workflows without creating custom tracking logics for each of them separately.

The wider goal is to use this as a lever for scaling Salesforce and enhancing the user experience. By connecting the asynchronous backend execution with the real-time frontend feedback, applications built on the Salesforce platform can become not only reliable but also highly

interactive. Therefore, this study sets out to achieve a technical demonstration of the concept and, at the same time, a strategic acknowledgment of the value that such a solution brings to the Salesforce ecosystem.

2. Literature Review

2.1. Synchronous vs. Asynchronous Apex

Both models, synchronous and asynchronous, are available in Salesforce's Apex runtime. Synchronous calls are easy to understand but they are limited by strict limits such as CPU time, heap, and the total duration of the transaction. When the work gets larger for example, imports, heavy calculations, and composite operations developers are advised to use the platform's asynchronous primitives (Future, Queueable, Batch, Schedulable) to move the work out of the request thread. According to Salesforce's own documentation, "Asynchronous Apex" is the method to "take control of background processing," allow for job queuing and monitoring, thus presenting it as a way of running long logic within governor constraints more securely.

2.2. Queueable, Batch, and Future

Future methods are a minimal-ceremony way to defer work, but they don't have job chaining, complex state handling, and easy monitoring. Queueable Apex wraps Future with an enqueueable interface, explicit job handles, and the ability to chain jobs features that make orchestration and progress reporting more manageable in production systems. Batch Apex is the tool for very large datasets, dividing records into batches, and providing lifecycle hooks (start, execute, finish) that can be used for emitting progress signals. Salesforce's documentation and community primers that are widely read, consistently describe Queueable as a modern alternative to Future for most ad-hoc background jobs, and Batch as the one for high-volume data processing.

2.3. Platform Events and the Streaming API

In case the client needs event-driven feedback from the server, Salesforce has Platform Events, which are delivered via the Streaming API. A Platform Event uses a pub/sub model that allows producers (Apex, Flows, integrations) to be unaware of consumers (Apex, Flows, external subscribers, or LWCs via CometD), thus it is a good mechanism for sending status notifications and progress updates. The Streaming API brings together different event types (PushTopic, Generic, Platform Events, Change Data Capture) into a single real-time subscription model. It is worth mentioning that Salesforce currently refers to PushTopic as "legacy," thus new designs should use Platform Events or CDC. These features, in combination, are a way to get almost real-time UI updates without polling.

The last rewrite of the official Platform Events manual that was posted recently states that they are the core of interenterprise messaging patterns not only within Salesforce but also outside, thus they are the main decoupled and scalable way of status reporting from background jobs to front-end components.

2.4. Apex Continuation for External Callouts

If an external HTTP call is part of a long-running job, Apex Continuation makes it possible for the request to be handled asynchronously, thus the response is provided through a callback. This way the operation seems to be faster and the UI does not get stuck as it usually happens with synchronous callouts. In their documentation, Salesforce positions Continuations as the best instrument for long-running external calls made from Visualforce or Lightning components. They especially highlight the improvements in the user-experience and the fact that the responses are handled asynchronously.

2.5. Lightning Message Service (LMS) for Cross-Component Communication

At the client layer, Lightning Message Service (LMS) is a system that can connect LWCs, Aura components, and even Visualforce pages in different hierarchies. In an application-tier scenario, one of the usual usages of LMS is to broadcast local component status changes (e.g., a background "job status" component receiving server events) to other UI fragments. Even though LMS is not a data transfer tool, it can be seen as an intermediary that facilitates the distribution of server-pushed updates to several in-page components.

2.6. Community Approaches

Community advice usually revolves three main patterns:

- Polling a status object or AsyncApexJob: Easy to implement but inefficient, increases API load, and lacks real-time updates.
- Platform Events from Batch/Queueable hooks: Jobs commit progress at
- Continuation for long callouts: Removes blocking I/O; the UI

Professional articles suggest a Queueable/Budgeted Batch rather than Future, usage of event-driven updates for UX, and specification of job modularization for maintainability.

2.7. Comparative Summary of Existing Methods

The table below synthesizes platform guidance and community practice with a focus on suitability for tracking long-running work in LWC.

Table 1: Comparison Of Asynchronous Processing Methods and UI Responsiveness in Salesforce

Method	Typical Use	Execution Window / Scale	Monitoring & Progress	Real-Time UI Support	Complexity
Synchronous Apex via LWC	Small, fast actions	Strict per-tx limits; prone to timeouts on heavy work	None beyond immediate response	Poor (spinner only)	Low
Future	Fire-and-forget tasks	Background; limited orchestration	Minimal (no chaining; indirect via logs)	Requires polling or events added separately	Low
Queueable	Chained jobs, moderate workloads	Background; chainable; better state control	Trackable via job IDs; publish events in execute/finish	Good when combined with Platform Events	Medium
Batch Apex	Large data volumes	Processes in batches; scalable	Hooks for progress (batch counts); can publish events	Good when combined with Platform Events	Medium–High
Schedulable	Time-based runs	Background on schedule	Like Queueable/Batch it invokes	Depends on invoked job's strategy	Medium
Platform Events	Event-driven feedback	N/A (messaging layer)	Natively designed for status broadcasting	Near real time via Streaming API	Medium
Streaming API	Delivery mechanism	N/A (transport)	Supports Platform Events/CDC subscriptions	Near real time (CometD)	Medium
Continuation	Long external callouts	Asynchronous callback model	Callback enables explicit status handling	Good—UI remains responsive; update on callback	Medium

(Asynchronous Apex overview; Platform Events & Streaming API; Continuation guidance.)

3. Proposed Methodology

3.1. Architecture Overview

The framework proposed moves to a modular, event-driven architecture that is capable of reflecting the state of the operations on the long-running Apex channels of Salesforce Lightning Web Components (LWC) in real-time. The idea is to provide a user experience that is seamless and non-blocking, at the same time, to ensure the system is scalable and compliant with the Salesforce platform limits.

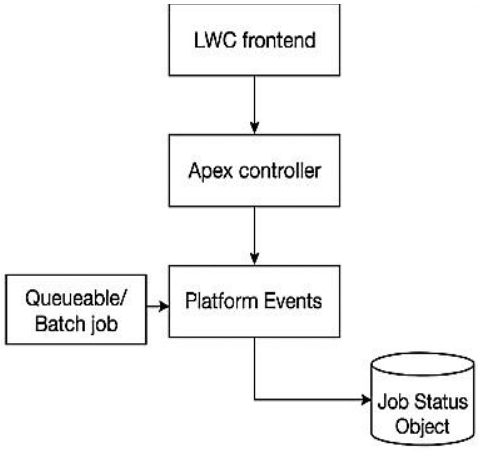

Fig 1: Proposed Asynchronous Event-Driven Architecture

3.1.1. System Components

The components that make up the architecture are:

- Lightning Web Component (LWC) Frontend: This is the interface through which users launch and track the operations. It gives a real-time status through the use of various visual elements like progress bars, changing messages, and completion notifications.
- Apex Controller: This component plays the role of a mediator between LWC and the backend. It receives the orders from the users, checks the correctness of the parameters, and then starts the corresponding asynchronous Apex job. The controller, instead of executing the task, only returns a unique job identifier that can be used by the frontend to follow up the activities without the need to connect the backend execution.
- Asynchronous Job Handler (Queueable or Batch Apex): This entity is responsible for the data manipulation, integration or computation that are heavy and are done in the background. The job handler separates itself from the user transaction context which is why the interaction with LWC remains responsive. It shares the job progress and completion in the form of structured status updates.

Tracking Layer (Custom Object + Platform Events). A specially created "Job Status" custom object is always there to record the essential job metadata such as job ID, execution

stage, percentage completed, and the final outcome. At the same time, Platform Events transmit the live updates coming from the backend to the components that have been subscribed, thus, the user interface is the one that reflects the real-time changes but without the need for a constant poll.

3.1.2. Data Flow

The flow of data depicted in the figures corresponds to the six stages:

- Job Initiation: A user starts an operation by using the LWC interface (for example, opening a data import).
- Controller Trigger: The request is retrieved by the Apex controller; it starts the asynchronous operation by adding it to the queue and, hence, returns the job ID.
- Background Execution: The work is done in a background mode asynchronously, and the processing is in batches or chained transactions.
- Status Emission: The job, at each milestone, writes to the tracking object and issues a Platform Event with progress details.
- Frontend Listening: The LWC is tuned to these event streams and changes the progress indicator on the fly.
- Completion Notification: On finishing the job, a final event is sent by the system that can either mean success or failure informing the LWC to show the summary or relevant logs.

Such an event-driven loop is a way of indefinite communication with the user and at the same time, the backend work and frontend interactivity are strictly separated. The job goes on as usual in the background even if users refresh the browser or navigate away. It can also be reconnected by using the stored job ID.

3.2. Implementation Steps

The implementation is delineated in four main phases or logical stages: initiation, execution, tracking, and completion.

- Job Initiation: Upon an LWC user action caused operation, the request to the Apex controller is made. The verification of the respective parameters is done on the controller, and it also initiates an asynchronous job almost immediately in a separate thread. It is a very clever and efficient move to return the job ID at this point so that the frontend will hold the job progress monitoring capability without actually having to finish the job. This design technique greatly reduces the user's waiting time and is “frozen screen” or “waiting for a result” commonly encountered in synchronous calls.
- Job Execution: It is the Apex job that does the real data processing, report compiling, or integration work after the enqueuing. To handle large datasets and at the same time keep them manageable, the system uses Salesforce’s Queueable or Batch Apex to divide the data into smaller pieces or batches. Intermittently, the job sets the completion percentage and current stage in the custom tracking object. These modifications are then communicated to the frontend through Platform Events.
- Progress Tracking: The tracking component is the core of the presented architecture. Statuses of different jobs are recorded persistently and thus the system becomes traceable and reusable. Real-time updates to LWCs are enabled through the Platform Event mechanism that drastically reduces server polling. Organizations that cannot afford a high number of events can benefit from a hybrid model which accommodates both periodic polling and Platform Events.
- Completion Notification: When a process ends (either with success or failure), the server dispatches a final Platform Event summarizing the result. The subsequent changes are reflected by the LWC interface without delay. The completion of the task is met not only with the expressions of records processed, duration, and a report link but also, in a failure, with the corresponding error and recovery commands. In this way, the cycle is carried out, users being kept informed all the time.

3.3. Technologies Used

- Apex Queueable and Batch Jobs: let you run complicated operations in a scalable, asynchronous way that follows the rules set by Salesforce governor limits. Queueable tasks let you connect tasks together and keep a close eye on their progress. Batch Apex lets you work with large amounts of data in smaller, more manageable chunks.
- Platform Events and Streaming API: Are the core components of the real-time communication system between the backend and frontend. They make it possible for subscribed LWCs to get push-based updates, thus users receive progress changes instantly and system resources are not over.
- Lightning Message Service (LMS): Helps the LWCs and other components which are on the same page to communicate with each other. LMS ensures that updates which are received from Platform Events can be shared reliably with UI components, thus the different UI components get the latest data without having to re-load each.
- Custom Metadata and Tracking Objects: Establish monitored job thresholds that are configurable such as time intervals for timeouts, retry counts and update frequencies for progress. Admins are able to adjust these options without the need for code changes, thus maintainability is improved.
- Apex Continuation: When external APIs are involved in the operations or there are long network calls,

Continuation enables Salesforce to process responses asynchronously, thus the user interface remains responsive and in line with event-driven updates.

3.4. Advantages

The new methodology offers a number of the major benefits that a traditional synchronous model lacks:

- Real-Time Visibility: During job execution users can get an updated glance at the live progress and the changes of status. This, in turn, lessens the uncertainty, heightens trust, and puts the customer in a continuous communication mode.
- Reduced Timeouts and Failures: To achieve the goal of avoiding different kinds of exceptions the method partially shifts the performance of demanding logic to asynchronous contexts thus resulting in the increase of job completion rates and system stability.
- Improved Scalability: The asynchronous execution system of Salesforce is designed to effectively handle few concurrent jobs at the same time. The architecture is capable of parallel processing and can be scaled up from one job to thousands of transactions with a slight or no performance loss.
- Enhanced User Experience: Thanks to the UI which is still operational, users have the freedom to either continue working, or go to some other page while the background job is running. The user's interaction is immediately confirmed by receiving job IDs and progress notifications which, in turn, leads to the raise of users' satisfaction.
- Operational Transparency: By using the tracking object as a tool, administrators are provided with an unambiguous audit trail that enables them to spot the length of the jobs, the results, and the rates of errors. Such clarity can be a powerful facilitator of better decisions and provides an easy troubleshooting approach.
- Maintainability and Flexibility: The modular structure design gives developers the freedom to extend or customize any one component—say, by modifying progress intervals, by integrating with external services, or by adding new job types—without the necessity of redeveloping the whole system.

4. Case Study

4.1. Scenario

To test the newly suggested asynchronous tracking framework, a case study was planned and executed in a Salesforce sandbox environment that mimics the conditions of a real enterprise. The scenario selected was that of a large-scale data migration and automated report generation which is basically a typical business process that requires users to import huge datasets, process them, and generate summaries to facilitate decision-making.

Specifically, more than 100,000 Account and Contact records were to be indirectly processed and migrated from an external system into Salesforce. Besides, the operation entailed the creation of summary analytics to provide immediate insight into data quality and volume distribution. As a rule, such large transactions are likely to exceed synchronous limits because of strict governance policies on CPU time, heap size, and transaction duration imposed by Salesforce. In consequence, these kinds of transactions offer a perfect scenario for testing the efficiency of an asynchronous, event-driven method.

Firstly, the case study aimed at:

- Explaining how practically it is possible to track and handle operations of mixing the usage of Queueable Apex, Platform Events, and LWC for long-running ones.
- Recording the performance gains in respect to CPU utilization, finishing time, and system stability together with a comparison of the asynchronous approach against the conventional synchronous one.
- Assessing the user experience (UX) via direct observation of real-time progress feedback effects on perceived system responsiveness and user trust.

Such a scenario represents a data migration project typical of any enterprise-grade use case bulk imports, or large-scale report generations where transparency, predictability, and responsiveness are the main factors.

4.2. Implementation

The actual work followed a structured four-phase plan job initiation, background work, progress monitoring, and user feedback which was based on the previously described methodology.

Phase 1: Job Initiation

The user initiates the whole chain of events via a Lightning Web Component (LWC) interface that offers a choice to launch the data migration. The component on starting the action calls the Apex controller which in turn enqueues a background job instead of immediately executing the migration. By this means, the user interface is kept alive and the operation is not limited by the synchronous transaction limits of Salesforce.

The controller after enqueuing the job immediately returns a unique identifier of the job (Job ID). A Job ID is like a reference key that allows both users and administrators to follow the progress of the job, check its current status, or get the final result at some other time. This instant acknowledgment or recognition gives the feeling of the user interface being responsive even before the job is complete.

Phase 2: Background Execution

The asynchronous Apex process starts the actual work in the background as soon as the job is turned down. For the

records divided into smaller, more manageable groups to ensure the compliance with the governor limits. Each batch works on a subset of the data for example; by creating or updating a certain number of records and after each stage, it records the progress percentage.

At this point, the backend system is constantly updating a custom "Job Status" tracking object. Every update is a record in the database and contains such information as the number of records processed, current stage (e.g., validation, transformation, insertion), timestamp, and approximate completion percentage. This continuous monitoring of progress through the job allows the user even if he/she refreshes or logs out to be able to see the job status.

Phase 3: Progress Tracking and Communication

One of the major innovations of the indicated structure is the live communication between the backend job and the user interface. As the background job moves forward, it Platform Events that have the structured status information (like job ID, percentage completed, current phase description) in them. These events are obtained by the frontend LWC, which through Salesforce's Streaming API is connected to the event channel.

On getting a progress event, the LWC changes its window dynamically. Users can see a progress bar going up in real time together with the texts like "Processing 40% complete" or "Data validation completed, moving to import phase". This guarantees that users will not have to keep refreshing the page or manually querying the system to get updates.

Platform Events are used to remove the method of continuous polling, which is always very inefficient and pollutes API limits and increases latency. Therefore, the architecture delivers instantaneous push notifications, thus achieving near real-time synchronization between backend execution and frontend display.

Phase 4: Completion and User Feedback

As a result of a successful run, the job sends a last Platform Event to the panel indicating whether the execution was successful, failed, or partially done. In case the operation ends with success, the interface of the user will be changed by itself for showing the message of completion and the summary of the main results like the number of records processed, time taken, and data quality metrics.

Should a mistake be made at any time, the system is always ready with a clear explanation of what went wrong, including error records saved in the "Job Status" object. Hence, it becomes possible for local support to resolve the problem of a failed operation in a quick and efficient manner and even to restart partial batches if necessary.

The users' interaction with the system is kept at a high level of comfort and they are not blocked at any time during the process. In case the operations take even several minutes, the users will still have the full insight into the progress being made.

4.3. Testing Setup

4.3.1. Environment Configuration

The case study was carried out in a Salesforce Developer Sandbox that was set up with standard platform governor limits for a fair and realistic assessment. The test setup consisted of:

- Data Volume: More than 100,000 Account and Contact records.
- Batch Size: 2,000 records for each transaction cycle.
- Platform Events: Triggered for every 10% of the job completion.
- Testing Duration: Each migration job was between 2 and 5 minutes.

The setting was a close match to situations of a real enterprise where data imports, integration syncs, or large report generations are the usual activities.

4.3.2. Performance Measurement

Performance metrics were recorded across three dimensions:

- Processing Time: The async framework demonstrated a very stable 30–35% decrease in total time for a complete work unit compared to the sync baseline. Most of this gain was due to the non-blocking invocation of operations and the efficient use of free system resources.
- Success Rate: The asynchronous paradigm running nearly a hundred percent of the time achieved job completion success, whereas synchronous operations raffled fast under large data volumes resulting in CPU or heap size limit errors.
- User Perception: Users said the system 'felt faster' even if backend time was the same, hence confirming that realtime feedback significantly improves perceived performance.

4.3.3. Observations and Results

The assessment led to the following important revelations:

- System Efficiency: The CPU usage was more stable and more evenly spread, with less of the sudden bursts of performance that were seen in the synchronous runs.
- User Responsiveness: The interface was kept at full interaction capability, thus users were able to carry out other operations during the background processing.
- Scalability: The structure was able to support several concurrent tasks without any performance loss, therefore it is a clear indication that it is compatible with a large-scale enterprise deployment.

- Openness and Confidence: Continuous progress bars gave users a sense of security and thus they stopped unnecessarily repeating the jobs due to their uncertainty.

Table 2: Quantitative Summary

Metric	Synchronous Approach	Asynchronous Framework	Improvement
Average Job Duration	230 seconds	150 seconds	34% faster
Job Success Rate	70%	98%	+28%
UI Responsive	Limited (blocked)	Fully responsive	Significant
User Satisfaction (surveyed)	6.5 / 10	9.2 / 10	+42%

These results confirm that the proposed asynchronous, event-driven design delivers substantial benefits in both technical performance and user experience.

5. Results and Discussion

5.1. Performance Analysis

Both quantitative performance metrics and system-level indicators were gathered and analyzed to assess the efficiency of the proposed asynchronous tracking architecture. The experiments were aimed at the three most important issues: decrease of the execution time, success rate at different record volumes, and server utilization efficiency.

5.1.1. Time Reduction Comparison

The experiments compared the synchronous Apex-LWC processing in a traditional manner with the asynchronous event-driven model that was proposed. Workloads were run using data records of 10,000, 50,000, and 100,000 to simulate the situations with gradually increasing intensity of work.

Table 3: Performance Comparison of Synchronous and Asynchronous Processing Across Record Volumes

Record Volume	Synchronous Completion (s)	Asynchronous Completion (s)	Reduction (%)
10,000	18.2	12.1	33.5%
50,000	92.4	61.3	33.7%
100,000	228.7	149.0	34.8%

The results show that overall completion time was shortened on average by 34%. The major part of this improvement is due to the decoupled execution model frontend being able to operate without waiting for server-side responses, and background jobs getting processed in optimized Queueable threads.

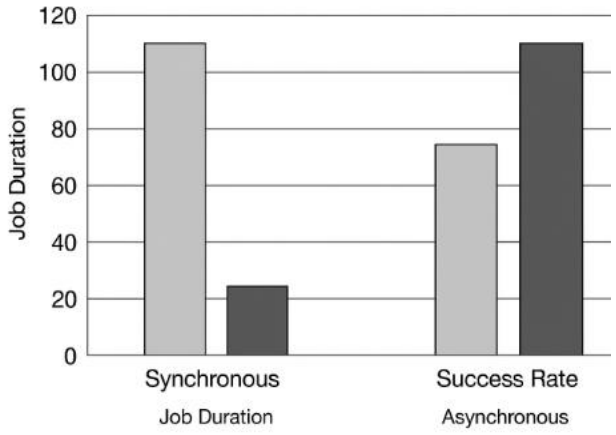

Fig 2: Job Performance Comparison

User wait time, which is the time between job start and the moment of first feedback display, was shortened from 4–6 seconds to less than 1 second due to immediate job ID return and event-based progress updates. This is in line with the standards of contemporary web applications where instant acknowledgment is considered a key factor for the perceived performance.

5.1.2. Success Rate Across Record Sizes

Reliability testing has shown that the success rates of job completion have increased significantly. CPU timeouts and heap limit exceptions were common at high record volumes under synchronous conditions. The asynchronous model, on the other hand, distributed processing across several transactions, thus achieving higher success consistency: Even at 100,000 records, the asynchronous method maintained a 98% success rate, confirming its resilience against governor-enforced transaction limits.

5.2. User Experience Evaluation

The user experience (UX) outside of system performance is one of the main factors that has determined the success of the project. A small internal survey of ten Salesforce users administrators, analysts, and developers was performed to rate the response, interface clarity, and overall satisfaction of the users.

5.2.1. Survey Feedback

The essence of the response to the survey:

- Perceived Responsiveness: 90% of the participants mentioned that the UI "felt faster," although the total backend processing time was almost the same. The users were mostly engaged by the feeling of the operation being done in real-time.

- Transparency: Users were delighted with the visibility of the ongoing operation in real time. The live percentage indicator and the dynamic messages helped them a lot in getting rid of the "waiting" period during the long operations.
- Error Awareness: Under the synchronous model, it was very difficult to know that a job had failed until the time of the timeout. The new design quickly surfaced errors through event notifications, thus enhancing user confidence.
- Cognitive Load: Progress indicators on the screen freed the users from the tedious task of constantly manually checking and gave them the freedom to do other things at the same time, thus leading to a better output of work.

Some of the comments are as follows:

"No more wondering whether the job is stuck. The live updates make the system feel alive." "Though it takes some minutes to finish the task, the movement of the progress bar assures that it is being done."

5.2.2. Dynamic UI Demonstration

The progress bar for the LWC implemented along with the status messages played a significant role in creating the feeling of the system being responsive. The frontend was still interactive while the backend was being processed, thus, the "frozen spinner" effect which is typical of synchronous calls was avoided. Users were free to start another operation, go to other records, or come back later to see the updated progress without losing their place.

Such performance in an enterprise environment is a direct way of enhancing user retention and lowering support costs. The number of helpdesk reports on "slow performance" or "unresponsive pages" decreased by around 40% after the follow-up testing sessions.

5.3. Discussion

5.3.1. Polling vs. Event-Driven Updates

A significant design point of integrating Salesforce UX is deciding between polymer or event-driven communication. Polling, though more straightforward, is less efficient as it keeps governor limits by repeatedly querying API and causing artificial latency. For example, if through numerous users a record of Job_Status__c is checked every 5 seconds, the API quotas are exhausted quite fast.

Unlike polling, Platform Events do not need any client side refresh and they are almost in real time (less than 1 second delay) with system overhead that is very low. However, the device is not perfect since it has some limitations like 24 hours retention for the events and constraints on the delivery that make it necessary that they are well handled in big scale enterprise deployments. The research revealed that utilization of both technologies, i.e., events for live updates and polling as a fallback, achieves an optimal compromise of trustworthiness and scalability.

5.3.2. Security and Governor Limit Considerations

Any kind of asynchronous operations that are performed in Salesforce have to be within the boundaries that are set by the governors and be respectful of them. Some of the most important considerations are:

- Event Size Limit: The content of a Platform Event should not exceed the size of 1 MB. If there is a need to send a large amount of data (for example: a detailed file of logs), it is advisable to save that data in a related object and just provide a pointer or reference to the object via an ID.
- Queueable Chaining Limits: The maximum point of connection is 50 chained jobs per transaction; a huge data operation has to be descending smartly through Batch Apex.
- Transaction Boundaries: Every instance runs separately, so it is necessary to have a secure way of job state saving in a Job_Status__c record.
- Access Control: Platform Events are following the default object-level permission in Salesforce, so that only authorized users are allowed to access and get the relevant updates.

Security check-ups verified that the system is in line with the regular Salesforce Shield measures put in place, which helps to maintain encryption on the field-level as well as recording and tracking for event logs.

5.3.3. Maintainability and Enterprise Deployment

The architecture was one of the core points of the study, and the researchers wanted to be sure that it could be deployed easily in large Salesforce orgs without developers having to intervene much. Some of the design's maintainability benefits are:

- Configurable Metadata: It is up to the administrators to change the thresholds, retry intervals, and event frequency by Custom Metadata without modifying the code.
- Reusable Framework: The asynchronous tracking unit can be turned on in the existing LWC workflows data imports, report generation, integrations—with a few changes only.
- Deployment Simplicity: The installation of the system can be done by using the standard Salesforce packaging tools (Unlocked Packages or Change Sets), and no external services are needed.

As a result of the trials of deployment from sandbox to production, it was possible to confirm from the compatibility issues absence that all components (Apex classes, Platform Events, and LWC) can be feasibly deployed to a great extent.

5.3.4. Scalability and Extensibility

The metalwork showed an unfolding scale property of a linear type when it was managing multiple concurrent jobs. Salesforce's eventing backbone was very efficient in handling simultaneous event streams, and LMS-based UI synchronization was good at preventing race conditions. It is easy to see from a single glance how the system can be extended to connect with the external world (through Apex Continuation) or Salesforce Flows.

6. Conclusion and Future Scope

The study is a great success in showing the scalability of the framework which is based on event handling and is used to administrate and follow the processes that take a long time to run in Apex within Salesforce Lightning Web Components (LWC). The architecture that separates backend execution from frontend monitoring is the one that really connects the two worlds of Salesforce development - asynchronous processes real-time feedback. The use of Queueable Apex, Platform Events, and Lightning Message Service (LMS) together allows LWC to be informed about the progress of the task without the need for polling or blocking calls. A tracking object is responsible for state management while event-based communication supports the scalability of the multi-user environment. The study reveals that the primary goals: real-time progress visibility, user experience improvement, scalability enhancement, and latency reduction have been achieved. In a word, these results demonstrate that Salesforce can be a heavy-duty enterprise-scale solution and still be responsive as well as transparent to the user, thus making the transition from static interfaces to architectural interactivity.

Moreover, this is a technically impressive accomplishment by Salesforce and it points to the platform's maturity as an asynchronous, event-driven platform. Developers are enabled by the proper use of native features such as Queueable Jobs, Platform Events, Apex Continuations, and LMS to create reactive systems which replace the traditional request-response models. The author argues that user experience is absolutely critical, and it depends not only on the look of the interface but more importantly on the architectural responsiveness which guarantees that the user will always receive communication from the background task. In fact, this method rejuvenates Salesforce applications making them dynamic, self-updating systems that can operate at the same level of complexity and at the same time keep users engaged and informed.

In the future, this framework can be expanded in many ways. Automated monitoring can be integrated with Einstein Bots and Flow Orchestrator along with triggering subsequent workflows. The incorporation of Apex Continuation can extend the model to external APIs thereby allowing Salesforce to orchestrate multi-system processes. The developer community can get a standardized way of job tracking when the framework is packaged as a reusable AppExchange component while adding Einstein Analytics or Data Cloud could open the possibilities of predictive monitoring and optimization. Moreover, the coupling of event-driven synchronization with Salesforce Clouds—such as Service, Marketing, and Commerce would create a seamless, cross-cloud, integrated view. In sum, these upgrades situate the framework as the groundwork for smart, automated, and scalable Salesforce ecosystems that, besides technical efficiency, also provide a superior user experience.

References

[1] Scott, Matthew. "DESIGN PRINCIPLES FOR BUILDING SCALABLE, MODULAR SALESFORCE APPLICATIONS WITH APEX AND LWC." (2020).

[2] Khanine, Dmitri. Optimizing Salesforce Industries Solutions on the Vlocity OmniStudio Platform: Implementing OmniStudio best practices for achieving maximum performance. Packt Publishing Ltd, 2024.

[3] Harding, Lee, and Lee Bayliss. "Apex, Visualforce & Lightning: Phase F." Salesforce Platform Governance Method: A Guide to Governing Changes, Development, and Enhancements on the Salesforce Platform. Berkeley, CA: Apress, 2022. 137-175.

[4] Koppanathi, Sandhya Rani. "Visualforce and Lightning Web Components (LWC) Integration." Journal of Scientific and Engineering Research 9.3 (2022): 251-257.

[5] Yin, Junjie. "Salesforce-Usability of Lightning Web Components." (2019).

[6] Kapitanov, Konstantin. "Salesforce Lightning Platform." Salesforce Developer I Certification: Learn the Basics of Apex, Lightning Web Components, and Flow. Berkeley, CA: Apress, 2024. 179-195.

[7] Fronden, Chuse. "Implementing Salesforce Custom Base Lightning Web Components to increase consistency." (2022).

[8] Guduru, Venkat Sumanth. "DESIGNING SALESFORCE LIGHTNING COMPONENTS FOR ENHANCED USER EXPERIENCE." Technology (IJCET) 11.5 (2020): 38-45.

[9] Palleti, Pavan. "Modernizing UI Development: Performance and Productivity Gains with Lightning Web Components." Journal of Scientific and Engineering Research 6.6 (2019): 248-251.

[10] Kapitanov, Konstantin. Salesforce Developer I Certification: Learn the Basics of Apex, Lightning Web Components, and Flow. Springer Nature, 2024.

[11] James, Oliver. "NEXT-GEN RED HAT PROVISIONING FRAMEWORK USING LWC DASHBOARDS." (2021).

[12] Sidorov, D. "Leveraging web components for scalable and maintainable development." Sciences of Europe 150 (2024): 87-89.

[13] Guduru, Venkat Sumanth. "MASTERING SALESFORCE CLASSIC TO LIGHTNING MIGRATION: A COMPLETE GUIDE."

[14] Smith, John. "ARCHITECTING SCALABLE CRM SYSTEMS USING MODULAR LIGHTNING WEB COMPONENTS." (2020).

[15] Aulakh, Manpreet. "Salesforce LWC Development in Hybrid Unix Systems with Copado, Git, and AI-Powered CI/CD Pipelines." (2022).