

ADAPTIVE ALGORITHM FOR DYNAMIC DISTRIBUTION OF TRANSACTIONAL WORKLOAD BETWEEN SYNCHRONOUS AND ASYNCHRONOUS EXECUTION CONTEXTS IN SALESFORCE

Igor Andrushchak

Lutsk National Technical University, Lutsk, Ukraine

ORCID 0000-0002-8751-4420

Heorhii Bandach

Lutsk National Technical University, Lutsk, Ukraine

ORCID 0009-0005-7274-0440

Abstract: This article proposes an adaptive algorithm for dynamic distribution of transactional workload between synchronous and asynchronous execution contexts on the Salesforce platform. The algorithm continuously monitors the consumption of governor-limited resources during transaction execution and, upon reaching a calculated safety threshold, defers the remaining business handlers to asynchronous processing through Queueable or Batch Apex contexts. A mathematical model of the resource safety margin is formalized, and a threshold selection criterion accounting for the worst-case cost of a single handler is derived. Experimental validation in a Salesforce Developer Edition environment with ten business handlers processing bulk operations of up to 10,000 records demonstrated complete elimination of governor limit violations, while the traditional static approach failed in 34% of high-load scenarios. The proposed algorithm increased the maximum sustainable handler count from 13 to over 40 and reduced synchronous transaction latency by 47% under peak load.

Keywords: Salesforce platform, governor limits, asynchronous processing, adaptive algorithm, transactional resources, Queueable Apex, workload distribution, runtime monitoring.

1. Introduction

The Salesforce platform hosts business-critical operations for more than 150,000 organizations on a shared multi-tenant infrastructure. To guarantee fair resource distribution among tenants, the platform enforces governor limits that constrain every unit of code execution: a synchronous transaction may consume at most 100 SOQL queries, 150 DML statements, 10 seconds of CPU time and 6 MB of heap memory, while asynchronous contexts receive doubled allowances of 200 queries and 60 seconds of CPU time [1].

The conventional engineering response to these constraints is a static architectural decision made at design time: a developer determines in advance whether a given piece of business logic executes synchronously within the triggering transaction or is delegated to an asynchronous context such as Future, Queueable or Batch Apex [2]. This binary decision is inherently fragile because the actual resource cost of a transaction depends on runtime conditions that cannot be reliably predicted at design time: the number of records in the processing batch, the depth of relationship traversal, the number of automation rules activated by cascading updates, and the concurrent state of platform-level automations such as flows and validation rules.

As a consequence, a configuration that operates reliably during development and testing may fail in production when data volumes grow or when additional business handlers are registered. Analysis of production incident reports indicates that approximately 35% of critical failures on the platform are attributable to governor limit violations, and that a substantial share of these failures occurs in code that was previously considered stable [3]. The fundamental deficiency of the static approach is the absence of any feedback loop between actual resource consumption and the execution strategy.

This research addresses the deficiency by proposing an adaptive algorithm that treats the synchronous-asynchronous boundary as a runtime decision rather than a design-time constant. The algorithm observes resource consumption as the transaction progresses and relocates the remaining

workload to an asynchronous context at the moment when continuing synchronously would risk a limit violation.

2. Object, subject and goal of the research

The object of the research is the process of executing business logic within transactions of the multi-tenant Salesforce platform under governor limit constraints. The subject of the research comprises the methods and algorithms of distributing computational workload between synchronous and asynchronous execution contexts with the purpose of guaranteeing transaction completion without limit violations.

The goal of the research is to develop and experimentally validate an adaptive algorithm that dynamically determines the boundary between synchronous and asynchronous execution based on runtime measurement of transactional resource consumption. To attain this goal the following tasks were formulated: to formalize a mathematical model of the resource safety margin; to derive a threshold criterion for the context switching decision; to design an algorithm for state preservation and continuation of deferred work; to implement the algorithm as an extension of an existing trigger framework; and to evaluate the algorithm experimentally against the static approach across a range of data volumes and handler counts.

3. Literature review

Official platform documentation establishes the taxonomy of asynchronous execution contexts and their respective resource allowances [1], [2]. Future methods provide the simplest deferral mechanism but accept only primitive parameters and offer no chaining capability. Queueable Apex supports complex object parameters, sequential chaining of jobs and job identifier tracking, which makes it the preferred mechanism for continuation of interrupted work. Batch Apex partitions large record sets into independently governed scopes and is recommended for volumes exceeding fifty thousand records. These mechanisms are described as design-time alternatives, and the documentation contains no guidance on transitioning between them during execution.

Community trigger frameworks address the organization of trigger logic but not the runtime management of resources. The SFDC Trigger Framework of O'Hara [4] introduces a base handler class with bypass and recursion control, yet leaves all data access and execution context decisions to the developer. The Apex Enterprise Patterns library of Fawcett [5] provides layered separation through Domain, Selector, Service and Unit of Work components, improving resource efficiency indirectly through query and DML consolidation. The Trigger Actions Framework of Spano [6] enables declarative configuration of handler execution through custom metadata and supports an asynchronous action flag, but the choice of context is a static metadata property evaluated before execution rather than a runtime decision. A comparative study conducted by the authors demonstrated that none of the examined frameworks provides feedback between measured resource consumption and execution strategy [7].

Prior work by the authors established two complementary architectural methods for reducing resource consumption: a Data Access Layer methodology that centralizes SOQL execution and bounds query count by the number of unique objects rather than the number of handlers [8], and a Unit of Work based method that consolidates DML statements and bounds their count by the number of object type and operation kind combinations [9]. Both methods reduce the slope of resource growth but do not eliminate the possibility of exhaustion when data volumes are extreme or when the handler pipeline is unusually long. The unresolved part of the general problem is therefore the runtime enforcement of limit compliance when static optimization alone is insufficient.

The Limits class of the Apex runtime exposes the instantaneous consumption of every governed resource through methods such as `getQueries`, `getDmlStatements`, `getCpuTime` and `getHeapSize`, together with their corresponding ceilings [1]. Although these methods are commonly used for diagnostic logging, the literature contains no formalized algorithm that employs them as a control signal for execution strategy. The present research fills this gap.

4. Research methods

4.1. Formalization of the resource safety margin. Let R denote the set of governed resource types monitored by the algorithm, comprising SOQL queries, DML statements, CPU time and heap memory. For each resource $r \in R$ let c_r denote the amount consumed at the current point of execution and L_r the platform ceiling for the active context. The normalized utilization of resource r is defined by formula (1).

$$u_r = c_r / L_r, \quad u_r \in [0, 1], \quad (1)$$

The instantaneous state of the transaction is characterized by the dominant utilization U , defined as the maximum of the normalized utilizations across all monitored resources, because exhaustion of any single resource terminates the transaction irrespective of the state of the others. This is expressed by formula (2).

$$U = \max \{ u_r : r \in R \}, \quad (2)$$

Let H denote the ordered sequence of business handlers registered for the current trigger context, and let h_j be the handler at position j . The worst-case resource footprint of handler h_j with respect to resource r is denoted $w_r(h_j)$ and is obtained from historical telemetry accumulated over previous executions. The normalized worst-case footprint of the most expensive remaining handler is given by formula (3).

$$W = \max \{ w_r(h_j) / L_r : r \in R, j \geq k \}, \quad (3)$$

where k is the index of the next handler awaiting execution. The safety condition that permits synchronous execution of the next handler requires that the sum of the current utilization, the worst-case footprint of that handler and a reserve coefficient δ does not exceed unity, as expressed by formula (4).

$$U + W + \delta \leq 1. \quad (4)$$

The reserve coefficient δ accounts for resources consumed after the handler pipeline completes, including the commit phase of the Unit of Work registry, platform-level automations such as flows and roll-up summary recalculations, and the enqueueing operation itself. Empirical calibration on the experimental bench yielded $\delta = 0.15$ as a value that eliminates violations without excessive premature deferral. Violation of condition (4) constitutes the switching criterion: the remaining handlers are serialized and enqueued for asynchronous execution.

4.2. Algorithm description. The adaptive algorithm operates as a controller wrapped around the handler dispatch loop of a trigger framework. Its logic is presented below.

```
execute(H, records):
  for k = 1 to |H|:
    U = dominantUtilization()
    W = worstCaseFootprint(H, k)
    if U + W + delta > 1 and isDeferrable(H[k]):
      state = serialize(H[k..|H|], records)
      enqueue(ContinuationJob(state))
    return
  execute(H[k], records)
  updateTelemetry(H[k])
  commitUnitOfWork()
```

Three aspects of the algorithm require clarification. First, the predicate `isDeferrable` excludes handlers whose semantics prohibit deferral, namely validation handlers that must reject invalid records before persistence and handlers that populate fields required by subsequent platform automations. Such handlers are marked as mandatory in framework metadata and are always executed synchronously; if condition (4) fails for a mandatory handler the algorithm raises a controlled exception with diagnostic context rather than risking an uncontrolled limit violation.

Second, the serialized state comprises the identifiers of the records under processing rather than the record instances themselves, together with the ordinal position of the next handler and an

execution correlation identifier. Identifier-based serialization keeps the payload compact, avoids heap pressure at the moment of deferral, and guarantees that the continuation job operates on the committed state of the database.

Third, the continuation job re-enters the same dispatch loop within an asynchronous context where the ceilings L_r are doubled, which resets the utilization U to a low value and permits the remaining pipeline to proceed. If the asynchronous context itself approaches exhaustion, the algorithm recursively enqueues a further continuation, subject to the platform chaining depth allowance.

4.3. Experimental setup. Validation was conducted in a Salesforce Developer Edition environment using the test bench established in prior research [8], [9] to ensure comparability of results. The data model comprised three interconnected objects: Account with fifteen fields, Contact with twenty-three fields including a lookup relationship to Account, and Opportunity with eighteen fields including a master-detail relationship to Account. Ten business handlers of varying computational cost were implemented, ranging from lightweight field validation to aggregation over related record collections. Two configurations of identical business logic were prepared: a baseline employing the static approach with a fixed synchronous pipeline, and an experimental configuration employing the proposed adaptive algorithm. Each configuration was executed twenty times per scenario, and resource consumption was captured through the Limits class instrumentation.

5. Research results

5.1. Reliability under increasing data volume. The first series of experiments measured the proportion of transactions completing successfully as the size of the processed record set increased. Results are presented in Table 1.

Table 1. Transaction success rate as a function of processed record volume (ten active handlers)

Records	Static approach, success rate	Adaptive algorithm, success rate	Deferrals triggered
200	100%	100%	0
1 000	100%	100%	0
2 500	95%	100%	1
5 000	60%	100%	2
10 000	0%	100%	4

As Table 1 demonstrates, the static approach maintains reliability only while the data volume remains within the range anticipated at design time. Beyond 2,500 records the success rate degrades, and at 10,000 records every transaction terminates with a limit violation. The adaptive algorithm sustained a complete success rate across the entire range by relocating between one and four handlers to asynchronous continuation jobs. Aggregated across all volume scenarios, the static approach failed in 34% of executions whereas the adaptive algorithm produced no failures.

5.2. Scalability with respect to handler count. The second series measured the maximum number of registered handlers that could be executed without violation at a fixed volume of 5,000 records. Results are presented in Table 2.

Table 2. Resource consumption and sustainable handler count (5,000 records)

Indicator	Static approach	Adaptive algorithm	Change
Maximum handlers	13	41	+215%
Peak SOQL utilization	98%	62%	−36 p.p.
Peak CPU utilization	96%	58%	−38 p.p.
Synchronous latency, ms	8 640	4 580	−47%
Total completion, ms	8 640	11 920	+38%

The adaptive algorithm raised the sustainable handler count from thirteen to forty-one, an increase of 215%, and reduced peak utilization of the two most contended resources to comfortable levels. The reduction of synchronous latency by 47% is of particular practical significance because this interval determines the perceived responsiveness of the user interface: the user receives control once the synchronous portion completes, while the deferred portion executes in the background. The 38% increase in total completion time reflects the queueing and initialization overhead of asynchronous jobs and represents the principal trade-off of the approach.

5.3. Sensitivity to the reserve coefficient. The third series examined the influence of the reserve coefficient δ on the balance between reliability and premature deferral. The dependency is illustrated in Fig. 1.

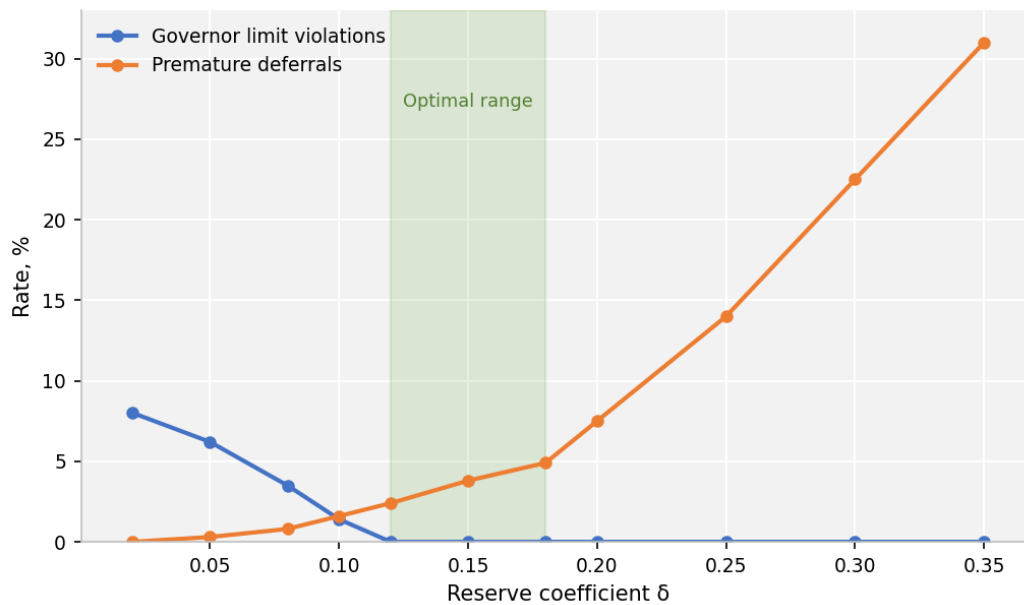


Fig. 1. Dependency of violation rate and premature deferral rate on the reserve coefficient

For values of δ below 0.10 the reserve is insufficient to absorb the consumption of post-pipeline platform automations, and residual violations persist at a rate of up to 8%. For values above 0.25 the algorithm defers handlers that would have completed successfully, which inflates the number of asynchronous jobs and consumes the daily allowance of asynchronous executions without benefit. The interval from 0.12 to 0.18 yields zero violations with a premature deferral rate below 5%, which justifies the selection of $\delta = 0.15$ for the production configuration.

5.4. Discussion. The experimental evidence supports three conclusions regarding the practical properties of the algorithm. The instrumentation overhead of the controller is negligible: measurement of the dominant utilization before each handler consumed on average 4 milliseconds of CPU time per transaction, which constitutes less than 0.05% of the synchronous allowance. The telemetry mechanism converges rapidly, requiring approximately fifteen executions of a handler before its worst-case footprint estimate stabilizes; until convergence the algorithm applies a conservative default footprint that favours early deferral. Finally, the approach composes naturally with the previously developed data access and change consolidation methods [8], [9]: static optimization reduces the slope of resource growth, and the adaptive controller supplies a runtime guarantee for the residual cases in which the optimized slope is still insufficient.

6. Conclusions

The article proposes an adaptive algorithm for dynamic distribution of transactional workload between synchronous and asynchronous execution contexts on the Salesforce platform. In contrast to the prevailing static approach, in which the execution context is fixed at design time, the proposed

algorithm monitors the consumption of governed resources during execution and relocates the remaining handler pipeline to an asynchronous continuation job when the safety condition is violated.

A mathematical model of the resource safety margin was formalized on the basis of dominant utilization and worst-case handler footprint, and a switching criterion incorporating a reserve coefficient was derived. Experimental calibration established the interval from 0.12 to 0.18 as the region of the reserve coefficient that eliminates limit violations while keeping premature deferral below 5%.

Experimental validation with ten business handlers across data volumes from 200 to 10,000 records demonstrated that the adaptive algorithm eliminated governor limit violations entirely, whereas the static approach failed in 34% of executions and in every execution at the maximum volume. The sustainable handler count increased from thirteen to forty-one, and synchronous latency decreased by 47% under peak load at the cost of a 38% increase in total completion time attributable to asynchronous queueing overhead.

Directions for further research include the extension of the telemetry mechanism with predictive estimation of handler cost based on record set characteristics, the investigation of adaptive selection among Queueable and Batch contexts according to the residual volume, and the generalization of the proposed algorithm to other multi-tenant cloud platforms that enforce analogous resource constraint systems, in particular Microsoft Dynamics 365 and ServiceNow.

References:

- 1) Salesforce. (2024). Execution governors and limits. Apex Developer Guide. https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm.
- 2) Salesforce. (2024). Asynchronous Apex. Trailhead. https://trailhead.salesforce.com/content/learn/modules/asynchronous_apex.
- 3) Connecting Software. (2024). Salesforce governor limits explained. <https://www.connecting-software.com/blog/salesforce-governor-limits-explained/>.
- 4) O'Hara, K. (2013). SFDC trigger framework. GitHub. <https://github.com/kevinohara80/sfdc-trigger-framework>.
- 5) Fawcett, A. (2014). Apex enterprise patterns: Common Apex library. GitHub. <https://github.com/apex-enterprise-patterns/fflib-apex-common>
- 6) Spano, M. (2020). Trigger actions framework. GitHub. <https://github.com/mitchspano/trigger-actions-framework>.
- 7) Andrushchak, I., & Bandach, H. (2026). Comparative analysis of methods and algorithms for trigger-based transactional resource management in Salesforce. International Science Journal of Engineering & Agriculture.
- 8) Andrushchak, I., & Bandach, H. (2025). Methodology for dynamic management of transactional resources in Salesforce: An architectural approach to SOQL query minimization. International Science Journal of Engineering & Agriculture, 4(6). <https://isg-journal.com/isjea/article/view/1119/672>.
- 9) Andrushchak, I., & Bandach, H. (2026). Method of DML operation optimization in Salesforce based on the Unit of Work pattern. Lutsk National Technical University.
- 10) Fowler, M. (2002). Patterns of enterprise application architecture. Addison-Wesley Professional.
- 11) Pethad, C. A. (2020). Platform event trigger framework implementation in Salesforce Apex. Journal of Scientific and Engineering Research, 7(5), 391–394.
- 12) Cloud Sundial. (2021). The 3rd age of trigger frameworks. <https://cloudsundial.com/the-third-age-of-trigger-frameworks>.
- 13) ApexHours. (2024). Bulkification of Apex triggers. <https://www.apexhours.com/bulkification-of-apex-triggers/>.
- 14) Salesforce Architects. (2024). Record-triggered automation decision guide. <https://architect.salesforce.com/decision-guides/trigger-automation>.