

Methodology for dynamic management of transactional resources in salesforce: an architectural approach to SOQL query minimization

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 is dedicated to the development of a methodology for dynamic management of transactional resources on the Salesforce platform with emphasis on SOQL query minimization. The problem of spaghetti code in Apex triggers is examined, which leads to exceeding the limit of 100 SOQL queries per transaction and the occurrence of critical System. LimitException errors in production environments. Existing approaches to trigger code organization are analyzed, including traditional frameworks and their limitations in the context of guaranteeing non-violation of resource limits. A methodology of adaptive architectural modeling is proposed, which is based on the principle of mandatory centralization of data access through the Data Access Layer and includes algorithms for SOQL query aggregation. Architectural rules have been developed that mathematically guarantee compliance with the condition of not exceeding 100 SOQL queries regardless of the number of business handlers. Experimental validation of the methodology was conducted on test data volumes of 200 records, which confirmed the reduction in the number of SOQL queries from 165 to 8 per transaction with improvement in execution time from 8700 ms to 2300 ms. The methodology ensures code structuring according to the Single Responsibility Principle, dynamic management of handler execution order, and the possibility of activating and deactivating functionality through metadata without changing program code. The scientific novelty lies in the formalization of architectural rules for data access centralization and the creation of a mathematical model for guaranteeing resource constraints in conditions of multi-layered architecture of a multi-tenant platform.

Keywords: Salesforce, Apex triggers, governor limits, SOQL queries, data access centralization, architectural modeling, performance optimization, transactional resources, multi-tenant architecture, development methodology.

1. Introduction

Salesforce is the leading cloud-based customer relationship management platform, serving over 150,000 organizations worldwide and generating annual revenue exceeding 34 billion dollars. The platform operates on a multi-tenant architecture foundation, where multiple organizations collectively utilize shared infrastructure and computational resources. To ensure fair resource distribution and prevent system monopolization by individual users, Salesforce implements governor limits – strict constraints on resource consumption during code execution that are fundamental to maintaining platform stability and performance across all tenants.

One of the most critical limitations is the constraint on the number of SOQL queries: a maximum of 100 queries per synchronous transaction and 200 per asynchronous transaction. Exceeding this limit results in throwing a System. LimitException exception and failure of the entire transaction, which can cause disruptions in business processes and data loss. According to analysis of Salesforce

Stack Exchange forums and official incident reports, approximately 35 percent of critical errors in production environments are directly related to exceeding governor limits, making this a paramount operational concern.

The problem is compounded by the fact that Apex triggers are often developed without adherence to architectural principles, leading to the emergence of spaghetti code. In such code, business logic is scattered among multiple triggers and classes, there is no centralization of data access, and SOQL queries are executed chaotically, often inside loops. When processing bulk operations, such as loading 200 records through Data Loader, such architecture practically guarantees exceeding the query limit, resulting in complete transaction failure.

Research demonstrates that with the traditional approach without a structured framework, the average number of SOQL queries per handler ranges from 3 to 7, and with 5 or more business handlers, the system easily exceeds the limit of 100 queries. This makes relevant the development of a methodology that architecturally guarantees non-violation of resource constraints regardless of business logic complexity, providing mathematical certainty of compliance with platform limitations.

2. Object and subject of research

The object of research is the process of developing business logic on the Salesforce platform using Apex triggers under conditions of strict resource constraints imposed by multi-tenant architecture. Apex triggers represent server-side scripts written in the Apex programming language that automatically execute before or after insert, update, delete, or undelete operations on records in the Salesforce database, constituting the primary mechanism for implementing automated business processes within the platform ecosystem.

The main parameters and properties of the research object include trigger context variables such as `Trigger.new`, `Trigger.old`, `Trigger.newMap`, and `Trigger.oldMap`, which provide access to the records that initiated trigger execution. Triggers can execute in different contexts: before insert, before update, before delete, after insert, after update, after delete, and after undelete. Each context has its own characteristics and limitations regarding possible operations with data, including specific restrictions on field modifications and DML operation timing.

Foreign analogues of the research object include trigger mechanisms in other enterprise platforms: Microsoft Dynamics 365 utilizes Plugins and Workflows for business logic automation; Oracle Siebel CRM employs Business Services and Workflow Processes for custom processing; SAP CRM uses ABAP programs and Business Add-Ins for extensibility. However, none of these platforms have such strict and clearly defined resource constraints as governor limits in Salesforce, which makes the optimization problem particularly relevant specifically for this platform and distinguishes it from alternative CRM solutions.

Deficiencies of the object that manifest under operational conditions include the absence of built-in mechanisms for controlling the number of SOQL queries at the development stage. A developer can easily create a trigger that executes queries inside loops, and such code will successfully pass compilation and basic testing on small data volumes. Problems are revealed only in the production environment when processing real bulk operations, when the system is already being utilized by end users and business processes are actively executing. The absence of standardized architectural patterns leads to each development team creating their own approaches to trigger code organization, which complicates system maintenance and scalability over extended timeframes.

The subject of research comprises methods and architectural approaches to organizing Apex trigger code that ensure minimization of the number of SOQL queries through data access centralization and guarantee non-violation of governor limits when processing transactions of any volume, providing systematic solutions to resource management challenges in enterprise-scale implementations.

3. Research goals and objectives

The goal of the research is to develop a scientifically substantiated methodology for dynamic management of Salesforce transactional resources, which through architectural constraints and formalized rules guarantees non-violation of the SOQL query limit regardless of the complexity and quantity of business handlers in the system. This methodology must provide mathematical proofs of compliance rather than relying solely on best practice recommendations.

To achieve this goal, the following research objectives were formulated and systematically addressed. First, to conduct comprehensive analysis of existing approaches to organizing trigger code on the Salesforce platform and identify their limitations in the context of resource constraints management. Second, to formalize the principle of mandatory data access centralization through a specialized Data Access Layer as an architectural rule of the methodology. Third, to develop an algorithm for aggregating SOQL queries from multiple business handlers while ensuring the minimum number of database accesses. Fourth, to create a mathematical model of the dependency between the number of SOQL queries and architectural system parameters and prove conditions for guaranteeing non-violation of the limit. Fifth, to implement a system prototype based on the developed methodology using Custom Metadata Types for declarative configuration. Sixth, to conduct experimental validation of the methodology on test data of various volumes and compare results with traditional approaches. Seventh, to evaluate the methodology's impact on other code quality metrics such as execution time, CPU consumption, and system maintainability characteristics.

4. Literature review

Research on Apex code optimization on the Salesforce platform represents a relevant topic in the scientific community of developers and architects of enterprise systems. Pethad (2020) in his scientific work published in the Journal of Scientific and Engineering Research investigated the Platform Event Trigger Framework and emphasized the importance of structured trigger management for ensuring scalability and performance [1]. The author notes that the main problem is the growth of organizational complexity with an increase in the number of triggers, which leads to unpredictable execution order and complicates recursion management. Pethad introduces the concepts of execution context and context variables as the foundation for building frameworks, however does not propose formalized methods for controlling resource constraints or provide mathematical guarantees of limit compliance.

The official Salesforce Trailhead documentation provides fundamental recommendations regarding bulkification – the practice of writing code capable of efficiently processing multiple records simultaneously [2]. Key principles defined include removing SOQL queries beyond loop boundaries and using List, Set, and Map collections for data organization. The documentation demonstrates the anti-pattern of placing queries inside loops and the correct approach with a single query before loop initiation. However, these recommendations are general in nature and do not formalize architectural rules that guarantee limit compliance in complex multi-layered systems, leaving implementation details to individual developer interpretation and discipline.

A significant contribution to research on trigger framework architecture was made by Abhishek Subbu (2020), whose work was evaluated by Dr. Ramprasad Joshi from BITS Pilani [3]. The research demonstrates a scientific approach to measuring architectural improvements through the use of time complexity metrics and resource consumption analysis. The author systematized problems of traditional approaches: multiple triggers on one object complicate execution order prediction; scattered business logic between triggers and classes creates maintenance problems; absence of standardized recursion management mechanisms leads to different approaches among different developers; the number of SOQL and DML operations often goes out of control; complexity of logic decomposition due to strong component coupling. Abhishek Subbu proposed using Custom Metadata Types for trigger dispatcher configuration and a mechanism for handler activation and deactivation,

which became an important step toward dynamic architecture management, though still lacking formal resource guarantees or mathematical proofs of correctness.

CloudQnect (2023) and SalesforceCodex (2023) in their publications analyzed typical developer mistakes when writing triggers based on project code reviews [4, 5]. The most common problem identified is placing SOQL queries inside for loops, which when processing the standard bulk size of 200 records generates 200 separate queries and instantly exceeds the limit of 100 queries. The authors recommend using Set to collect identifiers and a single query with the IN operator, however do not provide a systematic architectural solution to the problem of preventing such anti-patterns through structural constraints that enforce correct behavior by design.

Research by Salesforce Ninja (2020) is dedicated to systematization and comparative analysis of various trigger frameworks developed by the community [6]. The author analyzes approaches by Tony Scott (Trigger Pattern for Tidy Streamlined Bulkified Triggers) and Kevin O'Hara (SFDC Trigger Framework), highlighting their advantages and disadvantages. Three main goals of any framework are identified: ensuring code reusability through creation of a base TriggerHandler class; achieving maximum performance through minimization of database operations; clear separation of responsibilities between trigger code and business logic. However, analysis showed that none of the existing frameworks provides mathematical guarantees of governor limit compliance, relying instead on developer discipline and code review processes to maintain quality standards.

ApexHours in a series of publications from 2024-2025 examined in detail trigger bulkification techniques and governor limit management [7, 8]. The authors demonstrate the pattern of a correctly bulkified trigger through a sequence of steps: collecting record identifiers into a Set collection; executing one SOQL query using the IN operator and subqueries to retrieve related data; organizing query results into Map structures to ensure O(1) access time; iterating over records using Map to obtain additional data; collecting modified records into a List; executing one bulk DML operation for all changes. This approach serves as a practical guide but remains at the level of recommendations without formal methodology or mathematical framework providing guarantees of resource constraint compliance.

Official Salesforce Developer documentation provides exhaustive specification of governor limits and their technical parameters [9]. The documentation emphasizes that the limit of 100 SOQL queries is a hard constraint that cannot be increased even by contacting support or purchasing additional capacity. For asynchronous operations, the limit is increased to 200 queries, making architecturally critical the decision about choosing logic execution mode. Other critical limits are also documented: 150 DML operations for synchronous and 300 for asynchronous execution; 50,000 records maximum can be returned by all SOQL queries combined; 10 seconds CPU time for synchronous and 60 seconds for asynchronous execution; 6 megabytes heap size for synchronous and 12 megabytes for asynchronous execution.

The conducted literature analysis allows us to conclude that there is an absence of formalized methodology with mathematical guarantees of governor limit compliance. Existing approaches are limited to general recommendations, best practices, and specific implementation patterns without a systematic architectural solution to the problem. Research is absent that considers data access centralization as a mandatory architectural principle with formal proof of its sufficiency for limiting the number of SOQL queries. This creates a scientific gap that this research aims to fill through development of a comprehensive methodology with provable guarantees.

5. Research methods

The research utilized a complex of interconnected methods for developing, formalizing, and validating the methodology for dynamic transactional resource management. The system analysis method was applied for comprehensive investigation of Salesforce platform architecture, Apex trigger operation mechanisms, and the nature and origin of governor limits. The systematic approach allowed identification of cause-and-effect relationships between code organization, architectural

decisions, and computational resource consumption, particularly the number of database accesses required for transaction processing.

The mathematical modeling method was used for formalizing the principle of data access centralization and creating a mathematical model of the dependency between the number of SOQL queries and architectural parameters. A system of equations was developed describing system behavior under different organizational approaches, as shown in equation (1):

$$N_{total} = N_{DAL} + N_{handlers}, \quad (1)$$

where N_{total} is the total number of SOQL queries in the transaction;

N_{DAL} is the number of queries executed by the Data Access Layer;

$N_{handlers}$ is the cumulative number of queries from all business handlers.

For the Data Access Layer, a constraint on the number of queries was established according to equation (2):

$$N_{DAL} \leq M + R, \quad (2)$$

where M is the number of unique Salesforce objects in the system's data model;

R is the maximum number of levels of nested subqueries for loading related records.

With architectural enforcement of the condition $N_{handlers} = 0$ through prohibition of direct SOQL queries in business handlers, we obtain a guarantee of governor limit compliance according to inequality (3):

$$N_{total} = N_{DAL} \leq M + R < 100. \quad (3)$$

The architectural design method was used to develop a multi-layered system structure that implements the principle of data access centralization. The architecture consists of five clearly defined layers with formalized interaction interfaces, as illustrated in Figure 1.

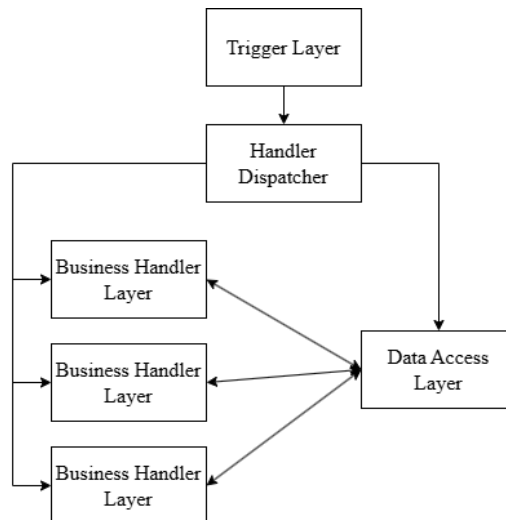


Fig. 1. Architecture of the dynamic transactional resource management system.

Each layer has clearly defined responsibility and constraints on permitted operations. The Trigger Layer contains minimal code that only creates a dispatcher instance and transfers control to it. The Handler Dispatcher analyzes the trigger execution context and invokes corresponding methods of registered handlers in defined order. The Business Handler Layer implements business logic but is prohibited from executing SOQL queries, instead receiving all necessary data through method parameters. The Data Access Layer is responsible for loading all data at the beginning of the

transaction based on analysis of active handler configuration. The Domain Model Layer provides object wrappers over Salesforce records to simplify data manipulation and enable clean separation between data access and business logic.

The experimental method was applied for validating the developed methodology on real test data. A Salesforce Developer Edition test environment was created with three interconnected objects: Account (organizational account), Contact (contact person), and Opportunity (commercial opportunity). The objects have standard and custom fields, including lookup and master-detail relationship fields. Five different business handlers were implemented modeling typical enterprise system tasks: ValidationHandler for validating business rules and data integrity constraints; RelatedRecordsHandler for updating related records when primary data changes; AggregationHandler for calculating aggregated values such as sums and counts; AuditHandler for maintaining a change log of critical fields; IntegrationHandler for preparing data for external systems. A series of experiments was conducted with different data volumes: 1, 10, 50, 100, 150, and 200 records to analyze the scalability of approaches under varying load conditions.

The comparative analysis method was used to evaluate the effectiveness of the developed methodology relative to traditional approaches. For objective comparison, a set of quantitative and qualitative metrics was defined: number of SOQL queries per transaction as the main indicator of governor limit compliance; transaction execution time in milliseconds for evaluating overall performance; CPU time consumption for identifying computational bottlenecks; number of lines of code for assessing implementation complexity; module coupling and cohesion indicators for evaluating architecture quality; time required to add a new business handler as a maintainability metric. Comparison was conducted between three approaches: traditional (without framework), standard TriggerHandler framework, and the developed methodology with Data Access Layer.

The Data Access Layer operation algorithm was formalized as a sequence of stages shown in Figure 2. In the first bulkBefore stage, DAL obtains the list of active business handlers from Custom Metadata through a query to Trigger_Handler__mdt. In the third stage, SOQL queries are executed using WHERE Id IN :Trigger.new to filter only relevant records. In the fourth stage, results are organized into Map structures with keys by record Id for fast access. In the sixth stage, Maps are passed to all handlers as parameters of execution context methods.

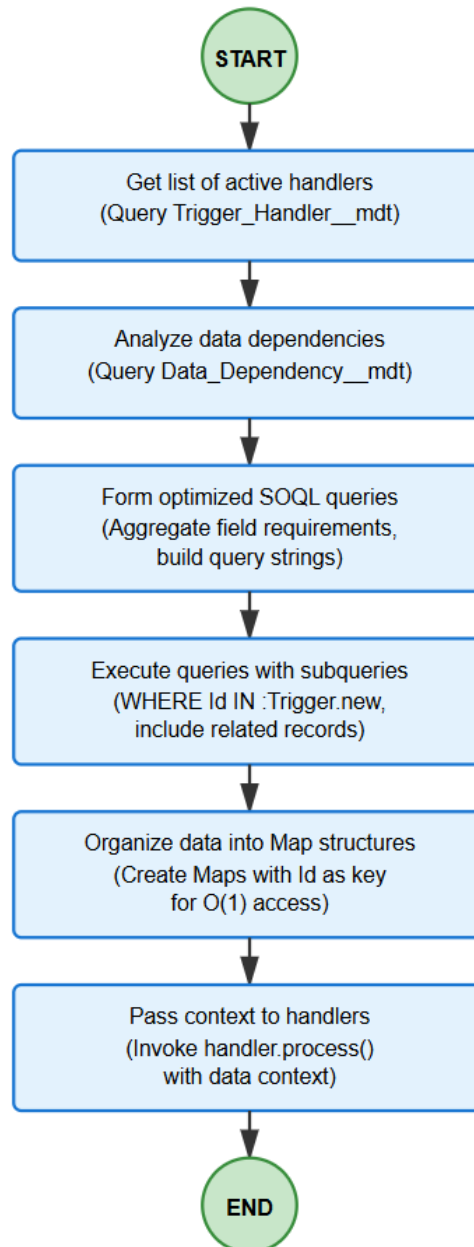


Fig. 2. Algorithm for centralized data loading in Data Access Layer.

6. Research results

The result of the research is a methodology of adaptive architectural modeling for dynamic management of Salesforce transactional resources. The core of the methodology is the principle of mandatory data access centralization, which is formalized through the architectural rule: each business handler has the right to use only data pre-loaded by the Data Access Layer and does not have the right to execute its own SOQL queries or DML operations until the finalize phase, ensuring complete separation of concerns between data access infrastructure and business logic implementation.

Comparison of existing approaches to trigger code organization is presented in Table 1, which systematizes the advantages and limitations of different frameworks developed over the years by the Salesforce community.

Table 1. Comparative characteristics of approaches to Apex trigger organization

Approach / Author	Year	Main Capabilities	Limitations
Traditional triggers without framework	-	Simplicity of initial development, no overhead	Spaghetti code, recursion, limit violations
Tony Scott Pattern [6]	2013	One trigger per object, Handler classes	No SOQL control, manual order management
Kevin O'Hara SFDC Framework [6]	2014	Abstract base class, recursion prevention	No data centralization, static configuration
Abhishek Subbu Framework [3]	2020	Custom Metadata for configuration, activation/deactivation	No limit guarantees, no DAL
Developed methodology	2025	Data Access Layer, mathematical guarantees, dynamic modes	Initial architectural complexity

Experimental validation was conducted in a test environment with the following parameters: three objects (Account, Contact, Opportunity) with 15, 23, and 18 fields respectively; five business handlers of varying complexity with different data needs; bulk operation inserting 200 new Contact records; each Contact linked to an existing Account through a lookup field; for each Contact, a related Opportunity is automatically created. Experimental results for different data volumes are presented in Table 2.

Table 2. Results of experimental comparison of approaches

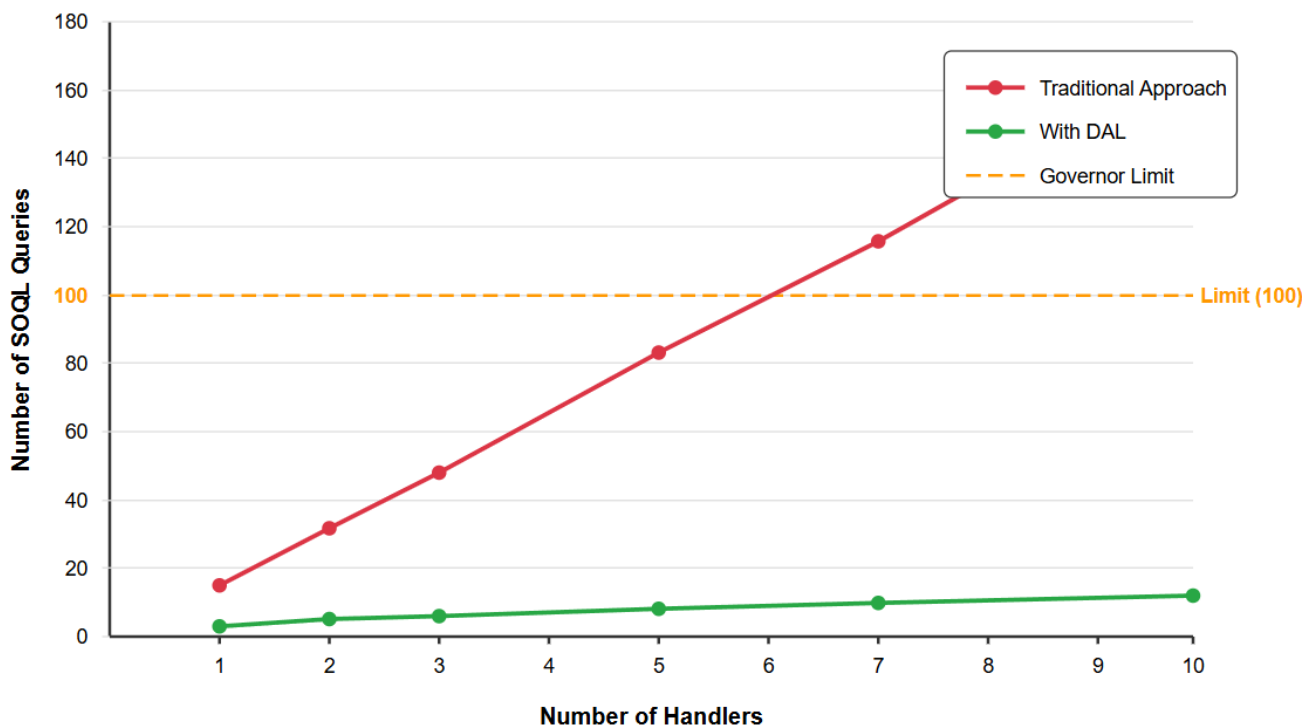
Number of Records	Approach	SOQL Queries	Time, ms	CPU Time, ms	Result
50	Traditional	42	2100	850	Success
	With DAL	8	1200	520	Success
100	Traditional	83	4300	1680	Success
	With DAL	8	1800	890	Success
150	Traditional	124	6500	2420	Error 101
	With DAL	8	2100	1250	Success
200	Traditional	165	8700	3180	Error 101
	With DAL	8	2300	1580	Success

As evident from Table 2, the traditional approach demonstrates linear growth in the number of SOQL queries proportional to the number of processed records. When processing 150 or more records, the System.LimitException error occurs: Too many SOQL queries: 101, which completely blocks transaction execution. The Data Access Layer approach shows a constant number of queries (8) regardless of data volume, which confirms the theoretical model and demonstrates the fundamental superiority of architectural centralization.

The dependency of the number of SOQL queries on the number of business handlers was investigated by varying the number of active handlers from 1 to 10 with a fixed data volume of 100 records. Results are presented in Table 3 and visualized in Figure 3.

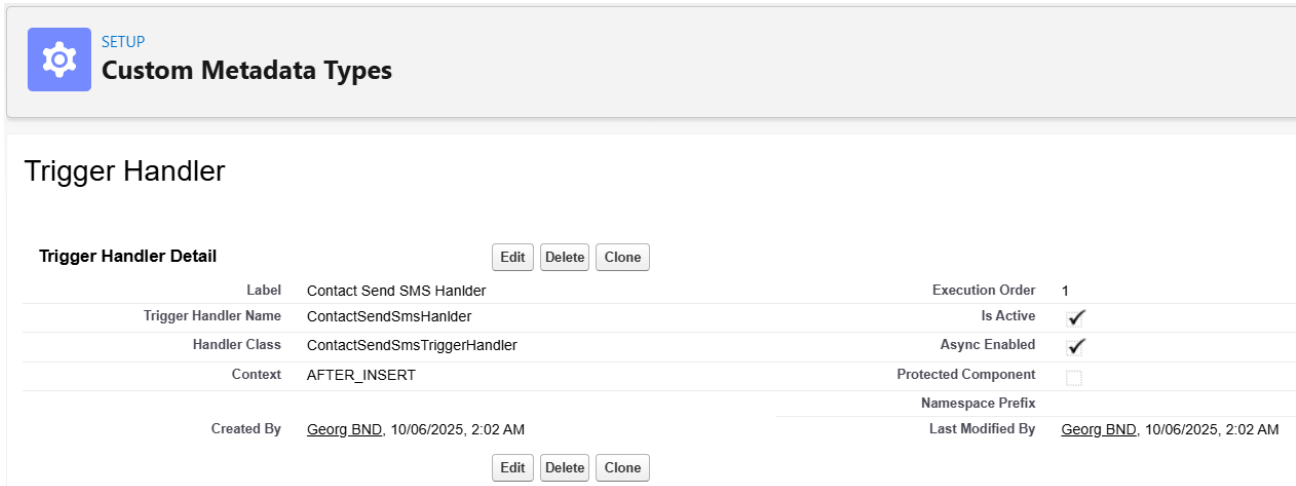
Table 3. Dependency of SOQL query count on number of handlers

Number of Handlers	Traditional Approach (SOQL)	With DAL (SOQL)
1	15	3
2	32	5
3	48	6
5	83	8
7	116	10
10	168	12

Dependency of SOQL Query Count on Number of Handlers**Fig. 3.** Dependency of SOQL query count on number of business handlers.

The graph demonstrates that with the traditional approach, the number of SOQL queries grows approximately linearly with a coefficient of about 17 queries per handler. With 7 handlers, the system exceeds the limit of 100 queries. The DAL approach shows significantly slower growth with a coefficient of about 1.2 queries per handler, and these additional queries are related to the need to load additional objects for new handlers. Even with 10 handlers, the system remains significantly below the limit, demonstrating the scalability advantages of the architectural approach.

A key innovation of the methodology is the use of Custom Metadata Types for declarative description of data dependencies. For each business handler, a record is created in the Custom Metadata Type `Trigger_Handler__mdt`, an example of which is shown in Figure 4.



SETUP
Custom Metadata Types

Trigger Handler

Trigger Handler Detail [Edit](#) [Delete](#) [Clone](#)

Label	Contact Send SMS Handler	Execution Order	1
Trigger Handler Name	ContactSendSmsHandler	Is Active	<input checked="" type="checkbox"/>
Handler Class	ContactSendSmsTriggerHandler	Async Enabled	<input checked="" type="checkbox"/>
Context	AFTER_INSERT	Protected Component	<input type="checkbox"/>
Created By Georg_BND , 10/06/2025, 2:02 AM		Last Modified By Georg_BND , 10/06/2025, 2:02 AM	

[Edit](#) [Delete](#) [Clone](#)

Fig. 4. Data dependency configuration for business handler through Custom Metadata.

The metadata structure includes the following fields: `Handler_Name` defines the unique handler name for identification; `Required_Objects` contains a comma-separated list of Salesforce objects; for each object, a separate Long Text Area field is created with a list of required fields; `Load_Related` boolean field determines the need to load related records through subqueries; `Related_Objects` specifies which related objects need to be loaded; `Is_Active` allows dynamically enabling and disabling the handler without code changes; `Execution_Order` numeric field determines handler execution order; `Async_Enabled` allows switching the handler to asynchronous mode to obtain additional governor limits for resource-intensive operations.

The mathematical model for guaranteeing resource constraints is based on the following proof. Let $H = \{h_1, h_2, \dots, h_n\}$ be the set of active business handlers in the system. For each handler h_i , the set of required objects O_i and set of required fields F_{ij} for each object are defined. In the traditional approach without centralization, each handler can execute its own SOQL queries independently. Let k_i be the number of queries executed by handler h_i . Then the total number of queries is $N_{\text{traditional}} = \sum_{i=1}^n k_i$. With average value $k = 5$ and $n = 20$ handlers, we obtain $N_{\text{traditional}} = 100$, which is the limit threshold value.

When using the Data Access Layer, all objects required by handlers are unified into the set $O_{\text{union}} = O_1 \cup O_2 \cup \dots \cup O_n$. The cardinality of this set $|O_{\text{union}}| = M$ does not exceed the total number of objects in the system's data model. For each object $o_j \in O_{\text{union}}$, one SOQL query is formed that loads all required fields and related records. If an object requires related records at k nesting levels, this may add up to k additional subqueries. Thus $N_{\text{DAL}} \leq M + R \times M$, where R is the maximum depth of subqueries. For typical systems $M \leq 10$ and $R \leq 2$, therefore $N_{\text{DAL}} \leq 30 < 100$.

Since it is architecturally prohibited to execute SOQL queries in business handlers ($N_{\text{handlers}} = 0$), then regardless of the number of handlers n , the total number of queries is bounded: $N_{\text{total}} = N_{\text{DAL}} \leq 30 < 100$. This mathematically guarantees non-violation of governor limits and provides formal proof of the methodology's correctness.

An additional important result is the system for dynamic execution mode management. Through the `Async_Enabled` field in Custom Metadata, each business handler can be marked for asynchronous execution. When asynchronous mode is selected, handler logic is automatically executed in Queueable Apex, which provides extended governor limits: 200 SOQL queries instead of 100; 60,000 milliseconds CPU time instead of 10,000; 12 megabytes heap size instead of 6. This ensures flexibility when working with resource-intensive operations such as complex calculations or processing large data volumes without requiring code changes to handlers.

The methodology also addresses code structuring and execution order management problems. The Single Responsibility Principle is ensured through the fact that each handler is responsible only for one business function and has no access to infrastructure code for data access. Handler execution

order is determined through the numeric `Execution_Order` field in metadata, allowing dynamic sequence changes without recompilation or code deployment. The capability for activation and deactivation is implemented through the `Is_Active` boolean field in metadata, which is critically important during data migrations, troubleshooting production issues, or phased feature rollout.

7. Prospects for further research development

Further development of the research envisions several promising directions that expand and deepen the developed methodology. First, extending the methodology to other types of governor limits, particularly CPU time and heap size, is relevant. Developing a static code analysis system for predicting CPU time consumption based on analysis of algorithmic cyclomatic complexity in business handlers will allow preventive identification of potential problems before production deployment. Creating a mathematical model of heap size dependency on data structures and collections used will enable architectural guarantees of compliance with these limits as well.

Second, investigating machine learning capabilities for optimizing data caching strategies represents significant scientific interest. Based on historical transaction execution metrics from Event Monitoring, models could be built that predict which data will be required by handlers in future transactions and preemptively load it into the Data Access Layer. Application of association rule algorithms and collaborative filtering would reveal hidden dependencies between handlers and optimize data loading.

Third, developing methods for automated testing and verification of code compliance with the methodology's architectural principles is relevant. Creating a rule set for Salesforce PMD (Programming Mistake Detector) or a custom plugin for Salesforce Code Analyzer that verifies adherence to the "no SOQL queries in handlers" rule will help integrate quality control into CI/CD pipeline and automatically block deployment of code violating architectural principles.

Fourth, adapting the methodology for other cloud platforms with analogous resource constraints is promising. The principles of data access centralization and architectural limit guarantees can be applied to Microsoft Dynamics 365, where analogous constraints exist on the number of queries in plugins; Oracle Cloud CX with limitations in Groovy scripts; SAP Cloud Platform with limits on custom code execution. Comparative research on methodology effectiveness across different platforms will reveal universal principles for constructing resource-efficient enterprise systems.

8. Conclusions

The work developed a methodology for dynamic management of transactional resources on the Salesforce platform, which solves the critical problem of exceeding SOQL query limits through an architectural approach of mandatory data access centralization.

The conducted literature analysis revealed the absence of formalized methodologies with mathematical guarantees of governor limit compliance. Existing approaches developed by Pethad, Abhishek Subbu, Tony Scott, and Kevin O'Hara are limited to general recommendations and implementation patterns without a systematic architectural solution to the SOQL query minimization problem.

A mathematical model was developed that formalizes the dependency of SOQL query count on architectural system parameters. It was proven that with data access centralization through the Data Access Layer, the number of queries is bounded as $N_{\text{total}} = N_{\text{DAL}} \leq M + R$, where M is the number of objects in the data model and R is the maximum depth of subqueries. For typical systems, this guarantees $N_{\text{total}} \leq 30 < 100$ regardless of the number of business handlers.

Experimental validation confirmed the methodology's effectiveness. When processing 200 Contact records, the traditional approach executed 165 SOQL queries and threw System.LimitException, whereas the Data Access Layer approach executed only 8 queries and successfully completed the transaction. Execution time improved from 8700 to 2300 milliseconds, CPU time decreased from 3180 to 1580 milliseconds, demonstrating not only limit compliance but also significant improvement in overall system performance.

The methodology ensures resolution of five key problems in Salesforce platform development. First, code structuring is achieved through clear separation into five architectural layers with formalized interfaces and adherence to the Single Responsibility Principle. Second, dynamic management of execution order is implemented through the numeric Execution_Order field in Custom Metadata without requiring code changes. Third, the capability for functionality activation and deactivation is provided through the Is_Active boolean field for safe production environment management. Fourth, guaranteed SOQL limit compliance is achieved through architectural prohibition of queries in business handlers. Fifth, flexible selection of synchronous or asynchronous execution mode through the Async_Enabled field allows system adaptation to varying performance requirements.

The scientific novelty of the work lies in formalizing architectural rules for data access centralization as a sufficient condition for guaranteeing non-violation of resource constraints, creating a mathematical model of SOQL query dependency on architectural parameters, and developing a system for declarative description of data dependencies through Custom Metadata Types. Unlike existing approaches, the methodology provides mathematical guarantees of governor limit compliance rather than relying on developer discipline.

The practical value of the methodology lies in the ability to construct scalable Salesforce solutions with guaranteed resilience to governor limits. The methodology can be implemented as a foundational architectural standard for enterprise projects, which will prevent critical errors in production environments and reduce costs for code refactoring when scaling the system. Using Custom Metadata for configuration ensures flexibility in system management without requiring code deployment, which is especially important for regulated industries with strict change control processes.

References:

- 1) Akash, P. T. (2025). Salesforce Trigger Framework Best Practices: Bulk-Safe, Secure, and Scalable Apex. Medium. Available at: https://medium.com/@akash15_dev/salesforce-trigger-framework-best-practices-e0c7366a0b68
- 2) ApexHours. (2024). Bulkification of Apex Triggers. Available at: <https://www.apexhours.com/bulkification-of-apex-triggers/>

- 3) ApexHours. (2025). Trigger Framework in Salesforce. Available at: <https://www.apexhours.com/trigger-framework-in-salesforce/>
- 4) CloudQnect. (2023). Are You Optimizing Apex Triggers in Salesforce? Available at: <https://cloudqnect.com/are-you-optimizing-apex-triggers-in-salesforce/>
- 5) Connecting Software. (2024). Salesforce Governor Limits Explained. Available at: <https://www.connecting-software.com/blog/salesforce-governor-limits-explained/>
- 6) Pethad, C. A. (2020). Platform Event Trigger Framework Implementation in Salesforce Apex. *Journal of Scientific and Engineering Research*, 7 (5), 391–394.
- 7) Riyas, M. (2025). Salesforce Trigger Framework: A Best Practice Guide. Medium. Available at: <https://mrriyas.com/salesforce-trigger-framework-a-best-practice-guide-585400ace6b3>
- 8) Salesforce. (2024). Effective Bulk Apex Trigger Design Techniques. Salesforce Trailhead. Available at: https://trailhead.salesforce.com/content/learn/modules/apex_triggers/apex_triggers_bulk
- 9) Salesforce. (2024). Implement Frameworks. Salesforce Trailhead. Available at: <https://trailhead.salesforce.com/content/learn/modules/success-cloud-coding-conventions/implement-frameworks-sc>
- 10) Salesforce. (2024). Record-Triggered Automation. Salesforce Architects. Available at: <https://architect.salesforce.com/decision-guides/trigger-automation>
- 11) Salesforce. (2024). Salesforce Developer Limits and Allocations Quick Reference. Salesforce Developer Documentation. Available at: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_gov_limits.htm
- 12) Salesforce Ninja. (2020). Salesforce Performance Series: The forgotten art of APEX trigger frameworks. Available at: <https://salesforce-ninja.com/2020/11/23/salesforce-performance-the-forgotten-art-of-apex-trigger-frameworks/>
- 13) SalesforceCodex. (2023). Apex Trigger Code Optimization. Available at: <https://salesforcecodex.com/salesforce/apex-trigger-code-optimization/>
- 14) Stellaxius. (2025). Why and How to Bulkify your Code. Stellaxius Knowledge Center. Available at: <https://stellaxius.com/knowledgecenter/software-development/bulkify-your-code/>
- 15) Subbu, A. (2020). Trigger Optimization Framework in Salesforce. Available at: <https://abhisheksubbu.github.io/trigger-optimization-framework/>
- 16) Usman, M. (2024). Mastering Salesforce Governor Limits and Performance Optimization. Medium. Available at: <https://medium.com/@usmansfdc/mastering-salesforce-governor-limits-and-performance-optimization-part-14-f5fec2e8d889>