RESEARCH ARTICLE

SAMRIDDHI Volume 13, Issue 2, 2021 Print ISSN: 2229-7111 Online ISSN: 2454-5767

Mitigating View State and Heap Size Constraints in Large-Scale Enterprise Visualforce Applications: A JavaScript Remoting–Based Architecture

Santhosh Reddy Basireddy

Senior Salesforce Developer USAA, San Antonio, Texas, USA

Abstract

Large-scale enterprise applications built on the Salesforce Platform frequently combine Visualforce pages with Apex controllers to support complex, data-intensive business workflows. As such applications grow in interaction complexity, two platform-level constraints become increasingly difficult to manage: the Visualforce view state size limit and the Apex heap size limit. This paper presents an architecture, developed and applied in a production enterprise Salesforce application supporting a multi-object business workflow, for eliminating recurring view state exceptions and improving page responsiveness. The root cause is traced to a common Visualforce implementation pattern in which apex:actionFunction action methods — constrained by the platform to return a PageReference rather than data — write their results into persistent, non-transient controller variables that are never cleared, causing the view state's contribution from these variables to accumulate across a user's session until the 170 KB limit is exceeded, while also incurring a full view-state round trip on every interaction. The proposed architecture relocates this data retrieval to JavaScript Remoting calls that return data directly, and migrates the associated rendering logic from server-side Visualforce/Apex tag rendering to client-side JavaScript-driven rendering, bypassing the view state entirely. We describe the design, implementation pattern, and operational trade-offs of this approach, and discuss its applicability to other Salesforce applications facing similar constraints. The approach eliminated the recurring view state exceptions in the application studied and improved page responsiveness, without requiring a migration away from the underlying Visualforce framework.

Keywords: Salesforce, Visualforce, Apex, view state, heap size, governor limits, JavaScript Remoting, enterprise software architecture

SAMRIDDHI : A Journal of Physical Sciences, Engineering and Technology (2021); DOI: 10.18090/samriddhi.v13i02.19

Introduction

The Salesforce Platform is widely used to build enterprise applications supporting business-critical workflows such as customer relationship management, operational risk management, and regulatory compliance tracking. Many such applications are implemented using Visualforce, Salesforce's original server-side page framework, paired with Apex, its proprietary strongly-typed programming language. As these applications mature, their underlying data models often grow substantially in size and structural complexity, incorporating dozens of interrelated custom objects connected through lookup and master-detail relationships.

This growth exposes two platform-enforced governor limits that are frequently under-appreciated in early-stage application design: the Visualforce view state size limit, currently set at 170 KB per page [1], and the Apex heap size limit, set at 6 MB for synchronous transactions and 12 MB for asynchronous transactions [2]. Applications that were designed without these limits in mind often begin to fail intermittently as data volume grows, producing exceptions that are difficult to diagnose because they manifest only under specific data conditions rather than consistently.

This paper documents an architecture developed to resolve recurring view state and heap size exceptions in a production enterprise Salesforce application supporting a complex, multi-object business workflow, and generalizes the approach as a reusable pattern for similarly affected

Corresponding Author: Santhosh Reddy Basireddy, affiliation, e-mail: Santhureddy245@gmail.com

How to cite this article: Basireddy, S.R. (2021). Mitigating View State and Heap Size Constraints in Large-Scale Enterprise Visualforce Applications: A JavaScript Remoting–Based Architecture. *SAMRIDDHI : A Journal of Physical Sciences, Engineering and Technology,* 13(2), 195-201.

Source of support: Nil

Conflict of interest: None

applications. The contribution of this work is threefold: (1) a root-cause analysis connecting a specific and common Visualforce anti-pattern — binding global controller variables directly to page markup — to view state accumulation and heap growth under bulk data conditions; (2) an architecture that relocates the affected data access to JavaScript Remoting, removing it from the view state entirely; and (3) a discussion of the design and security trade-offs introduced by this migration, informed by practical implementation experience.

Background and Related Work

Visualforce View State

The Visualforce view state is a serialized representation of a page's non-transient controller state, maintained across the lifecycle of user interactions with a page and re-transmitted with every subsequent request [1]. It encompasses all non-transient instance variables in the controller and any controller extensions, along with internal state associated with page components. Salesforce enforces a hard limit of 170 KB on the size of this serialized state; pages that exceed this limit throw a runtime exception and fail to load [1], [3]. Because the view state must be serialized, encrypted, transmitted, decrypted, and deserialized on every request, larger view states also degrade page load performance even before the hard limit is reached [1] [14].

Apex Heap Size Governor Limits

Independently of view state, Apex enforces a heap size governor limit describing the total memory consumed by objects and variables during a single transaction: 6 MB for synchronous execution contexts and 12 MB for asynchronous contexts such as Batch Apex or Queueable Apex [2], [4]. Because Salesforce operates a multi-tenant architecture, these transaction-level limits are enforced to protect shared platform resources [4], [15]. Exceeding the applicable heap limit produces a governor-limit runtime exception [4]. Bulk data operations — for example, loading, transforming, or aggregating large sets of related records within a single transaction — are common triggers when large collections of sObjects or their string representations are retained in memory simultaneously [4].

Visualforce Action Components and the PageReference Constraint

Standard Visualforce action components — including apex:commandButton, apex:actionSupport, apex:actionPoller, and apex:actionFunction — invoke a controller or extension method in response to a user or page event [10]. A defining constraint of these action methods is that they must return a value of type PageReference, or void, which is treated as a null PageReference [11]. This constraint exists because these methods were designed primarily to support page navigation and full or partial page refreshes, not to return arbitrary application data to the caller.

Because an action method bound to apex:actionFunction cannot itself return a data value — for example, a queried list or a computed aggregate — to the JavaScript that invoked it, a common implementation pattern is to have the action method write its result into one or more existing public controller instance variables that are already bound to Visualforce components elsewhere on the page, and then request a partial page refresh using the component's reRender attribute so that the bound components redisplay using the updated variable values [11]. This pattern allows apex:actionFunction to emulate returning data, at the cost of relying on persistent controller state rather than a direct method return value.

JavaScript Remoting as a Mitigation Technique

Salesforce provides JavaScript Remoting as a mechanism for invoking static Apex controller methods directly from client-side JavaScript, bypassing the standard Visualforce form-and-view-state request lifecycle entirely [7], [8]. Because remoting calls are stateless, asynchronous, and exchange only the specific data requested (typically serialized as JSON), they avoid contributing to view state altogether [7], [9]. JavaScript Remoting follows the broader Ajax model of asynchronous client-server interaction [16], while research on Ajax-based applications and cloud application development has documented the engineering complexity introduced by richer client-side behavior and cloud delivery models [17], [18]. Existing practitioner literature identifies JavaScript Remoting as an established technique for improving Visualforce page responsiveness [8], [9]; however, its use as a systematic, page-wide remediation strategy for view-state and heap-size exceptions in large, multi-object enterprise applications is comparatively less documented, motivating the case study and generalized pattern presented here.

Problem Statement

This paper reports a single-case, exploratory industrial case study, following established guidance for conducting and reporting case study research in software engineering [12]. The unit of analysis is a recurring architectural failure mode observed in a production enterprise application, and the study's purpose is to characterize its root cause and evaluate a proposed architectural remediation, rather than to establish generalized quantitative claims across a population of applications; the scope and limitations of this approach are discussed further in Section 6.1 [19], [20].

The architecture described in this paper was developed for a production Salesforce application supporting a large enterprise business workflow, whose data model had grown to more than 80 interrelated custom objects connected through an extensive network of lookup and master-detail relationships. Several Visualforce pages within the application were responsible for presenting aggregated views of related records — for example, summarizing child records

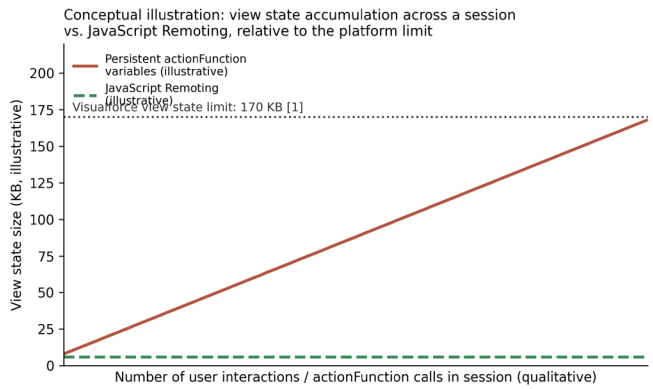

Fig. 1: Conceptual illustration of view state accumulation across a user session under the persistent actionFunction-variable pattern versus JavaScript Remoting, relative to the platform's 170 KB view state limit [1]. Curves are illustrative of the qualitative relationship described in Section 4, not measured data.

associated with a parent record for review by an end user. As the volume of underlying data grew, these pages began to intermittently fail with two distinct classes of governor-limit exceptions: view state size limit exceptions during ordinary page navigation, and Apex heap size exceptions during bulk data-loading operations.

Diagnosis of these failures was complicated by their intermittency: pages that functioned correctly for records with small numbers of related child records failed only once the volume of related data crossed an unpredictable threshold, making the underlying cause difficult to isolate through conventional testing alone.

Root Cause Analysis

View State Accumulation from Persistent Action-Function Variables

Investigation of the affected pages found that user interactions were handled through apex:actionFunction calls to controller action methods. Because these methods were constrained to return a PageReference or void rather than a data value [10], [11], they could not return their results directly to the invoking JavaScript. Instead, each action method wrote its result into one or more public, non-transient controller instance variables — in the pages examined, a List holding queried related records and a Map holding computed aggregate values — that were already bound to Visualforce components elsewhere on the page, and requested a partial page refresh via the reRender attribute so that those components would redisplay using the updated values.

Because Visualforce serializes all non-transient controller instance state into the view state on every request [1], these two variables were included in the view state on every subsequent request once populated — not only the request in which they were first set. Critically, because the variables were never explicitly cleared or reset, each successive apex:actionFunction call added to or updated their contents without removing what was already present, so the view state's contribution from these two variables grew monotonically over the course of a user's session on the page, independent of how large any single related-record query result was on its own. Once accumulated view state size exceeded the 170 KB limit, the page failed with a view state size exception [1].

Page Responsiveness and Apex Heap Size During Bulk Operations

A related but distinct effect explains the pages' degraded responsiveness even before the view state limit was reached. Because every user interaction relied on a standard apex:actionFunction call rather than JavaScript Remoting, each interaction incurred a full view state serialization, encryption, transmission, decryption, and deserialization cycle on top of the ordinary request processing [1] — and, as

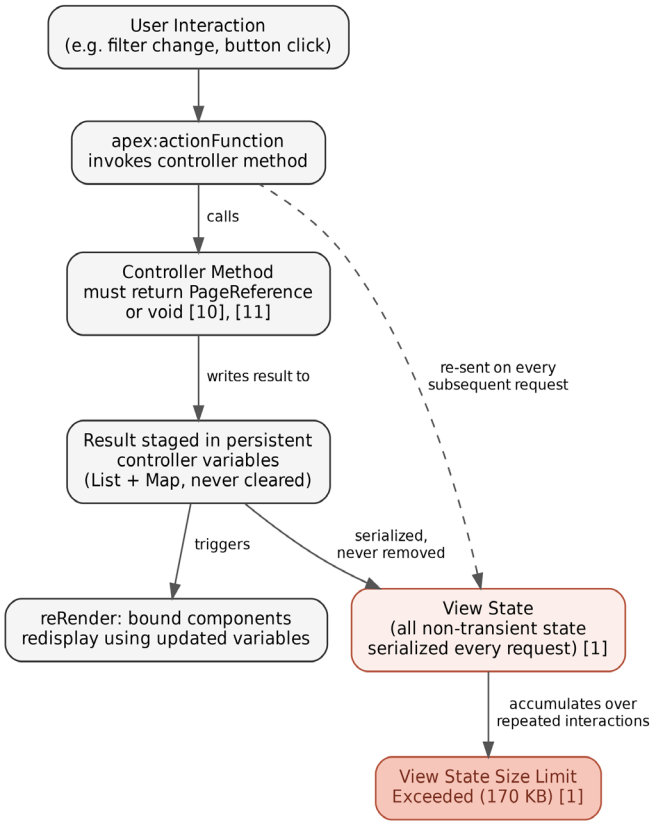

Fig. 2: Data flow prior to remediation. Because apex:actionFunction action methods must return a PageReference rather than data [10], [11], results are written into persistent controller variables and displayed via a partial-page reRender. Because these variables are never cleared, their contribution to the view state accumulates across the session.

described above, this cycle carried a steadily growing payload as the session progressed. This made pages perceptibly slower to respond as a session continued, independent of and prior to the eventual hard failure once the 170 KB limit was crossed.

Separately, during bulk data-loading operations that queried, transformed, or aggregated large sets of related records within a single transaction, controller logic that iterated over these result sets in memory accumulated Apex heap usage proportional to the volume processed, eventually exceeding the 6 MB synchronous heap limit [2], [4]. Unlike the view state issue described in Section 4.1, this failure mode was driven by the size of an individual transaction's in-memory data rather than by accumulation across a session, but it stemmed from the same underlying architectural pattern of processing growing data volumes without regard to platform-enforced resource limits.

The underlying issue in both cases was architectural rather than incidental: the pages had been designed without an explicit strategy for bounding either per-request data volume or session-level state accumulation, and the reliance on persistent controller variables as a workaround for the PageReference return-type constraint created a direct path from ordinary user interaction to both degraded responsiveness and eventual governor-limit failure.

Proposed Architecture

Relocating Data Access to JavaScript Remoting

The core remediation was to stop relying on apex:actionFunction and the persistent-variable workaround described in Section 4.1, and instead expose the required data through static Apex controller methods annotated for JavaScript Remoting, which can return data directly to the calling JavaScript rather than requiring it to be staged in controller instance variables [7]. Because remoting calls do not participate in the view state lifecycle [7], [9], this change removed the accumulating variables from the view state entirely, regardless of how many interactions occurred during a session [21].

```
// Before: actionFunction method cannot return data directly,
// so results are staged in persistent controller variables
// and displayed via a partial-page reRender
public class RelatedRecordsController {
    public List<ChildRecord__c> relatedRecords { get; set; }
    public Map<Id, Decimal> aggregatedValues { get; set; }

    public PageReference loadRelatedRecords() {
        relatedRecords = [SELECT Id, Name, Status__c
                          FROM ChildRecord__c
                          WHERE Parent__c = :parentId];
        aggregatedValues = computeAggregates(relatedRecords);
        return null; // required return type; data is
                     // communicated via the bound variables above
    }
}

<apex:actionFunction name="loadRelatedRecords"
    action="{!loadRelatedRecords}"
    reRender="relatedRecordsPanel" />

// After: JavaScript Remoting returns data directly;
// no persistent controller variables are required,
// so nothing accumulates in the view state
public class RelatedRecordsController {
    @RemoteAction
    public static Map<String, Object> getRelatedRecords(Id parentId) {
        List<ChildRecord__c> records = [SELECT Id, Name, Status__c
                                        FROM ChildRecord__c
                                        WHERE Parent__c = :parentId];
        return new Map<String, Object>{
            'records' => records,
            'aggregates' => computeAggregates(records)
        };
    }
}
```

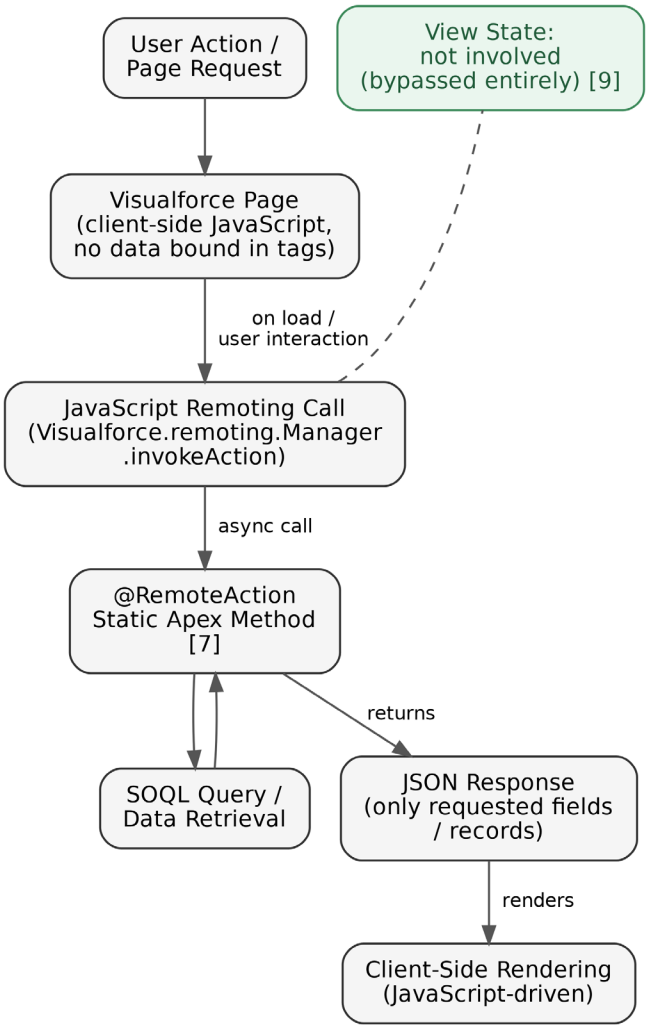

Fig. 3: Data flow after remediation. Data access moves to an asynchronous JavaScript Remoting call to a static @RemoteAction method; the response is rendered client-side, and the view state is bypassed entirely [7], [9].

The corresponding Visualforce page was modified to invoke this method asynchronously via the Visualforce Remoting JavaScript API in response to the same user interactions that previously triggered apex:actionFunction, populating the display client-side rather than staging results in server-side controller variables.

Migrating Rendering Logic to Client-Side JavaScript

Removing the data from view state addressed the view-state limit directly, but the associated Apex heap consumption

from server-side iteration and transformation of the same collections remained. The second element of the architecture migrated this transformation logic from server-side Apex, invoked through Visualforce component tags on page render, to client-side JavaScript operating on the JSON payload returned by the remoting call. Aggregation, filtering, and formatting logic that had previously executed in the Apex controller — accumulating heap usage proportional to record volume — was re-implemented to run in the browser after data retrieval, so that the corresponding Apex transaction's heap footprint was limited to the query and serialization of the requested data rather than its full in-memory transformation.

Design and Operational Considerations

Security enforcement: because methods annotated with @RemoteAction must be static, they do not automatically inherit record-level or field-level security context in the same way instance methods bound to a page controller do. Each remoted method was audited to explicitly enforce sharing rules and field-level security, since this responsibility shifts to the developer under this pattern [7], [13].

State management: removing data from the view state shifts responsibility for maintaining page state across user interactions to client-side JavaScript. This increases client-side code complexity relative to the declarative Visualforce binding model, and was managed through a consistent, reusable JavaScript module pattern applied across affected pages rather than ad hoc per-page implementations [22].

Payload sizing: because JavaScript Remoting payloads are not subject to the view state limit, care was taken to still request only the fields and records required for a given view, avoiding simply relocating an unbounded-growth problem from the view state to the network payload [6].

Incremental adoption: the migration was applied incrementally, page by page, prioritizing pages with the highest observed failure frequency, allowing the pattern to be validated and refined before being applied as a standard practice across the wider application.

Results and Discussion

Following application of this architecture to the affected pages, the recurring view state size limit exceptions and Apex heap size exceptions previously observed during ordinary use and bulk data operations were eliminated. Pages that had previously failed once related-record volume crossed an unpredictable threshold continued to function correctly at substantially higher data volumes, since their view state and heap consumption were no longer directly proportional to the size of the underlying related-record collections. Page responsiveness during bulk data loading also improved, consistent with the platform's own guidance that reduced view state size correlates with faster page load and reduced request stalling [1].

Threats to Validity

Following established guidance for reporting industrial case studies in software engineering [12], [19], [20], we discuss the validity of this study along four standard dimensions: construct validity, internal validity, external validity, and reliability.

Construct validity concerns whether the concepts studied — view state accumulation, heap size growth, and page responsiveness — were measured in a manner consistent with their accepted platform-level definitions. Because the observed outcomes (elimination of governor-limit exceptions, continued page functionality at higher data volumes) were assessed against Salesforce's own documented and enforced governor limits [1], [2], rather than an ad hoc or informally defined metric, construct validity is reasonably well supported. However, the qualitative characterization of "improved responsiveness" was based on practitioner observation during and after the migration rather than instrumented, repeatable timing measurements, and should be interpreted accordingly.

Internal validity concerns whether the described architectural change was in fact responsible for the observed outcomes, as opposed to other concurrent changes to the application. The migration was applied incrementally, page by page, with each affected page observed to stop producing governor-limit exceptions following its individual migration, which supports a causal link between the architectural change and the outcome. Nonetheless, because this was an evolving production application rather than a controlled experimental environment, the possibility of confounding concurrent changes cannot be fully excluded.

External validity concerns the extent to which these findings generalize beyond the single application studied. This work reports a single industrial case study of one enterprise Salesforce application; while the underlying platform constraints (the 170 KB view state limit and Apex heap size limits) and the anti-pattern identified (direct binding of growing collections to page markup) are general properties of the Visualforce framework rather than specific to this application, the effectiveness of the proposed remediation in other applications with different data models, usage patterns, or organizational constraints has not been independently verified. Replication across additional Visualforce applications would strengthen confidence in the generality of the pattern.

Reliability concerns whether independent researchers or practitioners could reach similar conclusions applying the same approach. The architectural pattern described — relocating data access to JavaScript Remoting and rendering logic to client-side JavaScript — is described at a level of technical detail (including illustrative code patterns) intended to support independent application and evaluation by other practitioners, which we consider the primary avenue for corroborating these findings.

Practical Implications

For practitioners maintaining large, mature Visualforce applications, this case study suggests two concrete recommendations. First, data-access patterns that bind growing collections directly to page markup should be treated as a scalability risk to be identified proactively — for example, through code review guidelines or static analysis — rather than addressed reactively once governor-limit exceptions begin appearing in production. Second, JavaScript Remoting should be considered a standard, page-wide architectural option for data-intensive Visualforce pages, rather than an isolated optimization applied only after a specific page has already failed, particularly for organizations not yet in a position to migrate affected pages to Lightning Web Components [5].

More broadly, this case study suggests that direct binding of growing, unbounded collections to Visualforce markup is a common but under-recognized architectural risk in enterprise Salesforce applications, particularly those whose data models expand significantly after initial release. Because the failure mode is data-volume-dependent rather than immediately apparent during early development and testing, it is easily introduced without being detected until an application is already in production and handling meaningful data volumes. The JavaScript Remoting–based pattern described here offers a systematic remediation that does not require migrating away from Visualforce entirely, which may be a practical constraint for organizations with substantial existing Visualforce investment not yet ready to migrate to Lightning Web Components.

This approach is not without trade-offs. Shifting state management and business logic to client-side JavaScript increases the complexity of client-side code and shifts security-enforcement responsibility onto the developer in ways that the standard Visualforce controller model handles more implicitly. Organizations adopting this pattern at scale should establish shared client-side modules and explicit security-review practices for remoted methods, rather than applying the pattern ad hoc on a page-by-page basis.

Conclusion and Future Work

This paper presented an architecture for resolving recurring Visualforce view state and Apex heap size governor limit exceptions in a large-scale enterprise Salesforce application, by relocating growing, directly-bound data collections to JavaScript Remoting and migrating associated rendering and transformation logic to client-side JavaScript. The approach eliminated the governor-limit exceptions previously observed in the application studied and improved page performance during bulk data operations, while preserving the existing Visualforce-based architecture.

Future work includes quantifying the performance impact of this migration with controlled before-and-after measurements, evaluating the pattern's applicability during incremental migration from Visualforce to Lightning Web Components [5], and examining whether similar directly-bound-collection anti-patterns and remediation patterns generalize to other page-based enterprise low-code platforms with comparable server-side state management constraints.

Data Availability

The code samples presented are illustrative and generalized to convey the architectural pattern described; no proprietary source code, business data, or organizational identifiers from the studied application are disclosed in this paper.

Author Contributions

The sole author designed and implemented the architecture described, and wrote and approved the manuscript.

Ethics

Not applicable — this study did not involve human subjects, personal data, or animal research.

References

[1] Salesforce, "Optimize the View State," Visualforce Developer Guide, Salesforce Developers. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.pages.meta/pages/pages_best_practices_perf_view_state.htm

[2] Salesforce, "Salesforce Developer Limits and Allocations Quick Reference," Salesforce Documentation. [Online]. Available: https://resources.docs.salesforce.com/latest/latest/en-us/sfdc/pdf/salesforce_app_limits_cheatsheet.pdf

[3] B. Cline, "How To Reduce Visualforce's View State," brcline.com, Mar. 2017.

[4] Salesforce, "Execution Governors and Limits," Apex Developer Guide, Salesforce Developers.

[5] A. Rivas, "New Resources for Moving from Visualforce to Lightning Web Components," Salesforce Developers Blog, Jul. 2020. [Online]. Available: https://developer.salesforce.com/blogs/2020/07/new-resources-for-moving-from-visualforce-to-lightning-web-components

[6] Salesforce, "JavaScript Remoting Limits," Visualforce Developer Guide, Salesforce Developers. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.pages.meta/pages/pages_js_remoting_limits.htm

[7] Salesforce, "JavaScript Remoting for Apex Controllers," Visualforce Developer Guide, Salesforce Developers. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.pages.meta/pages/pages_js_remoting.htm

[8] K. Bowden, Visualforce Development Cookbook, 2nd ed. Birmingham, UK: Packt Publishing, 2016.

[9] C. Isakson, "How To: Visualforce Javascript Remoting," Sundog Interactive Blog, Jan. 2014.

[10] Salesforce, "apex:actionFunction," Visualforce Component Reference, Salesforce Developers. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.pages.meta/pages/pages_compref_actionFunction.htm

[11] S. Saurabh, "Demystifying the PageReference," writeforce.blogspot.com, Aug. 2014.

[12] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," Empirical Software Engineering, vol. 14, no. 2, pp. 131–164, 2009.

[13] A. N. Topalli, "Learn MOAR in Spring '20 with Field Level Security in Apex," Salesforce Developers Blog, Jan. 2020. [Online]. Available: https://developer.salesforce.com/blogs/2020/01/learn-moar-in-spring-20-with-field-level-security-in-apex

[14] Salesforce, "Using the transient Keyword," Apex Developer Guide, Salesforce Developers. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_keywords_transient.htm

[15] C. D. Weissman and S. Bobrowski, "The design of the force.com multitenant internet application development platform," in Proc. 35th ACM SIGMOD Int. Conf. Management of Data (SIGMOD '09), Providence, RI, USA, 2009, pp. 889–896, doi: 10.1145/1559845.1559942.

[16] J. J. Garrett, "Ajax: A New Approach to Web Applications," Adaptive Path, Feb. 2005. [Online]. Available: https://adaptivepath.org/ideas/ajax-new-approach-web-applications/

[17] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes," ACM Transactions on the Web, vol. 6, no. 1, art. 3, 2012, doi: 10.1145/2109205.2109208.

[18] J. Cito, P. Leitner, T. Fritz, and H. C. Gall, "The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud," in Proc. 10th Joint Meeting on Foundations of Software Engineering, 2015, pp. 393–403, doi: 10.1145/2786805.2786826.

[19] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg, "Preliminary Guidelines for Empirical Research in Software Engineering," IEEE Transactions on Software Engineering, vol. 28, no. 8, pp. 721–734, 2002, doi: 10.1109/TSE.2002.1027796.

[20] J. Siegmund, N. Siegmund, and S. Apel, "Views on Internal and External Validity in Empirical Software Engineering," in Proc. 37th IEEE/ACM International Conference on Software Engineering, 2015, pp. 9–19, doi: 10.1109/ICSE.2015.24.

[21] Salesforce, "RemoteAction Annotation," Apex Developer Guide, Salesforce Developers. [Online]. Available: https://developer.salesforce.com/docs/atlas.en-us.apexcode.meta/apexcode/apex_classes_annotation_remoteaction.htm

[22] F. S. Ocariza Jr., K. Bajaj, K. Pattabiraman, and A. Mesbah, "An Empirical Study of Client-Side JavaScript Bugs," in Proc. ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 55–64, doi: 10.1109/ESEM.2013.19.