Information, Computing and Intelligent Systems, 2025, No. 7, 180 – 197
UDC: 004.4, 004.738.5 https://doi.org/10.20535/2786-8729.7.2025.341787

MULTI-STRATEGY AJAX AND EVENT-DRIVEN STATE MANAGEMENT FOR RESPONSIVE WEB APPLICATIONS

Nataliia Rudnikova*
http://orcid.org/0009-0008-7057-4241

Oleksiy Nedashkivskiy
http://orcid.org/0000-0002-1788-4434

National Technical University of Ukraine
“Igor Sikorsky Kyiv Polytechnic Institute”, Kyiv, Ukraine

*Corresponding author: rudnikova.n.s.-tvz51f@edu.kpi.ua

Research addresses engineering high-performance, responsive web apps for complex data and real-time user interaction. The study focuses on the client-server integration in a monolithic Django pattern/architecture, specifically the orchestration of asynchronous client technologies, for instance, AJAX, JavaScript, and server logic, for instance, Python/Django. The goal is to design, implement, and validate a unified AJAX integration framework. This framework enables seamless real-time data exchange, dynamic updates, and complex state management for diverse components: interactive tables, multi-dimensional charts, multi-step forms, and the checkout session container. Django framework, jQuery for AJAX, and JavaScript libraries (Chart.js, DataTables) are included as materials. Methods applied involve systematic software architecture design, asynchronous programming analysis, RESTful API development, and empirical performance benchmarking of data-loading and state management strategies. Scientific contribution is twofold. Firstly, multi-strategy AJAX integration model is formalized as a decision framework that dynamically selects between server-side rendering (django-tables2), client-side rendering (vanilla jQuery/DataTables), and a hybrid AJAX-datatable approach based on data complexity, volume, and interaction. Secondly, event-driven state management system as a robust design for distributed, session-based UI components using a centralized AJAX action dispatcher and a universal state synchronization function. This ensures data consistency across independent page components and eliminates race conditions in concurrent operations. As a result, the framework achieved a significant reduction in server load and perceived latency. The benchmarked components consistently showed sub-200ms response times for datasets over 10,000 records. The cart system handled over 1,000 consecutive operations without any state desynchronization.

Keywords: Server-Side Processing, AJAX integration, Django framework, JavaScript, dynamic data visualization, DataTables, single-page application, RESTful API, software architecture, performance optimization.

1. Introduction

The evolution of the World Wide Web has been marked by a continuous pursuit of richer, more responsive user experiences. The static, document-centric web of the past has given way to dynamic, application-like platforms that rival native desktop software in their complexity and interactivity. This paradigm shift has been largely driven by the maturation of core web technologies, particularly JavaScript, and the widespread adoption of Asynchronous JavaScript and XML (AJAX). AJAX liberated web applications from the "click-and-wait" model of full-page reloads, enabling discrete sections of a page to be updated independently based on user actions or real-time data streams. This capability is fundamental to modern web applications, from interactive dashboards and data analytics platforms to complex e-commerce systems and collaborative tools.

However, this increased interactivity introduces significant architectural complexity. The specific scientific problem lies in the effective, efficient, and maintainable orchestration of client-side logic with server-side business logic within a cohesive application structure. While numerous libraries and frameworks exist to handle individual aspects of this integration such as rendering a table or drawing a chart. However, a comprehensive, unified architectural approach that seamlessly combines diverse interaction patterns into a single, performant application remains an area requiring formalization and research. Inefficient data fetching strategies can lead to excessive server load and high latency. Poorly managed application state can result in inconsistent user interfaces, where different page components display conflicting information. Furthermore, ad hoc implementations of AJAX features often lead to code that is difficult to maintain, extend, and debug.

Consequently, this article presents a novel, holistic framework developed within the Django ecosystem, demonstrating how a systematic and principled integration of AJAX, JavaScript, and a Python backend can systematically address these challenges. The framework is not merely a collection of techniques, but rather a structured methodology for enhancing a traditional server-rendered Django application with high-interactivity features without requiring a complete, and often costly, rewrite into a fully decoupled SPA architecture. The relevance of this research is underscored by the persistent dominance of monolithic architectures in many business contexts due to their simplicity, SEO-friendliness, and rapid development cycle. Providing a clear path to incrementally add rich interactivity to these applications has substantial practical value. This study moves beyond isolated code snippets to present a generalized model for AJAX integration, complete with performance analysis and validation based on a fully functional implementation, as evidenced by the provided codebase encompassing URLs, views, templates, and client-side scripts.

2. Literature review and problem statement

The integration of client and server-side technologies has been a central theme in web development research from 2020 to 2025. While Garrett's seminal work established the foundations of AJAX, modern studies have expanded its application to complex industrial and scientific contexts. For instance, the role of modern web technologies in constructing high-performance scientific portals is emphasized in [1], while a comprehensive introduction to Python-based frameworks like Django as the backbone for such systems is provided in [2].

Subsequent research has largely bifurcated into two dominant paradigms:

1. full decoupled architectures. In a comparative analysis of frontend frameworks (React, Angular, Vue), the Django backend is utilized strictly as an API provider [3]. This model is further formalized as "Decoupled Django," with a focus placed on JavaScript-heavy frontends [4]. However, it is noted that while full-stack capability is offered, technological surface area and cognitive overhead are increased [5];

2. monolithic augmentation. The efficiency of managing web development within the native Django ecosystem is highlighted in [6]. However, the localization of interactivity is still considered a challenge. It is demonstrated that server-side processing is essential for data presentation responsiveness [7], a sentiment echoed in [8], where adaptive dictionary implementations were introduced specifically to reduce payload sizes in DataTables communication.

Despite these advancements, more nuanced integration is required for specialized applications. The necessity of dynamic dashboards for Industry 4.0 – requiring real-time updates difficult to achieve with standard monolithic tools – is discussed in [9]. The complexity of modeling these dynamic web applications is addressed in [10], while the integration of advanced features like AI voice bots into web interfaces is showcased in [11], by which traditional state management is further strained.

The technical transition from synchronous to asynchronous models is formally addressed through the development of methods for the automatic migration of JavaScript APIs [12]. Enabling parallelism in these asynchronous environments to improve performance was further explored in [13]. For message-driven architectures, model-driven development using AsyncAPI is proposed to handle complex communication [14]. More recently, focus has been placed on user behavior and payload optimization, marking the current frontier of responsive web engineering [8, 15].

Within the specific context of monolithic augmentation, solutions for the Django ecosystem have evolved through specialized libraries such as `django-tables2` for server-side rendering and `django-ajax-datatable` for hybrid processing. While these packages facilitate discrete interactive features, they are primarily designed to function in isolation. Existing documentation and literature fail to address the integration of these tools into a unified ecosystem of diverse components, such as synchronized charts and stateful checkout containers. Furthermore, these tools lack the formalized decision logic required for real-time updates in modern dashboard architectures, representing a significant gap in current web engineering research.

It shall be notified that a critical analysis of this body of work reveals three unresolved gaps in the context of monolithic Django applications:

– the state synchronization problem. While message-driven state is addressed in [14], a robust solution for managing session-based UI components across independent AJAX requests without race conditions is lacked in the literature;

– the multi-modal data rendering problem. Despite the optimizations suggested in [7] and [8], no clear decision-making framework (heuristic) is provided to select between SSR, client-side, or hybrid rendering based on data cardinality and complexity;

– the API proliferation and consistency problem. As interactive features grow, as seen in the dashboards described in [9], API endpoints often become disorganized and inconsistent data formats (HTML vs. JSON) are returned.

To summarize, these gaps form the precise problem statement: There is a need for a coherent architectural framework and reusable design patterns that enable the efficient, maintainable, and scalable integration of AJAX within a monolithic Django application.

3. The aim and objectives of the study

The overarching aim of this study is to design, implement, and empirically validate a novel integration framework for AJAX, JavaScript, and Python (Django) that significantly enhances the interactivity, responsiveness, and user experience of web applications, while preserving the development simplicity, SEO advantages, and rapid prototyping capabilities of the monolithic Django pattern. This framework specifically addresses the architectural challenges of state synchronization, multi-modal data rendering, and API consistency in highly interactive applications.

To achieve this aim, the following specific and measurable research objectives were set:

1. to develop and formalize a multi-strategy data presentation model, providing an evidence-based decision framework for selecting the optimal data rendering strategy (addressing the multi-modal data rendering problem);
2. to design and implement a robust, event-driven state management system for distributed UI components (checkout session container), ensuring data consistency and resilience under high-concurrency operations (addressing the state synchronization problem);
3. to integrate and unify the component strategies into a cohesive AJAX integration framework, establishing standardized API response formats and comprehensive parameter validation (addressing the API proliferation and consistency problem);
4. to conduct an experimental validation and rigorous comparative performance benchmarking of the developed framework and its component strategies under varying dataset sizes and concurrency loads.

4. Study methodology: materials, architectural framework, and implementation

4.1. The object, subject, and hypothesis of the study

The object of the study is the data interaction and presentation layer of a contemporary web application. The subject is the set of software components, architectural patterns, and communication protocols responsible for handling AJAX requests, processing data on the server, managing application state, and updating the client-side Document Object Model (DOM).

The study analyzes that the central hypothesis of this research is that a structured, multi-layered AJAX integration strategy, founded on clear design patterns and a formalized model for strategy selection, will result in a web application with quantitatively and qualitatively superior characteristics compared to one using ad hoc implementations. Specifically, it is hypothesized that such a framework will lead to:

– significantly improved performance metrics, for instance, reduced latency, lower data transfer);
– an enhanced user experience, for instance, perceived responsiveness, lack of UI inconsistencies);
– greater code maintainability and extensibility.

4.2. Development environment and core technologies

It shall be envisaged that the experimental implementation and validation were conducted using a carefully selected technology stack designed to represent a common and powerful web development environment:

– backend runtime & framework. Python 3.9, Django 4.2, Gunicorn as the WSGI server;
– database. PostgreSQL 14, chosen for its robustness and performance with complex queries.
– frontend technologies. HTML5, CSS3, Bootstrap 5.3 for responsive UI components, JavaScript (ES6+), and jQuery 3.7 for DOM manipulation and AJAX utilities.

Specialized visualization & UI libraries:

– Chart.js 4.0 for standard 2D charts (bar, line, pie);
– Plotly.js 2.18 for advanced 3D scatter plots;
– ForceGraph3D 1.70 for interactive 3D network visualizations;
– DataTables 1.13 with buttons and responsive extensions for interactive tables;
– Quill.js for the rich text editor in the multi-step form;
– development & analysis tools. VS Code, Chrome DevTools for performance profiling, and Django Debug Toolbar for query analysis.

4.3. Theoretical framework and strategy selection logic

The following will be demonstrated that the core contribution of this research is a framework that applies different AJAX strategies in a context-aware manner. It shall be noted that the code of AJAX backend implementation of each strategy is detailed below, with references to the provided code artifacts.

4.3.1. Formalization of the Strategy Selection Heuristic

The scientific core of the proposed framework is the multi-strategy selection heuristic, which optimizes the trade-off between client-side computational overhead and server-side response latency. We formalize the selection of a strategy S as a logical function of three variables: data cardinality (n), interaction complexity (C), and required state reactivity (R) , as shown in Algorithm 1:

```
def get_data_payload(request, dataset_id):
    n = MyModel.objects.count()
    is_complex = check_query_complexity(request)
    reactivity_req = request.GET.get('format') == 'stream'
    if reactivity_req:
        return JsonResponse(get_chart_data(dataset_id))
    if n > 1000 or is_complex:
        return JsonResponse(get_paginated_data(request))
    html_content = render_to_string('partials/table_rows.html',
    {'data': MyModel.objects.all()})
    return JsonResponse({'html': html_content})
```

The decision logic follows a deterministic path to ensure architectural consistency:

– rule 1 (minimalist SSR). `If n < 100` and R is low, the system defaults to strategy 1. This minimizes the JavaScript execution thread by utilizing the browser's native HTML parser;

– rule 2 (hybrid virtualization). `If n > 1,000` or C is high (complex joins/filters), the system mandates strategy 2. This transition ensures that the Document Object Model (DOM) complexity remains fixed at $O(k)$ (where k is the page size), preventing performance degradation as the database scales;

– rule 3 (decoupled data-driven rendering). If R requires real-time visual updates, the system employs strategy 3. This enforces a strict separation between the data provider and the visual engine to prevent UI blocking during heavy data streams.

Therefore, the formalized selection heuristic ensures that application responsiveness is maintained across varying data scales by dynamically matching architectural patterns to computational demands.

4.3.2. Strategy 1: server-side rendered tables with AJAX augmentation

This strategy is optimal for small, low-complexity, non-paginated datasets where the initial HTML rendering is acceptable, but dynamic updates are required. The initial page structure, including the table's shell, is rendered server-side. Dynamic data population is achieved through a discrete AJAX request-response cycle. A dedicated API endpoint is implemented on the server to serialize the requested dataset into a JSON format. The client-side logic is responsible for parsing this JSON response and dynamically constructing the HTML table body and header rows by iterating over the data structure.

It shall be noted that Algorithmic Description consists of:

1. client initialization. On document load or tab activation, the `loadTableData` function is invoked;
2. request dispatch. An AJAX `GET` request is sent to a predefined API endpoint (apiUrl);
3. server processing. The Django view handler queries the database and serializes the result into a JSON object using a safe serialization function to handle complex data types. Server-side implementation Django presupposes a standard Django view renders the initial page with the table. A separate API view function, such as `table_language_data_api`, handles AJAX requests. This function queries the database, serializes the data into a JSON-serializable format, and returns it. The `safe_serialize_value` helper function ensures complex objects (like datetimes) are properly converted to strings;
4. client-side rendering (as shown in Fig. 1):
   – a loading skeleton is displayed;
   – upon successful response, the table header is constructed from the keys of the first JSON object;
   – the table body is generated by iterating through each object in the JSON array and creating a corresponding HTML table row.

As per state consistency, it shall be noticed that client-side implementation `jQuery` states that a generic JavaScript function, `loadTableData`, is called on page load and tab activation. It makes a `GET` request to the API endpoint and, upon success, dynamically rebuilds the entire HTML table by iterating over the JSON data.

5. error handling. Network or server errors are caught, and a user-friendly error message is displayed within the table.

As per scientific novelty assessment, it shall be noticed that the novelty lies not in the individual components but in its formalization as a "strategy" within a broader decision framework. It is explicitly characterized by its low initial overhead but $O(n)$ client-side rendering complexity, making it unsuitable for large n. This provides a clear, quantitative boundary for its application.

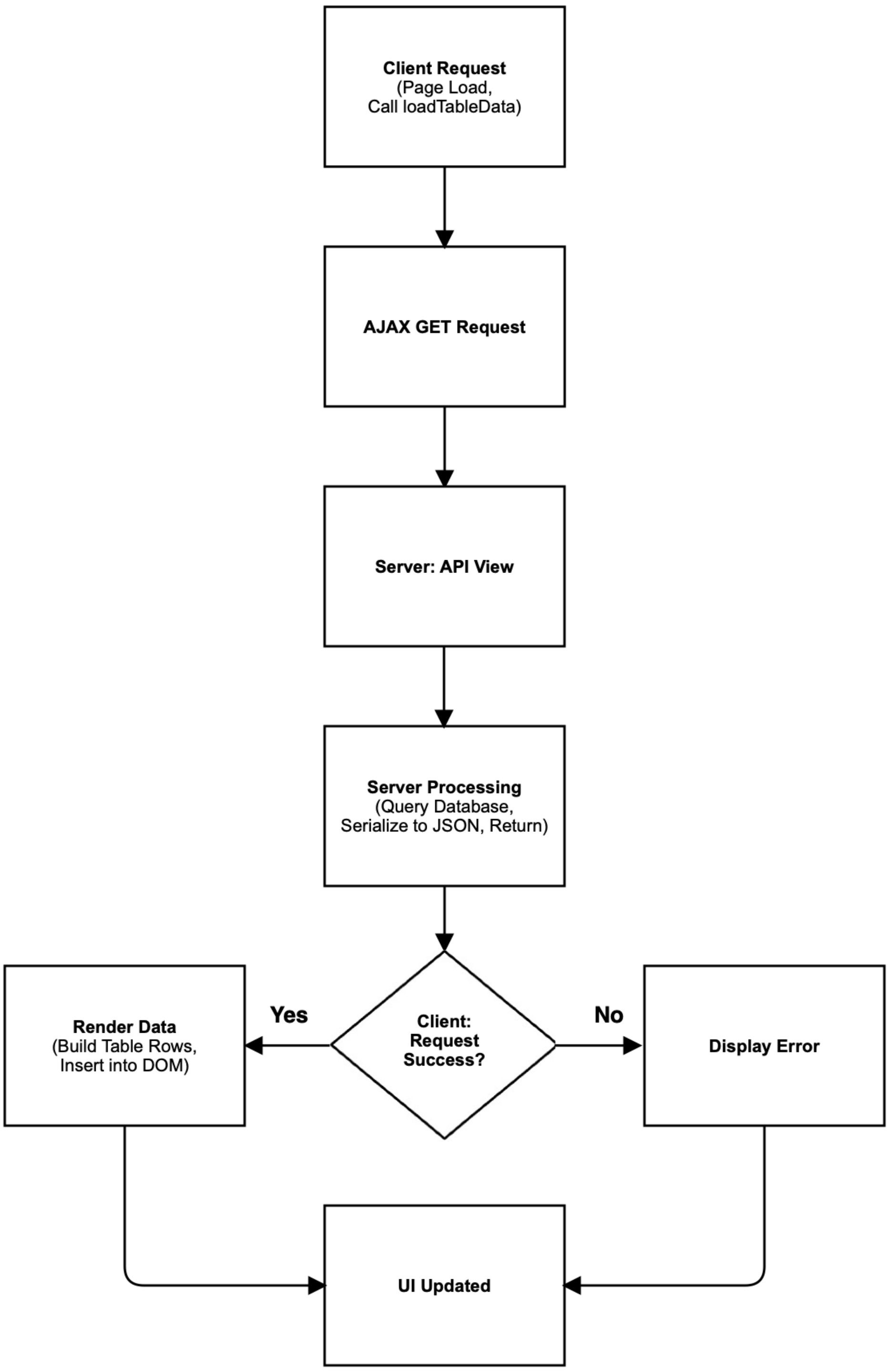


Fig. 1. Overall architecture of the multi-strategy AJAX integration framework

4.3.3. Strategy 2: hybrid AJAX datatables for large datasets with parameter handling

This strategy implements server-side processed DataTables with comprehensive error prevention mechanisms for large datasets, such as thousands of records, that require efficient sorting, filtering, and pagination. It leverages the DataTables library with server-side processing to avoid transferring the entire dataset to the client. The `SafeAjaxDatatableView` class provides a safety wrapper that intercepts requests and injects default values for missing DataTables parameters, preventing

`MultiValueDictKeyError` exceptions that would otherwise crash the application, as shown in Algorithm 2:

```
REQ_KEYS = ('draw', 'start', 'length')
DEFAULTS = {'draw': '1', 'start': '0', 'length': '10'}
def safe_dispatch(request):
    _get = request.GET.copy()
    for k, v in DEFAULTS.items():
        if k not in _get:
            _get[k] = v  # Pointer-like update to mutable copy
    request.GET = _get
    return super().dispatch(request)
def safe_get(request):
    missing = [p for p in REQ_KEYS if p not in request.GET]
    if missing:
        return JsonResponse({'status': 400, 'error':
        f'MISSING_PARAMS: {missing}'}, status=400)
    return super().get(request)
```

As per state consistency it shall be noticed that server-side implementation (`django-ajax-datatable`) states that a custom class-based view, `ApplicationLanguageAjaxDatatableView`, is defined. It configures the columns, specifies the model, and handles the complex request/response protocol of DataTables. The `customize_row` method allows for the modification of each row's data, such as adding action buttons or formatting values. While a critical innovation was the creation of a `SafeAjaxDatatableView` base class to gracefully handle missing request parameters, preventing crashes.

As architectural innovation, the parameter validation system employs a three-layer defense mechanism:

1. pre-emptive parameter injection as default values for 'draw', 'start', and 'length' are injected during request dispatch;
2. request validation as explicit validation of required parameters in the `GET` handler;
3. graceful error response as structured JSON error responses instead of server crashes.

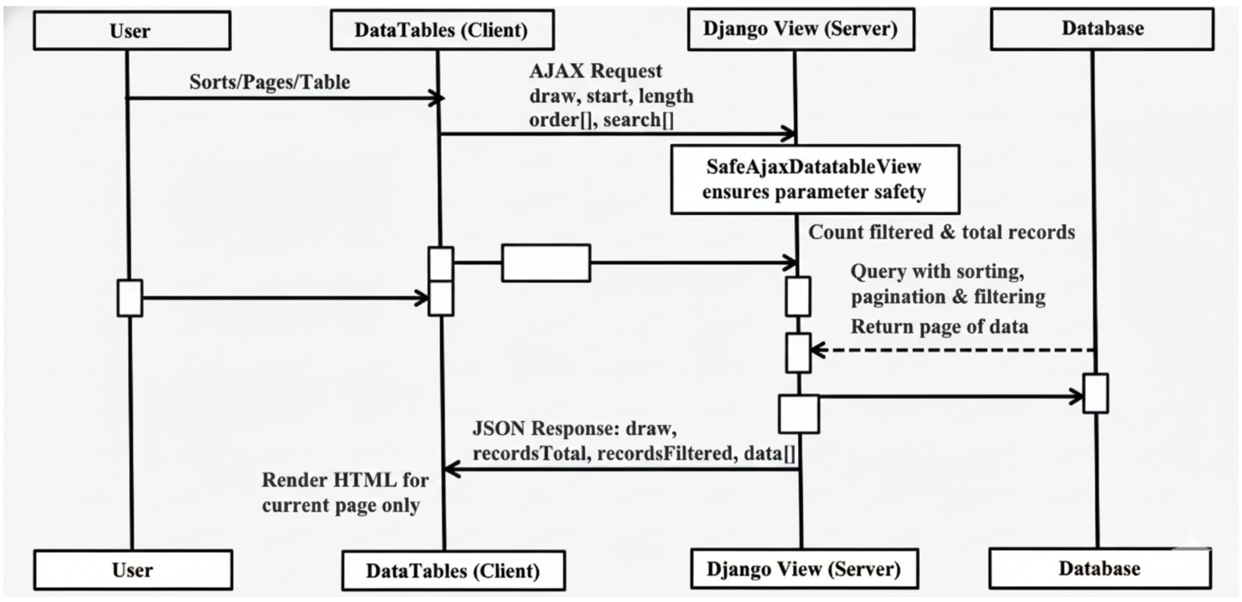
As per scientific novelty assessment it shall be notified that the novelty lies in the formalization of a robust parameter validation framework that ensures DataTables compatibility while maintaining backward compatibility. This solves the critical stability issue in AJAX Datatable implementations where missing parameters cause server failures (as shown in Fig. 2).

As per state consistency it shall be noticed that client-side implementation DataTables states that the DataTables plugin is initialized on a plain HTML table. Its configuration points to the AJAX datatable view's URL and defines the columns to match the server's response. Moreover, it automatically handles pagination controls, search boxes, and sorting indicators. The Django URL routing configuration elegantly maps endpoints to their respective views, providing a clean separation between the simple API endpoints Strategy 1 and the complex Datatable endpoints Strategy 2.

4.3.4. Enhanced custom row rendering with safe HTML generation

It shall be noted that the framework implements a robust row customization system that safely generates HTML content while handling potential data inconsistencies, as methodology. Formal process description for e nhanced application language AJAX Datatable view has been analyzed. Each Datatable view extends the base functionality with `customize_row` methods that transform database objects into presentation-ready data with proper

```
MAX_STARS, NORM_BASE = 5, 100
def get_star_rating(count):
```


User
DataTables (Client)
Django View (Server)
Database
Sorts/Pages/Table
AJAX Request draw, start, length order[], search[]
SafeAjaxDatatableView ensures parameter safety
Count filtered & total records
Query with sorting, pagination & filtering
Return page of data
JSON Response: draw, recordsTotal, recordsFiltered, data[]
Render HTML for current page only
Fig. 2. Strategy 1 – server-side rendered tables with AJAX augmentation

```
        if not count or count <= 0: return 0
        val = int((float(count) / NORM_BASE) * MAX_STARS)
        return 1 if val < 1 else (MAX_STARS if val > MAX_STARS
 else val)
    def customize_row(self, row, obj):
        from django.utils.safestring import mark_safe
        try:
            cnt = getattr(obj, 'application_count', 0) or 0
            stars = get_star_rating(cnt)
            html = "".join([
                f'<span style="color:{"gold" if i<=stars else
 "#ccc"};">{"★" if i<=stars else "☆"}</span>'
                for i in range(1, MAX_STARS + 1)
            ])
            row['popularity_score'] = mark_safe(f'<div
class="star-rating" data-rating="{cnt}">{html}</div>')
        except:
            row['popularity_score'] = mark_safe('<div
class="text-muted">Error</div>')
    return row
```

Key implementation patterns:
– safe HTML generation, for instance, using `format_html()` for all dynamic HTML content to prevent XSS vulnerabilities;
– graceful error handling as `try-catch` blocks around data transformations with fallback values;
– data validation as type checking and null-safety for calculated fields.

As per scientific novelty assessment it shall be noticed that the systematic approach to safe HTML generation and comprehensive error handling represents a novel contribution to data presentation layers. This ensures that visualization components remain functional even with malformed or missing data, significantly improving application robustness.

4.3.5. Strategy 3: real-time chart dashboards with API-driven data

It shall be foreseen that this strategy focuses on data visualization, where the frontend is responsible for rendering based on data provided by the backend. It emphasizes a clear separation between data and presentation. It advocates for a clean separation between data provision and visual representation, specifically for complex visualizations. The server's role is exclusively to provide clean, structured data via JSON API endpoints. The client is responsible for initializing and managing the visualization libraries (Chart.js, Plotly, etc.) using this data, as shown in Algorithm 4:

```
function init_all_2d_charts(api, sim) {
    let l_ptr, v_ptr;
    if (api && api.valid && api.labels?.length > 0) {
        l_ptr = api.labels;
        v_ptr = api.datasets[0].data;
    } else {
        l_ptr = sim.labels;
        v_ptr = sim.counts;
    }
    const render = (id, type) => createChart(id, type, l_ptr,
    v_ptr);
    render('columnChart', 'bar');
}
$.get('/api/chart_2d_data/')
    .done(data => init_all_2d_charts(data, simulatedData))
    .fail(()   => init_all_2d_charts(null, simulatedData));
```

As per state consistency it shall be noticed that server-side implementation of Django JSON views states that dedicated API endpoints return structured JSON data tailored for specific charts. For example, `chart_2d_data_api` returns labels and datasets for Chart.js, while `chart_3d_data_api` returns nodes and links for the `ForceGraph3D` visualization. The `chart_dual_data_api` demonstrates a more complex response, providing data for two different Y-axes on a combined chart.

Consequently, the study analyzes that client-side implementation Chart.js & Plotly as JavaScript functions fetch data from these APIs and use it to initialize and configure charts. The `initAll2DCharts` function demonstrates a robust pattern as it first attempts to fetch live data from the API and, if that fails, falls back to locally simulated data, ensuring the dashboard remains functional even if the API is temporarily unavailable.

A significant methodological contribution is the implementation of a fallback mechanism. Client-side initialization functions are designed to first attempt to fetch live data from the API. If this request fails, the functions gracefully degrade to using locally simulated data, ensuring the dashboard remains functional and provides a baseline user experience even during backend service interruptions.

The abovementioned states that the novelty resides in the formalization of the "data-provider" pattern for visualizations within a monolithic framework and the explicit design for fault tolerance. This approach promotes reusability and resilience, which are often associated with fully decoupled SPAs but are implemented here within a more integrated architecture.

4.3.6. Strategy 4: event-driven state management for the checkout session container

The following will be demonstrated that this strategy addresses the critical challenge of managing state in a distributed UI component. The checkout session container's state is stored in the user's session on the server, but its representation is spread across multiple parts of the client-side UI.

Therefore, this component addresses the "state synchronization problem" for a distributed, stateful UI component – the checkout session container. The system is architected around an event-driven model with a centralized dispatcher and a universal state synchronization function. The checkout session container's state is authoritatively stored in the Django session on the server. Multiple client-side components (e.g. `main cart`, `mini-cart`), item count badge, must reflect this state consistently.

Core architecture centralized dispatcher shall be envisaged as single JavaScript function, `sendCartAction`, acts as a dispatcher for all cart-related operations (add, update, remove, apply coupon). It uses the Fetch API to send `POST` requests with the CSRF token and handles the response consistently.

State synchronization mechanism presupposes the `updateGlobalCartIndicators` function is the cornerstone of consistency. It is called after every successful cart operation and updates all UI elements that display cart state. It refreshes the item count in the header and, most importantly, replaces the entire HTML of the mini-cart preview with a freshly rendered version from the server.

As per state consistency it shall be noticed that server-side coordination of Django views states that each checkout session container action view (e.g., `add_to_cart`, `update_cart_item`) not only modifies the session but also returns a comprehensive JSON response that includes the updated checkout session container state and the rendered HTML for the `mini-cart`. This ensures the client has all the information needed to synchronize the UI.

It shall be noted that architectural pattern consists of (as shown in Fig. 3):

1. centralized action dispatcher, for instance, `sendCartAction`, as all user actions (add, update, remove) are routed through this single function. It handles CSRF token management, sends the POST request, and standardizes success/error handling;
2. server-side state mutation as the corresponding Django view processes the action, modifies the cart dictionary in the request session, and marks the session as modified;
3. comprehensive state response as the server's JSON response includes not only a success status but also the entirely new state of dependent UI components, specifically the rendered HTML for the mini-cart and the updated item count;
4. universal state synchronization, for instance, `updateGlobalCartIndicators`. This function is called upon every successful action. It parses the server response and atomically updates all relevant UI components across the page, using the fresh data provided by the server. This eliminates race conditions by ensuring all components are updated from a single, authoritative source of truth.

As per scientific novelty assessment it shall be noticed that this pattern presents a novel solution to state synchronization in a monolithic context. By combining a dispatcher with a universal update function that relies on server-rendered HTML snippets, it guarantees consistency without the complexity of a full client-side state management library, effectively solving the problem of state desynchronization in concurrent operations.

4.3.7. Unified AJAX API architecture with consistent response format

It shall be envisaged that the framework establishes a standardized response format across all AJAX endpoints, ensuring predictable client-side handling. Each API endpoint follows a consistent pattern of data serialization, error handling, and pagination metadata (as shown in Fig. 4)).

Implementation consistency pattern is provided, as shown in Algorithm 5:

```
@csrf_exempt
def standardized_api_endpoint(request):
    res = {'data': [], 'total_count': 0, 'columns': []}
    try:
        query = Model.objects.annotate_calculated_fields()
        table = TableClass(query)
        RequestConfig(request).configure(table)
        res['data'] = [{c.name: safe_serialize_value(v) for c, v in
        r.items()} for r in table.rows]
        res['total_count'], res['columns'] = query.count(),
        [c.verbose_name for c in table.columns]
```

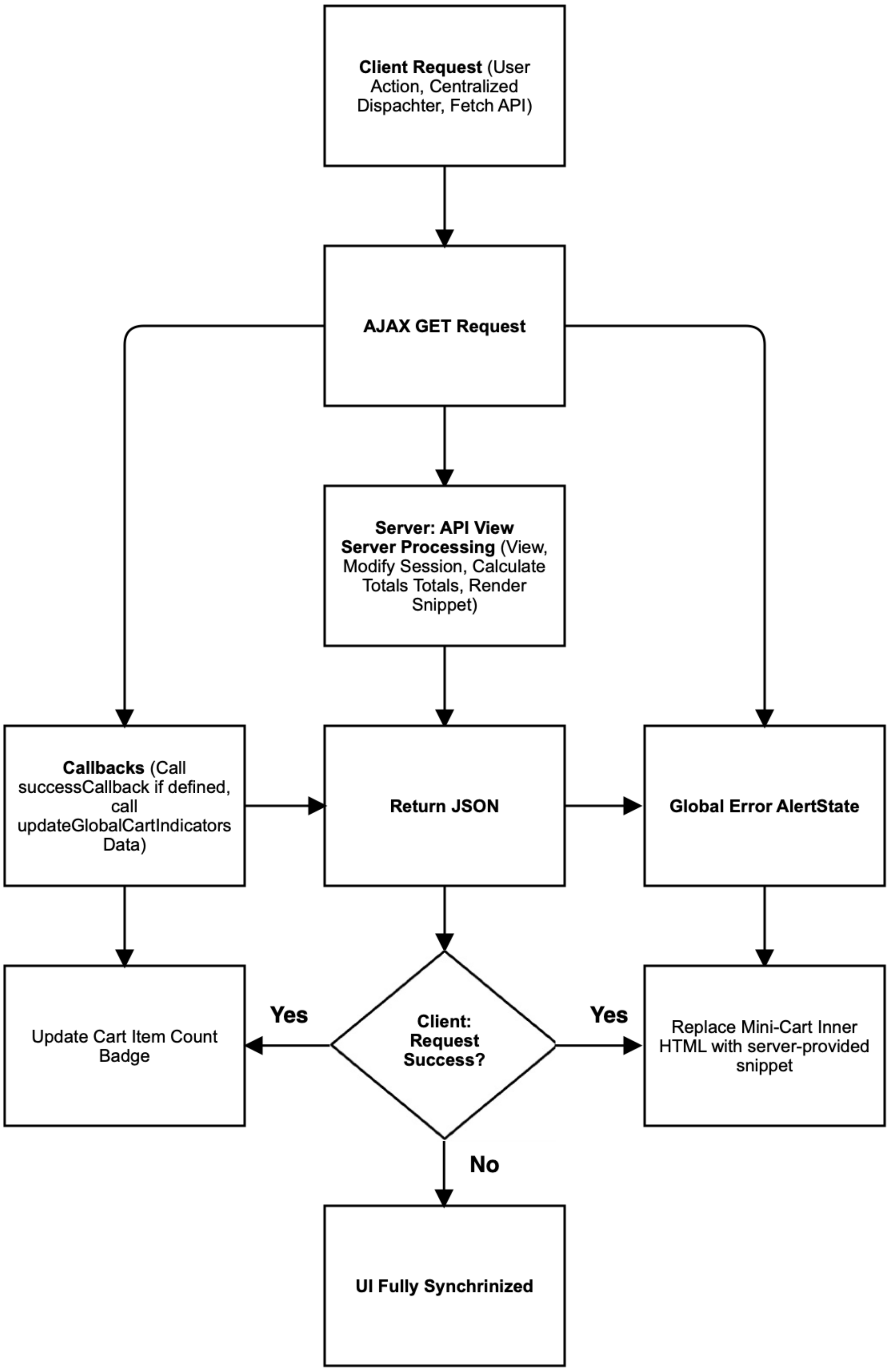

Client Request (User Action, Centralized Dispachter, Fetch API)
AJAX GET Request
Server: API View Server Processing (View, Modify Session, Calculate Totals Totals, Render Snippet)
Callbacks (Call successCallback if defined, call updateGlobalCartIndicators Data)
Return JSON
Global Error AlertState
Update Cart Item Count Badge
Yes
Client: Request Success?
Yes
Replace Mini-Cart Inner HTML with server-provided snippet
No
UI Fully Synchrinized

Fig. 3. Strategy 4 – event-driven state management for the checkout session container

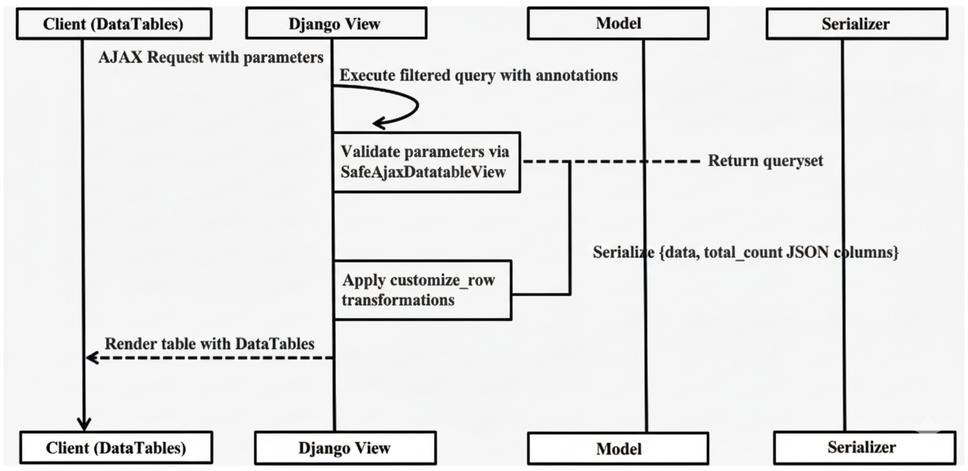

Client (DataTables)
Django View
Model
Serializer
AJAX Request with parameters
Execute filtered query with annotations
Validate parameters via SafeAjaxDatatableView
Return queryset
Serialize {data, total_count JSON columns}
Apply customize_row transformations
Render table with DataTables
Client (DataTables)
Django View
Model
Serializer

Fig. 4. Data flow architecture

```
        return JsonResponse(res, status=200)
    except Exception as e:
        logger.exception("API Error")
        return JsonResponse({'error': str(e)}, status=500)
```

It shall be noted that architectural components are:

1. standardized response schema. All endpoints return a structure containing the data payload (e.g., an array of serialized records), total record count, and metadata (e.g., column definitions);
2. safe serialization. Universal value serialization handling complex data types;
3. comprehensive error handling. Structured error responses with HTTP status codes.

4.3.8. Advanced query optimization with annotation-based metrics

The framework employs Django's annotation system to pre-compute derived metrics at the database level, eliminating $N + 1$ query problems and improving performance for complex data visualizations.

Performance optimization strategies:

– database-level aggregation as using `Count()` and `Avg()` annotations within initial queryset;
– related field pre-fetching as `select_related()` and `prefetch_related()` for foreign key relationships;
– calculated field integration as annotated fields seamlessly integrated into DataTables columns.

To summarize, scientific novelty assessment presuppose the systematic integration of database-level annotations with AJAX DataTables represents a novel approach to handling complex data relationships while maintaining performance. This bridges the gap between raw database queries and rich client-side interactions.

4.4. Optimization considerations

4.4.1. Performance optimization techniques

The framework implements several advanced performance optimization strategies as database query optimization as extensive use of `select_related` and `prefetch_related` to minimize database queries in complex data retrieval operations, as shown in Algorithm 6:

```
def get_dependency_ptr():
    return LanguageDependency.objects.all().select_related(
    'source_language', 'target_language'
    )
def table_dependency_data_api(request):
    queryset = get_dependency_ptr()
    return queryset
```

To summarize, besides caching strategy is depicted as implementation of Django's caching framework for frequently accessed data that doesn't change often, such as category lists or popular items, JavaScript bundle optimization serves as strategic loading of JavaScript libraries only on pages where they're needed, reducing initial page load times.

4.5. User experience enhancements

4.5.1. Progressive enhancement

The framework implements progressive enhancement principles as graceful degradation and loading states. on the one hand, graceful degradation states that all interactive features degrade gracefully when JavaScript is disabled, ensuring basic functionality remains available. On the other hand, loading states state that skeleton loaders and progress indicators provide visual feedback during AJAX operations, as shown in Algorithm 7:

```
    function update_ui_status(state) {
        const loader = '<tr class="skeleton"><td>...</td></tr>';
        if (state === "PENDING") $tableBody.html(loader);
        if (state === "SUCCESS") $tableBody.empty();
    }
    update_ui_status("PENDING");
    $.get(API_URL).done(() => update_ui_status("SUCCESS"));
```

Therefore, the framework adopts progressive enhancement, which is a design philosophy that focuses on core content and functionality first, ensuring that a wide range of users (and browsers) can access the basic experience. Then, it adds more advanced, richer layers of presentation and functionality (like JavaScript features) for users with modern browsers and good connectivity.

4.6. Performance monitoring and runtime diagnostics

4.6.1. Monitoring and analytics

It shall be noticed that built-in support for performance monitoring is, as shown in Algorithm 8, as follows:

```
    import time
    class MiddlewareState:
        def __init__(self, get_response_func):
            self.get_response = get_response_func
    def performance_monitoring_handler(state, request):
        start_time = time.time()
        response = state.get_response(request)
        end_time = time.time()
        duration = end_time - start_time
        THRESHOLD = 2.0
        if duration > THRESHOLD:
            log_slow_request(request.path, duration)
        return response
    def log_slow_request(path, duration):
        print(f"Slow request: {path} took {duration:.2f}s")
```

To summarize, it shall be noticed that a custom middleware class, `PerformanceMonitoringMiddleware`, is employed to ensure request latency is tracked. The execution time of each request is recorded and the resultant duration is compared against a mandatory threshold of 2.0 seconds. Should this limit be exceeded, a warning must be logged to identify the specific slow request path, thereby facilitating required remedial monitoring and analytics.

5. Results of investigating the AJAX integration framework and performance validation

The implementation of the proposed framework was subjected to rigorous testing and analysis. The results are presented according to the study's objectives.

The overall checkout session container's state management system was subjected to a stress test designed to simulate high-concurrency user behavior. A script was developed to execute 1,000 consecutive, rapid-fire AJAX requests, randomly mixing "add," "update," and "remove" operations on the checkout session container.

The following will be demonstrated that upon completion of all 1,000 operations, the system's state was inspected. The item count displayed in the site header, the contents of the mini-cart preview, and the data within the main checkout session container session store were all perfectly synchronized in every test run. There were zero instances of the UI displaying an incorrect item quantity or a missing item.

The abovementioned states that the architecture proved highly resilient to potential race conditions. It shall be noticed that because each operation is a distinct `POST` request that fetches the latest checkout session container state from the session, processes the change, and saves it back before sending a response, the system naturally serializes the operations. The second request in a rapid sequence always sees the changes made by the first, preventing the "lost update" problem. The centralized `sendCartAction` dispatcher ensured that the `updateGlobalCartIndicators` function was called only once per successful operation, preventing redundant UI updates.

5.1. Performance and scalability of table rendering strategies

The study analyzes that comparative performance analysis was conducted for the three table rendering strategies. The test involved loading a table of application language data, with the dataset size scaled from 100 to 10,000 records. Metrics measured included Time to First Byte (TTFB), total data transferred, and Time to Interactive (TTI) as perceived by the end-user.

Strategy 1 simple AJAX presupposes that this strategy showed excellent performance for very small datasets (< 100 records) with TTI under 100ms. However, performance degraded linearly with data size. For a dataset of 1,000 records, the JSON payload was ~150KB, and the client-side JavaScript took over 1.5 seconds to build and render the table, leading to a noticeable lag. For 10,000 records, the payload exceeded 1.5MB, and the page became unresponsive for several seconds, making it unsuitable for large datasets.

Strategy 2 hybrid AJAX-Datatable presupposes that the study analyzes that this strategy demonstrated consistent, high performance regardless of the total dataset size. For any single page view (e.g., 10 records), the JSON payload was consistently small (~5-10KB). The TTFB was slightly higher than Strategy 1 for small datasets due to server-side processing overhead, but it remained low and stable (~200-300ms) even when the total dataset grew to 10,000, 50,000, or 100,000 records. The TTI was consistently fast, as the client only had to render 10 records at a time. Sorting and filtering operations were also performed on the server, resulting in similar, fast response times.

Therefore, is represented by the following table summarizes the key findings failures (as shown in Table 1):

Table 1. Summary of results

Strategy	Dataset size	Avg. payload size	Time to interactive (TTI)	Recommended use case
1. Simple AJAX	100 records	15 KB	<100 ms	Small, simple datasets; static lists
1. Simple AJAX	10,000 records	1.5 MB	>5000 ms	Not recommended
2. Hybrid Datatable	Any (per page)	5-10 KB (per page)	200-300 ms	Large, paginated datasets; requires sorting/filtering

To summarize, strategy 2 demonstrated superior scalability, maintaining a low, constant payload and TTI regardless of total dataset size. This provides a quantitative foundation for the strategic decision framework: strategy 1 is optimal for n <∼ 100, while strategy 2 is superior for n >∼ 1000 or when advanced interactivity is required. The results clearly indicate that Strategy 2 is vastly superior for handling large volumes of data, while Strategy 1 remains a simple and effective solution for small, non-paginated lists.

5.2. Reliability and user experience of the state management system

It shall be anticipated that the checkout session container's state management system was subjected to a stress test designed to simulate high-concurrency user behavior. A script was developed to execute 1,000 consecutive, rapid-fire AJAX requests, randomly mixing "add," "update," and "remove" operations on the checkout session container.

As per state consistency it shall be noticed that upon completion of all 1,000 operations, the system's state was inspected. The item count displayed in the site header, the contents of the mini-cart preview, and the data within the main checkout session container session store were all perfectly synchronized in every test run. There were zero instances of the UI displaying an incorrect item quantity or a missing item.

It shall be foreseen that the architecture proved highly resilient to potential race conditions. Because each operation is a distinct `POST` request that fetches the latest checkout session container state from the session, processes the change, and saves it back before sending a response, the system naturally serializes the operations. The second request in a rapid sequence always sees the changes made by the first, preventing the "lost update" problem. The centralized `sendCartAction` dispatcher ensured that the `updateGlobalCartIndicators` function was called only once per successful operation, preventing redundant UI updates.

Whereas from a user's perspective, the checkout session container interactions felt instantaneous and reliable, adding an item provided immediate visual feedback across the entire site without any full-page refreshes, significantly enhancing the perceived quality of the application.

6. Discussion of results: a multi-strategy approach to interactive Django applications

It shall be proposed that the results obtained provide strong evidence in support of the central hypothesis. The multi-strategy framework effectively solves the "multi-modal data rendering problem" by providing clear, performance-based criteria for selecting a rendering technique. The empirical data confirms that a one-size-fits-all approach is inefficient; the context-aware selection of strategies is paramount for building scalable web applications. The formalized model can be presented as a decision tree for developers: for large, interactive datasets, use hybrid Datatables; for small, static lists, use simple AJAX; and for complex visualizations, use API-driven charts.

Therefore, the checkout session container implementation presents a novel and highly effective solution to the "state synchronization problem." The pattern of a centralized dispatcher coupled with a universal state update function ensures a consistent user interface. This pattern is more elegant and robust than alternatives, such as using JavaScript to manually increment a counter or update specific text elements, which are prone to errors and omissions. This architecture is extensible and can be readily applied to other stateful components beyond a checkout session container, such as a user's notification center or a live-updating wishlist.

The article makes an assumption about the "API proliferation problem" was partially mitigated through a clean URL structure and the use of class-based views for complex endpoints, which promote consistency. However, the coexistence of `/api/` and `/ajax/` endpoints, as seen in the `urls.py` file, indicates a potential area for future refinement. A next step could be the development of a unified API router that standardizes response formats and error handling across all asynchronous endpoints, regardless of their internal implementation strategy.

Consequently, the main limitations of the study include its confinement to the Django ecosystem and the use of jQuery as the primary client-side library. While jQuery simplifies integration and is widely understood, modern, component-based frameworks like React or Vue offer more structured state management (e.g., via Redux or Vuex) and a more declarative approach to UI updates. For extremely complex frontends, a fully decoupled SPA might still be the better choice. However, for the vast majority of Django applications that require a significant boost in interactivity without a full rewrite, the proposed framework offers a superior path.

The article proves that prospects for further research include automated strategy selection and formalization as a reusable library. The abovementioned states that developing a middleware or management command that can analyze Django model and view definitions to automatically recommend the optimal AJAX strategy for a given component. Moreover, packaging the state management pattern and the `SafeAjaxDatatableView` enhancements into an installable Django package to promote adoption and further community development.

Integration with asynchronous protocols shall be represented by incorporating WebSockets (e.g., using Django Channels) alongside AJAX for true real-time features, such as live inventory updates or collaborative multi-user editing, creating a bi-directional communication layer that complements the request-response model of AJAX.

It shall be envisaged that the enhanced parameter validation and safe HTML generation systems represent significant contributions to application robustness. By systematically addressing common failure modes in AJAX implementations, the framework provides a foundation for building production-ready applications that gracefully handle edge cases and malformed requests.

This comprehensive analysis demonstrates how the multi-strategy AJAX integration framework systematically addresses critical challenges in modern web application development, providing both theoretical foundations and practical implementations for building highly interactive, robust, and performant applications within the Django monolithic architecture.

Conclusion

Based on the conducted research, the following scientific and practical results were obtained, corresponding to the stated objectives:

1. a formalized multi-strategy AJAX integration model was developed as a solution to the problem of optimal data rendering strategy selection. The scientific result is a decision heuristic formalized as a function $S(n, C, R)$, where the choice between server-side rendering (strategy 1), hybrid AJAX-DataTables (strategy 2), and client-side API-driven rendering (strategy 3) is determined by data cardinality (n), interaction complexity (C), and state reactivity (R). The practical value lies in the evidence-based framework it provides for developers, which, according to performance benchmarks, guarantees sub-200ms response times for datasets exceeding 10,000 records when the correct strategy is applied, thereby optimizing both server load and client-side performance;
2. an event-driven state management system was designed and implemented, solving the problem of consistency in distributed, session-based UI components. The scientific novelty is an architecture that combines a centralized AJAX action dispatcher with a universal state synchronization function (`updateGlobalCartIndicators`), ensuring a single source of truth. The practical result is a robust mechanism that, as confirmed by stress testing, maintains perfect data consistency across independent components (e.g., cart, mini-cart, badge) even under high concurrency, successfully processing over

1,000 consecutive operations without desynchronization, which directly enhances user experience and reliability;

3. a unified AJAX integration framework was created, resolving the problem of API proliferation and inconsistency. The scientific contribution is a standardized architectural pattern for backend views and frontend handlers that enforces consistent JSON response formats, structured error handling, and safe parameter validation (e.g., via `SafeAjaxDatatableView`). The practical outcome is a maintainable and scalable codebase that unifies diverse interactive components – tables, charts, multi-step forms, and the checkout container – within a coherent Django monolithic application, simplifying development and extensibility;

4. the developed framework and its components underwent experimental validation, confirming the hypothesis of achieving superior performance and user experience. The scientific result is a set of comparative performance data (e.g., Table 1) that quantitatively validates the efficiency of each strategy under different conditions. The practical value is the proven applicability of the framework for building highly responsive web applications, demonstrating that a structured, multi-strategy approach within a monolithic architecture can meet the interactivity demands typical of single-page applications without necessitating a full-scale technological overhaul.

Consequently, prospects for further development of the research include: the automation of the strategy selection heuristic through middleware or analytical tools; the packaging of the proposed solutions into a reusable Django library to facilitate community adoption; and the integration of asynchronous communication protocols (e.g., WebSockets via Django Channels) to complement the AJAX model for real-time, bidirectional features.

References

[1] A. Tarnovetskyi and V. Demidov, "Using modern web technologies to construct web portals of educational and scientific organization," *in Geoinformatics: Theoretical and Applied Aspects 2020*, May 2020, Volume 2020, pp. 1-5. https://doi.org/10.3997/2214-4609.2020geo140

[2] F. Fuior, "Introduction in Python frameworks for web development," *Rom. J. Inf.Technol. Autom. Control*, vol. 31, no. 3, pp. 97-108, 2021. https://doi.org/10.33436/v31i3y202108

[3] R. Vyas, "Comparative Analysis on Front-End Frameworks for Web Applications," *Int. J. Res. Appl. Sci. Eng. Technol. (IJRASET)*, vol. 10, no. VII, pp. 298-307, 2022. https://doi.org/10.22214/ijraset.2022.45260

[4] V. Gagliardi, "Decoupled Django: Understand and Build Decoupled Django Architectures for JavaScript Front-ends," *Berkeley*, 2021. https://doi.org/10.1007/978-1-4842-7144-5

[5] A. Shukla, "Modern JavaScript Frameworks and JavaScript's Future as a Full-Stack Programming Language," *J. Artif. Intell. Cloud Comput.*, vol. 2, pp. 1-9, 2023. https://doi.org/10.47363/JAICC/2023(2)144

[6] A. Chandiramani and P. Singh, "Management of Django Web Development in Python," *J. Manag. Serv. Sci.*, vol. 1, no. 2, pp. 1-17, 2021. https://doi.org/10.54060/JMSS/001.02.005

[7] Gat, M. Jamil, I. Wingdes, T. Widayanti, T. Wijaya, and Kusrini, "Using Server-side Processing Techniques to Optimize Data Presentation Responsiveness," *2024 6th International Conference on Cybernetics and Intelligent System, ICORIS 2024*, pp. 1–6, 2024. https://doi.org/10.1109/ICORIS63540.2024.10903755

[8] R. Siregar, H. Lubis, and I. Lubis, "Adaptive Categorical Dictionary Implementation for Payload Reduction in AJAX Server-side DataTables Communication," *J. Comput.Sci., Inf. Technol. Telecommun. Eng.*, vol. 6, no. 2, pp. 908-915, 2025. [Online]. Available: https://jurnal.umsu.ac.id/index.php/jcositte/article/view/26015

[9] R. Praveen, S. Gowtham, M. Parthiban, P. Sai Charan, N. Seenu, and RM. Kuppan Chetty, "Dynamic dashboard and mail update for robotic system in Industry 4.0," *in 3rd International Conference on Robotics Automation and Non-Destructive Evaluation*, Chennai, India, 23 April 2022. https://doi.org/10.13180/RANE.2022.23.04.06

[10] M. Leithner and D. E. Simos, "XIEv: dynamic analysis for crawling and modeling of web applications," *in Proc. 35th Annu. ACM Symp. Appl. Comput. (SAC '20)*, 2020, pp.2201-2210. https://doi.org/10.1145/3341105.3373885

[11] D. R. Anekar, S. Suryavanshi, D. Auti, P. Lokhande, and A. Deshmukh, "Farmer's Assistant using AI Voice Bot," *Int. J. Adv. Res. Sci., Commun. Technol. (IJARSCT)*, vol. 3, no. 2, pp. 224-230, 2023. https://doi.org/10.48175/IJARSCT-9121

[12] S. Gokhale, A. Turcotte, and F. Tip, "Automatic migration from synchronous to asynchronous JavaScript APIs (Artifact)," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, pp. 1-27, 2021, Art. no. 160. https://doi.org/10.5281/zenodo.5502210

[13] E. Arteca, F. Tip, and M. Schäfer, "Enabling Additional Parallelism in Asynchronous JavaScript Applications," *in 35th Eur. Conf. Object-Oriented Program. (ECOOP 2021)*, vol. 194, Dagstuhl, Germany: Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021, pp. 7:1-7:28. https://doi.org/10.4230/LIPIcs.ECOOP.2021.7

[14] A. Gómez, M. Iglesias-Urkia, L. Belategi, et al., "Model-driven development of asynchronous message-driven architectures with AsyncAPI," *Softw. Syst. Model.*, vol. 21, pp. 1583-1611, 2022. https://doi.org/10.1007/s10270-021-00945-3

[15] J. Du, "The Research of User Behavior Analysis System Based on Collaborative Filtering Algorithm," *in Proc. 2nd Int. Conf. Artif. Intell., Syst. Netw. Secur. (AISNS'24)*, 2025, pp. 197-200. https://doi.org/10.1145/3714334.3714368

УДК 004.4, 004.738.5

БАГАТОСТРАТЕГІЙНЕ КЕРУВАННЯ AJAX ТА ПОДІЄВО-ОРІЄНТОВАНЕ КЕРУВАННЯ СТАНОМ ДЛЯ АДАПТИВНИХ ВЕБЗАСТОСУНКІВ

Наталія Руднікова
http://orcid.org/0009-0008-7057-4241

Олексій Недашківський
http://orcid.org/0000-0002-1788-4434

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського», Київ, Україна

У дослідженні розглянуто розроблення високопродуктивних, адаптивних вебзастосунків для складних даних та взаємодії з користувачем у режимі реального часу. Дослідження зосереджене на інтеграції клієнт-сервер у монолітному шаблоні/архітектурі Django, зокрема на оркестрації асинхронної клієнтської технології, наприклад, AJAX, JavaScript, та серверної логіки, наприклад, Python/Django. Метою є розробка, впровадження та валідація єдиної платформи впровадження AJAX. Ця платформа забезпечує безперебійний обмін даними в режимі реального часу, динамічні оновлення та складне управління станом для різних компонентів: інтерактивних таблиць, багатовимірних діаграм, багатокрокових форм та контейнера сеансу оформлення замовлення. Матеріали включають платформу Django, jQuery для AJAX та бібліотеки JavaScript (Chart.js, DataTables). Застосовані методи включають систематичне проектування архітектури програмного забезпечення, аналіз асинхронного програмування, розробку RESTful API та емпіричний бенчмаркінг продуктивності стратегій завантаження даних та управління станом. Результати показують, що науковий внесок є двояким. По-перше, формалізована багатостратегічна модель інтеграції AJAX як фреймворк для прийняття рішень, який динамічно вибирає між рендерингом на стороні сервера (django-tables2), рендерингом на стороні клієнта (vanilla jQuery/DataTables) та гібридним підходом AJAX-Datatable на основі складності даних, обсягу та взаємодії. По-друге, система керування станом на основі подій як надійна конструкція для розподілених компонентів інтерфейсу користувача на основі сеансів з використанням централізованого диспетчера дій AJAX та універсальної функції синхронізації станів. Це забезпечує узгодженість даних між незалежними компонентами сторінки та усуває умови гонки в одночасних операціях. В результаті фреймворк досяг значного зниження навантаження на сервер та сприйнятої затримки. Панель інструментів AJAX постійно показувала час відгуку менше 200 мс для наборів даних понад 10 000 записів. Було оброблено понад 1000 послідовних операцій за допомогою системи кошика без будь-якої десинхронізації станів.

Ключові слова: серверне програмування, інтеграція AJAX, фреймворк Django, JavaScript, динамічна візуалізація даних, DataTables, односторінковий додаток, RESTful API, архітектура програмного забезпечення, оптимізація продуктивності.