Algorithm for Building Client-Server Applications using Java, SQL, and Reactjs

Olena Komleva
Independent Researcher
Chicago, Illinois, USA
ORCID: 0009-0002-2796-6449

ABSTRACT
Purpose. This article develops a structured algorithm for building client-server applications with Java, SQL and ReactJS. It responds to a recurring weakness in full-stack practice: backend logic, database access, API contracts and frontend state are designed as separate concerns, although users experience their combined quality as one system. Methodology. No primary experiment was conducted. The article applies secondary empirical synthesis to twenty verified English-language sources spanning Java client-server implementation, REST API testing, database access defects, ORM performance, DBMS testing, React refactoring and comparative performance. Every source is coded by implementation layer and mapped to a development stage, and the full mapping is reported as an evidence-coding matrix so that the derivation can be inspected and contested. Results. Reliable development depends on early data-flow clarification, database-aware API design, controlled backend implementation, explicit frontend state organization, contract testing and end-to-end performance verification. On this basis the article proposes the J-SQL-R Algorithm, six consecutive yet iterative stages. Evaluation. A Weighted Development Index is computed for four contrasting project scenarios, a one-point perturbation of the complexity coefficients tests the stability of the resulting ranking, and the sequence is compared with three alternative development orders. Falsification criteria and a reproducible validation protocol are stated. Unique contribution. A research-grounded and testable procedure for Java, SQL and ReactJS projects.

General Terms
Algorithms, Design, Performance, Verification, Measurement, Experimentation, Software Engineering.

Keywords
Client-server applications; Java; SQL; ReactJS; REST API; full-stack development; database access; software engineering; empirical synthesis; application architecture.

1. INTRODUCTION
Client-server development remains one of the most stable and most demanding areas of applied programming. The vocabulary of software engineering has shifted toward cloud deployment, API gateways, single-page applications and performance observability; the underlying architectural problem has not shifted with it. A client issues requests, a server interprets business rules, a database stores and retrieves structured data, and the user interface converts system behavior into intelligible interaction. Combining Java, SQL and ReactJS in one product makes that problem unusually visible, because each layer carries its own logic, failure modes and optimization culture. Final quality depends less on the isolated correctness of the three layers than on the discipline with which they are joined.

The rationale comes from a common weakness in full-stack practice. Many teams begin with screens, then add endpoints, then adjust database tables under schedule pressure; others start from a schema and later discover that the frontend requires different data shapes. The application may still work, but it becomes difficult to test, optimize and extend. Research on REST API testing shows that web APIs demand systematic attention to states, inputs and contracts rather than manual endpoint checks [1, 2, 3]. Studies of Java applications and database access show that correctness and performance problems frequently originate in how application code communicates with persistent storage, not in the database engine or the language alone [4, 5]. React-based systems add a third dimension, since client-side refactoring and architectural decisions govern maintainability [6].

The purpose of this article is to develop a structured algorithm for building client-server applications using Java, SQL and ReactJS. The article reports no field experiment. It treats existing empirical studies as evidence units and converts them into a practical authorial methodology. The position matters: a framework should not present itself as empirical when no original participants, repositories or measurements were examined, yet it can hold a valid empirical dimension when it synthesizes prior findings systematically, states the coding that produced the synthesis, and yields a testable model. Four research questions follow. RQ1: which development factors recur most often in the literature on Java, SQL, REST APIs and React-based systems? RQ2: how can those factors be arranged into a practical algorithm? RQ3: how does the algorithm behave across project types that differ in data volume, regulatory load and interface complexity? RQ4: how can it be formulated so that it remains falsifiable?

The scientific significance of the article lies in integrating knowledge currently scattered across separate research areas: REST API testing, database access and CRUD throughput, frontend refactoring and execution environments, and comparative backend performance. Each strand is internally rigorous and externally isolated, so a team can read all of them and still receive no guidance on the order in which the corresponding decisions should be taken. This article joins those areas into one development sequence.

The article makes four contributions. It reports an evidence-coding matrix mapping every source to a layer, an actionable requirement and a stage, so that the path from published evidence to prescriptive step can be audited rather than assumed. It formulates the J-SQL-R Algorithm, a six-stage sequence with an explicit return path from measurement to schema design. It supplies a Weighted Development Index and applies it to four project scenarios, showing where the effort

bottleneck moves and how stable that position is under perturbation. It states falsification criteria and a validation protocol, converting the proposal from an assertion into a testable claim.

2. LITERATURE REVIEW

The literature on client-server development provides a broad but uneven knowledge base. A directly relevant starting point is the study by Patil and Shirbhate [7] on the design and implementation of a Java-based client-server application. Their work is useful because it holds attention on the core architectural relation between client, server and data exchange. A contemporary application built with Java, SQL and ReactJS requires more than that basic distribution of responsibilities. It needs explicit contracts, database-aware behavior, frontend consistency and testable integration boundaries.

A large share of the current evidence concerns RESTful APIs. Golmohammadi et al. [2] present REST API testing as a mature yet unresolved field and argue that APIs should be treated as behavioral interfaces rather than transport channels. Atlidakis et al. [1] show through RESTler that stateful fuzzing exposes defects which ordinary request-based tests miss, and Martin-Lopez et al. [3] demonstrate with RESTest that API testing can be automated without inspecting the implementation. Karlsson et al. [8] add property-based testing for OpenAPI-described APIs, Alonso et al. [9] address realistic input generation, and Kim et al. [10] report that automated generation still faces unresolved obstacles. The combined lesson is that the API layer must be designed from the outset as a verifiable contract between Java services and React clients.

Database-related literature supplies a second core dimension. Liu et al. [5] show that defects cluster at the boundary between application logic and data persistence, where incorrect assumptions about schema constraints, transactions, result sets or object-relational mapping stay invisible at the interface level until late stages. Chen et al. [4] demonstrate that performance degradation follows from inefficient access patterns concealed behind convenient ORM abstractions. Wieleba and Wieleba [11] compare database work in Spring and Symfony, and Grzeszuk and Miłosz [12] measure CRUD performance in Spring Boot and ASP.NET Core. These studies converge on one point: database design cannot be postponed until after backend implementation, because schema structure and access strategy fix much of the future performance envelope.

Testing of database engines forms a third evidence stream. Zhong et al. [13] propose SQUIRREL for testing database management systems through language validity and coverage feedback, and Li et al. [14] examine DBMS bug detection with context-sensitive instantiation and multi-plan execution. Both establish that SQL systems are complex execution environments, so a client-server algorithm must respect schema constraints, query semantics and optimizer behavior instead of assuming a passive repository.

Frontend and application-architecture research completes the picture. Ferreira et al. [6] identify the maintenance problems that appear when components, state and data flow are left uncontrolled. Markovic et al. [15] show empirically that modern web applications increasingly separate rendering, APIs and deployment concerns. Biørn-Hansen et al. [16] investigate performance overhead in cross-platform frameworks, and Kurant and Bylina [17] demonstrate that JavaScript engines influence hybrid application performance; both support the claim that frontend runtime behavior belongs inside full-stack methodology.

Comparative performance studies ground the article in practical engineering concerns. Wicha and Pańczyk [18] compare REST API technologies using Spring and Express.js and report that implementation choices alter response behavior even when functional requirements look similar. Godinho et al. [19] evaluate RESTful APIs built with .NET and Java Spring Boot, while Shyam Mohan and Goswami [20] compare Node.js and Java Spring Boot. None implies that Java is universally superior; they show that server-side technology must be tested under explicit workload assumptions, endpoint design, serialization choices and access patterns. The research gap is therefore clear. Separate studies on APIs, Java, SQL, React and performance are plentiful, but an integrated algorithm that translates their findings into one coherent procedure is still missing, and no existing proposal states the mapping from evidence to procedural step in a form another researcher could reproduce or refute.

The corpus is uneven, and the imbalance bounds what the synthesis can claim. Six sources address REST API verification [1, 2, 3, 8, 9, 10], five address SQL and database behavior [4, 5, 11, 13, 14], six report comparative backend or client performance [12, 16, 17, 18, 19, 20], two concern frontend architecture [6, 15], and one describes a baseline Java client-server implementation [7]. Seventeen appeared in 2020 or later. React architecture has attracted far less controlled study than REST API testing, which is a property of the published literature rather than of the selection, so conclusions about stage 4 of the proposed algorithm rest on a thinner evidence base than those about stages 2 and 5 and are stated with lower confidence.

3. METHODOLOGY

3.1 Research Design and Source Corpus

This article uses a qualitative conceptual design supported by secondary empirical synthesis. No participants were recruited, no original repository was mined, and no controlled experiment was performed. The analytical material consists of twenty English-language sources selected for relevance to client-server construction with Java, SQL, REST APIs and ReactJS. Sources were admitted when they reported empirical results, a tool evaluation or a systematic comparison bearing on the three technologies or on the interfaces between them, and when they were retrievable through a persistent identifier; opinion pieces, vendor documentation and tutorials were excluded.

3.2 Analytical Categories and Coding Procedure

The sources were analyzed through six categories: server-side construction; REST API reliability, covering stateful fuzzing, black-box and property-based testing and realistic input generation; SQL and database access, covering access bugs, redundant retrieval, ORM behavior and DBMS testing; frontend architecture; full-stack performance, meaning how backend, database and client behavior combine into the user-perceived result; and methodological actionability, that is, whether a study could be translated into a concrete development step.

The procedure ran in four stages. Each source was reviewed for its central contribution, then coded by its most relevant layer: Java backend, SQL persistence, API contract, React client, testing or performance. Coded findings were compared across layers to identify repeated risks: unclear data flows, unstable API contracts, inefficient database access, weak endpoint testing, excessive frontend coupling and late performance verification. A finding entered the algorithm only when it was

actionable, that is, when it could be restated as an instruction a team can follow or decline at a specific point in the sequence; the rest was retained as context. The recurrent patterns were reorganized into the stage structure of Figure 1, whose dashed return path marks the one feedback loop the sequence requires (Fig. 1).

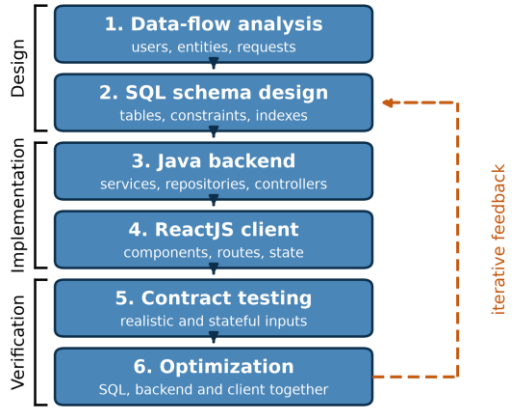

Fig 1: Stage structure of the J-SQL-R Algorithm with iterative feedback from optimization to schema design

Table 1 reports the coding in full: the source, its layer, the actionable requirement extracted from it and the stage that requirement produced. A reader who disputes an assignment can see which stage would change.

Table 1. Evidence-coding matrix: the twenty sources mapped to implementation layers, actionable requirements and stages of the J-SQL-R Algorithm

Ref.	Layer	Actionable requirement extracted	Stage
[1]	API	Exercise endpoints as stateful sequences; defects live in the transitions	5
[2]	API	Treat the API as a behavioral interface with its own verification program	1, 5
[3]	API	Derive tests from the external contract, before implementation is complete	5
[4]	SQL	Model access paths explicitly; ORM hides repeated retrieval with measurable cost	2, 6
[5]	SQL, Java	Fix constraints, transactions and result-set handling before controllers	2, 3
[6]	React	Organize components and state around stable user tasks, not around responses	4
[7]	Java	Separate client, server and data-exchange responsibilities explicitly	1, 3
[8]	API	Keep a machine-readable specification as the source of truth for tests and types	5
[9]	API	Verify endpoints with realistic inputs; random values miss domain failures	5
[10]	API, test	Do not substitute tooling for contract design; generation leaves gaps	1, 5
[11]	SQL	Choose the data-access strategy deliberately; framework choices shift performance	2, 3
[12]	Backend	Benchmark CRUD paths under a stated workload, not by runtime alone	3, 6
[13]	DBMS	Respect the DBMS as an execution environment with its own defect surface	2
[14]	DBMS	Do not assume one stable plan; semantics and plan selection interact	2, 6
[15]	Arch.	Separate rendering, API and deployment roles in the client architecture	4
[16]	Client	Account for client-side overhead independent of backend quality	4, 6
[17]	Client	Measure the client runtime; the environment affects perceived speed	6
[18]	Backend	Expect equal requirements to yield different response behavior per implementation	3, 6
[19]	Backend	State workload assumptions with every comparison; results do not transfer	6
[20]	Backend	Avoid universal technology rankings; measure in the target context	6

Because the study relies on secondary evidence, its empirical component is modest and explicit. REST API testing studies justify contract-first endpoint design, database access studies justify early schema and query planning, React refactoring research justifies deliberate state organization, and performance comparisons justify measurement in place of assumption. No effect size is pooled, since the sources differ in subjects, workloads and instrumentation to a degree that would make pooling misleading. The result is a structured synthesis whose individual steps remain traceable through Table 1.

3.3 Weighted Development Index

To quantify resource allocation across stages, the article proposes a Weighted Development Index, in which total effort is a function of estimated time and a relative complexity coefficient:

$$D_w = \sum_{i=1}^{n} (T_i \times S_i) \quad (1)$$

The index sums, across the six stages, the product of the estimated time in hours and the complexity coefficient, where i denotes the stage, T the time and S the coefficient. The coefficient is an ordinal judgment on a five-point scale expressing how much of a stage cannot be routinized: it rises with the number of cross-layer assumptions the stage must resolve and with the cost of revisiting its decisions later. Multiplying it by duration separates two quantities that schedule planning conflates, since a long stage of low difficulty and a short stage of high difficulty carry different risk. Table 2 reports the reference distribution, which corresponds to scenario S2 of the evaluation.

The six stages of Table 2 are the same six used in Table 3 and throughout the evaluation, so that effort, difficulty and evidence basis always refer to identical units of work. The values describe a medium-size project of five to seven domain entities built by a small team; they calibrate the index rather than recommend a schedule. Java backend construction at 30 hours and ReactJS client architecture at 26 take 47 percent of the 120-hour budget, while data-flow analysis at 14 and performance optimization at 12 receive 22 percent between them. The two stages that fix every later contract and that close the iterative loop are thus the cheapest, which is the argument for reordering the process rather than enlarging it.

Table 2. Reference distribution of effort and complexity across the six stages of the J-SQL-R Algorithm

Stage	Time (hours)	Complexity coefficient
1. Data-flow analysis	14	3
2. SQL schema design	22	4
3. Java backend	30	5
4. ReactJS client	26	5
5. Contract testing	16	4
6. Performance optimization	12	4

The main limitation is that the algorithm has not yet been validated on a live project. The article presents it as a research-grounded authorial model ready for empirical testing, and the evaluation below is analytical: it examines how the model behaves across project types and how sensitive its conclusions are to its own parameters, which is a weaker claim than measured improvement. A second limitation is the diversity of the corpus, since no single source supports the sequence as a whole. The methodology should be adapted to project scale, team maturity and domain constraints.

4. RESULTS

4.1 Cross-layer Determinants

The cross-layer synthesis identifies a set of critical architectural determinants. Robust full-stack systems emerge when the system is designed around data flow before individual implementation details are fixed. Most failures in Java, SQL and ReactJS systems do not arise because a technology is unsuitable; they arise because the relation between technologies is left implicit. A semantic gap opens between backend Data Transfer Object structures and the visual state requirements of the React client; a query returns correct results but performs poorly because the access path is redundant; a component grows around temporary state assumptions and later resists refactoring; an API is documented but never tested against stateful usage. These are integration failures, not merely coding failures.

The first result concerns the role of APIs. REST API testing research indicates that the API must be treated as a central contract from the start of development: endpoints should be described, tested and stressed with realistic data before the frontend depends on unstable assumptions [1, 3, 8, 9]. The mechanism explains why the ordering matters rather than merely asserting it. Each of these tools finds defects by exploring states a hand-written suite does not enumerate, and every such strategy requires a contract description that exists independently of the client. Where the contract is inferred from whatever the frontend happens to request, there is nothing stable to explore against, and the tools degrade into regression checks on current behavior. Kim et al. [10] reinforce the point by showing that automated generation still leaves substantial behavior unexercised.

The second result concerns database access. Empirical work on database access bugs in Java applications and on redundant ORM retrieval shows that SQL problems are frequently produced by application-level choices [4, 5]. Schema constraints, indexing strategies and transaction boundaries must be synchronized with Java persistence logic during initial design, so that data-access patterns are defined before controllers are implemented. The N+1 query problem is the standing risk, and its structure explains why it survives ordinary review: it is invisible in the application source, where a single collection traversal appears, and visible only in the query log under a data volume large enough for the per-row cost to matter. Chen et al. [4] show that redundant access degrades performance measurably, and Li et al. [14] add that plan selection itself varies, so a path that performs acceptably in development can degrade in production without any change to the code. The access path, not only the schema, must therefore be an explicit design artifact, which the algorithm requires at stage 2.

The third result concerns the client layer. ReactJS offers a powerful model for component-based interfaces, and that power turns into complexity when state ownership, API consumption and component boundaries stay undefined. Refactoring research supports a frontend architecture that mirrors stable user tasks rather than transient backend responses [6]. The asymmetry is the operative point: user tasks change on the timescale of product decisions, whereas response shapes change whenever a backend developer finds a more convenient serialization, so a component tree aligned to the latter inherits a faster rate of forced change. Deployment and runtime conditions shape modern applications as much as source structure [15, 16, 17]. This result rests on a thinner evidence base than the first two and is stated as the best-supported reading of that evidence rather than a settled finding.

4.2 The J-SQL-R Algorithm

On the basis of these results, the article proposes the J-SQL-R Algorithm as its authorial practical methodology, its name reflecting the three technological pillars of the target system. Table 3 gives the six stages, the action each requires, the evidence that motivates it and the outcome it is expected to produce. Two features of the ordering carry the argument: the data model and its access paths are settled in stage 2, before any controller exists, and the API contract is stable before the client is built in stage 4.

During the client architecture phase the methodology emphasizes atomic design principles and strict unidirectional data flow. Decoupling presentation components from global state management and side-effect logic raises testability, and

the resulting modularity lets the frontend absorb API contract changes with minimal regression risk [6].

Table 3. J-SQL-R Algorithm for building Java-SQL-ReactJS client-server applications

Stage	Core action	Evidence basis	Expected outcome
1. Data-flow analysis	Define users, entities, requests and response flows	REST API [2, 7, 10]	Clear functional and data limits
2. SQL schema design	Model tables, constraints, indexes and access paths	DB, ORM [4, 5, 11, 13, 14]	Reliable persistence, fewer defects
3. Java backend	Implement services, repositories and validation	Spring [5, 11, 12, 18]	Stable logic, controlled API
4. ReactJS client	Build components, routes, state and API consumers	React [6, 15, 16]	Usable interface, predictable state
5. Contract testing	Test APIs with realistic, invalid and stateful inputs	Contract [1, 3, 8, 9]	Reduced integration errors
6. Performance optimization	Measure SQL, backend and client performance together	Perf. [12, 17, 18, 19, 20]	Evidence-based improvement cycle

Stage 5 secures bi-directional integrity of the API layer by treating the OpenAPI specification as the source of truth, which permits automated generation of TypeScript interfaces for the React frontend and keeps contract verification consistent with the backend. Stage 6 measures database queries, backend response times and client rendering together rather than in isolation.

The methodology prioritizes contract-first design so that separation of concerns does not turn into architectural fragmentation. A React table designed without knowing pagination limits in SQL creates future friction; a Java service that returns large nested objects transfers cost to frontend rendering; a schema that ignores API access patterns behaves poorly under common requests. Every layer is therefore independent in implementation and interdependent in design.

4.3 Projected Validation Scenario

A compact empirical scenario clarifies how the methodology can be tested. A team could apply the algorithm to a medium-size task management application with user accounts, projects, tasks, comments and reporting screens, building a baseline version through a conventional screen-first approach and a second through the J-SQL-R sequence, then comparing endpoint defect counts, redundant SQL queries, mean API response time, frontend refactoring operations and the time needed to add a feature. Such a design is not difficult to implement, but it was not conducted for this article. Table 4 states the expected direction and magnitude of change.

Table 4. Theoretical projection of performance metrics before and after applying the J-SQL-R Algorithm. The values are reasoned predictions, not measurements

Metric	Before method	After method
Response time (ms)	320.0	210.0
Throughput (req/sec)	120.0	190.0
Error rate (%)	4.5	2.1
DB query time (ms)	180.0	110.0
Render time (ms)	210.0	140.0

These values are not measurements and must not be read as results. They are a theoretical projection: an estimate of the direction, and of the approximate magnitude, of the change the reordering would produce, reasoned from effect sizes the corpus already reports. No system was built, no system was measured and no participants were consulted. The reasoning behind each figure is given so that each can be disputed separately. The projected fall in database query time rests on the redundant-access measurements of Chen et al. [4], the largest single effect the corpus reports for a change of access strategy, and is the best supported. The movement in response time and throughput takes the direction, but not the size, of the differences comparative backend studies report between implementations of equivalent functionality [12, 18, 19, 20]; those studies compare technologies rather than development orders. The fall in error rate follows the defect-detection results of the contract-testing literature [1, 3, 8, 9], which do not quantify what share of those faults would reach production. The fall in render time is the weakest, since the corpus holds no controlled measurement of client rendering under a changed development order [6, 16, 17]; it is included because it is the value most likely to be refuted. Table 7 states when each would count as contradicted.

Figure 2 restates the five projected pairs as relative change against the baseline, which puts every metric on one comparable scale; the absolute levels remain in Table 4, where the error rate can be read without being flattened against an axis shared with values two orders of magnitude larger. Four of the five metrics fall while throughput rises, and the combination matters more than either movement alone: a change that only reduced latency by admitting less load would depress both quantities together, whereas simultaneous latency reduction and throughput growth is the signature of recovered capacity. The two largest timing gains belong to stages the algorithm moves earlier, which is the internal consistency check on the projection: had the largest gains fallen to stages whose position is unchanged, the projection would not have been attributable to the reordering at all (Fig. 2).

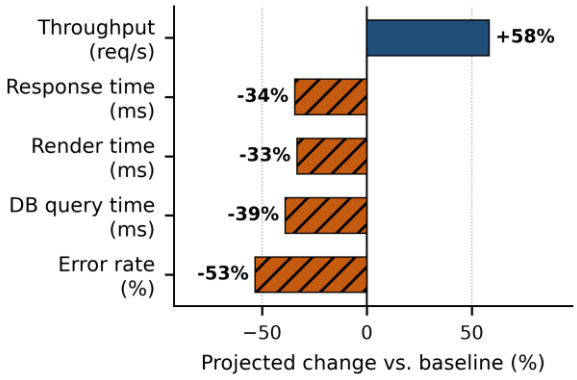

Fig 2: Projected change of each metric against the baseline, expressed as a percentage

5. EVALUATION

The algorithm cannot be evaluated by measurement here, since no system was built for it, but three questions can be answered analytically before any measurement is attempted. Does the sequence behave differently across project types? Are its conclusions stable under perturbation of its own parameters? And does it differ observably from the development orders teams already use?

5.1 Evaluation Design and Scenarios

Four scenarios span the dimensions along which client-server projects differ most: the number of domain entities, the read-to-write balance, the regulatory load and interface complexity. They are analytical profiles rather than samples, each chosen to stress a different stage. S1 is a small prototype of two or three entities with a short lifetime and no external consumers, where the danger is over-engineering the process itself. S2 is the medium task manager of Table 2, five to seven entities with several user roles and one external client, whose dominant risk is an unstable API contract reaching the client. S3 is a query-intensive reporting platform of twelve to fifteen entities with aggregation and large result sets, exposed to redundant and unindexed access as volume grows. S4 is a regulated enterprise system of ten to fourteen entities with an audit trail, fine-grained access control and strict validation, where verification effort and traceability dominate.

5.2 Scenario Analysis of Development Effort

The index was computed for each scenario by assigning stage durations and coefficients consistent with its profile, holding the six stages fixed. Table 5 reports the contributions and Figure 3 their composition. Totals rise from 134 for S1 to 1158 for S4, which is expected and uninformative on its own; what matters is that the composition changes, because a sequence prescribing the same emphasis everywhere would offer no guidance beyond a checklist.

Table 5. Stage contributions to the Weighted Development Index across the four scenarios, given as the product Ti x Si with the share of the scenario total in parentheses

Stage	S1 prototype	S2 task manager	S3 reporting	S4 regulated
1. Data-flow analysis	12 (9.0%)	42 (8.0%)	80 (8.5%)	160 (13.8%)
2. SQL schema design	16 (11.9%)	88 (16.9%)	210 (22.2%)	180 (15.5%)
3. Java backend	42 (31.3%)	150 (28.7%)	230 (24.4%)	320 (27.6%)
4. ReactJS client	42 (31.3%)	130 (24.9%)	190 (20.1%)	160 (13.8%)
5. Contract testing	12 (9.0%)	64 (12.3%)	104 (11.0%)	250 (21.6%)
6. Performance optimization	10 (7.5%)	48 (9.2%)	130 (13.8%)	88 (7.6%)
Total Dw (total hours)	134 (53)	522 (120)	944 (198)	1158 (244)

It changes in the direction the literature predicts. In S1 the two implementation stages take 62.6 percent of the index and verification 16.5 percent, which is the quantitative form of the observation that lightweight projects should not carry heavyweight process: the full sequence would spend a large share of a small budget on contracts no second consumer will read. In S3 the share of SQL schema design rises to 22.2 percent and overtakes the client layer, which matches the redundant-access evidence [4, 14] and marks stage 2 as the stage whose neglect is most expensive in read-dominated systems. In S4 contract testing rises to 21.6 percent while the client layer falls to 13.8: regulated systems move effort toward the stages that produce traceable artifacts. The sequence is therefore invariant while the investment profile is not, and a team that applies it without reweighting the stages to its own profile will misallocate effort while following the procedure correctly.

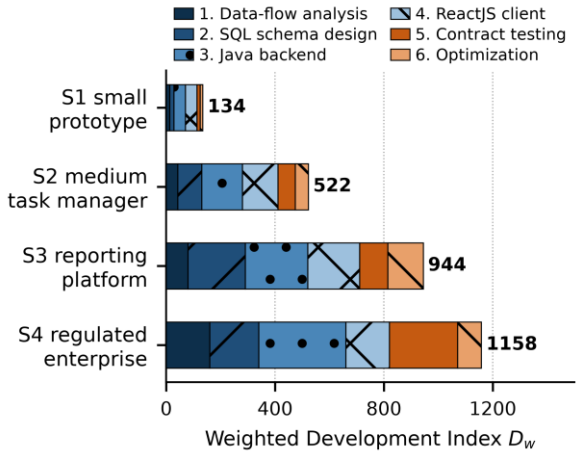

Fig 3: Composition of the Weighted Development Index across the four evaluation scenarios

5.3 Sensitivity of the Complexity Coefficients

The complexity coefficient is an ordinal judgment, so conclusions drawn from it are only as good as their stability under disagreement about its value. Figure 4 shifts each coefficient by one point in either direction, bounded by the ends of the scale, and recomputes that stage's contribution. The top of the ranking survives any single shift: Java backend construction varies between 120 and 150 and remains the largest contributor, ReactJS client architecture between 104 and 130 and remains second. The middle does not. SQL schema design ranges from 66 to 110, contract testing from 48 to 80, performance optimization from 36 to 60 and data-flow analysis from 28 to 56, so these intervals overlap one another and that of stage 2 reaches into that of stage 4. Any ordering among them is an artifact of the coefficients chosen. The index can therefore identify the dominant stage and compare investment profiles across project types, and cannot argue fine distinctions between adjacent stages.

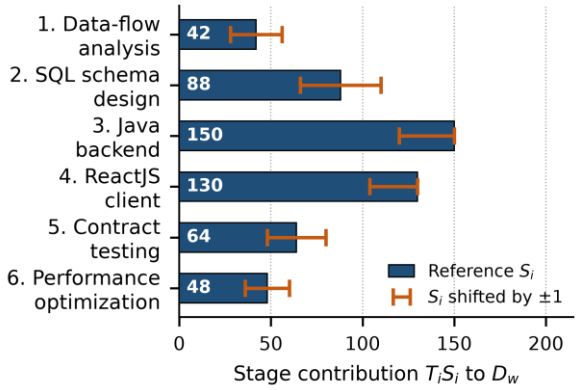

Fig 4: Sensitivity of each stage contribution to a one-point shift in its complexity coefficient

5.4 Comparison with Alternative Development Orders

Table 6 compares the sequence with three widely practiced alternatives: the screen-first order, in which interface requirements drive endpoints and endpoints drive the schema; the schema-first order, in which a normalized data model is settled before use cases are elaborated; and the API-first order, in which a specification is agreed before persistence or interface work begins. The criteria are chosen so that each could be operationalized in a later experiment.

Table 6. Analytical comparison of the J-SQL-R Algorithm with three alternative development orders

Development order	Where the contract is fixed, and what that costs
Screen-first	Implicitly, once screens exist. Exposure to redundant access is high, since queries follow screen needs [4, 5]; frontend rework on a contract change is high [6]; verification comes after implementation; no evidence is stated for the ordering.
Schema-first	In the schema, before the use cases are known. The schema is correct but access paths are unplanned; frontend rework is high; verification comes after implementation.
API-first	In the specification, before implementation. Endpoints are designed and persistence is secondary, so redundant access remains possible; frontend rework is low; verification follows the specification and precedes integration.
J-SQL-R	In stage 1 from the data flow, refined in stage 2 against access paths. Access paths are modeled with the schema, so exposure is low; the contract precedes stage 4 and is consumed through generated types, so rework is low; verification runs at stage 5 against the contract and at stage 6 across all layers; the evidence for each stage is stated in Table 1 and Table 3.

The comparison shows where the proposal is distinct and where it is not. Against API-first it offers no advantage in frontend stability, since both fix the contract before client work begins; the difference lies in stage 2, because API-first leaves persistence to be arranged around a specification written without reference to access paths. Against schema-first the difference is the reverse: both take persistence seriously, but schema-first fixes the model before the data flow that will use it is known. The J-SQL-R sequence is therefore best understood not as a new practice but as the composition of two existing ones, ordered so that data flow constrains the schema and the schema constrains the contract.

5.5 Falsification Criteria and Validation Protocol

A methodology that cannot fail is not a research contribution. Table 7 states the design of the validation study, the quantities to be measured and the outcomes under which specific claims would be rejected. Each criterion is attached to a stage, so that a failure identifies which part of the sequence is unsupported rather than condemning the whole. The protocol is deliberately modest in scale, since a study that cannot be run is no better than an untested claim.

Table 7. Validation protocol and falsification criteria for the J-SQL-R Algorithm

Element	Specification
Design	Two-arm comparison on identical requirements, development order the only manipulated variable (screen-first baseline against the J-SQL-R sequence), arm order counterbalanced across teams; the four profiles of Section 5.1 as scenario types
Measures	Endpoint defects found in contract testing; redundant SQL queries; mean and 95th-percentile API response time; frontend refactoring operations; hours to add one specified feature
Instrumentation	Stateful and black-box endpoint testing after RESTler and RESTest [1, 3]; query logging with execution-plan inspection [4, 14]; refactoring detection as in Ferreira et al. [6]; load generation at a fixed request profile
Falsification	(i) stage 2 rejected if S3 shows no reduction in redundant queries; (ii) stage 4 rejected if S2 shows no reduction in frontend refactoring; (iii) the index rejected if the rank correlation between the predicted contributions of Table 5 and observed effort is not positive; (iv) the algorithm confirmed unsuitable below its stated scale if S1 shows no benefit while consuming more effort
Threats	Learning effects between arms; team maturity as a confound; one application domain limiting generalization; the projections of Table 4 anchoring the measuring team's expectations

5.6 Threats to the Validity of the Evaluation

Three limitations should be stated plainly. The scenario durations and coefficients are authorial estimates, so the evaluation tests the internal behavior of the model and not its correspondence to any project, although the sensitivity analysis shows the dominant stages would survive a different analyst's figures. The four scenarios are not a sample, and profiles combining high regulatory load with a heavy read workload are not represented. Table 6 compares the alternatives as described in practice rather than as any team implements them, so it establishes conceptual difference rather than measured advantage. These limitations are removed by running the protocol of Table 7, not by argument.

6. DISCUSSION

The results suggest that a client-server application is better understood as a chain of decisions than as a stack of technologies. Java, SQL and ReactJS each solve a distinct class of problems, yet they produce value only where their assumptions meet, which is why the algorithm opens with data flow and not with code generation. API testing literature supports that ordering: when an API is poorly conceptualized, later tools can reveal problems but cannot compensate for weak contract thinking [1, 3, 8].

SQL and the client layer each deserve a more central place in full-stack methodology than they usually receive. In many projects the database is treated as storage behind repositories or ORM abstractions, a view that is convenient and risky: schema design, constraints, indexing and query planning are not low-level details but shape correctness and speed [4, 5]. React applications become difficult to maintain for the symmetrical reason, when components absorb business rules that belong in API contracts or when local state mirrors server state without a synchronization strategy [6]. Both observations support the same ordering requirement: frontend architecture should be designed after data flows and contracts are known [15]. Optimization must likewise measure the full request path, since serialization is the one point where an optimized query and an optimized component can still be separated by an oversized payload.

Comparative performance literature prevents technological absolutism. Studies of Spring, Express.js, .NET, Node.js and Java Spring Boot show that performance outcomes depend on implementation context, workload and surrounding architecture [18, 19, 20]. The algorithm therefore makes no claim that Java is always the best backend or that ReactJS is always the best frontend. It claims something narrower: once these technologies are selected, they should be joined through a disciplined sequence that makes data, contracts, tests and performance visible.

The authorial contribution is methodological. The J-SQL-R Algorithm introduces no new language, framework or tool. Its novelty lies in ordering known engineering concerns into a repeatable procedure and in stating, for each step, the evidence that motivates it and the observation that would refute it. In software practice order carries cost: database access considered only after API implementation makes performance problems expensive, React state designed before contracts stabilize multiplies refactoring, and testing postponed to the end becomes defect discovery rather than design support. Table 6 locates the distinctness precisely: against API-first the algorithm adds access-path design, against schema-first it adds the data-flow constraint that decides which schema is the right one.

Limitations remain. The article rests on secondary synthesis, so it cannot claim direct causal proof that the algorithm improves project outcomes. The corpus is relevant but not exhaustive, and several included studies address REST APIs or database systems broadly rather than Java, SQL and ReactJS applications specifically. The methodology may also prove too detailed for small prototypes and insufficiently detailed for heavily regulated enterprise systems.

The evaluation sharpens two of these limitations into testable statements rather than dissolving them. The scenario analysis shows that the sequence redistributes effort differently across project types, so the algorithm has content beyond a checklist, and that for the smallest scenario the redistribution may not repay its overhead, which the fourth falsification criterion of Table 7 is written to detect. Both results are properties of the model, not of any system built with it.

The practical implications are direct. Record what data moves, who owns it and how it changes; design SQL schemas against real access paths rather than conceptual entities alone; enforce validation and business rules through stable contracts; organize components around user workflows and intentionally shaped responses; cover contract, stateful and negative cases in testing; and measure the full request path rather than any single layer.

7. CONCLUSIONS

This article developed a structured algorithm for building client-server applications using Java, SQL and ReactJS. The study conducted no original experiment. It synthesized twenty English-language sources spanning REST API testing, Java database access bugs, ORM performance, DBMS testing, React refactoring and comparative backend performance, and reported the coding of every source against layer, requirement and stage so that the derivation can be audited. The central finding is that client-server quality depends on the coordination of layers rather than on the isolated strength of individual technologies.

The methodology is practical because it follows the real order in which client-server risks appear. Data misunderstanding comes first; database defects and inefficient access follow; API instability then propagates into the frontend; performance issues surface last, once all parts are connected.

The contribution is both scientific and applied. Scientifically, the article connects separate research streams into one conceptual model, states the mapping from evidence to procedure in an inspectable form, and evaluates that model analytically across four project scenarios with a sensitivity analysis of its own parameters. Practically, it supplies a usable procedure together with an indication of how its effort profile shifts between small prototypes, read-intensive reporting systems and regulated enterprise applications. The principal limitation is the absence of primary validation. Future research should run the protocol of Table 7 on real projects, using API defects, database access bugs, redundant queries, frontend refactoring frequency, response time and developer effort as dependent variables. That work would convert the methodology into an empirically evaluated framework, or identify, through the stated criteria, which of its stages do not earn their place.

8. REFERENCES

[1] Atlidakis, V., Godefroid, P., and Polishchuk, M. 2019. RESTler: Stateful REST API fuzzing. In Proceedings of the 41st International Conference on Software Engineering (ICSE 2019), 748-758. https://doi.org/10.1109/ICSE.2019.00083

[2] Golmohammadi, A., Zhang, M., and Arcuri, A. 2023. Testing RESTful APIs: A survey. ACM Transactions on Software Engineering and Methodology 33, 1, Article 27, 1-41. https://doi.org/10.1145/3617175

[3] Martin-Lopez, A., Segura, S., and Ruiz-Cortés, A. 2021. RESTest: Automated black-box testing of RESTful web APIs. In Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021), 682-685. https://doi.org/10.1145/3460319.3469082

[4] Chen, T.-H., Shang, W., Jiang, Z. M., Hassan, A. E., Nasser, M., and Flora, P. 2016. Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks. IEEE Transactions on Software Engineering 42, 12, 1148-1161. https://doi.org/10.1109/TSE.2016.2553039

[5] Liu, W., Mondal, S., and Chen, T.-H. 2024. An empirical study on the characteristics of database access bugs in Java applications. ACM Transactions on Software Engineering and Methodology 33, 7, Article 181, 1-25. https://doi.org/10.1145/3672449

[6] Ferreira, F., Borges, H. S., and Valente, M. T. 2024. Refactoring React-based Web apps. Journal of Systems and Software 215, 112105. https://doi.org/10.1016/j.jss.2024.112105

[7] Patil, O., and Shirbhate, A. 2024. Design and implementation of a Java-based client-server application. arXiv preprint arXiv:2404.10107. https://doi.org/10.48550/arXiv.2404.10107

[8] Karlsson, S., Čaušević, A., and Sundmark, D. 2020. QuickREST: Property-based test generation of OpenAPI-described RESTful APIs. In 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST), 131-141. https://doi.org/10.1109/ICST46399.2020.00023

[9] Alonso, J. C., Martin-Lopez, A., Segura, S., García, J. M., and Ruiz-Cortés, A. 2023. ARTE: Automated generation of realistic test inputs for web APIs. IEEE Transactions on Software Engineering 49, 1, 348-363. https://doi.org/10.1109/TSE.2022.3150618

[10] Kim, M., Xin, Q., Sinha, S., and Orso, A. 2022. Automated test generation for REST APIs: No time to rest yet. arXiv preprint arXiv:2204.08348. https://doi.org/10.48550/arXiv.2204.08348

[11] Wieleba, E., and Wieleba, B. 2023. Performance analysis of working with databases with Spring and Symfony. Journal of Computer Sciences Institute 26, 75-82. https://doi.org/10.35784/jcsi.3086

[12] Grzeszuk, M., and Miłosz, M. 2025. Performance comparison of CRUD operations in Spring Boot and ASP.NET Core frameworks. Journal of Computer Sciences Institute 35, 175-183. https://doi.org/10.35784/jcsi.7198

[13] Zhong, R., Chen, Y., Hu, H., Zhang, H., Lee, W., and Wu, D. 2020. SQUIRREL: Testing database management systems with language validity and coverage feedback. arXiv preprint arXiv:2006.02398. https://doi.org/10.48550/arXiv.2006.02398

[14] Li, J., Wang, K., Chen, Y., Zhou, Y., Wu, L., and Wang, J. 2023. Detecting DBMS bugs with context-sensitive instantiation and multi-plan execution. arXiv preprint arXiv:2312.04941. https://doi.org/10.48550/arXiv.2312.04941

[15] Markovic, D., Scekic, M., Bucaioni, A., and Cicchetti, A. 2022. Could Jamstack be the future of web applications architecture? An empirical study. In Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing (SAC 2022), 1872-1881. https://doi.org/10.1145/3477314.3506991

[16] Biørn-Hansen, A., Rieger, C., Grønli, T.-M., Majchrzak, T. A., and Ghinea, G. 2020. An empirical investigation of performance overhead in cross-platform mobile development frameworks. Empirical Software Engineering 25, 4, 2997-3040. https://doi.org/10.1007/s10664-020-09827-6

[17] Kurant, Ł., and Bylina, J. 2026. Impact of selected JavaScript engines on the performance of mobile hybrid applications. Journal of Software: Evolution and Process 38, 2. https://doi.org/10.1002/smr.70086

[18] Wicha, M., and Pańczyk, B. 2023. Performance analysis of REST API technologies using Spring and Express.js examples. Journal of Computer Sciences Institute 29, 352-359. https://doi.org/10.35784/jcsi.3796

[19] Godinho, A., Rosado, J., Sá, F., and Cardoso, F. 2025. An evaluative study comparing the performance of RESTful web APIs created with .NET and Java Spring Boot, 235-264. https://doi.org/10.1007/978-3-031-79086-7_20.

[20] Shyam Mohan, J. S., and Goswami, K. 2025. Performance analysis and comparison of Node.js and Java Spring Boot in implementation of RESTful applications. Software: Practice and Experience 55, 7, 1209-1233. https://doi.org/10.1002/spe.3418