

A Schema-Driven Form Engine Architecture for Data Capture in Field Service Environments

Khunaal Aryan

Department of Information Science and Engineering RV College of Engineering, Bengaluru, India

khunaalaryan.is22@rvce.edu.in

Dr. Padhmasree T

Department of Information Science and Engineering RV College of Engineering, Bengaluru, India padmashreet@rvce.edu.in

Abstract

Enterprise field service demands disciplined and trusted data collection across mobile, tablet, and desktop clients, often in locations with weak or no network connectivity. Most of the current systems depend on third-party SaaS tools that result in vendor dependency and limited customization. Others use platform specific native forms, which fragments the codebase and creates validation drift. This paper presents schema driven form engine architecture using a single JSON schema model, seven types of structured blocks, a platform-independent rendering layer and configurable validation rules. The architecture separates the definition of the form from the rendering, allowing the same schema to render consistently on Android, iOS and Web clients. An offline operation model adds local schema caching, form-state persistence and background sync when connectivity comes back. The architecturing was introduced in a live enterprise deployment at a multinational healthcare equipment manufacturer, replacing a discontinued SaaS capture tool with a full in house system.

Keywords: *schema driven forms, form engine, cross-platform rendering, offline-first, field service management, enterprise data capture, JSON schema, block taxonomy*

I. INTRODUCTION

Field service engineering in regulated industries, and especially healthcare equipment maintenance, poses a challenging data capture problem. Field Service Engineers document structured observations while performing service calls. These include equipment readings (temperature, vibration, and pressure), compliance checklist results, inspection findings, and formal service reports. These records have legal and quality significance, such as under standards like ISO 13485, which governs quality management systems for medical device manufacturers. This need has been often fulfilled by organizations with third party SaaS data capture platforms. Those tools are usually feature rich and fast to deploy. The problem is that deploying for enterprise use exposes hard limits: limited ability to customize form logic, dependence on vendor pricing and availability, limited integration with systems like Salesforce ServiceMax, and poor offline support in locations with unreliable connectivity.

This operational gap was created for a multinational healthcare equipment manufacturer when TrueContext (formerly ProntoForms) was removed from the ServiceMax package. TrueContext has been a popular solution for capturing field service data. The commercial offerings didn't offer the right combination of structured

multi-block capture, offline-first operation, consistent Android, iOS and Web rendering and deep Salesforce integration. As a result, the organization developed an in-house field service management application from scratch. "The main engineering challenge was building a form engine that fulfilled five criteria simultaneously. A single schema representation was needed that separated the form definition from the platform rendering. It had to work consistently across Android, iOS and Web from a single form definition. It required validation for numeric ranges, character restrictions, and required fields without duplicated logic across platforms. It needed offline access, completion, and persistence for FSE work. It also needed schema versioning so templates evolve while existing submissions remain interpretable.

The paper makes four contributions: First, it proposes a novel generalized block taxonomy and JSON schema model for enterprise field service form engines based on compliance-driven data capture. Secondly, it provides a platform independent rendering architecture, decoupling schema definition from UI realization. Third, it describes a configurable validation framework enforcing rules uniformly at the engine layer instead of inside each client. Fourth, it provides an offline operation model for compliance-critical environments with deterministic synchronization behavior. A live enterprise deployment is instantiated with the full framework and validated.

II. RELATED WORK

A. Dynamic Form Generation

Form generation dynamically refers to the construction of a user interface at runtime based on the meta-definition or schemas and not by using static code. Most studies have been done in relation to the development of forms in a web environment for clinical data collection and e-governance. The early systems highlighted the importance of separating the design and the presentation of the forms through the use of XML schemas. The majority of the studies conducted in the past still revolve around single platform-based forms within the web.

B. Offline-First Architecture

The theoretical foundation of the offline-first approach is laid down by Kleppmann et al. [10]. This is done by considering local-first software as their basis. They advocate local ownership of the data and operation independent of

network connections as key design aspects rather than fallback approaches. It is from this that the concept of the synchronization approach presented here can be derived. There are further restrictions due to the field service context. The data input forms are vital from a compliance perspective. Conflicts must not depend on user decisions, and the schema is to be accessible offline.

C. Cross-Platform Frontend Architecture

Piastou [3] compares React, Angular and Vue.js considering frontend performance and scalability, concluding that React performs better in complex and data-intensive UI environments. This is relevant to the Admin Portal, where Designers work with multi-page form builders and dynamic block configurations. Zhang et al. [5] review fault-tolerance in microservices and adaptive load balancing to achieve highly available web systems, thus support the proposed reliability architecture for the backend components. None of the studies analyze consistent rendering of UI on native mobile and Web platforms from one schema definition..

D. Microservices and API Gateway Patterns

Velepucha and Flores [1] review principles of microservices, decomposition patterns, and challenges of migration at scale. Their work suggests decomposing the backend into a Forms Service for schema lifecycle, a Submission Service for FSE submission events, and a Sync Forms Service for offline schema distribution. Zhao et al. [2] describe API gateway patterns in microservice systems, with APIGEE being used as the single entry point for clients to communicate with the form engine backend..

E. Message Queue Architecture

Maharjan et al. [4] benchmark message queue technologies in terms of throughput, latency and reliability. Their findings indicate that RabbitMQ exhibits the lowest latency at 4.1 ms in task-queue patterns, whereas Kafka performs better in high-throughput event streaming scenarios. This supports RabbitMQ as the async messaging layer between the Submission Service and downstream reporting or document generation. This decoupling is important for the form engine, as FSE submissions are confirmed without waiting for the document generation to finish..

F. RBAC and Access Control in Enterprise Systems

Ferraiolo et al [7] gives the theoretical and implementation basis for Role Based Access Control for enterprise applications. The proposed system uses three roles, Admin, Designer and FSE. This is a straightforward use of the RBAC pattern. The multi-tenancy model is provided by workspace-scoped boundaries, so Designers remain within the scope of the Workspaces they are assigned to, and FSEs receive forms through the User Groups they are assigned to.

G. CI/CD and DevOps Practices

Zampetti et al. [8] and Decan et al. [9] study the evolution of CI/CD pipeline and repository-based workflow automation. Their results inform the deployment architecture presented in Section V. The current state uses Jenkins-based

CI/CD, while the target state migrates to Azure DevOps pipelines, following the pipeline evolution patterns reported in the literature.

H. Gap Statement

The literature review leaves open the intersection of schema-driven form engine design, cross-platform rendering consistency, offline-first operation and compliance-driven validation for enterprise field service applications. Schema driven UIs, offline first systems, RBAC and Microservices are all well studied in isolation. In this paper we propose a cohesive architecture that combines these elements under the constraints of compliance-sensitive field service data capture.

III. PROBLEM FORMULATION

An enterprise field service form engine needs to meet requirements that are individually handled in previous work, but are not addressed in the aggregate by existing architecture patterns. In this section we present the requirements and the design constraints that motivate the framework.

A. Functional Requirements

R1 - Unified Schema Representation: form structure, block composition, validation rules and display properties all have to be represented in a single platform-neutral schema representation that is stored, versioned and distributed independently of any client implementation.

R2 - Cross-Platform Rendering Consistency: A single schema instance defining a form must render consistently in terms of structure, behavior, and validation response across native Android, native iOS and Web browser clients. Risk of non-conformance due to cross platform drift in rendering or validation.

R3 - Structured Block Taxonomy The schema shall support a defined input block taxonomy for healthcare equipment field service data capture including free text, extended narrative, numeric parameters with range constraints, single selection controls, multi selection controls and informational display blocks.

R4 - Engine Layer Validation Rule Enforcement: The form engine should enforce validation constraints, such as mandatory fields, numeric ranges, character type restrictions, and decimal precision limits from the schema definition, rather than re-implementing them for each client platform.

R5 - Offline Operability: FSEs should be able to access form templates, fill submissions and persist completed data without a network connection. Deterministic and lossless synchronization of locally persisted data to the central system shall be triggered by connectivity restoration.

R6 - Schema Versioning: form templates should support versioning so that schema updates preserve in-progress submissions. Each submitted form must be permanently linked to the form version active at the time of submission.

B. Design Constraints

The architecture operates under three key design constraints. First, the form engine must connect with the

existing microservices backend and API gateway infrastructure without altering the gateway contract. Second, offline schema distribution must align with device storage limits on Android and iOS clients. Third, the system must follow ISO 13485 quality management standards, which require complete auditability of form submissions. This includes submitter identity, active schema version, and the raw submitted values.

IV. THE PROPOSED FRAMEWORK

The framework has six integrated components: JSON schema architecture, block type taxonomy, validation rule framework, multi-page form structure model, platform-agnostic rendering abstraction, and offline operation model. I will explain each component below.

A. JSON Schema Architecture

The schema serves as the foundation of the framework. A form template is represented as a structured JSON document with three top-level sections. The metadata block includes a form identifier, version identifier, Workspace assignment, and creation timestamp. The pages array holds an ordered sequence of page definitions. The validation manifest comprises validation rule configurations applicable to the blocks inside the form. Each page definition contains an ordered blocks array. Each block definition includes a UUID block identifier, a block type discriminator, a display label, a required flag, a reference to validation rules in the validation manifest, and type-specific configuration.

For instance, a Dropdown block includes an options array, while a Numeric block includes precision and range configuration. Schema versioning uses a monotonically increasing integer version field at the form template level. When the schema updates, the previous version is archived rather than overwritten. The Submission Service links each submission record with the schema version identifier active at the time of submission. This meets R6 and allows historical submissions to render against their original schema.

TABLE I

JSON Schema Top-Level Structure

Field	Type	Description
form_id	UUID	Globally unique form template identifier
version	Integer	Monotonically incrementing schema version
workspace_id	UUID	Owning workspace assignment
created_at	ISO 8601	Schema creation timestamp
pages	Array	Ordered sequence of page definitions
validation_manifest	Object	Validation rule configurations keyed by block_id

A. Block Type Taxonomy

The taxonomy defines seven block types, each associated with a data capture mode found in healthcare equipment field service. The taxonomy emerged from analyzing existing paper-based and SaaS-based service record formats, followed by refinement through the Designer form-building process.

TABLE II

Block Type Taxonomy

Block Type	Input Mode	Validation Support
Short Text	Single-line alphanumeric	Character type, length, required
Long Text	Multi-line narrative	Length, required
Numeric Input	Decimal or integer value	Range (min/max), precision, required
Dropdown	Single selection from list	Required
Radio Button	Mutually exclusive choice	Required
Checkbox	Boolean or multi-select	Required
Read-Only / Info	Display only, input absent	N/A

The Numeric Input block deserves special attention because it is crucial for healthcare equipment servicing. Field service engineers (FSEs) record equipment readings such as operating temperature, vibration amplitude, and pressure level. These readings must remain within safe operating ranges. The Numeric Input block accepts rangeMin, rangeMax, and decimalPrecision values in the schema. The form engine enforces these constraints at input time and provides immediate feedback without requiring a network connection.

This satisfies R4 for compliance-critical data capture cases. The Read-Only / Information block serves a compliance function. It allows form designers to embed regulatory reference text, such as ISO 13485 Section 7.5, directly in the form context. The FSE sees the relevant reference while recording the associated parameter. This block adds nothing to the submission payload, but it renders consistently across all platforms.

B. Validation Rule Framework

Validation rules exist in the validation manifest and are linked by block_id from block definitions. Keeping validation configuration separate from block structure allows multiple blocks to share one rule definition. It also enables changes to validation rules without altering the display configuration.

The framework supports six types of validation rules. required enforces that a value must not be empty before the form can advance. numericRange enforces minimum and maximum bounds on Numeric Input blocks. decimalPrecision limits the allowed decimal places. numbersOnly restricts Short Text blocks to digits.

alphanumericOnly allows input of letters and digits. noSpecialCharacters permits letters, digits, and spaces while excluding punctuation and symbols. Validation runs in two stages. The form engine applies rules at input time within the client application, providing real-time feedback to the FSE.

During submission, the Submission Service revalidates field values against the validation manifest before saving. This prevents bypassing through data submitted outside the standard form interface.

C. Multi-Page Form Structure Model

Forms are structured as ordered sequences of pages, with each page containing an ordered sequence of blocks. The page model has three main purposes. It divides large forms into manageable units for the FSE. It controls advancement until the required fields on the current page pass validation.

It also provides offline form sessions with natural resumption points after a device restart or loss of connectivity. Page navigation state includes the current page index and completion status for previous pages. Offline devices save this state to local storage alongside partial submission data, allowing full resumption without data loss.

D. Platform-Agnostic Rendering Abstraction

The rendering abstraction enables one schema definition to produce a consistent user interface across Android, iOS, and Web clients. Each client platform has a form engine layer that implements a shared block rendering contract. For each block type, the rendering contract specifies input control type, how validation feedback is presented, mobile keyboard type, and how Read-Only blocks are displayed when disabled.

Platform implementations satisfy the contract with native UI components: Android Views or Jetpack Compose on Android, UIKit or SwiftUI on iOS, and React.js components on the Web. The engine layer handles schema parsing, block iteration, validation rule resolution, and page navigation uniformly.

This design keeps form logic separate from visual rendering. Form logic, including block order and validation rules, is defined once in the schema and executed once in the engine layer. Rendering is handled by native client implementations that fulfill the contract. Therefore, changes to validation rules, block order, or page structure only require a schema update, without altering client coded.

E. Offline Operation Model

The offline operation model addresses R5 through three stages: schema pre-distribution to client devices, local session persistence during completion, and background synchronization after connectivity is restored. Schema pre-distribution is managed by the Sync Forms Service. This backend microservice identifies applicable schemas for each FSE based on User Group and Workspace assignment, then sends current schema versions to the device cache when connectivity is available.

The local cache keeps a schema version index. After connectivity returns, the Sync Forms Service compares the device-side and server-side indices and distributes only changed schemas to minimize transfer size. During form completion, the form engine saves the partial submission state to local storage at each page boundary and after every validated field change.

This write-through approach preserves completed input during application termination or device power loss. On session resumption, the engine reads the saved state and returns the FSE to the exact form position. After connectivity is restored, the Submission Service client checks local storage for completed submissions waiting to sync, submits them in chronological order, and removes each local copy after server confirmation.

Failed synchronization attempts retry using exponential backoff. RabbitMQ is positioned between the Submission Service and downstream processing pipelines, so submission acknowledgment is separate from document generation and reporting completion. FSE confirmation remains quick even when downstream pipelines are busy.

V. IMPLEMENTATION AND INSTANTIATION

The framework has been implemented as the core of an in-house enterprise field service management application, replacing a discontinued third-party SaaS data capture tool for a multinational healthcare equipment manufacturer. The implementation uses a Spring Boot microservices backend, a React.js Admin Portal, native Android and iOS clients for FSEs, and APIGEE as the unified API entry point.

A. Backend Service Decomposition

The backend is divided into nine Spring Boot microservices, each with a distinct functional area. The Forms Service manages schema creation by Designers, increments version numbers on updates, publishes to FSE-accessible catalogs, and archives outdated versions. The Submission Service receives completed FSE payloads, revalidates them against the associated schema version, saves data to PostgreSQL MIAAS DB, and queues RabbitMQ messages for asynchronous downstream processing.

The Sync Forms Service implements offline schema pre-distribution through device-specific schema version indices and delta updates. The Enroll User Service handles the three-role RBAC model: Administrator accounts with full platform access, Designer accounts limited to their assigned Workspaces, and FSE accounts assigned to User Groups linked to Workspaces. The Document Generation Service receives structured XML payloads from the Submission Service and calls Open Text to create formally formatted PDF service records, which are the legally significant output of each field service visit.

B. Infrastructure and Deployment

All microservices are containerized and deployed on Amazon Elastic Kubernetes Service, with an AWS Application Load Balancer distributing traffic across service instances. The React.js Admin Portal is compiled into static

assets and hosted on AWS S3 through a VPC Endpoint, restricting access to the private VPC network and avoiding public internet exposure for the admin interface. APIGEE enforces authentication and authorization for API calls using OAuth 2.0 identity verification before routing requests to backend services.

CI/CD pipeline management adheres to patterns outlined by Zampetti et al. and Decan et al. Jenkins-based pipelines are utilized in the current deployment state and are being transitioned to Azure DevOps in the target architecture. Kibana provides real-time log visualization and system monitoring across service instances.

C. Designer Interface

The Designer interface is part of the Admin Portal used to create and configure form schemas. It displays the block taxonomy as a palette of components. Designers build forms by adding blocks to pages, setting display labels and required status, and selecting validation rules from a predefined rule set. The interface creates the JSON schema artifact from the Designer configuration and submits it to the Forms Service during publication. Schema generation occurs on the client-side in React.js before submission.

VI. DISCUSSION

A. Generalizability of the Framework

The framework was used in healthcare field service, but its main structure can apply to any field. The JSON schema model, block taxonomy, rendering abstraction, and offline operation model fit sectors that need organized, multi-platform, offline-capable data collection. Utilities field inspections, telecommunications maintenance, manufacturing quality checks, and building services management all share similar structural needs.

The block taxonomy may need to expand for fields with specific input needs. For example, a geolocation block for GPS-tagged observations or a media capture block for photos. The taxonomy detailed here focuses on healthcare equipment field service requirements and is kept narrow to avoid over-complicating the implementation context.

B. Build vs. Buy Considerations

The decision to develop an in-house form engine reflects enterprise needs shaped by, rather than solved by, the architecture. The main reasons for this choice were strong integration with Salesforce/ServiceMax, full control over how validation works, and the long-term cost of per-seat SaaS licensing for a large team of field engineers. The framework supports an informed architectural choice: an in-house schema-driven engine makes sense when there are deep customization needs, specific integration requirements, and licensing costs are a major concern due to operational size.

C. Limitations

This work has three limitations. First, the evaluation is based on a single enterprise deployment. Quantitative comparisons of rendering speed, synchronization throughput

under different connectivity, and how fast the schema parses based on form complexity were not included. Future assessments should add defined benchmarks for these areas. Second, the schema model currently does not allow conditional block visibility, meaning it cannot show or hide content based on earlier values. This would enhance the form definition language. Third, offline synchronization now relies on the chronological order of submissions, and better merge strategies for handling partial submission conflicts remain unexplored.

VII. CONCLUSION AND FUTURE WORK

This paper introduced a schema-driven form engine architecture for capturing enterprise field service data. The architecture aims to ensure consistent rendering across platforms, work offline, use a structured block taxonomy, and enforce compliance-grade validation. Its key contribution is separating the form definition, shown as a versioned JSON schema, from form rendering, which is assigned to native platform implementations under a shared agreement.

A single authoring action thus creates consistent and validated form experiences across Android, iOS, and Web clients. The framework was tested and validated in a real enterprise deployment at a multinational healthcare equipment manufacturer. It replaced a discontinued third-party SaaS data capture tool with a complete in-house system. The deployment demonstrated the framework operating under ISO 13485-regulated field service conditions.

Three areas for future work stem from this implementation. First, conditional block visibility would allow previous field values to control the display of later blocks, enabling forms that adapt to only show relevant inputs. Second, formal testing of rendering across platforms and syncing under realistic connectivity would improve the empirical foundation of the framework. Third, AI-assisted form generation using large language models to suggest block composition and validation settings from natural language service procedures is a natural extension that fits with enterprise AI integration.

REFERENCES

- [1] P. Velepucha and P. Flores, "A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges," IEEE Access, vol. 11, 2023. doi: 10.1109/ACCESS.2023.3305687
- [2] J. T. Zhao et al., "Management of API Gateway Based on Microservice Architecture," IOP Conf. Series: Journal of Physics, vol. 1087, 2018. doi: 10.1088/1742-6596/1087/3/032032
- [3] M. Piastou, "Comprehensive Performance and Scalability Assessment of Front-End Frameworks: React, Angular, and Vue.js," World Journal of Advanced Engineering Technology and Sciences, vol. 9, issue 2, 2023. doi: 10.30574/wjaets.2023.9.2.0153
- [4] R. Maharjan et al., "Benchmarking Message Queues," Telecom, MDPI, vol. 4, 2023. doi: 10.3390/telecom4020023
- [5] P. Zhang et al., "Adaptive Load Balancing and Fault-Tolerant Microservices Architecture for High-Availability Web

Systems Using Docker and Spring Cloud," Applied Sciences, Springer, 2025. doi: 10.1007/s42452-025-07320-7

- [6] M. Ndiaye et al., "Performance Analysis of Scaling NoSQL vs SQL: A Comparative Study of MongoDB, Cassandra, and PostgreSQL," IEEE SIST 2023, pp. 479-483. doi: 10.1109/SIST58284.2023.10223568
- [7] D. F. Ferraiolo et al., "Role Based Access Control in Enterprise Application, Security Administration and User Management," IEEE Conference on e-Commerce Technology, 2006. doi: 10.1109/CEC.2006.4018474
- [8] F. Zampetti et al., "CI/CD Pipelines Evolution and Restructuring: A Qualitative and Quantitative Study," IEEE ICSME 2021, pp. 471-482. doi: 10.1109/ICSME52107.2021.00048
- [9] A. Decan et al., "On the Use of GitHub Actions in Software Development Repositories," IEEE ICSME 2022, pp. 235-245. doi: 10.1109/ICSME55016.2022.00029
- [10] M. Kleppmann et al., "Local-First Software: You Own Your Data, in Spite of the Cloud," ACM SIGPLAN Onward! 2019. doi: 10.1145/3359591.3359737