Dynamic UI Generation Through API-Driven Solver Definitions: A Scalable and Sustainable Approach for Integrating Heterogeneous Solver Ecosystems in FOR-FREIGHT

Sozos Karageorgiou

25.11.2025

eBOS Technologies

1. Executive Summary

The FOR-FREIGHT platform integrates a diverse ecosystem of more than sixty solver services developed by multiple partners across Europe. These solvers vary significantly in purpose, complexity, data requirements, and API structures. Traditional UI development approaches—where each solver requires dedicated forms, validation logic, and rendering components—would have resulted in substantial integration overhead, inconsistency, and long-term maintenance costs.

To address this challenge, FOR-FREIGHT adopted a dynamic, metadata-driven architecture that enables the platform's user interface to generate solver-specific screens automatically at runtime. Instead of manually coding individual interfaces, the UI retrieves each solver's metadata definition—its endpoint, parameters, datatypes, validation rules, and output schema—and uses this information to construct input forms, execute solver calls, and render results.

This approach delivers a scalable, resilient, and future-ready architecture that dramatically simplifies onboarding new solvers, ensures consistent user experience, and reduces technical debt. It also demonstrates how metadata-driven design can facilitate interoperability across complex, multi-stakeholder logistics environments.

2. Introduction

The FOR-FREIGHT platform was designed as a modular, multi-service environment supporting a broad spectrum of optimisation, simulation, prediction, and analysis capabilities across multimodal freight operations. As the technical work matured, it became evident that the platform needed to integrate an exceptionally large and diverse problem-solving micro service (called solver hereon) ecosystem. More than sixty independent solver services, contributed by various partners across the consortium, had to be made accessible through a single unified user interface. These solvers varied drastically in scope, complexity, data requirements, response structures, technological maturity, and API conventions.

Ordinarily, integrating such a wide range of backend services would require significant manual front-end implementation effort. Each solver would demand its own dedicated UI, custom-built input forms, validation logic, and output rendering mechanisms. As solvers evolved—which is common in an R&D context—every change would cascade into additional front-end work, user experience inconsistencies, and increased maintenance load.

Recognizing early the unsustainable nature of this approach, the project adopted a novel architectural strategy: a dynamic, metadata-driven user interface capable of adapting automatically to each solver based solely on its API definition. This whitepaper presents the rationale, technical underpinnings, implementation approach, and broader implications of this strategy. The solution illustrates how metadata-based UI generation can unlock significant scalability, sustainability, and operational consistency in large, distributed, multi-partner platforms.

3. The Integration Challenge in a Multi-Solver Environment

The scale and diversity of the solver ecosystem created challenges that were both technical and organisational. Each solver was developed independently by different partners, using frameworks and languages ranging from Python and Java to specialised simulation engines. Their APIs differed not only in structure but also in naming conventions, parameter definitions, validation rules, and output schemas. Some solvers consumed simple numeric inputs, while others accepted complex JSON payloads or required multi-step interaction sequences.

The central operational requirement was that all solvers—regardless of origin or complexity—had to be accessible through a uniform user interface. Users should not need to contend with the idiosyncrasies of each solver's API. Instead, the platform needed to present a coherent, predictable, and easy-to-navigate UI through which all solvers could be invoked, configured, executed, and interpreted.

Implementing this through traditional development would have required creating and maintaining dozens of discrete UI modules. Each new solver would trigger a coding cycle. Each API update would require fixes. Every interface would require independent testing. The maintenance implications were substantial: inconsistent naming, diverging user experiences, and significant risk of regressions as solvers evolved. In an ecosystem where the solver landscape was expected to grow—not shrink—this model would scale poorly and impose a structural bottleneck on the project.

4. Rationale for a Metadata-Driven UI Architecture

Against this backdrop, the need for a more adaptive, scalable solution emerged. The underlying principle was straightforward: rather than hard-coding interfaces for each solver, the front-end should generate them dynamically based on descriptions stored as metadata.

This approach aligns well with the exploratory and iterative nature of R&D projects. Unlike commercial systems, where APIs evolve more slowly and stability is prioritized, R&D systems frequently undergo redesigns, parameter changes, and rapid experimentation. A dynamic, metadata-driven interface helps absorb this volatility without imposing ongoing development cycles.

Another key benefit relates to reducing dependency chains across partners. As each solver-owning partner delivered and updated their backend services, there was no need to synchronise front-end development for every solver. Instead, the metadata layer acted as a contract: once the solver's definition was added or updated in the repository, the front-end automatically adapted.

This strategy effectively decoupled the evolution of solvers from the development of the platform UI. It also created a single governance point—the metadata repository—where

solver definitions could be validated, reviewed, versioned, and managed across the consortium.

5. Architecture Overview

The metadata-driven front-end architecture consists of three major components working in orchestration: the solver metadata repository, the dynamic UI generation engine, and the backend solver services themselves.

The metadata repository captures the full definition of each solver's API in a structured format. These definitions include endpoint locations, HTTP method requirements, expected input parameters, data types, permitted value ranges, validation constraints, recommended UI component types, and the expected output schema. By encapsulating all this information, the repository serves as the authoritative knowledge base for both input rendering and output parsing.

On the client side, the front-end includes a dynamic UI engine that retrieves metadata for the selected solver at runtime. Based on the description, it constructs the required user input forms. These forms may include free-text fields, dropdown selectors, number inputs, geographic coordinate pickers, date controls, or multi-parameter groupings, depending on what the solver requires. The front-end handles validation locally, enforcing data types and constraints before the solver is invoked.

Once the user submits their inputs, the front-end assembles the solver request based solely on the metadata definition. The solver executes and returns a response, which the UI interprets and renders dynamically. This includes mapping output fields to appropriate display components, presenting structured tables, charts, maps, or descriptive text depending on the solver's expected output schema.

The architecture is therefore characterised by a clean separation of concerns. The solver owners define their solver behaviour in metadata. The front-end interprets the metadata without requiring adjustments to its underlying code. The backend services perform their computations independently. This structure is robust, adaptable, and well-suited to environments with high solver heterogeneity.

C4 Model (Container Level) – Metadata-Driven UI

System Context

[User] ---> interacts with ---> [FOR-FREIGHT Platform UI]

Containers

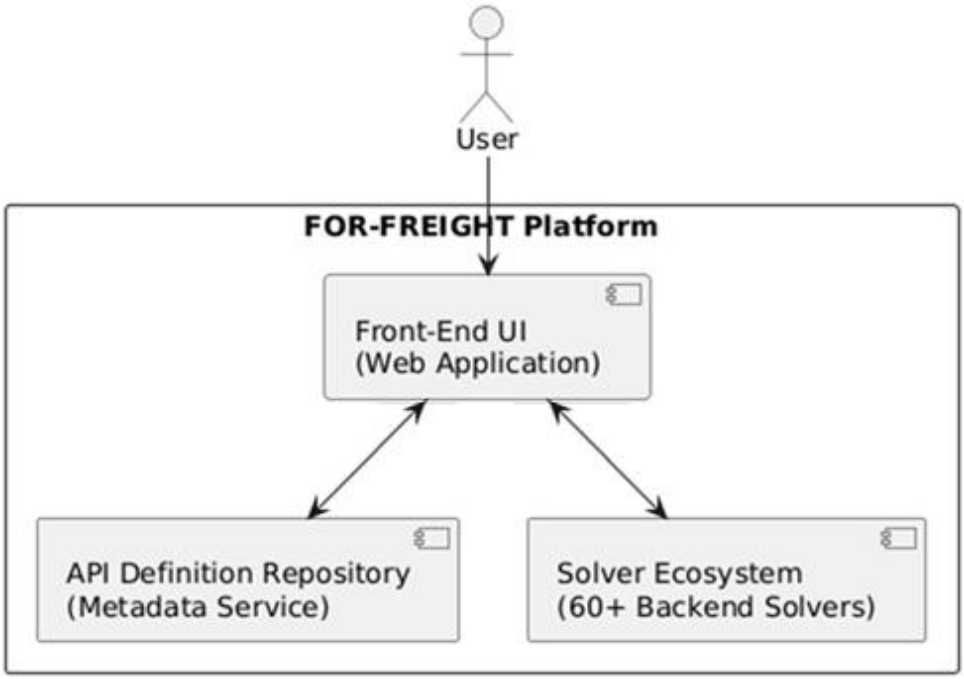

Figure 1- High-Level Architecture of the Metadata-Driven Solver Integration Framework

Figure 1 illustrates the three foundational components that enable dynamic user interface generation within the FOR-FREIGHT platform. On the client side, users interact with the platform through the Front-End UI, which functions as the central access point for invoking any solver. The front-end does not contain hard-coded forms or solver-specific logic; instead, it requests the relevant solver definition from the API Definition Repository. This repository houses detailed metadata describing the inputs, outputs, and operational characteristics of every solver integrated into the platform. The UI interprets this metadata at runtime and constructs an appropriate interface dynamically. The backend Solver Ecosystem, consisting of more than sixty heterogeneous services contributed by consortium partners, provides the computational capabilities of the platform. Each solver exposes its own REST endpoint, and once invoked by the UI, returns results back to the front-end for rendering. The architecture highlights the decoupled nature of the system: solver logic remains fully independent, metadata acts as an intermediary contract, and the front-end adapts without requiring modifications to support new or updated solvers.

Sequence Diagram – Dynamic UI Flow

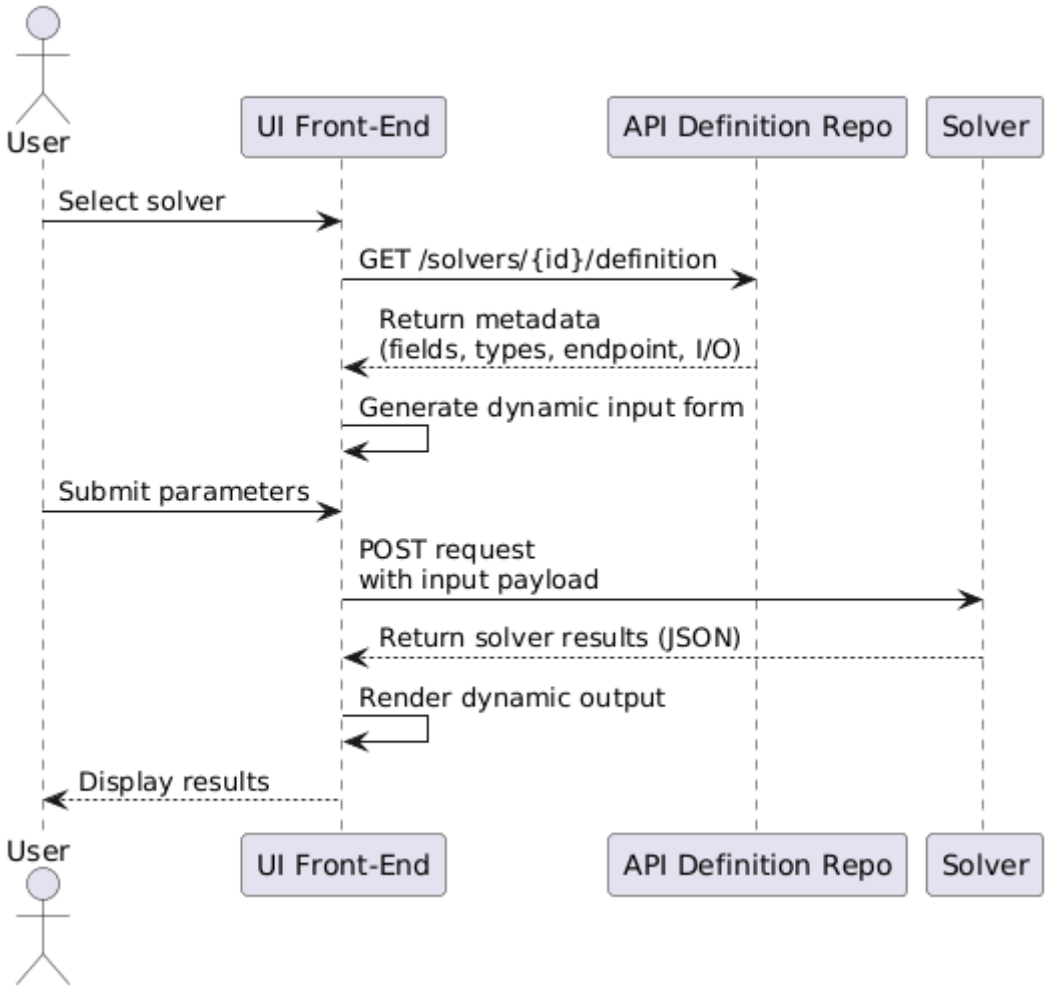

Figure 2- Dynamic Solver Interaction Sequence Diagram

The sequence diagram in Figure 2 demonstrates the complete runtime workflow that occurs when a user interacts with a solver through the dynamic UI. The process begins when the user selects a solver, prompting the front-end to request its metadata definition from the repository. The repository responds with a structured representation of the solver's expected inputs, validation rules, endpoint details, and output schema. Using this information, the front-end dynamically constructs the solver-specific input form and presents it to the user. After the user provides the required parameters and submits the request, the front-end composes a payload that conforms to the metadata definition and sends it directly to the solver's API endpoint. The solver processes the request and returns a JSON response containing the results of the computation. The UI then interprets these results, once again using the metadata to determine how to format and display them to the user. The diagram highlights the adaptive nature of the system and shows how metadata replaces manual coding throughout the invocation lifecycle, ensuring that the UI remains flexible and solver-agnostic.

Metadata Schema (Example)

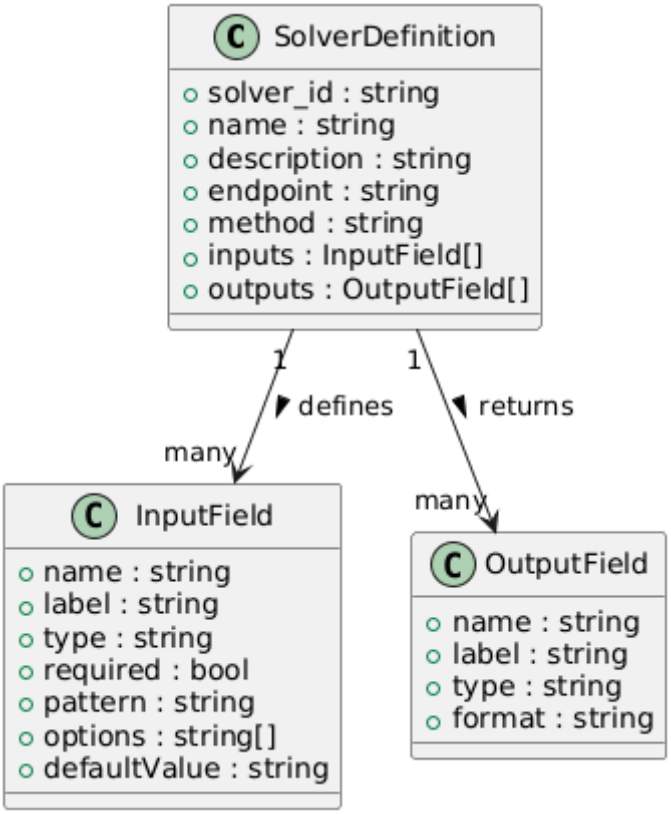

Figure 3- Solver Metadata Schema Class Diagram

The class diagram in Figure 3 illustrates the data model used to represent solver API definitions within the platform's metadata repository. The primary entity, SolverDefinition, encapsulates all information required by the front-end to generate a functional user interface. It contains fields describing the solver's identification, endpoint location, HTTP method, and high-level description. Additionally, it references two collections: InputField and OutputField. Each InputField instance describes a user-provided parameter, including its name, data type, validation constraints, label, and any selectable options. These fields guide the UI in generating the appropriate input controls and enforcing user-side validation. The OutputField class provides similar structure for solver outputs, enabling the UI to map returned JSON elements to meaningful display components such as text fields, tables, charts, or maps. The diagram conveys the structured yet flexible nature of the metadata model, showing how it captures solver diversity while maintaining a consistent interpretative framework across all services.,

UI Rendering Logic (Conceptual)

```
for each input in metadata.inputs:
   renderInputField(input.type, input.name, validationRules)
onSubmit():
   callSolver(metadata.endpoint, formValues)
onResponse():
   renderOutput(metadata.outputs, responseData)
```

6. Implementing Dynamic UI Generation in Practice

In practice, the implementation of the metadata-driven UI required careful consideration of several technical factors.

The first challenge was establishing a metadata schema expressive enough to capture the diversity of solver inputs and outputs while remaining maintainable across the project's lifespan. The schema needed to accommodate simple scalar fields as well as complex types. It also had to support human-readable labels, validation constraints, conditional logic, and rendering hints to ensure that the front-end could construct meaningful and usable interfaces.

The second challenge related to UI rendering logic. Dynamic UI generation is significantly more complex than static page rendering. The front-end needed to interpret the metadata at runtime, generate proper field components, enforce validation rules, and construct clean, user-friendly layouts. The architecture therefore required a robust form-handling engine, capable of generating nested structures, handling dynamic updates, and supporting user feedback mechanisms such as inline validation messages.

The third challenge cantered on error handling and resilience. Metadata-driven interfaces depend on the accuracy and consistency of the metadata supplied. Therefore, the platform required mechanisms to detect missing, incorrect, or incompatible metadata entries. Fall-back behaviours were designed to ensure that the UI remained functional even if solver metadata was imperfect, helping maintain usability and reduce friction during development and iteration cycles.

Finally, attention was given to front-end performance. Rendering dynamic components at scale can introduce latency, especially when forms are large or contain deeply nested structures. The development process incorporated caching strategies, lazy loading techniques, and optimised rendering paths to ensure that the user interface remained responsive regardless of the complexity of the solver definition.

7. Benefits and Impacts of the Metadata-Driven UI Strategy

The adoption of metadata-driven UI generation transformed the integration workflow within FOR-FREIGHT. One of the most significant benefits was the elimination of manual UI development for each solver. New solvers were onboarded simply by registering their metadata definitions. The moment the solver's definition was uploaded, the platform immediately supported it, without requiring a front-end update or dedicated UI development sprint.

This contributed to a dramatic reduction in development time and enabled the teams to focus resources on improving solver accuracy, user experience, and platform integrations rather than repetitive UI coding. Maintenance costs were also significantly reduced. When solver

interfaces evolved—as they often did during the project—only the metadata needed updating. The UI adapted automatically.

Another meaningful impact was the improvement in user experience consistency. Regardless of solver origin, user-facing forms followed identical layout patterns, naming conventions, validation behaviour, and interaction flows. This consistency is crucial in multimodal logistics operations, where users must be able to work efficiently with different tools without learning new interfaces for each.

An important dimension of this architecture is the transformation it brings to solver onboarding. Traditionally, integrating a new solver requires coordinated development across backend, frontend, and testing teams, with each UI element manually implemented to match the solver's API. With the introduction of the metadata repository, onboarding becomes a configuration-driven process rather than a development task. Solver owners simply register or update their solver's metadata—defining its parameters, validation rules, endpoint structure, and expected outputs—after which the platform immediately becomes capable of supporting it. No additional front-end coding, UI redesign, or deployment cycle is required. This shift not only accelerates the integration timeline but also empowers domain experts to onboard and iterate on solver definitions independently, without relying on UI developers. As a result, the platform gains a new level of agility: solvers can be introduced, updated, or deprecated rapidly, enabling continuous expansion of the ecosystem while preserving a unified and consistent interaction model for users.

Perhaps most importantly, this approach reinforces interoperability, a key principle in modern logistics digitalisation. By capturing solver interfaces as structured metadata, the platform establishes a common interpretive layer that could be reused across systems, shared with external partners, or integrated with workflow automation tools.

8. Considerations and Lessons Learned

The implementation of this architecture revealed several insights that may benefit similar initiatives in the future. While metadata-driven UI generation is powerful, it introduces the need for rigorous metadata governance. Solver owners must be trained to provide accurate definitions. Versioning policies are essential to track changes over time. Validation tools are necessary to ensure that definitions remain aligned with actual solver behaviour.

To support this governance process and ensure that solver owners could provide high-quality metadata consistently, the project complemented the technical architecture with a series of onboarding materials, including tutorials, example definitions, validation guidelines, and step-by-step instructions. These resources played an essential role in reducing ambiguity and ensuring that partners—regardless of their technical background—understood how to prepare and maintain correct solver metadata. By combining a well-defined metadata schema with practical instructional materials, the project lowered the entry barrier for contributors, improved the reliability of submitted definitions, and accelerated the onboarding of new solvers throughout the platform's lifecycle. This documented and guided approach proved particularly valuable in a multi-partner R&D environment, where contributors often differ in tooling, experience, and familiarity with the platform's internal conventions.

Additionally, although shifting part of the logic to the front-end diverges from strict separation-of-concerns principles, the decision was pragmatic and justified. In an environment characterised by rapid iteration and diverse solver contributions, the ability to adapt dynamically outweighed the drawbacks of front-end complexity.

Finally, the approach demonstrated how important it is for R&D platforms to prioritise flexibility. Instead of relying on rigid architectures that resist change, embracing metadata-driven, adaptive systems allows platforms to evolve alongside their technical ecosystems.

9. Conclusion

The metadata-driven UI architecture deployed within FOR-FREIGHT represents a forward-looking solution to one of the core challenges faced by modern digital logistics platforms: integrating, managing, and interacting with a large, diverse set of backend services. By shifting from static, manually coded interfaces to dynamic, metadata-generated ones, the project achieved greater scalability, improved consistency, reduced maintenance overhead, and enhanced long-term sustainability.

The approach not only met the immediate needs of integrating sixty-plus solvers but also established a foundational design pattern applicable to future developments in multimodal logistics infrastructures. As digitalisation accelerates and the ecosystem of logistics services grows, metadata-driven UI generation offers a reliable and extensible method for managing complexity while delivering a coherent and efficient user experience.