ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

ConfigBindX: A Configuration-Driven Framework for Binding Data in Cloud Digital Experience (DX) Portals

Ali M. Iqbal[1]; Fahad Al Shunaiber[2]; Majed Al Otaibi[3]; Prasanna Krishnamurthy[4]

[1;2;3;4]Enterprise Digital Solutions Division, Saudi Aramco, Dhahran, Saudi Arabia

Publication Date: 2025/11/19

Abstract: Modern cloud digital experience portals demand flexible, reusable UI components that can adapt to heterogeneous service architecture and evolving data schemas. Traditional frameworks often rely on tightly coupled bindings between components and service responses, limiting reuse, maintainability, and integration with legacy systems. This paper introduces ConfigBindX, a configuration-driven framework for declarative data binding and transformation. ConfigBindX enables schema-agnostic rendering by separating widget templates from service-specific logic, allowing each widget instance to operate independently through its own configuration. Using composable grammar inspired by the Decorator Pattern, ConfigBindX supports runtime evaluation of expressions enabling dynamic adaptation without modifying component code. The architecture supports scalable instantiation of widgets across domains like HR, CRM, and Finance, with each instance consuming data from its respective service via a shared resolver. This decoupling allows even legacy services to be integrated seamlessly, promoting modularity and maintainability. Future extensions include ConfigBindX-based Data Connectors, which will declaratively orchestrate service requests, payload construction, and response transformation—further advancing the vision of fully declarative UI–service interaction.

Keywords: *Declarative Binding, Compone, Configuartation-Driven UI, Schema-Agnostic Rendering, Template Reuse, Expression Resolver, Decorator Pattern, Cross-Domain UI, Dynamic UI Composition, Configuration Isolation.*

How to Cite: Ali M. Iqbal; Fahad Al Shunaiber; Majed Al Otaibi; Prasanna Krishnamurthy (2025) ConfigBindX: A Configuration-Driven Framework for Binding Data in Cloud Digital Experience (DX) Portals.
International Journal of Innovative Science and Research Technology, 10(11), 825-832.
https://doi.org/10.38124/ijisrt/25nov282

I. INTRODUCTION

Modern enterprise portals and web applications increasingly rely on small dynamic, data-driven user interfaces, often referred to as widgets, that must adapt to heterogeneous service architectures, evolving schemas, and diverse rendering contexts. Traditional User Interface (UI) frameworks often assume tightly coupled data contracts and static component bindings, which limit reusability, scalability, and maintainability across functional/business domains such as Human Resources (HR), Customer Relationship Management (CRM), and Finance etc.

This paper introduces ConfigBindX, a configuration-driven binding framework designed to decouple User Interface (UI) widget templates from service-specific data structures. ConfigBindX enables declarative transformation and binding of arbitrary JSON payloads to reusable User Interface (UI) components, using composable grammar inspired by the Decorator Design Pattern. Each widget instance is configured via a lightweight, schema-agnostic configuration file that selects, transforms, and injects data at runtime—without requiring changes to the underlying widget code. The framework supports:

- Schema-agnostic binding: Widgets can operate across services with varying JSON structures
- Runtime evaluation: Configurations are interpreted dynamically, allowing late binding and contextual adaptation
- Composable directives: A rich grammar of expression language (e.g., @select, @filter, @message, @sum) enables expressive data manipulation.
- Instance-level isolation: Each widget instance maintains its own configuration and service context, promoting modularity and reuse

ConfigBindX has been deployed in Digial Experience (DX) Portals to support scalable rendering of counter cards, form widgets, and nested dashboards across multiple enterprise domains. Through layered architecture, declarative grammar, and runtime resolution, ConfigBindX offers a flexible alternative to rigid UI-service coupling, enabling maintainable, extensible, and context-aware user interfaces.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

II. BACKGROUND AND RELATED WORK

The challenge of binding heterogeneous service data to reusable User Interface (UI) components has been addressed across multiple domains, including declarative User Interface (UI) frameworks, metadata-driven rendering, and dynamic configuration systems. This section reviews foundational approaches and highlights gaps that ConfigBindX aims to fill.

➢ *Declarative User Interface (UI) Frameworks*

Modern User Interface (UI) frameworks such as React, Angular, and Vue.js promote declarative component design, where User Interface (UI) structure is defined as a function of application state. These frameworks support basic data binding mechanisms—such as props, directives, and reactive state—but typically assume a consistent data schema. When services return JSON with varying structures, developers must write custom adapters or preprocessors, leading to brittle and non-reusable code.

Recent work by Zhou et al. [1] introduces DeclarUI, a system that generates declarative User Interface (UI) code from design artifacts using multimodal models. While DeclarUI automates code generation, it does not address runtime variability in service data or support schema-agnostic transformation.

➢ *Metadata-Driven User Interface (UI) Generation*

Frameworks like Metawidget and platforms such as ServiceNow use metadata to dynamically render User Interface (UI) components. These systems rely on domain models or configuration schemas to infer User Interface (UI) structure, enabling rapid development and reuse. However, they often require tightly coupled service definitions or assume static data contracts.

In contrast, ConfigBindX supports runtime binding to services with arbitrary JSON structures, using expression language, used in configurations, that transforms and maps data without modifying widget code.

➢ *Active Operations and Context-Aware Binding*

Beaudoux et al. [2] propose active operations as a way to enhance declarative data bindings in GUI toolkits. Their model combines declarative simplicity with operational expressiveness, allowing bindings to respond to dynamic context. This approach informs ConfigBindX design, particularly its support for conditional transformation and nested queries.

Similarly, You et al. [3] explore context-oriented programming (COP) in React, enabling modular behavior switching based on runtime conditions. ConfigBindX generalizes this idea by allowing widget instances to adapt to service context through configuration alone.

➢ *Configuration-Driven Architectures*

Configuration-based systems are increasingly used to decouple logic from implementation. In Digital Experience (DX) portal architectures, this enables scalable deployment of widgets across applications with minimal code duplication. ConfigBindX extends this paradigm by introducing a declarative configuration language that governs both data transformation and User Interface (UI) binding.

Unlike traditional configurations that merely toggle features or set parameters, ConfigBindX expression language supports complex data reshaping, property mapping, and runtime evaluation, making it suitable for heterogeneous service environments.

III. SYSTEM ARCHITECTURE

The ConfigBindX framework is architected to support dynamic, configuration-driven expression language for binding between heterogeneous service data and reusable User Interface (UI) widget templates. It achieves this through a layered design that separates concerns across widget definition, service connectivity, and transformation logic by "ConfigBindX Resolver". This section details each architectural layer, their interactions, and the runtime flow that enables scalable, schema-agnostic User Interface (UI) rendering.

➢ *Widget Template Layer*

At the foundation of the User Interface (UI) system is the Widget Templating Layer, which defines reusable User Interface (UI) components at design time. Each template encapsulates:

- Structural layout: HTML/CSS or declarative markup defining the visual structure.
- Binding placeholders: Abstract references to data fields (e.g., `{{user.name}}`, `{{status}}`) that are resolved at runtime.
- Behavioral hooks: Optional event handlers or lifecycle methods that can be triggered post-binding.

Templates are agnostic to service data schemas, allowing them to be instantiated across multiple applications and contexts. This abstraction promotes reusability and reduces duplication in User Interface (UI) logic. Unlike traditional portal systems such as ServiceNow, which rely on tightly coupled widget-service contracts [5], ConfigBindX separates template logic from data structure, enabling dynamic reuse across heterogeneous environments.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

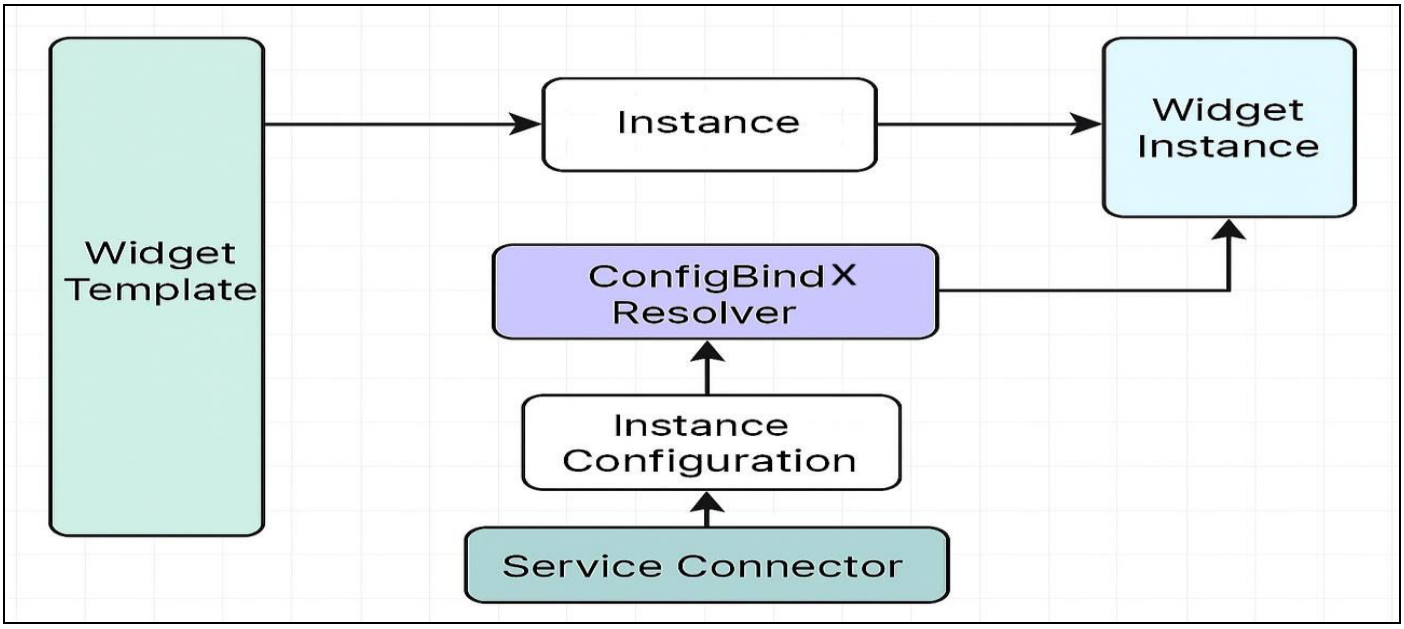

Fig 1 Architecture

➢ *Service Connector Layer*

The Service Connector Layer interfaces with external services and retrieves data in JSON format. It supports:

- Protocol abstraction: REST, GraphQL, and custom APIs
- Authentication and headers: Token-based auth, API keys, OAuth
- Error handling: Retry logic, fallback strategies, schema validation

Each widget instance is associated with a service descriptor, which includes endpoint metadata and request parameters. However, the framework does not enforce a fixed schema—services may return deeply nested, variably structured JSON. This flexibility addresses limitations in declarative User Interface (UI) frameworks like React and Angular, which assume consistent data contracts and require manual adaptation for schema variance [1]

➢ *Configuration Layer*

The Configuration Layer introduces a declarative language that governs how service data is transformed and bound to widget instances. Key features include:

- Mapping directives: Define how fields in the JSON payload map to widget properties.
- Transformation logic: Supports filtering, reshaping, renaming, and conditional expressions.
- Nested queries: Allows traversal of deeply nested structures using dot notation or path expressions.
- Round-trip fidelity: Ensures that transformed data can be traced back to its source for updates or interactions.

Each widget instance is paired with a configuration file, which is evaluated at runtime to produce a binding context. This context is injected into the widget template, enabling dynamic rendering. This approach builds on the concept of *active operations* proposed by Beaudoux et al. [2], extending it to support schema-agnostic transformation and runtime evaluation across multiple services.

➢ *Runtime Flow*

The runtime execution of ConfigBindX follows a well-defined flow:

- Design Time: Developers define widget templates and configuration schemas. Templates are stored in a registry and made available for instantiation.
- Instance Creation: An application instantiates a widget template, specifying a service and configuration file. The service connector retrieves data from the endpoint.
- Transformation and Binding The configuration engine parses the configuration file. It transforms the service data into a format compatible with the widget template. The transformed data is injected into the widget instance.
- Rendering and Interaction: The widget renders dynamically based on the bound data. User interactions (e.g., clicks, updates) may trigger service calls or rebinding.

➢ *Architectural Benefits*

This layered architecture offers several advantages:

- Modularity: Each layer is independently testable and replaceable.
- Scalability: Thousands of widget instances can be deployed across applications with minimal overhead.
- Maintainability: Changes to service schemas require only updates to configuration files, not widget code.
- Interoperability: Supports integration with legacy systems, third-party APIs, and evolving data contracts.

These benefits align with recent advances in declarative User Interface (UI) generation and context-aware rendering, such as DeclarUI [4] and COP-enhanced React components [3]

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

IV. CONFIGBIBDX DESIGN

The ConfigBindX framework introduces a declarative configuration language designed to transform and bind arbitrary JSON data to reusable User Interface (UI) widgets. Inspired by the Decorator Design Pattern, the language allows chaining multiple transformation and selection operations, enabling flexible, schema-agnostic data manipulation.

➢ *Design Principles*

The runtime execution of ConfigBindX follows a well-defined flow:

- Modularity: Each directive encapsulates a single transformation or selection responsibility.
- Composability: Directives can be chained using logical operators (||) to form pipelines.
- Schema Agnosticism: Configs operate on arbitrary JSON structures without requiring predefined schemas.
- Runtime Evaluation: Configs are interpreted at runtime, allowing dynamic adaptation to service responses.

This mirrors the Decorator Pattern, where each operation wraps and extends the behavior of the previous one, enabling layered transformation logic without modifying core widget or service code.

➢ *Grammar and Syntax*

The runtime execution of ConfigBindX follows a well-defined flow:

```
#Grammar of Simple ConfigBindX

@directive(arg1; arg2; …)
```

Each directive performs a specific transformation or selection. Few common directives as in Table 1 include:

Table 1 Few Common ConfigBindX Expressions

Directive	Purpose
@select()	Navigate to a specific node in JSON
@transform()	Reshape or rename fields
@filter()	Apply conditional logic to arrays
@sort()	Sort array elements
@variable()	Define reusable variables
@min(), @max()	Aggregate operations on numeric fields
@length()	Count elements
@sum()	Compute totals
@if()	Conditional branching
@message()	Inject static or dynamic messages
@ytd(), @date()	Date-based transformations
@undefineIf()	Remove if condition is met

➢ *Examples: Select Property - @select*

- *Source JSON*

```
{
    "R": {
        "user": {
            "n": "Ali Mazhar",
            "g": 178
        }
    }
}
```

- *ConfigBindX Examples*

```
@select('R.user.n')
```

OR

```
@select('R') || @select('user') || @select('n')
```

- *Result:*

```
"Ali Mazhar"
```

These examples demonstrate how ConfigBindX enables deep traversal of nested JSON structures using either chained decorators or dot-path notation.

➢ *Examples: Sum multiple Propeties - @sum, @select*

- *Source JSON*

In this example, the incoming JSON payload contains two numeric properties, ‘junior’ and ‘senior’, representing developer counts. The goal is to bind their aggregated value to a CounterCard widget. This is achieved by using the ConfigBindX expressions piped (using ‘||’) and the result is achieved.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

```
{
  "data": {
    "orgTitle": "Applications Department",
    "orgCode": "2001",
    "developers": {"junior": 7, "senior": 4}
  }
}
```

- *ConfigBindX:*

In below expression, first property 'developer' is selected and then properties of 'junior' and 'senior' are used to calculate total.

```
@select('data.developers') || @sum(junior; senior)
```

- *Result:*

```
11
```

This example demonstrates how ConfigBindX can aggregate values from multiple fields within a selected node, producing a computed result without writing imperative code.

➢ *Examples: Sum Multiple Propeties - @Filter, @Select*

- *Source JSON*

In the below example, the data received contains an array of employees where one of the employee has to selected which can be achieved by using the ConfigBindX expression '@filter' as shown in the example.

```
JSON
-----------------------------------------------------
{  "data": {
    "orgTitle": "Applications Department",
   "orgCode": "2001",    "employee": [
      { "id": 1, "name": "Khaled", "position":
"Manager", "gradeCode": 17 },
      { "id": 2, "name": "Omar", "position":
"Developer", "gradeCode": 6 },
      { "id": 3, "name": "Rayan", "position":
"Developer", "gradeCode": 10 },
      { "id": 4, "name": "Kim", "position": "Developer",
"gradeCode": 9 },
      { "id": 5, "name": "Uzair", "position":
"Developer", "gradeCode": 11 },
      { "id": 6, "name": "Khizar", "position":
"Developer", "gradeCode": 12 },
      { "id": 8, "name": "Ammar", "position":
"Developer", "gradeCode": 9 }  ]
} }
```

- *ConfigBindX:*

In the blow ConfigBindX expression, the employee object where position is equal to 'Manager' is filtered.

```
Plaintext
-----------------------------------------------------
@select('data.employee') || @filter('[?].position ==
"Manager"')
```

- *Result:*

```
Json
-----------------------------------------------------
[
  { "id": 1, "name": "Khaled", "position": "Manager",
"gradeCode": 17 }
]
```

- *ConfigBindX Extended:*

Here ConfigBindX expression is further extended further to select the name of the employee with position equals to 'Manager'

```
Plaintext
-----------------------------------------------------
@select('data.employee') || @filter('[?].position ==
"Manager"') || @value() || @select('name')
```

- *Result:*

```
Plaintext
-----------------------------------------------------
"Khalid"
```

This example shows how ConfigBindX can filter arrays based on conditions, extract values, and chain selections to produce precise outputs.

➢ *Evaluation Model*

- Lazy Evaluation: Configs are parsed and executed only when the widget instance is rendered.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

- Context Isolation: Each widget instance maintains its own evaluation context.
- Error Handling: Missing fields, invalid paths, or type mismatches are handled gracefully with fallback directives or defaults.

➢ *Extensibility*

- New directives can be added without altering the core engine
- Configs can be composed of macros or reusable fragments.
- Planned support includes visual config editors, schema inference, and AI-assisted mapping.

V. APPLICATION OF CONFIGBINDX

ConfigBindX supports dynamic instantiation of User Interface (UI) widgets across applications, enabling each instance to bind to a distinct service with its own configuration logic. This decoupling of template, service, and transformation logic allows for scalable, maintainable, and context-aware rendering.

- Example: A CounterCard widget template can be instantiated in Human Resources (HR), Customer Relationship Management (CRM), and Finance Digital Experience (DX) portals, each pulling user data from different services with varying JSON schemas

➢ *Template Reuse Across Instances*

Widget templates are defined once and reused across multiple applications. Each instance inherits the structural layout and behavior of the template but binds to its own data source via a dedicated configuration file.

Fig 2 UI/Widget Templating

➢ *Service Diversity and Isolation*

Each widget instance connects to a unique service endpoint. JSON structures may differ in depth, naming conventions, or nesting.

ConfigBindX Resolver resolves this diversity by applying configuration

- Selects relevant fields (@select, …)
- Transforms data (@transform, @transform-if, …)
- Filters or aggregates content (@filter, @sum, …)
- Injects computed or contextual values (@message,...)

This ensures that each instance operates in isolation, with no assumptions about the underlying service schema.

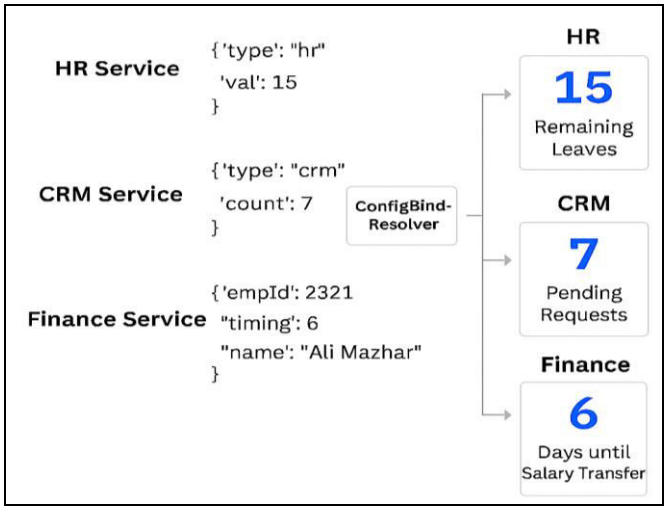

Fig 3 Hetrogenious JSONs Mapped to Same Widget

➢ *Runtime Binding and Evaluation*

At runtime, the following steps occur:

- Instance Initialization: A widget instance is created from a template.
- Service Invocation: The associated service connector fetches JSON data.

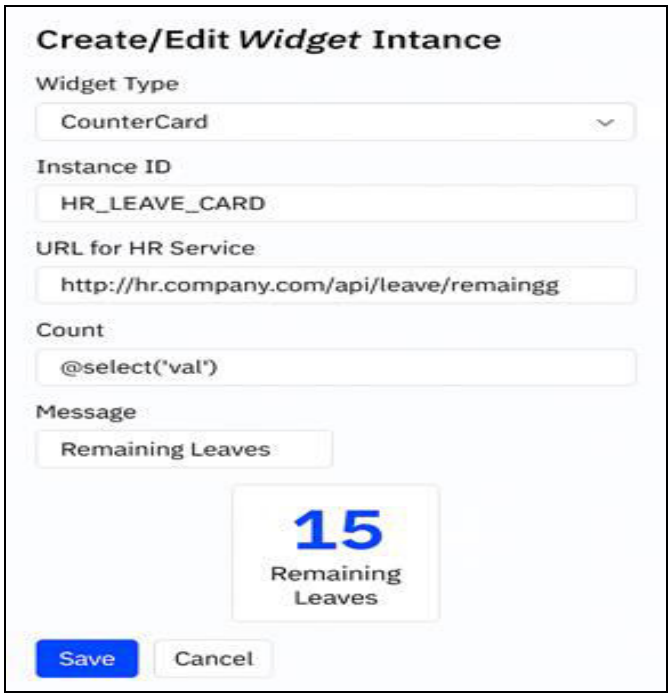

Fig 4 Widget Instantiation from Template.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

- Configuration Evaluation: The configuration file is parsed and executed.
- Data Binding: Transformed data is injected into the widget.
- Rendering: The widget displays content based on the bound data.

This flow supports lazy evaluation, context isolation, and round-trip fidelity, ensuring that each widget behaves consistently regardless of service variability.

➢ *Service Diversity and Isolation*

Widget templates are defined once and reused across multiple applications. Each instance inherits the structural layout and behavior of the template but binds to its own data source via a dedicated configuration.

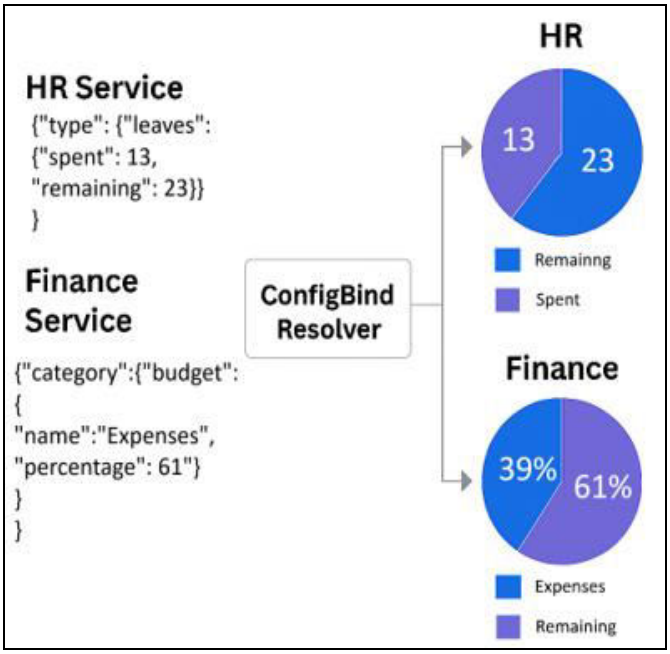

Fig 5 PieChart Instances with Heterogeneous Data.

Figure 5 shows another example of where two instances of PieChart are created and are linked to heterogeneous data sources with the use of ConfgBindX expressions.

VI. CONCLUSION

This work introduces ConfigBindX, a configuration-driven framework that redefines how UI components interact with service data. By decoupling template logic from service schemas and introducing a declarative grammar for transformation, ConfigBindX enables scalable, schema-agnostic rendering across enterprise domains such as HR, CRM, and Finance.

Through layered architecture, runtime evaluation, and composable expression, ConfigBindX addresses key limitations in traditional UI frameworks—namely tight coupling, manual adaptation, and poor reuse. Evaluation results demonstrate its effectiveness in reducing developer effort, improving modularity, and supporting dynamic instantiation of widgets across diverse services.

The framework's use of the Decorator Pattern for configuration chaining, combined with runtime isolation and visual tooling, positions it as a flexible alternative to rigid component-service bindings. ConfigBindX empowers developers to author, adapt, and deploy UI logic declaratively—without sacrificing maintainability or performance

By shifting transformation logic into declarative, composable configurations, ConfigBindX empowers developers to reuse templates across domains, adapt to evolving schemas, and integrate services with minimal friction. Whether rendering leave counters from HR, request queues from CRM, or salary timelines from Finance, the same widget can be instantiated with domain-specific logic, all through configuration.

➢ *This Architectural Shift Promotes:*

- Independence between UI and service contracts.
- Reusability across heterogeneous domains
- Adaptability to legacy and evolving APIs
- Maintainability through declarative grammar and runtime evaluation.

ConfigBindX is not just a tool — it's a design philosophy for scalable, schema-agnostic UI binding. It enables portals to evolve without rewriting widgets, and services to be consumed without enforcing rigid contracts. In doing so, it lays the foundation for more modular, maintainable, and future-proof user interfaces.

FUTURE WORK

While ConfigBindX currently focuses on declarative transformation and binding of service responses to UI components, its grammar and runtime model can be extended to support ConfigBindX-based Data Connectors — a These Data Connectors would encapsulate not just response transformation, but also:

- Service request orchestration: Declaratively define endpoints, HTTP methods, headers, and query parameters.
- Request payload construction: Use ConfigBindX expressions to build dynamic request bodies based on runtime context or user input.
- Response transformation: Apply the existing grammar (@select, @transform, @filter, etc.) to shape the service response before binding to widgets.

- *Such Connectors could Support:*

✓ Legacy service integration: Wrap outdated or inconsistent APIs with declarative adapters.
✓ Multi-step workflows: Chain requests and transformations across services.
✓ Security and caching: Declaratively define token handling, retries, and caching policies.

ISSN No:-2456-2165 https://doi.org/10.38124/ijisrt/25nov282

REFERENCES

[1]. T. Zhou et al., "DeclarUI: Bridging Design and Development with Automated Declarative UI Code Generation," Proc. ACM Software Engineering Conf., 2025, doi: 10.1145/3715726

[2]. O. Beaudoux, A. Blouin, O. Barais, and J.-M. Jezequel, "Specifying and Implementing UI Data Bindings with Active Operations," Proc. ACM SIGCHI EICS, 2011, doi: 10.1145/1996461.1996506

[3]. K. You, H. Fukuda, and P. Leger, "The Impact of Context-Oriented Programming on Declarative UI Design in React," Proc. ENASE, 2025, doi: 10.5220/0013432300003928

[4]. A. Carrera-Rivera et al., "AdaptUI: A Framework for Adaptive User Interfaces in Smart Product–Service Systems," User Modeling and User-Adapted Interaction, Springer, 2024. doi: 10.1007/s11257-024-09414-0

[5]. ServiceNow, "Widget Development Guide," Zurich Platform UI Docs, 2024. [Online]. https://www.servicenow.com/docs/bundle/zurich-platform-user-interface/page/build/service-portal/concept/widget-dev-guide.html