# Declarative UI Architecture Based on ECS and Reactive Systems

## 1. Introduction

Modern user interfaces are increasingly expected to be responsive, dynamic, and adaptable across a wide range of platforms—from high-performance desktop tools to mobile applications and embedded environments. Traditional object-oriented UI frameworks, however, often struggle to meet these demands, especially when performance, modularity, and real-time responsiveness are critical. In this paper, we propose a data-driven approach to UI development based on a combination of modern programming paradigms like Entity Component Systems (ECS) and the Reactive Programming approach. Unlike conventional class-based hierarchies, an ECS treats UI elements as independent entities composed of isolated, typed components. This separation allows for highly modular, cache-friendly processing and enables both fine-grained control and large-scale parallelism.

The system is implemented in C#, taking advantage of its powerful runtime, safe memory model, and high developer productivity. C# offers ideal conditions for building tooling and editor applications—domains where interactive UIs, hot-reload capabilities, and dynamic composition are essential. At the same time, C# can interoperate with native libraries via DllImport or low-level APIs like NativeLibrary, allowing seamless integration with performance-critical engines written in C or C++.

This design deliberately separates UI composition from rendering and runtime logic. UI definitions can be described and modified dynamically in C#, using parsers for formats like HTML5, CSS, or XAML, and then exported as ECS data. These ECS-structured descriptions can be consumed by any native engine capable of reading and interpreting component-based data—whether written in C++, Rust, or Zig—so long as it adheres to a C-compatible ABI.

Our goal is to provide a lightweight, declarative UI model that is platform-agnostic, extensible, and capable of scaling from minimal embedded toolsets to full-featured editor environments. Instead of attempting to replicate existing frameworks like Flutter or WPF, this system focuses on the core principles of reactivity, hierarchy management, and layout orchestration, with a minimal runtime footprint and a maximal degree of control.

This paper does not present a full-scale UI engine or design system. Rather, it offers a modular reference implementation-suitable for tool developers, engine architects, or system programmers—demonstrating how such a system can be designed around data, not inheritance

## 2. System Design and Architecture

Modern UI frameworks must reconcile performance, responsiveness, and modularity with the need for dynamic composition and cross-platform integration. The architecture described in this paper is

built around a data-driven model using the Entity Component System (ECS) pattern, supported by reactive data flows and a strict separation between data and logic. This section outlines the system's core design and explains the reasoning behind its key components.

## 2.1 Architectural Overview

The framework is divided into distinct layers:

1. Data Model (ECS-based): Encodes the UI state using entities and typed components.
2. System Layer: Operates on matching component sets to handle layout, input, animation, etc.
3. Rendering Interface: A pluggable backend (e.g., Skia) that interprets rendering data.

The overall data flow is strictly unidirectional:

---

Declarative UI → ECS Data → Systems → Render Commands → Output

---

This architecture allows full separation of runtime concerns, enabling tools to construct UIs without hardwiring logic or behavior. ECS enables clean, cache-friendly iteration over only the necessary parts of the system

## 2.2 Entity-Component Model

**What is ECS?**

The Entity Component System (ECS) is a data-oriented architectural pattern that replaces traditional inheritance trees with a flat structure composed of:

- Entities: Simple identifiers
- Components: Typed data blocks
- Systems: Logic processors that operate on specific component combinations

ECS excels at performance-critical applications by reducing indirection, improving cache locality, and separating data from logic.

**ECS in this Framework**

- Entities are compact identifiers with embedded versioning and bitflags
- Components are stored in sparse, cache-friendly arrays
- Systems iterate linearly over filtered entity sets, e.g., layout or input targets
- Hierarchical structure is encoded using an Order-based model (see 2.3)

## 2.3 Hierarchy and Layout Management

**Hierarchical Representation**

Instead of storing explicit child lists, the framework uses an Order property to encode hierarchy depth. A layout-ordered array enables subtree iteration using simple depth comparisons.

To improve flexibility, entities can optionally mark themselves as "external trees" using a bitflag. These nodes defer child tracking to a separate structure, e.g.:

---

```
struct TreeNode {
   EntityId Entity;
   List<EntityId> Children;
}
```

---

This hybrid model balances traversal performance with insert/delete flexibility, making it ideal for dynamic controls like ListView or TreeView.

**Layout System**
The layout system runs as a stateless pass over all relevant entities. Using dirty flags embedded in EntityId, it efficiently recalculates subtree sizes and positions. Layout is top-down (constraints) and bottom-up (size resolution), similar to Flutter's render pipeline, but entirely data-driven

## 2.4 Reactive Streams and Events

**What is Reactive Programming?**
Reactive programming is a paradigm centered around data streams and change propagation. In this model, components react to incoming events, state changes, or time-based triggers by emitting updates through observable channels.

**Reactive Integration**
The framework uses a lightweight reactive core:

- Stateless streams connect inputs, timers, or data providers to ECS mutations
- Observers subscribe to entity state changes and trigger re-layout, animations, etc.
- Systems can route reactive events (e.g. pointer input) directly to the correct entities via spatial indices (e.g., R-Tree)

This approach enables highly dynamic behavior without polling or callback entanglement

# 3. Component Model and Layout System

The heart of the UI framework lies in its component model: a declarative, modular system that defines the visual and behavioral structure of user interface elements. This chapter introduces the minimal set of components used to construct layoutable UI entities, along with the systems that interpret them

## 3.1 Layout-Relevant Components

The framework defines a set of basic, composable components that collectively determine layout behavior:

- LayoutComponent: Holds size and position
- HierarchyLink: Encodes parent reference and hierarchy depth
- Padding, Margin: Optional spacing definitions
- FlexProperties: Defines expansion and alignment behavior in flexible containers

- LayoutDirtyFlag: Encoded in the EntityId as a bitflag

This minimal set ensures the layout system operates purely on data, with no implicit object hierarchy or runtime polymorphism

## 3.2 Flat Hierarchy Encoding

The hierarchy of UI elements is encoded linearly using an Order value. Entities are stored in a contiguous, depth-sorted array (or an index view) such that children immediately follow their parent in memory. This enables efficient subtree traversal using simple range checks:

---

if (entity.Order > parentOrder) { /* it's a child */ }

---

To avoid O(n) insertion costs, entities with highly dynamic child structures can use a TreeNode fallback with an explicit child list. A flag in the EntityId signals this behavior, allowing both static and dynamic trees to coexist

## 3.3 Layout System Walkthrough

The layout system operates as a system pass over all layoutable entities. Its execution consists of two main phases:
Phase 1: Constraint Propagation

- A top-down walk distributes constraints (e.g. max width/height)
- Each entity calculates its local constraints based on parent data

Phase 2: Size Resolution

- A bottom-up traversal resolves actual sizes
- Entities compute their own size and propagate updates upward

Dirty state is tracked via a layout-specific bit in the EntityId. Only dirty subtrees are reprocessed, improving performance in interactive or frequently updated UIs

## 3.4 Example: Static Flex Container

A flex-style container can be implemented as a system that:

- Scans child entities of a FlexContainer
- Uses their FlexProperties to determine proportional size allocation
- Updates each child's LayoutComponent with calculated bounds

This system does not rely on class inheritance or explicit widget types. Instead, behavior is entirely data-driven and composable

## 3.5 Benefits of the Data-Driven Layout Model

| **Feature** | **Description** |
| --- | --- |
| Statelessness | Systems do not hold internal state—everything is driven by data |

| | |
|---|---|
| Separation of concerns | Layout logic is decoupled from visual rendering |
| Flexibility | Different layout strategies (grid, stack, flex) can coexist modularly |
| Performance | Linear passes, minimal indirection, tight data locality |

This model enables predictable, deterministic layout processing—ideal for tools, editors, and dynamically composable UI systems

# 4. Events and Interaction

User interaction in this framework is handled through a reactive, data-driven event system. Rather than using callback chains or widget-bound handlers, the system interprets input as streams of events that are dispatched, filtered, and transformed before affecting UI state

## 4.1 Event Streams

Interaction begins with platform input (pointer, keyboard, touch) being normalized into internal event streams. These are then propagated into the ECS domain using reactive constructs:

- Input sources emit PointerEvent, KeyEvent, or GestureEvent structs
- Systems subscribe to relevant streams and transform them into ECS mutations
- Observers may react to state changes and trigger layout or visual updates

This flow replaces imperative event handlers with composable, stateless reactions

## 4.2 Spatial Event Dispatching

To map input to UI entities, the framework employs spatial queries using a dynamic acceleration structure—typically an R-Tree:

- Each entity with an input-relevant component (Hitbox, Interactive) registers a bounding box
- Incoming pointer/touch events are resolved to one or more overlapping entities
- Matching entities are queued for processing in event priority order

The R-Tree enables fast hit-testing even with deeply nested or overlapping UI trees

## 4.3 Input Components

Entities can become interactive by attaching one or more input components:

- Hitbox: Defines the spatial bounds of interaction
- PointerState: Tracks hover/ press/ release status
- FocusTag: Marks entities as keyboard-focusable
- GestureBinding: Connects gestures (tap, drag) to command streams

These components are interpreted by input systems, not bound to a class hierarchy or widget type

### 4.4 Event Routing and Transformation

Once an event reaches a target entity, it may be:

- Consumed (stopping propagation)
- Transformed (e.g. normalized coordinates, adjusted delta)
- Forwarded (to parent/child based on routing rules)

Routing follows declarative flags such as RouteToParent, Bubble, or Capture, enabling flexible composition without deeply nested handler logic

### 4.5 Example: Button Press Flow

1. Pointer event enters the system via normalized PointerDown
2. R-Tree resolves the hit entity with a Hitbox
3. If the entity has a GestureBinding, a press gesture stream is activated
4. The press stream emits a Command, modifying ECS state
5. A LayoutDirtyFlag may be set, triggering reflow in the next pass

This flow is declarative, testable, and fully detached from runtime object behavior

### 4.6 Benefits of Reactive Interaction

| **Feature** | **Description** |
|---|---|
| Stateless Dispatch | No need for handler objects or callback references |
| Spatial Resolution | Accurate, efficient hit-testing via spatial indices |
| Composition-Friendly | Events can bubble, capture, or transform through ECS logic |
| Platform-Neutral | Input normalized at source, abstracted from OS-specific APIs |

This event system allows building complex, interactive interfaces without sacrificing modularity or performance

## 5. Rendering System

The rendering layer in this framework is designed to be minimal, deterministic, and data-driven. It interprets renderable ECS components and translates them into drawing commands for a backend such as Skia or another GPU-accelerated 2D renderer

### 5.1 Rendering Model

Rendering operates as a stateless pass over visible entities:

- Only entities with visual components are considered

- Systems compute their final visual state from layout and style components
- Output is a stream of rendering commands (e.g., draw rect, draw text)

These commands are sent to a rendering backend, which is abstracted away from the UI model and may be replaced without affecting UI logic

## 5.2 Visual Components

The following components define what is rendered:

- Background: Solid fill or gradient
- Border: Thickness, color, radius
- TextRender: Text string, font, alignment
- ImageRender: Texture ID or source reference
- Opacity: Optional override for global alpha

Additional components such as Clipping, Shadow, or ZIndex may be added to enrich visual behavior

## 5.3 Render Pass Execution

The render system traverses layout-sorted entities and emits drawing commands in the following order:

1. Backgrounds
2. Borders
3. Images and content
4. Text
5. Overlay elements (if any)

This fixed order ensures consistent layering and predictable output. Systems may batch similar draw calls for performance optimization

## 5.4 Transformations and Effects

Each renderable entity may include geometric transforms derived from layout data:

- Translation (X, Y)
- Scale (optional)
- Rotation (optional)

These are applied via a render stack and translated to GPU transformations where applicable.
Opacity and clipping are also interpreted during this phase, allowing for smooth composition of semi-transparent or masked elements

## 5.5 Backend Abstraction

The framework does not mandate a specific graphics library. Instead, the rendering layer:

- Emits a stream of primitive instructions
- Provides bounding box and style data per draw call
- Delegates actual drawing to a pluggable backend (e.g. Skia,

Veldrid, WebGPU)

This allows developers to target different platforms or embed the UI into existing rendering pipelines

### 5.6 Example: Text Label Rendering

1. TextRender component holds the text and styling
2. Layout system computes bounding box and alignment
3. Render system emits a DrawText command with:
    - Position
    - Font properties
    - Clipping box
4. Backend receives the command and draws the text using the correct glyphs

No subclassing, widget registration, or draw overrides are required

### 5.7 Advantages of the Render Model

| **Feature** | **Description** |
|---|---|
| Statelessness | No retained render trees; everything is computed per frame |
| Composability | Any entity with visuals can be layered, masked, or styled |
| Backend Independence | Easy to swap or extend rendering backends |
| Determinism | Consistent visual output through strict draw ordering |

This model supports predictable, high-performance rendering across platforms without sacrificing layout precision or styling flexibility

## 6. Conclusion

This framework illustrates a modular, data-driven approach to UI development built around the principles of Entity Component Systems and reactive programming. Rather than relying on traditional widget hierarchies or subclass-based frameworks, the proposed architecture treats the UI as a flat, declarative data model processed by stateless systems.

The advantages of this model include:

- Modularity: Behavior and layout are decoupled and composable.
- Performance: Linear traversal, sparse data access, and minimal indirection.
- Flexibility: Easily supports both static and dynamic UI trees.
- Interoperability: Clean separation of UI definition and runtime execution.

By treating UI state as pure data and processing it through clearly defined systems—layout, input, rendering—this design supports a wide range of

applications, from lightweight embedded tools to complex editors.

While the implementation is based on C#, the concepts generalize across platforms and languages. Declarative UI definitions can be parsed from external formats or composed programmatically, and rendered using a pluggable backend of choice.

Ultimately, the strength of this model lies in its simplicity: each subsystem does one thing, with clear boundaries, and all behavior emerges from data