

ReaCS and the VSC Architecture Paradigm: A Modern Reactive Framework for Unity

Chapter 1: Introduction

What is ReaCS & VSC?

ReaCS is not just a toolkit — it's a **paradigm** for building Unity applications through clear, reactive, and scalable architecture. Its core philosophy is rooted in **VSC: View – State – Controller**.

Unlike traditional Unity projects that entangle behavior in GameObjects, MonoBehaviours, and static references, ReaCS defines everything — from UI to gameplay systems — as a **reactive graph of data and relationships**. The View observes State. State is defined declaratively. Controllers (Systems) react automatically to changes. This design results in projects that are easier to understand, debug, extend, and optimize.

VSC is not a variant of MVC or MVVM. It is a fully reactive architecture that embraces Unity's strengths — ScriptableObjects, scene-based workflows, the UI Toolkit, and component-driven design — while avoiding its common architectural pitfalls.

In short, ReaCS empowers you to build Unity projects where:

- All data flows are explicit and observable
- Logic is reactive, not imperative
- Relationships are modeled, not nested
- The UI reflects the data, not the code

Why ReaCS Exists

Unity's built-in architecture doesn't scale. It encourages the use of:

- MonoBehaviours for logic and state
- Lists to store relationships
- Serialized references and GameObject links
- Event buses or update loops for communication

This leads to:

- ❌ Hidden dependencies
- ❌ Tight coupling between view, state, and logic
- ❌ Difficult-to-test gameplay code
- ❌ Performance bottlenecks from linear lookups or redundant updates

As projects grow, these problems compound. Features become brittle, updates break other systems, and optimization becomes a guessing game.

ReaCS was built to solve these issues from the root: by treating **state and structure as first-class data** and by making **systems purely reactive**. It removes glue code, unifies access to logic and relationships, and creates a consistent way of thinking about every piece of your game or app.

What ReaCS Solves

ReaCS doesn't just clean up your data — it transforms the way your architecture works.

With ReaCS:

- There are **no hardcoded relationships or singleton managers**
- There are **no lists of children or items embedded in a ScriptableObject**
- There are **no polling Update() methods or custom event buses**
- Systems are not services — they are **pure reactive observers**
- UI doesn't need ViewModels — it binds **directly to the state (with converters if needed like for string localization)**
- Logic becomes declarative, predictable, and testable

Your game becomes a **reactive graph**:

- When an experience is selected, the data changes
- That change propagates through reactive links
- Systems that observe that data apply behavior
- The UI reflects the updated values automatically






Every behavior can be traced.

Every change can be visualized.

Every system does only one thing — and does it reactively.

Preview of the Building Blocks

While the implementation details are covered in later chapters, here's a quick overview of how ReaCS works under the hood:

-  **State** lives in `Observable<T>` fields within `ObservableScriptableObject` instances.
 -  **Logic** lives in `SystemBase<T>`, and reacts declaratively to changes in fields using `[ReactTo]`.
 -  **Relationships** are modeled explicitly using `LinkSO<TOwner, TTarget>` or semantically named subclasses like `BuffAssignmentSO` or `TagAssignmentSO`.
 -  **Queries** retrieve structured data using shared, lazy-loaded accessors via `Query<T>()` — but never modify anything directly.
 -  **Performance** is backed by `ReaCSIndexRegistry`, which allows burstable, cache-safe, O(1) relationship lookups.
-

Why VSC is Not MVC or MVVM

VSC may look similar on the surface — but it is fundamentally different in purpose, scale, and structure:

Feature	MVC	MVVM	VSC (ReaCS)
View → State	✗ Indirect via controller	✓ via binding layer	✓ via <code>Observable<T></code>
State is centralized	✗ No	⚠ Duplicated	✓ Always in SOs
Logic location	✓ Controller code	⚠ Often leaks into ViewModel	✓ Stateless systems
Data relationships	✗ Hidden / code-based	⚠ Code + ViewModel logic	✓ Modeled declaratively
UI Integration	✗ Manual	⚠ Glue-heavy	✓ Bound through native reactivity
Scales well in Unity	✗ No	⚠ Not really	✓ Yes

MVC and MVVM bring too much ceremony and too little clarity to Unity.

They don't map cleanly to:

- Prefabs
- Scene lifecycles
- ScriptableObject workflows
- Visual tooling and reactivity

VSC, by contrast, was made for Unity.

Are ReaCS Systems Really "Controllers"?

While we refer to the logic layer in ReaCS as the "Controller" in VSC, it's important to understand that **ReaCS Systems are not traditional controllers in the MVC sense.**

In Traditional MVC:

- A **Controller** is an imperative object that responds to input (e.g. UI events, service calls).
- It typically contains business logic and directly mutates models or views.
- Controllers are often stateful, procedural, and context-driven.

In ReaCS:

- A **System** (via `SystemBase<T>`) is **stateless**, **reactive**, and **pure**.
- It only runs in response to observable field changes.
- It performs no orchestration, owns no session, and reacts only to **declared data**.
- All behavior is driven by **data diffs**, not by external events or triggers.

ReaCS Systems are better described as System-Controllers — they fulfill the role of the Controller (applying logic), but behave more like reactive rules

| than imperative code blocks.

This difference is what makes ReaCS logic predictable, testable, and scalable — and why the **“C” in VSC should be interpreted differently in the context of ReaCS.**

What This Document Covers

This guide is a deep dive into how to architect your entire Unity app or game using the ReaCS VSC paradigm.

You’ll learn:

- How to model **relationships without lists**
- How to write **pure systems that scale and test easily**
- How to bind UI **directly to live data**
- How to avoid **hidden logic, tangled updates, and fragile architecture**
- How to optimize for **performance with burst-safe queries**
- How to visualize, trace, and debug your **data-driven logic**

You don’t have to rewire Unity to do this.

ReaCS lets you work with Unity — not against it — while unlocking architecture that’s worthy of production-scale systems.



Chapter 2: Traditional Unity Architecture — Why It Falls Short

1. How Unity Encourages a Component-Based Architecture

Unity's design philosophy is centered on **component-based architecture** — and while this allows for quick iteration and accessible composition, it also leads to **accidental architecture**.

Most Unity developers start with MonoBehaviours as both their state and logic containers. They serialize data into public fields, wire references in the Inspector, and trigger behavior through event handlers, `Update()`, or `UnityEvents`.

This design appears harmless in small projects. But as features scale, teams grow, and systems become interdependent, this flexibility becomes a trap:

- There’s no architectural contract or visibility.
- Logic becomes entangled across disparate components.
- Debugging behavior becomes guesswork.

You don’t build a system — you accumulate side effects.



Sidebar: “It Worked Until It Didn’t”

A Unity developer prototype might start like this:

- Attach a script to an object

- Drag references into it
- Make changes in the Inspector
- Use `Update()` to apply some logic

It works.

Then it grows:

- A “Manager” is added to initialize things.
- The UI starts modifying state directly.
- Event handlers are wired into multiple scripts.
- Another developer adds their own polling logic to fix a bug.

Suddenly:

- One small change breaks unrelated features.
- Frame ordering becomes critical.
- Reinitializing a prefab in a new scene breaks state restoration.
- No one knows who sets the data — or when.

This is architecture by accident — and Unity makes it very easy to fall into.

2. The Problems with Lists as Relationships

One of the most common Unity patterns is:

```
public List<BuffSO> activeBuffs;
```

At first glance, this looks like a simple way to say:

“This object has these buffs.”

But this list is **an architectural black box**.

✗ Why Lists Break Down:

- **You can’t observe list membership.**

No event is fired when something is added or removed.

- **No metadata per entry.**

You might want to know:

- Who applied the buff?
- How long is it active?
- What its source was?

In Unity, this is only possible by:

- Creating custom wrapper classes
- Instantiating per-buff state containers
- Using inheritance-heavy models

- And even then, the data is often spread across GameObjects or non-reusable structs.

- **No indexing.**

Want to know if a buff of type "Stun" is active?

You iterate through the list every time.

- **The list has no semantics.**

It doesn't express *why* these objects are related — just that they're grouped.

- **No traceability.**

You can't debug "who added this" or "why it's still here."

✗ And most dangerously: List logic is repeated across MonoBehaviours.

You'll find things like:

```
foreach (var buff in activeBuffs)
    buff.Apply(monsterStats);
```

or

```
if (inventory.Contains(item)) { ... }
```

This leads to **non-DRY code** spread across different objects and systems.

Behavior becomes inconsistent, brittle, and difficult to test.

3. The Tangle of Update-Driven Logic

Unity's default scripting pattern is **imperative polling**.

Every script has some flavor of:

```
csharp
CopyEdit
void Update() {
    if (someState) { doSomething(); }
}
```

This leads to:

- Dozens of polling loops across unrelated components
- Inter-frame race conditions ("Why is it still false this frame?")
- Hidden order dependencies (some logic relies on what another script did earlier that frame)

The behavior graph is invisible.

A state change may cascade into five Update calls across five objects — and there's no way to trace the origin.

 **Key insight:**

Polling logic works **despite Unity**, not because of it.




It's a workaround for the lack of observable, declarative state.

4. Hidden Dependencies and Black Box Behavior

Unity encourages developers to wire up their references in the Inspector or resolve them at runtime using:

- `FindObjectOfType<T>()`
- `GetComponentInChildren<T>()`
- Serialized fields hidden in the prefab hierarchy

This results in:

-  Hidden relationships
-  Runtime-only discovery
-  Tight coupling to scene context and object presence

And because **logic and state are interwoven**, breaking one link in this chain silently disables features.

"Why did disabling this script break everything?"









That script was holding a reference to a controller that mutates your state.

But you didn't know, because the reference was invisible.

There was no signal that this object was critical to game logic.

But What About ReaCS?

ReaCS **still requires references** — but they are:

Traditional Unity	ReaCS
 Hidden in serialized fields	 Declared in public fields on systems
 Point to GameObjects or Components	 Point to ScriptableObjects (data only)
 Resolved at runtime (fragile)	 Wired in editor or via queries (explicit)
 Used anywhere in the logic	 Used only in reactive, single-purpose systems

Example:

```
csharp
CopyEdit
[ReactTo(nameof(ExperienceSO.isSelected))]
public class SelectModelSystem : SystemBase<ExperienceSO>
{
    public ModelSO model;

    protected override void OnFieldChanged(ExperienceSO experience)
    {
        model.color.Value = Color.green;
    }
}
```

```
}  
}
```

Yes — this system holds a reference to the model.

But:

- It's **explicitly declared**.
- It's **purpose-bound** (the system does one thing).
- It operates on **observable state**.
- It can be **replaced with a query** later (`Query<ModelPartQueries>()`) to get all of these ScriptableObjects

ReaCS doesn't eliminate coupling — it eliminates implicit coupling.

5. Fragility and Scalability Failures

When state and behavior live inside the same GameObject, you create fragile systems.

- Changing a field in one prefab might break the logic of an unrelated system.
- Extending functionality often requires duplicating components and wiring them manually.
- Removing a GameObject with logic attached may remove critical gameplay features.

There's no **shared contract** between systems.

No central place to reason about how state drives behavior.

As the project grows, **new features conflict with old ones**, and everyone starts asking:

- "What happens if I change this?"
- "Where does this get used?"
- "Who's responsible for this flow?"

This isn't a tech debt problem — it's a **structure debt** problem.

! Common Pain Points Table

Problem	Cause	Result
List-of-doom	Bufs, Inventory, Steps	Non-observable blobs
Hidden references	Manual drag & drop	Breaks silently
Magic constants	Hardcoded damage values	Not testable
GameObject logic	Mixed logic/state	Can't reuse or inspect
Update spam	Frame polling	Performance hit, untraceable

6. How Traditional Patterns Try to Fix It (and Fail)

Many Unity devs recognize these issues and try to apply known patterns:

- **MVC:** A “Manager” becomes a God object that owns all references.
- **MVVM:** UI logic gets duplicated in ViewModels, introducing binding layers and stale state.

But these solutions **don’t align with Unity’s engine model**:

- You still need to wire things manually in scenes.
- State is still tied to object lifetimes.
- You’ve just **moved the mess** — not removed it.

7. The Emotional Cost of Traditional Unity Architecture

This is the unspoken cost in most Unity projects:

Thought	Emotion
“Touching this will break something.”	😰 Fear
“Why is this happening?”	😕 Confusion
“Who owns this logic?”	😟 Uncertainty
“Where do I put this new feature?”	😓 Overwhelm
“I know it’s wrong, but I don’t have time to fix it.”	😞 Resignation

The result:

- Junior devs afraid to contribute
- Senior devs burned out
- Project velocity grinding to a halt

8. The Need for a Better Foundation

Unity’s component-based system enables quick iteration — but not safe scale.

To solve this:

- You can’t just enforce conventions
- You need a new **foundation** — one that’s:
 - Declarative
 - Reactive
 - Observable
 - Toolable
 - Performance-aware

That’s what **ReaCS and the VSC architecture paradigm** offer:

| A modern, reactive framework designed to work with Unity — not around it.

Next, we’ll explore **how ReaCS structures your entire application** using View, State, and Controller (System) — and why that shift changes everything.

Chapter 3: The VSC Architecture — View, State, and Controller in ReaCS

The heart of ReaCS is its architecture: **VSC** — **View**, **State**, and **Controller**. It's not just a framework convention — it's a declaration of how your Unity project should think.

This chapter breaks down each layer, shows how they interact, and explains how ReaCS transforms the default Unity development flow into a fully reactive, testable, and scalable system.

VSC is not derived from MVC or MVVM.

It is inspired by **reactive web frameworks** and **Entity-Component-System (ECS)** design — adapted to Unity's scene, prefab, and ScriptableObject workflow.

1. What is VSC (View – State – Controller)?

VSC is a clean separation of responsibilities across three orthogonal layers:

Layer	Responsibility	In ReaCS
View	Display state, route intent	UI Toolkit, scene UI
State	The single source of truth	<code>ObservableScriptableObject</code> , <code>Observable<T></code>
Controller	Apply behavior in response to state	<code>SystemBase<T></code>

Each layer is decoupled, but interconnected **through observable data** — not through function calls, events, or direct references.






This is a **reactive data flow**, not a procedural one.

2. The View Layer — A Reflection of State

In ReaCS, the **View** is any UI or 3D element that **binds to observable data**.

- Built using Unity UI Toolkit (or scene UI components)
- Reads directly from `Observable<T>` fields (e.g. `.text = someData.title.Value`)
- Forwards user intent via simple state changes (e.g. toggling `isSelected.Value = true`)

Key Properties

Principle	ReaCS View Layer
No logic	 No conditionals or branching inside UI handlers
No ownership	 Views don't own data or behavior
Pure display + triggers	 Reads and writes data only
Reactive UI binding	 Updates when state changes
Stateless when possible	 Can be instantiated, reset, reused

The Rule of Thumb:

Views don't "do" — they "reflect."

Even when a button causes a scene change, it doesn't invoke logic — it sets a field (e.g. `SceneRequestSO.targetScene.Value = "Menu"`) and a system reacts to it.

3. The State Layer — Observable, Structured, Declarative

In ReaCS, **State is everything**.

It lives in:

- `ObservableScriptableObject` (SOs that store data)
- `Observable<T>` fields (type-safe, reactive, serializable)
- `LinkSO<TOwner, TTarget>` (modeled relationships)

There are **no nested lists, embedded children, or serialized reference chains**.

Everything is flat, observable, and declared.

✓ Why "State" — Not "Model"

Unlike the "Model" in MVC or MVVM, ReaCS **State** is:

- Pure data (no logic)
- Reactive and observable
- Persistent across scene loads
- Indexed and shared
- Designed to be declarative — not instantiated per user/session

✓ Sidebar: ReaCS State vs MVC Model

Feature	MVC Model	ReaCS State
Holds data	✓	✓
Holds logic	✓ (often)	✗
Reactive	✗	✓
UI-bound	✗	✓
Instantiated per session	✓	✗ (shared globally)
Indexed	✗	✓
Declarative	⚠ Sometimes	✓

ReaCS state is not a DTO or a class — it's live truth, observable and connected to all systems and views.

♻ Pooling ObservableScriptableObjects (Advanced Use Case)




Just like `LinkSO` instances that we'll explore next and in-depth in the next chapter, **any** `ObservableScriptableObject` can be pooled and reused — especially in high-performance or short-lived

scenarios.





This is most useful when:

- You need to instantiate **temporary state** rapidly (e.g., popup configs, ghost UI, transient reactions)
- You want to reduce GC allocations from repeated SO creation
- You want to maintain full **reactivity and observability**, even for objects that only live briefly

By pooling `ObservableScriptableObject` instances:

-  You maintain **field-level observability** (Observers still track reactive changes)
-  You avoid expensive `ScriptableObject` allocations
-  You keep logic **fully compatible with ReaCS systems and UI bindings**

When to Use

Use Case	Pool It?
Long-lived global state	 Keep persistent
Instantiated per interaction or frame	 Pool
Used in prefab overlays / popups	 Pool
Created dynamically for simulations	 Pool

`ObservableScriptableObjects` are just data objects — and in ReaCS, even your state system can be made recyclable.

 **ReaCS does this for you automatically! Any `ObservableScriptableObject` created at runtime automatically sets up its own Pool and Gets and Release OnEnable & OnDestroy**

Relationships via `LinkSO<TOwner, TTarget>`

Instead of:

```
public List<ExperienceSO> experiences;
```

ReaCS models:

```
public class ExperienceListItemSO : LinkSO<ExperienceGroupSO, ExperienceSO>
{
    [Observable] public Observable<int> order;
    [Observable] public Observable<bool> isFeatured;
}
```

Benefits

Old Way (List)	ReaCS Way (LinkSO)
Opaque	Modeled and observable
No metadata	Add flags, timestamps, notes

Not reactive	Triggers systems when changed
Manual access	Indexable and queryable
Hard to debug	Visualizable in graph

Terminology Note:

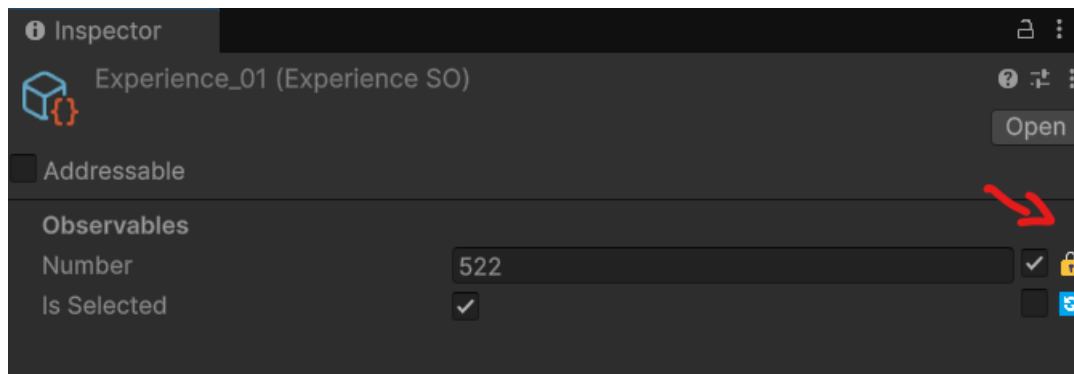
We often use `TOwner` and `TTarget` to express directionality.



In some symmetrical relationships, neutral names like `TLeft`, `TRight` may still apply — but explicit naming helps clarify logic most of the time.

Built-in Persistence with `ShouldPersist`

ReaCS gives developers effortless control over what data persists between sessions.

Every `Observable<T>` field in a `ObservableScriptableObject` comes with a built-in `ShouldPersist` flag — exposed directly in the Unity Inspector as a lock icon or checkbox.



Feature	Behavior
 Checked (<code>ShouldPersist = true</code>)	Field value is automatically serialized and restored across playthroughs
 Unchecked (<code>ShouldPersist = false</code>)	Field resets to its default value every time the app quits or playmode stops

This means:

- You don't need to write custom serialization logic
- You don't need to track save/load manually
- You can control **persistence per field**, not per object

Under the hood, ReaCS handles:

- Saving values to disk
- Restoring them on load
- Clearing them when reset is desired

This makes state management not only declarative, but persistent-aware — with zero boilerplate and full granularity.

Use it for:

- Save data
- Editor state restoration
- Resettable gameplay variables
- Runtime-only debug fields

4. The Controller Layer — Systems as Reactive Logic

In ReaCS, the **Controller** layer is implemented by `SystemBase<T>` — but these are **not traditional controllers**.

✅ ReaCS Systems are stateless, reactive functions that respond to state changes.

They are declared like this:

```
[ReactTo(nameof(ExperienceSO.isSelected))]  
public class HighlightModelPartsSystem : SystemBase<ExperienceSO>  
{  
    public ModelSO model;  
  
    protected override void OnFieldChanged(ExperienceSO changedSO)  
    {  
        model.highlighted.Value = changedSO.isSelected.Value;  
    }  
}
```

✅ Properties of Systems

Property	ReaCS System
Trigger	Field change on a declared SO
State	Stateless (or declarative cache only)
Reusability	High
Testability	High
Lifetime	Determined by scene presence (like MBs)
Connection	Indirect via observable state

⚠️ Note:

ReaCS systems don't use `Update()` in 98% of cases.

For simulations (e.g. physics, timers), `Update()` may be used inside a system — but this is always localized and deliberate.

📦 Sidebar: MVC Controllers vs ReaCS Systems

Feature	MVC Controller	ReaCS System
Imperative	✓	✗
Reactivity	✗	✓
Orchestrates flow	✓	✗
Stateless	✗	✓
Bound to data	✗ (often procedural)	✓ (observes fields)
Testable	⚠	✓

ReaCS systems do not coordinate — they **respond**.

They are not services — they are **field-bound observers**.

BurstSystemBase

ReaCS now supports Burst-compiled parallel jobs using native SO data:

```
public class VelocitySystem : BurstSystemBase<VelocitySO>
{
    protected override JobHandle OnUpdate(JobHandle inputDeps)
    {
        var velocities = Query<UnifiedObservableRegistry>()
            .BuildNativeLookup<VelocitySO, float2>(v => v.Value, Allocator.TempJob);
        ...
    }
}
```

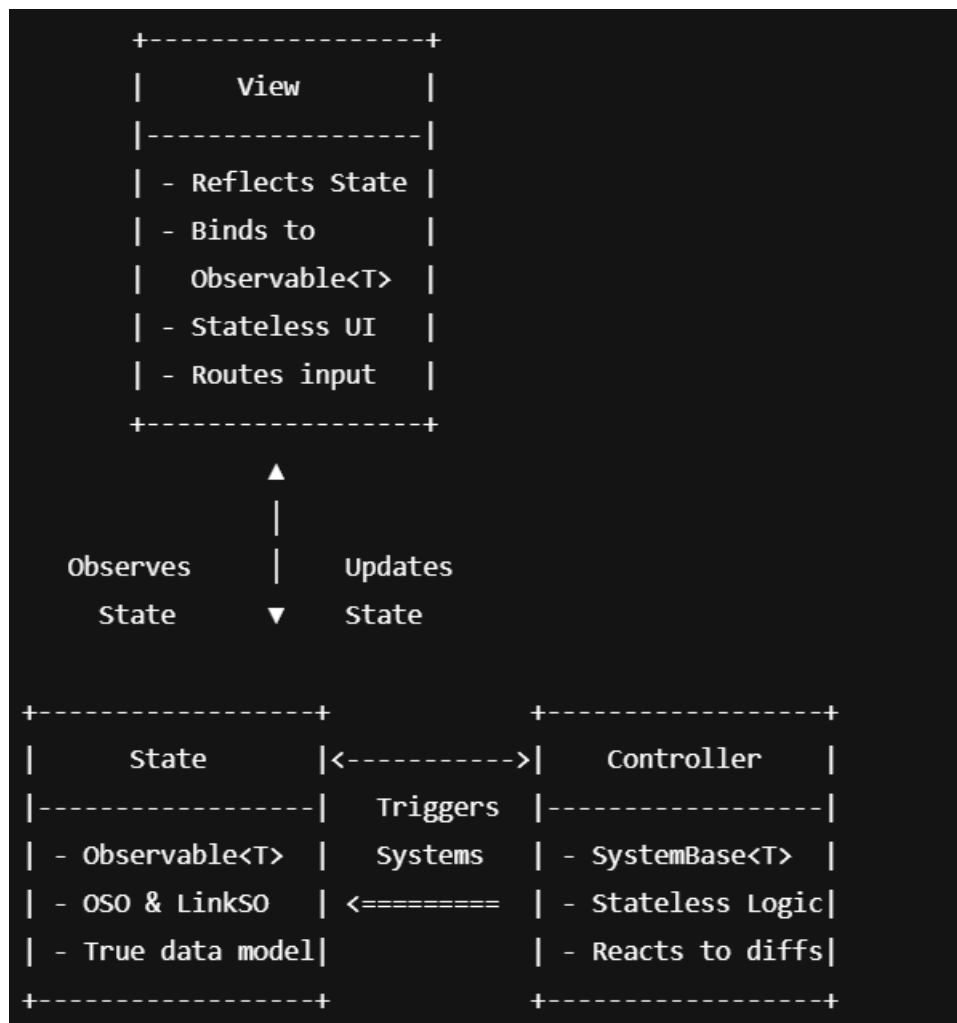
For non-burst systems that read or write Unity components, use regular `SystemBase<TSO>` and fetch Unity objects via the binding lookup.

5. How the Layers Interact in Practice

Imagine:

- A button in the View toggles `ExperienceSO.isSelected = true`
- A System reacts to this and sets `ModelSO.highlighted = true`
- Another System reacts to that and shows a UI drawer
- The UI reflects that drawer using bound state

Data Flow in VSC



There is **no direct function call** between layers.

Everything flows through **observable data**.

6. Why VSC Avoids MVC & MVVM Pitfalls

Concern	MVC	MVVM	VSC (ReaCS)
State duplication	✗	⚠ ViewModel copy	✓ Centralized observable
Logic leakage	✓	⚠ ViewModels	✓ Pure systems only
Coupling to UI	✓	⚠ Bindings	✓ Uncoupled
Tooling	✗	⚠ Glue code	✓ Visual Graph
Unity fit	✗	⚠ Extra layers	✓ Native SOs + Systems

ReaCS doesn't re-implement MVC.

It **replaces it with something native to reactive game dev**.

7. How to Structure Projects in VSC

Layer	Guidance
View	Bind directly to <code>Observable<T></code> , no logic
State	Keep all runtime state in SOs, one field = one truth
Controller (System)	Observe declared fields only, use no global state

- ✓ No poll-based logic
- ✓ No black box dependencies
- ✓ No update spaghetti

If you need to affect another SO — **model it** via `LinkSO`.

If you need to find an SO — **query it**, don't drag it.

This is how you build Unity projects that scale — clearly, reactively, and fearlessly.

Chapter 4: Modeling Relationships — No Lists, Only Links

In most Unity projects, **relationships are implemented as lists**. A `Monster` has a `List<BuffSO>`. A `Quest` has a `List<StepSO>`. A `Group` has a `List<ExperienceSO>`. This approach is familiar, intuitive — and fundamentally flawed.

In ReaCS, relationships are not embedded.

They are modeled explicitly as their own first-class data.

This chapter explains why ReaCS **never uses lists** for state relationships, how to model those links with `LinkSO<TOwner, TTarget>`, and how this change empowers tooling, debugging, scaling, and clarity.

1. The Problem with Lists

Let's say you start with:

```
public List<BuffSO> activeBuffs;
```

This list is meant to represent “which buffs this monster has.”

But the problems start immediately:

✗ Not Observable

You can't observe:

- When a buff is added
- When one is removed
- When the list changes

You'd have to write a wrapper like `ObservableList<T>` — and even that doesn't solve deeper issues.

✗ No Metadata

What if you need:

- Duration?
- Source of the buff?
- Is it temporary?
- How many stacks?

You can't attach metadata to entries in a list unless you:

- Create a wrapper class (`BuffInstance`) with all that data
- Instantiate and manage it yourself
- Maintain references from that back to the owner

This adds:

- Complexity
- Memory management issues
- More inheritance and boilerplate

✗ Not Indexable

Want to know:

“What buffs are currently applied to this monster that came from fire spells and last more than 5 seconds?”

Now you're filtering a `List<BuffInstance>` with LINQ every frame.

This works for 10 buffs — not a 1000.

✗ Hidden Structure

A list says:

“This monster owns these buffs.”

But that's often **semantically wrong**.

In reality:

- The relationship might be **temporal**
- Might be **reversible** (e.g. buffs know their targets)
- Might be **shared** (multiple monsters share a buff)

Lists **imply ownership and nesting**, which breaks down with complex systems.

✗ Violates DRY

Every system that interacts with a list does this:

```
foreach (var buff in activeBuffs)
{
    if (buff.type == BuffType.Freeze) ...
}
```

This logic is duplicated across:

- Combat system
- UI
- Cleanup
- Save system

And when that list changes shape or schema?

You update it everywhere.

2. What to Use Instead: LinkSO<TLeft, TRight>

In ReaCS, relationships are represented as actual **data objects**:

```
public class BuffAssignmentSO : LinkSO<MonsterSO, BuffSO>
{
    [Observable] public Observable<float> duration;
    [Observable] public Observable<bool> isActive;
}
```

This models the relationship as:

“This buff is assigned to this monster, with metadata.”

Each link:

- Lives in memory as a real ScriptableObject
- Is observable
- Has its own lifetime
- Can carry state

Drag & Drop Comparison

Concept	Traditional List	ReaCS LinkSO
“Buff is active on monster”	List<BuffSO>	BuffAssignmentSO : LinkSO<Monster, Buff>
“Quest has steps”	List<StepSO>	QuestStepAssignmentSO : LinkSO<QuestSO, StepSO>
“Model belongs to group”	List<ModelSO>	ModelGroupAssignmentSO : LinkSO<GroupSO, ModelSO>

In all cases:

- Links are actual SOs
- Systems can observe them

- They're debuggable and queryable

3. One-to-One, One-to-Many, Many-to-Many

ReaCS can model **any cardinality** through links.

✅ One-to-One

Just a single `LinkSO<A, B>` between two known instances.

Example:

```
ExperienceSO → ModelSO
```

Use case:

- Only one model per experience
- One-to-one UI screens

✅ One-to-Many

Multiple links from a single `TOwner`.

Example:

```
GroupSO → ExperienceSO
```

- Each `ExperienceListItemSO : LinkSO<GroupSO, ExperienceSO>`
- Order and visibility flags stored per link

✅ Many-to-Many

Links where:

- Multiple sources point to the same target
- Targets may be used by multiple owners

Example:

```
MonsterSO → TagSO
```

- Any monster can have multiple tags
- Any tag can apply to multiple monsters
- Each `TagAssignmentSO` can hold importance, source, duration

🧠 Semantic Reminder:

“TOwner” and “TTarget” describe direction — not logic.

The *meaning* comes from the subclass name.

4. AssignmentSO: When a Relationship Has State

The most powerful form of `LinkSO` is an **assignment**.

When a relationship has its **own logic**, it becomes a reactive object:

```
public class TagAssignmentSO : LinkSO<MonsterSO, TagSO>
{
    [Observable] public Observable<float> confidence;
    [Observable] public Observable<string> sourceSystem;
}
```

Now you can:

- React when a tag's confidence crosses a threshold
- Debug why a tag is assigned
- Serialize link metadata independently

Systems can observe the link itself — not just the entities.

Design Principle:

“If a relationship changes over time or needs explanation — make it a first-class data object.”

5. Performance: The Indexing Advantage

All `LinkSO` instances are automatically indexed in `ReaCSIndexRegistry`.

You can write:

```
Query<MonsterSO, BuffAssignmentSO>(monster)
```

This will return:

- All `BuffAssignmentSO` where `.owner == monster`

And:

```
Query<BuffSO, BuffAssignmentSO>(buff)
```

Will return all monsters that have that buff.

Why It's Fast

- ReaCS maintains in-memory maps per `LinkSO` type
- All queries are cached and burst-safe
- You don't scan, you look up

Traditional	ReaCS
<code>foreach</code> loop with conditions	✓ Constant-time index
Runtime GetComponent	✓ Pure SO access
Multiple passes across different lists	✓ One query, one filter

This scales to 1,000s of relationships — **no performance drop**.

LinkSOs Support Pooling for Performance and Reuse

One of the key advantages of modeling relationships as `LinkSO<TOwner, TTarget>` instances is that they are **fully self-contained objects** — which means they can be **pooled and reused** at runtime.

In high-frequency systems like:

- Buff applications
- Temporary tags
- Interaction effects
- Highlighting or selection

...it would be inefficient to constantly `CreateInstance` and `Destroy` LinkSOs. Instead, ReaCS supports **runtime pooling** of link objects:

- A `ReaCSLinkPool<T>` can allocate and reset links on demand.
- Fields like `owner`, `target`, and metadata are cleared and reassigned safely.
- Pools are automatically injected and lazily instantiated (no bootstrapping required).

This ensures:

- ⚡ **Low-GC, high-performance object reuse**
- 🧩 **State remains reactive** (Observable fields are reused, not replaced)
- 🧼 **Memory stays clean** (no dangling links or leaking SOs)

Unlike lists, which you must constantly resize and scan, LinkSOs behave like ECS components: reactive, targeted, and recyclable.

✓ ReaCS does this for you automatically! Any `LinkSO` created at runtime automatically sets up its own Pool and Gets and Release OnEnable & OnDestroy

6. Visual Debugging: Why Links Empower Tooling

Each `LinkSO`:

- Shows up in the **ReaCSGraphView**
- Can animate when changed
- Can display metadata fields
- Can show system reactions

This means you can:

- Visually trace who added a relationship

- See its lifetime
- See who uses it
- Watch the causality chain unfold in real time

Lists can't do that.

7. Naming Guidelines for LinkSOs and Assignments

Purpose	Name Format	Example
Pure structural link	<code><Source>To<Target>LinkSO</code>	<code>ExperienceToModelLinkSO</code>
Link with logic/state	<code><X>AssignmentSO</code>	<code>BuffAssignmentSO</code>
Interaction-specific	<code><Verb>LinkSO</code> or <code>InteractionSO</code>	<code>HighlightLinkSO</code> , <code>SelectionAssignmentSO</code>

If it has metadata — use AssignmentSO.

If it's a structural or static link — use `LinkSO`.

8. Relationship Modeling Recap

Do this	Why
✓ Use <code>LinkSO<TOwner, TTarget></code>	Relationship is observable and traceable
✓ Use <code>AssignmentSO</code> when you need metadata	Bufs, tags, items, selections
✓ Query all links via index	Fast, burstable
✓ Avoid embedded lists	No reactivity, no structure
✓ Visualize links	Tools can trace logic easily
✓ Use field-level observables	Systems can react per-link
✓ Name links semantically	Improves discoverability

In ReaCS, a relationship is not structure — it's data.

And like all data, it deserves to be observable, persistent, debuggable, and performant.

BONUS SECTION

Why Relationships Matter

In traditional Unity setups, object relationships are often encoded in fields like `List<T>` or references between GameObjects and ScriptableObjects. But this structure makes relationships:

- 🔍 Hard to query (who owns what?)
- 🔄 Hard to react to (what changes when something links or unlinks?)
- 🪄 Hard to test or simulate (you need the whole list hierarchy in place)
- ♻️ Hard to reuse and pool

ReaCS solves this by introducing **LinkSO<TLeft, TRight>**, a lightweight, reactive object that models a single relationship between two entities.



LinkSO<TLeft, TRight> — The Relationship as Data

Instead of saying "A contains B", ReaCS says: "There exists a link between A and B".

```
public class TagAssignmentSO : LinkSO<TagSO, EntitySO> { }
```

This represents: "EntitySO is tagged with TagSO".

It enables:

- Fast reverse lookups (which tags does this entity have? which entities have this tag?)
- Reactive systems (if a tag is added or removed, trigger a reaction)
- Clean unidirectional modeling — no nesting required.



Examples



Instead of:

```
public class GroupSO : ScriptableObject {  
    public List<ExperienceSO> experiences;  
}
```



Use:

```
public class ExperienceGroupAssignmentSO : LinkSO<GroupSO, ExperienceSO> { }
```

Then query all experiences linked to a group via:

```
var group = selectedGroupSO;  
var experiences = Query<ReaCSIndexRegistry>()  
    .RightOf<ExperienceGroupAssignmentSO, GroupSO, ExperienceSO>(group);
```

Or query all groups an experience belongs to (for tagging, collections, etc.):

```
var groups = Query<ReaCSIndexRegistry>()  
    .LeftOf<ExperienceGroupAssignmentSO, GroupSO, ExperienceSO>(experience);
```



Pooling LinkSOs at Runtime

Each link can be reused rather than created/destroyed each time. Use **ReaCSLinkPool<TLinkSO>**:


```
var pool = Use<ReaCSLinkPool<ExperienceGroupAssignmentSO>>();
var link = pool.Get();
link.Left.Value = group;
link.Right.Value = experience;
```

When the link is no longer needed:

```
pool.Release(link);
```

Pooling enables:

- 🔄 Reduced GC allocations
- ⚡ Runtime efficiency for frequent relationship updates
- 🧹 Cleaner lifecycle management

All links are auto-registered to `ReaCSIndexRegistry` on creation and unregistered on release.

🧠 Reacting to Relationship Changes

You can react to links being formed or broken by listening to changes on their `Left` or `Right` values:

```
[ReactTo(typeof(ExperienceGroupAssignmentSO))]
public class UpdateGroupUI : SystemBase<ExperienceGroupAssignmentSO>
{
    public override void OnChanged(ExperienceGroupAssignmentSO link, string field)
    {
        if (field == nameof(link.Left) || field == nameof(link.Right))
        {
            // Update UI, refresh query, etc.
        }
    }
}
```

📖 Use Cases

Use Case	LinkSO Example
Tagging	<code>TagAssignmentSO : LinkSO<TagSO, EntitySO></code>
Grouping	<code>ExperienceGroupAssignmentSO : LinkSO<GroupSO, ExperienceSO></code>
Ownership	<code>PartOwnershipSO : LinkSO<EntitySO, PartSO></code>
Selection	<code>SelectedObjectSO : LinkSO<UIContextSO, TargetSO></code>
Targets	<code>AITargetSO : LinkSO<EnemySO, PlayerSO></code>

✅ Benefits of Using LinkSO

- ✓ Fully observable: Can be reacted to like any other `ObservableScriptableObject`

- ✓ Fully indexable: Instantly queryable by either side
- ✓ Flat structure: No nesting, easier to debug and manage
- ✓ Pooled and efficient: Runtime-friendly
- ✓ Editor-friendly: Can be visualized and inspected like any other SO

✓ Summary

Every "contains", "owns", "tags", or "targets" relation in your app is a perfect candidate for a LinkSO.

Model your data flatly. Query by side. React to changes. Pool when needed.

Chapter 5: Queries and Shared Access — Fast, DRY, and Toolable

In ReaCS, you don't inject dependencies.

You don't wire objects together in a service graph.

You don't drag-and-drop ScriptableObjects into every system.

Instead, you access everything — data relationships, helper logic, and pooled objects — through two clean, consistent, and zero-overhead patterns:

```
var result = Query<T>();  
var helper = Use<T>();
```

This chapter explains how those two tools form the **foundation of shared logic and access** in the VSC architecture, and why they're essential for performance, clarity, and reusability.

1. Why Shared Access Matters

Systems often need to:

- Look up relationships: "Which model belongs to this experience?"
- Reuse logic: "Format this model part name"
- Allocate pooled objects: "Create a temporary BuffAssignmentSO"

Without shared access patterns, you'd end up with:

- Repeated logic scattered across systems
- Drag-and-drop references into every system MonoBehaviour
- Hardcoded wiring that's impossible to test or reuse

ReaCS solves this cleanly — without DI — through lazy, reusable stateless registries.

What Are Queries?

In ReaCS, **Queries** are the clean, fast, and composable way to retrieve data or helpers — no need to drag-and-drop references, set up Singletons, or inject services.

Instead of relying on MonoBehaviour wiring or clunky lookups, you simply declare:

```
var query = Query<MyCustomQuery>();  
var results = query.DoSomethingUseful();
```

Or for core registries:

```
var index = Query<ReaCSIndexRegistry>();  
var items = index.RightOf<LinkType, LeftType, RightType>(someLeft);
```

How It Works

Use<T>()

Registers a query, helper, or pool — like installing a tool.

Query<T>()

Accesses the globally shared instance.

Both `Use<T>()` and `Query<T>()` are zero-boilerplate:

```
Use<MyHelper>(); // Register once  
var helper = Query<MyHelper>(); // Access anywhere
```

Under the hood, these are just static lazy singletons — **not** Unity services or MonoBehaviours.

Common Queryable Tools

Type	Purpose
<code>ReaCSIndexRegistry</code>	Query by relationships (e.g., which experiences are in a group?)
<code>ReaCSLinkPool<T></code>	Reuse links instead of creating new ones every time
<code>EntityRegistry<T></code>	Keep track of MonoBehaviours that bind to a T-type SO
<code>ComponentDataBindingService</code>	Lookup all bindings of a TSO, including GameObjects
<code>MyCustomQuery</code>	Custom logic like “find all active enemies with cooldown < 1s”

Creating Your Own Queries

You can define your own query helpers like this:

Using `EntityRegistry<T>`

```

csharp
CopyEdit
public class ActiveEnemyQuery
{
    private readonly EntityRegistry<EnemySO> _registry = Query<EntityRegistry<EnemySO>>();

    public List<EnemySO> GetLowCooldownEnemies(float threshold)
    {
        return _registry.GetAll()
            .Where(e => e.cooldown.Value < threshold)
            .ToList();
    }
}

```

✓ Using **ComponentDataBindingService**

```

csharp
CopyEdit
public class ActiveEnemyQuery
{
    private readonly ComponentDataBindingService _bindings = Query<ComponentDataBindingService>();

    public List<GameObject> GetLowCooldownEnemyObjects(float threshold)
    {
        return _bindings.GetBindings<EnemySO>()
            .Where(binding => binding.Data.cooldown.Value < threshold)
            .Select(binding => binding.gameObject)
            .ToList();
    }
}

```

Both methods are runtime-safe, testable, and avoid slow scene-wide scans.

Why Not Dependency Injection?

ReaCS avoids classic DI and favors **Query-based Shared Access**, because:

Feature	Traditional DI	ReaCS Query
 Easy to reason about	✗	✓
 Zero boilerplate	✗	✓
 Toolable in Editor	✗	✓
 Fast (Burst-compatible)	✗	✓

 Works in Edit Mode		
 Per-system customization		

Example: Registering a Shared Pool

```
csharp
CopyEdit
Use<ReaCSLinkPool<TagAssignmentSO>>(); // Register

var pool = Query<ReaCSLinkPool<TagAssignmentSO>>(); // Use
var tagLink = pool.Get();
tagLink.Left.Value = someTag;
tagLink.Right.Value = someObject;
```

Example: Index Queries for Relationships

```
csharp
CopyEdit
var index = Query<ReaCSIndexRegistry>();
var parts = index.RightOf<PartOwnershipSO, RobotSO, PartSO>(robotSO);
```

You can also query in the reverse:

```
csharp
CopyEdit
var owner = index.LeftOf<PartOwnershipSO, RobotSO, PartSO>(partSO);
```

This is how the system becomes **toolable** — your editor tools and debug graphs can use the same queries as runtime systems.

Tip: Replace Drag-and-Drop with Queries

Instead of this:

```
csharp
CopyEdit
[SerializeField] private ExperienceSO data;
[SerializeField] private GameEvent onSelected;
```

You can use:

```
csharp
CopyEdit
public override void OnChanged(...)
{
    var data = Query<ExperienceQuery>().GetById("X42");
    var eventSO = Query<GameEventRegistry>().Get("OnSelected");
}
```

Cleaner. Centralized. Debuggable. No clutter in the scene.

Summary

ReaCS encourages you to think in terms of shared queries and reusable tools, not MonoBehaviour plumbing.

- ✓ Use `Use<T>()` to register
- ✓ Use `Query<T>()` to access
- ✓ Pool, index, and retrieve — all without scene wiring

2. Query() — Structured Access to Data Relationships

Queries in ReaCS are stateless classes that describe how to access structured data. They don't mutate anything — they're just helpers for traversing your declarative data model.

Use this to access **stateless query helpers** that pull from the `ReaCSIndexRegistry` or other indexed relationships.

Queries answer questions like:

- "Which buffs does this monster have?"
- "What tags are assigned to this experience?"
- "Which parts belong to this model?"

Example: ExperienceQueries

```
public class ExperienceQueries : IReaCSQuery
{
    public ModelSO GetModelForExperience(ExperienceSO experience)
    {
        return ReaCSIndexRegistry
            .Query<ExperienceToModelLinkSO>(experience)
            .FirstOrDefault()?.target.Value;
    }

    public List<ModelPartSO> GetPartsForModel(ModelSO model)
    {
        return ReaCSIndexRegistry
```

```

        .Query<ModelPartAssignmentSO>(model)
        .Select(link => link.target.Value)
        .ToList();
    }
}

```

These queries are:

- Declarative
- Stateless
- Fully reusable

Usage

```
var model = Query<ExperienceQueries>().GetModelForExperience(currentExperience);
```

That's it — no drag-and-drop, no injection, no registration.

You write the access logic once, and use it anywhere.

3. Use() — Shared Logic, Pools, and Helpers

Not all shared logic is a query. Sometimes, you need:

- A link pool
- A formatting helper
- A runtime utility

That's where `Use<T>()` comes in.


```
var pool = Use<ReaCSLinkPool<BuffAssignmentSO>>();
```

You can also use it for helpers:

```
var parser = Use<ModelPartNameParser>();
var config = Use<ARSceneConfig>();
```

All of these are:

- Stateless (or safe singletons)
- Lazily constructed
- Reused automatically
- Globally accessible without wiring

 Do not substitute Systems with Services. These are only supposed to be universal helpers to keep the codebase DRY by not repeating a specialized tasks in multiple systems. Always try to write logic in systems and if you see any repetition going on after a while, refactor to a Service.

4. How It Works — Static Generic Registries

No reflection. No dictionary lookups. No runtime injection.

Internals:

```
public static class Query<T> where T : IReaCSQuery, new()
{
    public static readonly T instance = new();
}

public static class Use<T> where T : IReaCSService, new()
{
    public static readonly T instance = new();
}
```

You call `Query<T>()` or `Use<T>()` — and the system gives you:

- A singleton instance of `T`
- Lazily constructed the first time it's used
- Fully cached and type-safe

Marker Interface Enforcement

To prevent misuse and clarify intent, ReaCS uses two marker interfaces:

```
public interface IReaCSService {}
public interface IReaCSQuery {}
```

Every:

- `ReaCSLinkPool<T>`, `EntityRegistry<T>`, etc. → implement `IReaCSService`
- `TagQueries`, `BuffQueries`, `ModelPartLookupHelper`, etc. → implement `IReaCSQuery`

Attempting to `Use<T>()` on a Query type — or `Query<T>()` on a Service — results in a compile-time error.

 This ensures clean boundaries between access and global mutation logic.

5. PoolService— Pooled Reactive Objects

Need to reuse dynamic `ObservableScriptableObject` instances?

Use a pooled access pattern:

```
var link = Use<PoolService<BuffAssignmentSO>>().Get();
// ...
Use<PoolService<BuffAssignmentSO>>().Release(link);
```

Each `ReaCSLinkPool<T>` instance:

- Reuses link objects across frames
- Resets all `Observable<T>` fields
- Supports high-frequency systems (e.g. buffs, tags, highlights)
- **Automatically registers the SO to `ReaCSIndexRegistry`** when pulled from the pool
- **Optionally unregisters on return**, if the link is not meant to persist

Pool Implementation

```
using ReaCS.Runtime.Core;
using System.Collections.Generic;
using UnityEngine;

namespace ReaCS.Runtime.Services
{
    public class PoolService<T> : IReaCSService where T : ObservableScriptableObject
    {
        private readonly Stack<T> _pool = new();

        public T Get()
        {
            return _pool.Count > 0 ? _pool.Pop() : ScriptableObject.CreateInstance<T>();
        }

        public void Release(T instance)
        {
            _pool.Push(instance);
        }

        public void Clear()
        {
            _pool.Clear();
        }

        public int Count => _pool.Count;
    }
}
```

6. ReaCSIndexRegistry — The Backbone of Relationship Lookups

At the core of ReaCS's reactive structure is the `ReaCSIndexRegistry` — a global, fast lookup table that tracks all:

- `ObservableScriptableObject` instances
- `LinkSO<TOwner, TTarget>` relationships

The `ReaCSIndexRegistry` is what makes all of this fast and DRY.

- Registers every `ObservableScriptableObject` and `LinkSO` by type
- Enables relationship lookups with:

```
QueryAll<T>()  
Query<TOwner, TLink>(owner)
```

- Works with both burstable and editor-safe backends

✅ This means no linear scans, reflection, or manual relationship traversal ever again.

This registry powers all queries and reactive graph traversal. It lets you find all:

- Links from a given owner (e.g. all buffs on a monster)
- Links to a given target (e.g. all quests involving this step)
- Dynamic runtime-created SOs

Querying by Relationship

```
// Get all model parts linked to this model  
ReaCSIndexRegistry.Query<ModelPartAssignmentSO>(model);
```

✨ Automatic and Manual Registration

Creation Method	Registered?
ScriptableObject asset	✅ via <code>OnEnable()</code>
CreateInstance at runtime	✅ First time only
Pulled from ReaCSLinkPool	✅ Pool ensures registration and un-registration

✅ Why It Matters

- All data relationships are declared and discoverable.
- Systems can reason about state using pure queries.
- Tools and visualizations can display runtime relationships.
- You avoid using `FindObjectOfType`, `GetComponent`, or manual scene wiring.

Performance Considerations

- Internally uses `Dictionary<Type, HashSet<ObservableScriptableObject>>`
- Can index by owner and/or target
- Future-proof: can be optimized with `NativeHashMap` if needed for DOTS or Burst

Practical Example

```
var tagLinks = ReaCSIndexRegistry.Query<TagAssignmentSO>(monster);  
foreach (var tag in tagLinks)
```

```
{
  if (tag.target.Value.name == "Burning")
    ApplyBurnDamage(monster);
}
```

7. EntityRegistry— Your access to scene entities

`ReaCSIndexRegistry` is purely **about data relationships** — like a database for your observable model layer.

- It tracks what belongs to what (`owner → target`)
- It powers queries across reactive state





`EntityRegistry<T>` on the other hand is part of the **view layer** — the MonoBehaviours in the scene that respond to that state.

Core Differences in Responsibility

Feature	<code>ReaCSIndexRegistry</code>	<code>EntityRegistry<T></code>
 Purpose	Index all data objects (SOs, Links)	Track scene-bound MonoBehaviours
 Domain	Reactive state (ObservableScriptableObject)	Runtime Unity entities (visuals, bindings)
 Lifetime	Persistent, serializable or pooled SOs	Volatile, tied to scene/object lifecycle
 Access Pattern	Queried by owner/target type	Queried by MonoBehaviour type
 Burst-compatible	Yes (plannable)	No (Unity objects, non-blittable)
 Use case	Systems reacting to data relationships	Systems acting on bound runtime objects

Why Not Merge Them?

Here's what could go wrong if we merged:

-  Unity objects (like `TransformBinding`) would pollute the SO-only registry
-  `ReaCSIndexRegistry` would no longer be safely Burst-compatible
-  Mixed querying would become ambiguous (`QueryAll<T>()` — which T? MB or SO?)
-  The graph tooling would get noise from non-reactive objects

But They're Both Unified by `Use<T>()`

The key is: even though they're separate registries, they **use the same access pattern**:

```
// For data (via index)
var buffs = ReaCSIndexRegistry.Query<MonsterSO, BuffAssignmentSO>(monster);

// For scene entities
foreach (var binding in Use<EntityRegistry<HealthBarBinding>>().All) { ... }
```

So you get a **uniform access model** with distinct concerns.

Analogy

Think of:

- `ReaCSIndexRegistry` as your **database**
- `EntityRegistry<T>` as your **scene graph**

You wouldn't want them in the same container — but you want them both accessible.

8. Best Practices

Goal	Use
Access structured data	<code>Query<T>()</code>
Share a stateless helper	<code>Use<T>()</code>
Reuse dynamic state	<code>Use<ReaCSLinkPool<T>>()</code>
Lookup scene entities	<code>Use<EntityRegistry<T>>()</code>
Lookup relationships	<code>ReaCSIndexRegistry.Query<>()</code>

Keep in Mind:

- Queries are stateless
- Pools must handle registration
- Use declarative structure over imperative wiring
- Indexing is fast and avoids scanning entire memory

8. Optional Roadmap: Burst Compatibility for ReaCSIndexRegistry

While `ReaCSIndexRegistry` is not currently Burst-compatible by default, the system is designed to allow future upgrades for high-performance use cases.

Potential Upgrade Path:

- Maintain current `Dictionary<Type, HashSet<SO>>` for flexibility and editor tooling.
- Add parallel `NativeHashMap<int, TData>` for specific Burst-compatible workloads.
- Use integer IDs or lightweight proxies (structs) instead of direct SO references.
- Convert data into Native containers on-demand or at scene load time.

When to Consider:

- If you're Burst-compiling heavy simulation systems.
- If reactive components need to update 10,000+ objects per frame.
- If you're integrating tightly with Unity DOTS.

For most projects, this optimization is **unnecessary** — ReaCS already offers near-instant access patterns with zero GC once initialized.

You can confidently build your game or app with today's structure — and refactor into Burst-compatible registries only if performance profiling demands it.

9: UnifiedObservableRegistry — Query Everything

This new query class provides full access to:

- Runtime SOs
- Editor-time SOs (for tooling)
- `ComponentDataBindings` via entity ID

```
var positions = Query<UnifiedObservableRegistry>().GetByEntity<PositionSO>(id);
var transform = Query<UnifiedObservableRegistry>().GetBindingsForEntity<PositionSO, PositionBinding>(id).First().component;
```

Burst-Safe Lookups

Use `.BuildNativeLookup<TSO, TField>()` to pull native data for Burst-compatible systems.

```
var positions = Query<UnifiedObservableRegistry>().BuildNativeLookup<PositionSO, float2>(p
⇒ p.Value, Allocator.TempJob);
```

10. Summary

Pattern	Role
<code>Query<T>()</code>	Declarative read-only access to structure
<code>Use<T>()</code>	Stateless access to pooled logic, objects, and helpers
<code>ReaCSLinkPool<T></code>	High-frequency pooled reactive data
<code>ReaCSIndexRegistry</code>	Central store for all SOs and relationships

ReaCS gives you shared access that is fast, declarative, and fully toolable — without injecting anything.

Chapter 6: Connecting ReaCS with Core Unity Systems

ReaCS doesn't exist in a vacuum. Real-world Unity apps need to move objects, handle player input, switch scenes, play sounds, and load content on demand. This chapter explains how to integrate ReaCS with Unity's most important systems — without breaking the reactive architecture.

1. Binding MonoBehaviours to ObservableScriptableObjects

ComponentDataBinding<TSO, TUC>

Each runtime Unity object (or prefab) can declare component data by attaching a `ComponentDataBinding<TSO, TUC>` MonoBehaviour. This class:

- Instantiates or references a `TSO` (ObservableScriptableObject)
- Automatically resolves the required Unity Component (`TUC`) on the GameObject
- Assigns a shared `EntityId` used across the whole hierarchy

```
public class PositionBinding : ComponentDataBinding<PositionSO, Transform> { }
```

This allows systems to directly access both the `PositionSO` and the associated `Transform` in a safe and fast way.

Automatic Entity Resolution

A new service, `SharedEntityIdService`, assigns a unique `EntityId` to each top-level GameObject with bindings. All child bindings in the hierarchy will share the same `EntityId`.

Chapter 3: Component Bindings and Entities

ComponentDataBinding<TSO, TUC>

Each runtime Unity object (or prefab) can declare component data by attaching a `ComponentDataBinding<TSO, TUC>` MonoBehaviour. This class:

- Instantiates or references a `TSO` (ObservableScriptableObject)
- Automatically resolves the required Unity Component (`TUC`) on the GameObject
- Assigns a shared `EntityId` used across the whole hierarchy
- Registers itself in the runtime `ComponentDataBindingService<TSO>` for easy system access

```
public class PositionBinding : ComponentDataBinding<PositionSO, Transform> { }
```

The generic base makes it trivial to bind Unity-native components (like `Transform`, `Rigidbody`, `AudioSource`, etc.) to their data-driven `TSO` counterpart.

This enables reactive, decoupled logic:

```
if (Use<ComponentDataBindingService<PositionSO>>().TryGetBinding(positionSO, out var binding))
{
    var transform = binding.component;
    transform.position = positionSO.Value;
}
```

You never have to drag references or use `FindObjectOfType()` — the SO becomes your anchor.

Automatic Entity Resolution

Every `ComponentDataBinding<TSO, TUC>` automatically receives an `EntityId` from `SharedEntityIdService`. That ID is shared among all bindings in the same GameObject hierarchy (parent and children).

This means systems can associate and relate SOs across components, even in deeply nested prefabs.

Example:

```
public class HealthBinding : ComponentDataBinding<HealthSO, Collider> { }  
public class PositionBinding : ComponentDataBinding<PositionSO, Transform> { }
```

Both `HealthSO` and `PositionSO` will share the same `EntityId` if they are under the same prefab root, enabling systems to reason across components.

Entity Construction

Instead of wiring bindings manually, use an **EntityFactory** system to construct full entities at runtime:

```
csharp  
CopyEdit  
var factory = Query<ReaCSEntityFactory>();  
  
factory.Create<TransformSO>((so, go) => {  
    go.AddComponent<TransformBinding>().Bind(so);  
});
```

Template or Shared Data Source

Each binding exposes a single `dataSource` field (an asset reference in the project) and a toggle `useAsTemplate`. If `true`, the binding clones and pools a copy of the SO at runtime — isolating changes. If `false`, it references the SO directly, enabling shared state.

```
[SerializeField] private PositionSO dataSource;  
[SerializeField] private bool useAsTemplate;
```

- **Use as Template:** For pooled or per-instance data.
- **Direct Reference:** For shared static data (e.g. colors, categories, read-only configs).

Unity Component Binding with Safety

The generic parameter `TUC` is resolved via `[RequireComponent]` enforcement at compile time. Your prefab must include the correct Unity component — or Unity will warn you immediately.

You can then use `.component` to access it in a system or logic class.




Binding MonoBehaviours to SOs

Previously, bindings implemented `IHasOwner<T>` to expose their associated SO.

But in **ReaCS 0.5.0 and later**, ownership is tracked automatically through:

- `EntityRegistry<T>` — maps `TSO` ↔ all bindings using that data
- `ComponentDataBindingService` — allows querying all bound components for a given `TSO`

This enables:

-  Reactive lookups without needing a specific interface
-  Type-safe registration through `ComponentDataBinding<TSO, TUC>`
-  Tooling that doesn't rely on manual wiring

Example

You can now fetch bindings or GameObjects like so:

```
var bindings = Query<ComponentDataBindingService>().GetBindings<EnemySO>();
```

Or get all SOs currently bound:

```
var activeEnemies = Query<EntityRegistry<EnemySO>>().GetAll(EntityId);
```

Summary of the change:

Old Pattern (deprecated)	New Pattern
<code>IHasOwner<T></code> interface	Automatic registration via <code>ComponentDataBinding<TSO, TUC></code>
Manual ownership queries	Use <code>EntityRegistry<T></code> and <code>ComponentDataBindingService</code>
Required custom accessors	Now tracked internally by ReaCS

This approach is faster, safer, and removes boilerplate — while keeping all data reactive and queryable.

Tooling Integration

All component bindings are automatically registered into the `ComponentDataBindingService<TSO>` at runtime, and optionally into graph/debug systems.

This powers:

- Visual inspection of GameObject ↔ SO links
- Reactive graph animation
- Runtime history tracking (field changes + component context)

Explanation

Instead of dragging single `Observable<T>` fields into `MonoBehaviours`, the recommended approach is to reference the full `ObservableScriptableObject`. Each binding component should implement:


```
public interface IHasDataSource<T> where T : ObservableScriptableObject
{
    T DataSource { get; }
    bool UseAsTemplate { get; }
}
```

Then, your binding can use the SO's fields reactively or imperatively.

✅ Why IHasDataSource<T> and not IHasOwner<T>

📌 Clarity of Intent

- **"This MonoBehaviour reflects this data"** is more precise than "is owned by".
- Avoids confusion between *data ownership* vs *hierarchical or logical ownership*.
- Useful in bindings like `TransformBinding`, `HealthBarBinding`, `LabelBinding`, etc.
- Makes it possible to have conditional behavior based on `UseAsTemplate`:
 - If not filled, SO instance will be created with standard data and pooled
 - If filled and `useAsTemplate = true`, SO instance will be created by copying sourceData SO and pooled
 - If filled and `useAsTemplate = false`, SO instance will be linked to existing project SO so we can define fields that can be saved or not between plays. Opens up persistence.

✅ Flexible Naming for All Bindings

- Works for UI, audio, visuals, particles, etc.
- `IHasData<HealthSO>` is clear even when no ownership relationship exists.
- You might even use both in advanced cases:
 - `IHasData<HealthSO>` — the binding reflects it
 - `IHasOwner<CharacterSO>` — the SO is *owned* by the character

Benefits of IHasOwner<T>

- ✅ Type-safe
- ✅ Enables filtering in systems
- ✅ Supports graph tools and tracing
- ✅ Toolable and introspectable in the Editor

If a `Binding` is added at runtime — meaning it's instantiated via `Instantiate()` or attached to a prefab in the scene — it still works perfectly **as long as**:

✅ You Follow This Rule

→ Inherit from `ReaCSEntity<T>`

```
public class TransformBinding : ReaCSEntity<TransformBinding>, IHasData<TransformSO>
{
    public TransformSO data;
    public Transform target;
}
```

✓ `ReaCSEntity<T>` automatically registers itself into `EntityRegistry<T>` during `OnEnable()`

```
protected virtual void OnEnable()
{
    Use<EntityRegistry<T>>().Register(this as T);
}
```

So any runtime-created entity will still be tracked immediately.

2. Simulation Systems with Update()

ReaCS systems are reactive **by default**, but you can still write **simulation systems** that:

- Don't use `[ReactTo]`
- Instead override `Update()` or `FixedUpdate()`

That means that your bindings can be just declarative and not deal with logic!

Example: TransformBinding & MovementSystem

```
public class TransformBinding : ReaCSEntity<TransformBinding>, IHasData<TransformSO>
{
    public TransformSO data; // ObservableScriptableObject the binding reflects
    public Transform target;
}
```

```
public class MovementSystem : SystemBase
{
    private EntityRegistry<TransformBinding> entities;

    protected override void OnEnable()
    {
        base.OnEnable();
        // Get all entities in scene with a TransformBinding
        entities = Use<EntityRegistry<TransformBinding>>();
    }

    void Update()
    {
        foreach (var entity in entities.All)
        {
            if (entity.data == null) continue;
        }
    }
}
```

```

        var direction = data.direction.Value.normalized;
        var speed = data.speed.Value;

        var step = direction * speed * Time.deltaTime;
        entity.velocity = step;
        entity.target.position += step;
    }
}
}

```

This hybrid lets you:

- Keep simulation logic grouped and stateless
- Avoid Observable spam
- Stay fully compatible with ReaCS and Unity's execution model

3. Entity Registry for Scene Access

To avoid `FindObjectOfType` or tag-based lookups, use:

```

public abstract class ReaCSEntity<T> : MonoBehaviour where T : MonoBehaviour
{
    void OnEnable() ⇒ Use<EntityRegistry<T>>>().Register(this as T);
    void OnDisable() ⇒ Use<EntityRegistry<T>>>().Unregister(this as T);
}

```

Now any system can safely and efficiently fetch scene entities:

```

foreach (var entity in Use<EntityRegistry<TransformBinding>>>().All)
{
    ...
}

```

4. Async in Systems (e.g., Addressables)

Async logic like loading models or textures should be done in **systems**, not SOs or bindings.

```

public class LoadModelSystem : SystemBase<ExperienceSO>
{
    async void OnFieldChanged(ExperienceSO so)
    {
        var handle = Addressables.LoadAssetAsync<GameObject>(so.modelGUID.Value);
        var prefab = await handle.Task;
        Object.Instantiate(prefab);
    }
}

```

This keeps side effects centralized and avoids async SO pitfalls.

5. Input Handling

✓ Traditional Input (Input.GetAxis / GetKey)

Use an `InputSO` :

```
csharp
CopyEdit
public class PlayerInputSO : ObservableScriptableObject
{
    [Observable] public Observable<Vector2> moveDirection;
}
```

Update it in a polling system:

```
csharp
CopyEdit
public class InputSystem : SystemBase<PlayerInputSO>
{
    public override void Update()
    {
        data.moveDirection.Value = new Vector2(
            Input.GetAxisRaw("Horizontal"),
            Input.GetAxisRaw("Vertical"));
    }
}
```

✓ New Unity Input System

You can listen to input events and route them to SO fields. Example with UnityEvents:

```
csharp
CopyEdit
public class PlayerInputBinding : MonoBehaviour
{
    public PlayerInputSO data;

    public void OnMove(InputAction.CallbackContext ctx)
    {
        data.moveDirection.Value = ctx.ReadValue<Vector2>();
    }
}
```

6. Localization via UI Toolkit Converters (No Scripts Required)

ReaCS now supports localization directly inside UI Toolkit with **zero custom MonoBehaviours**.

Step-by-step Setup:

1. Create your OSO with key:

```
public class LocalizedLabelSO : ObservableScriptableObject
{
    [Observable] public Observable<string> key;
}
```

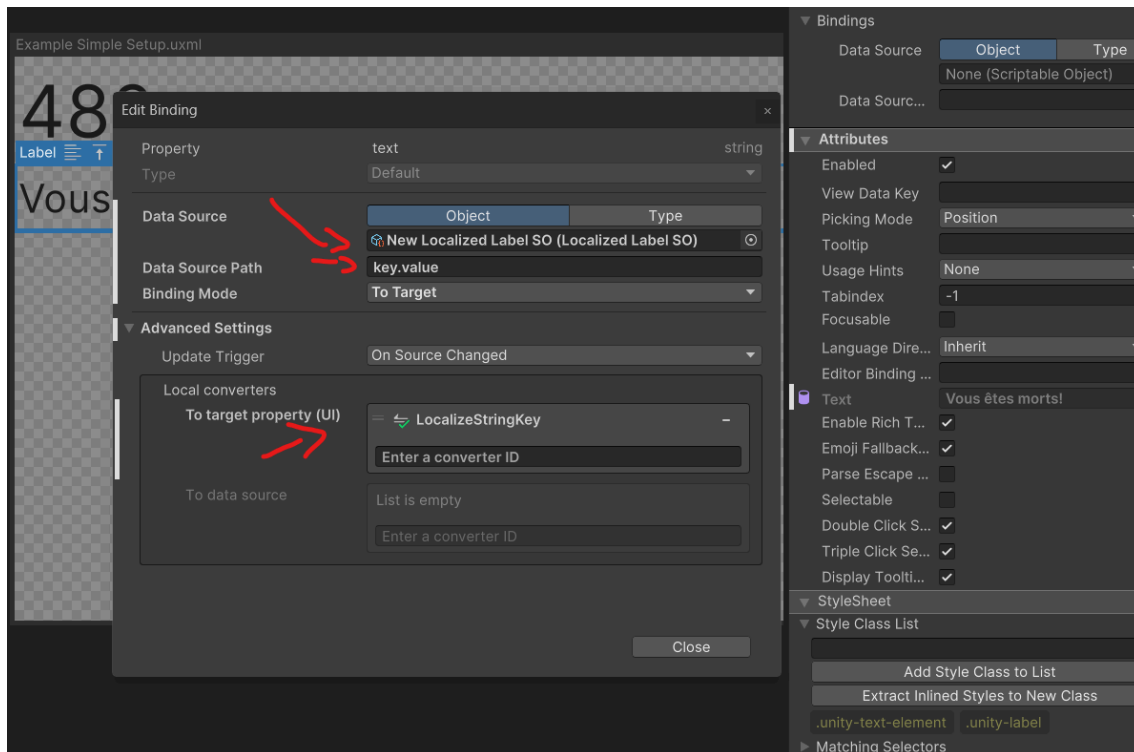
2. ReaCS comes ready with specific a specific converter:

```
using UnityEditor;
using UnityEngine.Localization;
using UnityEngine.UIElements;

public static class HandleStringKeyToLocalizedStringConversion
{
    #if UNITY_EDITOR
        [InitializeOnLoadMethod]
    #else
        [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.SubsystemRegistration)]
    #endif
    public static void RegisterConverters()
    {
        {
            var group = new ConverterGroup("LocalizeStringKey");
            group.AddConverter((ref string value) =>
            {
                var localizedString = new LocalizedString("NewTable", value);
                return localizedString.GetLocalizedString();
            });
            ConverterGroups.RegisterConverterGroup(group);
        }
    }
}
```

3. In UI Builder:

- Drag your `LocalizedLabelSO` into the **data source**
- Set the **path** to `key.value`
- Add the `LocalizeStringKey` converter to the `Label.text` binding



🎉 UI Toolkit now resolves the localized value at runtime & editor transparently — **no MonoBehaviour required.**

7. Addressables and Prefab Instantiation

Store GUIDs or keys as `Observable<string>` fields in your SOs. Let systems handle:

- Loading
- Tracking
- Instantiating prefabs

Use `ReaCSLinkPool<T>` to manage spawned instance data.

Addressables work perfectly with ReaCS by referencing prefab names or keys inside SOs.

ExperienceSO Definition Update

To support the reference:

```
csharp
CopyEdit
public class ExperienceSO : ObservableScriptableObject
{
    [Observable] public Observable<bool> isSelected;
    [Observable] public Observable<string> prefabKey;

    // This allows another system to react to it
}
```

```
[Observable] public Observable<GameObject> loadedInstance;
}
```

Example: Load on Select, Store Instance Reference

```
csharp
CopyEdit
using UnityEngine;
using UnityEngine.AddressableAssets;
using UnityEngine.ResourceManagement.AsyncOperations;
using ReaCS.Runtime;
using ReaCS.Runtime.Internal;

[ReactTo(nameof(ExperienceSO.isSelected))]
public class LoadSelectedExperienceSystem : SystemBase<ExperienceSO>
{
    public override void OnChanged(ExperienceSO data, string field)
    {
        if (data.isSelected.Value)
        {
            TryLoad(data);
        }
    }

    private void TryLoad(ExperienceSO data)
    {
        if (string.IsNullOrEmpty(data.prefabKey.Value)) return;
        if (data.loadedInstance.Value != null) return; // already loaded

        Addressables.LoadAssetAsync<GameObject>(data.prefabKey.Value).Completed += op =>
        {
            if (op.Status != AsyncOperationStatus.Succeeded)
                return;

            var instance = GameObject.Instantiate(op.Result);
            instance.name = $"Loaded_{data.name}";

            // Optionally bind the SO to the prefab
            if (instance.TryGetComponent<ComponentDataBinding<ExperienceSO, MonoBehaviour>
            >>(out var binding))
            {
                binding.Bind(data);
            }

            data.loadedInstance.Value = instance; // store reference reactively
        };
    }
}
```

```
}  
}
```

Unloading / Cleanup System

Later, you can write a **separate system** to unload on deselect:

```
csharp  
CopyEdit  
[ReactTo(nameof(ExperienceSO.isSelected))]  
public class UnloadDeselectedExperienceSystem : SystemBase<ExperienceSO>  
{  
    public override void OnChanged(ExperienceSO data, string field)  
    {  
        if (!data.isSelected.Value && data.loadedInstance.Value != null)  
        {  
            GameObject.Destroy(data.loadedInstance.Value);  
            data.loadedInstance.Value = null;  
        }  
    }  
}
```

✓ Behavior Recap

- Reacts only when `isSelected` is toggled
- Loads prefab once per SO when selected
- Unloads prefab when deselected or removed
- Uses `prefabKey` to determine what to load
- Optionally auto-binds the prefab's component to the same `ExperienceSO`

Summary

- `LoadSelectedExperienceSystem` does exactly one thing: load and store the instance.
- `ExperienceSO` holds the `GameObject` reference — fully observable.
- `UnloadDeselectedExperienceSystem` destroys it cleanly on deselect.
- You maintain separation of intent, pure reactivity, and no system-level memory tracking.

✓ Summary: Your Unity Bridge Layer

Tool	Purpose
<code>IHasOwner<T></code>	Link runtime bindings to data
<code>EntityRegistry<T></code>	Track scene objects in a system-safe way

Systems w/ <code>Update()</code>	Run simulation safely per-frame
SOs as intent + config	Keep runtime state centralized
Bindings as bridges	Drive Unity via clean separation
UI Toolkit + ConverterGroups	Enable localization at runtime — fully declarative, no code

This keeps your app reactive, testable, and fully decoupled from Unity's scene-centric messiness — while still integrating with all its powerful features.

Chapter 7: Entity-Based Runtime Architecture




`ReaCSEntityId` — The New Core Identifier

ReaCS now moves beyond `EntitySO`.

Instead of modeling runtime entities as ScriptableObjects, every runtime object now receives a **stable, integer-based identifier** via:

```
public struct ReaCSEntityId : IEquatable<ReaCSEntityId>
```

This ID is:

-  Assigned by the **SharedEntityIdService**
-  Shared by all bindings on the same GameObject hierarchy
-  Used in systems, queries, visual debugging, and native optimizations

Why Move Away from EntitySO?

EntitySO	ReaCSEntityId
Requires ScriptableObject instantiation	Pure runtime ID — no GC or asset overhead
Hard to sync across bindings	One ID shared across all components
Limited to Unity object lifecycle	Can be used in Burst jobs, native containers
Less performant for lookups	Fast indexed access, comparables, zero allocations

How It Works

When a `ComponentDataBinding<TSO, TUC>` is added to a GameObject at runtime or in the scene:

1. The binding receives or creates a `ReaCSEntityId`
2. All sibling bindings share the same ID
3. Systems and registries use this ID to link runtime behavior

Example:

```
public class HealthBinding : ComponentDataBinding<HealthSO, HealthUI>
{
```

```
// entityId is automatically assigned and synced
}
```

Now `HealthBinding`, `TransformBinding`, `CooldownBinding`, etc. — all on the same `GameObject` — **share a single `ReaCSEntityId`**.

Use Case Recap

✓ 1. Instantiate a prefab with multiple bindings

```
var go = Instantiate(enemyPrefab);
var health = CreateSO<HealthSO>();
var cooldown = CreateSO<CooldownSO>();

go.AddComponent<HealthBinding>().Bind(health);
go.AddComponent<CooldownBinding>().Bind(cooldown);
```

All bindings are now linked via the same `ReaCSEntityId`.

✓ 2. Access any component of that entity via a system

Let's say we want to reduce `HealthSO` every time the cooldown **expires** (e.g. when `cooldown.remaining` hits 0).

Assuming `CooldownSO` looks like this:

```
public class CooldownSO : ObservableScriptableObject
{
    [Observable] public Observable<float> remaining;
}
```

Then the system should be:

```
[ReactTo(nameof(CooldownSO.remaining))]
public class CooldownSystem : SystemBase<CooldownSO>
{
    public override void OnChanged(CooldownSO data, string field)
    {
        if (data.remaining.Value <= 0f)
        {
            var entityId = data.entityId.Value;

            var health = Query<EntityRegistry<HealthSO>>().GetByEntityId(entityId);
            if (health != null)
            {
                health.current.Value -= 10;
            }
        }
    }
}
```

```

        // Optionally reset or disable cooldown
        data.remaining.Value = 5f;
    }
}
}
}

```

Alternative: Simulation System (Time-Driven)

If you want it to tick every frame:

```

public class CooldownTickSystem : SystemBase<CooldownSO>
{
    public override void Update()
    {
        data.remaining.Value -= Time.deltaTime;

        if (data.remaining.Value <= 0f)
        {
            var entityId = data.entityId.Value;

            var health = Query<EntityRegistry<HealthSO>>().GetByEntityId(entityId);
            if (health != null)
            {
                health.current.Value -= 10;
                data.remaining.Value = 5f;
            }
        }
    }
}

```


Summary

Purpose	Use
React when value changes	<code>[ReactTo(nameof(Field))]</code> + <code>OnChange()</code>
Continuous ticking	<code>Update()</code> simulation system
Avoid errors	Always specify <code>[ReactTo(...)]</code> if using <code>OnChange</code>

No need for references, no drag-and-drop, no scene dependencies.

You can also if you want prefabs already in scene or not creating the bindings by code do the following:

To maintain **entity identity across all data components**, even those not tied to a Unity `MonoBehaviour`, you need to:

 Create a `ComponentDataBinding<TSO>` with no Unity component type — just the data.

This is how **pure data-only SOs** like `CooldownSO`, `HealthSO`, `PositionSO`, etc., can still:

- Share the same `ReaCSEntityId`
- Be registered in `EntityRegistry<T>`
- Participate in full runtime systems
- Exist even without any `GameObject` binding

Clean Pattern: Data-Only Component Binding

Define a data-only binding


```
public class CooldownDataComponent : ComponentDataBinding<CooldownSO>
{
    // No Unity component (TUC), just data and entity ID
}
```

This still assigns and tracks `ReaCSEntityId` automatically.

You're inheriting from:

```
public abstract class ComponentDataBinding<TSO>
    : ComponentDataBinding, IHasDataSource<TSO>
    where TSO : ObservableScriptableObject, new()
```

Which:

-  Handles `dataSource` reference (template or direct)
-  Clones from template if `useAsTemplate = true`
-  Assigns a shared `EntityId` via `SharedEntityIdService.GetOrAssignEntityId(transform)`
-  Registers the `data` to:
 - `ComponentDataBindingService<TSO>`
 - `ReaCSIndexRegistry`

That means **even if there's no TUC (Target Unity Component)** like `Slider`, `Transform`, etc., you're still:

- Creating the data
- Binding it
- Assigning an entity ID

- Making it queryable and observable

How It Works Without a View Component

You are leveraging this:

```
var go = new GameObject("Entity");
go.AddComponent<CooldownDataComponent>().Bind(cooldownSO);
go.AddComponent<HealthBinding>().Bind(healthSO);
```

All bindings on the same GameObject will get the same EntityId via GetOrAssignEntityId(transform).

Your system can then react to either one and look up the other via:

```
var id = data.entityId.Value;
var health = Query<EntityRegistry<HealthSO>>().GetByEntityId(id);
```

Runtime Behavior Checkpoints

Behavior	Working?
<code>CooldownDataComponent</code> auto-initializes <code>CooldownSO</code> ?	✓ Yes
<code>CooldownSO</code> gets assigned a valid <code>entityId</code> ?	✓ Yes (<code>IHasEntityId</code>)
<code>CooldownSO</code> is registered in <code>ReaCSIndexRegistry</code> ?	✓ Yes
<code>CooldownSO</code> is queryable via <code>EntityRegistry<CooldownSO></code> ?	✓ Yes
Multiple <code>ComponentDataBinding<></code> s on same GO share same ID?	✓ Yes

✓ 3. Native/Burst-Compatible Data Structures

Entity IDs can be used in:

- `NativeHashMap<ReaCSEntityId, float>`
- Shared component lookup tables
- Graph overlays and causal visualizations
- Runtime analytics, sorting, or indexing

Tools and Helpers

- `SharedEntityIdService` — assigns and tracks IDs
- `ReaCSBinding` — base class that exposes `entityId`
- `EntityRegistry<T>` — allows lookup by entity ID:

```
csharp
CopyEdit
var binding = Query<EntityRegistry<HealthSO>>().GetByEntityId(id);
```

Best Practices

Goal	Recommendation
Create full entity at runtime	Use a factory or helper method that adds all bindings and assigns the same ID
Link SOs reactively	Store their <code>entityId</code> directly in the SO on bind
Access entity-local data	Use <code>EntityRegistry<T>.GetByEntityId(...)</code>
Clean up	Unregister and destroy all SOs + bindings together

Summary

ReaCSEntityId is the foundation for stable, reactive, non-referential entity modeling.

- ✓ Clean, performant identity
- ✓ Shared by all components
- ✓ Usable in jobs and native systems
- ✓ Central to simulation, input, and visualization

Chapter 8: Tooling, Debugging, and Visual Tracing

Modern reactive systems demand modern debugging tools — and ReaCS delivers. Thanks to its declarative and observable architecture, ReaCS enables developers to visualize the flow of state, trace bugs, and understand interactions in a way that's simply not possible with traditional Unity workflows.

1. Why Tooling Is Easy in ReaCS

ReaCS enforces declarative state and relationships. There are:

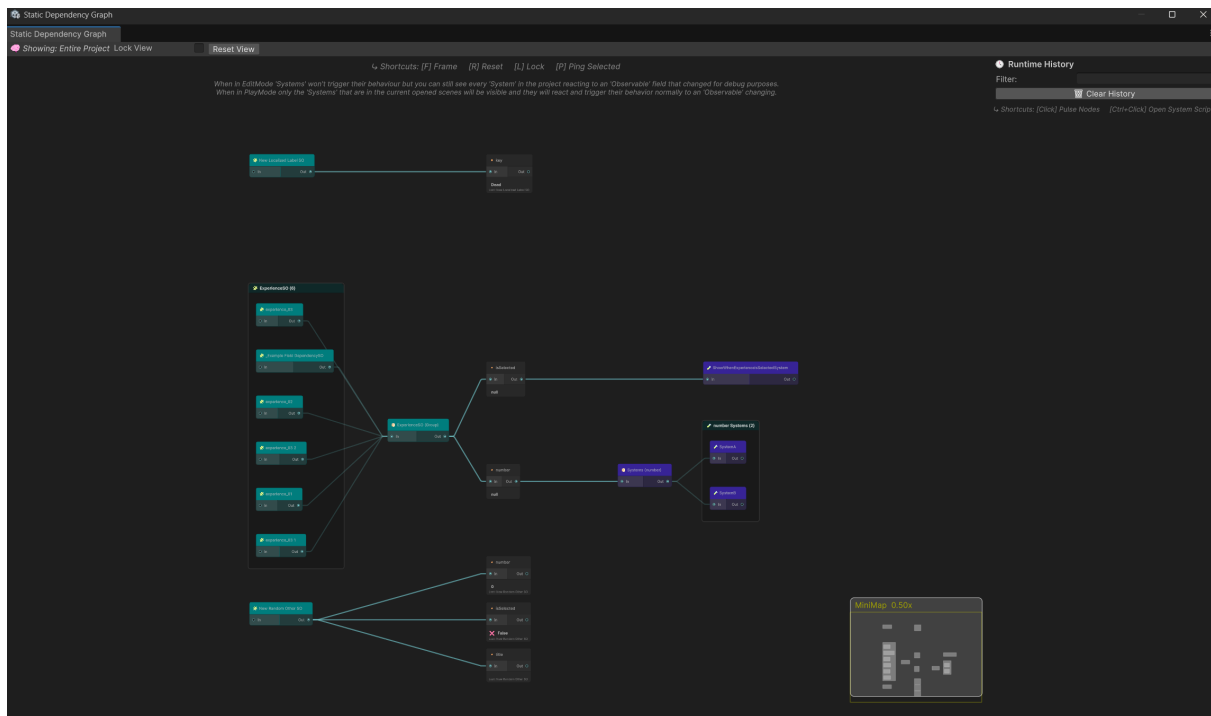
- No hidden `Update()` loops
- No implicit cross-references
- No black-box managers or magic strings

Every reaction is explicit:

- A `SystemBase<T>` reacts to specific `Observable<T>` fields

- Every change is traceable to an SO field and a system
- All reactive state is centralized in `ObservableScriptableObject` s

That means the architecture **is the tooling** — all we do is visualize it.



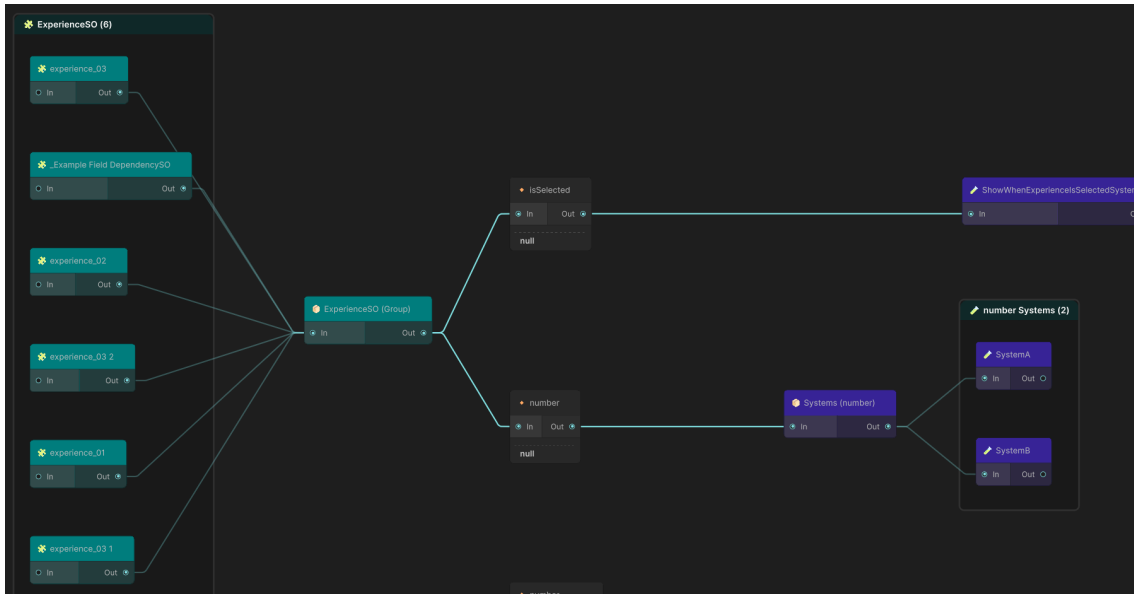
2. Static Dependency Graph (Editor View)

This graph view shows the **structural** relationships between:

- ScriptableObjects (SO)
- Their fields (Observable)
- Systems reacting to those fields

🎯 **Use it to understand architecture:**

- What systems are coupled to what data?
- Are we overreacting to a single field?
- Is one system touching too many unrelated fields?




3. Runtime Execution Trace (Live Causal Graph)


This dynamic graph shows the **flow of a reactive update** at runtime:

- [Trigger] → [Field A changed] → [System X] → [Field B changed] → ...

Each node contains:

- The SO name
- The field name
- The old → new value change

 This allows you to trace what really happened in the last frame.

 Use it to:

- Debug unexpected changes
- Find cascade chains
- See if a system caused an unintended write

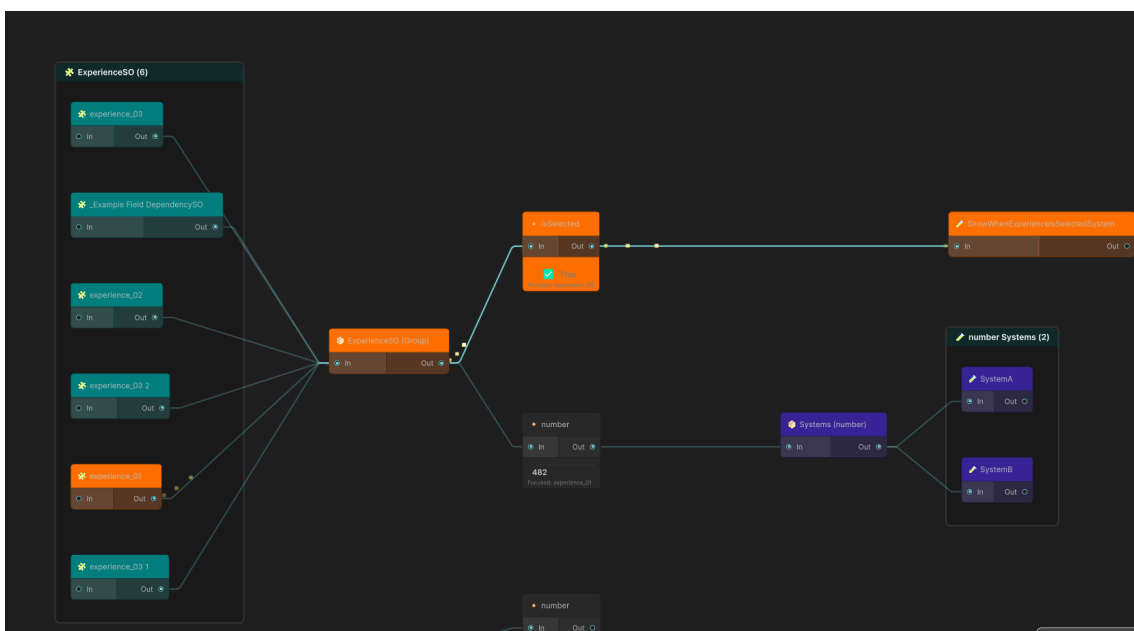
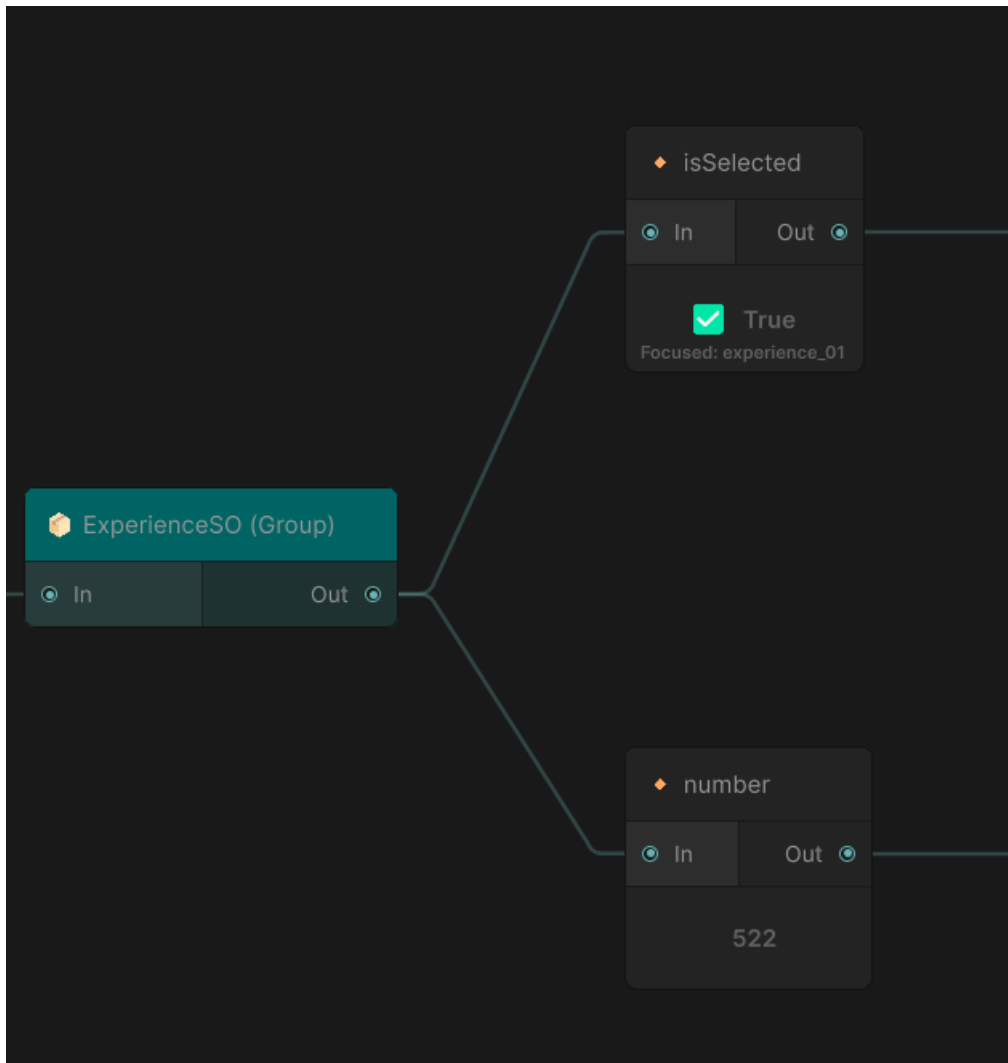


(This feature is WIP and coming soon!)

4. Field Value View & Pulse Animations

Every field node:

- Displays the **live value** of the field
- Shows which SO changed it last
- Pulses visibly on change
- Sends a ripple through any affected systems



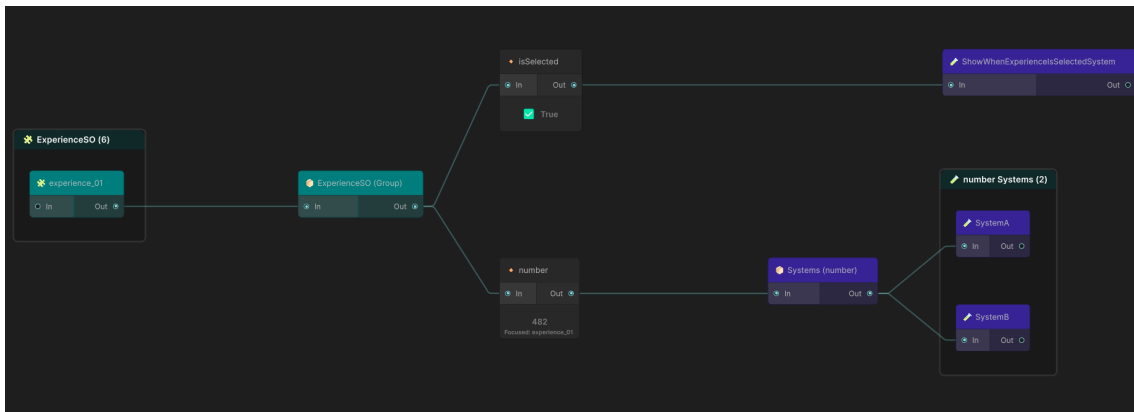
5. Focus Mode

Focus Mode lets you isolate a slice of the graph:

- Filter by SO
- Filter by field
- Filter by system

🔍 This makes debugging clean and comprehensible:

- No visual clutter
- No irrelevant edges
- Only the path that matters



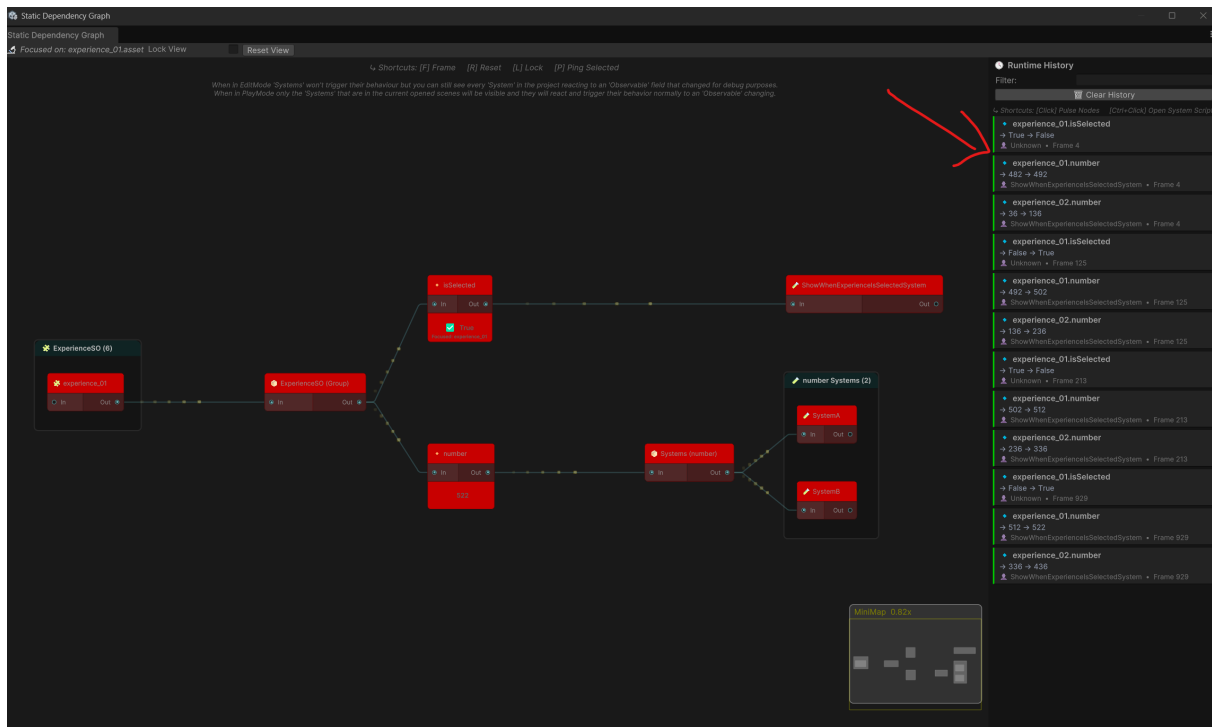
6. Runtime History Explorer

A list view showing **all reactive events over time**:

- Each frame's changes
- Field name, value change, triggering system

🕒 You can:

- Filter by system, SO, or field
- Jump into the execution trace for that step



Runtime History	
Filter:	
Clear History	
Shortcuts: [Click] Pulse Nodes [Ctrl+Click] Open System Script	
experience_01.isSelected	→ True → False
Unknown • Frame 4	
experience_01.number	→ 482 → 492
ShowWhenExperienceIsSelectedSystem • Frame 4	
experience_02.number	→ 36 → 136
ShowWhenExperienceIsSelectedSystem • Frame 4	
experience_01.isSelected	→ False → True
Unknown • Frame 125	
experience_01.number	→ 492 → 502
ShowWhenExperienceIsSelectedSystem • Frame 125	
experience_02.number	→ 136 → 236
ShowWhenExperienceIsSelectedSystem • Frame 125	

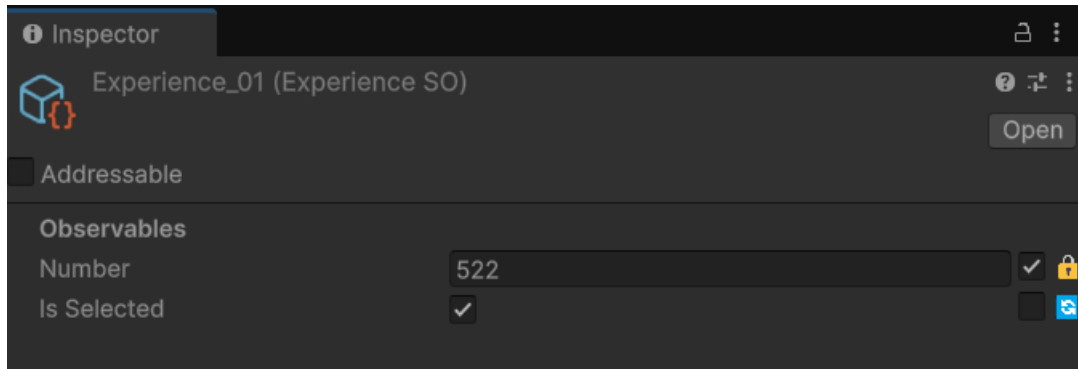
7. Inspector Features

Inside the Unity Inspector:

- ✓ Live value label beside each `Observable<T>`

- ☒ "Last changed by" metadata
- ☒ Lock icon toggle for `ShouldPersist`

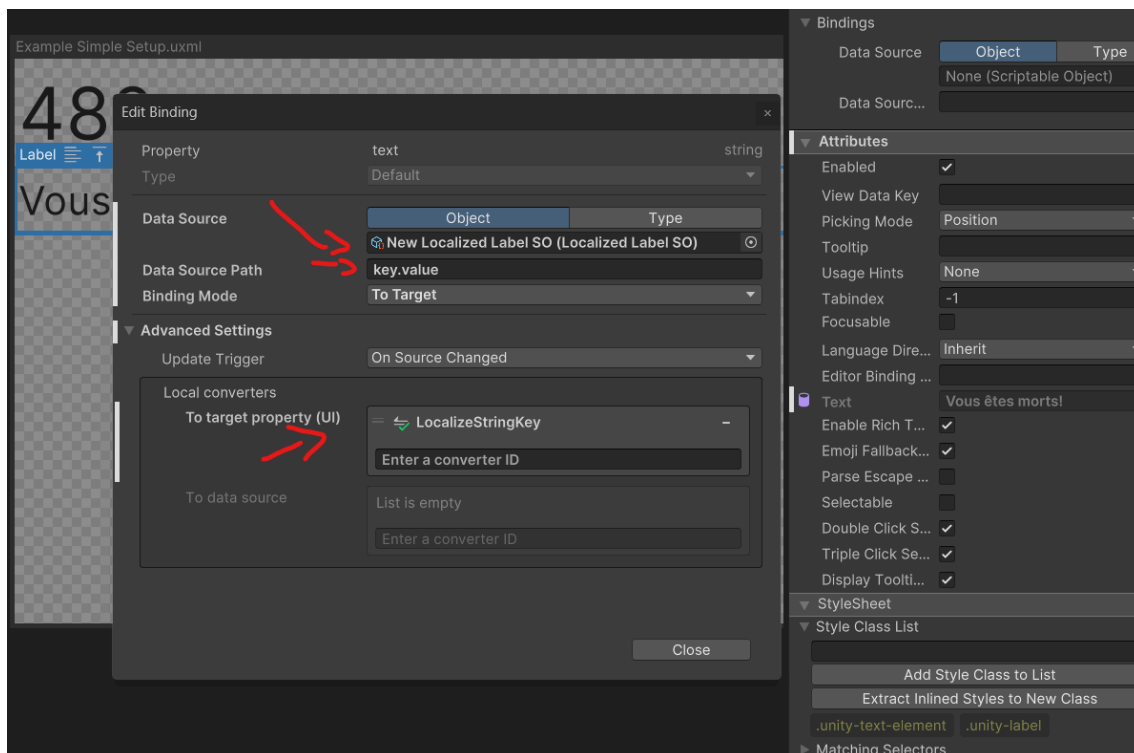
You can literally **click to control save behavior** of each field.



8. Bonus: Native Localization Support via Converters

You can bind localized text directly via UI Toolkit and ReaCS Observable fields:

- Use Unity's `ConverterGroups.Register()` with `LocalizedString.GetLocalizedString()` - This is done by default for you and included in ReaCS!
- Bind `Label.text` to `.key.value` and assign the `LocalizedStringKey` converter in UI Builder



- ☒ This enables runtime localization without MonoBehaviours or custom labels.

LinkSO Cleanup

Broken `LinkSO<, >` objects referencing missing assets are now auto-deleted in:

```
LinkSOCleanupUtility.CleanupBrokenLinks();
```

This is integrated with `ReaCSIndexRegistry` which automatically drops runtime links referencing destroyed SOs.

Summary: Why ReaCS Tooling Wins






Feature	Why it matters
Structural graph	Understand how your systems are wired
Execution trace	See what really happened and why
Animated pulses	Immediate visual feedback for field changes
Inspector labels	Debug without logs or breakpoints
History explorer	Review bugs after the fact
Focus mode	Zoom into relevant updates only

ReaCS is not just reactive — it's **transparent**. These tools transform debugging from frustration into exploration.

Chapter 9: Performance and Architecture Comparison


ReaCS offers a modern reactive data architecture designed to be fast, observable, and toolable — without sacrificing Unity compatibility. In this chapter, we compare ReaCS and its VSC architecture against traditional Unity MonoBehaviour setups, MVC/MVVM patterns, DOTs, and reactive frameworks like UnityEvents and C# events.

1. Unity's Performance Pitfalls by Default

-  Every `MonoBehaviour` has an optional Update loop — even when unused
-  Scattered memory layout leads to CPU cache misses
-  No structure to ownership — everything references everything
-  GC allocations from UnityEvents, lambda expressions, LINQ
-  Frequent use of `FindObjectOfType`, `GetComponent`, `SendMessage`, etc.

These patterns add up to invisible complexity, unpredictable behavior, and runtime slowdowns.

2. How ReaCS Fixes This

 ReaCS removes noise and indirection:

- `Observable<T>` changes are constant-time, zero alloc

- Systems are stateless and don't run unless needed (except for simulation cases)
- All data is traceable and strongly typed
- Relationship lookup is fast via index (not iteration or reflection)
- No more `Update` spam — only opt in when simulation is required

3. The Role of ReaCSIndexRegistry

The `ReaCSIndexRegistry` is a global reactive lookup engine:

- Registers all SOs and LinkSOs by type and relationship
- Enables instant queries by owner, target, type
- Avoids linear scans — ideal for systems that operate across entities

✅ Can be internally upgraded to `NativeHashMap` or DOTS-compatible index in the future

4. Pooling Support

LinkSOs and runtime SOs are regular objects — and can be pooled.

- `ReaCSLinkPool<T>` allows reuse of relationships like `BuffAssignmentSO`, `ModelPartAssignmentSO`, etc.
- Runtime OSOs can also be pooled (optional) for temporary data like combat stat previews, particle triggers, etc.
- Pools are injectable via `Use<T>()`, just like queries — no DI needed

5. Burst Compatibility Roadmap

While ReaCS is not ECS, many internals are Burst-friendly:

- `Observable<T>` fields are blittable, immutable, and can be mirrored to structs
- Index queries could be upgraded to use `NativeHashMap`
- Systems could mirror state to job contexts or Unity Physics bodies if needed

🚀 In the long term, ReaCS could simulate high-performance systems with ECS-style scheduling but retain full Unity editor friendliness.

6. Observable vs UnityEvents and C# Events

Feature	UnityEvent / C# Event	Observable (ReaCS)
Alloc-free?	❌ No	✅ Yes
Inspector friendly?	✅ Partial	✅ Fully SO-based
Debuggable?	❌ Hidden call stacks	✅ Tracked and visualized
Toolable?	❌ Not introspectable	✅ Graph + History view
Flexible?	⚠️ One-to-many, hard to remove	✅ Full traceability
Composable?	❌ No (manual chaining)	✅ Systems react declaratively

🔗 UnityEvents are fine for button clicks. But for **reactive state propagation**, `Observable<T>` is safer, debuggable, and deterministic.

7. Observable vs INotifyPropertyChanged

Feature	INotifyPropertyChanged	ReaCS Observable
Requires Reflection?	✅ Yes	❌ No
GC Allocations?	⚠️ Depends on binding framework	❌ None
Unity-compatible?	❌ Not natively	✅ Fully native
Toolable?	❌ Difficult to inspect	✅ Visual, live-traceable
Decoupled logic?	❌ Often bound to ViewModels	✅ Stateless Systems

🧠 `INotifyPropertyChanged` is great for WPF, but not for Unity's runtime needs.

8. Comparison Table

Feature	Unity Default	ReaCS	DOTS
Performance (runtime)	⚠️ Fragmented	✅ Fast w/ cache	✅✅✅ Max speed
Memory layout	❌ Scattered	✅ Predictable	✅✅✅ Optimized
Update overhead	❌ Per MB	✅ Event-driven	✅✅✅ Parallel
Reactive flow	❌ Manual	✅ Built-in	⚠️ Custom impl
DI and access patterns	❌ Ad hoc	✅ Use<Query>()	✅ Query world
Editor workflow	✅ Familiar	✅ Familiar	❌ Complex
Tooling	❌ Missing	✅ Visual/Graph	❌ Minimal
Ease of onboarding	✅ Easy	✅ Easy + DRY	❌ Hard

9. When to Use ReaCS Instead of DOTS

✅ Use ReaCS if:

- You want reactive, debuggable gameplay logic
- You care about persistence and clean state modeling
- You work heavily with UI Toolkit, addressables, or Unity scenes

❌ Use DOTS if:

- You need to simulate **hundreds of thousands** of objects per frame
- You're building physics or pathfinding at scale

💡 You can always combine both — ReaCS for logic and DOTS for parallel simulation.

✅ Summary

ReaCS isn't just fast — it's productive, debuggable, and Unity-native.

You get:

- Fast, indexable runtime access
- Reactive updates with zero allocations

- Visual debugging and change tracking
- Fully editable Unity workflow (ScriptableObject-based)

And most of all — **clarity**. You always know what's reacting, what's changing, and why.

Chapter 9: Best Practices, Patterns & Design Principles

This chapter gathers the collective wisdom from real ReaCS use cases and structural patterns to help you write clean, scalable, and fully reactive Unity applications. Treat it as both a cheat sheet and a guide to architectural integrity.

1. Modeling State

- ✓ Always use `ObservableScriptableObject` to represent runtime state. Even a single bool flag deserves one if it changes over time.
- ✓ Use `Observable<T>` fields to store any value that may change, be reacted to, or persisted.
- ✓ Use the `ShouldPersist` checkbox to control whether the field saves across sessions.
- ✗ Avoid nested SOs or deep lists — prefer flat relationships (`LinkSO<TOwner, TTarget>`).
- ✓ For ownership relationships between `ObservableScriptableObject` (e.g., "this belongs to..."), use `IHasOwner<T>` .

2. Modeling Relationships

- ✓ Model links with `LinkSO<TOwner, TTarget>` .
- ✓ Add metadata to links — e.g., `BuffAssignmentSO` can store duration, strength, or origin.
- ✓ Avoid storing `List<T>` inside SOs — instead, make the relationship a first-class SO type.
- ✓ Use `ReaCSIndexRegistry.Query<T>()` to find all relevant links or relationships.
- ✓ Index all `LinkSO` s so systems and tools can traverse them without scanning.

3. Working with Systems

- ✓ Use `[ReactTo(nameof(field))]` to make your system respond to a field.
- ✓ Use `OnFieldChanged()` for reactive logic, or `Update()` if simulating over time.
- ✓ Use `Use<T>()` to get access to:
 - Registries (`EntityRegistry<T>` , `ReaCSLinkPool<T>` , etc.)
 - Query helpers
 - Shared memory containers
- ✓ Use `Query<T>()` for access-only helpers.
- ✗ Do not place logic in MonoBehaviours. Bindings should only read or reflect state, not modify it.

4. Input and Control Flow

- ✓ Create a central `InputSO` for example with observable fields for direction, button states, etc.
 - ✓ Write a system that listens to `InputSO` and applies it to entities (e.g., movement, aiming).
 - ✓ Bind your UI or Input system to the SO — not directly to the game world.
 - ✗ Avoid writing control logic in UI callbacks or directly inside bindings.
-

5. Visuals, Animations, and Unity Integration

- ✓ Use MonoBehaviours like `TransformBinding`, `AudioBinding`, `HealthBarBinding`.
 - ✓ Implement `IHasOwner<T>` so each knows what SO drives it.
 - ✓ Use `ReaCSEntity<T>` to auto-register them in `EntityRegistry<T>`.
 - ✓ Let visuals read state and reflect — don't mutate state.
-

6. Undo/Redo, Snapshots & Time Travel

- ✓ Store a snapshot of every `Observable<T>` field every frame for full state rewind.
 - ✓ Snapshot can be compressed or limited (e.g., keep last 100 frames).
 - ✓ Use execution trace graphs to visualize per-frame updates and rollbacks.
 - 🧠 Each `ObservableScriptableObject` can act like a `FrameState` node in a time timeline.
-

7. Debugging & Tooling Tips

- ✓ Use Focus Mode to isolate only the node/field/system you're debugging.
 - ✓ Use Pulse animations to visually trace what changed.
 - ✓ Use the runtime history list to go backward and forward in event order.
 - ✓ Use the inspector "last changed by" tag to discover which SO caused a value change.
 - ✓ Use `ReaCSGraphView` in edit mode to study static dependencies.
 - ✓ Use `ExecutionTraceGraphView` in runtime to study causal updates.
-

8. Anti-Patterns to Avoid

- ✗ Do not write logic in UI callbacks (`onClick`, etc.) — let the UI update the SO, and systems do the rest.
 - ✗ Do not write lists inside SOs if the list represents a dynamic relationship — model it as a `LinkSO`.
 - ✗ Do not rely on `UnityEvents`, `C# events`, or `SendMessage` — they are not traceable, inspectable, or toolable.
 - ✗ Do not reference SOs across scenes unless explicitly designed to persist.
-

9. Naming & Convention Guidelines

- ✓ Use suffix `SO` for all `ScriptableObject` types:
 - `ExperienceSO`, `MonsterSO`, `InputSO`, etc.

✓ Use suffix `AssignmentSO` or `LinkSO` for link types:

- `BuffAssignmentSO`, `TagAssignmentSO`, etc.

✓ Name systems after the domain or reaction:

- `MoveEnemiesSystem`, `HidePopupWhenDeselectedSystem`

✓ Name observable fields in `camelCase`:

- `speed`, `health`, `isActive`, `labelKey`

✓ Use descriptive system comments:

```
[ReactTo(nameof(MonsterSO.health))]  
// Reacts to monster taking damage; hides them if health < 0
```

✓ Summary

ReaCS is built for clarity — and these best practices ensure you keep it that way. Use SOs for state, links for relationships, systems for logic, and Unity only for rendering or bridging.

A clean ReaCS project will:

- Be fully observable and debuggable
- Stay performant over time
- Be easier to onboard new devs into
- Scale cleanly from simple apps to full games

Stick to the principles above, and you'll build Unity projects that are reactive, robust, easily debuggable and remarkably maintainable.

Chapter 10: Recap and Final Thoughts

With ReaCS and the VSC Architecture Paradigm, you've now explored a complete rethinking of how to build clean, reactive, maintainable applications in Unity — without sacrificing performance, tooling, or usability.

This chapter summarizes the key ideas from each chapter and offers a final reflection on how to use ReaCS effectively.

Chapter-by-Chapter Recap

Chapter 1: What Is ReaCS & the VSC Architecture?

- ReaCS is a reactive architecture built on Unity's ScriptableObject system.
- VSC stands for **View-State-Controller**:
 - View = UI or MonoBehaviour bindings
 - State = ObservableScriptableObjects
 - Controller = Stateless Systems that react to State

- Inspired by ECS and reactive UIs, not MVC.

Chapter 2: How Unity Encourages Component-Centric Chaos

- Default Unity workflows evolve into tangled MonoBehaviour hierarchies
- Lists, events, and hard references create fragility
- ReaCS flattens state, removes event soup, and enforces clarity

Chapter 3: Anatomy of VSC

- Views display; State stores; Controllers react
- Systems are not traditional "controllers" — they're stateless, testable reactions
- Observable and LinkSO<TLeft, TRight> define the reactive world

Chapter 4: Relationships, Links, and the Problem with Lists

- Avoid embedded lists — model relationships explicitly with LinkSO
- Flatten data for faster access, clearer ownership, and Burst-compatibility
- Observable fields with ShouldPersist control save behavior per field

Chapter 5: Shared Access and Queries

- Use `Use<T>()` and `Query<T>()` instead of DI or manual references
- Pools and helpers are lazy singletons, DRY by default
- ReaCSIndexRegistry handles fast relationship lookups

Chapter 6: Connecting ReaCS to Unity

- Use `EntityRegistry<T>` to track runtime bindings
- Use `IHasOwner<T>` to link bindings to OSOs
- UI Toolkit can bind directly to Observable fields
- Localization via ConverterGroup + LocalizeStringKey = native, automatic

Chapter 7: Tooling, Debugging, and Visual Tracing

- Static dependency graph: see how everything connects
- Runtime trace: see what changed, who caused it, and why
- Pulse animations, history tracking, focus mode — all built-in

Chapter 8: Performance and Architecture Comparison

- Compared ReaCS to Unity Default, DOTS, MVVM, UnityEvents
- Zero-GC observable updates
- Predictable memory layout
- Compatible with Burst and Native containers

Chapter 9: Best Practices and Design Principles

- Use flat models, one-to-one links, and declarative updates
 - Keep systems stateless
 - Bindings display, systems compute
 - Use naming conventions for clarity and structure
-

Final Thoughts

ReaCS isn't just a framework — it's a shift in mindset.







Instead of:

- Pull-based logic, you get push-based updates
- Deeply coupled components, you get flat, indexed data
- Hidden bugs, you get visual traces
- Untrackable runtime behavior, you get pulse, history, and context

ReaCS doesn't aim to replace ECS or DOTS — it complements them by providing a developer-friendly, visually inspectable layer for reactive data and systems. It's Unity-native, fast, and elegant.

Whether you're building UI-heavy apps, games with dynamic relationships, or tools that demand reactive flow — ReaCS gives you the clarity and performance to ship with confidence.

What You Get with ReaCS

-  Fully observable state and reactions
-  Visual tools for graphing, tracing, and debugging
-  Fast runtime with optional Burst-based indexing
-  Simple, declarative system composition
-  Editor-friendly, Unity-native setup
-  Designed to scale across teams and projects

Thank you for exploring ReaCS and the VSC architecture. This document is both a reference and a foundation — adapt it, expand it, and help push Unity development forward.

Happy building!