

UI ARCHITECTURE

Reaktor Surface

An illustrated architecture for adaptive,
tactile, graph-native interfaces

Component meaning and behavior.

Explicit graph ownership.

Local rendering and tactile response.

Compose-first implementation.

Foundations

Subsystems

Applications

Contents

Part I — The problems and the foundations	3
1. Why contemporary interfaces become difficult to maintain	3
2. The concepts that connect application meaning to pixels	7
3. The load-bearing ideas	10
4. High-level architecture	13
5. Platforms, composition languages and rendering families	16
6. A worked example: one Send action through the whole system	19
 Part II — The framework and its subsystem contracts	 22
7. The small set of interfaces that holds the system together	22
8. Component definitions and the catalogue	27
9. Interaction machines that do not depend on Compose or React	30
10. Input semantics, editing and accessible operation	33
11. Atomic design as a practical composition model	36
12. Themes, tokens and interchangeable visual languages	39
13. Graph bindings, identity and lifecycle ownership	44
14. Adaptive layouts, navigation and coordinate systems	48
15. The Compose-first implementation	53
16. HostApplier and the portable composition boundary	56
17. React, reconciliation and the later RSD realization	60
18. Animation, transitions and temporal behavior	63

19. Tactile interaction as a system of material relationships	66
20. Procedural materials, shader backends and sensory output	71
21. Rendering efficiency, scheduling and memory	74
22. Assets, lazy loading and resource delivery	77
23. Measuring web, native and custom interfaces	81
24. Dynamic UI through graph updates	83
25. Custom interfaces and specialized engines	86
26. Verification and implementation boundaries	88
 Part III — Applications and implementation	 91
27. BestBuds: one conversation across several presentations	91
28. Blueprint: a dense workbench around a specialized canvas	94
29. Operational software: an overview and an offline field task	96
30. Learning software: explanation beside an interactive model	98
31. Commerce and content: stable presentation around authoritative transactions	100
32. Creative software: a specialized scene within a coherent shell	102
33. Constrained hosts: share capabilities rather than an unsuitable page	104
34. A Compose-first implementation plan	106
References	112

An interface is more than the image on a screen. It is a relationship between a person's intent, the application's state, the controls that express that state, and the platform that makes those controls usable. Reaktor Surface gives these responsibilities explicit boundaries. Its component contracts are defined in Kotlin. Its application capabilities and long-lived state belong to Reaktor Graph. Its two rendering families are Compose Multiplatform and React Strict DOM. Its interaction, layout, motion and tactile feedback execute close to the renderer rather than waiting for an application-wide round trip.

This architecture supports both quiet and expressive interfaces. A form can remain almost still. A social interface can respond through compressible surfaces, neighboring material movement and optional haptics. A workbench can combine dense controls with a specialized canvas. In each case, the component's meaning and the user's work remain independent of its current appearance. Supported subgraph updates change the affected interface through the existing Reaktor execution system; Surface does not introduce a separate update authority.

Part I — The problems and the foundations

1. Why contemporary interfaces become difficult to maintain

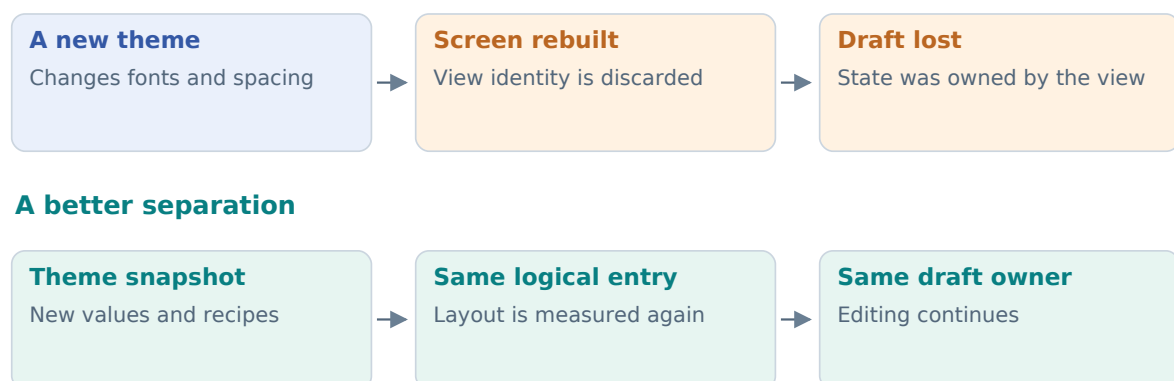
1.1 A small visual change can expose a large architectural problem

Imagine a person writing an event invitation. The title and description are complete, a location is selected, and the message is almost ready to send. The person switches to a dark theme. The screen looks different, but the description disappears. Nothing about a dark theme requires losing text. The failure occurs because changing appearance also replaced the object that owned the draft.

A similar problem appears when a tablet rotates. A single-column screen becomes a two-pane screen, and the application opens the conversation again as though it were new. It may reestablish a connection, reload history, jump to the newest message, and clear the text selection. The layout changed, but the implementation treated it as a change in application identity.

These examples are not arguments against themes or adaptive layouts. They reveal an ownership problem. Several independent decisions have been placed behind one convenient operation called “build the screen.” The operation creates state, chooses layout, acquires resources, installs listeners and draws components. Repeating it therefore repeats more work than intended.

One small change can expose several hidden couplings



Appearance changes. Application identity and retained user work do not.

Figure 1: A theme change should replace presentation values, not the owner of a draft. The upper sequence illustrates an unwanted coupling; the lower sequence separates appearance, layout and retained state.

The first challenge is to identify what is allowed to change independently. A theme may alter colors. A larger text setting may alter geometry. A navigation operation may introduce a new route entry. A graph update may replace a supported feature implementation. None of these operations automatically means that the person’s unfinished work should be discarded.

1.2 A visual component is often less complete than it appears

A button can be drawn with a rounded rectangle, an icon and a label. That makes it recognizable to many sighted pointer users, but it does not define its behavior. A keyboard user needs to reach it. An accessibility service needs a name and an action. A pointer press can begin inside the control and end outside it. A disabled control must not accidentally invoke its operation through an alternative event path.

A custom component therefore carries obligations that are invisible in its default screenshot. A menu must manage focus and dismissal. A text field must cooperate with input methods and editing services. A dialog must define where focus enters and where it returns. WAI-ARIA's guidance emphasizes that assigning a role also makes a behavioral promise; the role itself does not implement that behavior. 1

The architectural problem becomes worse when two platforms independently implement components with similar names. Both may have a Button, a disabled property and a loading property, yet interpret their combinations differently. Shared names are not a shared contract. A design system needs an executable definition of behavior, not only a matching palette.

A button has more than a painted rectangle



One component contract connects these responsibilities without making them the same thing.

Figure 2: The visible button is the meeting point of four responsibilities. Meaning identifies the operation, behavior interprets interaction, appearance expresses the design language, and the graph governs application authority.

1.3 Cross-platform reuse is not the same as identical rendering

A phone, a desktop window and a browser page do not impose the same conditions. A phone may lose a substantial part of its available height when the keyboard appears. A desktop user may expect menus, shortcuts and several windows. A web user may expect an ordinary link to support browser navigation and opening in another tab. Reusing application meaning is valuable; erasing these differences is not.

The difficult question is where to share and where to adapt. Sharing too little produces divergent implementations. Sharing too much can force every platform through the least capable abstraction. A component whose contract assumes only a mouse click cannot naturally describe touch cancellation, keyboard activation or a television's directional focus.

A better boundary shares intent and obligations while allowing the platform implementation to remain competent. "Choose one date" is shared meaning. The exact date picker may differ. "Open this resource"

is shared meaning. A browser realization can preserve link behavior while a native realization uses the application's navigation contract.

Portability consequently needs a scope. It applies to a supported component on a supported renderer with a declared behavior profile. It is not a property that every arbitrary Kotlin or JavaScript function acquires merely because it belongs to a multiplatform project.

1.4 Adaptation has become more than a few screen-width breakpoints

A device category is a poor description of available space. A desktop inspector can be narrower than a phone. A tablet can have a small application window. A foldable display may contain an occlusion. A large text preference may make a two-column arrangement unusable even though the outer width did not change.

Android's adaptive guidance uses the application's available window and canonical layout patterns rather than a permanent device label. CSS container queries provide a corresponding way to respond to a containing region on the web. ²³

The deeper problem is not choosing a threshold. It is preserving meaning while rearranging the interface. When a conversation moves from a full screen to a detail pane, the selected conversation remains the same. When a supporting panel becomes a sheet, its workflow state should remain associated with the same feature. Layout must be a presentation decision over application state, not a second navigation system that continually tries to synchronize itself with the first.

1.5 Responsiveness involves time as well as geometry

An interface can fit the screen and still feel unresponsive. A press may wait for a network result before changing appearance. Typing may stall because each keystroke updates a large shared object. A decorative background may keep rendering while hidden. An image may arrive after the text has been positioned and move the sentence the person is reading.

These failures involve different kinds of work. Some require less composition. Others require fewer layout passes, smaller image decodes, bounded queues or better ownership of animation. Treating all of them as "too many rerenders" conceals the actual cause.

A UI architecture must therefore describe the execution path, not merely the component hierarchy. It must distinguish immediate feedback from application results, and it must make resource use visible. A design language that looks elegant in a static preview may still be unsuitable if its blur, shadows or continuous effects exceed the budget of a supported device.

1.6 Generative interfaces increase the cost of unclear boundaries

A developer usually changes a screen before it is released. A runtime agent can propose a different arrangement while the person is already using it. That creates a new class of continuity problems. A field might be replaced after the person starts typing. A new suggestion might move a primary action beneath the pointer. A generated control might name an operation that the current user is not allowed to perform.

The answer is not to give generated interfaces a separate set of components and permissions. That would multiply the number of systems that must remain consistent. The same catalogue, state owners

and application capabilities should constrain both human-written and agent-composed interfaces. Generation proposes a change to an approved graph-owned region. The graph's checked update mechanism activates it, and the renderer presents the result.

The objective is a system whose variability is controlled rather than suppressed. Themes, adaptation, tactile effects and generation remain powerful, but none is allowed to blur the ownership of application state or the authority to perform an operation.

1.7 Recognize the kind of failure before choosing the remedy

The same visible symptom can have different causes. A control that appears late may be waiting for data, an asset decode, a large composition, or an unnecessary feature initialization. A control that loses its value may have been assigned a new identity, disconnected from its draft owner, or overwritten by a stale acknowledgment. Adding a loading spinner or memoizing the nearest function does not distinguish these cases.

A useful first classification asks whether the failure concerns meaning, ownership, geometry or execution. Meaning concerns what the control communicates and can do. Ownership concerns which object is responsible for state and resources. Geometry concerns the space and position assigned to content. Execution concerns when and at what cost the work happens.

The categories interact, but they suggest different corrections. A missing keyboard action requires a behavior and semantics correction. A lost draft requires a lifetime or identity correction. A jumping message requires a geometry and anchor policy. A slow press requires investigation of the immediate execution path. Surface gives each correction a place instead of treating every defect as another component-specific exception.

2. The concepts that connect application meaning to pixels

2.1 State, events and the difference between them

State describes what is true now. A form contains a title. A conversation has unread messages. A button is currently pressed. **An event** describes something that happened: a title was edited, a message arrived, or a press began.

This distinction matters because the same state may be observed many times. Drawing a button repeatedly while it remains pressed must not produce a new haptic every time. The haptic belongs to the event that began contact, not to the fact that the control is still pressed.

Application state and interaction state also have different owners. The submitted message belongs to the conversation domain. A pointer's current contact belongs to a local interaction controller. Both are state, but one may need to survive a view replacement while the other should disappear when its pointer capture ends.

A component's public inputs describe controlled facts supplied by its owner. Its local state describes the interaction session needed to realize those facts. The component emits events to request meaningful changes; it does not silently redefine the application's records.

2.2 Composition chooses the interface's structure

Composition answers a question such as: given this application state, which components should exist and how should they be nested? A conversation page may contain a header, a timeline and a message composer. An empty timeline may contain explanatory text instead of message rows. A feature setting may permit an event-planning region.

Kotlin composables and React components are programming models for expressing these choices. They provide ordinary control flow and reusable functions. A loop over messages is still a loop; a conditional region is still a conditional expression. Surface does not need to invent a second general programming language merely to describe those operations.

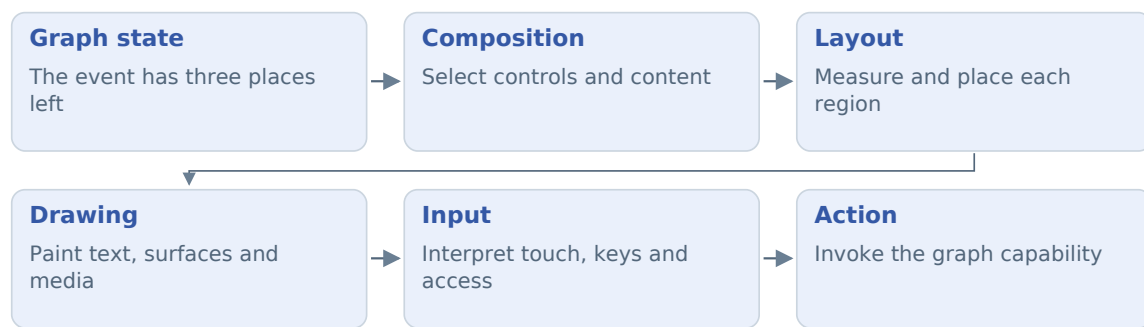
The result of composition is not the final image. Text still needs measurement, containers still need placement, and the renderer still needs to draw. React's own explanation separates the rendering calculation from the commit that applies changes. Compose likewise distinguishes composition from layout and drawing. 45

2.3 Layout assigns space; drawing realizes appearance

Layout determines the size and position of each region. A parent offers constraints, such as a maximum width. A child measures its content within those constraints and reports a size. The parent then places it. Text, media aspect ratios and intrinsic sizing can influence the result.

Drawing uses the resulting geometry. It paints text, images, borders and surfaces, possibly into composited layers. A decorative translation can sometimes be applied without changing the space allocated

From application state to visible feedback



The next application result starts another update; animation need not wait for that result.

Figure 3: Application data becomes a component structure, then measured geometry and drawing. Input returns through an explicit behavioral path. The arrangement is a conceptual pipeline; an implementation can skip stages whose dependencies did not change.

by layout. Expanding a paragraph, by contrast, changes the space that following content needs.

Keeping these stages separate explains an important performance choice. Moving a button face down by a few pixels to suggest contact should not ordinarily cause the entire form to measure again. Changing the font size legitimately does require new text measurement. Efficient execution means invalidating the work that depends on a change—not denying that the change has consequences.

2.4 A design system supplies decisions, not only components

A component library provides reusable controls. A design system also explains the choices behind them: what different text roles mean, which actions receive emphasis, how errors are expressed, how much spacing belongs between related elements, and how movement communicates change.

Tokens give names to reusable values. A reference token might name a palette color. A semantic token names a purpose, such as primary-action background. A **recipe** explains how a particular component uses those roles in its variants and states.

This separation makes a design system themeable. A primary action does not need to know the hexadecimal color assigned by a brand. It needs the semantic role that the selected design package resolves. The same control can then participate in several visual languages without duplicating the operation it invokes.

2.5 A lifecycle describes an owner’s existence

A lifecycle is not a synonym for “visible.” A conversation session can exist while its pane is hidden. A view can remain visible briefly during an exit animation after it has stopped accepting new actions. A texture can be released even though the document that used it remains open.

Lifecycle design answers when an object starts owning resources, when it can perform work, and when that ownership ends. The useful unit is the owner, not a universal list of lifecycle events copied into every class.

For example, a route-entry owner can retain a draft. A mounted renderer session can own a focus target and a decoded image lease. Hiding the view may pause its effects and release the image while leaving the draft untouched. Destroying the route entry can release its retained presentation state without deleting the underlying conversation record.

2.6 A graph makes relationships explicit

Reaktor Graph represents application structure using nodes, node-owned ports and connections. A node can own a repository, a workflow, a route or another capability. A port exposes a typed value or operation. Connections state which consumers rely on which providers. The graph's runtime also provides application scopes and navigation mechanisms. 6

The graph is useful because an interface does not operate in isolation. A Send action depends on a conversation operation. A participant picker depends on permitted people. A theme preference depends on a setting owner. These relationships can be explicit instead of being rediscovered through globals inside every control.

Graph-native UI does not require a graph node for every painted shape. A whole screen or reusable feature region can be owned by a graph node while its internal text and layout remain normal renderer structures. The application graph, the component hierarchy and the drawing scene answer different questions.

2.7 Contracts are promises; implementations are mechanisms

A contract states what a caller may rely on. An implementation states how that promise is achieved. “This control emits one activation for an admitted semantic action” is a contract. “This implementation uses a native button and observes its interaction source” is a mechanism.

This distinction makes reuse more precise. Two implementations may satisfy the same promise using different mechanisms. An implementation change is safe only to the extent that the required promises still hold. Reusing an interface name does not establish that equivalence; tests and qualification provide evidence.

A boundary also needs failure behavior. A requested image may be unavailable. A component version may be unsupported. A layout may not fit. Returning a structured finding allows the surrounding owner to choose a known fallback or alternative. Silently drawing an empty region makes the failure harder to diagnose and can remove an essential operation.

The same reasoning applies to optional effects. “This action remains usable without haptics” is a meaningful guarantee. “Every target has a vibrator” is an assumption the component should not require. Contracts describe what the application needs while capabilities describe what the current realization can supply.

3. The load-bearing ideas

3.1 Share meaning; adapt realization

The central reusable object is the **component contract**. It defines inputs, outputs, slots and behavioral obligations. An implementation realizes that contract using a renderer. A design recipe determines its appearance within a supported design language.

For a button, the contract includes an accessible name and an activation event. The recipe may produce a filled Material-like treatment or a hard-edged contact plate. The Compose and RSD implementations need not draw identical pixels, but they must agree on whether the button can activate and what event activation produces.

This rule prevents design language from becoming business logic. A destructive action remains destructive in every palette. A disabled operation cannot become enabled because an alternative skin omitted the disabled check. A transparent surface cannot make its label optional to assistive technologies.

The practical consequence is that tests are shared at the semantic level. Each renderer runs equivalent interaction traces, while visual comparisons permit declared platform differences. A font's metrics may differ. Losing the component's name or invoking its operation twice is not an acceptable rendering difference.

3.2 Give each structure and state an owner

A React boundary owns its React composition. A Compose boundary owns its Compose composition. A graph workflow owns the application state that must survive presentation changes. A renderer session owns immediate editing and physical feedback.

Ownership prevents competing writers. A visual editor cannot independently mutate the inside of a live React component while React continues producing a different tree. An agent submits a change through the owner. A parent can mount a child, supply its declared inputs and receive its events, but does not acquire arbitrary mutation rights over the child's internal state.

The same principle applies to resources. An image lease has an owner that can release it. A field registration ends when the participating component leaves its scope. A callback belongs to one runtime generation, so an old callback cannot accidentally resolve to a new control that reused its numeric slot.

Explicit ownership is also what makes the system understandable. When a draft disappears, the relevant question is not "which framework is broken?" It is "which owner was replaced, and why was the draft attached to it?"

3.3 Keep the fast feedback loop local

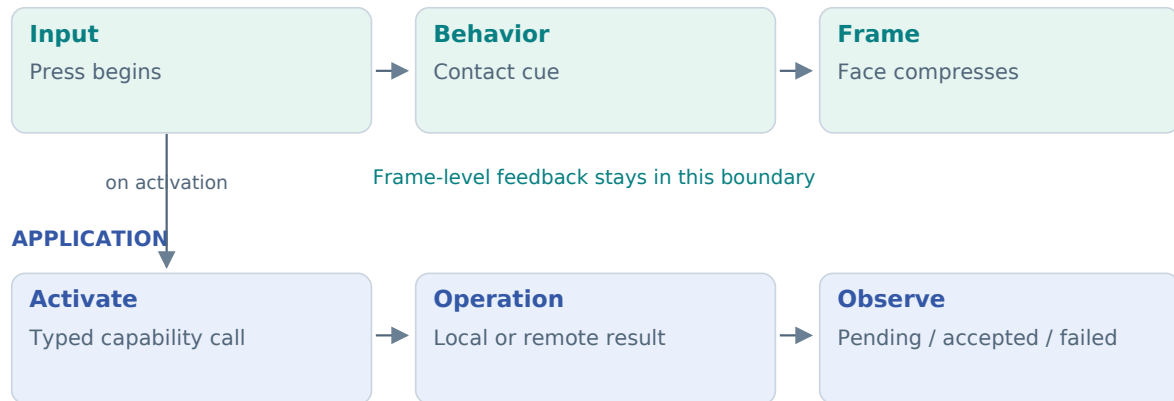
A person should feel contact before a remote operation finishes. Press compression, local focus treatment and immediate editing therefore run inside the local presentation boundary. They consume already-bound capabilities and local input rather than traversing the application graph each frame.

Meaningful operations still return through the graph. A button's activation may send a message or change a route. The resulting state is observed by the affected UI scope. This separation allows the

control to be responsive without pretending that the operation has already succeeded.

Two loops operate at different timescales

LOCAL PRESENTATION



Contact feedback is immediate. Success feedback waits for an authoritative result.

Figure 4: The presentation loop handles contact and motion locally. The application loop invokes a capability and observes its result. A renderer can provide immediate feedback during latency without misrepresenting the outcome.

The rule applies to both renderer families. It does not mean that every animation is written in Kotlin on every target. It means that frame-level work stays in the runtime best placed to execute it. A CMP implementation can use Compose’s animation and drawing paths; an RSD realization uses its qualified platform mechanisms.

3.4 Treat adaptation as a projection of existing state

An adaptive layout chooses how much of the interface is visible at once. It does not decide what the conversation means or create a second set of route entries. A presentation policy reads navigation state, environment and preferences, then assigns entries to panes.

A compact window may show the conversation alone. An expanded one may show its list and supporting details. The same route-entry and workflow owners remain. This is why the layout policy can be a pure function while the stateful navigation owner stays in the graph.

Theme changes follow an analogous rule. A new theme snapshot changes recipes and, where necessary, measurements. It does not automatically create a new route or a new draft owner. When a structural implementation change cannot preserve transient state directly, the boundary declares what must be transferred or restored.

3.5 Make tactility causal and bounded

Tactility means that the interface responds as though its elements have material relationships. A press has a location and a duration. The button face may move toward a stationary support. A neighboring card may respond to the same impulse. A procedural background may reveal a wave-like disturbance.

The relationship is causal: a particular input produces a particular response. It is also bounded: the effect has a scope, maximum amplitude and lifetime. A receiver does not automatically emit another input that starts an endless chain. A quiet theme can reduce the response without removing the component's meaning.

This model is richer than attaching unrelated animations to controls. The components and background share an interaction signal, but remain decoupled from each other's implementation. A card does not call a shader function directly. Both consume the field within their local scope.

3.6 Define dynamic UI as graph-owned change

A dynamic interface can change its content, arrangement or supported implementation. These changes still belong to an application owner. Surface assumes that Reaktor can activate supported subgraph updates using the existing graph, FFI and Hermes infrastructure.

The UI-facing contract is intentionally smaller than the update system. After a compatible subgraph is active, affected Surface bindings observe the new definition or state. The relevant renderer updates the region. Unchanged owners retain their state, and retired callbacks cannot enter the new generation.

Human authors and agents therefore use one change model. An agent does not directly install a private UI runtime beside the application. It proposes a change to an approved region whose capabilities and component catalogue are already constrained by the graph.

4. High-level architecture

Reaktor Surface: graph-owned meaning, local presentation

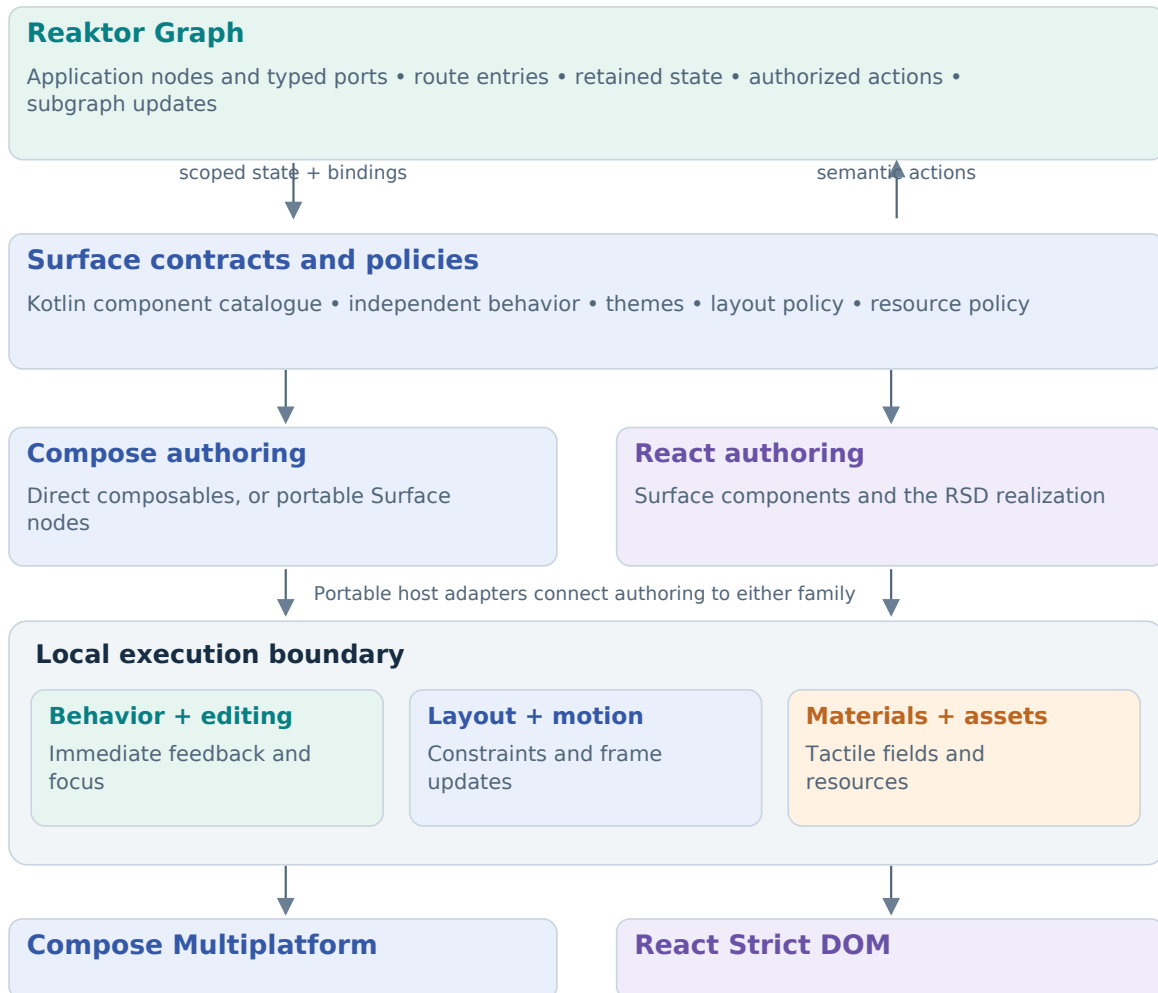


Figure 5: The graph provides ownership, state and application capabilities. Surface adds component contracts and presentation policies. Local execution realizes those contracts through Compose Multiplatform or React Strict DOM, with optional portable host adapters between authoring and realization.

4.1 The graph is the application foundation

Every running Surface application has Reaktor Graph. A Surface boundary attaches to an application owner and receives a checked view of its state and operations. It does not create another dependency-injection container, router or authorization service inside the design system.

This assumption simplifies component composition. A feature can declare the capabilities it consumes, and an inspector can explain those bindings. A child control does not need the entire graph object merely to invoke one operation. Generated or handwritten adapters can expose a small typed action interface.

For a conversation, that interface might contain `send`, `retry` and `openProfile`. The renderer sees the operations appropriate to the screen. The graph remains responsible for finding their implementations and enforcing authoritative policy. An in-process call need not be serialized simply because it travels between graph-owned capabilities.

4.2 Surface owns presentation contracts

Surface's shared definitions describe component meaning, theme roles, behavior policies, layout decisions and resource requirements. They are independent of Compose UI classes and DOM objects. A component can therefore be described before either renderer implementation is chosen.

The catalogue connects each definition to its implementations and tests. The theme resolver produces a coherent appearance snapshot. The presentation policy assigns route entries to visible regions. Together, these mechanisms determine what a mount needs to render without taking ownership of the application's business records.

A mount is one attachment of a Surface implementation to a host, such as a window or a region within a screen. It has an address and a lifetime. Several mounts can present the same business object while retaining separate focus, scroll and editing-session details.

4.3 The renderer owns platform realization

The renderer translates neutral meaning into platform behavior and drawing. For CMP, it can reuse Compose Foundation, Material components and platform integrations. For RSD, it uses the supported React Strict DOM vocabulary and the behavior of its web or native realization. RSD itself is a React-oriented substrate for sharing styled components across web and native; it is not the custom React-to-Compose reconciler. ⁷

The renderer also supplies measured constraints, input normalization, resource handles and frame timing. It knows how local logical coordinates map into actual drawing space. Shared algorithms can use those services without assuming a particular window toolkit or graphics backend.

Capability differences remain explicit. A renderer may support a component fully, support a declared reduced treatment, or report it unavailable. An opaque surface is a reasonable fallback for unavailable blur. Omitting an essential consent control is not.

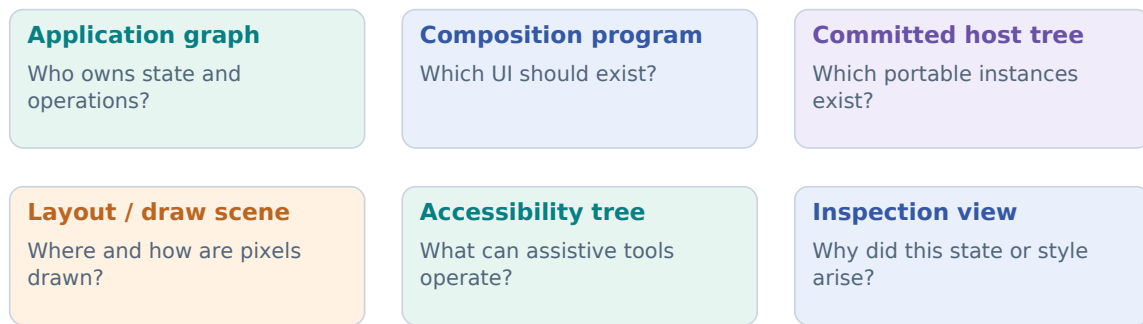
4.4 Different representations answer different questions

The application graph answers who owns state and operations. The composition program answers which controls should exist. A portable host tree, when used, records the committed instances produced by that program. Layout and drawing structures describe geometry and rendering. The accessibility tree describes semantic operation.

A drawing layer can be absent from the accessibility tree because it is decorative. A single semantic button can contain several visual layers. A graph-owned screen can contain thousands of virtualized item instances without giving each item its own actor and application scope.

Inspection relates these representations through stable identities and source mappings. It does not claim that the rendered output contains enough information to reconstruct every source-level loop or conditional. A visual editor changes supported declarations or asks the source owner to change code.

One application, several purpose-specific representations



A route node is not a button instance; a button instance is not a painted layer.

Figure 6: The architecture contains several related representations because each has a different job. None is allowed to silently become a competing owner of the application state represented elsewhere.

4.5 Compile decisions that do not need to remain dynamic

Token aliases, static component definitions and supported recipe variants can be validated before rendering. The runtime then consumes compact resolved values rather than repeatedly interpreting the same authoring document.

Other inputs remain live. Application state changes. A window is resized. A person presses a control. Surface binds these changing values at the smallest useful scope. It does not rerun every graph analysis whenever a pointer moves.

The distinction is similar to preparing a route before using it. The binding step determines which operation the Send action refers to. The next press uses that binding; it does not repeat global discovery. Authorization is still checked at the operation's authoritative boundary, because an old binding is not permanent permission.

4.6 Read the architecture through one change

Suppose a setting increases the application's text scale. The application owner records the preference. The affected Surface environment changes. Recipes resolve the required text roles, and layout measures the content again. A supporting pane may collapse because the larger text needs more room. A collection restores the item being read after remeasurement.

Notice which owners did not change. The conversation record is still the conversation record. The draft is still associated with the same workflow. The navigation entry is not replaced just because the pane moved. Only resources and cached results that depend on the changed typography or constraints become invalid.

This provides a practical way to use the HLD. Begin with the changed input, identify its owner, follow the dependencies that consume it, and stop where those dependencies end. An architecture is easier to understand when its arrows explain the consequences of a concrete event rather than merely connecting every subsystem to every other subsystem.

5. Platforms, composition languages and rendering families

5.1 Two terminal families

Compose Multiplatform and React Strict DOM are the terminal realization families. CMP is the first production target. It supports Kotlin-first component authoring and direct access to the Compose ecosystem. RSD is the later React-oriented realization for web and qualified React Native hosts.

A renderer family is not the same as a single platform. Each family still has target-specific behavior, input, graphics and lifecycle concerns. CMP integration with platform facilities is distinct from claiming that every control is an operating-system widget. The documented relationship between Compose Multiplatform and Jetpack Compose provides the relevant toolkit boundary. 8

Platforms share meaning, but not every interaction mechanism

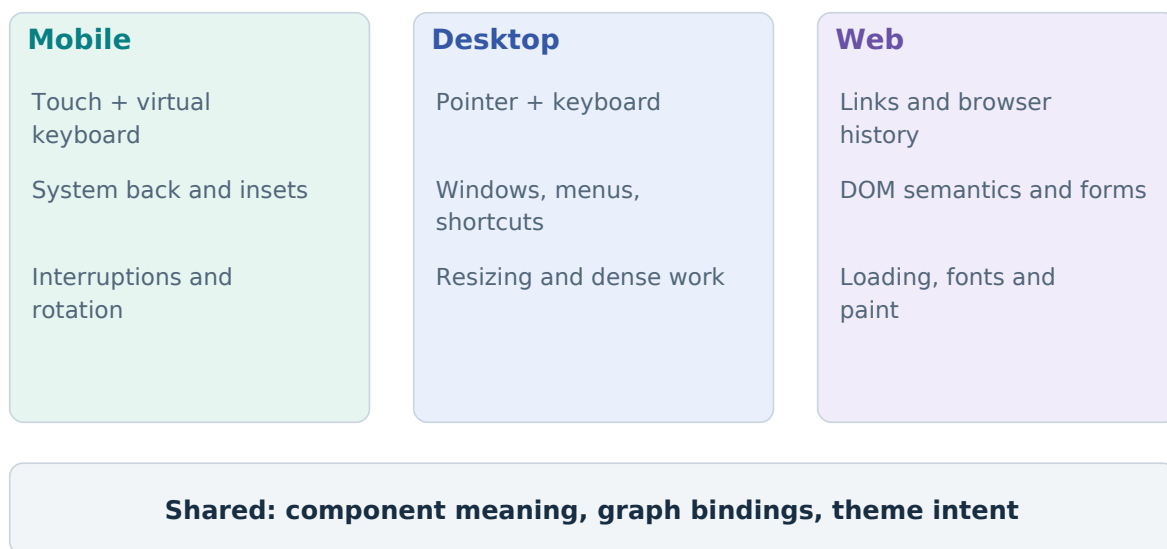


Figure 7: Platform profiles identify the conditions an implementation must handle. The shared design vocabulary does not erase different navigation, input, lifecycle and delivery expectations.

The shared API is designed so that adding the RSD renderer does not require changing what a button or field means. It will still require platform implementation work. Minimal architectural change is not zero engineering work.

5.2 Authoring and realization are separate choices

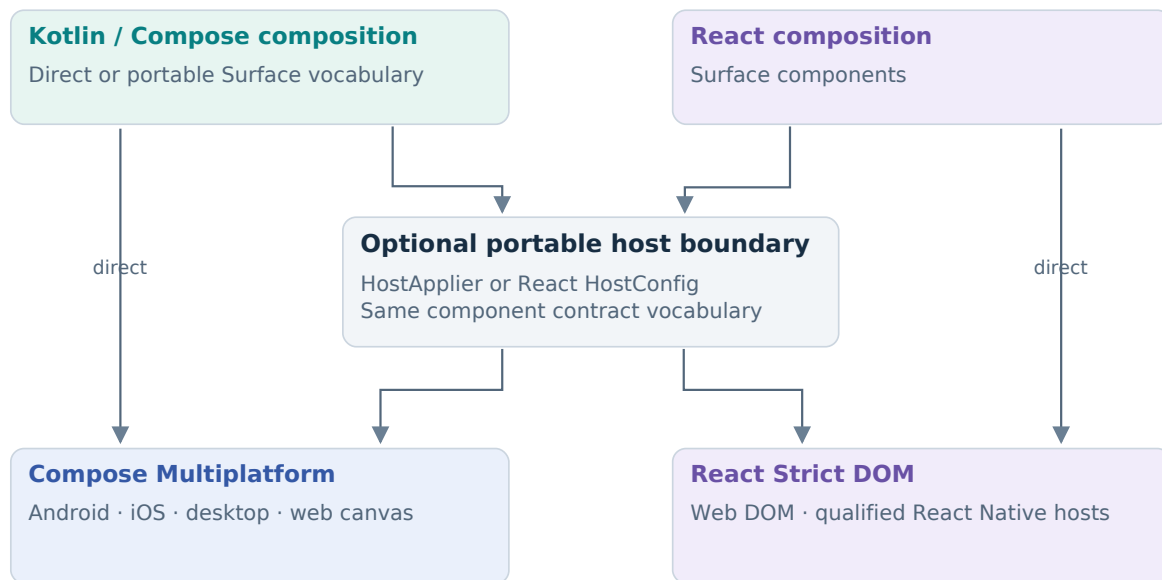
A developer can write Kotlin composables that directly use the CMP implementation. This is the shortest path for the first application. It avoids a portable host tree and cross-runtime transport when neither is needed.

A developer can later write React components using the RSD implementation. The same shared contracts

generate TypeScript-facing props and event types. Common behavior can be compiled to JavaScript or implemented through a qualified adapter with the same semantic traces.

Two additional interoperability paths are possible. React can emit Surface host operations that a CMP renderer consumes. Portable Surface composables can emit a host structure through a custom Applier and let RSD realize it. These are adapters between authoring and realization, not additional renderer families.

Two renderer families accommodate several authoring paths



A host adapter is not a third renderer and is not required for every direct Compose screen.

Figure 8: Direct Kotlin-to-CMP and React-to-RSD paths remain efficient defaults. Portable host adapters add cross-authoring options without becoming mandatory for every screen.

5.3 Portable composition is an explicit vocabulary

A composable function is not automatically portable to any possible tree. Ordinary Compose UI functions create the kinds of nodes expected by Compose UI's own machinery. A custom Surface Applier creates Surface host nodes instead. Only functions written for that portable vocabulary, or explicit compatible islands, belong inside that composition.

This matters for honest API design. A public `SurfaceButton` can have a CMP implementation and a portable host declaration. An arbitrary third-party Compose canvas cannot be promised to appear in a browser DOM merely because it was called from Kotlin.

The restriction preserves expert implementations. A specialized editor can remain a native island on CMP. A corresponding RSD profile can supply another editor implementation or a deliberate unavailable result. The component contract specifies the boundary; it does not require translating the editor's internal rendering code.

5.4 Platform differences should be explained at the boundary

A portable date value, a logical length and an asset identity are useful shared values. A platform image object or DOM element is not. Local adapters convert the shared values into the objects their toolkit requires.

Behavior follows the same pattern. The shared button machine recognizes focus, contact, cancellation and activation. A platform adapter decides how its native input events become those normalized inputs. This allows the implementation to preserve platform conventions without putting operating-system checks inside every shared reducer.

The architecture consequently combines a small shared meaning with rich local realization. It avoids two opposite failures: duplicating the entire design system per platform, and forcing every platform into an abstraction too weak to express its normal behavior.

5.5 Portability has a useful floor and explicit extensions

The portable floor contains the concepts that can be given reliable shared meaning: text, actions, fields, selection, layout relationships, resources and application bindings. A product can build a substantial interface from that floor without knowing the internals of either renderer.

Extensions preserve the cases that do not fit the floor. A map may need a native engine. A scientific model may need custom drawing. A rich editor may need its own text services. The extension exposes the boundary required for sizing, actions, lifecycle and inspection, while keeping its internal machinery intact.

This is different from a least-common-denominator design. The shared layer does not forbid advanced features; it requires those features to declare where they work and how the application behaves when they do not. A supported native picker can coexist with a supported web picker. The application asks for a value under a contract rather than insisting that every platform reproduce one platform's internal widget structure.

6. A worked example: one Send action through the whole system

6.1 Before interaction

The conversation owner exposes a presentation model and a typed action interface. The model contains the draft and the state needed to show whether sending is available. The Surface mount subscribes to that model and resolves the selected theme.

Composition creates a message composer organism. The organism contains an editing field, an attachment control and a Send button. The button receives controlled properties such as its label, emphasis, enabled state and busy state. It does not fetch the conversation repository itself.

The selected recipe determines the button's visual treatment. In a quiet desktop language it may be a restrained solid control. In an expressive language it may have a moving face above a fixed contact plate. Both implementations expose the same semantic action.

6.2 Contact begins

The renderer receives a pointer press or the equivalent platform input. Its adapter normalizes the event and sends it to the independent behavior machine. The machine records the active press and emits a contact cue.

The local motion driver responds to that cue. The face moves toward the support. If the recipe permits it, one field impulse is emitted into the composer's effect scope. The surrounding plate and background can respond without calling the conversation service.

An optional haptic acknowledges contact. It does not mean that the message was sent. The cue is executed once by the effect path; it is not replayed when the pressed state is drawn again.

6.3 Activation is admitted

The platform's semantic activation path produces one normalized activation. The behavior machine checks current controlled availability and emits a typed button event. The composition owner translates it into the bound conversation operation.

The graph's application boundary checks the current principal, conversation and operation arguments. It then performs the send workflow. A local pending message may appear immediately if the workflow supports optimistic delivery, but that state is owned by the application and is labelled as pending.

The control does not infer delivery from the completion of its spring. A failed send can return a recoverable state with retry and discard operations. A successful send can update the timeline and, if appropriate, produce a separate result cue.

6.4 The layout changes during the operation

Suppose the window becomes wider while the send is pending. The presentation policy now has room for the conversation list beside the timeline. It changes pane assignment but preserves the route entry and conversation owner.

The pending operation continues. The draft's declared state survives. Local focus and scroll restoration are handled by the relevant mount. No second send occurs, because rendering is not the action path.

Suppose instead that a supported subgraph update adds an event summary to the conversation. The graph activates the new region. Surface rebinds and renders that region through CMP or RSD. The new region does not acquire arbitrary authority over the draft.

6.5 Why this example is the architectural test

The example combines the most important boundaries: application state, independent behavior, atomic composition, theming, tactile feedback, layout, authorized operations and dynamic change. Each subsystem performs one job and exposes its connection to the others.

The result is not merely an animated button. It is a component whose behavior remains intelligible while its appearance, context and implementation vary. The same reasoning applies to a form submission, a node-editor command or an offline evidence capture.

The foundational rule is to preserve meaning and ownership while allowing presentation to change. The rest of the architecture turns that rule into interfaces that can be implemented and tested independently.

Part II — The framework and its subsystem contracts

7. The small set of interfaces that holds the system together

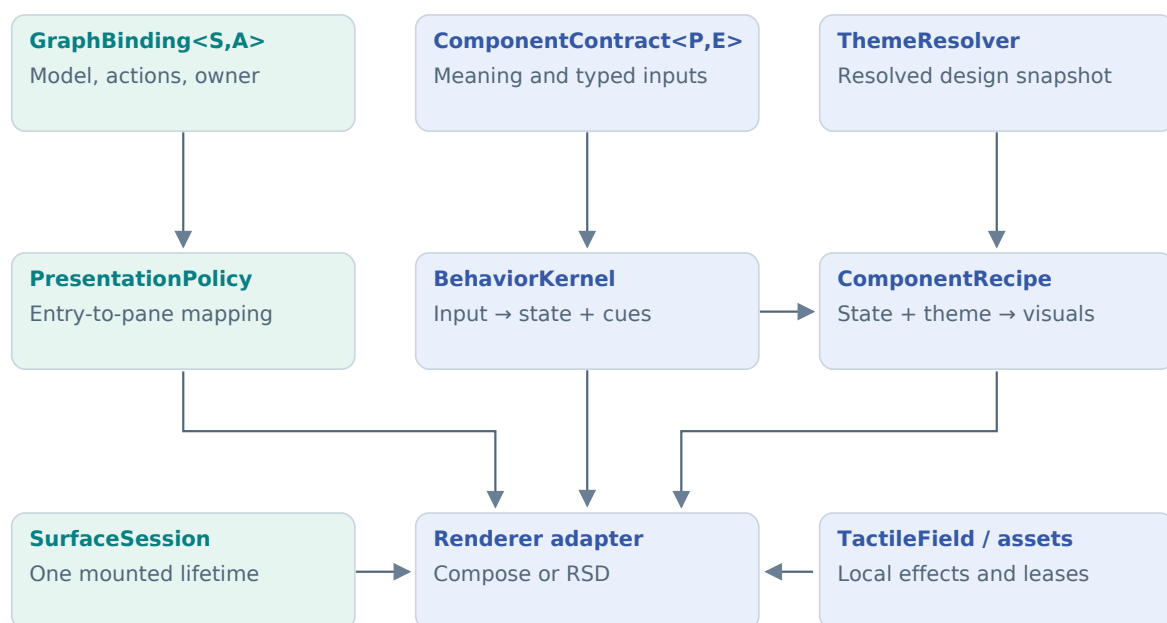
7.1 An interface is useful when it marks a real boundary

An architecture can become harder to understand when every implementation detail receives a new interface. The purpose of Surface's core interfaces is narrower. Each one separates responsibilities that need different owners, different implementations, or independent tests.

`ComponentContract` separates the meaning of a component from its realization. `BehaviorKernel` separates interaction decisions from toolkit event handling. `GraphBinding` separates a screen's required application capabilities from unrestricted access to the application graph. `PresentationPolicy` separates pane arrangement from navigation ownership. `ComponentRecipe` separates appearance from behavior. A renderer adapter brings these results together without becoming the owner of all of them.

Other boundaries concern execution. A `SurfaceSession` owns one mounted lifetime. A `HostCommitSink` accepts portable structure when a cross-authoring path requires it. `TactileField` coordinates local physical response. Resource and observation interfaces control assets and measurements. These are not independent frameworks; they are the seams between the parts of one framework.

The load-bearing interfaces and their connections



Portable host commits are another adapter boundary—not a new application model.

Figure 9: The central interfaces form a dependency structure. Application bindings, semantic contracts and theme resolution are inputs to local presentation. Rendering consumes their results; it does not replace their owners.

The interfaces below are architectural contracts. Concrete production packages can refine their naming and allocation strategy while retaining the ownership and behavioral rules. Platform-specific annota-

tions and objects belong in adapters, not in the shared signatures.

7.2 Component meaning: `ComponentContract`

A component definition describes the values accepted by a component and the events it can emit. The type parameters separate properties `P` from events `E`. A field's current value is a property; a requested edit is an event. A button's enabled state is a property; activation is an event.

```
interface ComponentContract<P : Any, E : Any> {
    val id: ComponentId
    val majorVersion: Int
    val properties: ValueCodec<P>
    val events: ValueCodec<E>
    val slots: List<SlotContract>
    val semantics: SemanticObligations
    val behaviorVersion: Int
}
```

A stable component identity allows generated tools and renderers to agree on the same component. The version identifies a compatible contract family. The codecs describe boundary values, but using the component locally does not require encoding them. Direct Compose calls can pass typed Kotlin values without a byte buffer.

Slots describe the component's composition surface. Semantic obligations describe requirements such as an accessible name or a supported activation action. The behavior version records changes whose significance cannot be captured by field shape alone. A field that changes when it commits a value can break an application even if its property names remain unchanged.

7.3 Toolkit-independent behavior: `BehaviorKernel`

The behavior kernel is a pure state transition. It receives controlled properties, current local state and normalized input. It returns the next state, semantic events and feedback cues.

```
interface BehaviorKernel<P : Any, S : Any, I : Any, E : Any> {
    fun initial(properties: P): S
    fun reduce(properties: P, state: S, input: I): Reduction<S, E>
}

data class Reduction<S, E> (
    val state: S,
    val events: List<E> = emptyList(),
    val cues: List<FeedbackCue> = emptyList(),
)
```

The kernel does not draw, start coroutines, call services or vibrate the device. It describes what should happen. The adapter decides how to execute its outputs in the owning runtime. This makes it possible to test a press-and-cancel sequence without launching a window.

Pure reduction does not mean every subsystem has to copy a large immutable object every frame. The public contract defines observable behavior. Hot implementations may use specialized storage internally, provided they preserve that behavior and expose coherent snapshots to tests and observers.

7.4 Application integration: GraphBinding

A mounted feature needs a view of application state and a set of operations. It should not discover those operations through arbitrary global lookups inside its atoms.

```
interface GraphBinding<S : Any, A : Any> {  
    val address: SurfaceAddress  
    val generation: Generation  
    val model: StateFlow<Versioned<S>>  
    val actions: A  
}
```

S is a presentation model derived from a graph-owned capability. A is a typed action interface, such as conversation operations. SurfaceAddress identifies the owner, optional route entry and mount. generation identifies the active implementation binding, not the user's message revision or the selected theme.

The binding is a projection of existing graph-owned ports. It is not another service registry. Its model remains observable, and its actions still enter the application's authoritative checks. A permission being valid when the binding was created does not authorize every future invocation indefinitely.

A versioned model is useful when several values must be read coherently. The title, selected item and available actions can refer to one application revision rather than exposing intermediate combinations from a transaction.

7.5 Appearance and arrangement

ThemeResolver turns a selection and environment into one checked snapshot. ComponentRecipe turns the relevant properties, local state and theme into an appearance specification. The appearance specification is neutral data or a qualified local extension; it is not a hidden route mutation.

```
interface ThemeResolver {  
    fun resolve(  
        selection: ThemeSelection,  
        environment: SurfaceEnvironment,  
    ): Check<ThemeSnapshot>  
}  
  
interface ComponentRecipe<P : Any, S : Any, V : Any> {  
    fun resolve(properties: P, state: S, theme: ThemeSnapshot): V  
}
```

Layout has a separate contract. PresentationPolicy reads navigation and constraints, then returns entry-to-pane assignments. It does not execute navigation commands during measurement.

```
interface PresentationPolicy<N : Any, U : Any> {
    fun plan(
        navigation: N,
        constraints: LayoutConstraints,
        environment: SurfaceEnvironment,
        preferences: U,
    ): PresentationPlan
}
```

The type `U` contains presentation preferences such as a desired inspector width. Keeping it separate from navigation prevents an ordinary pane resize from becoming a route change.

7.6 Execution and resource boundaries

A `SurfaceSession` represents one mounted presentation lifetime. It can become visible, focused or interactive, and it releases owned resources when closed. Those attributes are separate because an exiting view may still be visible but no longer interactive.

```
interface SurfaceSession : ResourceLease {
    val address: SurfaceAddress
    fun present(visibility: Visibility)
}

data class Visibility(
    val visible: Boolean,
    val focused: Boolean,
    val interactive: Boolean,
)
```

A resource lease establishes who must release an acquired object. It can wrap a decoded image, a native editor, an effect-field registration or a subscription. Closing a lease ends that ownership; whether an underlying object enters a bounded cache is a separate resource policy.

The remaining interfaces are introduced with their subsystems. They deliberately do not form one giant `SurfaceManager` containing navigation, rendering, physics, files, agents and networking. The central connection is the mounted boundary, which receives already-resolved dependencies and coordinates their lifetimes.

7.7 Renderer capability and the portable realization seam

The two renderer families expose a capability profile. It identifies supported component versions and the optional facilities available to a mount. The profile is metadata used for compatibility and fallback decisions; it is not a generic drawing API that hides every platform feature.

```
data class RendererProfile(  
    val id: String,  
    val components: Map<ComponentId, IntRange>,  
    val capabilities: Set<String>,  
)  
  
interface HostRenderTarget : SurfaceSession {  
    val generation: Generation  
    fun render(root: HostNode): Check<Unit>  
}  
  
interface HostRenderer {  
    val profile: RendererProfile  
    fun attach(  
        address: SurfaceAddress,  
        generation: Generation,  
        environment: StateFlow<SurfaceEnvironment>,  
        theme: StateFlow<ThemeSnapshot>,  
    ): HostRenderTarget  
}
```

HostRenderer is used when a portable committed structure actually needs a realization. HostCommitSink validates structural publication first. The target then performs the toolkit work on the required thread. A rendering failure is handled by the mounted boundary's fallback; it is not permission to publish an unrelated partially valid application definition.

Direct CMP composition does not need to call this host interface for every control. Its renderer-local API can use composable functions. The RSD implementation uses React's component model. Their common obligations come from the catalogue and capability profile, while their local function signatures remain appropriate to their toolkit.

This distinction avoids a misleading universal method that takes arbitrary platform content as Any. Portable values have codecs and contracts. Local implementation objects remain local. A typed graph binding supplies application actions in either case.

The result is a small common architecture with two honest realization APIs. The shared layer states what must agree. Each renderer retains the mechanisms it needs to satisfy those requirements well.

8. Component definitions and the catalogue

8.1 Three different things are often called a schema

A **component contract** describes a component's interface. A **composition program** uses components to create a screen. A **host message schema** describes the values crossing a runtime boundary. Treating all three as one schema causes unnecessary restrictions.

For example, the contract for an event card can describe a title, image reference and allowed actions. A Kotlin composition can decide whether to show one card or several. The host protocol can encode the committed card instances. The component contract does not need a new language for loops, and the host protocol does not need to serialize arbitrary closures.

Shared contracts are defined in Kotlin so that types, generation and the broader Reaktor model have one authoritative origin. This does not require every application screen to be generated from data. Handwritten Kotlin and React composition remain normal programming models.

8.2 Properties should describe meaning before presentation mechanics

A useful button property is `intent = Destructive`. It tells every implementation that activation has a destructive meaning. A property such as `shadowOffsetY = 4` tells an implementation how to draw, but says nothing about what the action means.

Both kinds of values can be useful, but they belong at different levels. The public semantic API defaults to intent, emphasis and size. The selected recipe resolves those into fills, text roles, borders and motion. Advanced local customization remains possible without making raw appearance the only language of the component.

```
data class ButtonProps(  
    val label: TextContent,  
    val intent: ActionIntent = ActionIntent.Neutral,  
    val emphasis: Emphasis = Emphasis.Standard,  
    val enabled: Boolean = true,  
    val busy: Boolean = false,  
)
```

Independent booleans are appropriate here because a control's availability and ongoing work are different facts. A button may remain focusable while a submission is busy, even if the interaction policy suppresses another activation. A single enum containing enabled, disabled, loading, error and success cannot represent all valid combinations without growing into a large cross-product.

Component-specific models are preferable to one universal state object. A tri-state checkbox needs checked, unchecked and indeterminate values. A menu needs an active item and expansion state. A text field needs editing and validation information. These contracts can share conventions without pretending their states are identical.

8.3 Portable values require explicit representation

A component contract should not use `Compose Color`, `Dp` or `ImageVector` as universal values. Those types already choose a toolkit representation. The shared contract instead uses color-space-aware values, logical lengths and asset identities. Renderer adapters perform the conversion.

Logical lengths need a stated meaning. A logical layout unit is not automatically a physical pixel. Text sizing must also respect the platform's scaling behavior rather than assuming one linear multiplier is sufficient for every target. A neutral value permits the adapter to make the correct conversion; it does not eliminate that work.

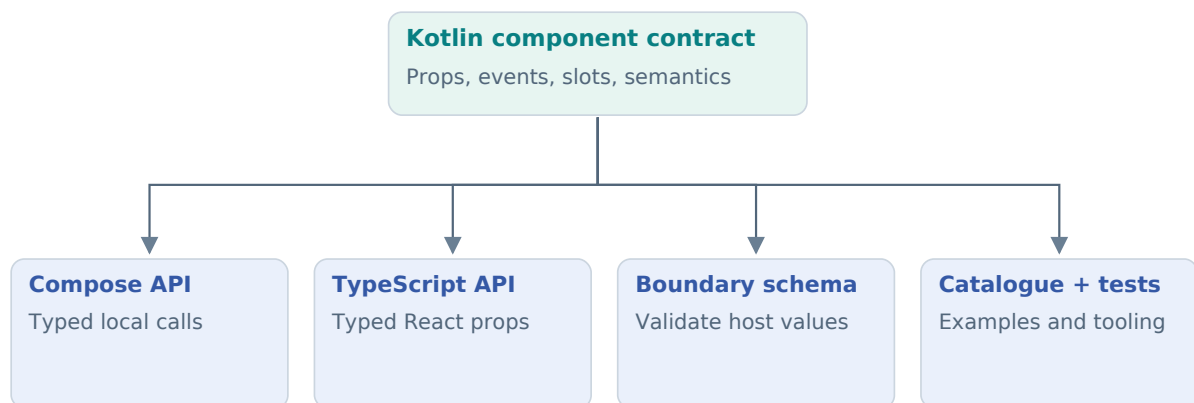
An asset reference identifies an image or icon through the resource system. It can carry semantic alternative text or decorative status through the component contract. A filename is not an accessible name. A decorative icon beside a visible label should not create a second spoken label accidentally.

Local-only slots remain useful. A direct CMP component can accept a composable lambda or a native integration handle. Such values are explicitly local. They are not put into a portable codec and silently expected to work in another runtime.

8.4 The catalogue connects definitions to implementations

The catalogue indexes versioned component contracts, supported implementations, examples and conformance requirements. It is the source used by code generation, the component laboratory, the visual editor and agent tooling.

One definition produces several developer-facing views



Generated projections share a fingerprint; handwritten platform behavior is tested against it.

Figure 10: A Kotlin-defined component contract produces platform-facing types, boundary validation and tooling metadata. Renderer implementations remain responsible for satisfying the shared behavior and semantics.

A generated TypeScript interface should not be a separately maintained interpretation of Kotlin source. The generation step produces a traceable fingerprint. A host can then check whether an incoming region refers to the contract combination it supports.

Code generation should remove duplication without concealing implementation. It can produce property types, codecs and catalogue entries. It should not generate opaque keyboard behavior that no one can reason about. Platform adapters and rich component implementations remain handwritten where that makes behavior clearer.

The catalogue also contains examples for states that are easy to forget: empty content, invalid input, unavailable resources, keyboard focus and reduced motion. A component is not complete because its attractive default state has a preview.

8.5 Compatibility is behavioral

Suppose a field previously emitted an update on each edit but now emits only on focus loss. Its serialized properties may be unchanged, yet an autosave workflow can behave differently. That is a compatibility decision, not merely an implementation optimization.

Similarly, changing whether an Escape key dismisses a panel can affect a surrounding workflow. Removing an accessible action changes operation even when the component still looks identical. Surface versions therefore distinguish data shape from behavioral expectations.

Additive optional values can often be compatible. Required-slot changes, removed events and altered semantic timing require explicit migration or a new contract version. The catalogue records compatibility claims, and conformance tests exercise the behavior those claims depend on.

The goal is not to version every pixel. It is to ensure that application authors can rely on a control's meaning while its implementation improves.

9. Interaction machines that do not depend on Compose or React

9.1 Why behavior belongs below the toolkit adapter

A press can begin, move outside a control, return, and end. A keyboard activation may not involve a pointer at all. An accessibility service can invoke the same semantic action through a different input path. A behavior model must represent the resulting meaning without being written as an assortment of toolkit callbacks.

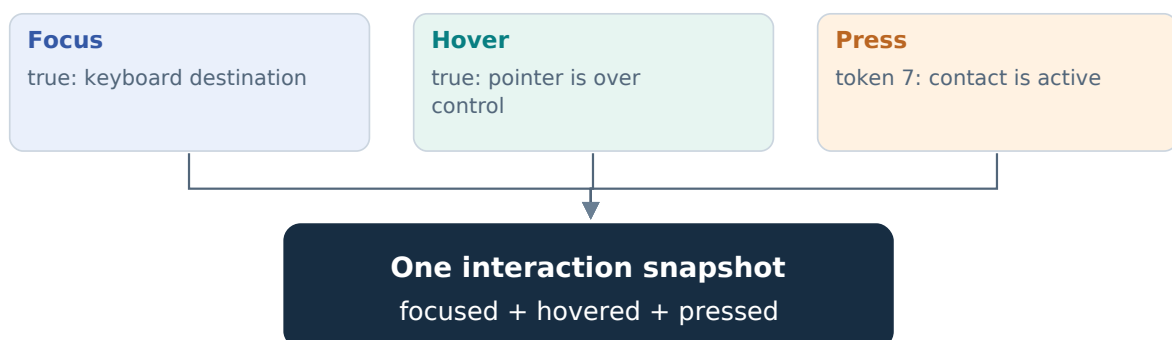
Putting the model in a pure Kotlin kernel has two benefits. First, the behavior can be tested without the rendering system. Second, a future RSD adapter can reuse the same decisions or run the same conformance traces. The rendering backend changes, but the meaning of cancel or activate does not.

The kernel should not implement the entire operating system's input model. It receives normalized inputs from a qualified adapter. The adapter uses native control behavior where appropriate, then exposes the relevant state changes and semantic activation to Surface.

9.2 Focus, hover and press are concurrent axes

A control can be focused, hovered and pressed at the same time. Treating these as mutually exclusive states produces errors: entering the pressed state may accidentally forget focus, and releasing may incorrectly restore an idle state even though the pointer is still over the control.

Focus, hover and press can coexist



Lifecycle ("mounted") and application activity ("sending") are separate state dimensions.

Figure 11: Independent interaction axes can coexist. Creation and disposal belong to a lifecycle; sending or validating belongs to application activity. Neither should be forced into the same mutually exclusive enum as hover and focus.

A button state can therefore contain focus flags, hover state and active press tokens. A drag operation may add a separate phase such as pending, dragging, settling or cancelled. The correct model is a small product of meaningful axes, not an exhaustive enumeration of every combination.

The state must also define what happens when controlled inputs change. Disabling a control during an active press should cancel its admitted contact behavior. It should not wait for a release event that may never arrive after the platform removes the control from interaction.

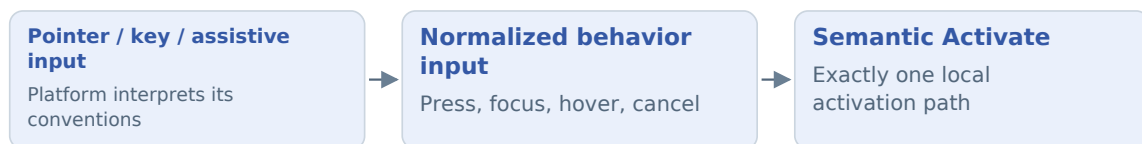
9.3 Separate contact tracking from activation

The reference button kernel recognizes Press, Release, Cancel, Focus, Hover and Activate. Release removes an active contact token. Activate is the only input that produces the button's semantic activation event.

```
sealed interface ButtonInput {
    data class Press(val token: Long) : ButtonInput
    data class Release(val token: Long) : ButtonInput
    data object Cancel : ButtonInput
    data class Focus(val focused: Boolean, val visible: Boolean) : ButtonInput
    data class Hover(val inside: Boolean) : ButtonInput
    data class Activate(val sequence: Long) : ButtonInput
    data object PropertiesChanged : ButtonInput
}
```

This separation avoids a common double-dispatch error. A renderer can observe press interactions for animation while relying on its semantic click path for activation. It must not both synthesize activation on release and also forward the platform's click as a second event.

Normalize input once; activate through one semantic path



A Release updates contact state. It does not also synthesize a second application click.

Figure 12: The platform adapter normalizes its input conventions. Contact state drives physical feedback, while one semantic activation path drives the application event.

The activation sequence is local to the component's admitted event stream. A repeated delivery of the same sequence does not produce another button event. This is not a replacement for server-side idempotency. Network retries and duplicate business requests still require the application's own operation identity and policy.

9.4 The output separates state, events and cues

For a new press, the reducer returns updated contact state and a contact cue. It returns no application event. For release, it removes the token and can return a release cue. For an admitted activation, it returns a typed event that the composition owner maps to a graph action.

The distinction lets different consumers use the result safely. A recipe reads the state to choose the current appearance. A sensory adapter consumes cues to produce one-shot feedback. Application code

handles semantic events. Repeatedly observing the state does not replay either of the event paths.

The reducer itself stays deterministic. The current time, pointer identity or event sequence enters through its input instead of being read from an uncontrolled global clock. Tests can then reproduce a cancellation or simultaneous-contact scenario exactly.

A production controller may retain a bounded queue of cues until the local runtime executes them. It must define how cancellation and disposal affect queued cues. A haptic for a control that no longer exists should not emerge long after its visual context has disappeared.

9.5 Shared behavior does not require identical internal machinery

A web button already has useful native semantics. A CMP implementation can reuse Foundation's interaction facilities. Surface should not replace these mechanisms with a handwritten raw-pointer detector simply to claim all input is "shared." Compose documents interaction sources as a way to observe and respond to component interactions. 9

The shared contract is about the admitted behavior. The adapter maps platform events into that contract without losing normal conventions. Where a platform owns an essential detail, such as text composition or accessibility action dispatch, the adapter delegates rather than reimplements it incompletely.

A headless reducer test proves the reducer. It does not prove that the adapter maps every real platform event correctly. Both layers require tests. The adapter tests should include pointer cancellation, keyboard activation, focus visibility, disabled transitions and assistive invocation.

9.6 A concrete trace

Start with a focused control and no active contacts. A pointer enters, so hover becomes true. Press token 7 begins, so the state now contains focus, hover and token 7. A repeated press notification for token 7 has no additional effect. Release removes token 7 but leaves focus and hover intact.

A subsequent normalized activation sequence 10 produces one event. Delivering sequence 10 again produces none. If the control becomes busy before sequence 11 arrives, the kernel suppresses the next activation under the button's chosen busy policy.

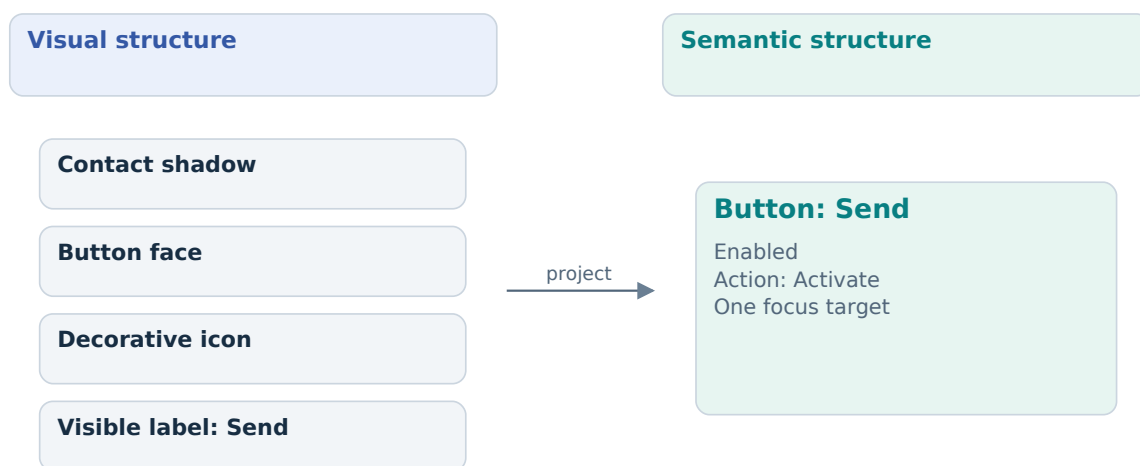
This trace is small enough to understand and useful enough to expose real bugs. It tests independent axes, duplicate contact handling, activation deduplication and controlled-input changes. The same trace can be used when the RSD adapter is introduced, so Compose-first development does not turn behavior into a platform-specific assumption.

10. Input semantics, editing and accessible operation

10.1 A semantic control is not a visual layer

A tactile button can have a contact plate, a moving face, a border, an icon and a label. Those five visual pieces should usually describe one semantic control. If each piece becomes an accessibility node or focus target, the person encounters an interface that does not match its visible meaning.

Visual structure and accessibility structure need not match



Decorative layers do not create duplicate focus targets or duplicate actions.

Figure 13: Several drawing layers can realize one accessible button. The semantic projection carries its name, enabled state and activation action; decorative layers do not create extra operations.

The renderer therefore maintains a semantic projection. It may merge appropriate descendants, expose relationships and attach actions to the correct node. A custom-drawn control requires the same care as a stock control. Drawing an attractive checkbox is not sufficient unless the checked state and toggle action are also operable.

Accessibility requirements belong in the component contract, while platform realization belongs in the adapter. The catalogue can require a name, value and action. The renderer must expose them through its platform's accessibility system and verify them with actual assistive technology.

10.2 Focus is a resource with rules

Focus identifies the destination of relevant keyboard or assistive input. It is not merely a colored outline. Opening a dialog must establish an appropriate initial focus destination. Closing it must return focus to a useful place. Reordering a list should not send focus to an unrelated item because it reused the same position.

Focus also has a scope. Two windows can show the same conversation while only one is the active

keyboard destination. A graph-owned selected conversation does not imply globally shared caret or focus state.

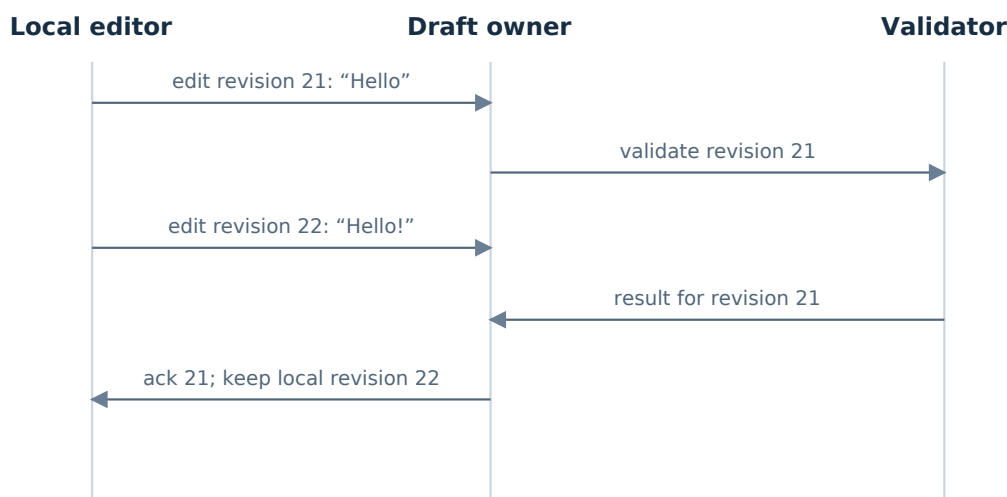
A component distinguishes focus from focus-visible treatment. A platform can provide different focus indication rules for different input modes, but every required interaction must remain perceivable and usable. The visual recipe can change the indicator's style within its contract; it cannot remove the indicator without a conforming alternative.

10.3 Text editing needs a fast local owner

Text editing includes more than a string. It includes caret position, selection, composition ranges, undo behavior and communication with an input method. A renderer must respond locally to these operations. Waiting for a remote validator or application-wide actor before moving the caret would put avoidable latency into every keystroke.

The application still needs ownership of the draft. The solution is not to keep two unrelated copies forever. It is to define a revisioned editing session. The local editor publishes edits, the draft owner acknowledges them, and external updates carry enough identity to avoid overwriting newer local input.

A late state update must not replace newer local text



An acknowledgment advances known persistence; it is not unconditional replacement text.

Figure 14: The editor has already produced revision 22 when validation for revision 21 returns. The old acknowledgment can be recorded without replacing the newer local text.

An acknowledgment means that a particular edit has been received or persisted according to the contract. It is not automatically replacement content. A validator's error for an older revision should not erase a newer valid edit. An external collaborative edit requires a declared merge or conflict policy rather than accidental last-writer behavior.

10.4 Form behavior should preserve work through failure

A form can be initially loading, locally invalid, submitting, rejected by the server, unavailable because of permission, or waiting for connectivity. These are different states with different recovery actions.

Initial loading may need a placeholder. A refresh can keep useful existing values visible. A server rejection should preserve the person's input and identify the affected fields. A permission change may disable submission while allowing a draft to be copied or retained according to product policy.

Submission should not be inferred from a visual transition. A button animation can finish before the server responds. A success message appears only after the authoritative operation reaches the state the message claims. Where an offline workflow accepts work locally, the UI explicitly distinguishes local recording from remote acceptance.

10.5 Internationalization changes geometry and semantics

A translated label can be longer than its source-language version. A right-to-left layout changes logical direction. Different scripts require suitable font coverage and shaping. These conditions affect measurement and behavior, not merely string replacement.

Surface uses start and end relationships in shared layout definitions. A directional message bubble can mirror, while a logo or photograph need not. Text roles refer to resources and typography policy rather than hard-coded line counts that assume one language.

Text scaling also needs deliberate support. A fixed-height field that clips its label is not adaptive merely because the surrounding window is wide. Component measurement must permit legitimate reflow, and the presentation policy may reduce simultaneous panes when larger content needs more space.

10.6 Accessibility policy is independent of agent access

The web accessibility target is WCAG 2.2 AA, evaluated across complete workflows rather than inferred from a component gallery. Native implementations require their corresponding platform tests. WCAG defines testable requirements for web content, but a conforming individual control does not certify every composition that uses it. ¹⁰

Agent observation is a separate policy. A private field may remain fully available to the person's screen reader while being redacted from a remote model's context. Removing accessibility semantics to hide data from an agent would confuse two different trust relationships.

This distinction is especially important for custom interfaces. An inspector can expose selected-object names and permitted commands to an agent without exporting every private property. The same control can provide richer accessibility information locally under the user's own interaction context.

11. Atomic design as a practical composition model

11.1 Reuse begins with recognizable responsibilities

Atomic design provides a vocabulary for moving from small units to complete interfaces: atoms, molecules, organisms, templates and pages. The methodology describes these levels as a way to reason about parts and wholes, not as a rigid sequence that replaces design judgment. 11

In Surface, an atom can be a text element, icon, button or editable field. A molecule combines a few units into one recognizable control, such as a labelled field. An organism represents a substantial interface region, such as a message composer or participant picker. A template describes an arrangement of regions. A page binds that arrangement to an actual product workflow.

Atomic composition: a conversation from reusable parts

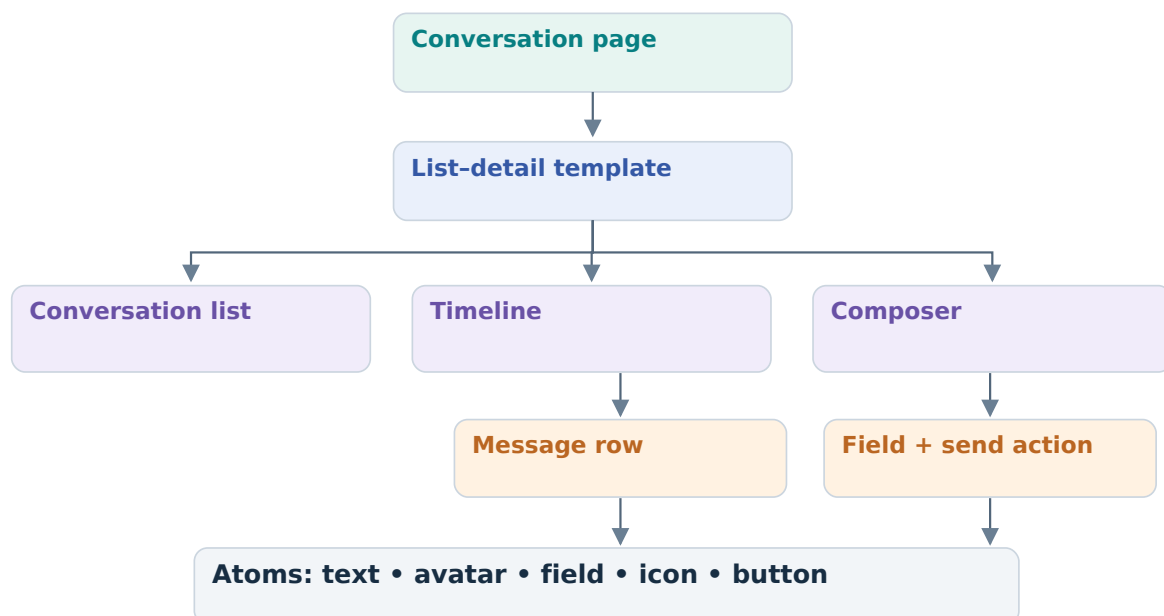


Figure 15: A conversation page combines a layout template with product organisms. The composer and timeline reuse smaller controls, but remain understandable product concepts rather than arbitrary wrappers.

The labels do not determine implementation complexity. A button atom can have a sophisticated behavior contract. A large organism can be stateless and easy to test. Neither automatically requires its own graph node or deployment boundary.

11.2 Product meaning belongs at the appropriate level

The framework can define a generic chip. BestBuds can define a campaign-stage indicator using that chip and its product semantics. The framework does not need to know what a campaign stage means to provide good typography, selection behavior or focus treatment.

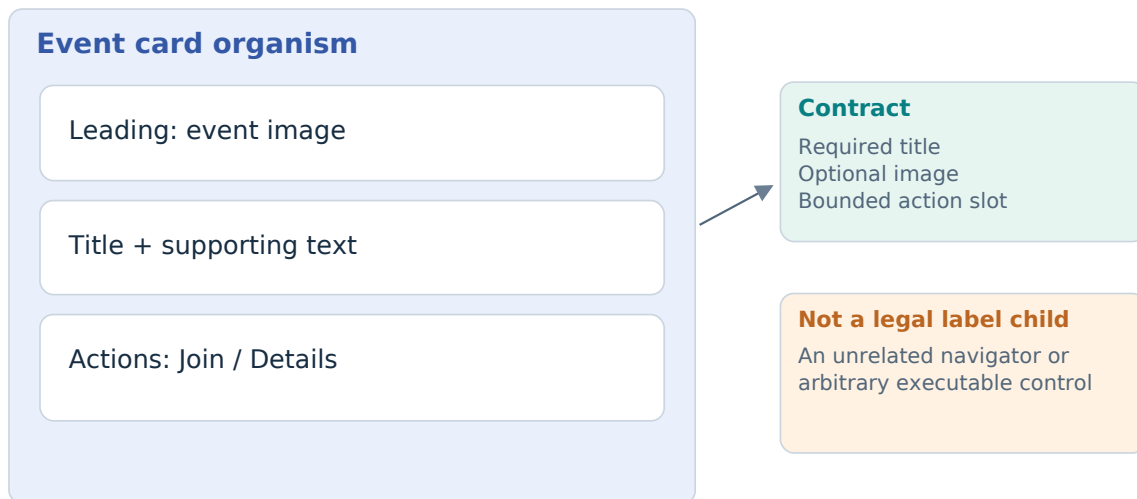
A product organism receives a presentation model and emits typed events. It should not locate repositories from ambient globals. That keeps it usable in previews, tests and generated compositions with controlled data.

For example, an event card can receive its title, image, availability description and action bindings. A page or feature owner decides where those values come from. The card can therefore be rendered in a feed, a conversation or a preview without bringing an entire application service graph into its implementation.

11.3 Slots express composition intent

A slot is a named place where content can be supplied. It is more informative than a generic child array when the component expects different kinds of content in different positions.

Slots describe allowed composition, not arbitrary placement



A template arranges organisms. Slots retain meaning when humans or agents compose them.

Figure 16: An event card separates its leading media, descriptive content and actions. Slot contracts keep those roles meaningful for manual and generated composition.

A label slot can permit inline text and a decorative icon. An action slot can permit a bounded set of controls. A supporting-pane slot can accept a product organism. These categories prevent an agent from inserting an unrelated navigator into a place intended for a short label.

Local code can expose richer composition when useful. The contract must simply identify which forms remain portable and which rely on a renderer-specific extension. Flexibility is not the same as pretending every arbitrary child is valid everywhere.

11.4 Compound controls need explicit shared state

Tabs are a useful example. The selected tab, the tab list and the displayed panel are related. Their relationship should be defined by a parent contract or explicit model, not by incidental position in a child list.

If an agent inserts a tab, the selected identity must still refer to the intended tab. Stable keys are prefer-

able to an index whose meaning changes when siblings are added. The same principle applies to menu items, reorderable lists and repeated form rows.

Controlled and locally managed variants also need a clear distinction. A field that receives an authoritative value cannot silently treat an unrelated default as a second source. The API states whether the owner controls the value or the component owns a local editing session that reports changes.

11.5 Avoid abstractions that hide the page

A screen should remain readable as a composition of meaningful regions. Extracting every pair of layout calls into a one-use wrapper can make the page harder to understand. The right question is whether the extracted name expresses a reusable concept or independently testable behavior.

A `MessageComposer` is useful because it names a coherent interaction region. A wrapper named `BottomRowContainer2` often reveals only that code was moved. Atomic design should improve the architecture's vocabulary, not reward the number of component files.

The same standard applies to graph decomposition. A feature boundary is useful when it owns a lifecycle, contract or updateable region. A spacer does not need an actor because it appears in a visual hierarchy. Sharing one graph foundation does not require uniform runtime machinery for every atomic level.

11.6 The catalogue becomes more useful to agents at higher levels

An agent asked to build an event-planning interface has less room for error when it can compose an approved participant picker, schedule summary and confirmation region. Reconstructing those experiences from hundreds of anonymous boxes gives the model more freedom but fewer guarantees.

Higher-level organisms do not eliminate customization. Their slots and properties expose the variation the product permits. When a new kind of organism is needed, it can be developed, tested and registered through the same process as human-authored components.

Atomic design consequently supports both development productivity and bounded runtime composition. The important asset is not a library of attractive shapes. It is a hierarchy of meaningful, tested units whose contracts remain usable across renderers and authors.

12. Themes, tokens and interchangeable visual languages

12.1 A palette is only one part of a theme

A palette assigns colors. A visual language describes more: how surfaces are separated, how controls receive emphasis, how corners and borders are used, how typography establishes hierarchy, and how interaction changes appearance.

Two themes can use the same colors and still feel different. One may be flat and editorial. Another may use a fixed contact plate, a moving face and a hard outline. A glass-like language may rely on translucency, while an accessible high-contrast profile may deliberately remove it.

Surface therefore separates four dimensions. **Brand** supplies product identity. **Design language** supplies visual grammar and response. **Mode** selects a supported light, dark or other appearance. **Density** adjusts the spacing and sizing appropriate to the interaction context. User preferences and renderer capabilities constrain the resulting combination.

These dimensions are not a promise that every possible combination is valid. A package declares the combinations it supports. Unsupported combinations produce a deliberate fallback or validation result rather than a partially themed interface.

12.2 Reference tokens, semantic roles and recipes

A reference token names a raw design value, such as a palette color or spacing step. A semantic role names its purpose, such as `content.primary` or `action.destructive.background`. A component recipe explains how a component uses those roles.

Consider the label on a strong button. The product screen requests a strong action. The recipe selects the strong-action background and its corresponding content role. The brand and mode resolve those roles to concrete values. The screen does not need to know which palette entry provides sufficient contrast on that background.

This chain is valuable for debugging. An inspector should be able to explain that a border came from the selected field recipe, which referenced an invalid-state role, which resolved through the current brand mode. Showing only the final color loses the reason behind it.

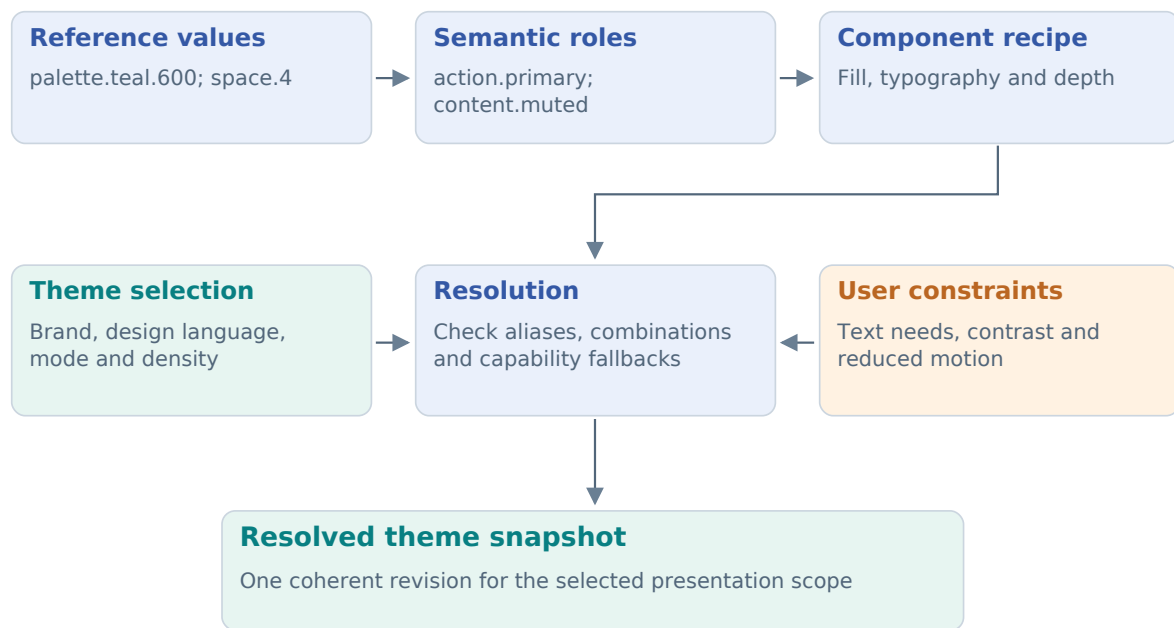
DTCG's typed token format provides an interchange vocabulary for tokens, aliases and composite values. Surface can use that format with explicit extensions rather than inventing an incompatible token island. The interchange representation is distinct from the compact values consumed during rendering.

12

12.3 A resolved theme is a coherent snapshot

The theme resolver checks definitions, aliases, resources and supported combinations before publication. It then produces one immutable snapshot for a scope. A snapshot has an identity and revision, so caches and inspectors can identify the decision set that produced a component's appearance.

Theme resolution separates values, meaning and component choices



Semantic roles survive palette changes; component meaning remains independent of the recipe.

Figure 17: The token pipeline preserves the difference between a raw value, a semantic purpose and a component-specific use. Theme selection and user constraints resolve these into one coherent snapshot.

```

data class ThemeSelection(
    val brand: String,
    val language: String,
    val mode: String,
    val density: String,
)

interface ThemeResolver {
    fun resolve(
        selection: ThemeSelection,
        environment: SurfaceEnvironment,
    ): Check<ThemeSnapshot>
}
  
```

The conceptual snapshot contains colors, typography and resource references, spacing, shape, depth and motion decisions. Generated accessors keep ordinary component code typed; a generic token map is an interchange or diagnostic representation, not an invitation to scatter string lookups throughout the application.

Coherent publication prevents mixed states. A consumer must not read the new background with an old foreground role because the package updated them separately. Preparing required fonts or a declared fallback before activation also reduces avoidable visual discontinuity.

12.4 Material is an adapter, not the universal model

Material provides a useful implementation foundation for many controls. A Surface Material adapter maps neutral roles into the values required by the chosen Material implementation. It includes typography and shapes as well as colors.

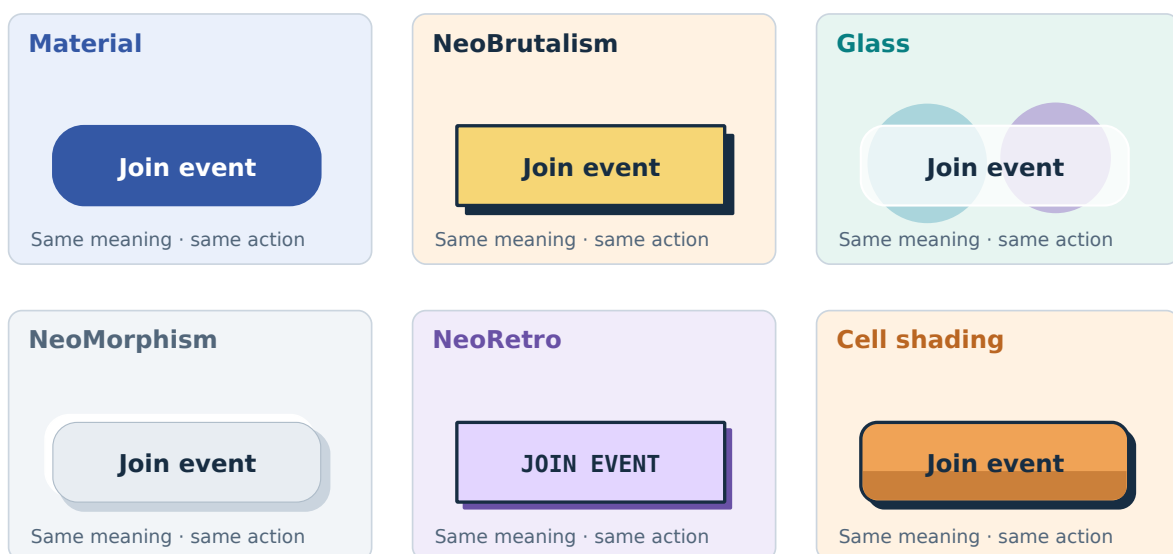
The mapping must be complete for the supported controls. A dark appearance can still inherit inappropriate defaults if only some roles are supplied. An adapter test should therefore check the actual role set used by components, rather than assuming that a function named “dark theme” guarantees every downstream value.

The shared model must also permit treatments that Material does not directly represent. A hard contact shadow, asymmetric product corner or field-reactive surface can be expressed through additional recipe concepts. These are not reasons to put all Material classes into every shared component contract.

Compose’s custom-design-system guidance supports replacing or extending theme systems rather than treating one supplied theme as the only possible design architecture. 13

12.5 Six languages, one semantic action

One action, six visual languages



Illustrative recipes, not pixel-parity targets. Focus and readable boundaries remain required.

Figure 18: Illustrative treatments for the same action. The visual language changes borders, depth, shape and typography, while the action and its accessibility obligations remain stable.

In a Material-like language, strong emphasis can be expressed with a filled control and familiar tonal relationships. In NeoBrutalism, the same emphasis may use a thick outline and an offset contact plate. Glass can use layered translucency when the renderer and contrast policy permit it. NeoMorphism can use opposing light and shadow, but needs explicit boundaries and focus indicators where subtle depth is insufficient.

NeoRetro can use deliberately angular geometry and distinctive typography. Cell-shaded treatment can use stepped tones and graphic outlines. These names describe families of recipes, not exact render-

independent pixel formulas.

A theme implementation cannot use a visual convention to change application availability. A destructive action still needs destructive semantics. A selected item remains selected even when its outline becomes a fill. The shared behavior and semantic contract remain underneath the visual experiment.

12.6 Data-only theme switching versus structural replacement

Most theme changes replace values and recipes while preserving component instances. A button keeps its key and interaction owner while its colors or motion profile change. Its draft-owning parent remains untouched.

Some languages need a different internal structure. A flat control might become a layered face-and-support implementation. Such replacement is allowed behind the same component contract, but requires explicit state and focus treatment. It is not equivalent to changing a CSS variable.

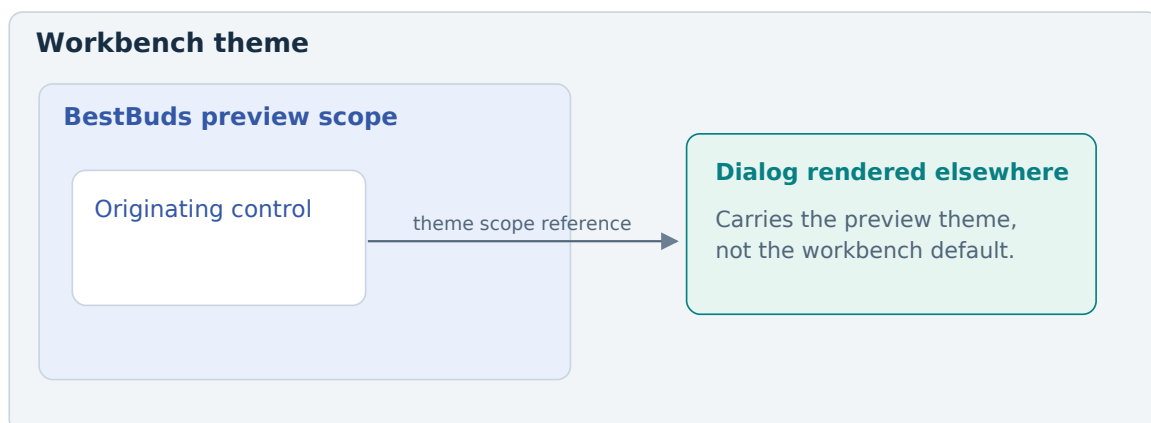
Transient contact state may be cancelled or simplified during the change. A user's typed value must be retained according to its state contract. An active input-method composition may require deferring replacement. The implementation should not silently destroy text merely because an appearance picker selected a more elaborate control.

Remote theme data can select approved recipe values and assets. Executable functions, arbitrary shaders and unrestricted platform code are not harmless token data. Their installation follows the graph's supported implementation-update boundary.

12.7 Theme scopes follow presentation ownership

A workbench may show its own chrome and several product previews at the same time. One global mutable theme cannot express this correctly. Each preview needs an effective scope, and menus or dialogs opened from it must retain that scope even if they are physically rendered elsewhere.

Theme scope follows ownership, including portalled overlays



DOM placement and theme ownership can differ without changing the application data owner.

Figure 19: A portalled dialog can be placed outside its originating DOM region while retaining that region's theme ownership. Physical placement and logical scope are different relationships.

On Compose, scoped providers supply the effective theme to a subtree. On the RSD path, context and scoped style variables provide the corresponding mechanism. The existing Reaktor React theme provider already illustrates the use of a wrapping scope rather than a document-global mutation. 14

A nested override should state exactly what it changes. A compact inspector may override density without changing the previewed product's brand. Non-negotiable accessibility constraints remain effective after local overrides.

12.8 Lowering recipes into each renderer

A neutral recipe does not mean Compose modifiers and CSS are interchangeable. Modifier order affects measurement, drawing and hit regions. CSS has its own layout and inheritance behavior. Each adapter lowers the same design intent into its toolkit's valid operations.

Static recipe decisions can generate StyleX-compatible styles for the RSD path. StyleX's static styling approach is useful here, but dynamic values and runtime interaction still need explicit treatment. Arbitrary Kotlin code is not automatically convertible into static CSS. 15

Recipes name the exact channels eligible for animation. A blanket transition on every property can animate geometry unintentionally and complicate performance. Palette-only changes should not cause media to reload. Typography changes should invalidate measurement because pretending the old geometry remains valid would produce stale layout.

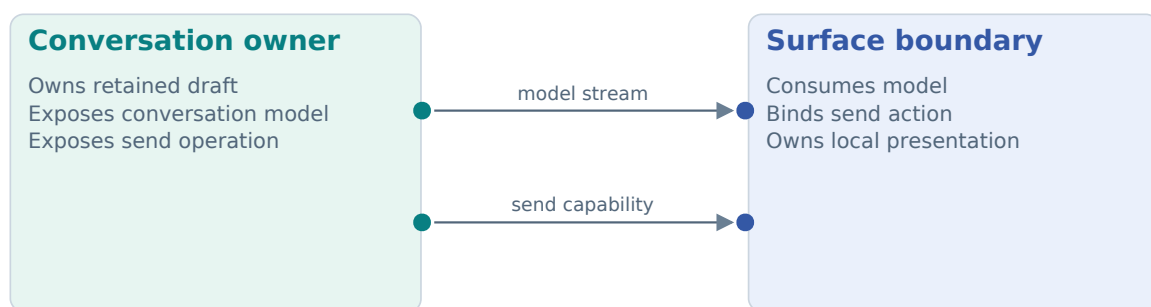
The theme system is successful when a new language changes the experience without changing component meaning, and when the inspector can explain both the selected values and the work their activation required.

13. Graph bindings, identity and lifecycle ownership

13.1 The graph describes application dependencies

A screen consumes application capabilities. A conversation timeline needs messages. A composer needs a draft and a send operation. A profile action needs a navigation binding. Reaktor's node-owned ports make these dependencies explicit rather than embedding service discovery inside drawing code. 616

Node-owned ports make dependencies explicit



A port is owned by its node. The view does not obtain the entire graph merely to call Send.

Figure 20: The conversation owner exposes a model and operations through node-owned ports. The Surface boundary receives a checked projection instead of acquiring unrestricted access to every application object.

The Surface integration layer resolves the relevant ports into a `GraphBinding`. That binding can use direct typed references in one runtime. It can use a qualified FFI realization when it crosses execution domains. The public operation remains the same conceptual capability; the connection mechanism is an implementation choice.

Inspection can then relate a control to the application operation it invokes. An agent or developer can ask which feature owns a draft or why an action is unavailable. This is more useful than inferring business dependencies from a visual tree.

13.2 Identity has several independent meanings

A component definition identifies a kind of control. A component instance key identifies one occurrence. A route entry identifies a particular navigation occurrence. A mount identifies one presentation attachment. A boundary generation identifies one active implementation binding.

These identities must not be collapsed into one integer called `id`. A theme revision can change without changing a route entry. Two windows can share a business object while having different mounts. A compact host handle can be reused internally, but it cannot become a stable external identity without

generation checks.

The graph already contains route-entry identity. Surface should use that identity instead of inventing a parallel navigation system. The existing `BackStackEntry` contract is a useful integration seam. 17

A deterministic inspection document also needs explicit ordering. A concurrent collection's current iteration order is not a persistent identity or a guaranteed declaration order. Stable IDs and a reproducible export order have different jobs. 18

13.3 The same business object can be presented more than once

Suppose a conversation is open in a main window and a detached window. The messages can share an application owner. The views should not share keyboard focus or scroll position merely because they refer to the same conversation.

One conversation can have several presentation attachments

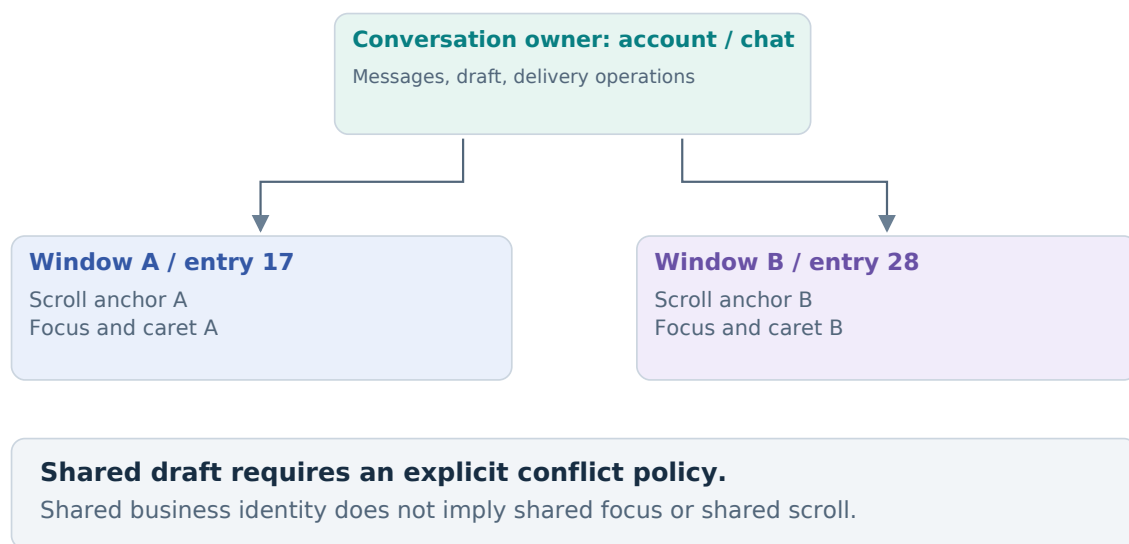


Figure 21: Business identity can be shared while each mount owns its own presentation state. A shared draft additionally requires an explicit multi-editor policy; sharing data does not define that policy automatically.

The draft requires a deliberate decision. It can be shared by account and conversation, with conflict handling for simultaneous edits. It can be isolated per composition session if that is the product's intended behavior. The architecture should make the choice visible rather than accidentally using whichever singleton happens to be available.

A separate route key alone is not sufficient. If both views write the same mutable active-chat binding, they are still sharing one stateful slot. Entry-aware content and keyed session ownership are prerequisites for correct simultaneous presentation. The existing route-binding and chat-session implementations make this a concrete integration concern. 1920

13.4 Existence, visibility, focus and interactivity are distinct

An application feature can exist while none of its views is visible. A hidden view can remain mounted to preserve selected presentation state. An outgoing screen can be visible during an exit animation but no longer admit new operations. A visible pane can be unfocused while another pane receives the keyboard.

These distinctions should be represented directly. One `isActive` boolean is usually too ambiguous to drive resource cleanup, navigation and input admission safely.

Lifetimes overlap; they do not start and stop together

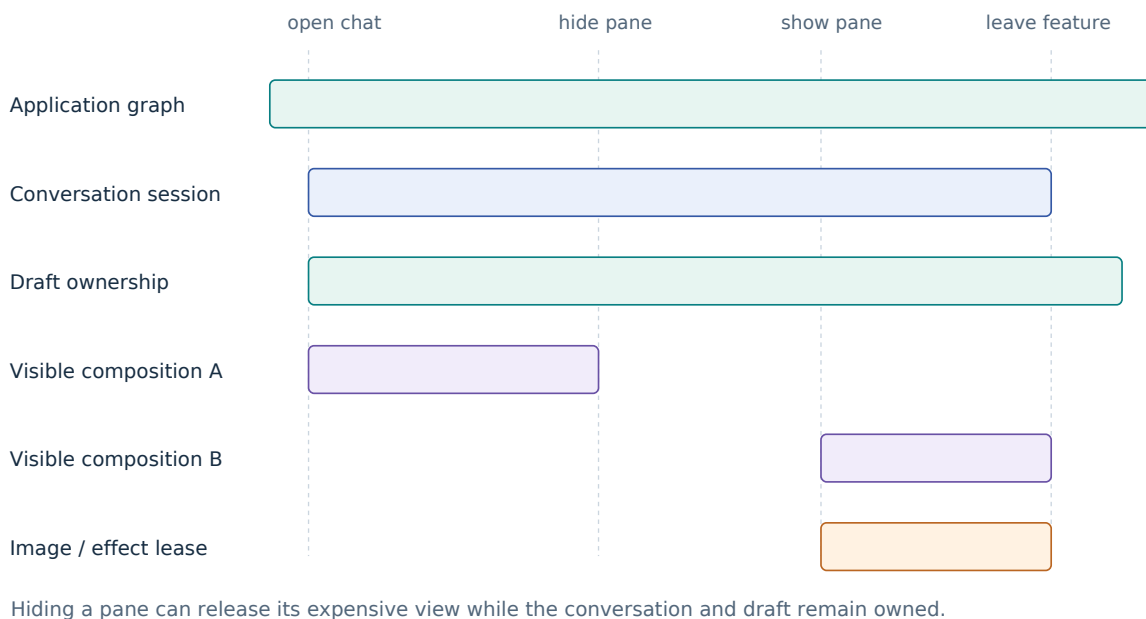


Figure 22: Application, workflow, composition and resource lifetimes overlap. A pane can disappear and return while the retained draft and conversation session continue to exist.

The renderer uses visibility to pause decorative work and manage expensive resources. Focus controls relevant input routing. Interactivity controls whether the current presentation can admit operations. The graph owner determines the feature's actual lifetime.

This separation also makes testing clearer. Hiding a pane should stop its unnecessary frame work. It should not close a conversation session unless the product explicitly ties that session to visibility. Closing the feature can release the session without deleting durable conversation history.

13.5 Composition lifetime is not application lifetime

Compose and React both provide local state tied to their composition structures. That is useful for short-lived UI details, but it is not automatically the correct place for a draft that must survive layout changes. Compose's state-hoisting guidance distinguishes UI-element state from longer-lived business state; React's state preservation depends on component identity in the rendered structure. ²¹²²

Surface places retained workflow state outside replaceable presentation branches. A local checkbox animation can remain in the component. A multi-step event draft belongs to an explicit workflow owner. The distinction should be based on required lifetime, not on a rule that all state must be global.

Effects also have owners. Starting a connection inside a composable body is unsafe because composition can run again. Lifecycle-aware effect mechanisms exist to attach work to an appropriate scope, but their scope must still match the intended application lifetime. 23

13.6 Platform lifecycle cannot be flattened completely

Android can recreate an activity in response to configuration changes. A browser can hide, freeze or discard a page. Desktop and mobile hosts have different window and process relationships. These platform events do not form one identical lifecycle enum, and a final cleanup callback cannot be assumed to run in every termination case. 2425

The portable layer therefore expresses required policies rather than pretending platform events are identical. It can say that visible animation pauses when presentation is suspended, or that a draft must be persisted after meaningful edits. Platform adapters map those requirements to their actual lifecycle facilities.

Persistence is separate from lifetime. A graph-owned object can outlive a view but still disappear if the process terminates. If the product promises recovery after process loss, the relevant state needs a storage contract. Keeping it in a longer-lived in-memory owner does not provide durability by itself.

13.7 Resource leases make cleanup explicit

A resource lease records that a scope is using something expensive or externally managed. Examples include a decoded image, a native editor instance, a platform observer and a field participant registration.

When the scope closes, its leases are released. An underlying cache may retain reusable data under a byte budget, but that is not the same as a forgotten reference keeping the entire view alive. Cleanup should be idempotent so repeated closure cannot corrupt resource accounting.

For asynchronous work, cancellation also has cleanup obligations. A superseded decode may have allocated a buffer. A cancelled host preparation may have created callback entries. Moving work off the UI thread does not remove the need to release those resources.

The useful invariant is simple: every retained object has an owner, and every owner has a defined point at which it stops retaining the object. Performance and correctness both become easier to reason about when this is explicit.

14. Adaptive layouts, navigation and coordinate systems

14.1 Adaptation begins with a question about the task

An adaptive layout is not just a different number of columns. It changes how much of a task is visible at once. A compact conversation interface prioritizes the current conversation. A larger window can show the list beside it. A still larger workspace can add a supporting event or profile region.

The task remains coherent because the underlying state is shared through its proper owners. The layout exposes more context instead of creating a different application workflow whenever the width changes.

One route model, three adaptive presentations

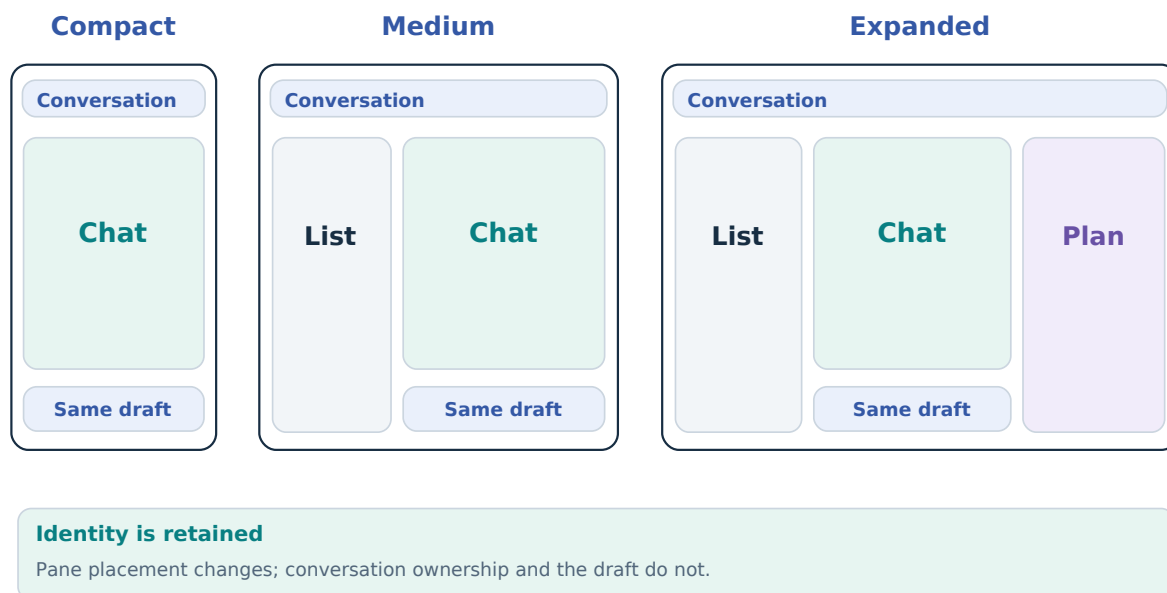


Figure 23: Compact, medium and expanded presentations expose different amounts of context. These are illustrative allocations: actual pane selection follows measured constraints, content needs and user preferences rather than device names alone.

The canonical list-detail, supporting-pane and feed patterns are useful starting points. They encode common relationships between content regions. Their value is not that every application must look alike, but that the layout policy can express a recognizable task structure. 2

14.2 Window environment and container constraints

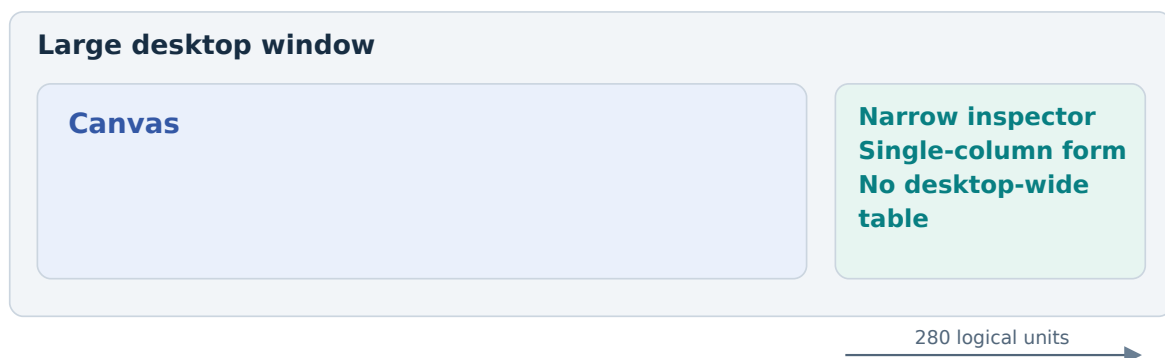
The **window environment** describes the host: logical dimensions, insets, input capabilities, text preferences and relevant posture information. **Container constraints** describe the space assigned to a particular component.

```
data class SurfaceEnvironment(
    val window: LogicalSize,
    val insets: Insets,
    val direction: LayoutDirection,
    val textScale: Float,
    val pixelDensity: Float,
    val inputs: InputCapabilities,
    val reducedMotion: Boolean,
)
```

The record expresses the conceptual information the policies require. A platform adapter may preserve additional text-scaling or posture information where a single scalar is not sufficient. Such information belongs to a declared capability rather than being discarded to fit a falsely universal model.

A desktop inspector can receive a narrow allocation inside a large window. Its layout uses that allocation. It should not render a wide table because the outer window is classified as expanded.

The available container, not the device name, determines fit



A window class selects the shell. The inspector measures its own actual allocation.

Figure 24: A large window can contain a narrow region. The shell responds to window conditions; each component measures and adapts within its actual container.

Android’s window-size guidance treats these classes as dynamic properties of application space. Surface uses them for broad policy decisions, while actual constraints determine whether the content fits. 26

14.3 A pure policy chooses a presentation plan

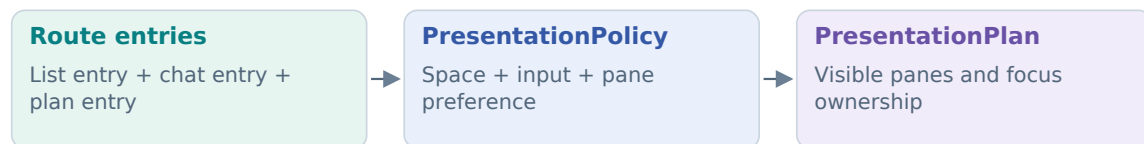
The presentation policy reads navigation, constraints and preferences. It returns a plan describing visible entries, their pane roles and focus relationships.

```
data class EntryPresentation(
    val entry: EntryId,
    val role: String,
    val interactive: Boolean,
)

data class PresentationPlan(
    val panes: List<EntryPresentation>,
    val focusedEntry: EntryId?,
)
```

A richer implementation can add minimum widths, overlays and collapse priorities. The important rule is that the result remains a presentation plan. The policy does not push or pop routes while measuring a pane.

Navigation state is projected into panes



Resizing changes the plan, not the route history. Back navigation still goes through the graph.

Figure 25: Navigation remains graph-owned. The presentation policy determines how existing entries appear within the available space. A resize changes pane allocation without silently adding history.

This design avoids feedback loops. If layout both reads and mutates navigation, a resize can repeatedly change the very state being laid out. Pure planning allows deterministic tests: given these entries and constraints, this arrangement is selected.

14.4 Content determines whether an arrangement is usable

A nominal breakpoint is not sufficient to prove that content fits. A translated title may be longer. A larger text preference may increase a form's minimum width. A resizable inspector may leave less room for the main panel.

The shared policy can define preferred and minimum pane widths, collapse order and available arrangements. The platform layout engine performs actual measurement. If constraints cannot be satisfied, the plan can reduce simultaneous panes or use an alternate presentation.

This does not require inventing a universal layout solver. Compose and the web already have capable layout mechanisms. Surface shares the task-level policy and the meaning of layout primitives, then qualifies how each renderer realizes them. Intrinsic measurement and text shaping remain platform work. ²⁷

14.5 Back navigation and dismissal need explicit priority

A back action can close a modal, dismiss a supporting sheet or pop a route entry. These are not always interchangeable. The presentation layer identifies the currently relevant input scope, but navigation authority remains in the graph.

For a compact layout, a supporting profile may occupy a foreground route. In an expanded layout it may be visible beside the conversation. The application must define whether Back removes the supporting entry, changes focus or navigates elsewhere. Merely moving the view into a side panel should not invent a new history.

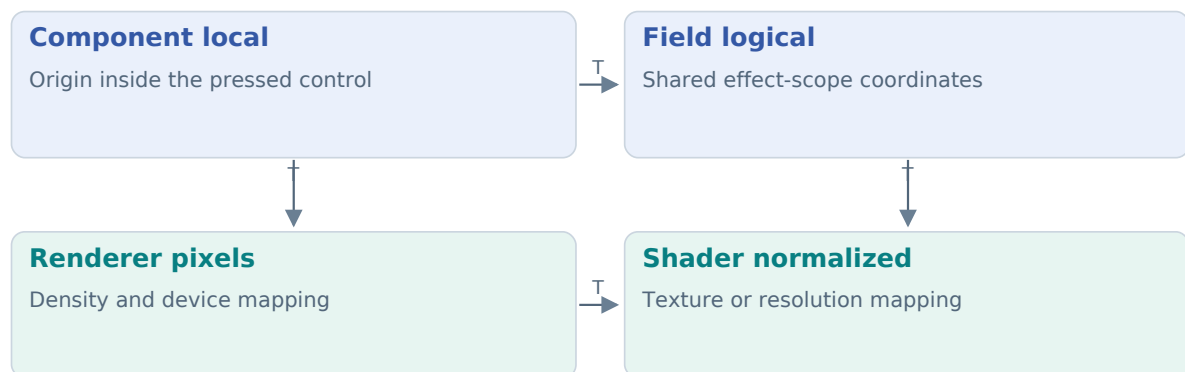
Web realization must also preserve browser history and link behavior. These are platform semantics, not visual effects. A theme cannot change whether a resource is navigable, and a decorative transition cannot be the sole mechanism that completes a route result.

14.6 UI scale, text scale, density and canvas zoom

Several values can make something appear larger, but they mean different things. Pixel density maps logical drawing units to physical pixels. UI zoom changes the logical amount of space available to application chrome. Text scaling follows the user's reading needs. Canvas content zoom changes a document or graph viewport.

Applying the same multiplier at several levels creates errors. A popup can be displaced, a hit region can fail to match its visible control, or an image can decode at an unnecessarily large size. The adapter must specify which coordinate system each value uses and which transforms have already been applied.

Coordinate systems must be named and transformed explicitly



UI zoom and canvas content zoom are separate transforms. Popup and hit-test coordinates follow the right chain.

Figure 26: Input origins, effect fields, drawing pixels and shader coordinates belong to different spaces. Explicit transforms keep feedback, hit testing and overlays aligned through scrolling, resizing and zoom.

A 125% workbench zoom should normally affect layout and usable logical space. Scaling only the final drawing layer can leave input and accessibility geometry behind. A graph's pan and zoom remain separate content transforms so the workbench controls do not grow every time the graph is magnified.

14.7 Layout stability is a contract for asynchronous content

An image can arrive after surrounding text is measured. A font can load after a fallback has been used. A generated region can become available after the user begins reading. These changes are normal; unexpected movement of the user's target should not be the default consequence.

Media reserves dimensions or aspect ratio before loading when that information is known. Placeholders approximate the expected structure. Collections restore position using item identity and relative offset rather than only a global pixel coordinate.

The invariant is not that nothing ever moves. Larger text and translated content legitimately require re-flow. The invariant is that the system anticipates asynchronous geometry where possible and preserves a meaningful anchor when geometry must change.

For a conversation, the anchor can be the message being read. For a form, it can be the active field. For a graph, it can be the selected node's relationship to the viewport. Each is more meaningful than assuming an old absolute offset remains correct after all layout inputs change.

15. The Compose-first implementation

15.1 Keep the first production path direct

The first implementation should make ordinary CMP applications better without requiring React, a custom portable tree or a binary bridge. A Compose screen consumes a graph binding, enters a theme scope and calls Surface's Compose components. Those components combine shared contracts and behavior with Compose's existing rendering facilities.

This path is deliberately short. A local button call does not need to encode its properties and decode them again in the same runtime. A normal form does not need an arena-backed tree merely to use shared semantics. The design's portability is established by its contract boundaries, not by forcing every local operation through the most general transport.

A component can nevertheless be registered in the shared catalogue. Its definition, behavior traces and recipe roles are the same definitions later used by RSD. Direct execution is an implementation optimization, not a separate design system.

15.2 The adapter translates, rather than owns, application behavior

A Compose button adapter has a small sequence of responsibilities. It observes the current controlled properties. It maps toolkit interaction into the behavior kernel. It resolves the component's appearance from the effective theme. It renders through an appropriate Compose implementation. Finally, it forwards admitted semantic events to the bound application action.

The adapter does not start a conversation session because it drew a Send button. That session belongs to the graph feature. It does not maintain a private theme registry unrelated to the surrounding scope. It does not make network success depend on animation completion.

A representative direct call can remain simple:

```
@Composable
fun ConversationComposer(
    model: ComposerModel,
    actions: ComposerActions,
) {
    SurfaceRow {
        SurfaceTextField(
            value = model.draft,
            onEdit = actions::editDraft,
        )
        SurfaceButton(
            label = TextContent.Plain("Send"),
            enabled = model.canSend,
            busy = model.sending,
            onActivate = actions::send,
        )
    }
}
```

The surface functions in this example belong to the direct CMP package. The product-level function receives already-bound models and actions. A portable package may provide corresponding declarations, but it cannot silently mix the two applier vocabularies inside one composition.

15.3 Remember identity, not every changing input

A component's interaction controller should survive ordinary property updates. Recreating it when the color changes would erase contact or focus state unnecessarily. Conversely, reusing the same controller for a different logical item can transfer state into the wrong control.

Stable component or collection keys determine identity. Controlled properties are supplied to the controller as changing inputs. A callback retained by an effect needs access to the latest applicable properties and handler rather than an old captured value. Compose's effect APIs distinguish effect lifetime from values that can change during that lifetime. 23

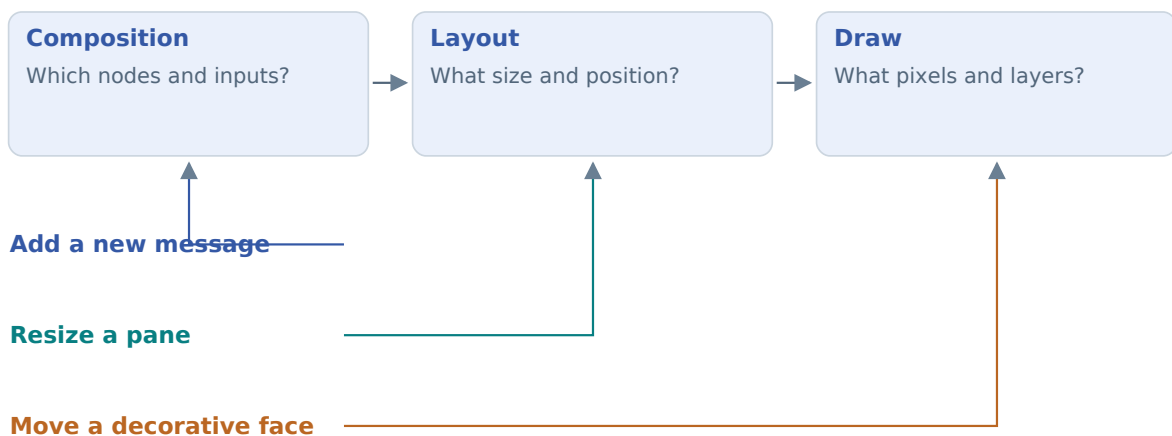
For example, disabling a pressed button should reconcile the active contact state. The controller should not be recreated solely to make that happen. The new properties enter the reducer, which cancels contact under its defined policy.

Theme data also remains an input. A recipe change updates visual resolution; it does not reset the behavior kernel unless the component implementation itself is being replaced under an explicit policy.

15.4 Observe work at the right phase

Compose separates composition, layout and drawing. A state read can affect the phase in which it occurs, so the placement of rapidly changing reads matters. 5

Invalidate the latest phase that can express a change



The precise invalidation path is an implementation result to measure, not a promise from a prop name.

Figure 27: A structure change can require composition. A constraint change affects layout. A decorative face translation may be realized later in the pipeline. The adapter should measure which work actually occurs instead of treating all updates as the same kind of rerender.

The direct CMP implementation can use this separation to keep tactile work local. A press response that only changes a graphical transform need not rebuild the full product layout each frame. A field whose text size changes does require measurement. The contract identifies the intent; the adapter chooses the correct phase-sensitive mechanism.

Avoid reading a whole animation state object in a high-level screen simply to pass one scalar to a child. Such a dependency can widen invalidation. Keep the observation near the component or layer that consumes it, while preserving a readable API.

15.5 Toolkit behavior remains valuable

Foundation and Material implementations provide useful control behavior. Surface can wrap or adapt them rather than replacing every control with a raw Box and pointer detector. A mature input field is especially valuable because text editing has complex platform interactions.

A custom tactile treatment may need a new visual structure while reusing the established semantic and interaction paths. The contact plate can be decorative. The face can animate locally. The control still has one activation and one appropriate focus target.

This is an important form of elegance: the framework shares the rules that must be shared and delegates machinery that a platform already implements well. A renderer-neutral behavior kernel does not require a renderer-neutral replacement for every native text service.

15.6 Compose-first must not mean Compose-shaped contracts

The easiest accidental coupling is to put Compose value types into public shared contracts because the first implementation uses them. That creates work later when RSD is introduced. Neutral component inputs, semantic roles and resource identities must therefore be established before a large catalogue is built.

The same is true of lifecycle. A shared component contract should not mean “close the feature when this composable leaves composition.” It should describe the feature’s state ownership and the view’s attachment separately.

A small portability probe is useful before API freeze. It can take the same component definition, serialize and validate representative values, and run the behavior trace without Compose. A tiny custom-Appplier experiment can verify the host vocabulary. Neither requires postponing production CMP components until a complete React renderer exists.

16. HostApplier and the portable composition boundary

16.1 Why an Applier exists

Compose Runtime can maintain a tree without requiring that the tree be a Compose UI layout tree. The Applier interface is the bridge that applies operations to the chosen kind of tree. It tracks a current node, traverses down and up, inserts and removes children, moves them and applies changes. 28

A **SurfaceHostApplier** is an application of that idea to Surface's portable host vocabulary. It creates host nodes describing registered components and their typed properties. A renderer then realizes those nodes through CMP or RSD.

The Applier is not a UI designer. It does not decide whether a page should be list-detail or which action is destructive. Composition and contracts already express those decisions. The Applier records structural operations correctly and hands committed output to the host boundary.

Direct Compose and portable Compose have different host trees



Figure 28: Direct Compose uses the normal Compose UI path. Portable Surface composition uses a custom Applier to produce a different kind of node tree. The two paths share contracts but do not use interchangeable internal nodes.

16.2 A tiny tree makes the idea concrete

Consider a portable group containing two text nodes. The composition runtime determines that the group should exist and that one text value changed. The Applier receives operations describing the resulting tree changes. It does not need to interpret the original Kotlin conditional that selected those values.

An internal mutable node can be simple at first:

```
class MutableHostNode(
    val key: InstanceKey,
    val component: ComponentId,
) {
    var properties: HostProperties = HostProperties.Empty
    val children = mutableListOf<MutableHostNode>()
}
```

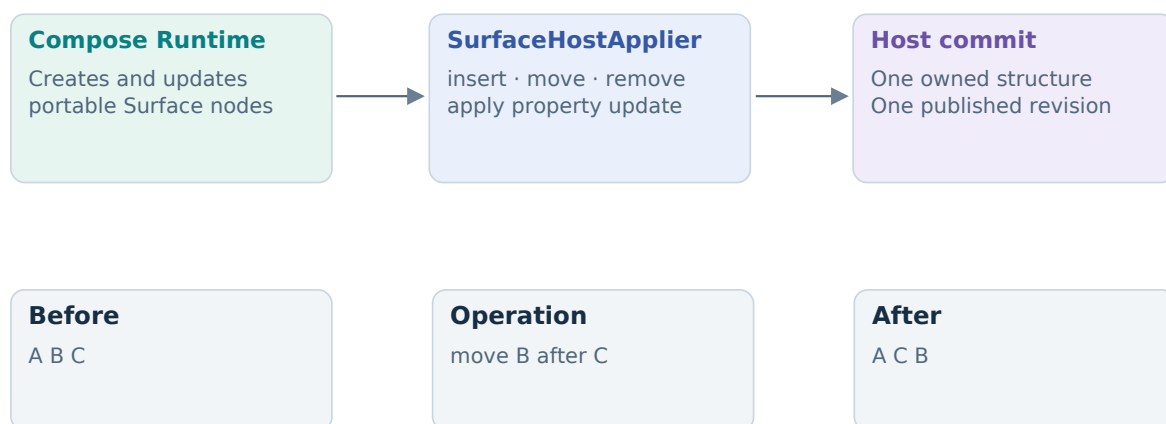
The production representation can evolve after measurement. An arena or specialized child buffer may eventually be useful, but is not required to understand ownership or validate the initial contract.

The node is local to the composition owner. It is not a graph node with an application coroutine scope. Its key relates it to the committed presentation instance and inspector, while its parent-child links describe UI structure.

16.3 Insert, move and remove are distinct operations

If a list changes from A, B, C to A, C, B, B should usually retain its identity. Recreating every row would lose useful local state and resource relationships. A move operation expresses the change without treating B as a different component.

An Applier applies operations; it does not infer application intent



The moved item keeps its key. Position changes are not equivalent to deleting its identity.

Figure 29: The Applier applies a structural move while preserving the moved instance's key. That key is an identity within the owning boundary, not merely its current index.

The AndroidX interface exposes both top-down and bottom-up insertion hooks, with insertion performed in one of them rather than both. Its move destination uses the specified pre-change index convention. A Surface implementation must follow those semantics exactly; guessing at index adjustments can corrupt reordered trees. 28

A minimal adapter can use a mutable list and explicit range operations. Tests should include adjacent moves, moving several siblings, removal of a subtree and repeated clear operations. These tests are more valuable early than choosing a sophisticated data structure before the semantics are stable.

16.4 Publication belongs at a coherent boundary

The privately owned desired tree can be updated while changes are being applied. Other consumers must not observe arbitrary intermediate combinations. Once the frontend has committed its changes, the host validates and publishes a coherent representation.

```
interface HostCommitSink {  
    fun publish(commit: HostCommit): Check<Long>  
}  
  
data class HostCommit(  
    val mount: MountId,  
    val generation: Generation,  
    val catalogFingerprint: String,  
    val baseSequence: Long,  
    val sequence: Long,  
    val root: HostNode,  
)
```

Immutability needs more than a `val` declaration. A `val` prevents reassignment of that reference, but a referenced byte array can still change. A read-only collection interface also does not prove that no other owner can mutate its backing storage. Publication therefore freezes or copies the relevant values, or transfers them under an explicit ownership contract. The simplest baseline copies committed property bytes; a leased low-copy representation can be introduced after its lifetime rules are established.

The initial representation can publish a full immutable snapshot. The sink checks the owning mount, generation, catalogue and base sequence before accepting it. More compact operation batches can be introduced later without changing the ownership model.

If publication fails, the last valid displayed tree remains. The frontend and host must then resynchronize or fail the affected boundary through its recovery path. It is unsafe to continue as though rejected changes had been accepted, because the frontend's desired structure and the displayed structure would have diverged.

16.5 The committed host tree is not the source program

The host tree records what has been committed, not why the program produced it. A Kotlin loop may produce ten rows. A React conditional may omit a panel. The output alone does not recover those source-level constructs.

This matters for Blueprint. A supported template declaration can be edited visually because it has an authoritative editable representation. Arbitrary source-owned composition is inspected and linked back to its source. A visual operation must be routed through that owner rather than silently changing the runtime output behind its back.

The same distinction prevents two structural writers. The Applier owns the tree it produces. A parent can mount a boundary and update its inputs. It cannot independently reorder the boundary's internal children while the Applier continues producing another order.

16.6 Derived renderer data remains derived

A renderer may need additional structures: resolved styles, layout metadata, dirty flags, hit-test indexes and accessibility projections. These can be derived from the committed host structure and runtime inputs.

They must not become a second editable authority. If a layout cache disagrees with the committed structure, the cache is invalidated or rebuilt. A full recomputation path provides a useful reference for testing incremental derivation.

This keeps optimization separate from meaning. A browser realization may use structures quite different from a native realization. Both can satisfy the same host contract without requiring an unnecessary C++ round trip or identical memory layout on every target.

17. React, reconciliation and the later RSD realization

17.1 What reconciliation contributes

React components calculate an element structure from props and state. Reconciliation relates that output to the previous structure so that the renderer can preserve appropriate identities and apply necessary changes. Commit is the point at which the resulting changes are applied to the host. 4

The useful analogy is editing a document rather than replacing the entire document file after every keystroke. Some content changes, some moves, and much remains the same. Keys help distinguish an existing item from a new occurrence occupying the same position.

Reconciliation identifies changes; commit makes them visible

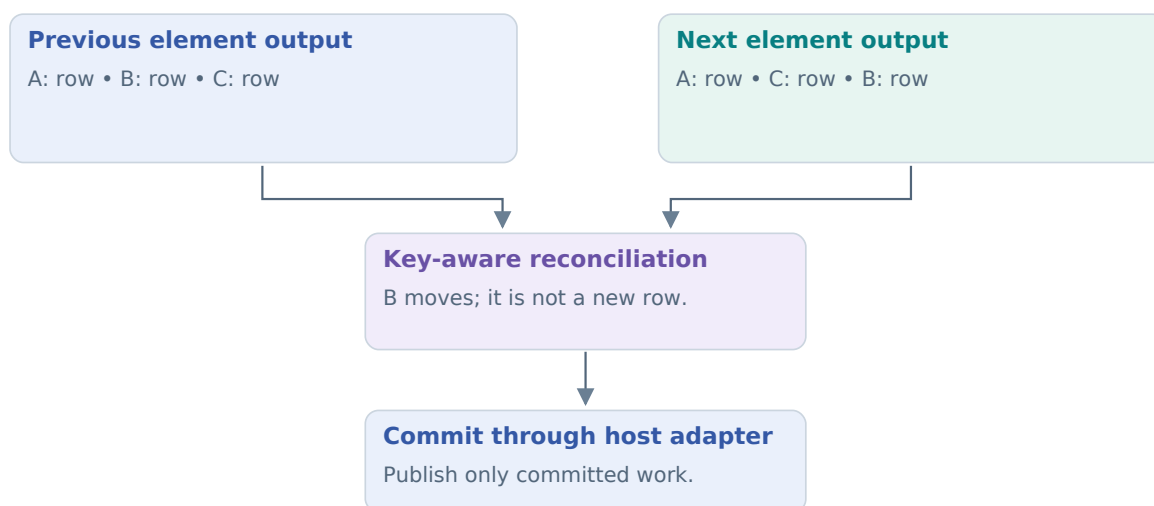


Figure 30: Reconciliation preserves the identity of keyed items while finding structural changes. Host publication follows the frontend’s commit boundary, not every speculative calculation.

A React render is not a business transaction. Evaluating a component must not send a message simply because the Send control appears. That operation belongs to the event path. The same restriction applies to tactile cues: rendering a pressed appearance is not another contact event.

17.2 The direct RSD path

The first React realization should use RSD directly. Surface’s generated TypeScript-facing contracts describe props and events. RSD component implementations realize them with the supported web or React Native behavior. Shared recipes are lowered through the appropriate styling layer.

This path does not need the custom React-to-Compose bridge. It adds a renderer implementation behind already-established contracts. The graph binding supplies state and actions, and the component receives only the capabilities it needs.

The state adapter must preserve snapshot identity when data has not changed. React's external-store interface provides an explicit subscription and snapshot contract. Surface's graph-to-React bridge should obey that contract rather than producing a fresh logically identical object on every read. 29

A resolved model can use narrow projections so that one changing item does not require every sibling to observe a new application-root object. Coherent multi-field changes still need an aggregate revision where consistency matters.

17.3 Sharing behavior across languages

The pure Kotlin behavior kernel can be compiled for a supported JavaScript target and exposed through generated wrappers. This avoids manually reinterpreting the machine in every component implementation.

Where a renderer uses a native behavior facility instead of calling the same kernel for every event, it must still satisfy the same contract traces. Shared source and shared observable behavior are related but not identical promises. A native field should keep its competent editing implementation while exposing the agreed value and event semantics.

Platform event normalization remains target-specific. Mouse events, touch interactions, keyboard activation and accessibility actions reach the semantic control differently. The RSD adapter is responsible for mapping them once, preserving native control behavior and avoiding duplicate activation.

This is why the behavior kernel was separated before building the CMP catalogue. React adoption becomes an adapter and conformance project rather than a rewrite of the component's meaning.

17.4 React-to-Compose is an optional host adapter

A custom React renderer supplies a HostConfig that realizes host operations. For Surface, the host instances are registered Surface nodes rather than DOM elements. Committed output enters the same checked host boundary consumed by the CMP implementation.

The react-reconciler package is an experimental integration boundary whose API is less stable than the public React APIs. Its own documentation also distinguishes render-phase preparation from committed work: creating an instance does not guarantee that it will enter the displayed tree. 30

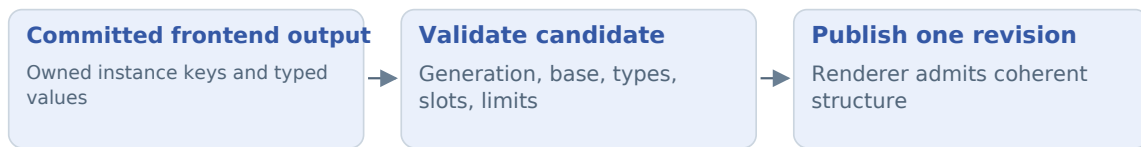
Surface must therefore qualify a specific React/reconciler combination. It should not depend on private Fiber fields or promise compatibility across arbitrary independent upgrades. Preparing an uncommitted node may initialize isolated data, but it must not produce visible application effects or register live callbacks on unrelated displayed structures.

The bridge sends meaningful committed changes in bounded batches. It does not cross from JavaScript to Kotlin once per pointer sample merely to animate a face. Native-side behavior and motion can continue locally after the structural inputs have been committed.

17.5 Bounded commits and event delivery

Host values identify the mount and active generation. A stale callback or old structural commit cannot be applied to a newly activated region that happens to reuse a local numeric handle. The runtime resolves those handles only within their valid ownership scope.

A host update is checked before it becomes interactive



Rejected work leaves the last valid structure. Uncommitted preparation produces no application effects.

Figure 31: Validation precedes publication. Generation, base revision, types, slots and ownership must agree before a candidate tree becomes the interactive representation.

Not all events have the same delivery semantics. A latest hover position can replace an older unconsumed hover position. A text edit or submission cannot be discarded under that policy. Ordered application operations and coalescible presentation observations need different queues and explicit back-pressure.

A large collection also needs a collection contract rather than a huge serialized tree. The host can materialize demanded items with stable identities while local scrolling remains responsive. A faster codec does not make an unbounded amount of rendering work acceptable.

17.6 What minimal-change React support really means

Minimal change means the shared component definitions, behavior rules, theme roles, graph bindings and lifecycle model do not need to be redesigned. It does not mean the RSD renderer appears automatically.

The later implementation still needs platform controls, style lowering, resource integration, subscriptions, focus and accessibility testing. The optional React-to-Compose renderer needs additional host-protocol qualification. Portable Compose-to-RSD needs its own frontend and leaf-realization integration.

The architecture preserves those paths without making them prerequisites for the first CMP release. A small schema and host probe is enough to challenge bad assumptions early. Production breadth expands only after the new realization satisfies the same component and workflow obligations as the first one.

18. Animation, transitions and temporal behavior

18.1 Animation describes change over time

An animation changes a presentation value over time. A button face moves down. A highlight fades. A progress indicator advances. A transition coordinates a change between interface configurations, such as opening a dialog or moving from a list into a detail screen.

These mechanisms are related but not identical. A component can animate without changing its logical structure. A structural transition may involve two views, focus transfer, resource retention and input admission. Treating both as a duration attached to a style misses the lifecycle decisions involved.

Core motion belongs to Surface. Basic press feedback, enter and exit behavior, navigation transitions and layout changes must work without the richer field and shader systems. Tactile adds material relationships and sensory expression; it does not own the only way to move a component.

18.2 Duration-based motion and spring-based motion answer different questions

A duration-based animation answers: how should the value move from one state to another over a chosen interval? An easing function controls the progress. This is useful when a coordinated sequence needs a known timing relationship.

A spring-based animation answers: how should the value respond as it approaches a target under a chosen physical response? Stiffness and damping affect the trajectory. The system can retarget while preserving current motion rather than always restarting from a fixed origin. Compose exposes duration-based and spring-based animation mechanisms with different controls. 31

Neither model is universally better. A progress value tied to actual work should not be faked by a spring that implies completion. A press response may feel more coherent when it reverses from its current velocity. A carefully timed crossfade may be appropriate for a mode change.

Surface's motion contract therefore describes intent and interruption policy, not only a universal duration. A renderer chooses a qualified implementation that preserves the required behavior.

18.3 A motion channel has one owner

Several systems can affect a component. Layout assigns its base position. A drag changes its content transform. A press changes the local face. A field contributes a small expressive offset. These values must be combined in a defined order.

Without channel ownership, an update from one system can erase the others. A React style update may replace a transform controlled by a native animation. A field response may fight a drag controller. A theme swap may reset the animated value while the gesture is still active.

The renderer's final transform is a composition of these channels, not a race between unrelated writers. The same principle applies to color, opacity and clipping. An implementation can optimize the combination while retaining a clear source for each value.

Each transform channel has one owner

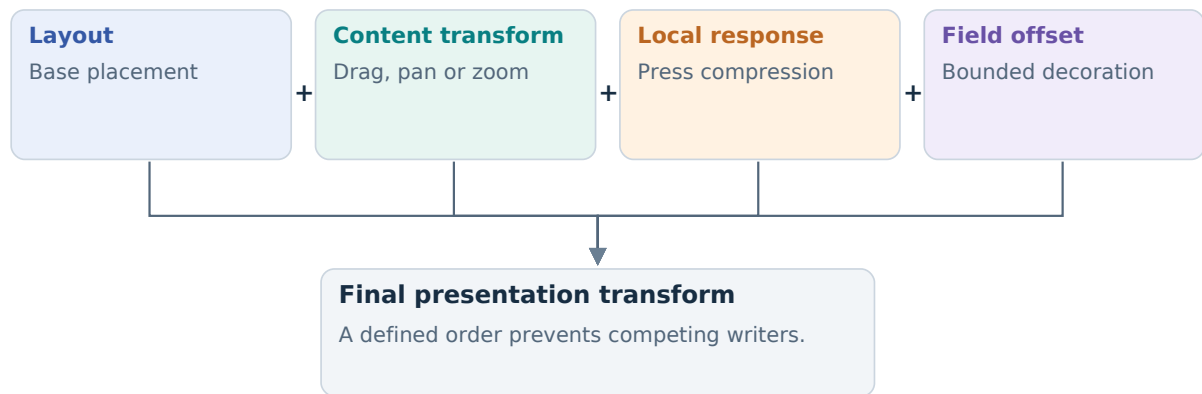


Figure 32: Different transform channels can contribute to one visible result, but each channel has one owner. Composition order is explicit so an animation driver, a drag controller and a recipe do not overwrite the same property independently.

18.4 Interruptions are normal input

A person can press again before a release animation settles. A sheet can be dragged upward and then reversed. A navigation action can occur while the previous transition is still visible. A window can resize during a shared-element transition.

The contract must define these cases. A response can retarget from its current presentation. A structural transition can complete or cancel under a policy. A new layout can rebase the animation geometry. The framework must not assume that every animation runs uninterrupted to its planned end.

A simple local value interface exposes the required boundary:

```

interface MotionValue : ResourceLease {
    fun retarget(target: Float, profile: SpringProfile)
    fun sample(atSeconds: Double): Float
    val settled: Boolean
}
  
```

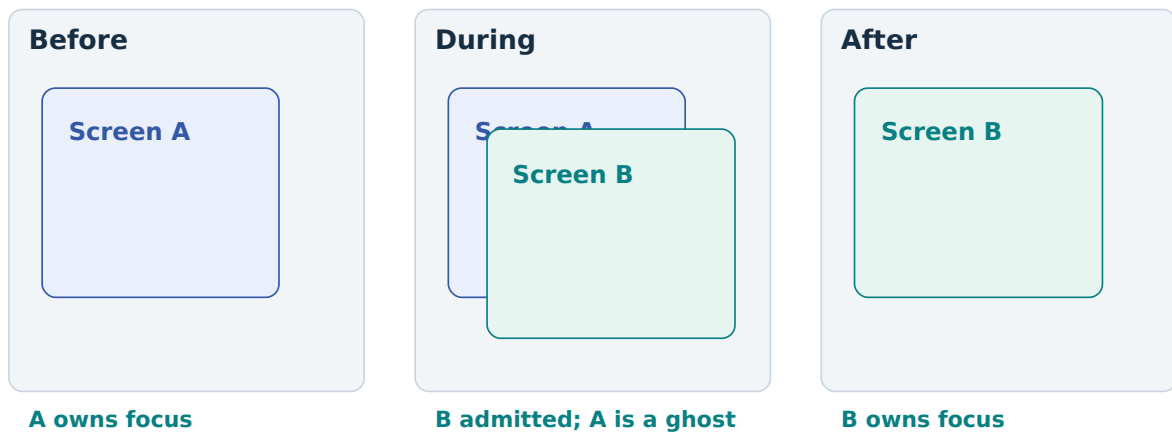
The owning driver supplies a monotonic clock. Its internal state retains current position and velocity where required by the response model. A sample is presentation data, not a graph mutation. Once settled, the driver stops requesting frames unless another change requires them.

18.5 Visibility during a transition is not permission to act

When screen A exits and screen B enters, both may be drawn temporarily. That does not mean both should continue admitting operations. The transition identifies which semantic control owns focus and when the incoming content becomes interactive.

Shared-element transitions require the same distinction. A photograph can appear to move from a card into a detail view, but the transition copies are drawing artifacts. They should not create duplicate accessibility targets or expose private content across unrelated scopes. CMP's shared-element APIs

A transition changes visibility without duplicating authority



Outgoing visuals may remain briefly, but must not continue admitting new actions.

Figure 33: The outgoing screen can remain as a visual layer during the transition. Input and application authority move according to the transition policy, so a decorative copy does not become a second active control.

provide implementation mechanisms; the application still supplies appropriate identity and ownership.
32

Business completion must remain independent. A route result cannot rely exclusively on an animation callback that might be cancelled. A purchase does not succeed because a confirmation panel finished sliding into view.

18.6 Reduced motion is an alternate presentation, not a broken state

A reduced-motion policy can shorten, simplify or remove designated movement while preserving useful feedback. The control still needs a clear pressed, selected or focused state. A navigation operation still reaches its correct destination.

The policy also affects active animation. Switching preferences should not leave a component halfway between two semantic states. The driver can settle to the required result or use a simpler transition while preserving input and ownership.

Testing uses a deterministic clock. A test can advance to the middle of a transition, reverse it, resize the host and verify the resulting state. This exposes lifecycle errors more reliably than waiting for a screenshot captured after every animation has already finished.

19. Tactile interaction as a system of material relationships

19.1 A material response starts with a believable relationship

A tactile interface suggests that its elements have weight, support and resistance. A button does not merely become smaller. Its face moves toward a support. A card can respond to the contact of a child control. A surrounding field can reveal the origin and spread of an interaction.

The purpose is not to simulate the physical world perfectly. It is to make response intelligible. The visible movement should agree with the interaction that caused it, remain within usable bounds and stop when it has conveyed its meaning.

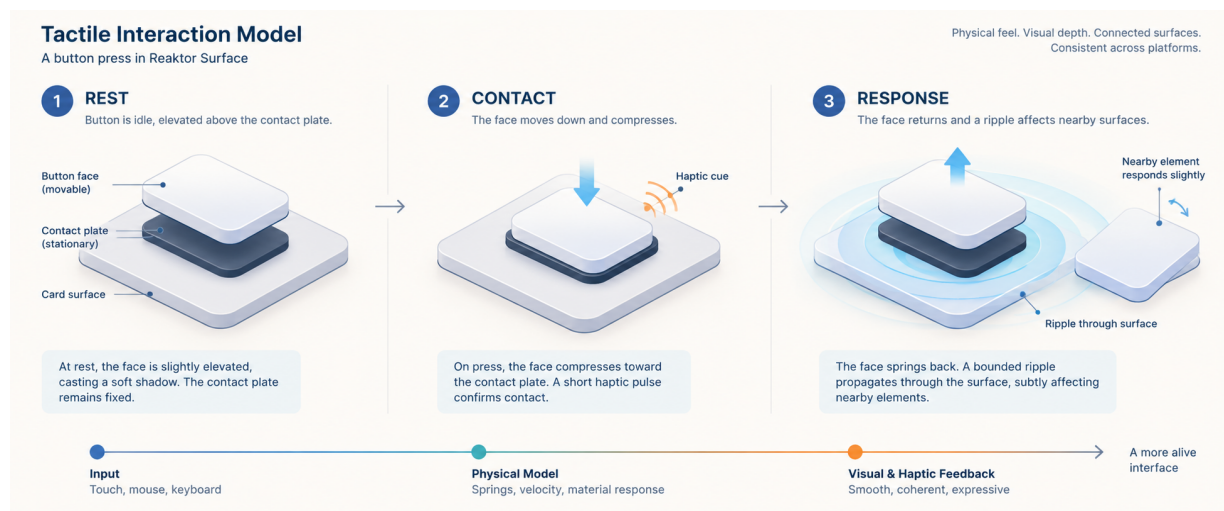


Figure 34: A conceptual tactile cutaway separates the moving face from its stationary contact plate. Contact produces local visual and optional haptic feedback. A bounded response can also affect nearby surfaces; none of these effects establishes application success.

A quiet interface can use the same model with almost no movement. The architecture remains the same because material response is selected by recipes and user policy rather than embedded in application workflows.

19.2 A spring is a model of response, not just a curve name

Let x be displacement from a target, $\dot{x}$ velocity and $\ddot{x}$ acceleration. A simple driven spring satisfies:

$$m\ddot{x} + c\dot{x} + kx = F.$$

The mass m controls resistance to acceleration. The stiffness k controls the restoring force for a displacement. The damping coefficient c removes motion. The external force F represents an applied drive. These quantities play different roles even if a design tool exposes them as nearby sliders.

For ordinary control animation, normalized parameters are often more convenient:

$$\omega_0 = \sqrt{k/m}, \quad \zeta = \frac{c}{2\sqrt{km}}.$$

The natural angular frequency ω_0 controls the characteristic speed of response. The damping ratio ζ describes the amount of damping relative to the system's critical value. A backend expecting a damping ratio must not receive the raw damping coefficient unchanged.

Spring response: stiffness and damping describe different properties

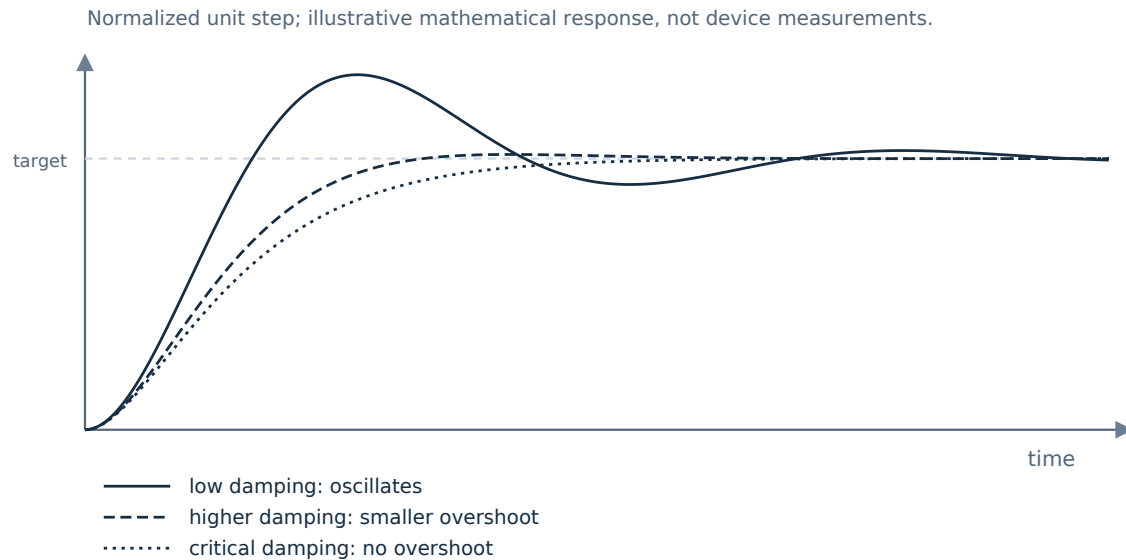


Figure 35: Illustrative normalized spring responses to a changed target. Lower damping permits more oscillation. The curves are generated from a mathematical response model, not measured performance or device-calibrated material data.

The graph makes the distinction visible. A spring that oscillates less is not necessarily stiffer; damping can change overshoot without expressing the same change as stiffness. Named profiles such as Rubber or Steel are useful experience presets, but they should not be presented as experimentally accurate models of those materials.

19.3 Ordinary controls use bounded response profiles

Most components do not need raw physical parameters. A small set of profiles can express a quick contact, a soft selection response or a restrained panel transition. A recipe selects a profile and an amplitude appropriate to the component.

The driver retargets from the current state when supported. It settles once displacement and velocity are sufficiently small. It then stops idle work. After backgrounding, a large elapsed interval is handled through a defined resume policy rather than thousands of catch-up steps.

Fling behavior needs its own model. Velocity-proportional drag and constant opposing friction are different dynamics. A field named `friction` is insufficient unless its units and equation are stated. A shared API should describe the behavior expected by the component rather than letting each renderer guess at a similarly named parameter.

The simplest satisfactory model is preferred. A single control can use the platform's spring implementation. Shared simulation becomes valuable when several participants need to respond to one local field

or a coordinated physical relationship.

19.4 Explicit coupling and spatial coupling

Explicit coupling describes a named relationship. The Send button can affect the composer plate that supports it. The relationship exists even if the two visual elements are not near each other according to a generic distance rule.

Spatial coupling describes participation in a local field. An impulse spreads from a location, and visible participants respond according to their position and material policy. Each receiver need not know every possible emitter in advance.

These relationships are inspectable, but they are not application service edges. A data dependency does not mean that two components should shake together. A navigation edge does not define a physical neighborhood. The effect scope has its own meaning within the graph-owned presentation boundary.

19.5 The field is the decoupling point

A button emits an impulse. A card samples the field. A procedural background consumes field parameters. The button does not call either receiver's implementation directly.

```
interface TactileField : ResourceLease {
    fun emit(impulse: FieldImpulse): Check<Unit>
    fun sample(position: Point2, atSeconds: Double): FieldSample
    val hasActiveWork: Boolean
}
```

An impulse identifies its scope, source instance and interaction sequence. Its origin is expressed in the field's logical coordinate system. It includes a monotonic time, bounded amplitude and maximum lifetime. Optional directional information can support a fling or impact response.

The field validates those inputs. A source outside the scope cannot create an unrelated cross-window effect. Duplicate cues are not re-emitted indefinitely. Expired impulses leave the active set, allowing the owner to stop frame work when no response remains.

19.6 A useful field does not require a fluid solver

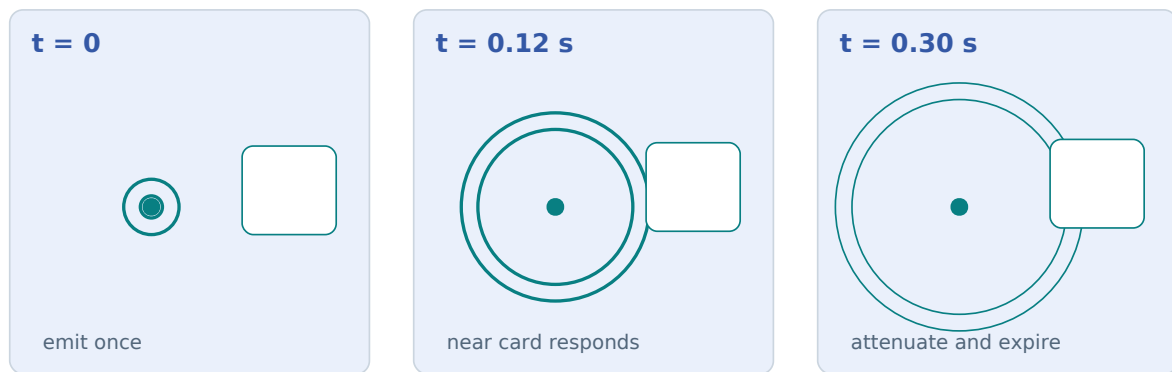
A first implementation can evaluate a bounded set of analytic impulses. For a point p at time t , one visual displacement model is:

$$d(p, t) = \text{clamp} \left(\sum_i A_i a(r_i) q(\tau_i) w(r_i, \tau_i) \right).$$

Here A_i is an impulse's amplitude. The distance r_i measures how far the point is from its origin. Its age is τ_i . The function a attenuates distance, q decays time, and w supplies the visible wave shape.

Each term has an understandable job. Attenuation prevents a small press from affecting the entire interface. Decay makes the response disappear. The wave function can create an outward-moving ring. Clamping prevents a combination of impulses from producing an excessive displacement.

A bounded impulse spreads through a local effect field



Spatial reach, amplitude and lifetime are limited. Receivers do not re-emit the input automatically.

Figure 36: A local impulse expands and decays. Participants respond while its influence reaches them. The field is bounded in space, time and magnitude rather than being an application-wide broadcast of perpetual motion.

This is a visual kernel, not a physically calibrated water model. A learning application that teaches fluid dynamics must not treat it as a substitute for its scientific simulation. The UI field exists to communicate interaction, not to redefine domain calculations.

With k active impulses and n visible participants, direct sampling takes $O(kn)$ evaluations. Bounding both counts provides a simple initial performance model. Spatial indexes or GPU buffers are optimizations to introduce only when a measured workload justifies them.

19.7 Prevent accidental feedback loops

The motion of a receiver must not automatically become a new input impulse. Otherwise a pressed button moves a card, the card emits another impulse, and the resulting movement repeatedly excites the original button.

Secondary effects can be designed deliberately, but need attenuation, depth limits and event identity. The default is one admitted interaction cue followed by bounded downstream response. This makes the effect explainable in the inspector and reproducible in tests.

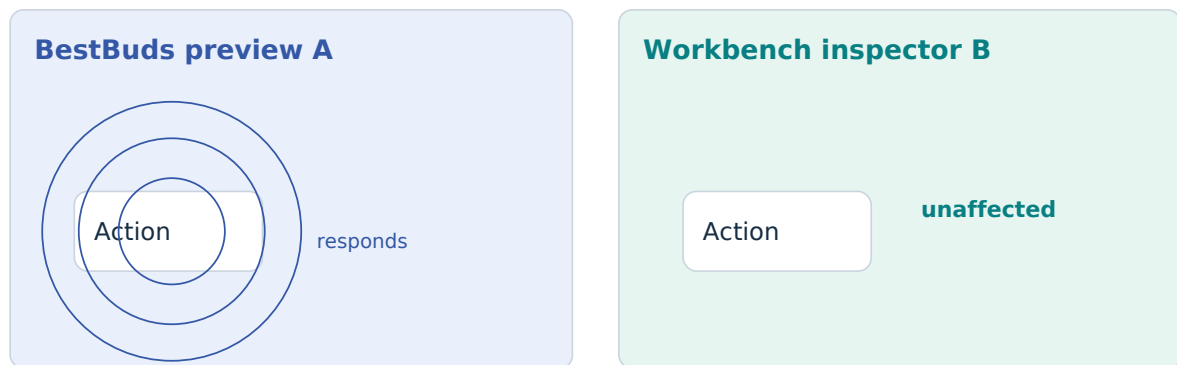
Coordinates also need ownership. Scrolling changes the relationship between a component and its field. Reparenting can change its scope. Device pixels are not interchangeable with logical field coordinates. The renderer supplies transforms and updates them when layout changes.

19.8 Material response must preserve usability

Decorative motion should not make controls difficult to acquire. A hit target can remain stable while a face moves a small amount within it. Actual draggable content, by contrast, needs input geometry that follows the intended document or viewport transformation.

A field should not distort the meaning of text, move a destructive action unpredictably or create a visual duplicate that can also activate. Accessibility bounds and focus remain associated with the semantic control.

Effect scopes prevent unintended cross-window coupling



Sharing application data does not join local tactile fields. Coupling must be explicitly declared.

Figure 37: Separate mounts and previews have separate effect scopes unless coupling is explicitly declared. Shared business data is not permission for one window's interaction to disturb another.

The reusable mechanics of BestBuds' `SignalTile` provide a concrete implementation seed: a fixed support, a moving face and local observations of press, hover and focus. Product-specific shapes and social color roles remain in the product's design package. 33

The system is successful when the relationship feels coherent and can still be turned down. Rich expression should be a selected capability, not a mandatory cost imposed on a quiet form or a low-power host.

20. Procedural materials, shader backends and sensory output

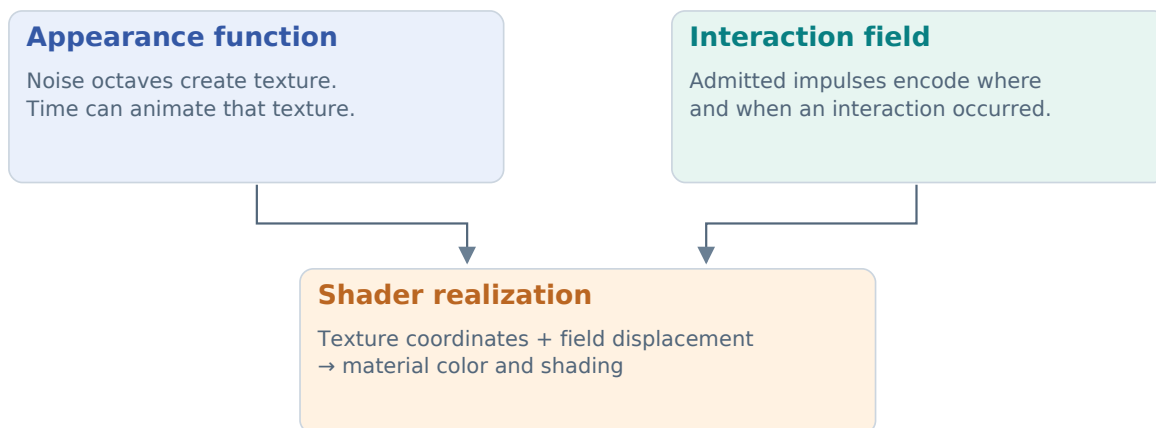
20.1 Appearance and interaction are different inputs

A procedural material is an image generated from functions rather than only a stored bitmap. Noise can supply texture. Time can vary that texture. Color ramps and lighting approximations can suggest a material.

Fractal Brownian motion combines noise at several scales. It is useful for layered variation, but it does not by itself describe the wave caused by a button press. The interaction field supplies that causal information. The shader combines the material's appearance function with the field's bounded influence.

34

Procedural appearance and causal interaction are separate inputs



fBm supplies variation. It is not a fluid simulator and does not know whether an action succeeded.

Figure 38: Procedural texture describes how a material looks. The field describes the effect of admitted interactions. Separating them allows the background treatment to change without rewriting every input component.

For example, the field can offset texture coordinates around an impulse. A different shader can use the same signal to alter normal direction or color. A nearby card can use it as a small translation. All consume one causal model without sharing one rendering implementation.

20.2 Shader support is qualified per backend

Android AGSL and Skia runtime-effect facilities are related choices, but they are not one universal runtime class. A shader package identifies its backend language, uniforms, texture inputs, color and coordinate conventions, and fallback. Each supported realization is compiled and tested separately. 3536

WasmGC support is only one condition for a CMP web-canvas path. It does not establish that a device has enough graphics capacity for an arbitrary effect, or that a shader backend exposes every expected

feature. Capability detection and the renderer profile determine what can be loaded.

An RSD web interface can use a qualified web effect, a static material, or an explicitly integrated CMP canvas island. That island must not steal focus or pointer input from foreground controls. Its clipping and coordinate transforms must follow the host. Loading an entire canvas runtime merely to draw a normal button is not the default architecture.

20.3 Cache programs, update uniforms

A shader program and the current time have different lifetimes. The program is compiled and cached using its source and backend fingerprint. Time and field parameters change during drawing. Updating those uniforms should not recreate the compiled program every frame.

The same distinction applies to cached drawing resources. Stable geometry or textures can remain in a resource scope. Values that vary per frame are supplied through the local draw path. A convenient cache wrapper can still be misused if its dependencies include a changing time value that causes repeated reconstruction.

Effects have explicit cost limits. A profile can restrict noise octaves, texture samples, resolution scale, off-screen layers and blur. Hidden decorative work pauses. Graphics-context loss selects a readable fallback while resources are recreated according to the host's policy.

Foreground text remains independently readable. Reducing the resolution of a background effect must not lower the quality of the text or interfere with its semantic representation.

20.4 Sensory feedback is event-driven

The sensory adapter maps an admitted cue to haptics or optional sound. Reaktor's existing semantic haptic interface uses named cues rather than assuming the same duration and amplitude have the same meaning on every device. 37

```
interface SensoryAdapter {  
    fun play(cue: FeedbackCue, intensity: Float)  
}
```

The adapter applies capability checks, user preferences, rate limits and lifecycle policy. An unavailable vibrator does not make the application action fail. Audio should coexist with media and accessibility output, and it must not be the only indication of an important result.

A physical impact can help select a bounded intensity profile. It does not prove perceptual equivalence across devices. Device qualification is still needed before describing two responses as equivalent experiences.

20.5 Coordinate the channels without conflating their purpose

Visual, haptic and audio outputs can share the same cause while following different preferences. Reduced visual motion does not automatically mean sound should be enabled. A silent device does not require disabling a focus indicator. A component's semantic state remains available even when every optional sensory channel is off.

Contact feedback and result feedback carry different information

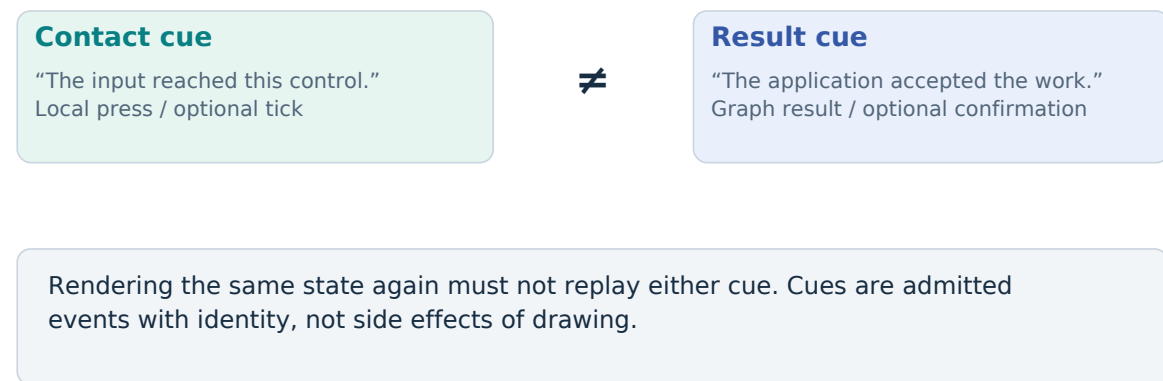


Figure 39: A contact cue acknowledges input. A result cue reflects application state. Both are one-shot events; neither is replayed simply because the same visual state is rendered again.

The effect system therefore coordinates events, not compulsory output. The recipe describes eligible responses. The host filters them through capabilities and preferences. The renderer and sensory adapters execute only the permitted channels.

A deterministic test can verify that one cue produces at most one request to each enabled adapter. It cannot by itself establish that the result feels comfortable. Perceptual review and actual device tests remain part of the component's conformance process.

21. Rendering efficiency, scheduling and memory

21.1 Performance begins with the dependency that changed

A control function running again is not the same event as a layout pass, a draw, a host-tree publication or a frame reaching the display. An optimization that reduces one count can leave the expensive work untouched.

A useful investigation starts with a concrete change. A message reaction changed. Which state projection observed it? Which component recalculated? Did layout change? Was the whole conversation encoded across the bridge? Did an image request restart? These questions connect the visible symptom to actual work.

A local change should not become a whole-screen update

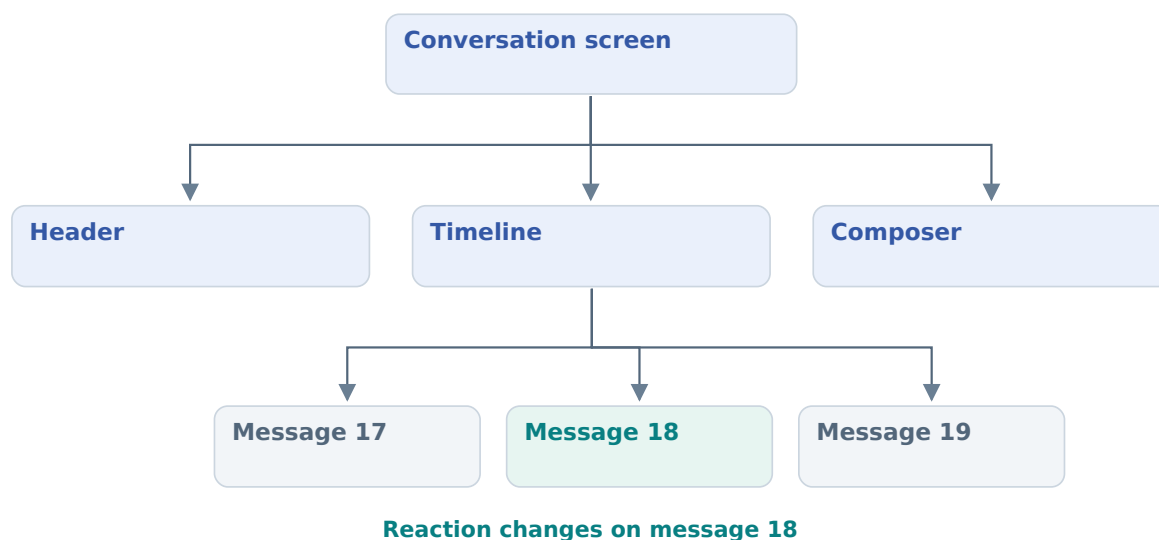


Figure 40: A reaction change belongs to one message’s relevant state projection. The architecture should not require every conversation region to rebuild or cross a runtime boundary for that local change.

The governing rule is: **invalidate the smallest correct scope, at the latest stage that can express the change.** The word “correct” matters. A stale component that never rerenders is not an optimization. A font change that genuinely affects geometry must invalidate measurement.

21.2 Recomposition and rerendering are normal mechanisms

Declarative frameworks recalculate output when dependencies change. That is their normal operation, not inherently a defect. The performance question is how much work the recalculation causes and whether it meets the interaction deadline.

In Compose, stable identity, dependency placement and compiler behavior influence skipping. Strong skipping changes which restartable composables can be skipped and how some parameters are compared. It does not make false immutability annotations safe or remove the need to model state correctly.

38

In React, narrow subscriptions and coherent snapshots come before indiscriminate memoization. A component may still update because its own state or consumed context changed. Surface's graph bindings should expose useful projections rather than force every control to subscribe to one rapidly changing application-root value.

A model aggregate is still appropriate when several fields must change together. The aim is not to split everything into independent streams. Doing so can expose inconsistent intermediate combinations. Dependency granularity and transactional coherence have to be designed together.

21.3 Cache according to actual dependencies

A resolved button recipe can be cached by the inputs that determine it. A measurement can be cached by text, font metrics and constraints. An image decode can be cached by asset identity and target dimensions. These caches have different keys because they preserve different results.

Incorrect keys create subtle stale behavior. A text measurement cache that ignores locale or font change can clip content. A recipe cache that ignores reduced-motion policy can retain an unwanted transition. A resource key that ignores authorization scope can expose private media incorrectly.

The reference correctness path remains available. An incremental projection can be compared with full recomputation. A cached measurement can be checked after relevant environmental changes. Optimization should reduce repeated work without silently changing meaning.

21.4 Schedule immediate and speculative work differently

Typing, focus and current pointer feedback are urgent. Preparing a panel that might be opened next is speculative. If both compete without priority, the system can spend resources on an invisible future feature while delaying the current input.

Bounded concurrency is therefore a first-class execution policy. Asset requests, host commits and expensive preparations should not accumulate without limit. Cancellation must release work that is no longer useful. Coalescing is appropriate only when the older value can actually be discarded.

React's scheduling transitions can make eligible rendering updates interruptible, but they do not move arbitrary work to another thread and are not visual animation transitions. Surface keeps immediate editing feedback outside delayed presentation work. 39

For CMP, the same architectural priority applies even though the scheduling mechanism differs. The renderer and its frame clock remain the owners of immediate presentation. Graph state processing and resource preparation should not hold a UI-thread critical section unnecessarily.

21.5 Frames are deadlines shared with the rest of the system

At 60 Hz, a frame interval is approximately $1000/60 = 16.7$ milliseconds. At 120 Hz, it is approximately 8.3 milliseconds. These intervals are not budgets available exclusively to Surface code. Input, layout, drawing, compositing and other system work share the path.

Average frame rate can hide pauses. A workload may be smooth most of the time but repeatedly stall when an image decodes or a panel opens. Qualification should examine deadline misses, latency distri-

butions and sustained behavior on named devices and release-like builds.

The portable contract does not promise identical performance across runtimes. It promises a measurable execution model and target-specific budgets. A capability can select a reduced material treatment when the full effect exceeds the supported profile.

21.6 Memoization, caching and pooling are different techniques

Memoization retains the result of a computation for repeated inputs. A cache retains useful data or a resource for future use. A pool retains storage or objects so they can be reset and reused for a different operation.

These techniques solve different problems. Pooling ordinary props or route entries can complicate identity and retain memory without improving the expensive part of the workload. Buffer reuse can help a measured serialization hotspot, provided ownership is explicit. Android’s memory guidance warns that pools can be counterproductive because of retention, reset and synchronization costs. 40

Surface therefore uses pooling selectively. A lease or ownership rule must prevent a buffer from returning to a pool while another runtime still reads it. “Zero copy” is a lifetime obligation, not a label that removes ownership costs.

Memory budgets are expressed in bytes as well as object counts. One retained image can be more expensive than thousands of small model values. Hidden views can release textures or editor resources while retaining the workflow state needed to restore them.

22. Assets, lazy loading and resource delivery

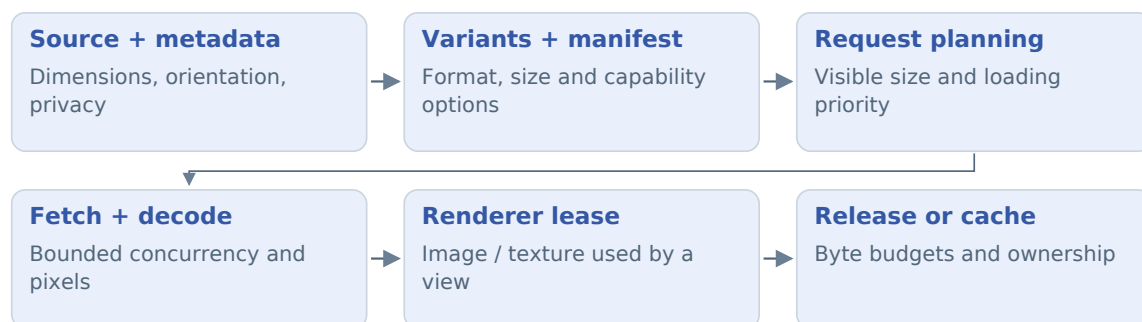
22.1 An image has several sizes and several lifetimes

An image has an encoded transfer size, decoded pixel dimensions, a displayed logical size and potentially a graphics-resource footprint. These quantities are related but not equal.

A highly compressed 4096 by 4096 image still needs 64 MiB for one four-byte-per-pixel decoded buffer. Additional copies and graphics resources are separate. Downloading fewer bytes does not guarantee a small memory footprint.

This is why an asset system needs more than a preferred format. It must choose variants, reserve geometry, prioritize work, decode at useful dimensions, account for memory and release resources when their owners stop using them.

An asset has a delivery and a memory lifecycle



Transferred bytes, decoded bytes and GPU residency are different resource costs.

Figure 41: The resource pipeline covers both delivery and ownership. A format decision affects transfer, while size-aware decoding and resource leases affect memory and rendering cost.

22.2 The manifest makes asset requirements explicit

An asset manifest identifies the resource and its available variants. Useful metadata includes dimensions or aspect ratio, encoded format, byte size, color and orientation information, fallback choices and cache/privacy policy.

The renderer can then request a variant appropriate to display constraints and pixel density. A thumbnail does not need the original camera image. A zoomable editor can request more detail when the viewport actually requires it.

```
data class AssetRequest(  
    val asset: AssetId,  
    val displaySize: LogicalSize,  
    val pixelDensity: Float,  
    val priority: LoadPriority,  
)  
  
interface AssetResolver {  
    suspend fun acquire(request: AssetRequest): Check<AssetHandle>  
}
```

The returned handle is a lease. The component owns its use while visible or retained under policy. The resource system can cache underlying data independently, but must respect its byte budget and authorization scope.

22.3 Choose formats for the asset's job

Photographs can use modern compressed formats such as WebP or AVIF where the target decoder and quality requirements support them. Simple icons often benefit from vector representation. Pixel-sensitive artwork can require lossless encoding. Video, animated images and fonts need their own delivery policies. Web image guidance treats correct dimensions and delivery behavior as important alongside format choice. ⁴¹

A single mandatory format would obscure these differences. A vector can be small yet computationally complex. A photograph can be tiny on the screen but enormous after decoding. A font subset can save bytes while accidentally omitting a script the product must display.

The manifest records what the runtime needs to choose intelligently. Untrusted media is also bounded: encoded and decoded limits, resource-origin checks and a policy for active or external content in vector assets. Privacy metadata should be removed where product policy requires it, rather than copied blindly into public delivery.

22.4 Reuse capable loaders, with the right requests

Surface should configure a proven platform loader rather than replace all image fetching and caching. For Compose, Coil's `AsyncImage` resolves loading size from constraints, while `rememberAsyncImagePainter` requires a suitable size resolver to avoid original-size loading by default. ⁴²

The architectural contribution is the request policy. It supplies intended display size, density, priority, privacy scope and cancellation. The loader supplies its specialized decoding and cache mechanisms. A custom resource system should not discard that leverage without a demonstrated need.

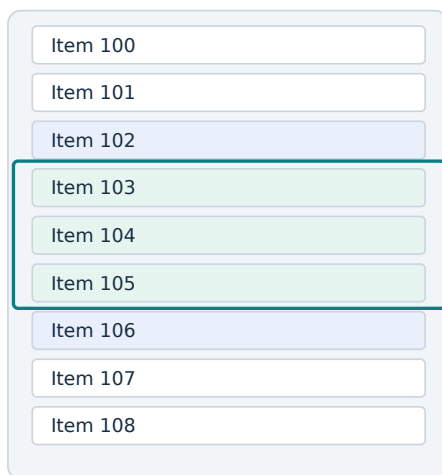
Theme changes should only affect dependent resources. Tinting an existing icon need not redownload its bytes. Changing a typography family does require a new font resource and remeasurement. A different decode size belongs in the relevant cache key.

22.5 Lazy loading is not one switch

Code splitting defers feature code. Data pagination defers records. Virtualization defers view instances. Asset lazy loading defers media requests. Deferred initialization postpones an expensive subsystem. Prefetch deliberately performs likely-next work before it is requested.

Virtualization bounds mounted work; pagination bounds retrieved data

Logical dataset



Retrieved pages

Data may include items not currently mounted.

Visible + nearby rows

Only this bounded window needs active UI, measurements and resource requests.

Use stable item identities so recycling or reordering does not move a draft into another row.

Figure 42: Pagination limits retrieved data, while virtualization limits mounted view work. A small overscan region can prepare nearby rows without creating active UI for the entire dataset.

These controls should be independent. A timeline may retain several pages of records while mounting only the visible rows and a small surrounding region. A heavy editor can load when opened without making the surrounding page wait. Prefetched assets can be cancelled when navigation changes.

Priorities distinguish critical, visible, nearby and speculative work. A likely web Largest Contentful Paint image should be discoverable early rather than hidden behind a blanket lazy-loading rule. High priority is selective; assigning it to every resource removes its usefulness. ⁴³

22.6 Preserve geometry while resources arrive

An image placeholder should reserve its known aspect ratio. A skeleton should approximate the eventual layout rather than begin as a small spinner that later becomes a large panel. Fonts need deliberate fallback and loading behavior.

A font change can still cause legitimate reflow. The goal is to reduce avoidable shifts and preserve a meaningful anchor. Web font guidance discusses format, subsetting and loading choices; those choices must be combined with the application's script coverage and layout requirements. ⁴⁴

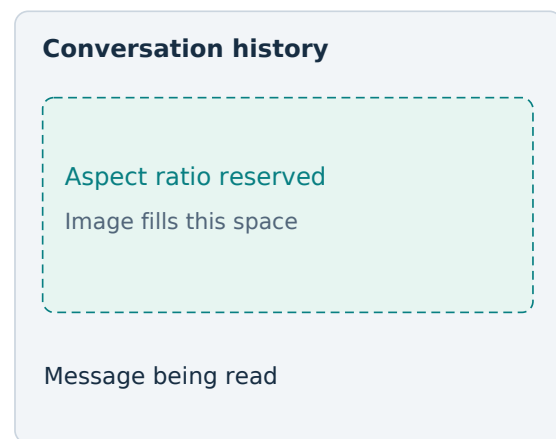
Private resources have an additional lifecycle. Logout, account switching and permission revocation must affect their use and cache policy. A content hash identifies bytes; it does not grant every account permission to view them.

Reserve media geometry before its bytes arrive

Unreserved: text jumps



Reserved: text stays anchored



Late content is allowed. Unexplained movement of the user's reading target is not the default.

Figure 43: Known media geometry is reserved before the image arrives. The resulting content can fill the region without moving the message the person is reading.

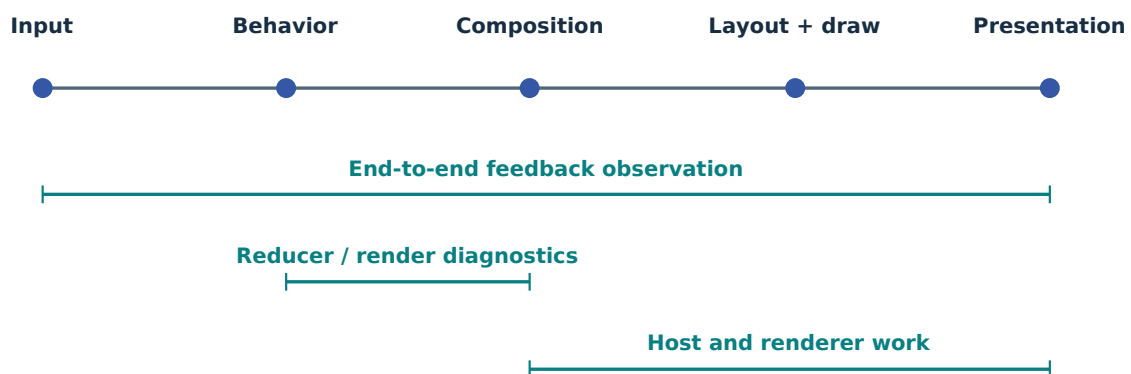
23. Measuring web, native and custom interfaces

23.1 Start with the user-visible question

“Is this interface fast?” is too broad for one measurement. A conversation needs quick contact feedback, responsive editing and stable scrolling. A commerce page needs essential content to appear promptly. A graph editor needs selection and drag feedback to remain responsive while a large scene changes.

Each question defines a scenario and an observation boundary. A component function completing does not automatically mean the person has seen the result. A frame callback is not a direct measurement of photons reaching the display. The metric must state what it actually observes.

Metrics answer different questions along the interaction path



A function finishing or a frame callback firing is not automatically proof of physical display time.

Figure 44: Different measurements cover different portions of the path from input to visible result. Diagnostic render timing is useful, but it does not replace end-to-end interaction measurement.

23.2 Preserve standardized metric definitions

Core Web Vitals use Largest Contentful Paint, Interaction to Next Paint and Cumulative Layout Shift. Their good thresholds are 2.5 seconds, 200 milliseconds and 0.1 respectively at the 75th percentile, with appropriate segmentation. Custom Surface metrics do not reuse those names for different events. 45

Field measurements and laboratory scenarios complement each other. Field data describes supported users and conditions. Laboratory tests help reproduce a regression. Neither a single load score nor a component gallery can establish every interaction’s performance.

Layout Shift also has a specific definition. It is not a complete measure of all movement a person may find disruptive. An application should separately test scroll anchors and active controls during content insertion. 46

23.3 Native collectors have different coverage

Android vitals and Apple MetricKit provide platform-specific quality and diagnostic signals. They differ in what they collect, where they collect it and how results are delivered. Surface preserves those distinctions rather than collapsing them into one “app vitals” number. 4748

A custom-rendered canvas needs particular care. A platform metric about ordinary scrolling may not cover its actual implementation. A browser’s normal content measurement may not describe when the graph drawn inside a canvas became useful.

The framework therefore defines additional scenario metrics where needed: navigation-to-usable-content, selection-to-inspector-ready, action-to-visible-feedback, decode duration, bridge backlog and retained-resource size. Each declares its start event, end event, clock, sampling and supported targets.

23.4 Correlate measurements without overwhelming the graph

The graph provides useful context: owning feature, route kind, action contract and active implementation. Surface observations can use that context in traces and diagnostics.

High-rate samples should remain in bounded local buffers or sampled traces. A span for every animation tick would add overhead and unnecessary volume. Metric labels remain bounded; individual message text or arbitrary component-instance IDs do not become global metric dimensions.

```
interface SurfaceObserver {  
    fun record(owner: OwnerId, category: String, durationNanos: Long)  
    fun finding(owner: OwnerId, finding: Finding)  
}
```

The observer is a boundary for measurements, not permission to export private state. Inspection and telemetry apply redaction, access and retention policy. The inspector itself has an overhead budget so observing the interface does not become the dominant source of missed deadlines.

23.5 Explain the cause, not just the number

A useful finding identifies what changed and which work followed. It can report that a font revision invalidated text measurement, that a broad subscription rebuilt several regions, or that an image was decoded far above its displayed size.

React’s Profiler supplies render timing, but layout, paint and end-to-end interaction still need the relevant collectors. 49 Compose likewise benefits from tracing the stages that actually consumed time rather than counting every recomposition as equally expensive.

Qualification uses named release-like workloads. Android Macrobenchmark provides a suitable mechanism for scenarios such as startup and scrolling; equivalent target-specific procedures are needed elsewhere. 50

No budget should be described as a measured result before the workload is run. The architecture defines where evidence comes from and how regressions are attributed. The product’s supported profiles determine the actual thresholds it must meet.

24. Dynamic UI through graph updates

24.1 The update mechanism is outside Surface

Reaktor's execution system is assumed to support checked replacement of eligible subgraphs through reaktor-graph, reaktor-ffi and Hermes. Surface consumes the activated result. It does not implement another deployment pipeline, code-delivery service or authority for structural application change.

The UI-facing requirement is concise: a supported update activates a coherent definition and capability binding for a region. Surface identifies the affected mounts, reconciles or replaces their presentation according to the boundary contract, and prevents stale work from the retired generation from entering the new one.

A subgraph update changes the UI through the same owner boundary

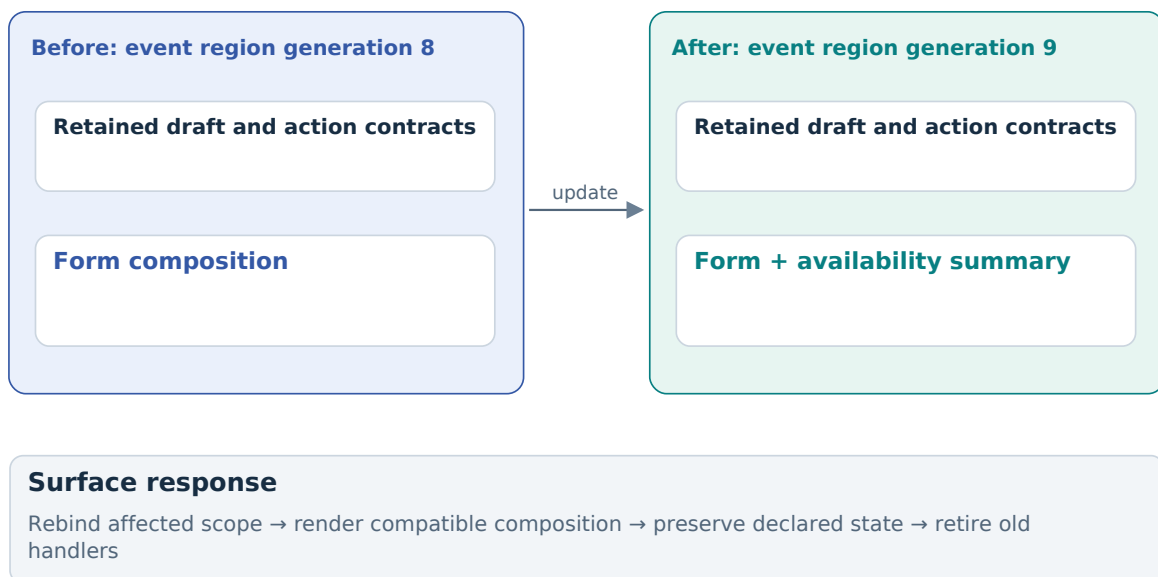


Figure 45: A supported subgraph update changes the affected composition while retaining declared state owners and boundary contracts. Surface responds to activation; it does not duplicate the update mechanism.

The installed renderer must still be able to realize the activated component contracts. Describing a new native capability in a graph does not install an unavailable native implementation. The activation system's compatibility result remains the prerequisite for presentation.

24.2 Three kinds of change remain different

An application-state update can change the text or availability of existing controls. A composition update can add or remove instances inside one owned boundary. A graph-definition or implementation update can replace a supported feature region.

These changes can all affect the screen, but they do not need the same heavyweight path. A keystroke is not a graph topology rewrite. A local animation sample is not a feature update. A new generated

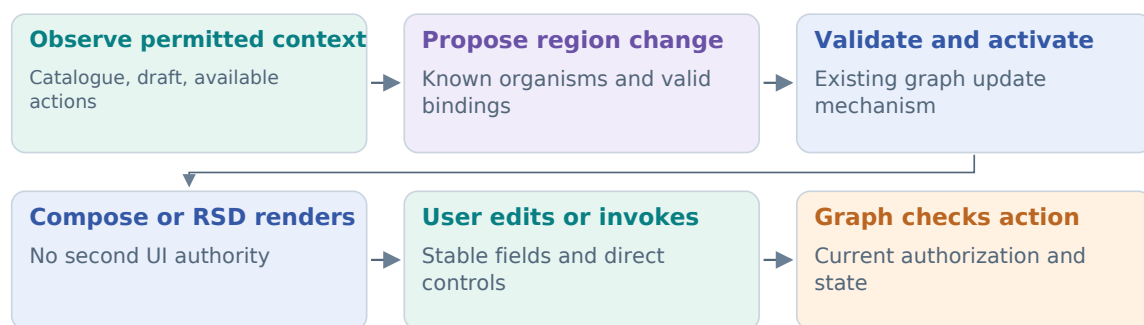
workflow region is more than a hover observation.

Surface therefore observes normal state changes through bindings and receives activated structural changes through the graph integration. This preserves the fast local loop while keeping application evolution governed by the graph.

24.3 A generated region is still graph-owned

A runtime agent can select known components, provide content and propose a supported arrangement within an approved region. The proposal refers to the application's catalogue and available action contracts. It cannot create authority merely by naming a port or rendering a persuasive button.

An agent changes an approved graph-owned UI region



Generation is optional. The same application remains usable through ordinary controls.

Figure 46: The agent observes a scoped view and proposes a change to an approved region. Validation and activation use the existing graph mechanisms. The resulting interface is realized by CMP or RSD and retains ordinary direct controls.

The graph change can update a region's configuration or its supported implementation. The renderer then performs normal composition. There is no second AI-only component library and no independent runtime tree that the agent may mutate while the frontend owner continues writing to it.

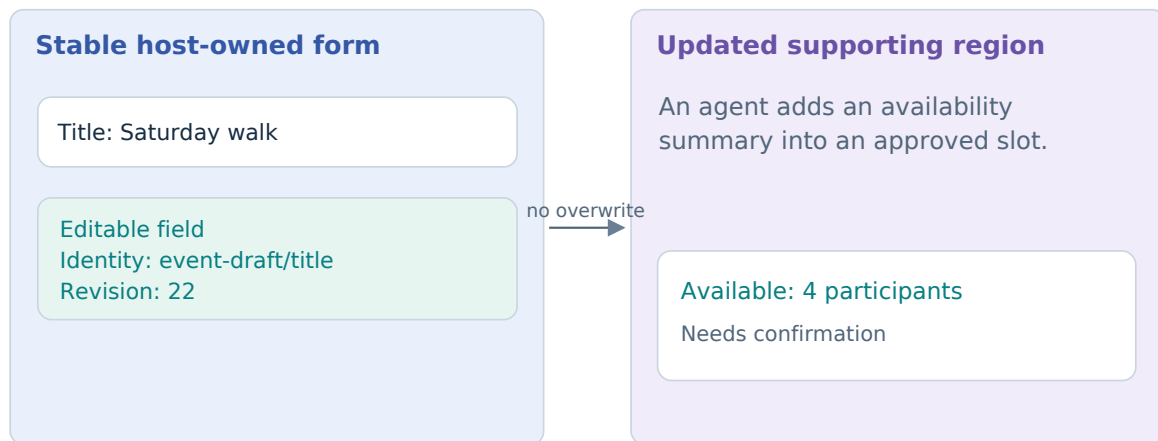
Executable generated code is not made harmless by a component allowlist. Such code requires the execution system's supported trust and capability profile. Surface's catalogue and input validation are necessary constraints, not a claim of complete hostile-code isolation.

24.4 Protect the interaction already in progress

A generated proposal may arrive while the person is editing. The existing draft remains associated with its owner and field identity. A new summary can fill an approved supporting slot without replacing that field.

If a structural change must replace the editing control itself, the boundary has an explicit handoff or defer policy. Active gestures and input-method composition cannot simply be discarded. The host provides a stable loading, error or fallback region that does not depend on the generated code being able to render itself successfully.

Generated regions preserve fields already being edited



The graph update changes the supporting subgraph, not the owner of the existing draft.

Figure 47: A generated supporting region can change without acquiring ownership of a field already being edited. Stable identity and revision policy preserve the person's work while new content appears.

24.5 Authorization remains at the application operation

An enabled appearance is not authorization. An action reference is not authorization. When the person or agent invokes the operation, the graph-owned application capability checks the current principal, resource and arguments at the appropriate authoritative boundary.

Trusted authentication, consent, payment and safety regions remain protected from arbitrary generated replacement. A model may suggest content or populate a draft without acquiring the ability to approve its own high-risk action.

Generation is optional. Every essential workflow retains a direct manual path, and generation failure preserves existing user work. This keeps dynamic composition an extension of the application rather than a dependency that ordinary interaction cannot survive without.

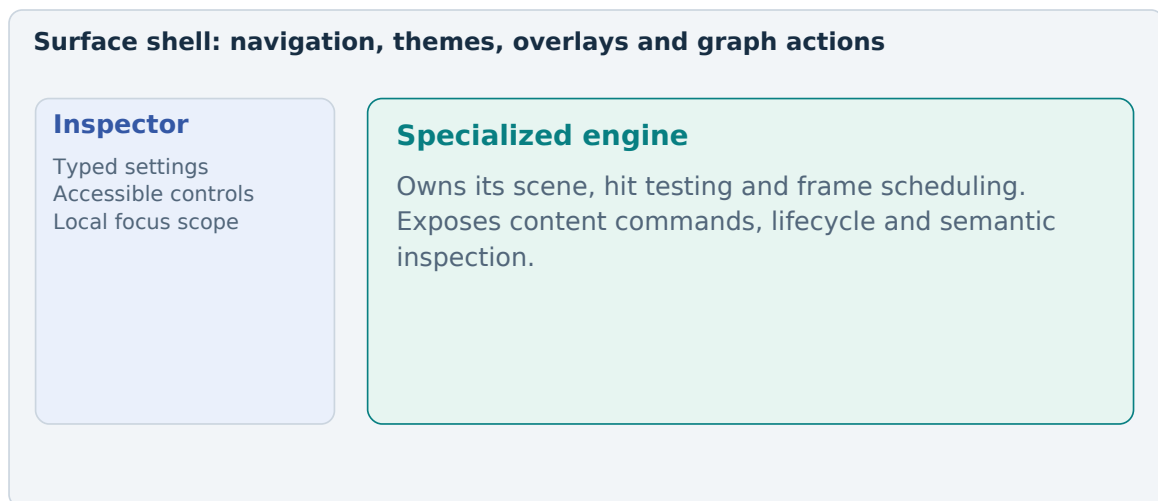
25. Custom interfaces and specialized engines

25.1 Not every interface should be built from ordinary boxes

A graph editor, waveform timeline, map or rich-text editor can need a specialized rendering and interaction model. Forcing its internals into a generic component tree may reduce performance and obscure its domain behavior.

Surface supports three levels of extension. Ordinary custom components compose existing catalogue entries. Custom layout and drawing components supply their own measurement and presentation. Embedded engines own substantial internal structures behind a registered boundary.

A custom engine lives behind a normal Surface boundary



A custom scene graph is a rendering detail, not permission to bypass application ownership.

Figure 48: The shared shell supplies themes, navigation, focus coordination and graph capabilities. The specialized engine retains its scene and frame machinery behind an explicit lifecycle and semantic boundary.

These extensions remain part of the architecture. They are not escape hatches from resource ownership, accessibility or application policy.

25.2 The extension contract describes the boundary

A custom engine specifies its sizing requirements, input and focus behavior, supported resource capabilities and cleanup. It exposes semantic actions and an inspectable representation or usable alternative for operations that cannot be understood from pixels alone.

A code editor can retain its mature editing engine. Its integration exposes document state, selection operations, focus, diagnostics and resource lifetime. It does not need to serialize every caret movement into a generic graph transaction.

A graph canvas can retain spatial indexes and geometry caches. Its committed application topology still belongs to Reaktor Graph. A local scene representation is a rendering projection, not a second source

of valid service connections.

25.3 Distinguish document change from presentation change

Dragging a graph node can show immediate local feedback. Committing the authored position is a separate operation on the document or graph owner. Hovering the same node should not change its saved coordinates.

A tactile theme may move a tool palette slightly. It must not distort the person's artwork unless that distortion is an explicit document operation. A scientific visualization must not substitute decorative motion for numerical results.

This distinction is essential for undo and collaboration. A committed document change has history and conflict semantics. Pointer samples and decorative offsets usually do not. They can be observed or shared at a bounded rate without being treated as durable authored content.

25.4 Semantic operation enables more than agents

A custom canvas should expose actions such as select object, move by a specified amount, inspect a port or propose a connection. These support keyboard users, assistive interfaces, tests and agents.

An agent can then operate the same capability as a person rather than guessing coordinates from a screenshot. It still follows authorization and review policy. A proposed connection is checked before it becomes authoritative, even if a local preview draws a tentative line.

When a required capability is unavailable, the extension selects an explicit alternate flow or reports the limitation. An empty rectangle that silently omits an essential control is not a valid degraded implementation.

26. Verification and implementation boundaries

26.1 Test the contract at several levels

The behavior kernel has headless tests. The renderer adapter has input and semantics tests. Recipes have supported-theme checks. Layout policies have constraint tests. Complete application workflows verify that these pieces cooperate while state changes and resources fail.

A button reducer test does not prove screen-reader operation. A screenshot does not prove cancellation. A successful host commit does not prove that an application action is authorized. Each test answers a specific question.

The catalogue can generate the component laboratory from the same definitions used by production code. The laboratory includes unusual but valid states: long labels, missing images, busy controls, larger text, right-to-left layout and reduced motion. Examples should be composable into actual workflows rather than confined to an attractive default gallery.

26.2 Preserve reference paths for optimizations

An incremental host projection can be compared with full recomputation. A compact encoded batch can be decoded and compared with its readable equivalent. A field response can be evaluated with a deterministic test clock and bounded inputs.

These reference paths help distinguish faster execution from changed semantics. If an optimization skips a necessary update, its improved timing is not a success. If a pool retains several large images indefinitely, fewer allocations do not imply a better resource policy.

Formal models can help with small state-transition questions, such as whether a retired generation can admit new events. They do not automatically prove arbitrary component code or perceptual quality. The scope and assumptions of the model remain explicit.

26.3 Use real failure scenarios

Representative scenarios include a theme switch while editing, an image arriving above the read anchor, a window resizing during a transition, a permission being revoked while a dialog is open, and a generated region arriving after the person changes its draft.

The bridge adds malformed values, stale generations and missing base revisions. Resource testing adds low memory, decode cancellation and graphics loss. Repeatedly opening and closing a heavy editor tests whether lifetime rules are actually implemented rather than merely documented.

A valid failure outcome can be a preserved last-known-good region, a deferred update or a clear unavailable capability. The system must not replace an unsupported path with an untested one while presenting the result as success.

26.4 Keep dependency direction acyclic

The shared model contains neutral values and behavioral contracts. Renderer integrations depend on that model. Graph integration connects those contracts to the existing application runtime. Product composition depends on the public Surface API and its chosen realization.

Toolkit-specific graph rendering must be separated before establishing a dependency direction that would otherwise make graph and UI depend on each other. The Reaktor graph and UI build definitions identify this concrete integration boundary. 5152

Physical modules are introduced when they provide real dependency, compilation or distribution isolation. A separate module per atom, animation or metric is unnecessary. Logical packages can establish the boundaries first, while a small number of modules enforce the relationships that matter.

26.5 The release criterion is a supported combination

A release is qualified for particular component, theme, renderer and environment combinations. Passing one Android example does not establish all desktop or RSD behavior. A shader working on one backend does not establish a portable material profile.

The evidence should show that required state and semantics survive supported variation, that local feedback meets its target profile, and that resource ownership remains bounded. The final Compose-first task plan turns these obligations into staged implementation work rather than requiring every future realization at once.

Part III — Applications and implementation

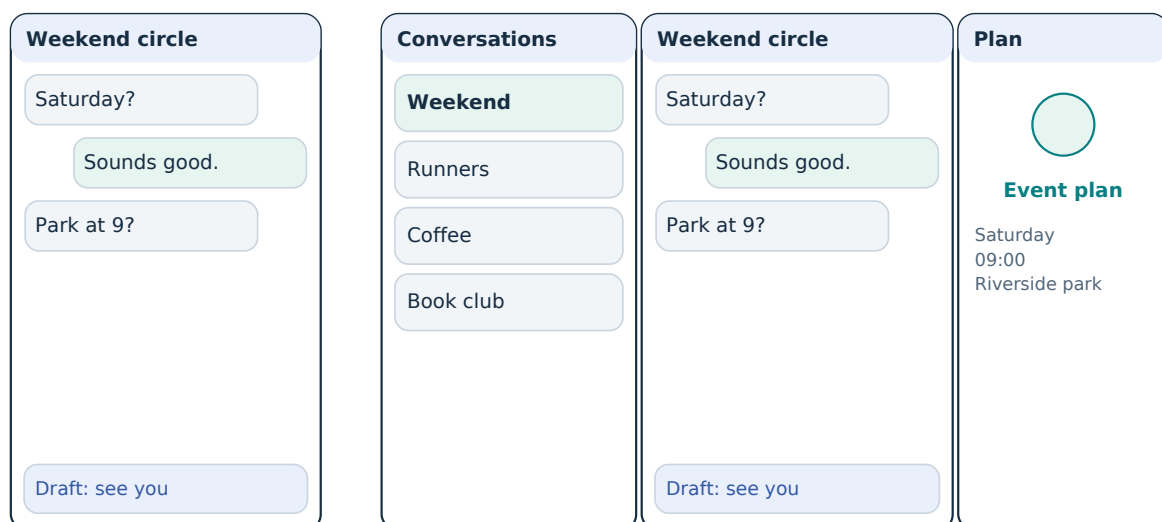
27. BestBuds: one conversation across several presentations

27.1 Start with the product's state, not its screen size

A conversation has messages, participants, delivery state and a draft. Those are application concepts. A conversation screen has a timeline viewport, a focus destination and a current layout. Those are presentation concepts. Keeping the two groups separate allows the same conversation to appear on a phone and in a multi-pane desktop window.

The feature binds a conversation model and typed operations from the graph. Its page composes a conversation-list organism, timeline, message composer and optional supporting content. Atoms such as text, avatars, fields and actions are reused, while campaign and social semantics remain product-specific.

BestBuds: one conversation, a focused phone and a richer desktop



The conversation session and draft are shared by contract; caret and scroll belong to each mount.

Figure 49: Schematic phone and desktop presentations of the same conversation. The larger presentation adds context rather than defining a second conversation. Each mount retains its own focus and scroll behavior.

A conversation session is keyed by its relevant account and conversation identity. A route entry identifies one navigation occurrence. A mount identifies the actual presentation. These identities are related but not interchangeable. Two mounts must not race to write a single global “current chat” value.

27.2 Follow one message from contact to outcome

A person types a short reply. The local editor updates immediately and publishes revisioned edits to the draft owner. The Send control becomes available when the model says the draft can be submitted.

On contact, the button compresses locally. Its supporting composer plate can respond to one field impulse. The response is bounded to the composer's scope; it does not shake every message that shares a service dependency. Optional haptics acknowledge contact.

Activation invokes the conversation's send operation. The graph-owned workflow can create a pending row and retain the operation identity needed for retry. The row displays pending state without claiming delivery. If the operation fails, retry and discard remain explicit capabilities. A later state observation does not replay the original contact haptic.

This separation makes the design language replaceable. A restrained desktop recipe can suppress the field and use a quiet pressed state. A more expressive social recipe can retain the moving face and background response. The delivery workflow remains the same.

27.3 Adaptation preserves the reading task

On a compact window, the active conversation occupies the main route presentation. On an expanded window, the list can appear beside it. A supporting profile or event summary can become a third region where the constraints permit it.

The presentation plan changes; the draft owner does not. A message that was being read remains the logical scroll anchor. When a photograph arrives above it, reserved aspect ratio and anchor restoration prevent an unnecessary jump to unrelated content.

The interface also respects larger text. It can reduce simultaneous supporting content rather than clip messages into fixed-height rows. A narrow inspector-style mount is treated as narrow even if it lives on a large monitor.

27.4 Generation adds a region, not a competing workflow

An authorized agent can propose an event-planning region inside the conversation. It uses approved participant, time and venue organisms and receives only the permitted context. The proposal becomes a checked change to that graph-owned region.

Once the change is active, the CMP or RSD realization renders it. A draft already being edited remains associated with its field owner and revision. The agent can supply a suggestion, but cannot approve its own publish action or replace trusted safety controls.

Progressive access is enforced at the data boundary. An unrevealed profile image is not delivered to an unauthorized client or model merely because a theme draws it blurred. Visual masking and authorization are separate mechanisms.

27.5 A concrete qualification session

Open a conversation, enter a draft and scroll to an older message. Change the theme, widen the window and load an image above the anchor. Then introduce a failed send, revoke one action's permission and fail the generated event proposal.

The expected result is specific. The draft and conversation session remain owned. The reading anchor remains meaningful. The failed operation has a recovery action. The revoked action cannot execute. Generation failure leaves a usable manual interface. Tactile contact feedback continues locally without misrepresenting any application outcome.

This scenario tests the architecture as a connected system rather than proving isolated components only in their default states.

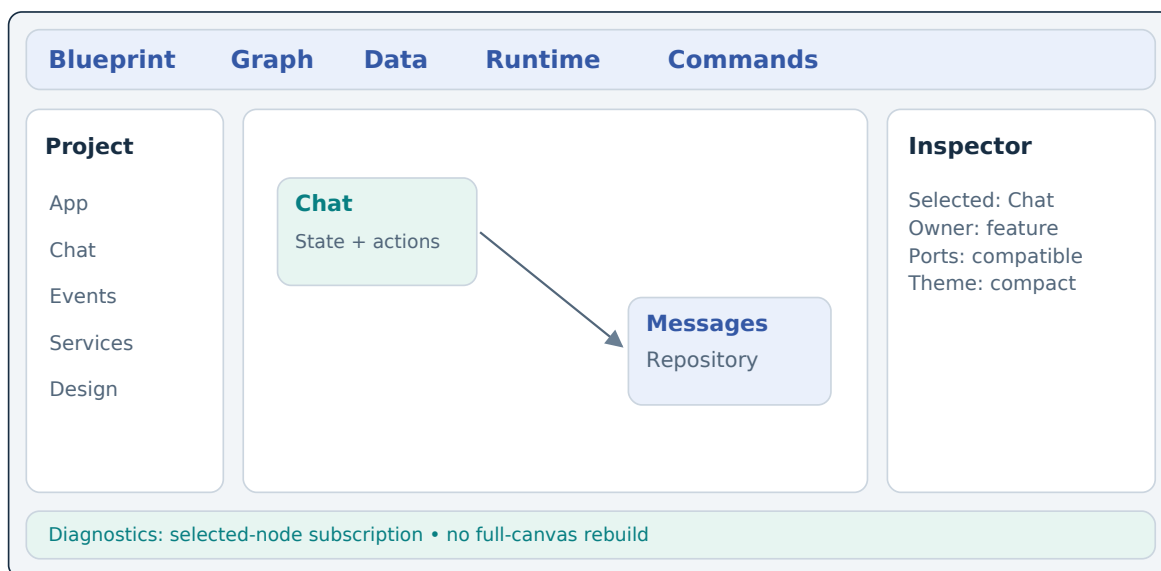
28. Blueprint: a dense workbench around a specialized canvas

28.1 The graph is the subject; the canvas is its presentation

Blueprint inspects and operates on application structure. Reaktor Graph remains authoritative for nodes, ports and valid connections. The canvas maintains the geometry and interaction structures required to display that graph efficiently.

These structures should not be confused. A canvas scene can contain control points and cached edge geometry that are not application graph entities. A displayed tentative wire is not yet an accepted application connection. The renderer must distinguish preview from committed topology.

Blueprint: shared controls around a specialized graph canvas



UI zoom changes the shell. Graph zoom changes canvas coordinates.

Figure 50: A schematic workbench combines catalogue controls with a specialized graph canvas. The inspector subscribes to selected entities, and diagnostics explain local work without forcing every telemetry update through the entire canvas.

The surrounding shell uses Surface controls for toolbars, menus, tabs, fields, dialogs and the command palette. Dense recipes adapt these controls to pointer and keyboard workflows. Their behavior contracts remain related to the mobile controls even when their spacing and presentation differ substantially.

28.2 A drag has two different results

During a node drag, the pointer produces local preview motion. The canvas updates hit testing and visual position immediately. It may publish bounded collaboration observations, but it does not need a full structural graph transaction for every pointer sample.

At commit, the authored position is applied to the appropriate document or graph owner. If the operation is a new connection, the graph validates the port relationship before it becomes authoritative. A rejected connection must disappear or remain clearly marked as an invalid preview, not look like a successful permanent edge.

Tactility can communicate the state of this interaction. A compatible port can receive a restrained attraction or highlight treatment. A rejected drop can produce a local response. These effects visualize the validator's result; they do not determine type compatibility themselves.

28.3 Different panes have different resource costs

A node inspector can be a normal Surface organism. A SQL editor or terminal may be a specialized island. A large canvas may need spatial hit testing, viewport culling and level-of-detail rendering. The shell coordinates their lifetimes without imposing the same internal representation on all of them.

Opening a heavy tool can load its implementation on demand. Hiding it can release expensive resources while retaining a document checkpoint. A theme change should not recreate every editor instance unless the implementation boundary explicitly requires it.

The inspector subscribes to the selected entity and its relevant state. A telemetry stream for unrelated nodes should not require rerendering the entire workbench. Detailed frame observations remain bounded and sampled so inspection does not overwhelm the system being inspected.

28.4 Zoom has two meanings

Workbench UI zoom changes chrome, spacing and available logical layout. Canvas zoom changes the graph's content coordinates. The two transforms are separate so increasing graph magnification does not also enlarge every toolbar and dialog.

Moving between displays can change pixel density. The renderer updates resource targets and coordinate mappings without changing application identity. Popup placement, hit testing and accessibility geometry follow the same declared transforms.

A tablet profile can prioritize inspection and a reduced set of simultaneous tools. A small phone profile may expose status and approved actions rather than scaling an entire desktop workspace into an unusable miniature. This is capability-based adaptation, not failure to share the design system.

28.5 Human and agent operations share the same authority

An agent can inspect contracts, explain a finding and propose a patch. It does not bypass review or directly mutate a frontend-owned subtree. A code-owned inspector remains source-owned; a supported visual declaration can be edited through its graph definition.

A qualification session combines dragging with telemetry updates, repeated heavy-editor opening and closing, a theme change, cross-display resizing and a stale update. Correct behavior preserves geometry and ownership, rejects invalid structure, bounds memory and attributes missed deadlines to actual work.

The key measurement is not when an empty canvas appears. It is when useful graph geometry and the required interaction are available.

29. Operational software: an overview and an offline field task

29.1 One domain can require very different pages

An order-fulfillment application has desktop staff reviewing exceptions and field staff capturing delivery evidence. Both work with the same records and permitted state transitions, but their immediate tasks differ.

The desktop page can show a virtualized table, filters, a detail panel and audit information. The field page can show a short sequence with large usable targets, capture controls and explicit local-save status. A tablet can combine the task list and detail when space permits.

Operations: desktop overview and offline field capture

The figure shows two side-by-side interface mockups. The left mockup, titled 'Orders and exceptions', is a desktop view featuring a table with four columns: 'Order', 'State', 'Quantity', and 'Action'. It lists four orders, with the third one highlighted in orange. The right mockup, titled 'Delivery capture', is a mobile view with three input fields labeled 'Delivery reference', 'Received quantity', and 'Evidence photo', followed by a 'Save on device' button.

Order	State	Quantity	Action
1042	Awaiting evidence	32 t	Inspect
1043	Accepted	18 t	Inspect
1044	Conflict to review	24 t	Inspect
1045	Pending sync	11 t	Inspect

Local capture and server acceptance remain separate states, with one explicit synchronization owner.

Figure 51: The desktop overview and mobile evidence task reuse status, form and action contracts. Their layouts differ because their work differs, not because every component contains an operating-system branch.

The shared assets are domain contracts, reusable organisms and semantic design roles. Reusing those does not require forcing a wide table onto the phone.

29.2 Offline recording is not remote acceptance

A field worker captures a photograph and enters a quantity. The local draft owner records the edits. The media system stores or references evidence according to its privacy and persistence policy. A local confirmation can mean that the device has retained the work.

The backend may not have accepted it yet. That distinction must remain visible. A successful press animation cannot move an order into an authoritative completed state. Synchronization exposes pending, accepted or conflicting results through the graph-owned workflow.

A restrained tactile treatment can confirm capture or a scan. It must not imply a stronger outcome than the application state supports. The same rule holds when optional sound or haptics are disabled: the visible status remains intelligible.

29.3 State conflicts deserve a designed interface

Suppose a desktop operator changes an order while the field device is offline. When the device reconnects, its draft may refer to an older revision. Automatically replacing the form with the server value can erase valid evidence. Automatically overwriting the server can erase the operator's change.

The domain defines a conflict policy. The UI presents that policy with the relevant values and recovery operations. Surface contributes fields, summaries, dialogs and stable editing behavior; it does not invent a universal conflict resolution rule for all records.

The draft remains associated with the record and revision. Changing theme or pane arrangement does not clear it. A filter change cancels obsolete list requests but does not cancel an already committed evidence operation.

29.4 Media and account changes constrain caching

Evidence photos are private resources. A thumbnail can serve the overview while a full image is acquired for inspection. Decoded images and retained camera resources have separate lifetimes.

Switching accounts must not leave the next user with an authorized-looking preview of the previous user's evidence. Cache identity, credentials and display permission are related but separate checks. A stable content hash does not remove the need to verify who may see the resource.

This is also why agent context must be scoped. An exception-summary agent may need selected order data, but not every captured image or every account's history.

29.5 Deployment profiles and acceptance

A camera-heavy field client can use direct CMP. A web-first administrative client can later use RSD. The shared component definitions remain useful even when the implementation uses different native capture facilities.

A generated inspection checklist can be an approved graph-owned region. It cannot demand unrelated secrets or add arbitrary executable validation. Publishing an order transition still invokes the real application capability.

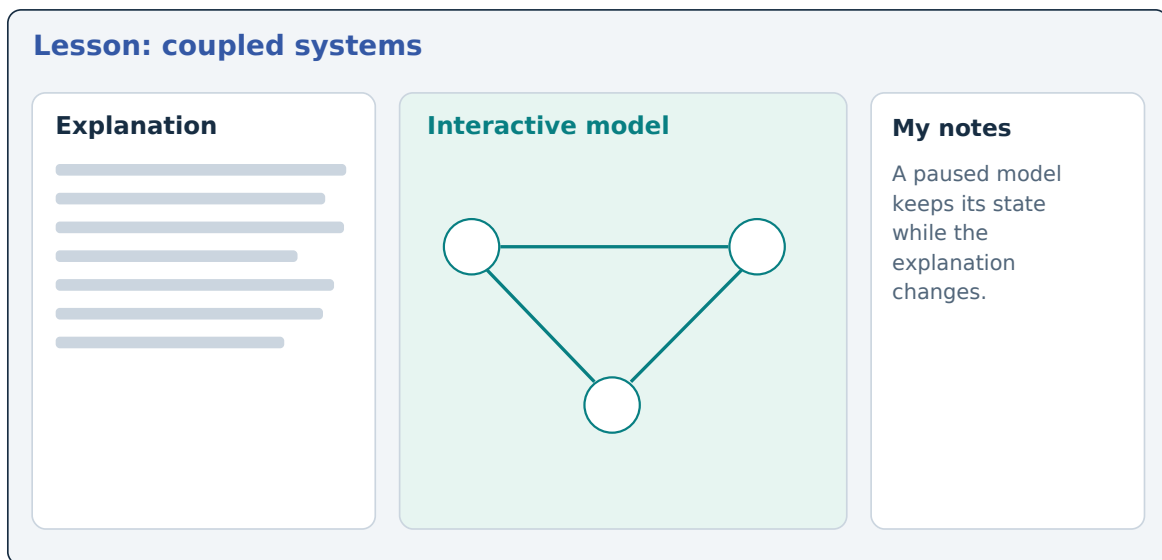
The qualification scenario records evidence offline, changes layout, receives a conflicting revision and switches account. The expected outcome preserves work, exposes the intended conflict policy, keeps evidence correctly scoped and never labels pending synchronization as remote completion.

30. Learning software: explanation beside an interactive model

30.1 The arrangement follows the learning task

A learning application can combine an explanation, interactive model, exercise, notes and progress summary. These are product organisms rather than unrelated pages. A reading mode prioritizes text. An exploration mode allocates more room to the model. An expanded layout can show supporting explanation beside manipulation.

Learning: explanation, manipulable model and retained notes



Educational model dynamics and decorative tactile response have different owners.

Figure 52: Text, a custom model and retained notes participate in one Surface composition. Their owners remain distinct so changing an explanation does not reset a paused simulation or erase a note.

The graph owns lesson identity, attempts, progress and permitted actions. The interactive model owns its internal numerical state unless the application explicitly persists or synchronizes it. The notes owner retains user edits across layout and theme changes.

30.2 Distinguish educational physics from UI physics

A spring diagram can use physical response to teach. Dragging a parameter can show a constraint. A snapping relationship can indicate a valid connection. An impulse can illustrate causal propagation.

However, a scientific model has its own equations and fidelity requirements. It must not silently reuse the approximate visual wave kernel merely because both produce moving shapes. Decorative Tactile behavior belongs to the interface. A lesson's numerical results belong to the educational model.

The same distinction prevents a theme change from changing the lesson's answer. A material language

can alter tool affordances and surrounding controls without modifying the model parameters or the recorded result of an exercise.

30.3 Accessibility can expose the model's structure

A custom visualization should offer semantic operations. A learner can select a variable, move a parameter by a specified amount, pause the model or inspect a relationship. These commands support keyboard and assistive interaction as well as agents.

A table, description or stepwise mode can provide another route to the relevant information. Reduced motion can preserve causal explanation one step at a time rather than forcing continuous animation. The goal is not to speak every pixel; it is to expose the model's meaningful state and operations.

The graph can connect these capabilities to the rest of the lesson. It does not need to receive every rendered particle position if those positions are internal numerical or drawing detail.

30.4 Generated explanation is content with an owner

An agent can propose an explanation or compose approved exercise organisms. The proposal enters an approved graph-owned region. The application retains provenance and the user's attempt state separately from generated text.

The agent cannot mark mastery merely by rendering a success panel. Submitting an attempt invokes the learning application's operation, which determines the result according to its policy. A generated explanation can be replaced without rewriting an already recorded attempt.

A note being edited also remains independent. The supporting explanation may change while the note field retains its identity and revision. An active input-method session is not discarded to make room for newly generated content.

30.5 Resource priorities reflect the task

The main lesson text should become usable before an optional procedural background or heavy visualization finishes loading. Audio and video need captions or alternatives and explicit playback lifetimes. A hidden simulation stops unnecessary drawing while retaining the state needed to resume it.

The qualification session pauses the model, edits notes, enlarges text, uses keyboard controls, changes layout and regenerates explanation. Correct behavior preserves the notes and model state, keeps alternatives operable and prevents generated presentation from changing authoritative learning records.

This example demonstrates why a custom engine and a design system are complementary. The engine retains specialized computation while Surface provides coherent composition, ownership and interaction around it.

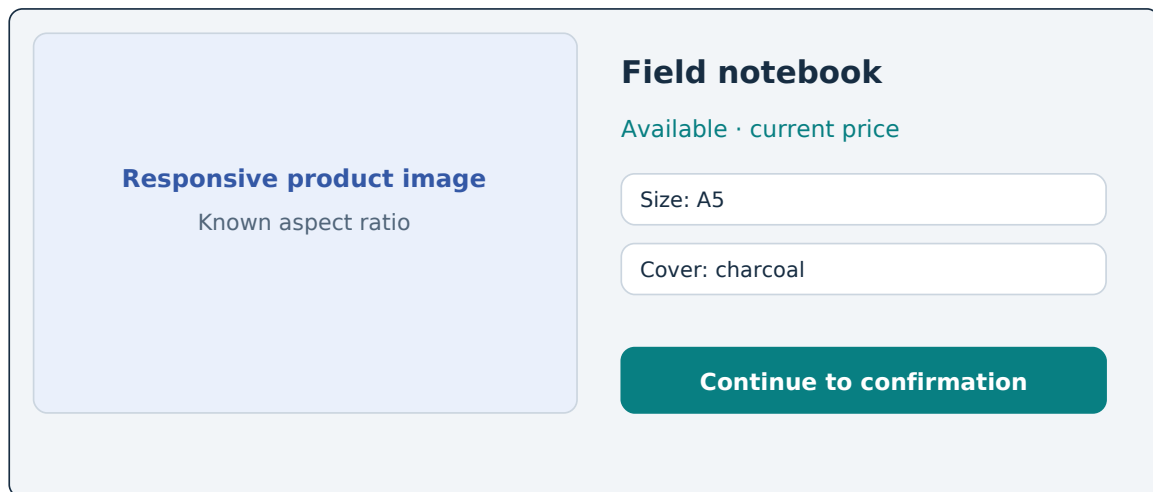
31. Commerce and content: stable presentation around authoritative transactions

31.1 Use the web's strengths when the task is document-oriented

A catalogue or content application often depends on readable documents, navigable resources, forms and predictable loading. The RSD web realization can preserve those qualities while sharing component contracts and design decisions with a CMP mobile application.

Surface does not require a canvas to qualify as cross-platform. The web path uses its appropriate semantic elements and platform behavior. Native presentation can use installed media or payment integrations behind qualified boundaries.

Commerce: stable media, selection and trusted purchase controls



Responsive product image
Known aspect ratio

Field notebook
[Available · current price](#)

Size: A5

Cover: charcoal

Continue to confirmation

Suggested comparisons may change. Payment and confirmation authority remain host-owned.

Figure 53: A product page reserves media geometry, presents stable selection controls and retains trusted confirmation behavior. A generated comparison can supplement the page without acquiring purchase authority.

The product page composes responsive media, product information, option selection, availability and actions. A larger layout can add a supporting summary. A compact layout prioritizes the content and primary operation without changing the meaning of the selections.

31.2 Loading policy is part of the design

The leading image should be discoverable and sized appropriately. Off-screen product media can load later. Fonts have deliberate fallbacks. The selected option and primary action remain in a stable region while supporting content arrives.

These choices affect more than performance scores. A price or purchase action that moves while the person is about to activate it can cause a real interaction error. A layout that reserves media space

protects both reading continuity and target stability.

The image system chooses transfer and decode variants based on actual display needs. A small card does not require an original-size photograph. A high-detail view can acquire a richer variant only when the person opens it.

31.3 Feedback must not overstate transaction success

A product card may lift slightly under the pointer. An Add action can acknowledge contact immediately. The cart, price and stock state still come from authoritative application operations.

A smooth transition into a summary does not prove a purchase succeeded. If stock changed or authorization failed, the UI preserves the selected values and exposes recovery. The confirmation region remains tied to the real transaction outcome.

The design language can change from editorial-flat to tactile or glass-like. It cannot change the action's approval requirements. A translucent treatment also needs an opaque fallback and readable foreground content; essential operations cannot depend on optional visual effects.

31.4 Agent assistance remains bounded

An agent can assemble a comparison from approved product organisms or suggest filters. It can update a supporting graph-owned region. It cannot replace the trusted authentication, consent or payment controls with its own persuasive imitation.

A generated Buy control still resolves to an allowed application capability. Naming the operation does not grant permission. The application checks the current user, selected product, quantity and applicable confirmation policy.

If generation fails, the product page and direct controls remain usable. If a person has already selected options, a new suggestion does not silently reset them. The same editing and revision rules that protect a conversation draft protect a purchase configuration.

31.5 Qualification combines metrics and outcomes

Measure the actual web-vital definitions for the relevant population, then run controlled delays for images and fonts. Change stock while a selection is active. Exercise keyboard checkout, a rejected transaction and a streamed comparison.

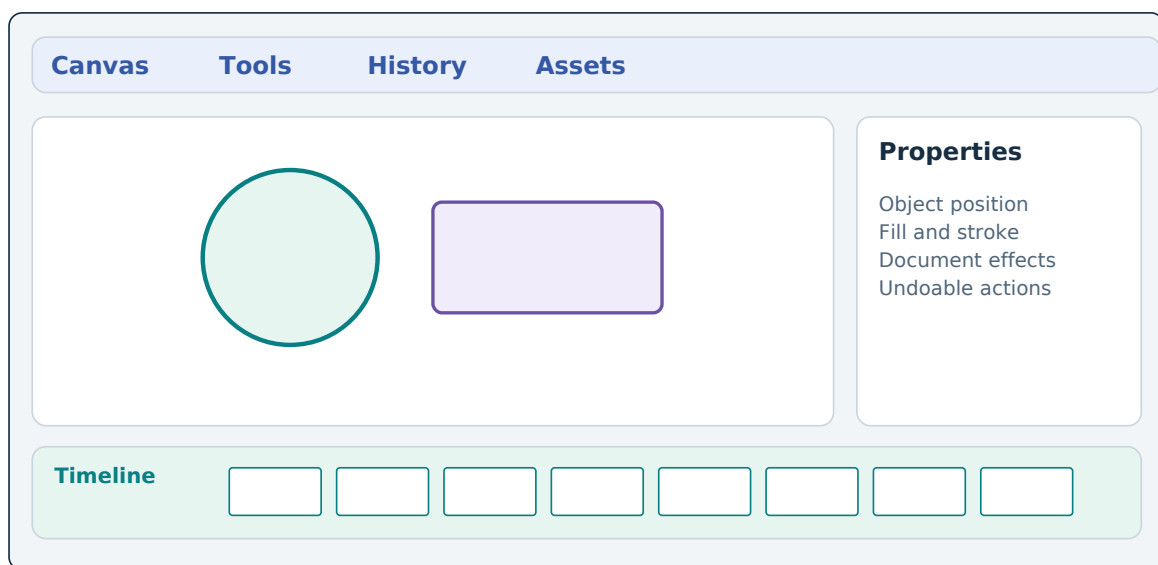
The expected outcome is stable operation and truthful state. Optional tactile work must not compete with critical content loading or input. A transaction failure must not be hidden by a successful-looking animation. The same measurements and semantic checks apply when every decorative effect is disabled.

32. Creative software: a specialized scene within a coherent shell

32.1 Keep the engine expert at its own work

A media or illustration editor combines a canvas, timeline, properties, asset library and command history. The specialized engine owns scene evaluation and domain-specific manipulation. Surface owns the surrounding controls, theme scopes, focus coordination and graph-bound application capabilities.

Creative tools: document truth is separate from tactile chrome



UI fields may animate the tools. Only explicit document commands change authored artwork.

Figure 54: The document engine retains its specialized scene and history. Surface supplies the surrounding controls and lifecycle coordination. Tactile effects on the tools are separate from effects authored into the document.

The engine's scene graph is not the application graph. It can contain shapes, tracks or render passes whose meaning is internal to the document. The Reaktor graph owns the document session, access policy and commands exposed to the rest of the application.

32.2 Preview and commit have different semantics

A pen stroke or object drag needs immediate local feedback. The engine can update a preview as input arrives. The committed document operation has history, persistence and collaboration semantics.

Undo should operate on meaningful document commands, not every decorative frame. A hover response should not create history. A theme field must not distort artwork unless the person explicitly applies a document effect.

This separation is also necessary for agents. An agent can propose a named edit or a preview branch. It does not mutate engine internals outside the command model. Applying the proposal remains visible

and reversible where the document model supports it.

32.3 Multiple coordinate systems remain aligned

Canvas pan, zoom and rotation transform document coordinates. UI zoom transforms the surrounding controls. Pixel density affects rendering resolution. The hit-testing and overlay paths must use the corresponding transforms rather than one shared unqualified scale.

A selection outline can be drawn in a separate overlay, but it still identifies the same document object. A visual ghost during a drag is not a second selectable object. Accessibility and agent commands refer to semantic objects rather than their momentary pixel positions.

On a pen-enabled tablet, palettes may be arranged differently from a pointer-driven desktop. A phone profile can offer review and a constrained editing subset. A browser profile advertises the graphics and media capabilities it actually supports.

32.4 Resource pressure is a normal operating condition

Large assets need thumbnails, tiles or proxies. Decoded pixels, playback buffers and GPU residency have separate budgets. A hidden editor can release graphics resources while retaining a document checkpoint and meaningful session state.

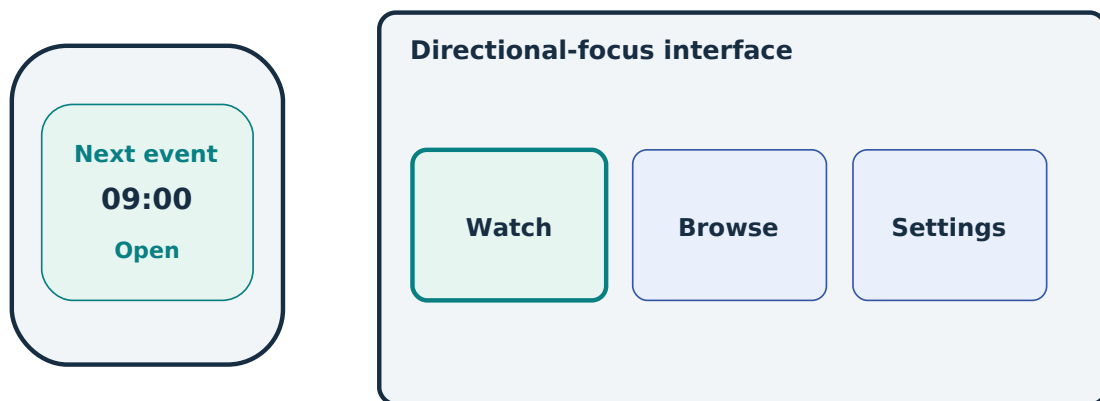
Graphics loss should return control to a trusted shell capable of recovery. A failed custom renderer must not remove the application's ability to navigate away or preserve committed work. Loading the next asset must not retain every previous full-resolution decode indefinitely.

The qualification session edits while loading media, changes theme, rotates and zooms, backgrounds and resumes, and introduces graphics loss plus a concurrent change. Correct behavior preserves document coordinates and history, rejects stale operations and does not discard work during resource cleanup.

33. Constrained hosts: share capabilities rather than an unsuitable page

A watch, television, widget and nonvisual interface impose different attention and input constraints. A television emphasizes directional focus. A watch favors short interactions. A widget exposes a limited template under a platform lifecycle. A voice or text interface presents semantic operations rather than reading a visual tree indiscriminately.

A constrained host exposes a useful subset, not a shrunken desktop



Input, attention and lifecycle change the template. Graph actions retain their meaning.

Figure 55: A compact watch-style surface and a directional-focus interface reuse meaning while choosing different interaction structures. Their supported subsets are explicit rather than being scaled-down desktop layouts.

These hosts can use a qualified local graph subset and the relevant capability bindings. Reaktor Graph remains the foundation, but the host does not need the full application's renderer, all data or every privilege. Its component profile states what can be displayed and invoked.

A widget can show a compact event summary and launch an installed action. It need not run a continuous field simulation. A watch can map a contact cue to its supported haptic behavior without supporting procedural shaders. A television can use restrained focus transitions while preserving predictable directional navigation.

A generated suggestion must fit the host's approved template and attention budget. Unsupported content is reduced, handed off to a richer surface or rejected explicitly. An important operation remains available through direct controls rather than requiring a conversation with an agent.

The qualification scenario removes optional effects, loses connectivity, introduces oversized generated content and reopens through a deep link. Essential state and actions remain understandable. The richer destination resumes the correct application context. A stale region must not redirect the person into an unrelated session merely because the host reused a presentation slot.

These profiles are extension contracts, not automatic consequences of compiling the graph for another execution target. A worker or server can supply data and validation, but it is not an interactive display backend simply because it can run application logic.

A common acceptance model across application families

The social and operational examples expose a shared truthfulness requirement. A local contact response can be immediate, but remote acceptance remains an application result. The same separation protects a message send, an order transition and a purchase. Reusing that rule is more valuable than forcing their buttons to have identical dimensions.

The learning and creative examples expose a shared integrity requirement. Decorative UI motion must not alter the domain model. A field can animate a tool or illustrate an affordance, while a simulation result or document edit remains owned by its specialized command model. The boundary makes expressive interfaces compatible with precise work.

The workbench and constrained-host examples expose a shared portability requirement. A wider range of capabilities does not mean the same page should be shown everywhere. A small host presents a useful subset, while a desktop can expose several regions at once. The graph preserves the meaning and authorization of the operations available in either presentation.

Together these requirements define the architecture's reusable core. It preserves application truth, state ownership and controlled execution while allowing composition and realization to vary. The implementation plan tests those properties in a complete CMP application before expanding the renderer matrix.

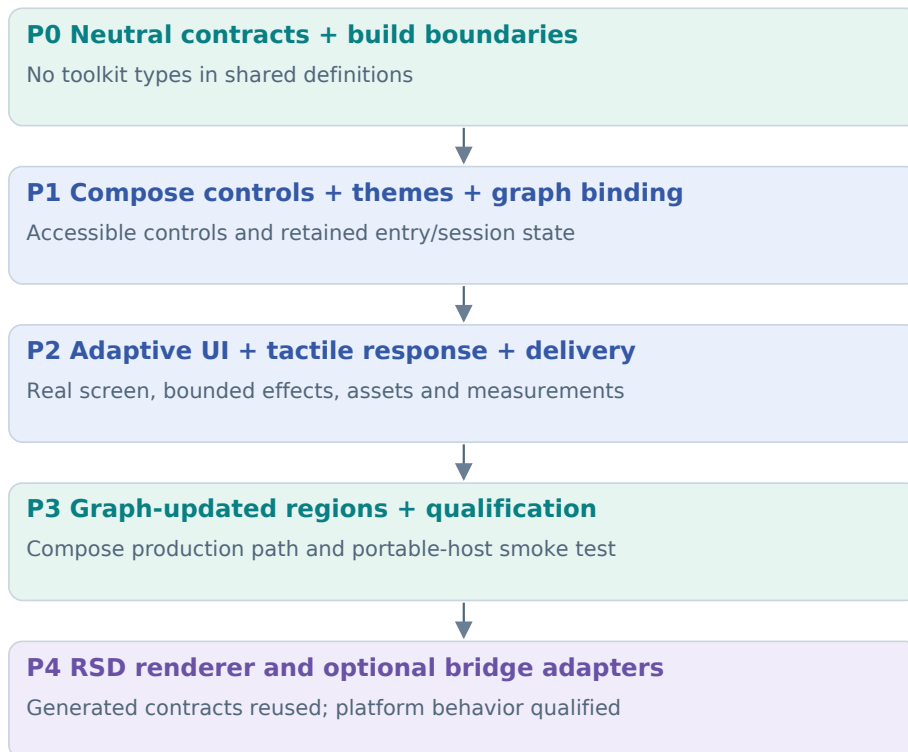
34. A Compose-first implementation plan

34.1 Sequence by risk, not by the number of component names

The first objective is a usable CMP vertical slice: a graph-bound form and conversation region with correct state ownership, theme switching, adaptive presentation and one convincing tactile interaction. A large catalogue built before these boundaries are proven would multiply the cost of correcting them.

React is a later implementation target, but portability is an early design constraint. Neutral contracts, independent behavior and generated TypeScript shapes are established before the shared API freezes. A small portable-host probe challenges structural assumptions without making a complete host runtime a prerequisite for ordinary CMP screens.

Compose-first delivery: freeze shared contracts, add renderers later



A small host/schema probe occurs before API freeze; full React production support follows Compose.

Figure 56: The critical path delivers CMP functionality first. Shared contracts and a small portability probe prevent the first renderer from defining the universal API accidentally. Full RSD qualification follows after the foundation is exercised by real applications.

Tasks below identify prerequisites and observable completion criteria. They are dependency units rather than calendar estimates. Platform qualification can run in parallel where the shared contracts are stable, while changes to core identity or behavior require coordination.

34.2 Foundation: neutral meaning and testable behavior

Task	Prerequisites	Deliverable and completion criterion
S01 — Establish package boundaries	None	Separate neutral contracts, graph integration and toolkit implementations. Add dependency checks that reject Compose or React types in the neutral layer and prevent a graph–UI module cycle.
S02 — Define identities and boundary values	S01	Define component, entry, mount and generation identities; text, asset and logical-geometry values; bounded codecs and validation findings. Round-trip representative values and reject incompatible fingerprints.
S03 — Implement independent behavior kernels	S02	Implement button, toggle and selection behavior without UI imports. Tests cover concurrent focus/hover/press, cancellation, controlled-input changes and duplicate activation delivery.
S04 — Generate the catalogue projections	S02	Generate Kotlin-facing helpers, TypeScript declarations, boundary schemas and test metadata from one definition. A changed contract updates every generated projection coherently.
S05 — Probe the portable host seam	S03, S04	Build a tiny custom-Appier tree with a text node, button and keyed list. Verify insert, move, remove and commit publication. No production screen is forced through this path.

The neutral layer is intentionally small. It should not absorb a layout engine, a new application service registry or every property in Material. Its purpose is to make future implementations agree on values and behavior.

S03 is complete when a behavior trace can be replayed independently of a window. S05 is complete when the portable node vocabulary can express the initial controls without using toolkit-specific values. Neither task establishes accessibility or performance on a device; those belong to the realization tasks.

The early TypeScript projection can be checked without shipping an RSD renderer. It verifies that the contract actually has a reasonable cross-language shape. Unsupported local-only slots remain explicitly marked rather than translated into misleading generic types.

34.3 First CMP slice: controls, themes and graph ownership

Task	Prerequisites	Deliverable and completion criterion
C10 — Theme resolution and Material adapter	S02, S04	Implement token resolution, complete supported Material roles, typography, shape and nested scopes. Two modes and two distinct recipes render coherently without losing a controlled draft.
C11 — Button and action realization	S03, C10	Adapt native interaction and semantics to one activation path. Implement normal, busy, unavailable, focus and pointer states with the same headless behavior contract.

Task	Prerequisites	Deliverable and completion criterion
C12 — Editing and field contracts	S03, C10	Integrate platform editing with revisioned draft updates. Test selection, IME composition, paste, validation failure and stale acknowledgments.
C13 — Selection, dialog and overlay primitives	C11, C12	Implement controls needed by a real form, including focus entry/return, dismissal and theme propagation to overlays. Keyboard and accessibility tests cover complete interactions.
C14 — Graph-bound Surface sessions	S02, C11	Bind models and actions through graph-owned ports. Introduce entry-aware context and explicit mount lifetime. Two mounts cannot overwrite one shared active-entry slot accidentally.

C10 should correct the role-mapping problem before new skins multiply it. A theme preview must exercise the controls that actually consume those roles, including fields, overlays and disabled content. Merely comparing the palette objects is insufficient.

C12 is a particularly important integration test. The editor must remain responsive while the graph draft owner receives updates, and older results must not overwrite newer input. A theme or layout change must not be used as a reason to reconstruct the draft.

C14 should use a keyed session example rather than only one static screen. A conversation opened twice is a useful test because it reveals the difference between business identity and presentation identity. Its actions remain typed and subject to application policy.

Desktop can provide a fast iteration host, but touch, keyboard and native mobile lifecycle behavior need representative Android and iOS qualification early. A successful desktop preview is not evidence for all CMP targets.

34.4 Adaptive presentation and the tactile differentiator

Task	Prerequisites	Deliverable and completion criterion
C20 — Measured adaptive scopes	C14	Replace device-name branching with window environment and local constraints. Implement list-detail and supporting-pane plans. Resize repeatedly while preserving entry identity and draft state.
C21 — Core motion and transitions	C11, C13, C20	Implement interruptible press, enter/exit and navigation motion with a deterministic test clock. Verify reversal, resize during transition, reduced motion and correct input admission.
C22 — Shared tactile field	S03, C21	Implement bounded impulses and explicit coupling. One press affects its face, a second component and a background input locally. Tests reject duplicate/foreign-scope impulses and stop settled work.
C23 — Shader and sensory adapters	C22	Implement one supported procedural treatment plus semantic haptics. Cache shader programs, update uniforms locally, qualify fallback and emit each sensory cue once.
C24 — Product organisms	C12, C13, C20	Compose a message composer, event card and participant picker from the tested catalogue. They accept models/events and do not acquire repositories internally.

The tactile field should be proven with one carefully controlled scene rather than a large catalogue of unrelated effects. The scene needs a visible causal relationship: contact, local compression, neighboring response and decay. It also needs a quiet profile that preserves the same usable controls.

C23 does not require every shader backend at once. Qualify one CMP realization and an explicit static or simpler fallback. Additional backends follow the same uniform, resource and capability contract. A failed effect must leave readable content and operable controls.

C20 and C21 are connected. A panel can be resized during a transition, and its focus owner must remain correct. The qualification suite should deliberately combine these changes rather than test each feature only while the rest of the application is idle.

34.5 Delivery, observation and a release-quality CMP application

Task	Prerequisites	Deliverable and completion criterion
C30 — Asset manifest and request policy	S02, C10	Implement size-aware media requests, variant selection, reserved geometry, privacy scope and cancellation. Verify transfer/decode dimensions and cleanup after account changes.
C31 — Virtualized collection and anchor policy	C20, C30	Build a paged timeline with stable keys and item-relative anchors. Late media and inserted rows do not unexpectedly move the content being read.
C32 — Performance observation	C14, C21, C30	Add bounded renderer, action and resource diagnostics correlated with graph ownership. Define named scenarios and actual observation limits.
C33 — Custom engine boundary	C13, C14, C32	Integrate one graph canvas or editor with lifecycle, focus, resource and semantic-command contracts. Closing or hiding it releases the declared expensive resources.
C34 — CMP qualification matrix	C23, C24, C31, C32, C33	Run representative native/desktop and qualified web-canvas scenarios. Exercise large text, direction, interruption, low memory and repeated view replacement in release-like builds.

C30 is complete only when resource policy is visible in requests and measurements. An image-format conversion alone does not establish size-aware decoding or bounded residency. Private media tests include logout and permission changes, not just cache hits.

C32 should make a regression explainable. A finding can identify a widened subscription, an unnecessarily large decode or an effect that continued while hidden. The system should not export every frame as a graph transaction or log private field values by default.

C34 establishes the first production-quality Surface profile. It is a set of supported combinations, not a blanket claim about all future platforms. The application should be useful without agents and with optional effects disabled.

34.6 Integrate graph updates without building another updater

Task	Prerequisites	Deliverable and completion criterion
G40 — Activated-region observation	C14, C34	Connect Surface to the existing graph activation result. A changed region receives coherent definitions/bindings, preserves declared state and rejects retired-generation presentation work.
G41 — Approved generated region	S04, C24, G40	Permit an agent to propose an event-planning region using the catalogue and granted graph actions. Invalid proposals leave a stable fallback and existing edits intact.
G42 — Interactive-update scenarios	G40, G41	Update supporting content during typing, resize and permission changes. Active fields retain identity or follow an explicit handoff/defer policy. No new updater is implemented inside Surface.

These tasks assume the lower-level subgraph-update capability. They test the UI boundary's response, not release distribution, Hermes loader internals or state-migration machinery owned elsewhere.

G41 begins with known organisms and approved slots. It does not require arbitrary generated code to execute in the user's application. A more powerful execution profile can be integrated later through the graph system without changing which layer owns application authority.

The essential acceptance condition is continuity: activated structure, action bindings and event admission agree. A newly displayed action must not use an incompatible old handler table, and a retired control must not invoke a newly reused callback slot.

34.7 Add RSD after the shared design has been exercised

Task	Prerequisites	Deliverable and completion criterion
R50 — Generated contract consumption	S04, C34	Consume the generated TypeScript catalogue and behavior exports. Verify identical property validation and semantic traces before implementing the full visual catalogue.
R51 — RSD web realization	R50	Implement themes, controls, overlays, assets and graph subscriptions through RSD. Preserve semantic web behavior and qualify real loading and interaction measurements.
R52 — Cross-renderer conformance	R51, G42	Replay component and workflow traces on CMP and RSD. Differences are explicit platform choices, not silent changes in activation, state ownership or generated-region behavior.
R53 — Qualified React Native profiles	R52	Add the supported RSD native hosts with their own input, accessibility, lifecycle and performance tests. Web correctness is not treated as native proof.
R54 — Optional React-to-CMP host adapter	S05, R50, G40	Implement a version-qualified custom reconciler adapter. Validate commit, cancellation, identity, editing and callback lifetime across the host boundary.
R55 — Optional portable Compose-to-RSD path	S05, R51	Realize portable Surface nodes through RSD while preserving one structural owner. Toolkit-specific Compose content remains an explicit local island or unsupported capability.

R51 is the primary React delivery path. R54 and R55 are optional interoperability capabilities, not prerequisites for every web interface. Keeping this order prevents an ambitious bridge project from delaying a straightforward RSD realization of the established catalogue.

A later renderer should mostly add adapters and conformance evidence. If it requires changing the meaning of every component, the earlier neutral contracts were not actually neutral. The early S04 and S05 probes reduce that risk, while the CMP application provides practical feedback about which contracts are worth stabilizing.

34.8 The minimum complete vertical slice

The first integrated demonstration should contain one graph-owned conversation or form, two visual languages, compact and expanded layouts, a revisioned field, local press feedback, one bounded cross-component impulse, size-aware media, a meaningful scroll anchor and a supported graph-updated region.

Run it while changing theme, resizing, typing, loading media and failing an application operation. Confirm that the draft survives, focus remains useful, the operation's result is truthful, hidden work stops and the inspector can explain the resulting cost.

This is the point at which the architecture's main promises become observable. A large list of component names is not an equivalent milestone. **Compose delivers the first complete experience; shared contracts keep the route to React open without making the first experience wait for every future adapter.**

References

- 1 [WAI-ARIA — roles and expected behavior.](#)
- 2 [Android — canonical adaptive layouts.](#)
- 3 [CSS Containment Level 3 — container queries.](#)
- 4 [React. Render and Commit.](#)
- 5 [Compose — rendering phases.](#)
- 6 [Reaktor. Graph ownership and runtime. Source code.](#)
- 7 [React Strict DOM — official project.](#)
- 8 [JetBrains. Compose Multiplatform and Jetpack Compose.](#)
- 9 [Android Developers. Handling user interactions in Compose.](#)
- 10 [W3C WCAG 2.2.](#)
- 11 [Atomic Design — methodology.](#)
- 12 [DTCG Design Tokens Format 2025.10.](#)
- 13 [Android Developers. Custom design systems in Compose.](#)
- 14 [Reaktor. React theme scope. Source code.](#)
- 15 [StyleX — official introduction.](#)
- 16 [BestBuds. BestBuds graph composition. Source code.](#)
- 17 [Reaktor. Back-stack entry identity. Source code.](#)
- 18 [Reaktor. PortGraph storage. Source code.](#)
- 19 [Reaktor. Route binding ownership. Source code.](#)
- 20 [BestBuds. Conversation session state and actions. Source code.](#)
- 21 [Android Developers. Where to hoist state.](#)
- 22 [React. Preserving and Resetting State.](#)
- 23 [Android Developers. Side effects in Compose.](#)
- 24 [Android Developers. The activity lifecycle.](#)
- 25 [Chrome for Developers. Page Lifecycle API.](#)
- 26 [Android — window size classes.](#)
- 27 [Android Developers. Custom layouts in Compose.](#)
- 28 [AndroidX. Compose Runtime Applier interface. Source code.](#)
- 29 [React — useSyncExternalStore.](#)
- 30 [React. react-reconciler: custom renderer interfaces and lifecycle. Source documentation.](#)
- 31 [Android Developers. Customize animations in Compose.](#)
- 32 [Android Developers. Shared-element transitions in Compose.](#)

- 33 [BestBuds](#). SignalTile material-response component. Source code.
- 34 [The Book of Shaders](#) — fractal Brownian motion.
- 35 [Android](#) — AGSL.
- 36 [Skia](#) — SkSL and runtime effects.
- 37 [Reaktor](#). Semantic haptics API. Source code.
- 38 [Compose](#) — strong skipping.
- 39 [React](#) — startTransition.
- 40 [Android](#) — memory management.
- 41 [web.dev](#) — image performance.
- 42 [Coil](#) — Compose image loading.
- 43 [web.dev](#) — optimizing LCP.
- 44 [web.dev](#) — optimizing web fonts.
- 45 [web.dev](#) — Web Vitals.
- 46 [web.dev](#) — optimizing CLS.
- 47 [Android](#) vitals.
- 48 [Apple](#) MetricKit.
- 49 [React](#) — Profiler.
- 50 [Android Developers](#). Macrobenchmark overview.
- 51 [Reaktor](#). Graph dependencies. Source code.
- 52 [Reaktor](#). UI dependencies. Source code.