deepseek

A Programming Paradigm for Spatiotemporal Composability

Yifan Shi[1,2], Wei Zhang[1], Tianyi Cui[2]

[1]**Peking University** [2]**DeepSeek-AI**

Abstract

Modern software—from plugin systems to self-evolving agent harnesses—increasingly requires *dynamic composition*, yet its formal foundations remain underdeveloped. We identify two orthogonal dimensions of the problem: *temporal composability*, the ability to completely reverse a component's side effects upon removal, and *spatial composability*, the ability to declare and reactively manage inter-component dependencies. We address the two dimensions by lifting classical effect and coeffect concepts to runtime mechanisms, respectively. We formalize *revertible effects*, in which each context transformation is equipped with an explicit inverse, and prove that effect tracking preserves the compositional operation, guaranteeing complete state recovery upon component removal. We formalize *reactive coeffects*, in which dependencies are registered in a typed context and satisfaction changes trigger automatic component activation and deactivation. A component lifecycle model operationalizes how revertible effects and reactive coeffects interact with diverse control flows, yielding a unified context type that integrates effect and coeffect contexts into a coherent programming paradigm. We implement these ideas in *Cordis*, a meta-framework of spatiotemporal composability that provides a core library with effect tracking and coeffect resolution, as well as a declarative component loader with configuration reconciliation and hot module replacement.

Contents

1. Introduction

Composition—assembling complex systems from simpler parts—is a foundational principle of software engineering [1]. Traditionally, composition is *static*: function calls, module imports, and class inheritance are resolved at compile time and remain fixed throughout execution. However, modern software increasingly demands *dynamic* composition, where components are loaded, unloaded, and reconfigured at runtime. Plugin architectures [2] and self-evolving agent harnesses both require systems that can safely add and remove functionality on the fly, yet current practice defers to coarse-grained mechanisms [3] that reconfigure only by restarting, discarding runtime state. Despite the growing practical importance of dynamic composition, its theoretical foundations remain underdeveloped, compared to the rich formal frameworks available for static composition.

1.1. Dimensions of Composability

To characterize the requirements of dynamic composition, we identify two orthogonal dimensions beyond the well-studied algebraic aspects of composition:

- **Temporal composability** addresses the *time* dimension: upon removal of a component, the modifications it made to the shared environment must be completely and safely reversed. This requires tracking every resource allocation, event registration, and state mutation the component performs, and guaranteeing their orderly reclamation upon removal.
- **Spatial composability** addresses the *space* dimension: components must be able to declare, discover, and resolve their dependencies on one another in a structured and verifiable manner. This requires managing dependency topology and coordinating component lifecycles in response to dependency changes.

In the static setting, temporal composability reduces to lexical scoping (e.g., RAII [4], bracket patterns [5]), and spatial composability reduces to module import resolution. In the dynamic setting, where components arrive and depart at runtime, both dimensions become significantly harder: temporal composability must handle long-lived, stateful effects whose scope is not lexically bounded; spatial composability must handle dependencies that appear, disappear, or change identity during execution.

1.2. Motivating Examples

1.2.1. Plugin Systems

Plugin systems are a canonical instance of dynamic composition. We use Visual Studio Code (VSCode), one of the most widely-used extensible IDEs, as a representative example.

Temporal limitation. VSCode runs all extensions in a shared process called the extension host. Although extensions can be installed dynamically, this host provides no mechanism to unload an individual extension's code at runtime. Once an extension's `activate` function has executed, disabling or uninstalling it requires restarting the entire host, affecting all loaded extensions. Purely declarative extensions such as themes, keybindings, and snippets carry no

code and can be removed freely. Among the top 100 extensions by install count, however, 87 contain executable code[1] and will therefore require such a restart upon removal. Although VSCode provides a `deactivate` hook, it serves only as a graceful shutdown callback when the host process is already terminating, and thus does not enable live removal. Moreover, the hook separates effect disposal from effect creation (in `activate`), violating locality of concern and making complete cleanup difficult to verify.

Spatial limitation. VSCode does provide `extensionDependencies` for declaring dependencies between extensions, but it sees little use: among the top 100 extensions by install count, only 7 declare `extensionDependencies` on non-built-in extensions.[1] This scarcity reflects the shape of the extension API, which exposes fixed, surface-level extension points such as commands, views, and language features. Extensions contribute to the host through these points rather than depending on one another, so inter-extension dependencies rarely arise. Moreover, VSCode's mechanism for inter-extension interaction provides no structural contract: it exposes an extension's functionality to others through `vscode.extensions.getExtension(...).exports`, but the returned value is untyped (`any` by default), so the dependent cannot rely on a checked interface. In short, VSCode steers extensions toward a fixed set of host-provided extension points, and offers no safe, structured way for them to depend on one another.

These two limitations are not unique to VSCode; they recur across plugin systems generally [2, 6], differing only in degree.

1.2.2. Self-Evolving Agent Harnesses

Modern AI agents rely on runtime agent harnesses [7–9]. These systems may compose diverse tool suites [10] and execution environments, govern permissions and sandboxing, maintain session state and persistence, provide context management and memory systems [11], orchestrate subagents and multi-agent workflows [12], and expose interfaces to users and automation. A future harness may generate and deploy modifications to its own components while continuously serving requests. Model-synthesized reusable tools provide a narrower precursor to component-level self-modification [13]. Each such modification is itself an instance of dynamic composition.

Because these modifications occur continuously and with limited or no human oversight, dynamic composability becomes indispensable. Without temporal composability, each self-modification forces a full restart that discards all process-local accumulated state; at such frequency the cumulative unavailability becomes substantial and in-flight tasks are disrupted repeatedly; worse, a faulty self-modification can disable the very process needed to recover. Without spatial composability, each module must itself detect and adapt to changes in the modules it depends on as they appear, disappear, or change identity, and can do so only by ad hoc means; worse, a naive code-replacement strategy may silently break dependents or introduce circular dependencies that surface only at reload time.

1.2.3. The Coarse-Grained Workaround

One reason dynamic composability has received limited formal attention is that operating systems and container orchestrators already provide a coarse-grained substitute. Process termination and restart yield temporal composability at the granularity of entire address spaces;

[1] Data retrieved from the Visual Studio Code Marketplace on June 9, 2026.

container orchestration [3] yields spatial composability at the granularity of entire services. In practice, most software tolerates the lack of fine-grained composability by deferring to these coarse-grained mechanisms: a misbehaving module is handled by restarting the process, and a service dependency is managed by the orchestrator.

However, this workaround imposes substantial costs. Temporally, each restart discards all process-local accumulated state (e.g., caches, connections, partial computations), and rebuilding it takes seconds to minutes [14]; maintaining availability in the interim requires redundant replicas, incurring resource overhead to compensate for the inability to recover a single component. Spatially, container-level orchestration cannot express dependencies between components sharing an address space, and introduces network overhead for interactions that could be local function calls. Both mechanisms operate at the boundary of processes and containers, yet modern systems increasingly compose at a finer level. This granularity mismatch demands a compositional abstraction that manages effects and dependencies at the same level as the components themselves.

1.3. Contributions

The two dimensions of dynamic composability concern, respectively, how computations *modify* and how they *depend on* their environment. These two directions are precisely what *effect systems* [15, 16] and *coeffect systems* [17, 18] formalize: effects provide the formal vocabulary for reasoning about environmental modifications, and coeffects for reasoning about environmental requirements. However, existing formulations restrict reasoning to compile-time analysis over lexically fixed scopes, and do not extend to dynamic scenarios where components arrive and depart at runtime. By lifting effects to a revertible runtime model and coeffects to a reactive dependency resolution mechanism, we obtain a unified formal foundation for dynamic composability, one that is language-agnostic and applicable to any software architecture requiring dynamic composition. We make the following contributions:

1. We formalize **revertible effects** (Section 3.1): a model in which each context transformation is paired with an explicit inverse function, yielding *effect functions* whose tracking and recovery are composition-preserving operations. This provides the algebraic foundation for dynamic temporal composability.
2. We formalize **reactive coeffects** (Section 3.2): a typed dependency context in which components declare their requirements as a dependency set, and a satisfaction-based notification mechanism automatically classifies state transitions as activating, deactivating, or neutral. This provides the algebraic foundation for dynamic spatial composability.
3. We establish a **component lifecycle** model (Section 3.3) that operationalizes how revertible effects and reactive coeffects interact with diverse control flows, and integrates the effect and coeffect contexts into a unified **context type** (Section 3.4), constituting a programming paradigm for dynamically composable systems.
4. We implement these ideas in **Cordis** (Section 4), a meta-framework of spatiotemporal composability that provides a core library realizing the formal model with effect tracking and coeffect resolution, as well as a declarative component loader with configuration reconciliation and hot module replacement. We validate the design through a case study on the Koishi chatbot platform, an ecosystem of over 4000 community plugins in production use.

2. Preliminaries

This section provides a concise overview of effect and coeffect systems—the two theoretical pillars underlying our work. We assume familiarity with basic type theory and category theory; the goal here is to fix notation and introduce the key abstractions that Section 3 will operationalize as runtime mechanisms.

2.1. Effects

In the simply typed lambda calculus (STLC) [19, 20], a typing judgment $\Gamma \vdash t : T$ states that term t has type T under context Γ. An *effect system* refines the type to record what side effects a computation may produce, yielding judgments of the form

$$\Gamma \vdash t : T_{\text{effect}} \tag{1}$$

Here, the result type is annotated with an element of an effect algebra that records which side effects the computation may produce, enabling compositional reasoning about stateful computations. This approach originates with Lucassen and Gifford [21], who introduced a kinded type system distinguishing types, effects, and regions to discover scheduling constraints in parallel programs.

Monadic effects. Moggi [15] first modeled computational effects categorically via monads; Wadler [22] popularized the approach in Haskell. A monad (T, η, μ) on a category $\mathcal{C}$ encapsulates an effectful computation as a value of type $T(A)$, with $\eta : A \to T(A)$ lifting pure values and $\mu : T(T(A)) \to T(A)$ sequencing nested computations. Classic instances include the Maybe monad (partiality), State monad (mutable state), and IO monad (external interaction).

Algebraic effects. Plotkin and Power [16, 23] showed that algebraic operations determine monads, establishing a framework in which effect interfaces are decoupled from their implementations. An effect signature Σ declares a set of operations (e.g., $\text{get} : () \to S$, $\text{put} : S \to ()$ for state); programs invoke operations freely without committing to a particular interpretation. Plotkin and Pretnar [24] subsequently introduced *effect handlers*, which interpret operations by providing continuation semantics:

$$\text{handle } e \text{ with } \{ \text{ op}(v, \kappa) \mapsto \ldots \} \tag{2}$$

The handler receives the operation argument v and the delimited continuation κ, which it may invoke zero, one, or multiple times, enabling exceptions, coroutines, and non-determinism within a uniform framework [25]. Languages such as Koka [26, 27], Eff [28], and OCaml 5 [29] have adopted algebraic effects with varying design trade-offs.

2.2. Coeffects

Dually to effects, a *coeffect system* [17, 30] enriches the context rather than the type, yielding judgments of the form

$$\Gamma_{\text{coeffect}} \vdash t : T \tag{3}$$

Here, the context is annotated with an element of a coeffect algebra describing what the computation requires from its environment, such as resources to access, capabilities to hold, or

services to depend on. While effects model a program's impact on the world, coeffects model the world's constraints on the program.

Comonadic coeffects. The idea of using comonads to structure context-dependent computation was first developed by Uustalu and Vene [31], who proposed symmetric (semi)monoidal comonads as the dual of Moggi's monadic framework for effects, capturing notions such as dataflow and attribute evaluation. Petricek et al. [17] built on this foundation to propose coeffects as a unified static analysis of context-dependence. A comonad (D, ε, δ) captures context-dependent computation: $\varepsilon : D(A) \to A$ extracts the current value from a context, and $\delta : D(A) \to D(D(A))$ duplicates context for nested access. The Environment comonad $D(X) = E \times X$ models dependence on a fixed environment E; the Stream comonad $D(X) = \mathbb{N} \to X$ models dependence on temporal data.

Graded coeffects. For finer-grained tracking, *graded* coeffect systems use a pre-ordered semiring $\mathcal{S} = (S, \leq, +, \times, 0, 1)$ as the coeffect algebra [32], a discipline later unified with graded effects by Gaboardi et al. [18]. Elements of S annotate each variable binding to quantify its usage: 0 for unused, 1 for linear use, n for bounded use, ∞ for unrestricted use. The semiring operations compose coeffects sequentially $(\times)$ and in parallel $(+)$, enabling precise resource tracking, sensitivity analysis [33], and information-flow control [34, 35] within a unified algebraic framework [36].

2.3. Relationship to Dynamic Composability

Effect and coeffect systems organize reasoning about computation along two complementary directions: effects describe how a computation *modifies* its environment, whereas coeffects describe how it *depends on* its environment. These two directions correspond to the two dimensions of dynamic composability identified in Section 1:

- **Temporal composability** demands that a component's modifications to the shared environment be reversible upon unloading. The relevant effects are the stateful ones, which durably transform that environment; reversing such a transformation requires it to admit an inverse.
- **Spatial composability** demands that inter-component dependencies be declared and managed reactively. Such dependencies are the very thing coeffects capture, and managing them amounts to resolving each against what the environment supplies.

However, classical effect and coeffect systems are static instruments: effects are tracked within lexically fixed scopes and discharged by compile-time handlers; coeffect annotations are verified against contexts determined before execution. Dynamic composition, by contrast, requires these guarantees to hold for components that arrive and depart at runtime, against contexts that evolve continuously. No fixed lexical scope can delimit a plugin loaded after deployment; no compile-time context can anticipate dependencies that emerge from runtime configuration.

This motivates a shift in perspective: rather than extending static type systems with more annotations, we reify the conceptual structures of effects and coeffects so that a runtime can operate on them directly, establishing dynamically the guarantees these systems provide statically. Section 3 develops this construction.

3. Revertible Effects and Reactive Coeffects

This section lifts the concepts of effects and coeffects introduced in Section 2 to runtime mechanisms, constructing a dynamic composition theory. The central idea is to turn the *type contexts* carrying effects and coeffects into *context types*—runtime-operable types that reify the context as a first-class entity. For the effect type, we model it as a context transformation equipped with an inverse, achieving dynamic temporal composability. For the coeffect context, we model it as a type carrying dependency information, achieving dynamic spatial composability. The interaction of the two mechanisms with different control flows gives rise to a component lifecycle model, and the unified context that carries both effects and coeffects constitutes a programming paradigm in its own right.

3.1. Revertible Effects

Temporal composability is the ability to load and unload components at runtime such that, upon unloading, the shared environment is recovered to its pre-composition state. This requires that every modification a component makes to the environment be both trackable and reversible. We therefore model an effect as a function of type $\Gamma \to \Gamma \times (\Gamma \to \Gamma)$: applied to the current context, it yields the modified context together with an explicit inverse. The inverse is what makes the effect reversible, and returning it to the runtime is what makes the effect trackable. We call such effects *revertible*: by recording and composing these inverses during execution, complete environment recovery becomes a structural guarantee.

3.1.1. Effect Context

Given any impure function $f_{\text{impure}} : X \to Y$, we transform it into a pure form $f : \Gamma \times X \to \Gamma \times Y$, where Γ is the context and all possible side effects can be represented as transformations on Γ. For any fixed input $x : X$, the induced map $\gamma \mapsto \text{pr}_1(f(\gamma, x))$ captures the side effect of f independently of the return value. As a first approximation, assume every effect's inverse is available a priori; the admissible effects on Γ then form a subgroup $\mathfrak{F}_\Gamma \leq \text{Sym}(\Gamma)$ under composition $\circ$, whose axioms correspond to structural requirements of the system:

- *Closure* ensures that sequential composition of effects remains admissible;
- *Identity* (id_Γ), the identity function on Γ, acts as the unit of composition;
- *Inverse* requires every effect $f \in \mathfrak{F}_\Gamma$ to admit a revert $f^{-1} \in \mathfrak{F}_\Gamma$, ensuring that any composition of effects can be undone.

The inverse requirement is justified by a scope restriction: Γ models only the system's internal state, i.e., state over which it retains full control. Every modification the system makes to its own context is reversible, because the prior value can always be recovered. Operations that target entities outside the system boundary (network requests, file I/O) do not modify Γ; from the perspective of $\mathfrak{F}_\Gamma$, they act as id_Γ. External interactions are handled by domain-specific strategies (e.g., compensating transactions [37]) outside the scope of this model.

To record effects within the context itself, we introduce the following core definition:

Definition 1. Given a context Γ, define its *effect context* as:

$$\partial\Gamma := \Gamma \times \mathfrak{F}_\Gamma \tag{4}$$

It can be understood as a pair (γ, φ), where:

- $\gamma \in \Gamma$ represents the current context state;
- $\varphi \in \mathfrak{F}_\Gamma$ is the transformation that recovers the context to its initial state.

In particular, the initial state can be represented as $(\gamma_0, \mathrm{id}_\Gamma)$.

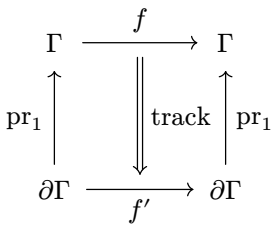
Given the presence of φ above, all effects performed on $\partial\Gamma$ can be automatically tracked and fully recovered. We now give the concrete constructions.

Definition 2. Define the transformation track_Γ from $\mathfrak{F}_\Gamma$ to $\mathfrak{F}_{\partial\Gamma}$:

$$\begin{array}{lcccccc} \mathrm{track}_\Gamma & : & (\Gamma \to \Gamma) & \to & \partial\Gamma & \to & \partial\Gamma \\ \mathrm{track}_\Gamma & = & f & \mapsto & (\gamma, \varphi) & \mapsto & (f(\gamma), \varphi \circ f^{-1}) \end{array}$$

This transformation converts any function $f \in \mathfrak{F}_\Gamma$ on the context Γ into a function on the effect context $\partial\Gamma$. When executing $\mathrm{track}_\Gamma(f)$ in state (γ, φ), the inverse of f is composed with φ, effectively tracking the effect of f in the context. Each $\mathrm{track}_\Gamma(f)$ is itself a bijection on $\partial\Gamma$, with inverse $(\gamma', \varphi') \mapsto (f^{-1}(\gamma'), \varphi' \circ f)$; we accordingly define the admissible effects on $\partial\Gamma$ as the image $\mathfrak{F}_{\partial\Gamma} := \mathrm{track}_\Gamma(\mathfrak{F}_\Gamma)$, which by Theorem 4 is a subgroup of $\mathrm{Sym}(\partial\Gamma)$ isomorphic to $\mathfrak{F}_\Gamma$.

Theorem 3. The following diagram commutes. That is, $\forall f \in \mathfrak{F}_\Gamma$, $\mathrm{pr}_1 \circ \mathrm{track}_\Gamma(f) = f \circ \mathrm{pr}_1$.



Theorem 4. track preserves the $\circ$ operation. That is, $\forall f, g \in \mathfrak{F}_\Gamma$:

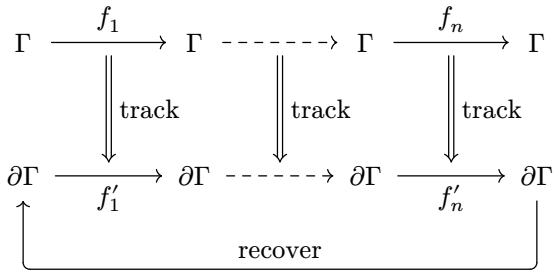
$$\mathrm{track}_\Gamma(f \circ g) = \mathrm{track}_\Gamma(f) \circ \mathrm{track}_\Gamma(g) \tag{5}$$

This theorem ensures that for every function in $\mathfrak{F}_\Gamma$, we can construct an equivalent track version whose effects are tracked on $\partial\Gamma$.

Definition 5. Define the transformation $\mathrm{recover}_\Gamma$ on $\partial\Gamma$:

$$\begin{array}{lcccc} \mathrm{recover}_\Gamma & : & \partial\Gamma & \to & \partial\Gamma \\ \mathrm{recover}_\Gamma & = & (\gamma, \varphi) & \mapsto & (\varphi(\gamma), \mathrm{id}_\Gamma) \end{array}$$

This transformation applies the recovery function φ to the current state γ and resets φ to the identity. This effectively recovers all effects on γ. The following diagram illustrates how recover returns the context to its initial state after a sequence of effects $\mathrm{track}(f_1), \cdots, \mathrm{track}(f_n)$ has been applied to $\partial\Gamma$:



3.1.2. Revertible Effect Functions

The track/recover model of the previous section assumes that inverses are given and recovers all accumulated effects atomically. In practice, however, the inverse of each effect is not known a priori: it must be supplied by the caller at the point of effect application. Moreover, recover is all-or-nothing: it cannot selectively undo one effect while retaining others. To address both issues, we enhance the model at both the input and output sides:

1. On the input side, we not only transform Γ but also return the corresponding inverse function, yielding manual tracking capability: $\Gamma \to \Gamma \times (\Gamma \to \Gamma)$, i.e., $\Gamma \to \partial\Gamma$;
2. On the output side, we not only transform $\partial\Gamma$ but also return the corresponding inverse function, yielding partial recovery capability: $\partial\Gamma \to \partial\Gamma \times (\partial\Gamma \to \partial\Gamma)$, i.e., $\partial\Gamma \to \partial^2\Gamma$.

This enhancement preserves structural consistency between input and output, so we can still define corresponding theory that maintains the mathematical properties of track. We enhance $\mathfrak{F}_\Gamma$ to $\mathfrak{E}_\Gamma$ and $\mathfrak{E}^*_\Gamma$:

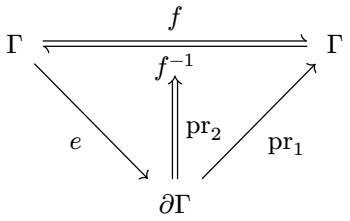
Definition 6. Define the effect function $\mathfrak{E}_\Gamma$ and strict effect function $\mathfrak{E}^*_\Gamma$ as:

$$\begin{aligned} \mathfrak{E}_\Gamma &:= \Gamma \to \Gamma \times (\Gamma \to \Gamma) \\ \mathfrak{E}^*_\Gamma &:= (e : \Gamma \to \Gamma \times (\Gamma \to \Gamma)) \\ &\quad \times ((\gamma : \Gamma) \to (\mathbf{let}\ (\delta, g) = e(\gamma)\ \mathbf{in}\ g(\delta) = \gamma)) \end{aligned} \tag{6}$$

where $e(\gamma)$ yields a pair (δ, g) representing:

- $\delta : \Gamma$ is the new context;
- $g : \Gamma \to \Gamma$ is the inverse function of the current effect.

The constraint $g(\delta) = \gamma$ in the above definition can be visualized as the following commutative diagram, ensuring that the second component of the return value is indeed an inverse of the transformation itself:

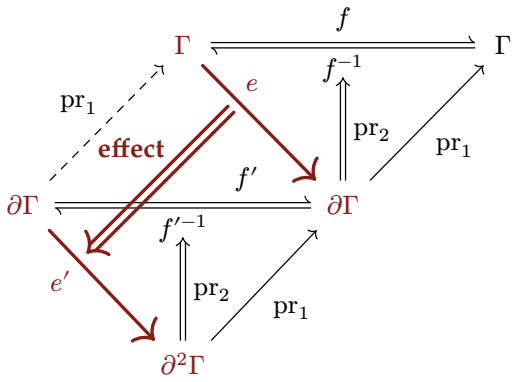


Just as track lifts $\mathfrak{F}_\Gamma$ to $\mathfrak{F}_{\partial\Gamma}$, we define effect to lift $\mathfrak{E}_\Gamma$ to $\mathfrak{E}_{\partial\Gamma}$:

Definition 7. Define the effect function transformation effect_Γ as:

$$\begin{aligned} \mathrm{effect}_\Gamma &: \mathfrak{E}_\Gamma \to \partial\Gamma \to \partial^2\Gamma \\ \mathrm{effect}_\Gamma &= e \mapsto (\gamma, \varphi) \mapsto \begin{array}{l} \mathbf{let}\ (\delta, g) = e(\gamma)\ \mathbf{in} \\ ((\delta, \varphi \circ g), \mathrm{track}_\Gamma(g)) \end{array} \end{aligned}$$

The following diagram illustrates how effect "translates" the operation on $\mathfrak{E}_\Gamma$ to $\mathfrak{E}_{\partial\Gamma}$. We shall prove that this translation indeed preserves diagram commutativity.



Theorem 8. effect is closed on $\mathfrak{E}^*$. That is, $\forall e \in \mathfrak{E}^*_\Gamma, \text{effect}_\Gamma(e) \in \mathfrak{E}^*_{\partial\Gamma}$.

Since effect functions $\mathfrak{E}_\Gamma$ are no longer endomorphisms on the context, they cannot be directly composed. We therefore define a new operation to express effect composition:

Definition 9. Given functions $f, g \in \mathfrak{E}_\Gamma$, define their effect composition $f \diamond g$ as:

$$\begin{array}{rcl} f \diamond g & : & \Gamma \to \partial\Gamma \\ f \diamond g & = & \gamma \mapsto \begin{array}{l} \textbf{let } (\delta, s) = g(\gamma) \textbf{ in} \\ \textbf{let } (\varepsilon, t) = f(\delta) \textbf{ in} \\ (\varepsilon, s \circ t) \end{array} \end{array}$$

Theorem 10. The $\diamond$ operation is closed on $\mathfrak{E}^*$. That is, $\forall f, g \in \mathfrak{E}^*_\Gamma, f \diamond g \in \mathfrak{E}^*_\Gamma$.

We can now prove properties for effect analogous to those of track.

Theorem 11. effect preserves the $\diamond$ operation. That is, $\forall f, g \in \mathfrak{E}_\Gamma$:

$$\text{effect}_\Gamma(f) \diamond \text{effect}_\Gamma(g) = \text{effect}_\Gamma(f \diamond g) \tag{7}$$

Together, these constructions constitute *revertible effects*: each effect function in $\mathfrak{E}^*_\Gamma$ explicitly provides its own inverse, effect tracks these inverses on the effect context $\partial\Gamma$, and the $\diamond$ operation composes them while preserving revertibility. Temporal composability is achieved by this construction: loading a component amounts to applying a sequence of effect functions (accumulating inverses in φ), and unloading it amounts to applying φ to recover the context to its pre-composition state.

Whether unloading one component disturbs another depends on commutativity under $\diamond$: if $f \diamond g = g \diamond f$, recovering f leaves g's contribution intact, and the two effects can safely reside in independent components. When effects do not commute, the system must enforce an ordering constraint, either within a single component (via the LIFO discipline of effect iterators, Section 3.3) or across components (via reactive coeffects that impose a dependency-driven activation order, Section 3.2).

3.2. Reactive Coeffects

Spatial composability is the ability for components to declare dependencies on one another and for the system to resolve, provide, and revoke those dependencies at runtime. This requires that dependency satisfaction be re-evaluated whenever the shared context changes, so that a

component activates precisely when its dependencies become available and deactivates when they are withdrawn. We therefore model dependencies of a component as a specification and classify each change to the context, against that specification, as activating, deactivating, or neutral. Classifying against the specification is what detects a change in satisfaction; responding to that classification is what drives activation and recovery. We call such coeffects *reactive*: by classifying context changes and driving activation and recovery from them, correct dependency ordering becomes a structural guarantee.

3.2.1. Coeffect Context

Traditional inversion-of-control (IoC) containers [38, 39] typically model dependencies as simple key-value mappings. This section formalizes IoC as a coeffect context that synergizes with revertible effects to provide a mathematical foundation for dynamic composition.

Definition 12. Given a type family $\mathcal{V} : K \to \mathrm{Type}$, define the *coeffect context* as the dependent partial function type:

$$\Sigma \coloneqq (k : K) \rightharpoonup \mathcal{V}_k \tag{8}$$

where $\sigma : \Sigma$ is a finite partial function assigning to each $k \in \mathrm{dom}(\sigma) \subseteq K$ a value of type $\mathcal{V}_k$. We write:

- $\sigma(k)$ for application (defined when $k \in \mathrm{dom}(\sigma)$);
- $\sigma[k \mapsto v]$ for extension (defined when $k \notin \mathrm{dom}(\sigma)$);
- $\sigma \setminus k$ for restriction (defined when $k \in \mathrm{dom}(\sigma)$);
- $k \in \mathrm{dom}(\sigma)$ for membership.

The use of a type family $\mathcal{V}$ ensures that each dependency key k is associated with a specific value type $\mathcal{V}_k$, providing static type safety for dependency access. The preconditions on extension and restriction correspond to the idempotence requirement in revertible effects: a dependency cannot be provided twice ($k \notin \mathrm{dom}(\sigma)$ for extension) nor revoked if absent ($k \in \mathrm{dom}(\sigma)$ for restriction). Based on this context structure, we define two core operations:

Definition 13. The get and set operations on Σ are defined as:

$$\begin{array}{lcccccc}
\mathrm{get} & : & (k : K) & \to & \Sigma & \to & \mathcal{V}_k \\
\mathrm{get} & = & k & \mapsto & \sigma & \mapsto & \sigma(k) \\
\mathrm{set} & : & (k : K) \times \mathcal{V}_k & \to & \Sigma & \to & \Sigma \times (\Sigma \to \Sigma) \\
\mathrm{set} & = & (k, v) & \mapsto & \sigma & \mapsto & (\sigma[k \mapsto v],\ \lambda\sigma'.\sigma' \setminus k)
\end{array}$$

where $\mathrm{get}(k)$ requires $k \in \mathrm{dom}(\sigma)$ and $\mathrm{set}(k, v)$ requires $k \notin \mathrm{dom}(\sigma)$ as preconditions.

Notably, $\mathrm{set}(k, v)$ has type $\mathfrak{E}_\Sigma$, precisely an effect function on the coeffect context. We can therefore directly apply the effect machinery from Section 3.1: effect_Σ provides automatic tracking and recovery of dependency registrations. This is the synergy between reactive coeffects and revertible effects: coeffect operations are effects, and effects are revertible.

3.2.2. Specification and Notification

The preceding definitions describe how individual dependencies are registered and accessed. Accessing an absent dependency, however, is a runtime failure. A component should therefore

activate only once all the dependencies it declares are present, rather than accessing them optimistically and failing when one is missing. This raises two questions: whether a component's declared dependencies are jointly satisfied, and how the system should respond when that status changes. The coeffect context Σ carries a natural observational structure that makes both questions tractable: for any coeffect specification $d \subseteq K$, define the *satisfaction predicate*:

$$\sigma \vDash d := \forall k \in d.\ k \in \mathrm{dom}(\sigma) \tag{9}$$

This predicate is decidable (since $\mathrm{dom}(\sigma)$ is finite). Since all mutations to σ pass through effect functions (whose inverses recover the previous domain), changes to satisfaction are detectable at each effect boundary. This is the algebraic basis of reactivity: the effect system guarantees that every coeffect change is observed.

Definition 14. A *coeffect specification* is:

$$\mathfrak{D}_\Sigma := \mathrm{Set}(K) \tag{10}$$

representing the set of dependencies a component declares from the environment.

What makes this specification reactive is how it classifies state transitions. Any effect that transforms σ to σ' can be classified by a specification $d \in \mathfrak{D}_\Sigma$ according to whether d's satisfaction status is altered:

Definition 15. Given a coeffect specification $d \subseteq K$ and states $\sigma, \sigma' \in \Sigma$, define:

$$\mathrm{notify}_d(\sigma, \sigma') := \begin{cases} \text{activating} & \text{if } \sigma \nvDash d \wedge \sigma' \vDash d \\ \text{deactivating} & \text{if } \sigma \vDash d \wedge \sigma' \nvDash d \\ \text{neutral} & \text{otherwise} \end{cases} \tag{11}$$

This is well-defined because $\sigma \vDash d$ is decidable and all state transitions are mediated by effect functions. The reactive invariant is: an activating transition triggers execution of the component's effects (with full effect tracking), while a deactivating transition triggers recovery of the accumulated effects. The precise operational semantics of these transitions depend on their interaction with control flows, and are developed in Section 3.3.

Spatial composability follows from the interaction between set and notify. If component A provides a key k (via $\mathrm{set}(k, v)$) and component B declares $k \in d_B$, then B can activate only after A has fully activated and provided k (since $\sigma \vDash d_B$ requires $k \in \mathrm{dom}(\sigma)$). Conversely, unloading A removes k from $\mathrm{dom}(\sigma)$, which would break B's satisfaction; the system therefore ensures B has fully deactivated before A's recovery begins. Thus the dependency ordering (dependents activate after their dependencies; dependencies unload after their dependents) emerges from the notification mechanism without being separately specified.

3.2.3. Isolation and Interception

The basic coeffect context Σ models a flat dependency table. In practice, however, the system may need to bind distinct values to the same logical dependency for different components. This section extends the coeffect context with two mechanisms: *coeffect isolation* (the same key resolves differently in different contexts) and *coeffect interception* (cross-cutting behavior on dependency access).

Coeffect Isolation. By introducing *isolation realms*, coeffect isolation allows the same dependency to bind to different values in different contexts. This has broad applications in multi-tenant systems, testing environments, and component sandboxes.

Definition 16. Define the coeffect context with isolation as:

$$\Sigma^{\text{iso}} := \text{Map}(K, R) \times ((r : R) \rightharpoonup \mathcal{V}_r) \tag{12}$$

It can be represented as a pair (ρ, σ), where:

- $\rho : \text{Map}(K, R)$ is the isolation realm table, mapping logical keys to realm identifiers;
- $\sigma : (r : R) \rightharpoonup \mathcal{V}_r$ is the dependency table, a partial dependent function from realm identifiers to typed values.

The two-layer mapping structure decouples the logical layer from the storage layer, making dependency access context-aware. When accessing a key k, the system first resolves $\rho(k)$ to obtain a realm identifier r, then accesses $\sigma(r)$ for the actual value.

Definition 17. The get, set, and isolate operations on Σ^{iso} are:

$$\begin{array}{rcccccc}
\text{get} & : & (k : K) & \to & \Sigma^{\text{iso}} & \to & \mathcal{V}_{\rho(k)} \\
\text{get} & = & k & \mapsto & (\rho, \sigma) & \mapsto & \sigma(\rho(k)) \\
\text{set} & : & (k : K) \times \mathcal{V}_{\rho(k)} & \to & \Sigma^{\text{iso}} & \to & \Sigma^{\text{iso}} \times \left(\Sigma^{\text{iso}} \to \Sigma^{\text{iso}}\right) \\
\text{set} & = & (k, v) & \mapsto & (\rho, \sigma) & \mapsto & ((\rho, \sigma[\rho(k) \mapsto v]),\ \lambda(\rho', \sigma').(\rho', \sigma' \setminus \rho'(k))) \\
\text{isolate} & : & K \times R & \to & \Sigma^{\text{iso}} & \to & \Sigma^{\text{iso}} \times \left(\Sigma^{\text{iso}} \to \Sigma^{\text{iso}}\right) \\
\text{isolate} & = & (k, r) & \mapsto & (\rho, \sigma) & \mapsto & ((\rho[k \mapsto r], \sigma),\ \lambda(\rho', \sigma').(\rho' \setminus k, \sigma'))
\end{array}$$

The coeffect isolation mechanism essentially implements a runtime ad-hoc polymorphism system. Through isolation realm identifiers, the same dependency key can resolve to entirely different values in different contexts, and this polymorphism can be dynamically adjusted at runtime. Compared to traditional dependency injection, coeffect isolation provides finer-grained control, enabling customized isolation for specific components; all operations remain effect functions ($\mathfrak{E}_{\Sigma^{\text{iso}}}$) and thus inherit revertibility.

Coeffect Interception. The second mechanism, *coeffect interception*, attaches cross-cutting metadata to dependency access, adding behavior without modifying the dependency value. This metadata can be either context-carried or component-declared, so we extend both the coeffect context and the coeffect specification:

Definition 18. Define the coeffect context and specification with interception as:

$$\begin{aligned}
\Sigma^{\text{inter}} &:= ((k : K) \to \mathcal{M}_k) \times ((k : K) \rightharpoonup (\mathcal{M}_k \to \mathcal{V}_k)) \\
\mathfrak{D}^{\text{inter}} &:= (k : K) \rightharpoonup \mathcal{M}_k
\end{aligned} \tag{13}$$

The context Σ^{inter} is a pair (ι, σ): ι is the *context-carried* metadata installed on the context itself, empty (ϵ_k) by default; and σ maps each key k to a *provider function* from metadata $\mathcal{M}_k$ to value $\mathcal{V}_k$. A specification $d \in \mathfrak{D}^{\text{inter}}$ carries the *component-declared* metadata, assigning each key its metadata $d(k)$, with $\text{dom}(d)$ serving as the dependency set. Each key equips its metadata with a monoid $(\mathcal{M}_k, \oplus_k, \epsilon_k)$: the merge $\oplus_k$ is associative with identity ϵ_k (the empty metadata).

Definition 19. The get, set, and intercept operations on Σ^{inter} are:

$$
\begin{array}{rcccccc}
\text{get} & : & (k:K)\times\mathcal{M}_k & \to & \Sigma^{\text{inter}} & \to & \mathcal{V}_k \\
\text{get} & = & (k,\mu) & \mapsto & (\iota,\sigma) & \mapsto & \sigma(k)(\mu\oplus_k \iota(k)) \\
\text{set} & : & (k:K)\times(\mathcal{M}_k\to\mathcal{V}_k) & \to & \Sigma^{\text{inter}} & \to & \Sigma^{\text{inter}}\times(\Sigma^{\text{inter}}\to\Sigma^{\text{inter}}) \\
\text{set} & = & (k,\psi) & \mapsto & (\iota,\sigma) & \mapsto & ((\iota,\sigma[k\mapsto\psi]),\,\lambda(\iota',\sigma').(\iota',\sigma'\setminus k)) \\
\text{intercept} & : & (k:K)\times\mathcal{M}_k & \to & \Sigma^{\text{inter}} & \to & \Sigma^{\text{inter}}\times(\Sigma^{\text{inter}}\to\Sigma^{\text{inter}}) \\
\text{intercept} & = & (k,\nu) & \mapsto & (\iota,\sigma) & \mapsto & ((\iota[k\mapsto\iota(k)\oplus_k\nu],\sigma),\,\lambda(\iota',\sigma').(\iota'[k\mapsto\iota(k)],\sigma'))
\end{array}
$$

When a component with specification d accesses key k, the system evaluates $\sigma(k)(d(k)\oplus_k \iota(k))$: the component-declared metadata is merged with the context-carried metadata ι, and the provider function is applied to the result. This merge follows each key's own semantics (e.g. scalar fields are overwritten while set-valued fields are unioned) and is right-biased, so $\iota(k)$ takes priority and can override the component's declaration, letting an enclosing context constrain how a component uses a coeffect without modifying that component (e.g. Section 5.2).

3.3. Component Lifecycle

The effect and coeffect models developed in the preceding sections define algebraic structures and specifications independently. To give them operational meaning, we combine them into the notion of *component*—the runtime entity whose behavior they jointly govern. Here Γ serves simultaneously as the effect context and the coeffect context; a concrete unification is given in Section 3.4.

Definition 20. A *component* over a context Γ is defined as:

$$\mathfrak{C}_\Gamma := \mathfrak{D}_\Gamma \times \mathfrak{E}_\Gamma \tag{14}$$

representing a pair (d, e) equipped with a mutable lifecycle state, where:

- $d \in \mathfrak{D}_\Gamma$: the coeffect specification, declaring the dependencies required from the environment;
- $e \in \mathfrak{E}_\Gamma$: the effect function, defining the effects contributed when the component is active.

The two sides of a component jointly determine its *target state*: from the effect side, the component must be *alive* (its effect has been applied to the context and the corresponding inverse has not yet been invoked); from the coeffect side, all declared dependencies must be satisfied ($\sigma \vDash d$). When both conditions hold the target state is `ACTIVE`; otherwise it is `INACTIVE`. Whenever the target state changes—regardless of which side caused it—a *transition* is initiated: `RELOAD` executes the component's effect function e, accumulating side effects on the context; `UNLOAD` applies the accumulated inverse to recover the context. In its simplest form, the lifecycle is the two-state model shown in Figure 1. In practice, transitions interact with various control-flow structures, requiring extensions for idempotent recovery, iteration, coherence, and asynchrony.

Figure 1 | Base component lifecycle

3.3.1. Idempotent Recovery

A component's `UNLOAD` applies the accumulated inverse to recover the context; for this to be safe, each inverse must run at most once. At the level of recover this holds by construction: Defin-

ition 5 resets φ to id_Γ, so a second recover is a no-op. This obligation arises only for the *partial* disposer that effect hands back to the caller (the $\partial\Gamma \to \partial^2\Gamma$ component of Definition 7), which excises a single effect's inverse from the accumulated φ: invoking it twice would apply that inverse twice and corrupt the context. We therefore make each returned disposer *idempotent*.

We keep each disposer's *armed* status as its own private state, so the guarded inverse stays a pure function of the state γ it acts on rather than depending on hidden mutable state. Building a guard is *generative* [40]: each application generates a fresh handle for the new disposer, so no two disposers share a handle and it stays internal to the guard. The implementation realizes this as the `armed` variable freshly captured in each `dispose` closure.

Definition 21. The *idempotent guard* $\mathrm{idem} : (\Gamma \to \Gamma) \to (\Gamma \to \Gamma)$ (and likewise over $\partial\Gamma$) makes an inverse fire on at most its first invocation. Wrapping is generative: each application generates a fresh handle h, private to the disposer it produces and absent from the type of idem. The state records the handles already spent; a fresh h is absent, hence live, and firing marks it spent:

$$\mathrm{idem}(g) = \mathbf{let}\ h \text{ fresh } \mathbf{in}\ \gamma \mapsto \begin{cases} g(\gamma)[h \mapsto ()] & \text{if } h \notin \mathrm{dom}(\gamma) \\ \gamma & \text{if } h \in \mathrm{dom}(\gamma) \end{cases} \tag{15}$$

Because h is chosen when the guard is built rather than supplied by the caller, and the spent set only grows, the disposer fires at most once yet stays a pure preconditioned effect function; $\partial\Gamma$, $\partial^2\Gamma$, and $\mathfrak{F}_\Gamma$ are left intact, and the spent set is not part of the revertible state.

The idempotent variant of effect_Γ (Definition 7) then differs by a single substitution: the returned inverse $\mathrm{track}_\Gamma(g)$ is wrapped by idem (which generates its own handle), while the forward accumulation $\varphi \circ g$ (recovered by the already-idempotent recover) is left untouched.

$$\begin{aligned} \mathrm{effect}^{\mathrm{idem}}_\Gamma &: \mathfrak{E}_\Gamma \to \partial\Gamma \to \partial^2\Gamma \\ \mathrm{effect}^{\mathrm{idem}}_\Gamma &= e \mapsto (\gamma, \varphi) \mapsto \begin{array}{l} \mathbf{let}\ (\delta, g) = e(\gamma)\ \mathbf{in} \\ ((\delta, \varphi \circ g), \mathrm{idem}(\mathrm{track}_\Gamma(g))) \end{array} \end{aligned}$$

Since only the returned inverse is wrapped, the accumulation φ and the homomorphism results of Section 3.1.2 are unaffected.

3.3.2. Iteration

A component's `RELOAD` transition may execute multiple effects in sequence, and `UNLOAD` must recover them. We model multi-step execution using *effect iterators*, where each step returns the modified context, an inverse, and a continuation:

Definition 22. Define the effect iterator $\mathfrak{E}^{\mathrm{iter}}_\Gamma$ and strict effect iterator $\mathfrak{E}^{\mathrm{iter}*}_\Gamma$ as the following recursive types:

$$\begin{aligned} \mathfrak{E}^{\mathrm{iter}}_\Gamma &:= \mu\mathfrak{I}.\ \Gamma \to \Gamma \times (\Gamma \to \Gamma) \times \mathrm{Maybe}(\mathfrak{I}) \\ \mathfrak{E}^{\mathrm{iter}*}_\Gamma &:= \mu\mathfrak{I}.\ (e : \Gamma \to \Gamma \times (\Gamma \to \Gamma) \times \mathrm{Maybe}(\mathfrak{I})) \\ &\qquad \times ((\gamma : \Gamma) \to (\mathbf{let}\ (\delta, g, o) = e(\gamma)\ \mathbf{in}\ g(\delta) = \gamma)) \end{aligned} \tag{16}$$

where $e(\gamma)$ yields a triple (δ, g, o) representing:

- δ is the new context;
- g is the inverse function of the current effect;
- o indicates the continuation:

- Nothing signals iteration termination;
- $\mathrm{Just}(i)$ provides the next iteration step.

The effect iterator transformation $\mathrm{effect}_\Gamma^{\mathrm{iter}}$ extends effect_Γ to the iterator structure through recursive invocation:

Definition 23. Define the effect iterator transformation $\mathrm{effect}_\Gamma^{\mathrm{iter}}$ as:

$$\begin{aligned}
&\mathrm{effect}_\Gamma^{\mathrm{iter}} \;:\; \mathfrak{E}_\Gamma^{\mathrm{iter}} \;\to\; \partial\Gamma \;\to\; \partial^2\Gamma \\
&\mathrm{effect}_\Gamma^{\mathrm{iter}} \;=\; i \;\mapsto\; (\gamma, \varphi) \;\mapsto\; \begin{array}{l} \textbf{let } (\delta, g, o) = i(\gamma) \textbf{ in} \\ \textbf{match } o \\ \quad \mid \mathrm{Nothing} \Rightarrow ((\delta, \varphi \circ g), \mathrm{track}_\Gamma(g)) \\ \quad \mid \mathrm{Just}(i) \Rightarrow \textbf{let } (s, r) = \mathrm{effect}_\Gamma^{\mathrm{iter}}(i)(\delta, \varphi \circ g) \textbf{ in} \\ \qquad\qquad (s, \mathrm{track}_\Gamma(g) \circ r) \end{array}
\end{aligned}$$

At each recursive step, the inverse g is composed onto φ in application order, so the accumulated recovery function $\varphi \circ g_1 \circ \cdots \circ g_k$ naturally recovers effects in LIFO order when applied.

Between any two consecutive steps, the system may *interrupt* the transition if the target state has changed, applying the inverse accumulated so far to recover the context. This property is intrinsic to the iterator model: the `Maybe`$(\mathfrak{E}^{\mathrm{iter}})$ continuation at each step constitutes a natural interruption boundary without requiring additional mechanism, as illustrated in Figure 2. In this sense the effect iterator is a reified delimited continuation, the structure that mainstream languages expose through the `yield` operator [41], so the model maps directly onto the generators they already provide.

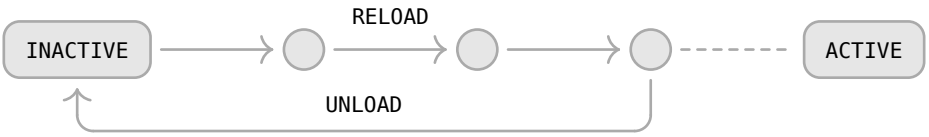


Figure 2 | Iterative transition with step-boundary interruption

A single-step effect $(\mathfrak{E}_\Gamma)$ is the degenerate case where the iterator immediately returns `Nothing`, so the transition is atomic with no interruption boundary. For multi-step transitions, the granularity of steps determines responsiveness: finer steps allow faster reaction to target-state changes, at the cost of more frequent condition checks.

As a further extension, replacing the inverse component $\Gamma \to \Gamma$ in the iterator's return type with another iterator yields a *bidirectional* effect iterator $\mathfrak{E}_\Gamma^{\mathrm{bidi}} = \mu\mathfrak{I}.\, \Gamma \to \Gamma \times \mathfrak{I} \times \mathrm{Maybe}(\mathfrak{I})$, whose triple (δ, p, n) provides both a forward continuation n and a reverse step p. This allows `UNLOAD` to also proceed incrementally and be interruptible back to `RELOAD`. However, few programming languages provide native bidirectional iteration semantics, limiting this extension's practical applicability.

3.3.3. Coherence

When a dependency is replaced at runtime, the target state may transition `ACTIVE` → `INACTIVE` → `ACTIVE` in rapid succession. If the `RELOAD` initiated for the first `ACTIVE` is still in progress (or has not yet been interrupted) when the second `ACTIVE` arrives, the system faces an `ACTIVE` → `ACTIVE` scenario where the dependency values have changed. Simply continuing the original `RELOAD` would bind the component to stale dependencies, violating consistency.

To detect such staleness, we introduce an *epoch* mechanism that versions the target state. Define the epoch function $\varepsilon_d : \Sigma \to E$ for a coeffect specification d as:

$$\varepsilon_d(\sigma) := \langle \sigma(k) \mid k \in d \rangle \tag{17}$$

Two epochs are equal iff all dependency values coincide: $\varepsilon_d(\sigma) = \varepsilon_d(\sigma') \Leftrightarrow \forall k \in d.\ \sigma(k) = \sigma'(k)$. The epoch of `INACTIVE` is a distinguished constant $\bot_E$ unequal to any active epoch.

Whenever a transition begins, the target state's current epoch $\varepsilon_{\text{target}}$ is recorded as $\varepsilon_{\text{inert}}$. At each iterator step boundary, the system compares $\varepsilon_{\text{target}}$ with $\varepsilon_{\text{inert}}$: only when they match does the next step proceed; otherwise the transition is aborted and the accumulated inverse is applied.

The epoch mechanism effectively extends the lifecycle from a one-dimensional binary `INACTIVE/ACTIVE` model to a multi-dimensional star structure. Different dependency configurations of `ACTIVE` form independent branches, all connected to the `INACTIVE` state, as shown in Figure 3.

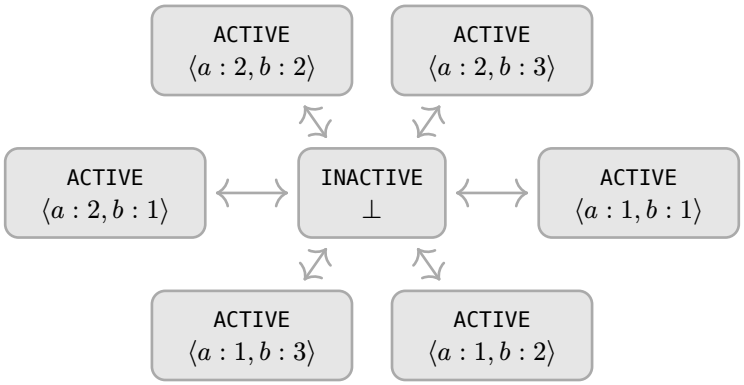


Figure 3 | Epoch-based lifecycle: each dependency configuration forms an independent branch

3.3.4. Asynchrony

The synchronous model assumes that each transition completes instantaneously relative to external state changes. When transitions involve asynchronous computation, this assumption breaks. We model non-immediacy abstractly: a transition produces a value of type $\mathrm{Future}(A)$, where Future is an opaque type constructor whose defining property is that between submission and resolution, external state may change.

Under this model, `RELOAD` and `UNLOAD` occupy real time during which the target state may change. This threatens resource safety in two ways: immediately rolling back a `RELOAD` in progress violates LIFO ordering of effects not yet finished executing, and rapid oscillation may invoke the same inverse twice. To restore safety, we promote `RELOAD` and `UNLOAD` from instantaneous transitions to *inertial states*: once entered, a transition runs to completion before the system acts on any change in target state. The resulting state machine is shown in Figure 4.

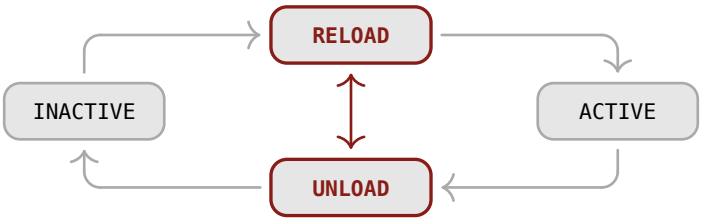


Figure 4 | Inertial state machine for asynchronous transitions

The semantics of inertial states are:

- If a component is in `RELOAD` and the target changes to `INACTIVE`, the `RELOAD` runs to completion, then `UNLOAD` begins. If the target reverts to `ACTIVE` before `RELOAD` finishes, the component proceeds normally to `ACTIVE`.
- Symmetrically, if a component is in `UNLOAD` and the target changes to `ACTIVE`, `UNLOAD` completes first, then `RELOAD` begins. If the target reverts to `INACTIVE`, the component proceeds to `INACTIVE`.
- A pending reverse transition is deferred until the current inertial state resolves, at which point it is either executed or discarded based on the then-current target.

Iterator step boundaries compose with inertial semantics: at each step, the system checks whether the epoch still matches. If not, the transition aborts (applying accumulated inverses) and the deferred transition begins; this is the synchronous interruption mechanism operating within each inertial transition.

3.4. The Context Paradigm

The component definition in Section 3.3 uses a single context Γ to carry both effects and coeffects. This section gives that unification a concrete construction, and argues that the resulting context type constitutes a programming paradigm in its own right.

3.4.1. Unified Context

For a context Γ, the effect context $\partial\Gamma$ (Section 3.1) provides a higher-level abstraction, recording the previous-level context and that level's accumulated side effects. Making this structure recursive and combining it with the coeffect context Σ yields the following type:

Definition 24. The context type Γ_∞ is defined as:

$$\Gamma_\infty := \mu\Gamma.\ \Gamma \times (\Gamma \to \Gamma) \times \Sigma \tag{18}$$

where the three projections are:

- Γ: the current context state (recursive);
- $\Gamma \to \Gamma$: the accumulated inverse for effect recovery;
- Σ: the coeffect context carrying dependency information.

Under this definition, effect becomes an endomorphism on Γ_∞, unifying the ∂-tower into a single self-similar type. The coeffect context Σ is structurally integrated: dependency operations $(\mathrm{set}, \mathrm{get})$ act on the Σ component, while the inverse function in the second component tracks their reversal. Since the type family $\mathcal{V}$ underlying Σ is unconstrained, any global state that the system needs to share across components can be encoded as a dependency with an appropriate value type—Σ subsumes all shared mutable states, not just inter-component

dependencies. Every interaction between a component and its environment passes through this single entity.

Hierarchical composition. The recursive structure of Γ_∞ supports hierarchical control: a parent context aggregates multiple child-level effects, forming a tree-shaped control structure that maintains modularity while enabling unified cross-level management. The effect transformation realizes a literal "plug-in" metaphor:

- Loading a component corresponds to executing its effects (plugging in);
- Unloading a component corresponds to recovering its effects (unplugging, without affecting other running components);
- Components at different levels of the hierarchy are independently loadable and unloadable; a parent context aggregates and manages the effects of all its children, enabling arbitrarily nested composition.

Denotation and realization. Typing an operation as an effect on Γ_∞ fixes its *denotation*, a successor state paired with an inverse, but not its *realization*, which determines how that inverse is carried out.

Definition 25. An effect function $f : \mathfrak{E}_{\Gamma_\infty}$ admits two *realizations*:

- *In-place* realization mutates the context and returns a nontrivial inverse; the successor aliases the input, and recovery runs the inverse to reverse the mutation.
- *Derived* realization leaves the input intact and returns a fresh context in the recursive structure of Γ_∞ with the identity as its inverse; recovery discards the derived context.

In a purely functional setting the two coincide; an imperative host may choose either per operation. Section 4.1.2 will provide a representative implementation of both realizations.

3.4.2. Situating the Context Paradigm

Programming paradigms differ fundamentally in how they handle side effects. Two established poles define the spectrum:

Explicit state threading (functional). To preserve referential transparency, purely functional languages model side effects as explicit transformations on state. The `State` monad $S \to (A, S)$ [22] threads an environment through every computation. This approach yields strong compositional guarantees: effects are visible in types and amenable to equational reasoning. However, it imposes significant ergonomic costs: every function in the call chain must accept and return the state parameter, even when it merely passes the state through unchanged. As the number of effect dimensions grows (logging, configuration, I/O), monadic stacking or effect-handler boilerplate proliferates.

Implicit mutation (imperative/OOP). Mainstream imperative languages permit components to modify shared state and access dependencies without explicit declaration at the call site. On the effect side, a representative example is React's `useEffect` hook: it registers a persistent side effect on the component's internal fiber, yet neither the effect target nor the registration mechanism appears as an explicit parameter—identification relies on call-order position within hidden runtime state. On the coeffect side, Java's service locator pattern (e.g., Spring's `ApplicationContext.getBean(...)`) retrieves dependencies from a global registry at runtime, requiring null checks and type casts at each call site; dependency relationships are implicit and scattered across the codebase. More generally, understanding how `f()` modifies or

depends on the system requires reading its implementation transitively. Refactoring becomes fragile because moving or removing a call may silently break distant invariants.

The context paradigm combines the traceability of the functional approach with the ergonomics of the imperative approach. Effects and coeffects are both mediated through an explicit context parameter. Each operation is therefore attributable to the specific context on which it was invoked, and hence to the component that context belongs to.

Beyond combining the strengths of both poles, the context paradigm lets the developer handle each effect and dependency individually and composes them into the system's behavior automatically. For revertible effects, the developer supplies the inverse of each atomic operation, and the inverse of any composite follows by composition (Section 3.1), so loading and teardown are reversible by construction. For reactive coeffects, a component declares only the dependencies it needs, and the runtime resolves and re-wires them automatically (Section 3.2), keeping them consistently wired as providers are added, removed, or replaced. In both directions, correctness that would otherwise rest on developer discipline becomes a structural property of the paradigm.

4. Implementation and Case Study

This section presents *Cordis*, which realizes the formal models of Section 3 as a practical programming abstraction. Cordis is a *meta-framework* of spatiotemporal composability: unlike application frameworks that target a specific domain (e.g., web routing, ORM, UI rendering), it prescribes no concrete scenario; its sole responsibility is to supply universal dynamic composition semantics. The implementation is layered into three tiers: (1) the *core library* (Section 4.1) implements the effect and coeffect systems directly; (2) the *component loader* (Section 4.2) extends the core with configuration reconciliation and hot module replacement; (3) application frameworks such as *Koishi* (Section 4.3) build domain-specific functionality on top of the former two tiers.

4.1. Core Library

Table 1 summarizes the correspondence between theoretical constructs and their runtime counterparts. In particular, we use the runtime names introduced below throughout this section, reserving the theoretical symbols for the formal correspondence. We also write `@@name` for a framework-internal symbol key, so the brackets in `ctx[@@store]` denote symbol-keyed access to an opaque slot on the context, rather than indexing into a string-keyed map.

A *fiber* is the runtime instance of a component: the stateful object that carries a component through the lifecycle of Section 3.3. It bundles the component's static specification with the live transition state read by the lifecycle algorithm. The static specification comprises its coeffect specification `fiber.inject` (d) and effect function `fiber.apply` (e), together with two related contexts: `fiber.parent`, the context on which the fiber was instantiated, and `fiber.ctx`, the fresh child context derived from it (Definition 25) in which the fiber runs. The transition state comprises the accumulated inverse `fiber.dispose` (recover), the target-state version `fiber.epoch` ($\varepsilon_d(\sigma)$, where $\perp$ denotes `INACTIVE`), and the in-flight transition handle `fiber.inertia`. Table 1 lists these against their theoretical counterparts.

Theory (Section 3)	Implementation
Γ_∞	`ctx`, the first-class context
$\mathfrak{E}_\Gamma$, $\mathfrak{E}_\Gamma^{\text{iter}}$	Effect callback returning / yielding inverses
$\text{effect}_\Gamma(e)$	`ctx.effect(callback)`
Σ, Σ^{iso}, Σ^{inter}	`ctx[@@store]`, `ctx[@@isolate]`, `ctx[@@intercept]`
$\text{get}(k)$, $\text{set}(k, v)$	`ctx.get(key)`, `ctx.set(key, value)`
$\text{isolate}(k, r)$	`ctx.isolate(key, realm)`
$\text{intercept}(k, \nu)$	`ctx.intercept(key, metadata)`
$\mathfrak{C}_\Gamma$	`fiber`, the runtime instance of a component
$d : \mathfrak{D}_\Gamma$	`fiber.inject`
$e : \mathfrak{E}_\Gamma$	`fiber.apply`
$\varepsilon_d(\sigma)$	`fiber.epoch`
recover	`fiber.dispose`, the accumulated inverse

Table 1 | Theory-to-implementation correspondence

The remainder of this section builds the core library from the bottom up. Section 4.1.1 realizes revertible effects, the sole primitive through which a context is mutated; Section 4.1.2 realizes reactive coeffects over it; Section 4.1.3 composes both into the component lifecycle; and Section 4.1.4 exposes the context-level operations built on them.

4.1.1. Effect Tracking

This section realizes revertible effects (Section 3.1). Every context mutation in Cordis flows through a single primitive, `ctx.effect`: coeffect provision, component instantiation, and every other context-mutating operation reduces to a `ctx.effect` call, so any operation performed through the context is automatically tracked and recovered upon component unloading. Operationally, `ctx.effect` is the realization of $\text{effect}_\Gamma^{\text{iter}}$ (Definition 23): it takes a callback of type $\mathfrak{E}_\Gamma^{\text{iter}}$ and lifts it to $\mathfrak{E}_{\partial\Gamma}^{\text{iter}}$, yielding a dispose closure that, when invoked, recovers the effect. Cordis accepts both $\mathfrak{E}_\Gamma$ and $\mathfrak{E}_\Gamma^{\text{iter}}$ through this one operation (ad-hoc polymorphism); we take the iterator form as representative, since a plain effect function is the degenerate iterator that yields a single inverse.

Algorithm 1 shows the construction of `ctx.effect`. We write $f \circ g$ for the disposer that runs f after g, and `id` for the no-op; prepending each new inverse therefore yields LIFO recovery.

Algorithm 1 Effect tracking

```
1  async function execute(callback, guard)
2      iter ← callback()
3      inverse ← id
4      while guard()
5          (value, done) ← await iter.next()
6          if value then inverse ← value ∘ inverse
7          if done then break
8      return inverse
9  function effect(ctx, callback)
10     armed ← true
```

```
11      task ← execute(callback, () ↦ armed)
12      async function dispose()
13          if not armed then return
14          armed ← false
15          recover ← await task
16          recover()
17      ctx.dispose ← dispose ∘ ctx.dispose
18      return dispose
```

The engine `execute` drives the callback as an effect iterator ($\mathfrak{E}^{\text{iter}}_{\Gamma}$, Definition 22) and folds the inverse yielded at each step into a single composite. Before each step it consults a caller-supplied `guard`; once the guard trips, iteration stops and only the inverses accumulated so far remain. This is the step-boundary interruption of Section 3.3.2: the `Maybe`$(\mathfrak{E}^{\text{iter}})$ continuation is realized by the iterator's `done` flag together with `guard`.

`ctx.effect` is a thin wrapper over `execute` that adds two things. First, idempotent self-disposal (realizing idem, Definition 21): the guard reports the `armed` flag, and the returned `dispose` flips `armed` to false, which simultaneously halts any in-flight iteration and makes recovery fire at most once. Second, parent composition: `dispose` is prepended to the enclosing context's accumulated inverse `ctx.dispose`, so a child effect's inverse is itself an effect on the parent, realizing the recursive structure of $\partial^2\Gamma$. The component level (Section 4.1.3) reuses the same `execute` with a guard that tests epoch stability instead of `armed`.

4.1.2. Coeffect Operations

This section realizes reactive coeffects (Section 3.2). All coeffect operations act on three symbol-keyed slots that each context carries:

- `@@store`: the value store $\sigma : (r : R) \rightharpoonup \mathcal{V}_r$ from realm symbols to typed values;
- `@@isolate`: the realm table $\rho : \mathrm{Map}(K, R)$ from coeffect keys to realm symbols;
- `@@intercept`: the interception table $\iota : (k : K) \to \mathcal{M}_k$ assigning each key its metadata.

The first two compose into the two-layer resolution $k \to \rho(k) \to \sigma(\rho(k))$: `ctx.get(key)` (Algorithm 2) reads the realm symbol $\rho(k)$ from `@@isolate`, then the bound value $\sigma(\rho(k))$ from `@@store`. The ρ indirection lets isolation redirect a key to an independent binding, whereas `@@intercept` is consulted only when a binding is accessed, adjusting how it is used rather than what it resolves to. We realize these operations in two parts: (1) *provision and notification*, which install or retract bindings and propagate the change to dependents; and (2) *isolation and interception*, which reshape how a key resolves.

Provision and notification. Since $\mathrm{set}(k, v)$ has type $\mathfrak{E}_{\Sigma}$ (Section 3.1), coeffect provision is a `ctx.effect` call and inherits its automatic tracking and recovery. Algorithm 2 implements `ctx.set(key, value)`, the concrete $\mathrm{set}(k, v)$: the callback binds a value into the store under the realm symbol $\rho(k)$, and the returned dispose function removes it. Both installation and removal invoke `notify` to propagate the change to dependent components.

Algorithm 2 Coeffect operations

```
1   function get(ctx, key)
2       realm ← ctx[@@isolate][key] ▷ ρ(k)
3       return ctx[@@store][realm] ▷ σ(ρ(k))
```

```
4  function set(ctx, key, value)
5      function callback()
6          realm ← ctx[@@isolate][key] ▷ ρ(k)
7          ctx[@@store][realm] ← value ▷ σ[ρ(k) ↦ v]
8          notify(ctx, [key])
9          return function()
10             delete ctx[@@store][realm] ▷ σ \ ρ(k)
11             notify(ctx, [key])
12     return ctx.effect(callback)
```

Algorithm 3 propagates each binding change to dependents by testing, for each live fiber, whether a changed key appears in its `fiber.inject` and resolves to the same realm; if so, it calls `refresh` (Section 4.1.3) to re-evaluate that fiber against the new state. This is the reactive classification of Definition 15: a change that flips $\sigma \vDash d$ activates or deactivates the fiber, and `refresh`'s idempotence renders a neutral change harmless. The interaction of this re-evaluation with diverse control flows is developed in Section 4.1.3.

Algorithm 3 Reactive notification

```
1  function notify(ctx, keys)
2      for fiber in all_fibers do
3          for key in keys do
4              if key ∈ fiber.inject and fiber.ctx[@@isolate][key] = ctx[@@isolate][key] then
5                  refresh(fiber)
6                  break
```

Isolation and interception. The two operations do structurally the same thing: each derives a child context that adjusts one inherited table for `key`, leaving the parent untouched, so recovery is implicit: discarding the child context suffices, with no explicit inverse to run. `ctx.isolate(key, realm)` overrides the realm mapping ρ with `realm`, or a freshly generated symbol by default (realizing isolate, Definition 17), so two contexts that assign different symbols to the same key resolve to independent bindings. `ctx.intercept(key, metadata)` merges `metadata` into the interception table ι (realizing intercept, Definition 19): following that definition, the new metadata is combined with whatever the context already carries for `key` and takes priority over it.

4.1.3. Component Lifecycle

A component is instantiated as a *fiber* by `ctx.use`. This section gives the fiber (introduced in Section 4.1) operational meaning as the inertial state machine of Section 3.3.4. Two fields drive the algorithm below: `fiber.parent`, the parent context of `fiber.ctx` that forms the component hierarchy (the recursive structure of Γ_∞, Section 3.4.1), and `fiber.inertia`, a handle to the in-flight asynchronous transition (or null if idle).

Algorithm 4 shows component instantiation. A *component* pairs a coeffect specification `component.inject` (d) with an effect function `component.apply`; instantiation binds the component's `config` into `fiber.apply` (Line 9), the config-applied effect function (e) that the lifecycle then runs. The `callback` function (Line 2) is the effect tracked in the *parent* fiber: when executed, it initiates the child's lifecycle by calling `refresh` (Algorithm 5); when recovered, it

forces the child's epoch to $\bot$ and triggers `unload`. This ensures that unloading a parent cascades to all children, a direct consequence of the $\diamond$ composition on the parent's effect context.

Algorithm 4 Component instantiation

```
1  function use(ctx, component, config)
2      function callback()
3          refresh(fiber)
4          return function()
5              fiber.epoch ← ⊥
6              unload(fiber)
7      fiber ← Fiber(parent: ctx, inject: component.inject)
8      fiber.ctx ← ctx[fiber ↦ fiber]
9      fiber.apply ← () ↦ component.apply(fiber.ctx, config)
10     ctx.effect(callback)
11     return fiber
```

Algorithm 5 realizes the inertial state machine of Section 3.3.4, in which `RELOAD` and `UNLOAD` are inertial: once entered, a transition runs to completion before the system responds to a target-state change. The `refresh` function recomputes the fiber's epoch from the coeffect store and, if the fiber is not already in a transition, initiates either a `reload` or `unload` task[2]. The `reload` function records the current epoch and executes the component's effect function `apply`. Upon completion, it checks whether the epoch still matches: if so, the fiber settles into `ACTIVE`; if not (regardless of whether the new epoch is $\bot$ or a different active configuration), it chains into `unload`. Symmetrically, `unload` recovers all tracked effects in LIFO order and then either settles into `INACTIVE` or chains into `reload`. This mutual recursion implements the inertial property: once a transition begins, it completes before any new transition can start.

Algorithm 5 Component lifecycle

```
1  function refresh(fiber)
2      epoch ← ε_d(σ)
3      if epoch = fiber.epoch then return
4      fiber.epoch ← epoch
5      if fiber.inertia then return
6      if epoch ≠ ⊥ then
7          fiber.inertia ← create_task(reload(fiber))
8      else
9          fiber.inertia ← create_task(unload(fiber))
10 async function reload(fiber)
11     epoch₀ ← fiber.epoch
12     recover ← await execute(fiber.apply, () ↦ fiber.epoch = epoch₀)
13     fiber.dispose ← recover ∘ fiber.dispose
14     if fiber.epoch = epoch₀ then
15         fiber.inertia ← null
```

[2] `create_task` schedules an async function to run concurrently and returns a handle to it (stored in `fiber.inertia`). We write it explicitly for language independence: with eager scheduling (e.g., TypeScript promises), the call is implicit and the returned promise is the handle, whereas with lazy scheduling (e.g., Python coroutines, Rust futures) the host must spawn the task for it to progress.

```
    else
        fiber.inertia ← create_task(unload(fiber))
async function unload(fiber)
    await fiber.dispose()
    fiber.dispose ← id
    if fiber.epoch = ⊥ then
        fiber.inertia ← null
    else
        fiber.inertia ← create_task(reload(fiber))
```

The epoch is computed by resolving each declared key against the current coeffect store and tupling the resolved values (Section 3.3.3). Since `notify` (Section 4.1.2) recomputes the epoch on every coeffect change, a fiber reloads precisely when its resolved values change.

The algorithm operates at two complementary levels. At the *transition* level, `reload` and `unload` check the epoch at completion, enabling inertial chaining across transitions. At the *step* level within each transition, the effect execution (Algorithm 1) checks the epoch at each iterator step boundary, enabling partial rollback within a single transition. These two mechanisms correspond to the inter-transition and intra-transition epoch checks of Section 3.3.3.

4.1.4. Context Access

The coeffect operations of Section 4.1.2 form a *reflective* API: a coeffect is written with `ctx.set(key, value)` and read with `ctx.get(key)`, both keyed by name. Cordis layers a second, more native way to extend and consume the context on top of this reflective API: *property access*. A component can access a coeffect as the property `ctx[key]`, as if it were native structure of the context, rather than through a method call. In TypeScript, Cordis realizes this with a `Proxy` whose `get` trap mediates every property access. Algorithm 6 shows how a context resolves such an access to a coeffect, atop the primitive `get` of Section 4.1.2.

Algorithm 6 Proxy-mediated context access

```
function resolve(ctx, key)
    fiber ← ctx.fiber
    repeat
        if key ∈ fiber.inject then return get(ctx, key)
        if fiber = root then throw UNDECLARED_ACCESS
        fiber ← fiber.parent.fiber
```

Algorithm 6 walks the fiber chain upward from the accessing context: at the first fiber that declares `key` in its `inject`, the access is authorized and the value is fetched by `get` (Algorithm 2); if the walk reaches the root without such a declaration, the access is rejected. This is where the proxy differs from the bare `ctx.get`: `ctx.get(key)` is a total lookup that returns the bound value or nothing and never fails, whereas the proxy rejects any access to an *undeclared* coeffect, enforcing the coeffect specification d at the point of use. A declared-but-absent value cannot arise here, since a fiber becomes `ACTIVE` only once all its declared coeffects are satisfied (Section 4.1.3).

This rejection is a *runtime* check performed at the point of access. Because a component's coeffect specification d is declared statically, the same violation is in principle detectable at

compile time, by resolving each `ctx[key]` against the declared d before execution; Section 5.3 discusses how a host language's type-level dependency declarations and compile-time metaprogramming can carry out exactly this mediation.

4.2. Component Loader

The core library equips *component developers* with imperative primitives for dynamic composition, such as `ctx.effect`, `ctx.use`, and `ctx.set`. A separate concern arises for *application orchestrators*, who assemble pre-existing components into a running system and adjust the composition over its lifetime. The *component loader* addresses this concern by introducing a declarative configuration layer: the orchestrator specifies the desired composition as a persistent data structure, and the loader translates changes to this specification into the corresponding imperative fiber operations.

4.2.1. Declarative Configuration Layer

A Cordis system's runtime is a composition of fibers, which form a tree following the context hierarchy they inhabit (Section 3.4.1). By serializing each fiber into an *entry* and organizing the entries into the same tree structure, the declarative layer captures the structure of an application as a persistent specification.

Definition 26. An *entry* declares a single fiber, recording:

- `id` — a stable identifier, used as the reconciliation key when its group's child list changes;
- `url` — the URL of the component module to instantiate;
- `isolate` — an isolation annotation applied to the entry's context;
- `intercept` — an interception annotation applied to the entry's context;
- `config` — the configuration bound into the component to form its effect function `apply`;
- `disabled` — whether the entry is administratively turned off.

At runtime, an entry manages the fiber it declares and responds to changes in these fields, binding the declarative specification to its imperative realization.

These entries form a *configuration tree* that is the authoritative record of what the system loads. An entry may be a leaf mapping to a single fiber, or its component may in turn load further components, making the entry a branch node. Cordis provides components for such grouped and nested loading: `@cordisjs/group` takes a list of child entries as its configuration and loads them as a subgroup, while `@cordisjs/include` loads an external configuration file (YAML or JSON) and grafts its entries in as a nested subtree.

When an entry's record changes, the loader reconciles incrementally: rather than tearing the fiber down and rebuilding it wholesale, it inspects which fields changed and applies the least disruptive operation for each.

- `id`, `url` — rebuilds the entry, since its identity or its component has changed;
- `isolate` — rewrites the entry's realm map, migrating any coeffect the entry provides and notifying the dependents whose resolution shifts;
- `intercept` — updated in place, as interception metadata is consulted at read time and needs no reload.
- `config` — handed to the component, which decides how to apply the new payload, typically by diffing it against the previous one and reloading only on a material change. In particular, an `@cordisjs/group` entry's `config` is its list of child entries, so it applies the

update as a keyed diff over child `ids`, creating, removing, or updating each child; since updating a surviving child re-enters this same per-field dispatch, group reconciliation and entry update recurse together down the tree.
- `disabled` — unloads the fiber when set and reloads it when cleared;

Cordis also lets a component update its own `config` or disable itself at runtime. In either case the loader writes the change back to the configuration layer, so the persisted specification stays faithful to the live system.

4.2.2. Hot Module Replacement

Hot module replacement (HMR) applies the revertible-effect pattern at the *module* level: when source files change, typically during development, the system replaces the affected modules in-place without restarting the process. Because a fiber already bounds all of its component's effects and coeffects, a module that is itself a component can be replaced through fiber operations alone: disposing the old fiber recovers everything the component installed, and a new fiber instantiated from the reloaded module reinstalls it. HMR therefore needs no developer-annotated acceptance boundaries, as opposed to Webpack [42] or Vite [43] HMR.

The `@cordisjs/hmr` component provides the HMR *engine*, which operates in three phases.

Phase 1: Module classification. The engine takes two inputs: the *stashed* set (file URLs whose contents have changed since the last reload) and the *externals* set (modules that cannot be hot-replaced and instead trigger a full restart). Writing `get_imports(url)` for the modules that `url` directly imports, it classifies the changes' dependency subgraph, marking each module `accepted` or `declined`:

Algorithm 7 Module classification

```
function classify(stashed, externals)
    accepted ← stashed
    declined ← externals
    pending ← ∅
    for url in stashed do
        pending ← pending ∪ (get_imports(url) \ (accepted ∪ declined))
    repeat
        progress ← false
        for url in pending do
            if get_imports(url) ∩ accepted ≠ ∅ then
                accepted ← accepted ∪ {url}
                pending ← pending \ {url}
                progress ← true
            else if get_imports(url) ⊆ declined then
                declined ← declined ∪ {url}
                pending ← pending \ {url}
                progress ← true
            else
                pending ← pending ∪ (get_imports(url) \ (accepted ∪ declined))
    until not progress
    declined ← declined ∪ pending
```

```
22      return (accepted, declined)
```

Seeded with the imports of the stashed files, the fixed point accepts a module once one of its imports is accepted and declines one once all of its imports are declined; any module left undecided, caught in an import cycle, defaults to declined.

Phase 2: Stale-entry detection. Using `accepted` and `declined`, the engine then filters the component entries down to the *stale* ones, whose dependency tree reaches a changed module. It walks each entry's tree with `get_dependencies`, which collects the transitive imports of a module while respecting `declined` as a boundary:

Algorithm 8 Stale-entry detection

```
1   function get_dependencies(root, declined)
2       deps ← ∅
3       function traverse(url)
4           if url ∈ deps or url ∈ declined then return
5           deps ← deps ∪ {url}
6           for child in get_imports(url) do traverse(child)
7       traverse(root)
8       return deps
9   function detect(entries, accepted, declined)
10      stale_entries ← ∅
11      for entry in entries do
12          tree ← get_dependencies(entry.url, declined)
13          if tree ∩ accepted ≠ ∅ then
14              accepted ← accepted ∪ tree
15              stale_entries ← stale_entries ∪ {entry}
16      return stale_entries
```

An entry is stale exactly when its tree intersects `accepted`; that tree is then folded into `accepted`, so every stale module along it is invalidated in the next phase.

Phase 3: Transactional reload. Finally, the engine reloads the stale entries. It invalidates the accepted modules' caches[3], backing up each removed module to enable rollback, then re-imports each stale entry's component module by its `url` and swaps in a fresh fiber:

Algorithm 9 Transactional module reload

```
1   function reload(ctx, accepted, stale_entries)
2       backup ← invalidate_caches(accepted)
3       try
4           for entry in stale_entries do
5               entry.fiber.dispose()
6               entry.fiber ← ctx.use(import(entry.url), entry.config)
7       catch error
8           restore_caches(backup)
```

[3]On Node.js, this means clearing the caches of both the ES module and CommonJS module systems, since a module imported through the ES loader can appear in both.

```
9           for entry in stale_entries do
10              entry.fiber.dispose()
11              entry.fiber ← ctx.use(backup[entry.url], entry.config)
12          throw error
```

The transactional guarantee ensures that the system never enters a half-reloaded state: if any module fails to import (e.g., due to a syntax error), the caches are restored and every stale entry is rebuilt from `backup[entry.url]`, the previous component whose cache was just restored, undoing the swaps already made.

4.3. Case Study: Koishi

Koishi is an open-source chatbot application framework built on Cordis[4]. Over four years of development, it has accumulated over 4000 community-contributed plugins[5], ranging from instant-messaging (IM) adapters and database drivers to administrative consoles and end-user features. Its scale and diversity make it a representative validation of Cordis's dynamic composability in a production setting.

Expressiveness and generality of the meta-framework. Koishi runs as a server-side bot whose every capability is realized as a plugin over the context primitives of Section 4.1; Koishi itself contributes only the chatbot-domain vocabulary. The same model reappears in a wholly different runtime: Koishi's web console is a second, independent Cordis application whose plugins compose the primitives of the browser and its user interface rather than those of the server. The disparate settings above establish two properties of the model of Section 3. (1) It is expressive: its primitives suffice to carry a complete production system, the host framework supplying only domain vocabulary. (2) It is general: it fixes how effects and coeffects compose while leaving their meaning to each application, and so presupposes neither a particular domain nor a particular runtime.

Temporal composability without cognitive overhead. The plugin systems surveyed in Section 1.2.1 cannot unload an individual extension's effects without restarting the extension host. Koishi routinely performs this operation: an orchestrator disables a plugin from the console and its effects are withdrawn in place; during development, the HMR engine re-applies edited plugins on save while preserving cache state and live connections elsewhere in the system. Cordis makes such removal not merely possible but effortless for the plugin author. Because effects performed through the context are tracked and their inverses composed automatically (Section 3.1), even an inexperienced author obtains ordered cleanup for a plugin's context-mediated effects without writing an uninstall path. This achieves the locality of concern whose absence Section 1.2.1 identifies: correctness that would otherwise rest on each author's diligence is instead discharged once, by the abstraction.

Spatial composability across an open ecosystem. In contrast to the plugin systems of Section 1.2.1, where inter-plugin dependencies are largely absent, Koishi's ecosystem exhibits a genuine dependency topology: IM adapters provide access to each messaging platform, database drivers provide persistent storage, and functional plugins declare these as coeffects and access them. Reconfiguring a provider at runtime, such as switching the storage backend or

[4]Koishi currently uses Cordis v3. This paper presents Cordis v4, which refines the effect and coeffect semantics and redesigns the loader; the core compositional model is shared across both versions.

[5]Koishi uses the term *plugin* for the concept this paper formalizes as *component*.

reconnecting an adapter, reactivates only the dependents whose resolved dependency changed (Section 3.2); a plugin whose dependency is unavailable stays inactive until it appears, without erroring. What the case study substantiates is that this composition holds across independently authored code: a plugin and its dependencies are typically written by different authors who coordinate on nothing beyond the coeffect that connects them, so reactive coeffects keep the assembly consistent across an open ecosystem of independent contributors.

Threats to validity. The evidence here is drawn from a single ecosystem in a single host language, so it cannot separate the merits of the paradigm from those of its TypeScript realization or of Koishi's particular domain, and it is observational rather than a controlled comparison against an alternative architecture. What the case study establishes is thus an existence-and-adoption result rather than a quantitative one; measuring the abstraction's overhead and its effect on developer productivity against a baseline remains future work.

5. Discussion

The formal model and implementation presented in the preceding sections introduce a programming paradigm for dynamic composability. This section examines how the paradigm extends to broader engineering concerns, and discusses the design tensions and open problems.

5.1. Service Multiplexing

Dynamic component platforms such as OSGi [44] organize composition around *services*: units of capability that a provider publishes under an interface and a consumer binds to. The Cordis coeffect model echoes this notion, with a service corresponding to the interface behind a key. Components that `provide` a service are its *providers*, and components that `inject` a service are its *consumers*. A single service may be implemented by multiple providers, and this multiplicity can be realized in two forms. (1) *Exclusive binding*: several implementations share one interface but at most one is bound at a time; the orchestrator selects which implementation is bound, and switching between them requires unloading one provider and loading another, momentarily perturbing every consumer's dependency. (2) *Service broker*: a central service that acts as the entrypoint for the interface is injected by both the backing providers and the consumers, so that multiple providers coexist and the broker dispatches each request among them. Compared to exclusive binding, the broker absorbs this perturbation: updating a backing provider leaves the broker in place, so consumers see no change to their dependency and no reload is triggered.

The service broker underlies three capabilities: load balancing, rolling updates, and cross-process invocation.

Load balancing. When several providers coexist, the broker distributes requests among them according to a configurable policy (e.g., round-robin, least-loaded, latency-weighted) or an explicit target named by the consumer. Because providers are ordinary components, they can be added or removed to scale capacity up or down; each provider registers with the broker through a revertible effect, so unloading it reverts the registration and drops it from the broker's routing set automatically.

Rolling updates. Upgrading a service implementation at runtime reduces to a controlled provider transition [45, 46]. To carry out the transition, the new provider is loaded as an

additional fiber and registers with the broker; once it becomes `ACTIVE`, traffic is gradually shifted from the old providers to the new one (e.g., by adjusting selection weights), and the old providers are unloaded once they no longer carry in-flight requests. This provider transition turns what is traditionally an infrastructure-level operation (e.g., container orchestration, blue-green deployment) into an application-level composition pattern.

Cross-process invocation. The service broker can also be applied across process boundaries [47]. Each process hosts its own Cordis context with local providers; a coordinating component links them, treating each as a remote provider. Cross-process service access is mediated by an RPC mechanism that preserves the interface, making the distribution transparent to consumers. One caveat is that a cross-process call incurs latency and may fail mid-flight, so exposing it synchronously would block the caller. An interface intended to be exposed across processes must therefore be designed against an asynchronous contract.

5.2. Access Control and Sandboxing

Given an application assembled from independent components, securing the application calls for two complementary mechanisms: (1) constraining what dependencies a component may access, and (2) isolating untrusted code from the host environment. Cordis supports the first through dependency declarations and interception; the second requires an external isolation boundary.

Capability-based access control. The dependency access mechanism (Section 4.1.4) already constitutes a form of access control over proxy-mediated properties: a component can only access dependencies it has declared; an undeclared access raises an error. This is structurally similar to capability-based security [48–50], where authority is conferred by possession of a reference rather than by ambient authority. The `inject` declaration acts as a capability request, and the context proxy acts as a capability mediator. Since these requests are declared statically, the complete set of proxy-mediated capabilities a component requires is known before it runs, letting the orchestrator review and approve them at load time rather than discovering accesses as they happen.

This mediation generalizes to fine-grained policy through the interception mechanism. Access-control metadata can be carried by contexts or declared by components (Definition 18), and the provider consults it when the dependency is invoked to decide whether a request is permitted. For example, a filesystem dependency may carry metadata declaring which paths a component may read or write, and the provider checks each call against the metadata. Because this interception lives on the context rather than in either party's code, an orchestrator can adjust it to constrain any component's access to a dependency without modifying the provider, e.g., granting read-only database access to a community component whereas a core component retains full access. Moreover, since interception affects only how a dependency is invoked, not whether it is satisfied, it can be installed, reconfigured, or removed at runtime without triggering any reload or perturbing the dependency graph.

Sandboxing untrusted components. When a component's code cannot be trusted, language-level access control is insufficient, since a malicious component with access to the host runtime can reach the underlying objects directly, rendering such checks moot. True isolation requires an execution boundary beyond the reach of language-level means, such as software fault isolation [51], an isolated language runtime, a sandboxed process, or a virtualized container [52]. Whatever the mechanism, the untrusted component runs in its own isolated context and reaches host-provided dependencies through a bridge, generalizing the cross-

process invocation of Section 5.1: the same transparency argument renders this bridged access indistinguishable from local injection. On the host side, the bridge is an ordinary fiber whose capabilities can be attenuated by the access control described above.

5.3. Language Independence and Selection

Although Cordis is implemented in TypeScript, the context paradigm is language-agnostic: spatiotemporal composability is defined only by its two composability dimensions, and thus can be realized in any language that meets certain requirements along both. We analyze these requirements along each dimension in turn.

Temporal composability. At its most basic, temporal composability requires *closures*: a revertible effect pairs an action with an inverse, and that inverse must be captured as a value, along with the state it restores, so it can be replayed on teardown. Beyond this, a component's code and the side effects of loading it must be introducible and retractable at runtime.

How a language meets this second requirement depends on its execution model. In managed runtimes, this takes the form of a programmatic module registry, where a loaded module can be evicted from the registry and garbage-collected once unreferenced; Node.js, for instance, exposes such a registry.[6] Native code exposes no module registry, so introduction and retraction take the form of explicit dynamic linking and unlinking (e.g., `dlopen/dlclose` on Unix, `LoadLibrary/FreeLibrary` on Windows) [53], i.e., loading object code into a running process and later detaching it. WebAssembly takes one path or the other depending on its embedder: a module instance is reclaimed by the host's collector under a managed embedder (e.g., a JavaScript host), or released when a native embedder drops it (e.g., Wasmtime). Across these mechanisms, the revertible effects model treats loading as an effect on the context, with inverses that undo the registration of symbols, types, or handlers the module introduced.

Spatial composability. Spatial composability requires a mechanism for components to declare their dependencies and for the runtime to provide and inject these dependencies. This reduces to a dependency injection (DI) problem [39], which manifests at two levels that differ across languages: how dependencies are *typed* and how their access is *mediated*.

At the type level, the language should provide a way for developers to express well-typed dependency access. A consumer obtains a coeffect by reading its key from the context, so the context type (Section 3.2.1) must record each key's coeffect. Typeclasses (Haskell) [54] and traits (Rust) [55] achieve this by letting a provider extend the context type from its own module through an `instance` or `impl` [56]. TypeScript's module augmentation [57] likewise lets a provider module merge declarations into the context type.

At the runtime level, dependency access must be dynamically mediated: the coeffect behind a key may change as providers are loaded and unloaded, and may be resolved differently across contexts. The language therefore needs a way to interpose on access transparently, leaving the consumer's code unchanged, e.g., via JavaScript's `Proxy` object [58] or Python's descriptor protocol (`__get__`) [59]. Absent such a primitive, runtime reflection [60, 61] can mediate access dynamically, at the cost of type safety and developer experience.

Across both levels, metaprogramming facilities supply the typing and the mediation together. Annotations [62] and decorators attach metadata to a declaration, which a processor

[6] CommonJS exposes the module cache via `require.cache`; ES modules provide no public eviction API, though modules can still be managed through engine-internal interfaces.

expands into the accessor that mediates access; compile-time metaprogramming (e.g., Rust procedural macros, Scala macros [63], Zig comptime) emits, for each dependency, a typed declaration together with such an accessor, dispensing with a general-purpose interception primitive.

5.4. Mutual Dependencies and Component Granularity

In the reactive coeffect model, a dependency cycle simply leaves the involved components permanently inactive: given two components A and B, if A requires a key provided by B and B a key provided by A, neither's satisfaction predicate can ever become true. Unlike deadlock in concurrent systems, this state is statically predictable (from the dependency declarations alone) and produces no runtime error; it is a silent non-activation rather than a failure.

In practice, most apparently mutual dependencies can be decomposed into finer-grained components that eliminate the cycle. Consider two components: a server (providing a network interface) and an access controller (enforcing authorization policies). The two components interact bidirectionally: the access controller mediates requests arriving at the server, and the server exposes an endpoint for modifying access-control policies. A monolithic design would make each component depend on the other. However, the two interaction directions are logically independent concerns. Decomposing them yields four components: server-core, access-control-core, request-mediation (depending on both cores to apply access control to incoming requests), and policy-management (depending on both cores to expose policy modification via the server). Through this approach, the cycle is eliminated because neither core depends on the other; only the integration components depend on both.

This decomposition is always possible in principle, since every bidirectional interaction can be factored into independent unidirectional bindings, but it increases the number of components: in the general case, given n mutually interacting components, the number of integration components can grow quadratically with n, since each pair of interacting components may require a distinct component for each direction of interaction. This does not affect correctness or runtime performance (components are lightweight), and finer granularity can be beneficial: users gain the ability to load only the specific integration bindings they need, effectively increasing the system's composability. However, it may affect developer experience: more components require more configuration, more naming, and more cognitive overhead in understanding the dependency graph.

Mitigating this granularity cost is an engineering concern rather than a theoretical one. Practical strategies include package bundling (i.e., grouping related fine-grained components into a single installable unit), convention-based wiring (i.e., automatically connecting components whose names or types match a pattern), and scaffold tooling (i.e., generating boilerplate integration components from declarative specifications). These strategies preserve the formal guarantees of the acyclic model while reducing the authoring burden to something closer to the monolithic case.

5.5. Dependency Typing and Versioning

In the formal model, a dependency link is established purely by key identity: a component providing key k satisfies any component declaring k in its dependency set. The type family $\mathcal{V}_k$ ensures type-level agreement within a single compilation unit, but this guarantee breaks

down when components are developed and built independently, which is a common scenario in component ecosystems. This breakage leads to two distinct problems.

Interface drift. A provider may modify the interface associated with k (adding fields, changing method signatures, altering behavioral contracts) between versions, while a consumer compiled against an earlier interface continues to declare the same key k. The dependency is satisfied at the coeffect level ($k \in \mathrm{dom}(\sigma)$), yet the runtime value no longer conforms to the consumer's expectations, leading to type errors, method-not-found failures, or silent behavioral divergence [64].

Key collision. Two independently developed providers may use the same key name k to denote entirely unrelated interfaces. Since key identity alone establishes the link, a consumer expecting one provider's interface will accept the other's value without any compatibility check. Unlike interface drift, where the provider and consumer at least share a common lineage, key collision involves no relationship whatsoever between the expected and actual types, making the resulting failures unpredictable and difficult to diagnose.

Both problems point to the same gap: the coeffect model provides only *nominal* linking (by key name) but no *versioned* or *structural* linking (by interface compatibility) [65]. We discuss three approaches to the gap, from most infrastructure-coupled to most language-agnostic.

Key namespacing. Extending the key space from K to $K \times P$, where P identifies the interface-defining package, eliminates key collision by construction: independently developed interfaces with the same local name occupy distinct keys. This is the most direct solution but also the most coupled: it embeds the package namespace into the formal model itself, making the system dependent on an external package registry for key identity.

Peer dependencies. A lighter coupling is to declare version constraints through the host-language package manager [66]. This is the approach Cordis currently adopts. Component dependencies are semantically *peer dependencies*: a component does not bundle its dependencies internally but expects the runtime context to supply them. Package managers with peer dependency support (e.g., npm) can enforce version compatibility: if the version of the package providing a key falls outside a consumer's declared peer range, the incompatibility is caught at install time rather than surfacing as a runtime failure. However, this approach has two limitations: (1) it depends on providers faithfully adhering to semantic versioning, which is an unenforceable convention; (2) package managers typically resolve each dependency to a single version, which prevents loading components from multiple versions of the same package within one application.

Structural compatibility. A fully language-agnostic approach would replace the membership check $k \in \mathrm{dom}(\sigma)$ with a compatibility predicate that verifies the provider's actual interface structurally subsumes the consumer's expectation. This is analogous to structural subtyping [67]: a provider satisfies a consumer if the provided interface is a subtype of the required interface. The challenge lies in defining this predicate language-agnostically: structural compatibility is straightforward for record types (width subtyping) but becomes complex for behavioral contracts (e.g., pre/postconditions [68], effect specifications [21]), and undecidable once parametric polymorphism introduces bounded quantification [69].

These three approaches address different aspects of the problem. Designing a unified dependency model that combines these approaches while preserving the dynamic composition guarantees of the coeffect model remains an open problem.

6. Related Work

Dynamic composability intersects several established research areas. We survey the most relevant lines of work and distinguish our contribution from each of them.

6.1. Effect and Coeffect Systems

Section 2 reviewed effects and coeffects as the theoretical pillars underlying our work. Three lines extend them in directions relevant to Cordis: recasting effects as capabilities the context must supply, giving effects a reversible rather than an interpretive semantics, and unifying effects and coeffects under a single graded discipline.

Algebraic effects as capabilities. Algebraic effects (Section 2.1) make effect operations visible to the type system. The extension closest to our work is Brachthäuser et al.'s Effekt language, which reinterprets effect types as *capabilities* [70, 71]: an effect type expresses what a computation requires from its context rather than what side effects it may produce. The same reinterpretation has reached mainstream practice: the Effect-TS library [72] records an effect's context requirements in a dedicated type parameter, so that a computation's type states which services its surrounding context must supply; like Effekt, it discharges effects by interpretation over static types rather than reverting them. This perspective, like ours, treats the context as a mediator of capabilities. Cordis and Effekt differ in two respects. In purpose, algebraic effects make effects visible to enable *modular interpretation*, giving one operation many handler semantics, whereas Cordis makes them visible to enable *tracking and reversion*, pairing every context transformation with an inverse. In setting, Effekt disciplines effects statically at the type level, defaulting to scope-based reasoning in which capabilities are second-class and confined to their lexical scope, and recovering first-class use through boxing, which lifts that restriction by tracking captured capabilities in types; Cordis instead disciplines effects at runtime, aiming at complete resource recovery on component removal.

Reversible effect semantics. A parallel line gives effects a reversible semantics rather than an interpretive one. Heunen et al. [73] model side effects in a reversible setting by adapting Hughes' arrows to *dagger arrows* and *inverse arrows*, capturing effects such as serialization and mutable store whose operations admit inverses. This is the formal account closest to our revertible effects: both pair each effect with an inverse rather than discharging it through a handler. The two differ in where reversibility resides. Heunen et al. work in a denotational, categorical setting where reversibility is a global property, guaranteed by construction since every computation is invertible, and the inverse is recovered from categorical structure. Cordis instead tracks inverses at runtime, requiring not that the whole computation be reversible but only that each atomic effect admit an inverse, from which the inverse of any composite is derived by composition (Section 3.1).

Graded types as unified effects and coeffects. Orchard et al. [74] proposed *graded modal types* as an umbrella notion encompassing both effect reasoning (via graded monads) and coeffect reasoning (via graded comonads), realized in the Granule language, demonstrating that a single type system can track both what a computation does and what it needs; more recent work extends coeffects to imperative Java-like languages [75, 76] and to call-by-push-value [77]. All of these operate at the type level: effects and coeffects are static annotations checked at compile time over lexically fixed scopes. Our contribution is orthogonal to this analysis: we lift the same two notions to runtime mechanisms, which lets Cordis handle dynamic composition.

Temporal retraction and spatial dependency are re-resolved as the set of loaded components evolves, instead of being settled once over a fixed program text.

6.2. Programming Paradigms

Section 3.4.2 established the context paradigm as a discipline that mediates effects and coeffects through an explicit context. Two established paradigms warrant explicit comparison: one shares our terminology, the other our treatment of crosscutting concerns.

Context-oriented programming. COP [78, 79] equips a language with *layers*—partial method and class definitions that are activated and deactivated at runtime according to the execution context, so that behavior adapts without the base code naming its context dependencies [80]. COP and Cordis coincide in treating context as a first-class, runtime-mutable entity and in activating and deactivating behavior dynamically, but the resemblance is nominal. In COP, "context" denotes the ambient execution situation (e.g., location, user, mode), and activation changes method dispatch within a dynamically scoped extent; a layer neither tracks the side effects it induces nor reverts them, and activation is not governed by dependency satisfaction. In Cordis, the context is the Γ_∞ entity mediating effects and coeffects: activation runs a component's revertible effects and is driven by reactive coeffect satisfaction (Section 3.2), and deactivation reverts them in full. COP varies what behavior runs; Cordis composes and reverts what effects and dependencies a component installs. Their difference is one of trade-off. COP folds activation into the host language's method dispatch, gaining dynamically-scoped layer extents at the cost of language specificity, whereas Cordis, as a language-agnostic overlay, resolves activation reactively over a shared context. Cordis can thus express as a coeffect only COP's global, value-driven fragment: context-dependent selection among implementations, but not dynamically-scoped activation.

Aspect-oriented programming. AOP [81, 82] modularizes a crosscutting concern into an *aspect*: a *pointcut* that quantifies over *join points* selected in the base program, and *advice* woven in at each. Cordis addresses the same problem of contextual behavior that would otherwise scatter across components, but its analogue of an aspect is a coeffect: a shared point of mediation many components declare a dependence on, so that crosscutting behavior can be reshaped there without editing any of them. The two paradigms then differ on two axes. (1) *Declaration versus obliviousness*: an AOP pointcut is oblivious and quantified, matching arbitrary join points whose code is unaware it is advised, whereas Cordis confines crosscutting to the coeffects each component declares, so its reach is exactly that declared surface. This yields determinacy and traceability: an application orchestrator can inspect and govern what cross-cuts a component at the configuration layer, without reading or analyzing its source, whereas an AOP concern is legible only through the aspects that quantify over it. (2) *Lifecycle integration*: a crosscutting change in Cordis is carried by a component's effects, reverted when the component unloads and propagated reactively to its dependents, so it is one move within the dynamic composition model; dynamic-AOP systems [83, 84] can also weave and unweave at runtime, but as a standalone operation, neither bound to a component's lifecycle nor triggering re-resolution among the advised code.

6.3. Temporal Composability

Temporal composability concerns replacing or removing a component in a running program while recovering the effects it installed. Prior approaches divide by how they treat a departing

component's state and effects: carrying state forward to a successor version, recovering effects through developer-authored cleanup, or reversing effects automatically within a scope fixed in advance.

Stateful forward migration. A broad family of systems replaces components in a running program without downtime by carrying their state forward across versions. All observe the same timing discipline: a component may be swapped only once it reaches a safe, interaction-free point. Kramer and Magee established this criterion as *quiescence* [45], which Vandewoude et al. later relaxed to the less disruptive *tranquility* [46]; our rolling-update pattern (Section 5.1) enforces it by draining in-flight requests before unloading a provider. Dynamic software updating (DSU) then migrates state forward through hand-written transformation functions: Hicks et al.'s general-purpose DSU for C [85], Stoyle et al.'s type-safe update points via con-freeness analysis [86], and Hayden et al.'s Kitsune [87] all map old-version data to new-version representations, inheriting heap objects, open files, and connections in place while re-initializing whatever is left unmigrated. The same discipline extends to persistent state: Overeem et al. [88] convert a running event store's data between schema versions through hand-written upgrade operations while keeping the system available. Erlang/OTP [14] takes the same stance at the process level, migrating state through `code_change/3` and recovering from faults by restarting supervised processes rather than reverting their effects; JavaScript's Hot Module Replacement (e.g., webpack [42], Vite [43]) does the same at the module level, handing state forward through the `module.hot` or `import.meta.hot` API across a reload. Compared with Cordis's module replacement (Section 4.2), these approaches migrate in-memory state more gracefully: Cordis reverts the old component's tracked effects and reapplies the new component's from a clean slate, so a component's own in-memory state does not survive a reload unless placed in a longer-lived dependency, and layering DSU-style forward migration atop revertible effects is future work. Cordis's approach is nonetheless more general in two respects: it needs no hand-written migration functions of the kind DSU and HMR require, and it supports unloading a component entirely and recovering its resources, not merely updating one in place.

Developer-authored recovery. A second family recovers a component's effects through cleanup or compensation logic that the developer writes by hand. Plugin lifecycle conventions (e.g., OSGi [44], Eclipse's extension points, IntelliJ and VSCode) delegate cleanup to developer-written unload callbacks; the Command pattern [89] encapsulates an operation together with an `undo` method for undo/redo stacks; the saga model [37] structures a long-lived transaction as steps each paired with a compensating action; algebraic effect handlers can attach finalizers that run on teardown [90]; and event sourcing [91] retracts state by appending compensating events rather than executing a true inverse. In all of them the inverse is an unenforced duty, decoupled from the operation, so that a forgotten one leaks resources silently (as documented empirically in Section 1.2.1). React's `useEffect` hook [92] comes closest to pairing an effect with its inverse structurally, returning a cleanup the runtime invokes before each re-execution and on unmount. Its shortfall is composability: a hook may be called only at the top level of a component or another hook, never inside a conditional, loop, or nested function, and its effect body accepts neither an async function nor an iterator. Effects thus cannot be assembled from other effects or interleaved with control flow, leaving nothing from which a composite inverse could be derived. Cordis effects carry no such restriction: they are ordinary operations that compose freely and may run asynchronously, and require a hand-written inverse only for each atomic effect, from which the inverse of any composite is derived by composition, so that assembling existing effects requires writing no inverses at all. This structural pairing of every effect with its inverse makes complete recovery an invariant of the system rather than a matter of developer discipline.

Statically scoped reversal. A third family reverses effects automatically, by construction, but confines reversal to a scope fixed in advance. Software transactional memory [93, 94], descended from hardware transactional memory [95], records a read/write log so that a group of memory operations either commits or aborts, rolling memory back to its pre-transaction state. Reversible computing, from Landauer and Bennett's thermodynamic analyses [96, 97] to reversible languages such as Janus [98], goes further and makes every step of a whole computation globally invertible. Linear types [99], RAII [4], and Rust's ownership system [55] tie a resource's release to a lexical region. Each fixes the scope and reach of reversal statically; Cordis, by contrast, fixes no such scope in advance: it reverts arbitrary context operations over a component's lifecycle, and treats lexical resource management as complementary, appropriate for local resources within a single component.

6.4. Spatial Composability

Spatial composability concerns how a component's dependencies on others are declared and bound. Prior mechanisms divide by how binding responds to change: wiring dependencies once at initialization, reacting to the availability of whole components, or propagating change at the granularity of individual values.

Initialization-time dependency wiring. Two established mechanisms wire components together at initialization time. Dependency injection frameworks [39] (e.g., Spring [100], Guice, Angular, Inversify) inject dependencies into components at initialization, and UI framework context (e.g., Vue.js's `provide/inject` and React's Context API) passes them along a component tree. Some support dynamic scoping (e.g., Spring's prototype/request scopes, Angular's hierarchical injectors), but neither re-resolves reactively: when a provider is replaced or withdrawn at runtime, existing dependents are neither deactivated nor re-initialized, and none offers lifecycle management of the kind our component state machine provides. Cordis's reactive coeffects (Section 3.2) supply this: the notification mechanism triggers lifecycle transitions whenever the satisfaction predicate changes.

Availability-reactive component models. The closest precedent to our reactive coeffects reacts to service availability. OSGi's Declarative Services and iPOJO [101, 102] let components declare provided and required services, with the runtime automatically activating and deactivating them as services appear and disappear; iPOJO's Gravity project [102] explicitly targets autonomous runtime adaptation to changing service availability, and its provide/require model directly prefigures Cordis's `ctx.provide/ctx.get` pattern. R-OSGi [47] extends the same abstraction transparently to distributed settings via RPC, mapping network failures to service-withdrawal events, a pattern Section 5.1 discusses as an extension of the Cordis model. All these systems recover through a deactivation callback, which is limited in two ways. First, the callback is hand-written, so resource safety rests on developer discipline and a forgotten one leaks silently. Second, the callback is synchronous: should teardown require an asynchronous exchange with the departing dependency, the frameworks offer no protocol to await it, forcing a blocking wait against a reference that may already be stale. Cordis's reactive coeffects close both gaps: deactivation reverts the dependents' accumulated effects, and its inertial `UNLOAD` state (Section 3.3.4) runs asynchronous teardown to completion before acting on further change.

Value-level reactivity. Functional reactive programming (FRP) [103] and its modern incarnations (e.g., signals [104, 105] in SolidJS, Vue's reactivity system, Angular Signals) propagate change at a *value-level* granularity: when a signal changes, derived computations are re-evaluated synchronously or under a scheduler [106]. Cordis's reactive coeffects act at a *component-*

level granularity, adding asynchronous lifecycle semantics that value-level propagation does not model. The two are complementary rather than competing: a Cordis coeffect can itself carry reactive values, and a component updates on only the parts it actually consumes, refining component-level reactivity into finer-grained reactive coeffects that span both levels.

7. Conclusion

We have presented a formal foundation for dynamic composability by lifting two type-theoretic concepts of effects and coeffects to runtime mechanisms. *Revertible effects* address temporal composability by equipping context transformations with explicit inverses and proving that effect tracking preserves the compositional operation, guaranteeing complete state recovery upon component removal. *Reactive coeffects* address spatial composability by formalizing typed dependency contexts with satisfaction-based notification, coeffect isolation, and coeffect interception. A *component lifecycle* model with orthogonal extensions for diverse control flows operationalizes the interaction of both mechanisms. Together, these yield a unified *context type* that integrates the effect and coeffect contexts into a coherent programming paradigm. We realize this paradigm as the Cordis meta-framework, with a core library providing effect tracking and coeffect resolution, as well as a declarative component loader with configuration reconciliation and hot module replacement. The Koishi case study validates the design of Cordis in a production system with over 4000 community plugins.

Beyond human-curated plugin ecosystems, a compelling direction for future validation is self-evolving agent harnesses (Section 1.2.2), where an AI agent generates and replaces its own harness components continuously and with little human oversight. Applying Cordis in such a setting would validate the temporal guarantees of complete recovery under rapid component replacement, as well as the spatial guarantees of dependency coordination under frequent topological change. Such validation would demonstrate the paradigm's applicability as a foundation for recoverable, coordinated, and continuous self-evolution in agent harnesses and other autonomous systems.

References

[1] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, 1972, doi: 10.1145/361598.361623.

[2] D. Birsan, "On Plug-ins and Extensible Architectures," *ACM Queue*, vol. 3, no. 2, pp. 40–46, 2005, doi: 10.1145/1053331.1053345.

[3] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, 2016, doi: 10.1145/2890784.

[4] B. Stroustrup, *The Design and Evolution of C++*. Addison-Wesley, 1994.

[5] S. Marlow, S. Peyton Jones, A. Moran, and J. Reppy, "Asynchronous Exceptions in Haskell," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, in PLDI '01. New York, NY, USA: Association for Computing Machinery, 2001, pp. 274–285. doi: 10.1145/378795.378858.

[6] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley, 2002.

[7] R. Lopopolo, "Harness Engineering: Leveraging Codex in an Agent-First World." [Online]. Available: https://openai.com/index/harness-engineering/

[8] Anthropic, "Harness Design for Long-Running Application Development." [Online]. Available: https://www.anthropic.com/engineering/harness-design-long-running-apps

[9] L. Wang *et al.*, "A Survey on Large Language Model Based Autonomous Agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, 2024, doi: 10.1007/s11704-024-40231-1.

[10] Y. Qin *et al.*, "Tool Learning with Foundation Models," *ACM Computing Surveys*, 2025, doi: 10.1145/3704435.

[11] C. Packer, V. Fang, S. G. Patil, K. Lin, S. Wooders, and J. E. Gonzalez, "MemGPT: Towards LLMs as Operating Systems," *CoRR*, 2023.

[12] T. Guo *et al.*, "Large Language Model Based Multi-Agents: A Survey of Progress and Challenges," in *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence*, in IJCAI 2024. 2024, pp. 8048–8057. doi: 10.24963/ijcai.2024/890.

[13] T. Cai, X. Wang, T. Ma, X. Chen, and D. Zhou, "Large Language Models as Tool Makers," in *Proceedings of the Twelfth International Conference on Learning Representations*, in ICLR 2024. 2024.

[14] J. Armstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," Doctoral dissertation, 2003.

[15] E. Moggi, "Notions of computation and monads," *Information and Computation*, vol. 93, no. 1, pp. 55–92, 1991, doi: 10.1016/0890-5401(91)90052-4.

[16] G. Plotkin and J. Power, "Adequacy for Algebraic Effects," in *Foundations of Software Science and Computation Structures*, F. Honsell and M. Miculan, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 1–24.

[17] T. Petricek, D. Orchard, and A. Mycroft, "Coeffects: unified static analysis of context-dependence," in *Proceedings of the 40th International Conference on Automata, Languages, and Programming - Volume Part II*, in ICALP'13. Riga, Latvia: Springer-Verlag, 2013, pp. 385–397. doi: 10.1007/978-3-642-39212-2_35.

[18] M. Gaboardi, S.-y. Katsumata, D. Orchard, F. Breuvart, and T. Uustalu, "Combining effects and coeffects via grading," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, in ICFP 2016. Nara, Japan: Association for Computing Machinery, 2016, pp. 476–489. doi: 10.1145/2951913.2951939.

[19] A. Church, "A Formulation of the Simple Theory of Types," *The Journal of Symbolic Logic*, vol. 5, no. 2, pp. 56–68, 1940, doi: 10.2307/2266170.

[20] B. C. Pierce, *Types and Programming Languages*. MIT Press, 2002.

[21] J. M. Lucassen and D. K. Gifford, "Polymorphic Effect Systems," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '88. San Diego, California, USA: Association for Computing Machinery, 1988, pp. 47–57. doi: 10.1145/73560.73564.

[22] P. Wadler, "Monads for functional programming," in *Program Design Calculi*, M. Broy, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 233–264.

[23] G. Plotkin and J. Power, "Notions of Computation Determine Monads," in *Foundations of Software Science and Computation Structures*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 342–356. doi: 10.1007/3-540-45931-6_24.

[24] G. Plotkin and M. Pretnar, "Handlers of Algebraic Effects," in *Programming Languages and Systems (ESOP)*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 80–94. doi: 10.1007/978-3-642-00590-9_7.

[25] M. Pretnar, "An Introduction to Algebraic Effects and Handlers. Invited tutorial paper," *Electron. Notes Theor. Comput. Sci.*, vol. 319, no. C, pp. 19–35, Dec. 2015, doi: 10.1016/j.entcs.2015.12.003.

[26] D. Leijen, "Koka: Programming with Row Polymorphic Effect Types," *Electronic Proceedings in Theoretical Computer Science*, vol. 153, pp. 100–126, Jun. 2014, doi: 10.4204/eptcs.153.8.

[27] D. Leijen, "Type directed compilation of row-typed algebraic effects," in *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, in POPL '17. Paris, France: Association for Computing Machinery, 2017, pp. 486–499. doi: 10.1145/3009837.3009872.

[28] A. Bauer and M. Pretnar, "Programming with algebraic effects and handlers," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 108–123, Jan. 2015, doi: 10.1016/j.jlamp.2014.02.001.

[29] K. Sivaramakrishnan *et al.*, "Retrofitting parallelism onto OCaml," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, Aug. 2020, doi: 10.1145/3408995.

[30] T. Petricek, D. Orchard, and A. Mycroft, "Coeffects: a calculus of context-dependent computation," in *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, in ICFP '14. Gothenburg, Sweden: Association for Computing Machinery, 2014, pp. 123–135. doi: 10.1145/2628136.2628160.

[31] T. Uustalu and V. Vene, "Comonadic Notions of Computation," *Electronic Notes in Theoretical Computer Science*, vol. 203, no. 5, pp. 263–284, 2008, doi: 10.1016/j.entcs.2008.05.029.

[32] A. Brunel, M. Gaboardi, D. Mazza, and S. Zdancewic, "A Core Quantitative Coeffect Calculus," in *Proceedings of the 23rd European Symposium on Programming Languages and Systems - Volume 8410*, Berlin, Heidelberg: Springer-Verlag, 2014, pp. 351–370. doi: 10.1007/978-3-642-54833-8_19.

[33] J. Reed and B. C. Pierce, "Distance makes the types grow stronger: a calculus for differential privacy," *SIGPLAN Not.*, vol. 45, no. 9, pp. 157–168, Sep. 2010, doi: 10.1145/1932681.1863568.

[34] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '99. San Antonio, Texas, USA: Association for Computing Machinery, 1999, pp. 147–160. doi: 10.1145/292540.292555.

[35] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, May 1976, doi: 10.1145/360051.360056.

[36] U. Dal Lago and F. Gavazzo, "A relational theory of effects and coeffects," *Proc. ACM Program. Lang.*, vol. 6, no. POPL, Jan. 2022, doi: 10.1145/3498692.

[37] H. Garcia-Molina and K. Salem, "Sagas," in *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, in SIGMOD '87. 1987, pp. 249–259. doi: 10.1145/38713.38742.

[38] R. Johnson and B. Foote, "Designing Reusable Classes," *Journal of Object-Oriented Programming*, vol. 1, pp. 22–35, 1988.

[39] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern." [Online]. Available: https://martinfowler.com/articles/injection.html

[40] A. M. Pitts and I. D. B. Stark, "Observable Properties of Higher Order Functions that Dynamically Create Local Names, or What's New?," in *Mathematical Foundations of Computer Science 1993* (*MFCS 1993*), in Lecture Notes in Computer Science, vol. 711. Springer, 1993, pp. 122–141. doi: 10.1007/3-540-57182-5_8.

[41] R. P. James and A. Sabry, "Yield: Mainstream Delimited Continuations," in *First International Workshop on the Theory and Practice of Delimited Continuations* (*TPDC 2011*), 2011, pp. 20–32.

[42] webpack, "Hot Module Replacement." [Online]. Available: https://webpack.js.org/api/hot-module-replacement/

[43] Vite, "HMR API." [Online]. Available: https://vite.dev/guide/api-hmr

[44] OSGi Alliance, *OSGi Core Release 8*. OSGi Alliance, 2020.

[45] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990, doi: 10.1109/32.60317.

[46] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 856–868, 2007, doi: 10.1109/tse.2007.70733.

[47] J. S. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed Applications Through Software Modularization," in *Proceedings of the ACM/IFIP/USENIX 8th International Middleware Conference*, in Middleware '07. 2007, pp. 1–20. doi: 10.1007/978-3-540-76778-7_1.

[48] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM*, vol. 9, no. 3, pp. 143–155, 1966.

[49] M. S. Miller, K.-P. Yee, and J. Shapiro, "Capability Myths Demolished," in *Technical Report SRL2003-02*, 2003.

[50] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical Capabilities for UNIX," in *Proceedings of the 19th USENIX Security Symposium*, 2010, pp. 29–46.

[51] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient Software-Based Fault Isolation," in *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, in SOSP '93. 1993, pp. 203–216. doi: 10.1145/168619.168635.

[52] A. Barth, A. P. Felt, P. Saxena, and A. Boodman, "Protecting Browsers from Extension Vulnerabilities," in *Proceedings of the 17th Annual Network and Distributed System Security Symposium*, in NDSS '10. 2010.

[53] W. W. Ho and R. A. Olsson, "An Approach to Genuine Dynamic Linking," *Software: Practice and Experience*, vol. 21, no. 4, pp. 375–390, 1991, doi: 10.1002/SPE.4380210404.

[54] P. Wadler and S. Blott, "How to Make Ad-hoc Polymorphism Less Ad Hoc," in *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '89. 1989, pp. 60–76. doi: 10.1145/75277.75283.

[55] N. D. Matsakis and F. S. K. II, "The Rust Language and Type System," in *ACM SIGPLAN ML Family Workshop*, Gothenburg, Sweden, Sep. 2014.

[56] D. Dreyer, R. Harper, M. M. T. Chakravarty, and G. Keller, "Modular Type Classes," in *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '07. 2007, pp. 63–70. doi: 10.1145/1190216.1190229.

[57] Microsoft, "Declaration Merging." [Online]. Available: https://www.typescriptlang.org/docs/handbook/declaration-merging.html

[58] T. Van Cutsem and M. S. Miller, "Proxies: Design Principles for Robust Object-oriented Intercession APIs," in *Proceedings of the 6th Symposium on Dynamic Languages*, in DLS '10. 2010, pp. 59–72. doi: 10.1145/1869631.1869638.

[59] R. Hettinger, "Descriptor HowTo Guide." [Online]. Available: https://docs.python.org/3/howto/descriptor.html

[60] P. Maes, "Concepts and Experiments in Computational Reflection," in *Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA*), 1987, pp. 147–155. doi: 10.1145/38765.38821.

[61] G. Bracha and D. M. Ungar, "Mirrors: design principles for meta-level facilities of object-oriented programming languages," in *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (*OOPSLA*), 2004, pp. 331–344. doi: 10.1145/1028976.1029004.

[62] R. Rouvoy and P. Merle, "Leveraging component-based software engineering with Fraclet," *Annals of Telecommunications*, vol. 64, no. 1–2, pp. 65–79, 2009, doi: 10.1007/s12243-008-0072-z.

[63] E. Burmako, "Scala Macros: Let Our Powers Combine!," in *Proceedings of the 4th Workshop on Scala*, in SCALA@ECOOP '13. 2013, pp. 3:1–3:10. doi: 10.1145/2489837.2489840.

[64] S. Raemaekers, A. van Deursen, and J. Visser, "Semantic Versioning and Impact of Breaking Changes in the Maven Repository," *Journal of Systems and Software*, vol. 129, pp. 140–158, 2017, doi: 10.1016/j.jss.2016.04.008.

[65] P. Lam, J. Dietrich, and D. J. Pearce, "Putting the Semantics into Semantic Versioning," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, in Onward! '20. 2020, pp. 157–179. doi: 10.1145/3426428.3426922.

[66] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Dependency Solving: A Separate Concern in Component Evolution Management," *Journal of Systems and Software*, vol. 85, no. 10, pp. 2228–2240, 2012, doi: 10.1016/j.jss.2012.02.018.

[67] L. Cardelli, "Structural Subtyping and the Notion of Power Type," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '88. 1988, pp. 70–79. doi: 10.1145/73560.73566.

[68] B. Meyer, “Applying "Design by Contract",” *Computer*, vol. 25, no. 10, pp. 40–51, 1992, doi: 10.1109/2.161279.

[69] B. C. Pierce, “Bounded Quantification is Undecidable,” *Information and Computation*, vol. 112, no. 1, pp. 131–165, 1994, doi: 10.1006/inco.1994.1055.

[70] J. I. Brachthäuser, P. Schuster, and K. Ostermann, “Effects as capabilities: effect handlers and lightweight effect polymorphism,” *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, 2020, doi: 10.1145/3428194.

[71] J. I. Brachthäuser, P. Schuster, E. Lee, and A. Boruch-Gruszecki, “Effects, capabilities, and boxes: from scope-based reasoning to type-based reasoning and back,” *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA1, 2022, doi: 10.1145/3527320.

[72] Effect, “Effect: A TypeScript library for building robust applications.” [Online]. Available: https://effect.website/

[73] C. Heunen, R. Kaarsgaard, and M. Karvonen, “Reversible Effects as Inverse Arrows,” in *Proceedings of the Thirty-Fourth Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXIV)*, in Electronic Notes in Theoretical Computer Science, vol. 341. 2018, pp. 179–199. doi: 10.1016/j.entcs.2018.11.009.

[74] D. Orchard, V.-B. Liepelt, and H. Eades III, “Quantitative program reasoning with graded modal types,” *Proc. ACM Program. Lang.*, vol. 3, no. ICFP, 2019, doi: 10.1145/3341714.

[75] R. Bianchini, F. Dagnino, P. Giannini, E. Zucca, and M. Servetto, “Coeffects for sharing and mutation,” *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, Oct. 2022, doi: 10.1145/3563319.

[76] R. Bianchini, F. Dagnino, P. Giannini, and E. Zucca, “A Java-like calculus with heterogeneous coeffects,” *Theoretical Computer Science*, vol. 971, p. 114063, 2023, doi: https://doi.org/10.1016/j.tcs.2023.114063.

[77] C. Torczon, E. Suárez Acevedo, S. Agrawal, J. Velez-Ginorio, and S. Weirich, “Effects and Coeffects in Call-by-Push-Value,” *Proc. ACM Program. Lang.*, vol. 8, no. OOPSLA2, Oct. 2024, doi: 10.1145/3689750.

[78] R. Hirschfeld, P. Costanza, and O. Nierstrasz, “Context-oriented Programming,” *Journal of Object Technology*, vol. 7, no. 3, pp. 125–151, 2008, doi: 10.5381/jot.2008.7.3.a4.

[79] P. Costanza and R. Hirschfeld, “Language constructs for context-oriented programming: an overview of ContextL,” in *Proceedings of the 2005 Symposium on Dynamic Languages (DLS '05)*, ACM, 2005, pp. 1–10. doi: 10.1145/1146841.1146842.

[80] G. Salvaneschi, C. Ghezzi, and M. Pradella, “Context-oriented programming: A software engineering perspective,” *Journal of Systems and Software*, vol. 85, no. 8, pp. 1801–1817, 2012, doi: 10.1016/j.jss.2012.03.024.

[81] G. Kiczales *et al.*, “Aspect-Oriented Programming,” in *ECOOP'97 — Object-Oriented Programming, 11th European Conference*, in Lecture Notes in Computer Science, vol. 1241. Springer, 1997, pp. 220–242. doi: 10.1007/BFb0053381.

[82] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, “An Overview of AspectJ,” in *ECOOP 2001 — Object-Oriented Programming, 15th European*

Conference, in Lecture Notes in Computer Science, vol. 2072. Springer, 2001, pp. 327–353. doi: 10.1007/3-540-45337-7_18.

[83] A. Popovici, T. Gross, and G. Alonso, "Dynamic Weaving for Aspect-Oriented Programming," in *Proceedings of the 1st International Conference on Aspect-Oriented Software Development* (*AOSD 2002*), ACM, 2002, pp. 141–147. doi: 10.1145/508386.508404.

[84] J. Bonér, "What Are the Key Issues for Commercial AOP Use: How Does AspectWerkz Address Them?," in *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development* (*AOSD 2004*), ACM, 2004, pp. 5–6. doi: 10.1145/976270.976273.

[85] M. Hicks, J. T. Moore, and S. Nettles, "Dynamic Software Updating," in *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, in PLDI '01. 2001, pp. 13–23. doi: 10.1145/378795.378798.

[86] G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "Mutatis Mutandis: Safe and Predictable Dynamic Software Updating," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, in POPL '05. 2005, pp. 183–194. doi: 10.1145/1040305.1040321.

[87] C. M. Hayden, K. Saur, E. K. Smith, and M. Hicks, "Kitsune: Efficient, General-Purpose Dynamic Software Updating for C," *ACM Trans. Program. Lang. Syst.*, vol. 36, no. 4, 2014, doi: 10.1145/2629460.

[88] M. Overeem, M. Spoor, and S. Jansen, "The Dark Side of Event Sourcing: Managing Data Conversion," in *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering*, in SANER '17. 2017, pp. 193–204. doi: 10.1109/SANER.2017.7884621.

[89] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA: Addison-Wesley, 1994.

[90] D. Leijen, "Algebraic Effect Handlers with Resources and Deep Finalization," technical report MSR-TR-2018-10, Apr. 2018.

[91] M. Fowler, "Event Sourcing." 2005.

[92] J. Lee, J. Ahn, and K. Yi, "React-tRace: A Semantics for Understanding React Hooks," *Proc. ACM Program. Lang.*, vol. 9, no. OOPSLA2, pp. 471–498, 2025, doi: 10.1145/3763067.

[93] N. Shavit and D. Touitou, "Software Transactional Memory," in *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, in PODC '95. 1995, pp. 204–213. doi: 10.1145/224964.224987.

[94] T. Harris, S. Marlow, S. Peyton Jones, and M. Herlihy, "Composable Memory Transactions," in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, in PPoPP '05. 2005, pp. 48–60. doi: 10.1145/1065944.1065952.

[95] M. Herlihy and J. E. B. Moss, "Transactional Memory: Architectural Support for Lock-Free Data Structures," in *Proceedings of the 20th Annual International Symposium on Computer Architecture*, in ISCA '93. 1993, pp. 289–300. doi: 10.1145/165123.165164.

[96] R. Landauer, "Irreversibility and Heat Generation in the Computing Process," *IBM Journal of Research and Development*, vol. 5, no. 3, pp. 183–191, 1961, doi: 10.1147/rd.53.0183.

[97] C. H. Bennett, "Logical Reversibility of Computation," *IBM Journal of Research and Development*, vol. 17, no. 6, pp. 525–532, 1973, doi: 10.1147/rd.176.0525.

[98] T. Yokoyama and R. Glück, "A Reversible Programming Language and its Invertible Self-Interpreter," in *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, in PEPM '07. 2007, pp. 144–153. doi: 10.1145/1244381.1244404.

[99] P. Wadler, "Linear Types Can Change the World!," p. , 1993.

[100] C. Walls, *Spring in Action*, 6th ed. Manning Publications, 2022.

[101] C. Escoffier, R. S. Hall, and P. Lalanda, "iPOJO: an Extensible Service-Oriented Component Framework," in *IEEE International Conference on Services Computing*, 2007, pp. 474–481. doi: 10.1109/SCC.2007.74.

[102] H. Cervantes and R. S. Hall, "Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model," in *Proceedings of the 26th International Conference on Software Engineering*, in ICSE '04. 2004, pp. 614–623. doi: 10.1109/ICSE.2004.1317483.

[103] C. Elliott and P. Hudak, "Functional Reactive Animation," in *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, in ICFP '97. 1997, pp. 263–273. doi: 10.1145/258948.258973.

[104] G. H. Cooper and S. Krishnamurthi, "Embedding Dynamic Dataflow in a Call-by-Value Language," in *Programming Languages and Systems (ESOP 2006)*, in Lecture Notes in Computer Science, vol. 3924. Springer, 2006, pp. 294–308. doi: 10.1007/11693024_20.

[105] I. Maier and M. Odersky, "Deprecating the Observer Pattern with Scala.React," technical report EPFL-REPORT-176887, 2012. [Online]. Available: https://infoscience.epfl.ch/record/176887

[106] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, W. De Meuter, and others, "A Survey on Reactive Programming," *ACM Comput. Surv.*, vol. 45, no. 4, 2013, doi: 10.1145/2501654.2501666.