RECQ: A Rigorous Architecture and Pattern Language for Reactive Event-Driven Microservices, Realized in Evento

Gabor Galazzo

Independent Researcher
gabor.galazzo@gmail.com

Abstract. Event-driven microservices scale and tolerate failure, but the freedom that makes them attractive — any service may call or react to any other — also lets their global behavior decay beyond comprehension. The Reactive Manifesto states *what* such a system should be, and cloud vendors and CQRS/Event-Sourcing frameworks supply building blocks and prescriptive advice; none, however, *enforces* a model that keeps the system analyzable. This paper makes three contributions: *RECQ*, an architecture and pattern language that admits exactly seven precisely-defined component roles, formally characterized as the well-formed cells of a trigger × state classification; *Evento*, an open-source Java framework, server and GUI that realizes RECQ by enforcing its constraints at runtime; and *analyzability by construction* — because the component model is closed and enforced, the application's interaction graph is recovered exactly from source, without the heuristic reconstruction unconstrained microservices demand. The constituent mechanisms are deliberately established ones; the contribution lies in their closed, enforced assembly and the analyzability that follows from it. A worked case study exercises the full component vocabulary, an anonymized census of a production manufacturing-execution system (235 kLOC, 206 components) shows the model carrying an industrial workload, and a qualitative comparison positions RECQ against cloud-vendor offerings and related frameworks; quantitative evaluation of quality, effort and performance-model accuracy is deferred to ongoing companion work.

Keywords: Software architecture · Reactive systems · Event-driven microservices · CQRS · Event Sourcing · Domain-Driven Design · Pattern language.

1 Introduction

The technological substrate of distributed systems has advanced far faster than the methodology used to build them. Cloud platforms make it trivial to spin up and connect computational units of any size, and the microservice and event-driven styles have become the default for systems that must scale and tolerate failure. Yet the very freedom that makes these styles attractive — any service

may call, or react to, any other — is also what lets a system decay into the *Big Ball of Mud* [20]: a tangle whose global behavior can no longer be reasoned about, debugged, or analyzed as a whole. The consequence is familiar: once a microservice estate grows past a certain size, no one can say with confidence which components a given request touches or how it will perform under load.

The Reactive Manifesto [7] and the Reactive Principles [6] articulate *what* a well-behaved distributed system should be — responsive, resilient, elastic and message-driven — and the Reactive Design Patterns [31] offer concrete solutions. What this guidance does not supply, and what neither the major cloud vendors nor the established CQRS/Event-Sourcing frameworks supply either (Sect. 2), is a way to *enforce* those qualities: they provide building blocks and prescriptive advice, leaving the discipline that keeps a system comprehensible to the developer. Tooling can observe a running system through tracing, but it cannot recover a faithful structural or performance model from code that imposes no model in the first place.

This paper argues that the missing ingredient is a *closed, enforced component model*. We present **RECQ** (Reactive, Event-Driven Commands and Queries), an architecture and pattern language that admits only a finite set of precisely-defined component types and a fixed set of communication rules, turning the qualitative goals of the reactive school into enforced structural mechanisms. Because the admissible components and their interactions are fixed in advance, a RECQ system is *analyzable by construction*: its structure and runtime flows can be read directly from the source, with none of the heuristic reconstruction that unconstrained microservices demand. We further present **Evento**, an open-source Java framework, server and GUI that realizes RECQ by treating the pattern constraints as invariants it enforces at runtime. None of the constituent mechanisms is new — snapshot-based event sourcing, single-active consumers and saga compensation all have established treatments (Sect. 2); what is new is their *assembly* under a closed, enforced model, and the exact static analyzability that assembly yields. RECQ was first outlined in a short paper [23] and a master's thesis [24]; the present paper gives the first full account of the pattern language, of its enforced realization, and of the analyzability that the enforcement makes possible.

Contributions. This paper makes three contributions:

(C1) **RECQ**, a pattern language — System, Communication and Component patterns — that constrains event-driven microservices to a closed component model derived from reactive foundations (Sect. 3), positioned against cloud-vendor offerings and existing frameworks (Sect. 2).
(C2) **Evento**, a realization that *enforces* the model rather than merely advising it, and from which an analyzable model of the application is extracted directly from code (Sect. 4).
(C3) a worked case study that exercises the full component vocabulary, an anonymized census of an industrial application providing evidence at production scale, and a qualitative comparison establishing analyzability-

by-construction as the property that distinguishes RECQ from its neighbors (Sects. 5–6).

The remainder of the paper is organized as follows. Section 2 sets out the reactive foundations and related work. Section 3 defines the RECQ pattern language and Section 4 its realization in Evento. Section 5 works through a case study; Section 6 discusses analyzability and compares RECQ with related approaches. Section 7 states limitations and Section 8 concludes.

2 Background and Related Work

2.1 Reactive foundations

Distributed systems trade the simplicity of a single address space for scalability and fault tolerance, and in doing so confront the tensions captured by the CAP theorem [8] and the attendant choice between ACID and BASE consistency. Decomposition into microservices makes individual units more tractable, but the freedom to let any service call any other readily degrades into the *Big Ball of Mud* [20]: a system whose global behavior can no longer be reasoned about. The question that motivates RECQ is therefore not *which* building blocks to use, but *what discipline* keeps a distributed system comprehensible as it grows.

The Reactive Manifesto [7] answers that question at the level of system qualities. It identifies four traits and, crucially, a dependency among them: a system should be *responsive* (the user-facing goal), which it sustains under load and failure by being *elastic* and *resilient*, both of which are made attainable by a *message-driven* foundation of asynchronous, location-transparent communication. The Manifesto states the *what*; the Reactive Principles [6] refine it into design-level guidance — isolation, single responsibility, sharing nothing, embracing asynchronous message-passing — and the catalog of Reactive Design Patterns [31] supplies concrete, reusable solutions such as supervision, sharding and event-sourced state.

We take these reactive foundations as the *base point* for the architecture rather than as an after-the-fact checklist. The recurring building blocks of event-driven microservices — Domain-Driven Design [17], CQRS [46], Event Sourcing [22], the Saga [40,41], and, conceptually, the actor model [28] — are precisely the material from which reactive systems are built. What has been missing is a way to assemble them that ties each reactive trait to a specific, *enforced* mechanism instead of merely permitting it. Section 3 develops RECQ as that assembly: a closed component model in which each reactive trait is discharged by such a mechanism.

2.2 Industrial practice: cloud-vendor offerings

All three major cloud platforms promote event-driven, CQRS and Event Sourcing styles, but they do so by supplying *building blocks* together with *prescriptive guidance*, not an enforced architecture. On AWS, CQRS and Event Sourcing are

documented as patterns to be assembled from EventBridge, Kinesis, Lambda and DynamoDB Streams, with the Saga pattern realized through Step Functions (orchestration) or EventBridge (choreography) [3,4]. Azure offers the richest catalog — Event Grid, Event Hubs, Service Bus, Durable Functions for saga orchestration, the Cosmos DB change feed as an event store, and the Dapr runtime [13] for actors, pub/sub and state — yet its Architecture Center frames CQRS and Event Sourcing explicitly as *design patterns*, cautioning that tooling "can't automatically generate CQRS code" and that combining the two "introduces significant complexity" and "requires a different design approach" [35,36]. Google Cloud is the most plumbing-oriented, framing event-driven systems as producer → router (Eventarc/Pub-Sub) → consumer, with Workflows for orchestration [27].

In every case the *discipline* — which component types may exist, what each may send or receive, how each scales — is left to the developer. The platforms provide observability through runtime tracing (e.g. distributed traces), but none derives a *structural or performance model* from an enforced component model. This is precisely the space RECQ targets.

2.3 Frameworks

The closest neighbor to Evento is the **Axon Framework** [5], which provides an annotation-driven DDD/CQRS/ES model (`@Aggregate`, `@CommandHandler`, `@EventSourcingHandler`, `@EventHandler`) and treats commands, queries and events as messages on a bus with location transparency. Axon is the prior art that comes nearest to a component model, but its annotation vocabulary is *open and unconstrained*: handlers may be declared on any class, a command handler may `apply` arbitrarily many events, and which messages a component may send is nowhere restricted — so the annotations describe entry points, not a component model. Static analysis can certainly recover substantial portions of an Axon application's structure — but only heuristically and incompletely, by the best-effort reconstruction discussed in Sect. 2.4, because nothing bounds what a handler may do. RECQ closes exactly this gap: the component set is finite, each type's capabilities are fixed, and the framework rejects what the model forbids — the difference that turns static extraction from heuristic into exact (Sect. 6.1). Axon offers no automatic derivation of a structural or performance model. **Eventuate** [18,41] similarly supplies aggregates, sagas and CQRS views as building blocks. **Lagom** and **Akka** take the actor route: Akka Persistence Typed has no DDD aggregate marker, an event-sourced actor merely gluing state, commands, handlers and events through an `EventSourcedBehavior`. The actor model [28] is in fact the conceptual ancestor of RECQ's isolated, location-transparent, message-passing components; RECQ can be read as a *typed, role-constrained specialization* of the actor idea — one point in the design space of actor variants surveyed by De Koster et al. [14] — in which the admissible roles and their interactions are fixed a priori.

Beyond the CQRS/ES frameworks, a wider family of systems likewise trades freedom for structure. **Orleans** [11] constrains the actor model to *virtual* actors

whose identity and lifecycle the runtime owns; workflow engines such as **Temporal** [43], **Camunda** [12] and **Azure Durable Functions** [9] confine orchestration logic to deterministic, replayable workflow code — a closure adopted, as in RECQ, to make execution recoverable, though there for fault tolerance rather than analysis; and the actor language **Rebeca** [42] restricts actors precisely so that systems become amenable to model checking. A further family expresses reactive constraints in the *type system*: ecosystems such as Kotlin coroutines/Flow and Scala ZIO make asynchronous dataflow explicit and safe within a program, but their guarantees end at the process boundary — they constrain code, not architecture, and yield no cross-service component model or interaction graph. Architecture description languages such as AADL, Acme and Wright [33,19,26,2] make structure explicit and formally analyzable — Wright, whose formalized connectors support behavioral analysis of the architecture, is in this sense the closest ancestor of analyzability by construction — but in a design artifact *beside* the code, whose correspondence to the implementation must be maintained by discipline. RECQ differs from each in its operating point: the constraints apply to mainstream application code (annotated Java components), are enforced at runtime, and are chosen so that the architecture is read from the code itself rather than from a model maintained alongside it.

2.4 Automatic analysis of software architectures

A long line of model-driven work derives performance models from *design artifacts*: PUMA4SOA, for example, transforms UML models annotated with MARTE into Layered Queueing Networks [1], and Pinciroli, Aleti and Trubiani build queueing-network models of seven microservice *design patterns*, "flexible in their input parameterization", to analyze their performance impact [38] — patterns whose quality trade-offs have themselves been studied empirically against industry practice [44]. These models are hand-parameterized and design-level, not extracted from the running application. A complementary strand instead works *from code*: Cerny and colleagues envision an extensible methodology that would statically reconstruct microservice source into a formal system model for verification [45], building on a broader effort to recover microservice architecture by static analysis [10] — itself the microservice instance of a long-standing architecture-recovery literature whose techniques, comparative evaluations show, disagree with each other and with ground truth on unconstrained code [25]. Such reconstruction is necessary precisely because conventional microservices are unconstrained, so the model must be recovered heuristically and incompletely.

RECQ inverts this trade-off: because the component model is *closed and enforced by the framework*, the structural model is not reconstructed but simply *read off* the code reliably, and runtime metrics attach to it directly — a property we develop fully in the companion performance-model work. The present paper establishes that enforced model; the contribution here is the architecture and its realization, not the performance analysis itself — the reader should expect

structural results in what follows, and no quantitative performance data, which that companion work owns.

3 The RECQ Pattern Language

Building on the reactive foundations of Sect. 2.1, RECQ (Reactive, Event-Driven Commands and Queries) is an architecture and pattern language whose aim is to turn the *qualitative* goals of the Reactive Manifesto into *structural*, enforced mechanisms. Where the Manifesto and its principles prescribe how a system should behave, RECQ prescribes how a system must be *built* so that the behavior follows by construction. Each reactive trait is discharged by a specific, enforced mechanism, as summarized in Table 1; in establishing them, RECQ simultaneously honors a broader body of long-standing design guidance — Command–Query Separation [21], Separation of Concerns [16], the Single Source of Truth principle [37], the SOLID principles [32], the Uniform Access Principle [34], Event Sourcing [22], Messaging [41] and Domain-Driven Design [17].

Table 1. How RECQ discharges each trait of the Reactive Manifesto [7] through an enforced mechanism.

Reactive trait	RECQ mechanism (enforced, not merely permitted)
Message-driven	all interaction flows through the Message Gateway (asynchronous, correlation-id) and the System State Store; components never address one another directly
Elastic	components are replicable; the command side (Aggregate, Service) and read side (Projection) scale independently, and the consistent consumers (Projector, Saga) scale by partitioning into contexts
Resilient	component Isolation and location transparency contain failure; the System State Store is the Single Source of Truth, so state is rebuildable by event replay; consumers advance with dead-event queues
Responsive	the externally observable goal the mechanisms above sustain; RECQ adds no separate mechanism but makes each component's response-time profile explicit in the Capability Table (Table 2), so responsiveness is designed-in rather than hoped-for

The contribution of RECQ lies not in any individual ingredient — each is well established — but in the decision to admit only a *closed* and precisely-constrained set of them. The language is expressed as three complementary patterns: the *System Pattern* (the architectural structure of a RECQ system), the *Communication Pattern* (the behavioral rules governing how elements interact), and the *Component Pattern* (the methodological definition of each component's responsibilities). Because the admissible component types are finite and their

interactions are fixed, a RECQ application is *analyzable by construction*: its structure and runtime flows can be recovered from source code without the heuristic reconstruction that unconstrained microservices demand (Sect. 2.4) — a property we develop in Sect. 6.

3.1 RECQ System Pattern

A RECQ system is built from exactly three kinds of module (Fig. 1):

Components self-contained units of software that implement the application logic, expressed as *actions*;
Message Gateway the module responsible for communication between components, in terms of requests and responses;
System State Store (SSS) the module responsible for persisting the state of the system as an ordered event log.

A RECQ system is therefore a set of computational units — the components — that communicate by exchanging messages. When a component's action changes the state of the system, that change is published as an *event* in the System State Store; components may also subscribe to state changes in the SSS in order to update their own internal state without an explicit message from another component.

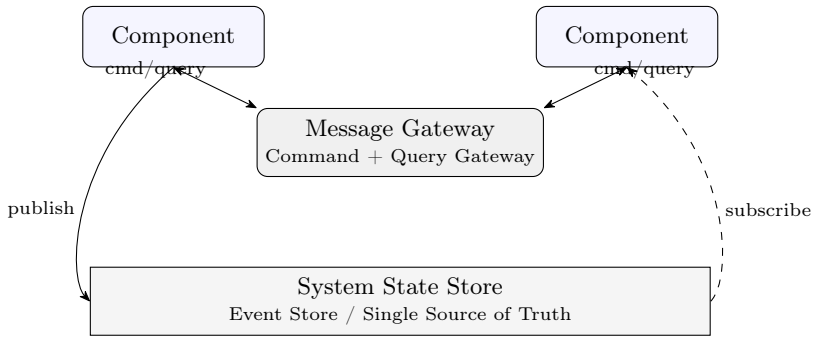

Fig. 1. The RECQ System Pattern. Components communicate only through the Message Gateway (commands and queries) and publish/subscribe state changes only through the SSS (events); no other communication channel is permitted.

Component properties. Every component must respect three properties: *Isolation* — it is decoupled from every other component in both time and space, making no assumption about the presence or the location of others (location transparency); *Containment* — cross-resource logic that genuinely belongs together must be contained within a single component (this is the Domain-Driven Design notion

of an aggregate: strongly-coupled concepts form one richer domain that contains them); *Delegation* — conversely, anything that is *not* a direct responsibility of the component must be delegated outside it, keeping each component to a minimal set of responsibilities. Isolation and delegation work together: to cause an effect elsewhere, a component does not call a specific collaborator but merely asks the system to perform the action, and the system routes the request to a responsible component.

Component capabilities. A component may implement its logic using only the following capabilities: it *may* receive messages addressed to it and reply to the sender; *may* hold persistent internal (local) state; *may* publish state-change events; *may* consume state-change events; and *may* send messages to other components. Finally, every component must be *replicable*: deploying multiple instances must never cause incorrect behavior. Replicability must not be conflated with *horizontal scalability*: the former concerns *availability* (instances act as failover), the latter concerns *throughput* (instances process work in parallel). All RECQ components are replicable. The stateless and per-instance components scale by adding instances directly; the two *consistent consumers* — the Projector and the Saga — admit a single *active* instance per *context*, and so scale not by parallel instances over one event stream (which would violate ordering) but by partitioning the stream into independent contexts that advance concurrently. Throughput thus grows with the number of contexts, at the price of deliberate context design — a point we make precise in Sect. 3.3.

Message Gateway. The Message Gateway is an abstract module that mediates all component-to-component communication. By interposing it, components satisfy the isolation property: they never address a specific component but instead issue a request as a message and await the response. The gateway realizes a publish/async-response pattern keyed by a *correlation id*, and exposes two submodules. The *Command Gateway* offers a single asynchronous `send` method that takes a Command message and returns either the event produced by the handling component or a failure. The *Query Gateway* offers a single asynchronous `query` method that takes a Query message and returns a `QueryResponse` carrying the requested data.

System State Store. The SSS persists the system state as the ordered sequence of state-change events; it is, to all effects, an Event Store and the Single Source of Truth (SSOT) of the system. It is the only module that knows the entire history of the state; every local representation held by a component is a derived reference. The SSS must expose two actions: *publishing* a domain event (appending a new state-change event, optionally associated with an aggregate), and *reading* the events of the system from a given instant onward (optionally filtered by aggregate and event type).

3.2 RECQ Communication Pattern

The Communication Pattern fixes *how* modules may interact, and admits only three channels.

Component to Component. Realized as asynchronous request/response through the Message Gateway. A component may send only two kinds of message: *Commands* — a request to change the system, whose response is only the confirmation (or failure) of the action — and *Queries* — a request that does not change the system and returns data (the *Views*). This restriction is exactly CQRS imposed at the communication level.

System State Store to Component. Realized as publish/subscribe, managed not by the Message Gateway but by the combination of the SSS and the *Consumer State Stores* used by Projectors and Sagas. A Consumer State Store persists how far a consumer has progressed through the event log, enabling retry logic and guaranteeing ordered, consistent advancement.

Component to System State Store. The only request a component may issue to the SSS is to publish a state-change event; this uses a synchronous protocol managed by the SSS itself.

No other channel. These three channels are exhaustive, and the exclusion is normative rather than a matter of scope: any side channel between components — a shared cache, a distributed lock, a database visible to more than one component, shared file storage — is *forbidden* by the Isolation property, because it couples components in a way the model cannot see and would silently invalidate the closed-world assumption on which analyzability rests (Sect. 6.1). Two shared stores are nonetheless sanctioned, because each is internal to a single logical unit and explicit in the model: the read model written by a Projector and read by its Projection (the pair jointly realizes the query model), and the Consumer State Store shared among the replicas of one consumer. State genuinely external to the system (a payment provider, an e-mail server) is not a channel between components: it is owned by exactly one Service, which wraps it behind the command interface.

3.3 RECQ Component Pattern

The Component Pattern defines seven component types with distinct, minimal semantics. Rather than asserting this minimality informally, we characterize the seven in Sect. 3.4 as the well-formed cells of a small classification, so that their number follows from the model rather than from experience. Each type is described by a *Capability Table* summarizing the messages its handlers may react to, the invocations it may issue, the kind of state it requires, and the consistency–responsiveness profile it declares; Table 2, at the close of this subsection, gives the consolidated view.

State type captures both consistency and the unit of scaling. **None**: the handler needs only its input message, so it is stateless and scales by free replication. **Instance**: state is keyed to a single domain object, so handlers run concurrently for distinct objects (locked per object) and scale with the number of objects. **Context**: state must remain consistent within a *context*, so at most one handler is active per context; such a component does not parallelize within a context but scales *across* contexts, which advance independently. Context is thus the deliberate partitioning unit by which the otherwise single-active consumers regain horizontal scalability — but partitioning is the consequence, not the purpose. Before it is a unit of scaling, a context is the unit of *temporal* consistency: every event published within it, by however many aggregates, is consumed by each consistent consumer in one total order, so declaring a context declares which aggregates' facts must be mutually ordered. Designing contexts therefore means drawing *two* boundaries at once — the spatial boundary of what state belongs together, and the temporal boundary of whose events must be folded in publication order.

We summarize each component's consistency and responsiveness as a compact two-letter *profile*. We deliberately do *not* write it with the letters of CAP [8]: the profile is read *per component* and is not an instance of Brewer's system-level impossibility result, but a design heuristic in the finer-grained spirit of PACELC, which already reasons about consistency and latency below the level of the whole system. **C**/**c** denote strong/weak consistency of the component's state. **R**/**r** denote strong/weak *responsiveness* — a reaction time bounded a priori, in the sense of the Reactive Manifesto [7], weak when the bound holds only for requests that do not contend on the same resource. A dash marks a dimension for which the profile makes no claim: in the consistency position because the component holds no domain state of its own, in the responsiveness position because ordered, single-active consumption admits no a-priori bound. Partition tolerance, the system being distributed, is presupposed for every component and is therefore not part of the profile. The Profile column of Table 2 is thus a per-component consistency–responsiveness reading, not a formal classification.

Invoker. A bridge from outside the system to the inside: it can only *invoke* other components, holds no internal state, and cannot receive messages from within the system. Typical invokers are a REST controller, a GUI controller, a CLI, or a cron trigger. Being stateless it scales freely, so its reaction time is bounded by replication (R); holding no domain state, it makes no consistency claim. Profile: –**R**.

Aggregate. Implements the *command model* of CQRS and embodies the DDD aggregate. It represents a single domain instance whose state is reconstructed by replaying, from the SSS, the events of that aggregate (identified within the incoming message). Its single *command handler* takes a Command and the aggregate state and returns either the resulting state-change event — then published to the SSS — or a failure. A command handler may *not* issue Queries: all information needed to decide whether a command is admissible must be in its

own state (queries are only eventually consistent and so cannot establish consistency); it *may* issue Commands, since commands are by definition consistent (e.g. delegating system-wide unique-id generation to a responsible aggregate). Because its state must be kept consistent (C), a lock prevents concurrent access to the same aggregate, so responsiveness is only weak (r): the a-priori bound holds for commands that do not contend on the same instance — yielding **Cr**.

Service. Also a command-model component, but unlike the aggregate it has no state, or its state is external to the system (e.g. sending an e-mail, or a payment delegated to an external provider). Its command handler takes only the command, may issue Commands but not Queries, and need not return an event. Consistency is not the system's concern (c), and its reaction time depends on the external interaction, so only a weak bound can be claimed (r) — weaker still if implemented in an ACID fashion: **cr**.

Projector. Implements the *write side* of the CQRS query model. Its *event handler* (`on`) consumes one event at a time from the SSS, in order, and builds the local read model, returning nothing. Within a single context a projector admits one *active* instance [41] to preserve ordering; horizontal throughput is gained not by replicating it over that context but by partitioning the event stream into independent contexts that are projected concurrently. It keeps, in a shared Consumer State Store (per projector type, using the Shared Database technique), the id of the last consumed event, guaranteeing consistency with the SSOT up to that point. An event handler may issue Queries (e.g. Federated Query) but never Commands. Profile: **C**–.

Projection. Implements the *read side* of the CQRS query model. Its *query handler* (`handle`) answers Query messages with data materialized by the projector. Being stateless — the local model is asserted consistent — it scales at will. It may issue Queries but not Commands. Profile: –**R**.

Saga. Implements the Saga pattern for distributed transactions. Like a projector it consumes events in order, but it acts back *on the system*: its *saga event handler* (`on`) takes the current saga state and an event and returns the new saga state, and may issue both Commands and Queries. Its state is like an aggregate's but persisted locally and shared across all active instances of a saga type, via a Saga Consumer State Store (Shared Database). It inherits the projector's consistency and scaling model: a single active instance per context, scaling across contexts. Profile: **C**–.

Observer. A stateless reaction to a single event: its event handler (`on`) depends on one event, returns nothing, but may issue Commands and Queries like a saga. An observer is in effect a saga that begins and ends with a single event, or a service invoked from inside the system by an event. Being stateless it scales easily, and — unlike sagas and projectors — it is not bound to ordered, consistent event consumption. Profile: –**R**.

Figure 2 places the seven types against the two mediators and shows, for each, the only messages it may exchange: who issues Commands and Queries through the Message Gateway, who publishes events to and consumes events from the SSS, and how the Projector and Projection share a read model. The closed set of arrows — and the absence of any other — is precisely what the Component and Communication patterns fix, and what Sect. 6.1 turns into analyzability by construction. Table 2 consolidates the seven descriptions above into the Capability Table.

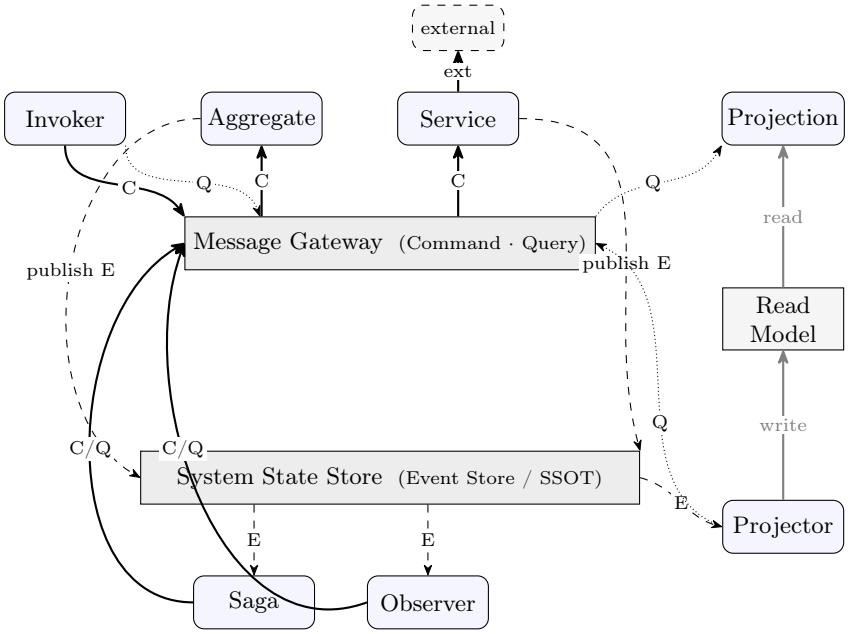

Fig. 2. The RECQ Component Pattern: the seven component types and the only interactions each may have (redrawn from the thesis). Solid = Command, dotted = Query, dashed = Event, grey = local read-model access; arrowheads show direction. Commands and Queries flow only through the Message Gateway and events only through the System State Store; the Aggregate always (and the Service optionally) publishes events, and the Projector materializes a read model that the Projection reads. No other edge is admissible.

3.4 A formal characterization of the component space

The seven components are not an arbitrary list but the well-formed inhabitants of a small classification. We characterize a component type by two primary

Table 2. Consolidated Capability Table of the seven RECQ component types (after thesis Tab. 9). C = Command, Q = Query, E = Event; Profile = the per-component consistency–responsiveness reading defined in the text (C/c = strong/weak consistency, R/r = strong/weak responsiveness, – = no claim). Each capability column maps to a concrete construct of Sect. 4.1: "recv. C" to `@AggregateCommandHandler` / `@CommandHandler`, "recv. Q" to `@QueryHandler`, "listen E" to `@EventHandler` / `@SagaEventHandler`, and the send columns to typed gateway calls.

	recv. C	recv. Q	listen E	send C	send Q	State	Profile
Invoker	no	no	no	yes	yes	None	–R
Aggregate	yes	no	no	yes	no	Instance	Cr
Service	yes	no	no	yes	no	None	cr
Projector	no	no	yes	no	yes	Context	C–
Projection	no	yes	no	no	yes	None	–R
Saga	no	no	yes	yes	yes	Context	C–
Observer	no	no	yes	yes	yes	None	–R

features — the *trigger* it consumes and the *state scope* it maintains:

$$T \in \{\mathsf{Ext},\ \mathsf{Cmd},\ \mathsf{Qry},\ \mathsf{Evt}\}, \qquad S \in \{\mathsf{None},\ \mathsf{Instance},\ \mathsf{Context}\}.$$

T records whether a component reacts to an external stimulus, a Command, a Query, or an Event from the SSS; S records the consistency-and-scaling scope of its state (defined above). The admissible types are then fixed by four well-formedness rules that restate the System and Communication patterns:

(W1) Single trigger. A component reacts to exactly one kind of trigger, so its identity is the pair $\langle T, S\rangle$, refined by W4.

(W2) CQRS. Only Cmd- and Evt-triggered handlers may change system state by emitting events; Qry-triggered handlers may not.

(W3) Consistent decisions. A state-changing decision may use only consistent state; hence a command handler may not issue Queries, and the read side (Qry) holds no authoritative state ($S = \mathsf{None}$).

(W4) Effect split. When more than one type inhabits a cell, the *effect* distinguishes them: an Evt/Context consumer that writes a read model is a Projector, while one that emits Commands to coordinate the system is a Saga. Formally, the discriminant is a third feature $e \in \{\mathsf{upd}, \mathsf{emit}\}$ — whether the handler's effect is to *update* a read model or to *emit* further messages — and a well-formed type takes exactly one value of e.

Because an Invoker is a pure boundary (the Delegation property), it carries no domain state, which excludes the two Ext cells with state. Enumerating $T \times S$ under these rules yields Table 3: every named component occupies exactly one class, four cells are *forbidden*, and two are *empty but admissible*. This enumeration lets us make minimality precise (Proposition 1).

Table 3. The RECQ component space $T \times S$. Named cells are the seven components; "—" marks a cell forbidden by the rule shown; "∘" an empty but admissible cell (a candidate type, discussed below).

Trigger T	$S =$ None	$S =$ Instance	$S =$ Context
Ext	Invoker	— (boundary)	— (boundary)
Cmd	Service	Aggregate	∘
Qry	Projection	— (W3)	— (W3)
Evt	Observer	∘	Projector / Saga (W4)

The two empty cells are instructive, and the first exposes a general property of the classification: *a context can be reified as an instance.* A Cmd/Context component would be a globally-serialized command authority — a system-wide singleton handling commands one at a time; in RECQ it is unnecessary, because an Aggregate with a fixed, well-known identifier already offers a serialized command target: the context, reified as one domain object. The same move is available at every granularity. A school class and its students are each domain objects, but the class is also, semantically, the context that manages its students: modeling the class as an aggregate that contains its students implements a context boundary with Instance machinery — and, in extremis, the entire system collapses into a single aggregate instance, one consistency boundary processing commands one at a time. The argument can be put in simulation terms: given any Context-scoped component, construct an Aggregate under a fixed, well-known identifier whose state is the entire context state and whose commands are the original component's triggers translated one-for-one; the aggregate consumes them in a single serial order, exactly as the single-active consumer would, so every behavior of the context-scoped component is reproduced. What the construction loses is not behavior but the granularity at which the runtime may parallelize. Instance therefore subsumes Context in expressive power; what the classification distinguishes is granularity and scaling, not expressiveness — Context names the case where consistency must span many instances *without* paying the serialization that folding them into one object implies. Promoting Cmd/Context to a first-class type would thus only forfeit the per-instance parallelism that defines the Aggregate. A Evt/Instance component would be a per-object event reactor; a Saga whose lifecycle spans a single event already covers the stateful case and an Observer the stateless one, so the cell adds no expressive power.

Proposition 1 (Minimality, relative to the feature space). *Under rules* (W1)–(W4) *and the boundary constraint, the feature space* $\langle T, S \rangle$ *admits exactly* ***seven*** *inhabited well-formed classes — the seven RECQ components — alongside four forbidden and two empty admissible cells. The set is* irredundant*: no two components share a class, and removing any one leaves an inhabited well-formed class unrealized.*

Proof. The space has $|T| \times |S| = 4 \times 3 = 12$ cells; we classify each. *(Boundary)* an Ext component is a pure boundary carrying no domain state, so Ext/Instance

and Ext/Context are forbidden, leaving Ext/None = Invoker. *(W3)* a Qry-triggered handler holds no authoritative state, so Qry/Instance and Qry/Context are forbidden, leaving Qry/None = Projection. *(Command side)* Cmd/None = Service and Cmd/Instance = Aggregate, while Cmd/Context is admissible but empty. *(Event side)* Evt/None = Observer, Evt/Instance is admissible but empty, and by (W4)'s effect split Evt/Context yields two types — Projector (e = upd) and Saga (e = emit). This exhausts the twelve cells (four forbidden, two empty-admissible, six inhabited classes, one of which splits), giving seven components. *Irredundancy:* each occupies a distinct $\langle T, S, e\rangle$ and is the sole realizer of it, so deleting any component empties a well-formed class. □

Closure is a design argument, not a corollary. We deliberately keep one further claim — that the two empty admissible cells warrant no eighth type — *outside* the proposition, because it does not follow from (W1)–(W4): the rules admit both cells, and a future revision of the model could legitimately name them. What rules them out is the pragmatic subsumption argument developed above. Cmd/Context is a globally-serialized command target already offered by a fixed-identifier Aggregate — a context reified as a single instance — and promoting it forfeits the Aggregate's per-instance parallelism; Evt/Instance, a per-object event reactor, is expressively covered by a single-event Saga (stateful) or an Observer (stateless). Both arguments are appeals to expressive equivalence and engineering economy, and we present them as such: the formal content of Proposition 1 is the enumeration and irredundancy; minimality at seven is the enumeration *plus* this design judgment.
Completeness is, in any case, relative to the feature space $\langle T, S, e\rangle$: a richer space could expose further roles, so Proposition 1 is a theorem about the model, not a universal taxonomy of reactive systems — a point we record as a limitation (Sect. 7).

3.5 RECQ as a specialization of the microservice pattern language

A RECQ architecture is a *specialization* of the event-driven microservice design cataloged by Richardson [41,39]: rather than offering each pattern as an independent choice, RECQ fixes one coherent set of choices and realizes them by construction. Table 4 makes the mapping explicit; pattern names follow Richardson's catalog, two of which (†) are named in the online pattern language [39] rather than the 2018 book.

Table 4. How RECQ realizes the microservice patterns of Richardson [41,39].

Microservice pattern	RECQ realization
Decompose by subdomain	the Aggregate component (DDD aggregate)
Self-contained Service†	component Isolation, Containment, Delegation
CQRS	command vs. query split, in both messages and components
Database per Service	the Projector / Projection pair
Shared Database†	SSS + Consumer State Store — *local (micro) only*
Aggregate (consistency)	the Aggregate component
Saga (consistency)	the Saga component
Event Sourcing / Domain event	intrinsic in the SSS and the Communication Pattern
Messaging	the Message Gateway + Communication Pattern
Remote Procedure Invocation	realized over messaging (no direct RPC)

† named in the online pattern language [39], not the 2018 book.

4 Realization in Evento

Evento is an open-source Java framework (and companion server and GUI) that realizes the RECQ pattern language. Its central design decision is that the pattern constraints of Sect. 3 are not documentation but *invariants the framework enforces*, which is what makes a RECQ application analyzable by construction (Sect. 6).

4.1 Components as annotated handlers

Each of the seven RECQ component types maps to a Java annotation in the package `com.evento.common.modeling.annotations.component` (`@Aggregate`, `@Service`, `@Projector`, `@Projection`, `@Observer`, `@Saga`, `@Invoker`), and each capability of Table 2 maps to a handler annotation (`@AggregateCommandHandler`, `@CommandHandler`, `@EventSourcingHandler`, `@EventHandler`, `@QueryHandler`, `@SagaEventHandler`). The annotations carry exactly the metadata the model requires: an `@Aggregate` declares a `snapshotFrequency` (the event-replay optimization), stateful consumers (`@Projector`, `@Observer`, `@Saga`) declare a `version`, and a `@SagaEventHandler` declares the `associationProperty` that correlates events to a saga instance. Listing 1.1 shows an aggregate: a command handler that validates intent and returns *one* domain event, and an event-sourcing handler that folds that event into state.

```
@Aggregate(snapshotFrequency = 100)
public class TodoListAggregate {

  @AggregateCommandHandler(init = true)
  TodoListCreatedEvent handle(CreateTodoListCommand cmd,
                              TodoListState state) {
    if (cmd.getListId() == null
        || cmd.getListId().isBlank())
      throw new IllegalArgumentException(
          "listId␣is␣required");
    return new TodoListCreatedEvent(
```

```
        cmd.getListId(), cmd.getName());
  }

  @EventSourcingHandler
  TodoListState on(TodoListCreatedEvent e,
                   TodoListState state) {
    if (state == null) state = new TodoListState();
    state.setName(e.getName());
    return state;
  }
}
```

Listing 1.1. An Evento `@Aggregate` (the case-study domain of Sect. 5): the command handler returns a single event; the event-sourcing handler rebuilds state.

4.2 Enforcing the constraints

At start-up an `EventoBundle` scans its packages and registers components by annotation; per-component *managers* (e.g. `AggregateManager`) build a message-name → handler routing table, so that a component can only ever be reached through a message of the kind its capabilities allow. Three enforcement points are worth highlighting, because they turn the Capability Table into executable rules:

- **One event per command.** An aggregate command handler is typed to return a single `DomainEvent`; the runtime rejects a `null` return and immediately event-sources the returned event before publishing it. The "at most one event" rule of the Aggregate is thus structural, not conventional.
- **Single active consumer.** Projectors and Sagas run inside dedicated engines (`ProjectorEngine`, `SagaEngine`) that acquire an exclusive `ConsumerLock` keyed by a `consumerId` (bundle / component / version / context); a second instance that fails to acquire the lock does not consume. Because the `consumerId` includes the *context*, the lock is per-context: replicas of the same consumer within a context provide failover, while different contexts hold different locks and advance in parallel — the context-based horizontal scaling of Sect. 3.3.
- **Ordered, exactly-once progress.** A `ConsumerStateStore` persists each consumer's checkpoint (and error history), while a `DedupeStore` provides an exactly-once gate per `(consumerId, eventId)` and a `DeadEventQueue` retains poison events. Observers, being stateless and order-independent, use the dedup gate but not the ordered checkpoint.

4.3 The platform

Components are packaged as *bundles* that communicate with the Evento Server over a binary protocol: messages are encoded as CBOR and framed with a 4-byte length prefix, with transparent chunking so there is no message-size limit. The transport is built on Netty with a virtual-thread business executor, optional TLS, and heartbeat-based liveness. Production consumer state stores are provided as JDBC implementations for PostgreSQL and MySQL.

The server is the realization of the System State Store, and it offers at the level of the event log the same trade-off between consistency and timeliness that the Capability Table draws, as the consistency–responsiveness profile, at the level of components (Table 2). The choice is between two event-store modes, selectable at deployment, that differ in *how event sequence numbers are generated*:

CPES (Consistent and Partitionable Event Store). A single auto-increment sequence, shared across all cluster nodes, assigns event identifiers. This guarantees a strict total order of events system-wide — strong consistency (**CP**) — at the cost of contention on the shared sequence, which can reduce availability under high load.

APES (Available and Partitionable Event Store). Sequence numbers are generated locally by a Snowflake-style algorithm, with no cross-node coordination, so the store stays available and high-throughput (**AP**). Because clock skew and network latency can perturb the global order, a configurable *fetch delay* (`evento.es.fetch.delay`, default 69 ms) lets the cluster settle before events are consumed: a consumer is served only events older than $now - delay$, yielding eventual rather than strict ordering. The delay must cover the cluster's clock skew plus its replication latency; no formal sufficiency bound is established, and sizing it for a given deployment is an open empirical question. The risk of undersizing is concrete and worth stating plainly: a consumer tracks a single last-consumed sequence number and never looks back, so an event that becomes visible only *after* the consumer's checkpoint has passed its timestamp — because lag or skew exceeded the delay — is permanently skipped by that consumer; no gap detection or reconciliation mechanism exists — and none is possible from the consumer's side, because Snowflake identifiers are sparse: a skipped event leaves no gap signature in the sequence a consumer observes. (Store-side reconciliation — a periodic audit comparing store counts against consumer checkpoints — is feasible and is planned framework work, but does not exist today.) The fetch delay is therefore a safety parameter, not a tuning knob, and its sizing reduces to two quantities a deployer can measure: the maximum inter-node clock offset, bounded by NTP/chrony discipline and exported by its monitoring, and the worst-case commit-visibility lag of the event store, available from the store's replication metrics; the delay must exceed their sum with margin. Common topologies bound both terms tightly — a single-node store has no replication lag at all, and an NTP-disciplined single-zone cluster keeps offsets in single-digit milliseconds — but deployments that cannot tolerate a missed event under any circumstance should choose CPES, whose shared sequence makes late visibility impossible by construction. The two modes thus fail in orthogonal ways: CPES under load risks *liveness* (contention on the shared sequence) but never ordering or loss; APES, if mis-sized, risks *completeness* (a skipped event) but never blocks. The industrial system of Sect. 6.2 runs CPES in production.

The two modes thus instantiate, for the event log as a whole, the consistency-versus-availability choice that pervades the architecture: CPES mirrors the consistent consumers (Projector, Saga), APES the freely-replicated ones.

The cost of enforcement. The enforcement points of Sect. 4.2 have concrete, bounded runtime costs, which we characterize mechanistically here; measured latencies and throughput are deliberately left to the companion empirical work. The `ConsumerLock` is a database advisory lock (`pg_try_advisory_lock` on PostgreSQL, `GET_LOCK` with zero timeout on MySQL) acquired once per event *batch*, not per event: the consumer engine attempts the lock at the start of each poll cycle, processes the fetched batch while holding it, and releases it with the batch — one extra database round-trip per batch, amortized across its events, plus one pinned connection per concurrently active consumer. Acquisition is non-blocking and fail-fast: a replica that loses the race skips the cycle and retries at the next poll, so contention costs no blocking or spinning. Failover follows directly from the lock's session scoping: if the holding instance dies, its database connection closes and the lock is released automatically, so a standby replica's next acquisition succeeds — no coordinator and no rebalancing protocol are involved. The one-event-per-command rule costs nothing in steady state: it is a property of the handler's signature, checked at registration, plus a `null` check at dispatch. Likewise the bytecode scan of Sect. 4.4 runs at registration time only and contributes nothing to message processing.

Scaling caveats. Two bottlenecks deserve explicit mention. First, under CPES the shared auto-increment sequence is a genuine serialization point: every published event contends on it, so a write-heavy deployment will saturate the sequence before it saturates the nodes. The architectural responses are to adopt APES wherever eventual ordering suffices, or — since the consistent consumers are per-context and never observe cross-context order anyway — to shard the sequence by context, preserving exactly the order the model relies on while removing the global contention point. Second, context partitioning shifts the load-balancing burden onto *context design*: each context is served by a single active Projector or Saga instance, so a *hot* context — one that concentrates a disproportionate share of the event stream — caps the throughput of its consumers regardless of how many contexts exist, and skewed partitions waste the parallelism that contexts are meant to provide. Choosing context keys whose traffic is balanced is therefore part of the modeling task, a limitation we return to in Sect. 7. Three observations bound the problem. First, the bottleneck is confined to the hot context's consistent-consumption lane: Projections replicate freely and Aggregates shard per instance, so a hot context throttles its own Projector and Saga progress, not the system's reads or its command side. Second, the standard mitigations of partitioned systems apply: split the hot context into finer contexts along a domain boundary or, when no natural subdivision exists, assign synthetic context keys by hashing [29,15] — at the price of forfeiting cross-key ordering within the former context, which is precisely the trade that deliberate context design makes explicit. Third, contexts are static strings: rebalancing means re-

deploying consumers under a revised context map, and no dynamic rebalancing protocol exists. When a single context genuinely cannot be subdivided, its consistent consumers face the vertical limits of one active instance — the model's honest ceiling.

4.4 From code to an analyzable model

Because components are discovered by annotation, the framework can also *describe* the application without running it. Algorithm 1 summarizes the construction Evento performs. *Discovery* reuses the same annotation scan as start-up (`EventoBundle`): each component's handlers are registered, and the message-name $\rightarrow$ handler map of Sect. 4.2 doubles as the routing table *tgt* that fixes, for every message type, its responsible handler. The *emit set* of a handler is then recovered statically: its declared return type is the event (or command) it publishes, and an ASM-based scanner (`AsmInvocationScanner`) abstractly interprets the handler's bytecode, recording the payload type at every Command or Query gateway call (`send`, `query`) and following calls by breadth-first traversal so that helper methods are included. The traversal is deliberately bounded to the handler's *declaring class*: private helpers and lambdas defined in the same class are followed transitively (to any depth), but the scanner does not descend into other classes. A handler that delegates its gateway calls to an injected collaborator, sends a payload typed as the abstract `Command` or `Query` base rather than a concrete subtype, or dispatches reflectively therefore yields an *under-approximated* emit set. We record this confinement of emit logic to the declaring class as an explicit assumption below. Unlike the other closed-world assumptions it cannot be made impossible — but its violation is *detected*: at registration the framework runs a complementary confinement check that sweeps every class in the scanned packages and flags (i) any gateway call site in a class that is not a registered component, and (ii) any handler gateway call whose payload type cannot be resolved to a concrete subtype. Findings are reported as warnings by default; under a strict setting, registration fails. The silent residue thus narrows to reflective dispatch, which no static check can see. Finally the server joins emit sets to triggers (`HandlerGraphUtils`): an edge runs from a handler to the handler registered for each payload it emits. The server exposes the result as a payload and component *catalog* and, with collected handler metrics, as a *flows* endpoint (`FlowsController`) that returns a performance model the GUI renders as interaction trees.

Closed-world assumptions. The extraction just described is static analysis, not mere declaration reading: Phase 2 recovers each handler's emit set from its bytecode. In general Java such an analysis is defeated by well-known escape hatches — reflective dispatch, message types constructed or chosen at runtime, polymorphic payloads resolved by dynamic dispatch, plugins loaded after start-up, and generated code that the scanner never sees. Algorithm 1 is exact only under a *closed-world dispatch* assumption: every message is sent through the typed gateway methods with a payload whose concrete type is fixed at the call site, every

Algorithm 1: Static construction of the interaction graph $G(A)$, as realized by Evento.

Input: application packages P
Output: interaction graph $G = (V, E)$
$V \leftarrow \emptyset$; $E \leftarrow \emptyset$; $tgt \leftarrow \emptyset$
`// Phase 1 - discovery + routing table (EventoBundle.start)`
foreach *component class c annotated* `@Aggregate`, . . . ,`@Invoker` *in* P **do**
  foreach *handler method h of c (identified by its handler annotation)* **do**
    $V \leftarrow V \cup \{h\}$; $trig(h) \leftarrow$ the message type h consumes
    if h *receives that message* **then** $tgt[trig(h)] \leftarrow h$
`// Phase 2 - emit set per handler (AsmInvocationScanner)`
foreach $h \in V$ **do**
  $emit(h) \leftarrow \{returnType(h)\} \cup \textsc{AsmScan}(h)$
`// Phase 3 - edges (HandlerGraphUtils)`
foreach $h \in V$ ***and*** $m \in emit(h)$ ***with*** $tgt[m]$ *defined* **do**
  $E \leftarrow E \cup \{(h,\ tgt[m],\ m)\}$
return (V, E)

handler is registered by its annotation (there is no programmatic registration API), each handler's emit logic is transitively confined to its declaring class (the bound of the bytecode traversal above), and the deployed bundles are exactly those scanned. Evento's design *closes* most of these hatches rather than hoping they stay shut: the typed gateways are the only transport exposed to handler code (the underlying server connection is internal to the bundle runtime and never injected into components), handlers are discovered exclusively by annotation scan, an aggregate handler's emitted event is its declared return type, and bundles are fixed at deployment. The confinement assumption is the exception — it is not made impossible, but it is audited: the registration-time check above detects violations and, in strict mode, rejects them (Sect. 6.1 returns to this point). Under this closed world the extraction is exact; outside it — in unconstrained microservice code — the same information could be recovered only through the heuristic reconstruction surveyed in Sect. 2.4. These assumptions are exactly the hypotheses under which Proposition 2 holds. The resulting model underpins the companion performance-analysis work and, in the present paper, constitutes the concrete evidence — formalized in Proposition 2 — that RECQ delivers analyzability by construction.

Artifact availability. Evento is open source, dual-licensed AGPL-3.0 / commercial, and distributed through Maven Central and Docker Hub; the repository carries continuous integration, CodeQL analysis and an OpenSSF Scorecard. The version described in this paper, including the confinement check of Sect. 4.4, is archived as the signed GitHub release `v2.1.0`[1]; its release artifacts carry

[1] `https://github.com/EventoFramework/evento-framework/releases/tag/v2.1.0`

a Sigstore keyless signature and a SLSA build-provenance attestation, so the published binaries are verifiably built from the tagged source. The case-study application of Sect. 5 derives from the framework's public tutorial and is available with it. Because the industrial system of Sect. 6.2 is not publicly inspectable, we state what a reader *can* verify independently from the public artifacts alone: (i) the enforcement points of Sect. 4.2 are exercised by the tutorial application (single-event returns, consumer locking, dedup); (ii) the interaction graph of Fig. 3 is reproducible by running the case study and querying the platform's flows endpoint; (iii) the confinement check of Sect. 4.4, including its strict mode, is demonstrable by adding a gateway call to a non-component class; (iv) the seven component annotations and their capability surface are inspectable in the framework source; and (v) the release signature and provenance are verifiable with standard `cosign`/attestation tooling.

5 Worked Case Study: a Collaborative To-Do List

To show how the pattern language guides design from requirements to running code, we model a small but non-trivial domain: a collaborative to-do list, drawn from the publicly documented Evento tutorial. The example is deliberately neutral and self-contained; it exercises every one of the seven component types and yields the interaction graph of Fig. 3.

5.1 Requirements and intent

Users create to-do lists and add, check and remove items; lists may be queried for display; an item check should trigger a side effect outside the domain (for example, notifying an external service); and completing a list must be archived in an external system under a guaranteed, ordered, compensatable protocol. Following Domain-Driven Design, we first capture *intent* as commands and the resulting facts as events, rather than modeling tables. Table 5 summarizes the mapping from user actions to commands, the events they produce, and the component responsible; the rows exercise every one of the seven component types.

Table 5. From user intent to RECQ components for the to-do list domain. The example exercises all seven component types.

User action	Command	Event	Component
Enter the system	(HTTP request)	—	Invoker
Create a list	`CreateTodoList`	`TodoListCreated`	Aggregate
Add an item	`AddTodoItem`	`TodoItemAdded`	Aggregate
Check an item	`CheckTodoItem`	`TodoItemChecked`	Aggregate
Complete a list	`CompleteTodoList`	`TodoListCompleted`	Observer / Aggregate
Build the view	—	(reacts to events)	Projector
View lists	`FindTodoLists` (query)	—	Projection
Notify on item check	`Notify`	(reacts to `TodoItemChecked`)	Observer / Service
Archive on completion	`ArchiveList`, `MarkListArchived`	`ListArchived`	Saga / Service

5.2 Command side: the aggregate

The consistency boundary is a single to-do list, so the domain has one aggregate, `TodoListAggregate`. Each command handler validates the command against the aggregate's replayed state — a list must exist before items are added, an item cannot be checked twice — and emits exactly one event, which the framework event-sources back into the aggregate state (Sect. 4.2). No query is issued from the command side, in keeping with the Aggregate's capabilities (Table 2): every decision is taken from locally-consistent state. One operation resists the single-event discipline — checking the final open item also completes the list — and we defer its resolution to Sect. 5.6.

5.3 Query side: projector and projection

Reads are served by a separate model. A single `TodoListProjector` consumes the domain events in order and materialises a read-optimised view (one active instance, as required for the consistent write side of the query model), while a `TodoListProjection` answers queries such as `FindTodoLists` from that view and scales freely with read demand. The split is exactly CQRS, and it is where the *elastic* trait of Table 1 is realized: query load is absorbed by replicating the stateless projection without touching the command side.

5.4 Boundaries and cross-domain reactions

External access enters through an `Invoker` — typically a REST controller — that maps HTTP requests onto commands and queries and represents the system's traffic source. The simplest cross-domain reaction illustrates *delegation*: the to-do list does not itself send notifications. Instead a notification `Observer` reacts to the `TodoItemChecked` event and issues a `Notify` command to a `Service` that owns the external interaction. Because the reaction depends on a *single* event, keeps *no* state, and need not be ordered, the Observer (stateless, profile –**R**) is the right role; nothing more is warranted.

5.5 Cross-domain consistency: a saga

A second reaction is not so simple. When a list is completed — the aggregate emits `TodoListCompleted` in response to a `CompleteTodoList` command, issued once its final item is checked (Sect. 5.6) — workspace policy requires the list to be archived in an *external* archive system, and the list marked archived *only after* the archive confirms; if the archival fails, the outcome must be recorded so the failure can be handled. This is a distributed transaction spanning the `TodoList` aggregate and an external domain, and it spans *several* events, so neither an Observer nor a single command suffices. It is the role of a `Saga`.

The `ListArchivalSaga` is born by `TodoListCompleted` and keeps its own *transaction state* (consistent, single-active per context — here the list — so profile **C**–). On birth it issues an `ArchiveList` command to the archive `Service`;

on the resulting `ListArchived` event it issues a `MarkListArchived` command back to the aggregate and reaches its terminal state. Should the service report failure, the saga issues a compensating command that records the failed outcome rather than leaving the list in limbo — the structured failure handling that distinguishes a Saga from an Observer. The saga thus coordinates two domains into one consistent outcome without either component calling the other directly: every step is a Command through the Message Gateway, and every trigger an Event from the System State Store.

5.6 Friction: when one command seems to need many events

The Aggregate's one-event-per-command rule (Sect. 4.2) is the constraint practitioners push against first, and the to-do domain itself runs into it: checking the *final* open item also completes the list, so the natural modeling of `CheckTodoItem` emits *two* events — `TodoItemChecked` and `TodoListCompleted`. Evento rejects this, and not only the double emission: even an either/or handler that sometimes returns one event and sometimes the other is inexpressible, because a handler declares a *single concrete* return type — the very feature that makes its emit set statically readable (Sect. 4.4). RECQ absorbs the case along the seam it makes explicit: *whose fact is each event?* That the check left no item open is a facet of the check itself, known to the aggregate at decision time, so it folds into one richer event: `TodoItemChecked` carries a `listNowComplete` flag. Completion is then a separate decision, taken through the components designed for reactions: a completion `Observer` reacts to `TodoItemChecked` and, when the flag is set, issues a `CompleteTodoList` command; the aggregate validates it against its consistent state and emits `TodoListCompleted` — again one command, one event (Fig. 3).

The same two moves scale to the canonical case from event-sourced practice: placing an order is expected to emit `OrderPlaced`, `InventoryReserved` and `LoyaltyPointsAwarded` from a single `PlaceOrder` command. Facts that belong to the deciding aggregate are facets of one business fact and fold into a single richer event — `OrderPlaced` carrying the order lines and the quantities to reserve. Facts that belong to *other* aggregates (inventory, loyalty) cross consistency boundaries, and no single command handler — in RECQ or in any event-sourced system — can decide them atomically anyway. They are delegated to a Saga, exactly as in Sect. 5.5: born by `OrderPlaced`, it issues `ReserveInventory` to the inventory aggregate and, on `InventoryReserved`, `AwardLoyaltyPoints` to the loyalty aggregate, each step again one command, one event, with compensation if a step fails. The rule thus relocates composite behavior rather than forbidding it: single-responsibility decisions stay in aggregates, reactions and cross-boundary coordination move to the components designed for them — and every emitted event remains statically attributable to its handler, the property Sect. 7 argues is what the constraint buys.

Set against the multi-event idiom of other stacks, the rule reshapes style as much as structure. An Axon aggregate may `apply` several events from one command handler [5], which reads naturally but leaves the emit set implicit in

the handler's control flow; an Akka `EventSourcedBehavior` may persist arbitrary event sequences. RECQ channels the same intent into different idioms: it encourages *intention-revealing composite events* — one business fact with a richer payload — over an event-per-side-effect spray, and *explicit delegation* — an Observer or Saga that visibly owns the reaction — over multi-event invariants hidden inside one handler. As for coverage, we claim the two moves cover the cases *encountered*: every multi-event intent met so far, in this case study and in the industrial system of Sect. 6.2 — whose codebase resolves the pattern twenty-five times by composite event and four times by saga delegation — decomposes into facets of one fact (fold into a richer event) and reactions or cross-boundary facts (delegate). We do not claim universal coverage: a domain could present a command whose natural outcome resists both moves, and encountering one would count as evidence against the model's expressiveness; none has appeared in our use to date. The strongest candidate attack we know is the *batch import*: a single `ImportOrders` command that must create N independent aggregate instances — N facts about N distinct consistency boundaries, with no one richer event that naturally carries them. The model's answer is that the import is itself a domain fact: an `Import` aggregate accepts the command and emits a single `ImportAccepted` event carrying the rows, and a saga born by that event issues one `CreateOrder` per row — each, again, one command, one event. The moves do cover the case, but not for free: the fan-out is sequentialized through a coordinator, and the intermediate state (some orders created, others pending) is visible while it runs. We prefer stating that cost to claiming the rule is frictionless.

5.7 The resulting flow

Because every interaction is mediated by the Message Gateway or the System State Store, the messages a component can send and receive are fixed by its type, and so the application's whole interaction graph — commands from the invoker to the aggregate; events from the aggregate to the projector, observers and saga; the notification observer's and saga's commands to their services; the completion observer's and the saga's commands back to the aggregate; and queries to the projection — is determined by the components' declared capabilities alone. Figure 3 shows that graph as Evento extracts it directly from the code (Sect. 4.4); no separate model was drawn, and the same artifact that documents the design is the one the framework analyzes. Section 6.1 makes precise why this recovery is always possible for a RECQ application.

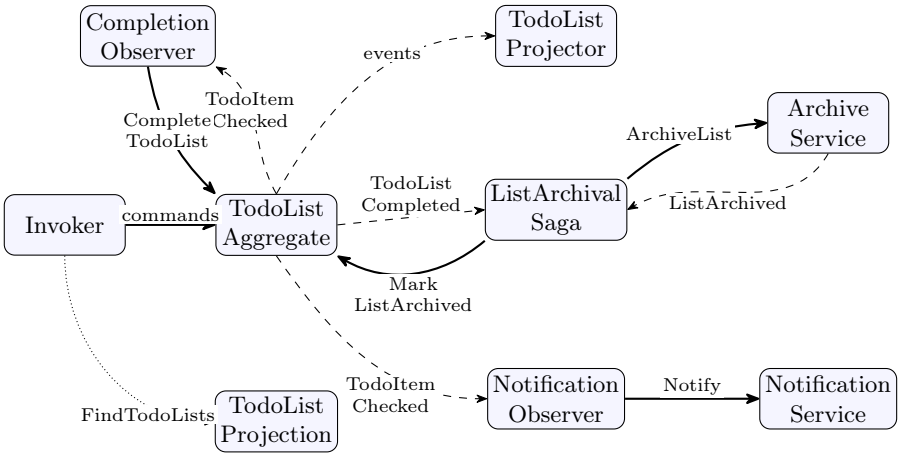

Fig. 3. The interaction graph of the to-do list, recovered from source by Evento — the union of the application's flows, not one invocation. Nodes are handlers, edges are messages: solid = Command, dotted = Query, dashed = Event. The completion observer's and the saga's commands back to the aggregate are the graph's only cycles. Unlike Fig. 2, which is a hand-redrawn schema of the *model*, this graph is an artifact of the framework: generated by the extraction of Sect. 4.4, not hand-drawn or manually maintained.

6 Discussion

6.1 Analyzability by construction

The recurring theme of this paper is that RECQ's value is not any single pattern it adopts but the consequence of adopting a *closed* set of them. When the admissible component types are finite and their communication is restricted to the Message Gateway and the System State Store, the dependency structure of an application is no longer emergent: it is determined by the components' declared capabilities (Table 2). This is what makes the structural and flow models of Sect. 4.4 extractable directly from source. The case study illustrates the point concretely — the interaction graph of Fig. 3 was generated, not drawn — and it explains why the same extraction is hard for conventional microservices: there, the model must be reconstructed heuristically because nothing in the code constrains what a service may do [45]. RECQ pays for this analyzability with expressive constraint; whether that trade is worthwhile in practice is the empirical question we defer to future work.

A formal statement. We make the claim precise. Model a RECQ application A as a finite set of *handlers*; each handler h belongs to a component of one of the seven types (Sect. 3.3) and, by that type's row in Table 2, declares a single *trigger* message type $trig(h)$ that it consumes and a finite set $emit(h)$ of message types it may emit (the Commands and Queries it sends and the Events

it publishes). Messages are routed *by type*: the Message Gateway forwards a Command or Query to the handler registered for that type, and the System State Store delivers a published Event to exactly the handlers that subscribe to it (Sect. 4.2); write $tgt(m)$ for the handlers the start-up registry binds to message type m. Define the *interaction graph* $G(A) = (V, E)$ with V the handlers and

$$E = \{\, (h, h', m) \;:\; m \in emit(h),\ h' \in tgt(m) \,\}.$$

Proposition 2 (Analyzability by construction). *For every RECQ application A satisfying the closed-world dispatch assumptions of Sect. 4.4 — messages are dispatched only through the typed gateway interfaces, handlers are registered only by their annotations, every payload type is fixed at compile time, and each handler's emit logic is transitively confined to its declaring class — the interaction graph $G(A)$ is computable from the static handler declarations and handler bodies alone, without executing A; the computed graph is* sound *(every edge denotes a message exchange the model permits) and* complete *(every inter-component message that can occur at runtime is an edge of $G(A)$).*

Proof (sketch). By the Communication Pattern (Sect. 3.2) the only inter-component messages are Commands and Queries through the Message Gateway and Events through the System State Store; the System Pattern's Isolation and Delegation properties forbid any direct component-to-component channel, so no exchange occurs outside these mediators. Each message carries a static type; by rule W1 a handler consumes exactly one trigger type, and by its row in Table 2 the kinds it may emit are fixed, so $emit(h)$ is read from the handler's declarations (the annotations and dispatched message types recovered by the metadata builder and bytecode scan of Sect. 4.4). Routing is by type through the registry, hence $tgt(m)$ is a function of the registry, independent of any payload value. Therefore E is exactly the finite set above and is computed statically: *completeness* holds because any runtime message is one of the three mediated kinds and so already an edge — provided no message can be dispatched outside the mediators, a premise we examine below — and *soundness* because every edge is, by construction, an admissible message under the patterns. □

What the completeness direction rests on. The premise just flagged deserves to be made explicit: that every runtime message passes through the typed gateways is a property of the *Evento runtime*, not of the RECQ model, and we state it as a hypothesis rather than a proved theorem. The supporting argument is the shape of the API surface. Handler code never constructs or receives the underlying server connection: components obtain only the typed `CommandGateway` and `QueryGateway` references that the runtime injects, the transport layer is internal to the bundle runtime, and an aggregate does not publish its event at all — the runtime event-sources and publishes the handler's *return value* (Sect. 4.2). The gateways are therefore the only transport the programming model exposes. This is API-surface enforcement, not capability-security: a component determined to subvert the model — say, by opening its own connection to the cluster — lies outside any static guarantee, as it would in any framework short of a

verified runtime. Likewise, the confinement hypothesis (emit logic stays within the declaring class) is a stated bound of the analysis rather than a runtime invariant: a handler that delegates its gateway calls to an injected collaborator yields an *under-approximated* emit set (Sect. 4.4), degrading completeness for that handler while leaving soundness intact. The framework narrows this gap mechanically: the registration-time confinement check of Sect. 4.4 sweeps the scanned packages for gateway call sites outside component classes and for sends with statically unresolvable payload types, warning by default and rejecting registration under a strict setting. It is worth enumerating, in the main text, exactly what survives both the API-surface enforcement and the confinement check: reflective dispatch; native code reached through JNI; bytecode-instrumentation agents and `ClassLoader` manipulation, which can rewrite what the scanner saw; and a component that hand-builds its own transport connection to the cluster. These are the vectors a hostile or careless codebase retains, none has been observed in practice, and all require deliberate effort — there is no idiomatic Java path from a handler to the wire except the typed gateways. For the *benign* violations (a helper class issuing messages, an abstract-typed send), the industrial census of Sect. 6.2 offers an empirical reading of how often they occur in a production codebase that never enabled strict mode: not once. Read at the right altitude, then, Proposition 2 states: *under deliberately restrictive assumptions, static extraction is exact.* The assumptions are not fine print beneath a guarantee — they are the price the architecture knowingly pays, and the proposition's content is that this price buys exactness. An unconstrained codebase pays nothing and gets heuristics; a RECQ application pays in expressive freedom and gets a graph it can trust.

The construction is not hypothetical: Algorithm 1 is the procedure Evento runs at registration time. Its hypotheses — the closed-world dispatch assumptions made explicit in Sect. 4.4 — are, with the one exception of emit-logic confinement just discussed, discharged not by convention but by *enforcement* (Sect. 4.2): could a component compute a callee's identity from runtime data — as an unconstrained microservice may — *tgt* would cease to be a static function and the proposition would fail, which is precisely why reconstructing conventional microservice architectures is heuristic and incomplete [10,45]. Proposition 2 concerns the *structural* model; annotating $G(A)$ with runtime metrics to obtain a quantitative *performance* model is the subject of companion work, but the graph that work operates on is recovered exactly, by construction.

6.2 Beyond the worked example: an industrial application

The to-do list of Sect. 5 demonstrates the full vocabulary at tutorial scale; it does not show that the constraints survive contact with a production domain. For that we report, in anonymized form, on a proprietary manufacturing-execution system (MES/MOM) for discrete manufacturing, built on Evento by an industrial development team and operated in production. The author led the system's architecture, so this is *supervised* use of the model, not independent adoption

— a distinction we return to below — but the components were designed and written by developers who did not create RECQ.

At the time of writing the system comprises roughly 235 kLOC of Java in 2,084 classes across 16 modules. Its census exercises the component vocabulary at industrial scale: 206 RECQ components — 5 Aggregates, 41 Projectors, 59 Projections, 4 Sagas, 52 Services and 45 Invokers — bound to a message vocabulary of 288 Commands, 178 Events and 187 Queries through 823 handlers. Three observations from this deployment bear directly on claims made in this paper. First, *context partitioning is used in production*: the event stream is partitioned into seven geographic contexts, one per manufacturing plant, so the per-context single-active consumers of Sect. 3.3 advance in parallel across plants — the natural domain boundary supplies the partition, as Sect. 4.3 prescribes. Second, *the friction moves of Sect. 5.6 recur at scale*: twenty-five event classes carry composite-intent flags (the richer-event leg) and the four sagas delegate follow-on commands across more than forty call sites (the delegation leg); the two sanctioned resolutions were, in fact, the resolutions used. Third, *the confinement assumption held without enforcement*: a static census of the codebase finds 589 gateway call sites across 58 classes — every one inside an annotated component class, and not a single send typed as the abstract base. Zero violations of the confinement assumption of Sect. 4.4, in a codebase developed before the registration-time check existed and without strict mode, suggest the assumption captures the natural idiom of the programming model rather than a discipline developers must fight. The system is also a live instance of the hybrid perimeter of Sect. 7: its legacy ERP integrations are wrapped behind stateless Service components, so the interaction graph records exactly where the managed world ends — and those wrappers are, as the model predicts, precisely where the graph's completeness stops.

The Observer's absence deserves a word, since it is the one role the census does not exercise. In this domain the absence is a characteristic, not a gap: every reaction either updates a read model (a Projector), spans several events with transaction state (a Saga), or answers a query — no fire-and-forget reaction to a single event arose. We cannot rule out the alternative reading, that practitioners collapse the Observer into adjacent roles where it would fit; but the worked example of Sect. 5.4 shows the role carrying genuine weight where a domain does call for it.

What this evidence does and does not show should be stated plainly. It shows that the closed component model is expressive enough for a non-trivial industrial domain, that a development team other than the model's author can build under the constraints, and that context partitioning carries a real workload. It does not constitute independent adoption (the architecture was led by the model's author), the system is not publicly inspectable, and the census exercises six of the seven types (the Observer's absence is discussed above). We weigh these limits in Sect. 7.

6.3 Comparison with related approaches

Table 6 consolidates the comparison developed in Sect. 2. Across both the cloud platforms and the established frameworks the picture is the same: building blocks and guidance, a component model that is open or absent, and analysis confined to runtime tracing or, in Axon's case [5], monitoring rather than a derived analytical model. Nor is RECQ the first design to buy analyzability with constraint: typed and virtual actors, workflow engines and architecture description languages each fix part of the design space toward a similar end (Sect. 2.3). We therefore claim for RECQ neither the invention of CQRS, Event Sourcing or the actor model, nor a new analyzability paradigm, nor the idea of modeling microservice performance — which others do from design artifacts [38,1] — but a specific, previously unoccupied combination: CQRS/Event-Sourcing concepts under a *closed, enforced* component model, imposed on mainstream application code, that enables reliable static extraction of the interaction graph.

What the static graph buys over a runtime model. The contrast with Axon, the nearest neighbor, is sharpest on one concrete point, so we make it concrete. Axon Server can display an interaction model *observed* from a running system; the graph of Fig. 3 was *extracted* before any deployment, from bytecode alone. The difference is not cosmetic. First, coverage: a runtime model contains an edge only after some request has traversed it, so rarely-exercised paths — the completion observer's `CompleteTodoList` delegation, which fires only when the final open item is checked, or the saga's `MarkListArchived` leg, which fires only after an external archive confirms — appear late or never; both are present in the static graph regardless of traffic, and by Proposition 2 the *absence* of an edge is evidence of absence, where absence in a runtime model is merely silence. Second, timing: the static graph exists before first deployment, which is exactly when architectural review and performance prediction want it; a runtime model requires a running, instrumented system under representative load. None of this diminishes runtime observation — Evento, too, annotates the extracted graph with telemetry — but the division of labor differs: the closed model yields the structure without traffic, and observation then only fills in the numbers.

6.4 RECQ as a cloud-agnostic discipline

Although Evento is one concrete realization, RECQ is a model rather than a runtime. Its roles and rules map naturally onto the primitives the cloud vendors already provide: a Message Gateway onto EventBridge, Event Grid or Pub/Sub; the System State Store onto an event-store or change-feed; a Saga onto Step Functions or Durable Functions; a Projector onto a change-feed consumer. To make one mapping concrete: a Projector on Azure is an Event Hubs consumer pinned to a consumer group of size one per context (the single-active rule), checkpointing offsets to durable storage (the Consumer State Store) and materializing its read model into Cosmos DB; its RECQ obligations — ordered consumption, per-context exclusivity, idempotent replay — become concrete

Table 6. Qualitative comparison of RECQ/Evento with related frameworks and cloud-vendor offerings. "blocks" = provided as composable services or primitives but not enforced; "guidance" = documented as a pattern to assemble.

	RECQ/ Evento	Axon	Eventuate	Akka/ Lagom	Cloud vendors (AWS/Azure/GCP)
Closed, constrained component model	yes	partial[a]	no	no	no
Model enforced (not just advised)	yes	partial	partial	partial	no (guidance)
CQRS / Event Sourcing	yes	yes	yes	blocks	blocks/guidance
Saga support	yes	yes	yes	yes	blocks
Auto structural (flow) model	yes	runtime	no	no	no (tracing)
Auto performance model from code	by design[†]	no	no	no	no

[a] Axon provides typed roles (`@Aggregate`, `@Saga`, handler annotations) but the vocabulary is open: components may combine capabilities freely and nothing bounds the admissible types (Sect. 2.3). "runtime" = a flow view observed from a running system (Axon Server monitoring), as opposed to Evento's static extraction before deployment. [†] the structural graph is recovered by construction (Prop. 2); its quantitative performance annotation is developed in companion work.

configuration choices (partition keys, checkpoint stores, lease containers) that the platform offers but does not impose. Read this way, RECQ contributes the *discipline* those platforms leave to the developer, and Evento shows that the discipline can be enforced and, with it, the analyzability recovered. The architecture is thus not in competition with managed event infrastructure but a principled way to use it. Nor is the model Java-specific: the seven roles and the communication rules are language-agnostic, and what Sect. 4.4 requires of a host platform is only annotation-discoverable handlers and inspectable compiled code — a substrate that other managed runtimes (e.g. .NET, with attributes and IL) provide equally well. Evento is one realization, not the boundary of the idea.

7 Limitations and Threats to Validity

Limitations of the architecture. RECQ's properties come from constraint, and constraint has costs — but the first apparent cost should be named for what it is. That the Projector and the Saga are single-active *per context* (Sect. 3.3) is not a restriction the architecture imposes on the domain; it is the form a domain requirement takes. A consistent consumer folds events into a local representation, and the folding order can change that representation drastically: when several aggregates publish facts about the same part of the domain, the order in which those facts are consumed *across aggregates* is itself a domain constraint — the consistency-side choice, in CAP terms — and serializing consumption per context honors it without inventing race conditions among substreams of the event log. A context, as Sect. 3.3 defines it, is the unit of temporal consistency before it is a unit of scaling. The industrial system of Sect. 6.2 makes the requirement concrete: a KPI projector folds events from many aggregates — eighteen event handlers feeding one view — into a per-plant production timeline that is correct

only if events fold in publication order, while plants' timelines are mutually independent; the context is the plant, chosen for exactly this temporal boundary, not for throughput. And when a reaction needs no such ordering, the model says so *by role*: the Observer consumes deduplicated but unordered, fire-and-forget — that is precisely why the role exists. The genuine cost, then, is not the single-active rule but the *design burden* it places on contexts, which must be drawn for two things at once: the temporal-consistency boundary they declare and the traffic balance they yield. A domain that resists a clean partition retains a throughput ceiling on its consistent read-model and distributed-transaction paths, and skewed or *hot* contexts can dominate throughput (Sect. 4.3). Three working heuristics help. Pick keys whose cardinality comfortably exceeds the parallelism the deployment needs; bound the expected traffic share of any single key at design time — a key expected to carry most of the stream is a context-design smell, not a tuning problem; and monitor per-context consumer lag in operation, the direct signal of skew. Natural domain boundaries (region, plant, tenant) should be preferred where they exist; synthetic keys by hashing [29,15] are the fallback, with the ordering trade-off of Sect. 4.3, and Kleppmann [30] surveys the wider partitioning-and-ordering design space. The Aggregate's single-event-per-command rule keeps the write side analyzable but forces some designs to spread an operation across several commands or to introduce a Saga (Sect. 5.6 works the canonical example). As in any CQRS/Event-Sourcing system, the read side is only eventually consistent with the write side, which the application must accommodate. Finally, the discipline imposes a learning curve: developers accustomed to issuing arbitrary calls must instead think in terms of the seven component roles and their capabilities. Tooling absorbs part of this cost — the GUI renders the component catalog and the extracted flows (Sect. 4.4), so a newcomer reads the system's actual structure instead of reverse-engineering it, and the seven roles give design discussion and code review a fixed, small vocabulary — and the supervised industrial experience of Sect. 6.2 is at least consistent with the constraints being learnable; measuring onboarding time is part of the comparative study in preparation.

Hybrid and legacy environments. The closed world of Sect. 4.4 is a property of the *managed perimeter*, and enterprise systems rarely live alone inside one: legacy services, shared caches, third-party APIs and semi-opaque middleware are facts of most migrations. RECQ's sanctioned answer is to wrap each unmanaged dependency in a stateless Service that owns the interaction (Sect. 3.2), and it is worth being precise about what analyzability then survives. Inside the perimeter nothing changes: the interaction graph remains exact for every RECQ-mediated exchange. At the boundary the analysis degrades *gracefully*: the wrapped dependency appears as the Service node that fronts it, so the graph records *that* and *where* the system leaves the managed world, but is silent about whatever the dependency does on its own — its internal call structure, its retries, its fan-out are invisible, and any performance annotation of those edges must come from measurement rather than construction. The failure mode to avoid is the *side channel*: if unmanaged code shares mutable state with components directly —

a shared cache or database written from outside the perimeter — the Isolation property is violated and the completeness of Proposition 2 no longer covers those flows; no static claim is made for them. The discipline in hybrid environments is therefore the discipline of the perimeter: every crossing is funneled through a Service, at the cost of writing and maintaining those wrappers.

Why one event per command. Since both the friction and the payoff of this rule recur throughout the paper, we state its justification explicitly. First, and decisively, the rule is what makes the emit set statically recoverable: a command handler with a single explicit return type lets the extraction of Sect. 4.4 read the emitted event *deterministically* from the handler's signature, whereas a collection-valued or polymorphic return would reintroduce exactly the uncertainty that defeats static reconstruction — the interaction graph of Proposition 2 could no longer be built reliably. Second, an operation that appears to need several events is usually announcing facets of a single business fact, which fold into one richer event carrying the relevant data (Sect. 5.6). Third, the rule is the Single Responsibility Principle applied to decisions: one handler decides one thing, and behavior that genuinely spans consistency boundaries is delegated to a Saga, which coordinates a sequence of single-event commands with structured compensation. The constraint thus trades a modeling idiom for an analytical property, with the Saga as the principled escape valve.

Threats to validity. We prove (Proposition 1) that the seven component types are exactly the inhabited, irredundant classes of the trigger $\times$ state classification of Sect. 3.4; closure at seven additionally rests on the explicitly informal subsumption argument given there. The result is, moreover, *relative to the chosen feature space* $\langle T, S, \text{effect}\rangle$: a richer space could expose further roles, so closure is a property of the model rather than an absolute claim. Likewise, Proposition 2 holds under the closed-world dispatch assumptions of Sect. 4.4: code that escapes them — reflection, runtime-constructed message types, plugin loading — is outside the model. Evento closes most of these hatches by construction and audits the remainder: violations of the emit-logic confinement are detected by the registration-time check of Sect. 4.4 (and rejected in its strict mode) rather than prevented, so escaping detection requires reflective dispatch; and the completeness direction of the proposition is conditional on the soundness of the runtime's enforcement itself, which we argue from the API surface but do not verify (Sect. 6.1). The per-component consistency–responsiveness profile (Sect. 3.3) is a design heuristic rather than a formal classification: it reads responsiveness as the bounded reaction time of the Reactive Manifesto, per component, and deliberately avoids the letters of CAP to prevent confusion with Brewer's system-level result. Our evaluation in this paper is a worked case study, a *qualitative* comparison, and the anonymized industrial census of Sect. 6.2; its external validity is bounded on both flanks. The case study is self-authored — its domain comes from the framework's own tutorial, so it cannot show that the constraints are workable for practitioners who did not design the model. The industrial system was built by a team beyond the author, which mitigates but does not remove the threat:

its architecture was led by the model's author, so no fully author-independent adoption of RECQ exists yet, and the system is not publicly inspectable. External validity therefore rests, for now, on *supervised* industrial use. A further open question is attribution: how much of the analyzability benefit comes from the *framework* and how much from the architectural *discipline* itself — i.e., what survives if RECQ is followed partially, or outside Evento's enforcement? This paper shows only that the enforced combination suffices; separating the two contributions would require a controlled comparison (RECQ-by-convention against RECQ-enforced) that we leave to the empirical line below. We make no quantitative claim here about software quality or development effort, nor about the accuracy of the extracted performance model. This scoping is a deliberate research-program decision, not an omission: the evaluative weight of the present paper is carried by the property it does establish formally and mechanically — analyzability by construction (Proposition 2, Algorithm 1, Fig. 3) — while the empirical questions it opens are the subjects of dedicated studies. A comparative study, in preparation, measures whether building under RECQ yields higher artifact quality in less development effort than conventional CQRS/Event-Sourcing stacks, using maintainability, defect rate and onboarding time as quality and effort proxies; a second line validates the accuracy of the performance models derived from the extracted graph and runtime telemetry against measured behavior (Sect. 8).

8 Conclusion and Roadmap

We have argued that the freedom of event-driven microservices, while the source of their appeal, is also what makes them hard to reason about, and that neither the reactive guidance nor the cloud vendors nor the established frameworks close this gap: they offer building blocks and advice, not an enforced model. RECQ responds with a closed, precisely-constrained component model — three patterns and seven component roles — that ties each qualitative goal of the Reactive Manifesto to an enforced structural mechanism, and Evento realizes it by enforcing those constraints in code. The defining consequence is *analyzability by construction*: because the model is fixed and enforced, the structure and the runtime flows of an application can be recovered directly from its source, as the case study and the Evento tooling demonstrate.

This property is the foundation for two lines of work we are pursuing. First, a comparative empirical study of whether building under RECQ's discipline yields artifacts of higher quality in less development effort than conventional CQRS/Event-Sourcing or plain microservices — the value question this paper deliberately leaves open — measuring maintainability, defect rate and onboarding time on comparable implementations of a common specification. Second, the automatic derivation of performance models from the extracted structure and runtime metrics, enabling bottleneck identification and capacity decisions without a separate modeling step, validated by comparing the models' predicted latency and throughput with measured behavior under load. Together with the

present contribution — the architecture and its realization — these chart a path from a disciplined way of building reactive systems to systems that can be understood, measured and tuned as a matter of course.

References

1. Alhaj, M., Petriu, D.C.: Approach for generating performance models from UML models of SOA systems. Proc. CASCON (2010)
2. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology **6**(3), 213–249 (1997). https://doi.org/10.1145/258077.258078
3. Amazon Web Services: Decompose monoliths into microservices by using CQRS and event sourcing. AWS Prescriptive Guidance, https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/decompose-monoliths-into-microservices-by-using-cqrs-and-event-sourcing.html, accessed June 2026
4. Amazon Web Services: Implement the serverless saga pattern by using AWS step functions. AWS Prescriptive Guidance, https://docs.aws.amazon.com/prescriptive-guidance/latest/patterns/implement-the-serverless-saga-pattern-by-using-aws-step-functions.html, accessed June 2026
5. AxonIQ: Axon framework. https://www.axoniq.io/products/axon-framework (2019), accessed June 2026
6. Bonér, J.: The reactive principles. https://www.reactiveprinciples.org/ (2022)
7. Bonér, J., Farley, D., Kuhn, R., Thompson, M.: The reactive manifesto. https://www.reactivemanifesto.org/ (2014)
8. Brewer, E.A.: Towards robust distributed systems (invited talk). Proc. ACM PODC (2000)
9. Burckhardt, S., Gillum, C., Justo, D., Kallas, K., McMahon, C., Meiklejohn, C.S.: Durable functions: Semantics for stateful serverless. Proc. ACM Program. Lang. **5**(OOPSLA) (2021). https://doi.org/10.1145/3485510
10. Bushong, V., Das, D., Cerny, T.: Reconstructing the holistic architecture of microservice systems using static analysis. In: Proc. 12th Int. Conf. on Cloud Computing and Services Science (CLOSER) (2022)
11. Bykov, S., Geller, A., Kliot, G., Larus, J.R., Pandya, R., Thelin, J.: Orleans: Cloud computing for everyone. In: Proc. 2nd ACM Symposium on Cloud Computing (SoCC) (2011). https://doi.org/10.1145/2038916.2038932
12. Camunda: Camunda: Universal process orchestration. https://camunda.com/, accessed June 2026
13. Dapr Authors (CNCF): Dapr — distributed application runtime. https://dapr.io/, accessed June 2026
14. De Koster, J., Van Cutsem, T., De Meuter, W.: 43 years of actors: A taxonomy of actor models and their key properties. In: Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016). pp. 31–40. ACM (2016). https://doi.org/10.1145/3001886.3001890
15. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07). pp. 205–220. ACM (2007). https://doi.org/10.1145/1294261.1294281

16. Dijkstra, E.W.: On the role of scientific thought. In: Selected Writings on Computing: A Personal Perspective, pp. 60–66. Springer (1982)
17. Evans, E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley (2004)
18. Eventuate Inc.: Eventuate. https://eventuate.io/ (2020), accessed June 2026
19. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language. Addison-Wesley (2012)
20. Foote, B., Yoder, J.: Big ball of mud. Pattern Languages of Program Design 4 (1997)
21. Fowler, M.: Command query separation. https://martinfowler.com/bliki/CommandQuerySeparation.html (2005)
22. Fowler, M.: Event sourcing. https://martinfowler.com/eaaDev/EventSourcing.html (2005)
23. Galazzo, G., Giordana, A., Franceschinis, G.: Evento: A performance oriented framework for cloud native reactive systems. In: QualITA'23, 19th National Workshop on Software Quality. Florence, Italy (2023)
24. Galazzo, G.: RECQ Patterns and Evento: Architectural Proposal and Java Framework for Reactive Systems. Master's thesis, Università degli Studi del Piemonte Orientale "Amedeo Avogadro" – DiSIT, Alessandria, Italy (2023)
25. Garcia, J., Ivkovic, I., Medvidovic, N.: A comparative analysis of software architecture recovery techniques. In: Proc. 28th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE) (2013). https://doi.org/10.1109/ASE.2013.6693106
26. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Foundations of Component-Based Systems, pp. 47–67. Cambridge University Press (2000)
27. Google Cloud: Event-driven architectures (eventarc). Google Cloud Documentation, https://cloud.google.com/eventarc/standard/docs/event-driven-architectures, accessed June 2026
28. Hewitt, C., Bishop, P., Steiger, R.: A universal modular actor formalism for artificial intelligence. In: Proc. 3rd Int. Joint Conf. on Artificial Intelligence (IJCAI) (1973)
29. Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: Proceedings of the 29th Annual ACM Symposium on Theory of Computing (STOC '97). pp. 654–663. ACM (1997). https://doi.org/10.1145/258533.258660
30. Kleppmann, M.: Designing Data-Intensive Applications. O'Reilly Media (2017)
31. Kuhn, R., Hanafee, B., Allen, J.: Reactive Design Patterns. Manning (2017)
32. Martin, R.C.: Design principles and design patterns (the solid principles). https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf (2000)
33. Medvidovic, N., Taylor, R.N.: A classification and comparison framework for software architecture description languages. IEEE Transactions on Software Engineering **26**(1), 70–93 (2000). https://doi.org/10.1109/32.825767
34. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, 2nd edn. (1997), uniform Access Principle
35. Microsoft: CQRS pattern. Azure Architecture Center, https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs, accessed June 2026
36. Microsoft: Event sourcing pattern. Azure Architecture Center, https://learn.microsoft.com/en-us/azure/architecture/patterns/event-sourcing, accessed June 2026

37. Pang, C., Szafron, D.: Single source of truth (ssot) for service oriented architecture (soa) (2014), proc. ICSOC
38. Pinciroli, R., Aleti, A., Trubiani, C.: Performance modeling and analysis of design patterns for microservice systems. In: IEEE Int. Conf. on Software Architecture (ICSA) (2023). https://doi.org/10.1109/ICSA56044.2023.00012
39. Richardson, C.: Microservices pattern language. https://microservices.io/patterns/, accessed June 2026; the continuously-updated online catalog, which names patterns (e.g. Self-contained Service, Shared Database) not all present in the 2018 book
40. Richardson, C.: Pattern: Saga. https://microservices.io/patterns/data/saga.html
41. Richardson, C.: Microservices Patterns: With Examples in Java. Manning (2018)
42. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundamenta Informaticae **63**(4), 385–410 (2004)
43. Temporal Technologies: Temporal: Durable execution platform. https://temporal.io/, accessed June 2026
44. Vale, G., Correia, F.F., Guerra, E.M., de Oliveira Rosa, T., Fritzsch, J., Bogner, J.: Designing microservice systems using patterns: An empirical study on quality trade-offs. In: IEEE Int. Conf. on Software Architecture (ICSA) (2022). https://doi.org/10.1109/ICSA53651.2022.00015
45. Wojtak, C., Gajewski, D., Cerny, T.: Vision: An extensible methodology for formal software verification in microservice systems. In: IEEE Int. Conf. on Software Architecture Companion (ICSA-C) (2025)
46. Young, G.: Cqrs documents. https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf (2010)