THE WORKSHOP

Event Storming an Architecture

Event Storming at its most precise. Take a Process Level wall and a team of developers; leave with aggregates, bounded contexts, and a design the room actually believes in. The session that turns 'we understand the flow' into 'we know where the code goes'.

2026-07-24
barkingiguana.com/writing/the-workshop-event-storming-an-architecture/

A facilitator's playbook — print, annotate, and keep on the table.

Contents

Third of three Event Storming posts, after Big Picture and Process Level. Brandolini calls it Software Design EventStorming; I call it Architecture because it's the session you run when the next question is "where does the code go?" It assumes you're comfortable with the Process Level palette and uses DDD vocabulary (aggregate, bounded context, invariant), which is why it sits after the DDD cluster.

At a glance

- *Who, for how long:* a facilitator with design instincts, four to eight developers and architects who'll implement the result, a domain expert who can veto unrealistic clusters, and a product owner to carry the output into the backlog. Plan 3h 15min inside a 3.5-hour block.
- *What you walk out with:* aggregates, bounded contexts, crossing events versus crossing commands, explicit policies and read models, and the anti-corruption layers around external systems, drawn on a wall the room committed to in front of each other.
- *When to reach for it:* a Process Level flow is clear and you're about to build or re-architect it, split a monolith, or design a new service into an existing landscape. Not for flows that aren't agreed yet (run Process Level first), domains half the room doesn't know, or rooms without the developers who'll implement the design.

Intent

Take a Process Level event map — a flow the team already agrees on — and turn it into a design: aggregates, bounded contexts, crossing events, policies, a rule that triggers a command in response to an event ("when X happens, do Y"), and the places you need an anti-corruption layer. The session is shorter than Process Level, narrower in scope, and denser in content. The output is a design the room committed to in front of each other; that's what makes it survive the first sprint.

When to use it

Run Architecture when:

- Process Level has clarified a flow and you're about to build or re-architect it
- You're splitting a monolith and need to agree on the seams
- A new service is being designed into an existing landscape and you need to decide what it owns, what it subscribes to, and what it emits
- Two teams own overlapping responsibilities and you need boundaries before the next release

Don't run Architecture when:

- The flow isn't clear yet — run Process Level first
- The domain is unfamiliar to half the room — you'll end up re-doing Process Level under a different label
- You don't have developers with design responsibility in the room — the session only commits if the people who'll implement it are the ones drawing boundaries

A few terms before we start

Three DDD terms get used throughout this session. If you've been following the DDD posts earlier in this series, you'll recognise them; if you've arrived here cold, the following is orientation rather than a replacement for Eric Evans or Vaughn Vernon.

Aggregate. The smallest unit of transactional consistency — a cluster of events and state that *must* change together to keep the business correct. Classic example: a shopping cart. The line items and the total are one aggregate because the total has to equal the sum of lines at every moment; if a crash left one updated and the other not, the cart is lying. The catalogue price of a book? That sits outside. The cart can tolerate the catalogue price drifting while it's sitting in the customer's basket.

Invariant. A rule that must always be true for an aggregate to be valid. *"The cart total equals the sum of line items." "A cancelled order cannot be packed." "Every refund traces to a charge."* Invariants define aggregate boundaries: if you can state a rule that spans two events, those events probably belong in the same aggregate.

Bounded context. A larger linguistic and design boundary around a group of aggregates that share a vocabulary. Where Invoice, Payment, and Refund might be three separate aggregates, they often live inside one Billing bounded context where everyone uses the same language for *currency*, *period*, *ledger*. Outside the context, the same word can mean something different — a "customer" in Billing is an account with a payment method; a "customer" in Support is a person with a name and a ticket history.

A session can conflate aggregate and bounded context on first pass. Most clusters you find will be either a single aggregate or a small context containing two or three aggregates; you resolve which in Phase 3.

Participants

Facilitator. Ideally someone with both Event Storming experience *and* enough design background to spot when a proposed aggregate is about to fall over under its own invariants. If you can't find that person, pair a facilitator with a technical lead.

Developers and architects. The heart of the room. These are the people who'll implement the design — they need to do the drawing, and they need to commit in front of each other.

A domain expert who can veto clusters that don't match reality. Not to propose boundaries, but to stop the developers drawing ones that'll break on contact with the business.

A product owner or equivalent. To turn the output into backlog shape in the days after.

Group size: 4–8. Architecture sessions are thinking-hardest work; above eight voices, the argument space fragments and decisions stop landing.

Prerequisites

A Process Level wall. The ideal is that it's physically up in the room; in practice the Process Level session might have been two weeks ago, the original wall is gone, and all you have is photographs and a transcribed event list. That's enough. The day before the Architecture session, redraw the Process Level wall: print the event list big, cut it into strips, stick them in time order on fresh paper. Ten minutes of preparation buys you an hour of session productivity.

The Architecture session assumes *a wall is in the room*. It does not assume it's the original wall.

Materials and timing

Phase	Duration	Materials	Key question
Arrivals, orient to the wall	~15 min	–	"Do we all still agree?"
Review Process Level wall	15 min	Existing wall	"Anything drifted?"
Identify aggregate candidates	35 min	Dots of 3–4 colours	"What must change together?"
Draw boundaries	30 min	Marker, wall	"Where are the seams?"
Crossings: commands vs events	25 min	Blue, orange, arrows	"Who tells who what happened?"
Make policies and read models explicit	30 min	Purple, green	"What reacts when? On what data?"
External vs internal systems	15 min	Pink, yellow	"Whose contract do we live with?"
Wrap-up, owners	15 min	–	"Who owns what next?"
Buffer	20 min	–	–
Total	Plan for 3h 15min inside a 3.5-hour block. First-time sessions almost always overrun Phases 3 and 4.		

A note on note colours

Architecture uses the same Process Modelling palette as Process Level — orange events, blue commands, small yellow actors, lilac policies, pale-green read models, a query-shaped projection of state, optimised for a screen or report rather than for writes, pink hotspots. What changes is which notes carry weight.

- Purple policies go from occasional to mandatory. At Process Level you could let most policies stay implicit (events quietly triggering commands) and only stick purple on the wall when the room argued one out. At Architecture, every event that crosses a boundary triggers *something* on the other side — and that something is named as an explicit *"whenever X, then Y"* rule. If you don't know the policy, you don't know the design.
- Pale-green read models become pinning. Any policy whose decision depends on data needs the data named. *"Before reserving stock, check current stock level."* The green note is the facts the policy reads; it's also a hint about which aggregate owns the read model and which one owns the writes.

Architecture also adds one colour and two new physical materials that Process Level doesn't use:

- Large yellow stickies — aggregates. Brandolini's canonical use of the big yellow note: a cluster of events and commands that must change together to keep an invariant true. Large yellow goes *behind* the events and commands it owns.
- A thick marker to draw bounded-context boundaries directly on the paper.
- Coloured dots (four or five colours) to let small groups cluster events into candidate aggregates before anyone commits to a boundary line.

Facilitator playbook

Phase 1 — Orient to the wall (15 min)

Walk the Process Level wall end-to-end, out loud, pointing at each event. Invite corrections. Make them on the wall. Then close the door:

> *"Good. From this point on, the events on the wall are the ground truth. If you disagree with the flow, that's a different session. Today we're asking where the code boundaries go."*

What to watch for:

- Someone re-opening Process Level decisions. Let small corrections happen; block major re-litigation. *"That's a Process Level conversation — let's book it and move on."*
- Events that have quietly shifted since the session. Worth updating before you design on top.
- Silent consent that isn't real consent. Name one person directly: *"You were at the Process Level session — does this flow still match what you remember?"*

Phase 2 — Identify aggregate candidates (35 min)

This is the heart of the session. Give the framing out loud:

> *"An aggregate is a group of events that belong together because they share the same rules. If two events can never be out of sync without the business being wrong, they're in the same aggregate. If they can drift for a second without anyone caring, they're probably in different ones. We're looking for the natural seams."*

Then teach the tests. People in their first Architecture session have the framing but not the technique. Give them five concrete tests they can point at two events and apply out loud:

1. The crash test. *"If the server crashed between these two events, would the business be in an invalid state?"* If yes, they must happen together — same aggregate. If no, a brief gap between them is survivable — probably different aggregates. *Same aggregate:* Cart Line Added and Cart Total Updated — if a crash left the line in but the total not updated, the cart is lying. *Different aggregates:* Payment Captured and Receipt Emailed — if the email didn't send, the customer is still correctly charged and you can retry later.
2. The shared-rule test. *"Is there a rule that depends on both of these events being in sync?"* If you can write a rule like *"the total equals the sum of lines"* or *"a cancelled order cannot be packed"*, the events the rule touches are in the same aggregate. *Same aggregate:* Order Placed and Order Cancelled — governed by the rule *"an order is exactly one of: placed, packed, dispatched, delivered, cancelled."* *Different aggregates:* Order Cancelled and Refund Issued — the refund follows from the cancellation, but the rules governing refunds (amount, eligibility, accounting) are separate from the rules governing order state.
3. The same-ID test. *"Is this ID the thing the event is about, or the thing it points to?"* Every aggregate has one identity. Events inside it carry that identity as their primary key; events in other aggregates may reference it but aren't about it. The test is a hint, not a rule.
4. The Conway's-Law test., "any organisation that designs a system… will produce a design whose structure is a copy of the organisation's communication structure" (Mel Conway, 1968), *"If someone had a question about this event, which team or role would they ask?"* Events that share an owner are usually in the same aggregate. Important caveat: this test is backward-looking — it catches aggregates your org already reflects. Use it to *confirm* a cluster, never to *define* one. If you let Conway's Law lead, you'll draw boundaries on team lines that dissolve the next time the org re-shuffles.
5. The long-running-process test. *"Is there a stateful wait — a human decision, an approval, a scheduled delay — between these two events?"* If so, you almost certainly have a long-running policy (also called a process manager or saga, a long-running policy that coordinates a sequence of commands and events across boundaries) between them: a persistent state waiting for the next input. That's not automatically an aggregate boundary — sometimes the long-running state lives inside an aggregate as a sub-process — but it's a signal to look closer at where the state lives while nobody is touching it.

Tell the room: *"Try two or three of these on every pair of events you're not sure about. The tests will sometimes disagree — that means the boundary is genuinely tricky and worth a conversation."*

Hand out coloured dots — one colour per candidate aggregate — and let small groups put dots on events they think belong together. Don't draw boundaries yet. The dots are cheap and disposable; boundaries are not.

External systems and the tests. All five tests assume both events are in systems you own. Real architectures have events on the other side of payment processors, email providers, carrier APIs. When a test points at an event across an external boundary, the honest answer is: *these are in different aggregates and they're separated by an anti-corruption layer; the invariant is eventual-consistency-plus-reconciliation, not same-aggregate-atomicity.* Phase 6 is where you mark those explicitly; until then, pink-note them and move on.

Phase 3 — Draw boundaries (30 min)

Now the marker comes out. For each agreed cluster of dots, draw a thick line around it on the paper. The act of drawing is deliberate: it makes the commitment visible and forces the room to look at the edges.

> *"Each line we draw is a design decision we all agreed to. If you're not sure, say so now. Once it's drawn we move on."*

For each boundary, ask: *"is this one aggregate, or is it a bounded context containing a few aggregates we haven't separated yet?"* Split where the answer is "bounded context" — a Billing context containing Invoice, Payment, and Refund as three aggregates is common.

Two rules to state out loud, in the room, as the wall fills up. These are the rules that keep the wall from becoming a flowchart:

1. Events don't cause events. An event is always followed by something that reads it — a policy, a person, a clock — which in turn issues a command. If someone asks *"so does Payment Captured cause Stock Reserved?"*, the answer is *"no — the Inventory aggregate reads Payment Captured, a policy fires, that policy triggers Reserve Stock, and Reserve Stock produces Stock Reserved."*
2. Commands don't cause commands. A command produces exactly one event (on success) or fails. If the next command needs to happen, it's triggered by a second policy that subscribes to the first command's event. Never draw an arrow from one blue note directly to another — there is always an event between them, even if you haven't named it yet.

What to watch for:

- The god aggregate. One cluster absorbing most of the wall. Usually called "Order" or "Subscription." Almost always wrong. *"What invariant forces all of this into one place?"* If there isn't one, split.
- Too many tiny aggregates. One event per aggregate means you're building a distributed monolith. *"If these two always change together, they probably belong together."*
- The org-chart design. Boundaries landing exactly on team lines. *"Are we drawing the domain or the org chart?"*
- Boundaries drawn too early. They get defended instead of examined. If there's genuine disagreement, keep the dots, erase the marker line, discuss.

Phase 4 — Crossings: commands vs events (25 min)

Walk each boundary. For every arrow that crosses a line, classify it:

- Crossing event (the default) — the publisher announces a fact; the subscriber reacts on its own terms via a policy on the receiving side. *"Payment Captured — anyone who cares can listen."*

- Crossing command (the exception) — the sender tells a specific receiver to do a specific thing and expects to know it was done synchronously. *"Issue refund for invoice X."*

Draw them with a different colour marker from your boundary lines. The default is crossing events plus a policy on the receiving side. If you drew a command across a line, you usually forgot to draw the policy, the rule on the receiving side that translates the event into a command *inside* its own context. The reshape:

> *Before: Order context sends* `Reserve Stock` *command into Inventory context.*
>
> *After: Order context publishes* `Order Confirmed` *event. Inventory context owns a* `whenever Order Confirmed then Reserve Stock` *policy that issues the command inside Inventory.*

Crossing commands are legitimate when a specific sender needs a specific receiver to do a specific thing *and* needs to know it was done synchronously, almost always because a human decision is driving it (a support agent, an admin, a customer clicking a button). If the chain is system-to-system, default to events.

Phase 5 — Make policies and read models explicit (30 min)

Any purple policies already on the wall from Process Level stay where they are. For *every* crossing event, the subscribing aggregate does something in response — and that something now has to be on the wall, even if it felt obvious. Put a purple policy note inside the subscriber's boundary, with the form *"when X, then Y"*.

> *"Purple note next to the landing point. 'When Payment Captured arrives, reserve stock against the new order.' That's the contract the subscriber has with the event."*

If the policy needs information to decide, add a pale green read model note next to it. *"When Payment Captured arrives, check current stock level, reserve if available."* The "current stock level" is a read model.

A note on reactive vs long-running policies. Most purple notes are *reactive* — when X, do Y, done in one step. Some policies carry state across multiple events. A refund approval is an example: the policy starts when Refund Requested fires, waits for human approval or rejection, then issues or denies. Brandolini calls these long-running policies; Vernon calls them process managers; the generic DDD term is saga. Both kinds are purple. If you find a policy whose *when* and *then* are separated by hours, days, or a human decision, flag it — it needs its own persistent state, which is often either its own aggregate or a sub-entity.

Phase 6 — External vs internal systems (15 min)

Walk the yellow notes from Process Level. For each, ask: *"is this something we own, or something we integrate with?"*

- External (payment processor, email provider, carrier, tax API): we don't control its shape. We need an anti-corruption layer between us and them — a translation that keeps their vocabulary out of our domain model.
- Internal (our own services, schedulers, workers): design decisions we're making now. Each one gets a bounded context and aggregates of its own.

Ring externals with one colour and internals with another. For each external, name the anti-corruption layer explicitly on the wall: *"ACL: Stripe → our Payment."* *"ACL: carrier X → our Delivery."*

Worked example — Pagebound's Payment Captured fan-out

Pagebound's Process Level session on the order-to-delivery flow left one event marked as the pivot the whole design hangs off: Payment Captured. The session went into Architecture to answer one question: *"When Payment Captured fires, who listens and what do they do?"*

Five people in the room: the commerce lead, two commerce engineers, the fulfilment tech lead, the SRE who owns the payment integration. A printed copy of the Process Level wall on fresh paper along one side of the room.

Phase 2 produced five aggregate candidates from the fifteen events on the Process Level wall:

- Order — *Checkout Started, Payment Submitted, Order Confirmed, Order Cancelled.* The customer's intent and its fate. Invariant: *"an order is exactly one of: pending, confirmed, cancelled."*
- Payment — *Payment Captured, Payment Failed, Refund Issued.* Money state. Invariant: *"every refund traces to a capture; the ledger balances."*
- Inventory — *Stock Reserved, Stock Released, Stock Decremented.* Availability. Invariant: *"reserved + available ≤ on-hand."*
- Fulfilment — *Items Picked, Order Packed, Label Printed, Handed to Carrier.* The physical workflow. Invariant: *"an order ships exactly once."*
- Delivery — *Scanned At Hub, Out For Delivery, Parcel Delivered.* The carrier's view. Mostly external; our representation is a reflection of the carrier's API.

Plus a side-effect subscriber (not a full aggregate):

- Notifications — sends emails. No invariant; no persistent state. An adapter over SendGrid.

Here's the whole choreography on a wall — Payment Captured at the centre, fanning out to three subscribing aggregates and one adapter:

A few things worth noticing in the wall:

Three subscribers react to Payment Captured directly; Fulfilment waits for Stock Reserved. The Order aggregate confirms the order. Inventory reserves the stock (and only after reading its stock-levels read model to decide whether there *is* stock). Notifications sends the receipt. Fulfilment deliberately doesn't subscribe to Payment Captured — it waits for Stock Reserved, because packing an order that doesn't have stock reserved would be premature. This is exactly why Payment Captured is multi-subscriber and Stock Reserved is a further handoff: Fulfilment only acts on orders that have *actually made it through* the inventory gate.

Commands chain via internal events, not directly. Inside Fulfilment, the policy kicks off a chain — pick, pack, hand off — that goes through several internal events. None of them appear on the wall individually (we condensed them into one command-chain note for space) but the pattern is: each command emits an event, the next command subscribes to it, never a direct command-to-command call. That's what keeps Fulfilment testable and replayable.

Notifications is labelled "adapter — not a full aggregate". It has no invariant. It doesn't own state that can go wrong. It's a listener that reacts to domain events by calling SendGrid. Including it in the fan-out diagram shows that the pattern *accommodates* side-effect subscribers alongside real aggregates — not every subscriber has to be a full aggregate with its own rules.

What can go wrong

The god aggregate. One cluster absorbing most of the wall. *Recovery:* Pick one invariant and ask what really depends on it. Split where the answers diverge. *Stop if:* A second attempt produces the same pattern. You may be designing at the wrong scope.

The distributed monolith. Every command crosses a boundary. *Recovery:* Stop drawing boundaries. Ask: *"if we had one service, what would actually need to split?"* Redraw from zero. *Stop if:* The second attempt produces the same pattern. The Process Level flow is one indivisible thing, or the scope is wrong.

The org-chart design. Boundaries landing exactly on team lines. *Recovery:* Name it out loud. *"We're drawing the org chart, not the domain. Let's redraw the domain and argue about ownership afterwards."* *Stop if:* The team lines are fixed by a decision outside the room. Make the constraint explicit; don't pretend you designed freely.

The solution architect. One person has a design in their head before the session and is steering towards it. *Recovery:* Pair them with the most junior developer; give both of them one cluster to dot. *Stop if:* Three nudges in and they're still steering. The session is producing their design, not the team's.

Process Level ambiguity leaks in. The wall has events that different people describe differently. *Recovery:* Stop and fix the Process Level event. This is scope creep but it's worth 5 minutes. *Stop if:* More than three events have this problem. Schedule another Process Level session; don't build an architecture on a wobbly wall.

Outputs

Same day, 24 hours:

- Panoramic photographs including boundary markers and crossing arrows.
- Transcribed aggregate list, bounded context list, command/event API list, and hotspot list in a shared document.
- A short summary: *"Here's the design we landed on, here's what's still open, here's what happens next."*

The product owner's (or tech lead's) week:

- Turn the bounded contexts into vocabulary. Each context has a name the team will use for the next year. Pin them down; don't let them drift.
- Reshape the backlog along the boundaries. Stories crossing three bounded contexts are a smell; split them where the boundaries say to.
- Book follow-ups for the external integrations. Each anti-corruption layer is a small workshop of its own — don't let them wait until the first sprint blows up.

- Walk the design with anyone who couldn't attend. Tech leads on adjacent teams especially — their reaction tells you whether the boundaries survive at the edges of the system.

Where to go next

- Event Storming a Process — the input. If the wall didn't produce a design, the Process Level wall it was built on might need revisiting.
- Event Storming a Domain — the grandparent. Revisit Big Picture when you've built enough of the design to discover a new cross-team hotspot.
- Event Storming: Building Shared Understanding — the narrative version, showing a smaller team working through the technique for the first time.

About this playbook

This playbook is part of *The Workshop*, a reference series of facilitator playbooks published at barkingiguana.com. The canonical, up-to-date version lives at barkingiguana.com/writing/the-workshop-event-storming-an-architecture/.

These posts are LLM-aided. Backbone, original writing, and structure by Craig. Research and editing by Craig + LLM. Proof-reading by Craig.