THE WORKSHOP

C4 Modelling

A simple set of nested diagrams — Context, Container, Component, Code — that turns a wall of sticky notes and tribal knowledge into an architecture artefact that survives outside the room.

2026-07-31
barkingiguana.com/writing/the-workshop-c4-modelling/

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

Contents

The natural follow-on to Event Storming an Architecture. The Event Storming wall is the raw material; the C4 diagram is what you pin on the team room wall, put in the onboarding pack, and show the auditor. Same model, different medium — and the C4 session is where one becomes the other.

C4 Modelling

C4 is a way of drawing software architecture at four nested levels of zoom (Context, Container, Component, Code) invented by Simon Brown around 2011 as a reaction to the "boxes and arrows that nobody can explain" diagram he kept meeting in the wild. Each diagram picks one level of abstraction and stays inside it: a system context diagram doesn't show databases; a container diagram doesn't show classes; a component diagram doesn't show deployment topology. Two supplementary views (dynamic and deployment) sit alongside the four nested ones, so the full framework is really five notations. Also known as the C4 model or Simon Brown's C4; sometimes confused with UML (C4 is lighter and doesn't carry the semantic baggage), with arc42 (a documentation template that can happily host C4 diagrams), and with the 4+1 view model (Kruchten's older multi-view approach, which C4 partly descends from but simplifies). This post covers the workshop that produces the first usable set of diagrams for a system, usually run immediately after an Event Storming an Architecture session, turning the wall of stickies into durable documentation.

At a glance

- *Who, for how long:* a facilitator who holds the zoom level, one or two architects / tech leads, two or more developers who build the system, and an operations representative if deployment is in scope. Four to six people, three hours.
- *What you walk out with:* a C1 System Context, a C2 Container diagram, selectively one or two C3 Component diagrams, plus a tool decision, a named owner, and a maintenance cadence for the durable version.
- *When to reach for it:* an Event Storming an Architecture session has just landed and the wall needs to become durable, or onboarding / compliance / boundary arguments are surfacing the cost of architecture that only exists in tribal knowledge. Not for systems whose behaviour you haven't pinned down yet (run Event Storming first), code-level structure inside one class (your IDE draws it), or single-file utilities.

What's It For

You have a system. Somewhere between three people's heads, a README nobody updates, and a wiki page from 2023, the architecture exists. It lives in the tribal knowledge of the two developers who've been there longest. Every new hire learns it by osmosis; every incident review surfaces a bit more of it; every cross-team conversation discovers that somebody's mental model is six months out of date.

A team has just run an Event Storming an Architecture session. The wall is covered in orange, blue, pink, and yellow notes. Boundaries have been drawn with a thick marker around aggregate (a small consistency boundary you reach via a single root entity) clusters. Crossing arrows point from one bounded context (a vocabulary boundary inside the system, where the same word can mean different

things on different sides) to another. Everyone in the room understands the design. A week from now, the wall will be photographed, the photo will be dropped into Confluence, and the photo will stop being something anybody can read. The design is correct; the medium is wrong.

Or: a team is onboarding its fourth developer in six weeks. Each one has asked the same question (*"so where does billing actually live?"*) and each time the answer has been a different whiteboard sketch by whoever was closest. The sketches are all *approximately* correct and none of them are the same. The team needs one picture, in one place, at one level of detail, that everyone agrees on.

Or: the security team has asked for an architecture diagram before they'll sign off on a compliance audit. What arrives is either an infrastructure diagram (every VM and load balancer), a deployment diagram (every pipeline step), or a conceptual diagram that's too abstract to answer "which service has the personal data in it." None of these are what the auditor wants. The auditor wants a Container diagram with trust boundaries on it, and nobody has ever drawn one.

Or: a team is splitting a monolith and the architects keep having the same argument in slightly different forms. They share a vocabulary but not a picture. One person thinks the billing service is a container; another thinks it's a component inside the platform container. The argument is really about *which level of zoom they're each at*, and they don't know it because they've never drawn the levels out explicitly.

C4 is the workshop you reach for when the architecture exists in practice but doesn't exist on paper, and the cost of that mismatch has started showing up in onboarding, incident reviews, compliance requests, and boundary arguments.

Reach for it when:

- An Event Storming an Architecture session has just run and you want to turn the wall into durable documentation while the decisions are still fresh; this is the canonical use case
- A team's architecture exists in tribal knowledge and onboarding costs are rising
- Multiple people are drawing inconsistent whiteboard sketches of the same system
- A compliance, security, or audit request needs a real architecture diagram and the options on hand are either infrastructure drawings or hand-waving
- Two teams are arguing about the architecture and it turns out they're each at a different zoom level without knowing it
- A monolith split is underway and the team needs to see the before and the after at the same level of detail
- A multi-region deployment is being planned and the current architecture doesn't distinguish container from infrastructure

What It's Not For

Skip it when:

- You haven't decided what the system actually *does*. You're not ready for C4, you're ready for an Event Storming session
- The goal is to describe code structure inside one class. That's a C4 Code diagram, which your IDE draws for you

- You want a detailed infrastructure inventory. That's an infrastructure diagram, and C4 deployment views are lighter than that by design
- The scope is a single script, a single Lambda, or a single-file utility
- The diagrams will be drawn by one architect and emailed round for approval. C4 only works if the people who own the system own the diagram

The trade-offs to weigh against the benefits:

Benefits

- A small set of nested diagrams at the correct levels of abstraction, each answering one question clearly, drawn by the people who will maintain them
- An onboarding artefact that survives the original drawers leaving
- A common vocabulary for architecture conversations; "is that a C2 or a C3 concern?" is a faster question than most of the alternatives
- For teams coming out of an Event Storming an Architecture session, a durable record of the bounded-context decisions the wall captured
- A diagram the auditor, the security team, and the new hire can all read without a translator, at the level each of them needs

Costs

- 12–18 person-hours for a 3-hour session
- The recurring cost of keeping the diagrams current, which is real and will quietly consume the value if it isn't paid
- A tooling decision the team has to commit to and maintain
- The organisational friction of naming an owner for the durable version
- Political cost when the diagram reveals that the current architecture doesn't match the current team boundaries

Stop-the-session signals

- The group cannot name which level of zoom they are currently at
- A second attempt at C2 produces the same org-chart shape as the first
- The Event Storming wall the session was built on turns out to be wrong in foundational places
- Nobody in the room will own the durable version

Ending early is not failure. Producing a diagram the team doesn't own and won't maintain is.

Definitions & Background

C4's name names the four canonical levels: C1 System Context, C2 Container, C3 Component, C4 Code. Three supplementary views (System Landscape, Dynamic, Deployment) sit alongside them. Most teams need C1 and C2; everything else is on demand.

You will not draw all of them in one session, and you should not try. Each level answers a different question for a different audience.

C1: System Context. *The board-room diagram.* One big box in the middle: your system, named the way the business names it. Around it: the people who use it (customers, operators, admins) and the external systems it integrates with (Stripe, SendGrid, the warehouse API, the tax service). Arrows in and out, labelled with what flows across them. No internal structure. No databases. No services. This is the diagram you show an investor, a board member, a new hire on day one, an auditor asking *"what does your system even do?"* It answers *"what is this thing, who uses it, and what does it talk to?"* and nothing else.

C2: Container. *The workhorse.* Zoom into the system box. A container in C4 is *"a separately runnable or deployable unit, or a data store the system depends on at runtime"*: a web app, an API, a mobile app, a database, a worker, a message broker, a scheduled job. Brown's current definition is "something that needs to be running for the system as a whole to work"; a database counts even though you don't deploy it in the conventional sense. Not a Docker container (though a Docker container is often a C2 container). The C2 diagram shows all the containers that make up your system and the relationships between them: which web app talks to which API, which API reads and writes which database, which queue sits between which two services. This is the diagram the team lives with. It's the one you pin on the team-room wall. It's the diagram that most directly maps onto the bounded contexts from an Architecture session: each bounded context typically corresponds to one or two containers. If you only draw one C4 diagram, draw this one.

C3: Component. *The selective zoom.* Zoom into one container. A component is a grouping of related functionality inside a container, usually a cluster of classes, modules, or packages that have a single responsibility. This is where aggregates from the Event Storming session usually end up: one C3 Component per aggregate, or per cluster of closely related aggregates inside the same container. Crucially, you do not draw a C3 diagram for every container. Most containers are simple enough that a C2 diagram plus the code is sufficient. Only containers with enough internal complexity that a developer can't hold the shape in their head deserve a C3. Drawing C3 for every container is one of the most common C4 failure modes and produces dozens of diagrams nobody looks at.

C4: Code. *The auto-generated level.* Zoom into one component. A C4 Code diagram is a class diagram, or something equivalent. Don't draw these by hand. Your IDE will generate them on demand. IntelliJ, Visual Studio, and every decent language server can show a class diagram for a package in a few clicks. Mention this level once in the session so people know it exists, then move on. Investing workshop time in hand-drawing C4 is a waste; the generated version is always more accurate, never goes stale, and costs nothing to recreate.

System Landscape (supplementary). *The enterprise view above C1.* Not a numbered level but a supplementary view that sits above C1. Shows multiple systems across an organisation and how they relate. Useful in post-acquisition integration, in multi-product companies, or when you're trying to explain to a new CTO what they've just inherited. Most single-product teams never draw one. You know you need it when someone asks *"which of our systems even exists?"* and the room can't name them all without checking a spreadsheet.

Dynamic views. *The scenario walk-through.* A dynamic view takes the elements from any C4 level (usually C2 or C3) and draws them in the order they collaborate for one specific scenario. It looks like a sequence diagram (numbered arrows, step by step) but uses C4 elements rather than UML lifelines. Dynamic views are how you explain *"what happens when a customer signs up?"* or *"what happens when a*

payment fails and gets retried?" without redrawing the whole system. They pair naturally with the event flows you captured in a Process Level or Architecture Event Storming session: one dynamic view per hot scenario is usually enough.

Deployment views. *The infrastructure mapping.* A deployment view takes the containers from your C2 diagram and places them on the physical or cloud infrastructure they run on: regions, availability zones, VMs, Kubernetes clusters, serverless functions, managed databases, CDNs. This is the diagram the SRE and on-call teams want. It's also the diagram that matters most for multi-region systems, because the same Container diagram can map to wildly different deployment topologies (single-region, active-active multi-region, primary-replica with failover) and the operational implications are completely different each way. If your system runs in more than one region, a deployment view is not optional.

Five notations, two of them optional for simple systems, all of them answering different questions for different people. The mistake to avoid is treating them as a sequential checklist. Draw the ones you need for the question you have. A fresh team usually needs C1 and C2. A team onboarding new hires may only need C2. A team doing a compliance audit may need C2 plus a deployment view with trust boundaries. A team debugging a complex flow may need C2 plus a dynamic view. Start with the question, pick the level that answers it, and stop.

Inputs

A C4 session is less material-heavy than an Event Storming session. A whiteboard is enough for the first-pass drawings; the whole point of C4 is that its notation is simple enough to draw with a handful of rectangles and labelled arrows. The important material is actually *digital*: a tool the team will commit to for the durable version. Structurizr (Simon Brown's own tool), draw.io / diagrams.net, Excalidraw, Mermaid with a C4 plugin, or a plain-text DSL like Structurizr Lite all work. Pick one in the session. Don't leave the room without the decision made.

What you need on the day:

- A whiteboard or large wall surface with markers in at least two colours, the primary working surface for first-pass drawings.
- An Event Storming wall, if one exists from a recent Event Storming an Architecture session. With one, the C4 session is a translation exercise and moves twice as fast. Without one, the session will be slower because the group is both discovering and drawing at the same time.
- A clear three-hour block with no interruptions. The session has six working phases plus a break and a 20-minute slack buffer.
- A shortlist of candidate digital tools for the durable version: Structurizr, draw.io / diagrams.net, Excalidraw, Mermaid with a C4 plugin, PlantUML with C4-PlantUML. The team picks one before leaving the room.
- The right people in the room (see *Who's Needed*).

Outputs

What lands on the table at the end of the session:

- A C1 System Context diagram: one box for the system, the actors and external systems around it, arrows labelled with what crosses them. The diagram a new hire, an investor, or an auditor can read at a glance.
- A C2 Container diagram: the workhorse. Every separately runnable or deployable unit and every datastore the system depends on at runtime, with the connections between them. The diagram the team pins on the wall.
- C3 Component diagrams, selectively, only for containers complex enough to earn a zoom-in. Many sessions produce zero, one, or two C3 diagrams; that's correct.
- A decision log of which levels were drawn and why, including explicit *no* decisions ("Payment worker: no C3, code is sufficient").
- A tool decision for the durable version, an owner name, and a deadline, typically within a week of the session.
- A maintenance cadence: review at every architecture-affecting PR, monthly, at retrospectives, or whenever the next major change lands. Pick one the team will actually do.
- Drift notes: where the C4 diagrams diverged from the Event Storming wall, captured as follow-ups.

Photograph the whiteboard at each level, high-resolution, lit well enough that labels are legible.

These outputs feed into:

- Event Storming an Architecture. If drift surfaces between the Event Storming wall and the C2 diagram, that drift is information for the next Architecture session.
- Process Level Event Storming. The event flows from a Process Level session feed directly into C4 dynamic views. One dynamic view per important scenario, drawn with the containers and components from your C4 diagrams, is often the clearest way to explain "what happens when X?" to a new joiner or a security reviewer.
- Example Mapping. When a C3 component's rules are unclear, Example Mapping is the pattern that turns the unclear rules into rules-and-examples before any code gets written. A C3 component whose behaviour nobody can state is a component waiting for an Example Mapping session.
- Threat Modelling *(publishes later)*. The C2 Container diagram with trust boundaries drawn on it is one of the canonical inputs to a threat modelling session. The containers, the external systems, the arrows between them, and the data that crosses each arrow are exactly what the threat model needs as its starting picture.
- Architecture Decision Records *(publishes later)*. The decisions a C4 session surfaces (why this container and not two, why this datastore is shared, why this integration goes through an anti-corruption layer) are natural ADR candidates. The diagram shows *what* the architecture is; the ADRs explain *why* it is that way, and they compound over time in a way a diagram on its own can't.

Who's Needed

Group size: 4–6. Same shape as an Architecture session. Smaller than a Process Level session because this is a design and documentation activity, not a discovery one. Two people and you lose the pressure-testing; eight and the diagram gets drawn by committee, which produces the same "boxes and arrows nobody can explain" you were trying to escape.

- Facilitator. Someone who knows C4 well enough to stop the session drifting across levels, and who has the willpower to say *"that's a C3 concern, park it"* when someone starts drawing classes on the C2 diagram. They don't need to be the architect; they need to be willing to hold the scope.
- Architects / tech leads. The people who carry the architectural shape in their heads. Usually one or two. This is the session where their internal model becomes external, and they'll often discover they disagreed about something they thought they agreed on.
- Developers who build and change the system. At least two. The same rule as an Architecture session: diagrams drawn by people who don't write code get ignored by people who do. A C2 diagram the team didn't help draw is a wiki page, not an artefact.
- One operations representative, if the system has any meaningful deployment concerns. SRE or platform engineering. They come alive during the deployment view and are often the only person in the room who can name every region, every load balancer, and every piece of infrastructure accurately.
- Optional: a technical writer or documentation lead. If you have one. They'll be the person who keeps the diagram alive after the session, and sitting in means they can translate between the whiteboard shorthand and the durable format later.

Who to leave out:

- Product and design. They were central to earlier sessions; this is about technical structure. Inviting them turns a C4 session into a scope conversation.
- Anyone whose job title says "architect" but who will not be in the room when the diagram decays. Architectural pronouncements from people who won't maintain the artefact are the reason most architecture diagrams die.
- Stakeholders, leadership, and sponsors. They see the C1 diagram afterwards. They do not participate in drawing it. A C1 diagram drawn in front of an audience becomes a marketing diagram.

How To Run It

Phase	Duration	Materials	Key question
Scope and level framing	15 min	Whiteboard	“Which diagrams do we need?”
System Context (C1)	25 min	Whiteboard, markers	“What is this system and what does it talk to?”
Container (C2)	45 min	Whiteboard, markers, Event Storming wall if present	“What runs, where does state live, how does it connect?”
Break	10 min	,	,
Component (C3): selective zoom	30 min	Whiteboard	“Which containers earn a zoom-in, and what’s inside?”
Review and name-check	20 min	Whiteboard, Event Storming wall	“Does this match reality? What drifted?”
Wrap-up, tooling, owners	15 min	,	“Who owns the durable version and by when?”
Buffer	20 min	,	,
Total	2h 40min inside a 3-hour block		

The six working phases are 150 minutes. The remaining 30 minutes are for the boundary arguments that always run long at the C2 level, the moment when someone challenges a container that turns out to be two, and the wrap-up conversation about which tool the team will commit to for the durable version. Twenty minutes of genuine slack. Don’t try to fill it.

One level at a time

The session alternates between whole-group drawing and small-group review. Drawing should be whole-group at C1 (because the audience is the whole group), small-group at C2 (paired architects and developers work fastest), and whole-group again for the review. The facilitator’s main job is to keep the group at *one level of zoom at a time*. The single biggest failure mode of a C4 session is someone starting to draw components on the container diagram, or starting to draw infrastructure on the container diagram, because they’re trying to show every concern at once.

The rhythm is scope, draw, step back, drop a level. Scope the level (what question are we answering?), draw it, step back and ask *“is this answering the question?”*, then either drop a level or stop. If a level doesn’t earn its place, don’t draw it. A session that produces C1 and C2 and explicitly decides not to draw C3 is more valuable than a session that produces C1, C2, and six half-finished C3 diagrams.

A running example runs through the phases below: Pagebound, an online independent bookshop, deliberately ordinary so the moves are visible without being drowned in domain novelty. If you’ve read the Event Storming an Architecture post, this is the same system, picked up at the moment the Event Storming session ended.

Phase 1: Scope and level framing (15 min)

Before any box is drawn, agree which diagrams the session will produce. This phase is short, and skipping it is the single biggest reason C4 sessions run long.

Open with:

> *"C4 gives us five notations: Context, Container, Component, plus dynamic and deployment views. We are not going to draw all of them today. We are going to decide, first, which diagrams answer the questions we actually have, and draw those. Every level we draw needs a reason to exist."*

Ask the group what the diagrams are *for*:

> *"Who is going to read these, and what do they need to know? If someone picks up this diagram a month from now, what question should it answer for them without needing a translator?"*

Write the answers on the side of the whiteboard. Typical answers:

- *"The team needs to agree on container boundaries"* → draw C2
- *"We need to show the security team what crosses trust boundaries"* → draw C2 plus a deployment view with trust boundaries
- *"New hires keep asking the same onboarding questions"* → draw C1 and C2
- *"We want to document the design from the Event Storming session"* → draw C2, probably C3 for one or two containers, and maybe a dynamic view

Then check whether an Event Storming wall exists:

> *"Is there an Event Storming wall we can use as input? If yes, it tells us where the container and component boundaries already are; we're translating, not designing from scratch."*

If there is an Event Storming wall, this is a translation session and it will move twice as fast. If there isn't, the session will be slower because the group is both discovering and drawing at the same time. Either works; the clock budget is different.

What to watch for:

- "Let's do all the levels." Push back immediately. *"Which of those levels is answering a real question? If we can't name the question, we don't draw the level."* A session that produces two good diagrams is worth more than one that produces five shallow ones.
- Scope creep into deployment or dynamic views. Fine if it's been chosen deliberately. Not fine if it's happening because someone remembered they want to discuss the Kubernetes topology. Park deployment until the end of C2, and only draw it if C2 is clean.
- Disagreement about the audience. *"The diagram is for the team"* and *"the diagram is for the board"* are two different diagrams. Pick one audience per level. If you need both, plan two outputs.
- A hidden agenda. Someone is in the room because they want the session to bless a decision they've already made. Name it: *"We're drawing what the system actually is, not what we wish it was. If there's a should-be diagram we need, we'll draw it separately."*

Phase 2: Draw the System Context (C1), 25 min

Start with the single biggest box in the middle of the whiteboard. Give it the name the business uses for the system, not the internal project codename, not a technical nickname. *"Pagebound,"* not *"commerce-platform-v2."* If the business has a public name for this thing, use it.

Open with:

> *"This is the one-paragraph picture. One box in the middle: the system. Around it: the people who use it, and the external things it talks to. We are not drawing anything inside the box on this diagram. That's the next level. If you catch yourself wanting to draw a database, resist. The database goes on the next diagram."*

Work around the box. Who uses this system? What external systems does it integrate with?

In the running example (Pagebound, from the Event Storming an Architecture post) C1 looks like this as a list:

- The system (centre box): "Pagebound." An online independent bookshop with a customer-facing web app, internal warehouse tooling, a payment integration, a carrier integration, and a tax integration.
- Users:
  - Customer: the book buyer. Uses the web app to browse, add to cart, check out, track delivery, and request returns.
  - Warehouse operator: the picker and packer. Uses internal tooling to work pick tasks and print shipping labels.
  - Support agent: handles returns, refunds, lost-parcel tickets, and one-off customer questions.
  - Buyer: Pagebound's internal book buyer. Decides which titles to stock; their tooling is part of the overall system the session is scoping.
- External systems:
  - Stripe: payment processing and stored payment methods. Pagebound sends charge and refund requests; Stripe sends back payment events and webhook callbacks.
  - Carrier API (Royal Mail, DPD, or similar): posts parcel events back to Pagebound (scanned at hub, out for delivery, delivered). Pagebound hands parcels off at the warehouse and then tracks them via webhook.
  - SendGrid: transactional email. Pagebound sends email requests for order confirmations, receipts, and shipping updates.
  - Tax API (Avalara or similar): VAT calculation at checkout time. Called synchronously during order confirmation.
  - Identity provider (Auth0, Okta, or similar): authentication for customers and internal staff.

Arrows labelled with what crosses them: *"browses, buys"* from Customer to the system; *"charges and refunds"* from system to Stripe; *"payment events"* from Stripe back; *"hands parcel over, receives tracking"* between system and carrier; *"sends email"* from system to SendGrid; *"calculates tax"* from system to tax API; *"authenticates users"* from system to the identity provider. Ten to fifteen arrows in total. No internal structure. No databases. No services.

This whole diagram should fit on one piece of A3 or one whiteboard panel and be readable at a glance by someone who's never seen the system before. If it doesn't fit, you're drawing too much detail.

What to say when someone wants to draw more:

> *"Everything inside our box goes on the next diagram. This diagram has to be readable by someone who doesn't work here. Every box we add to this level raises the bar for reading it."*

What to watch for:

- Drawing internal structure. Someone starts putting the web app, the API, and the database inside the central box. Stop them. *"That's C2. We're not there yet."* It happens in every first session.
- Forgetting the humans. A C1 diagram without actors is almost always wrong; even a fully automated pipeline has a human operator somewhere. Prompt: *"Who uses this? Who watches it when it breaks?"*
- External systems that are really internal. *"Stripe is just an internal dependency."* No: Stripe is external because you don't control its API. If its shape changes, you have to react. That's the test.
- The codename trap. The central box is called *"sub-svc-v2"* and nobody outside the team knows what that means. Rename it. This is the diagram outsiders read.
- Arrows without labels. Unlabelled arrows are useless. Every arrow should say *what crosses*, not just *that things cross*.

Phase 3: Draw the Container diagram (C2), 45 min

This is the workhorse diagram and the one the session spends most time on. Zoom into the central box. Everything that was behind the one big rectangle on the C1 now has to be drawn explicitly: the web app, the API, the databases, the workers, the queues, the scheduled jobs.

Open with:

> *"Now we open up the box. Every container is something that runs, stores state, or holds data: a web app, an API, a database, a worker, a queue, a scheduled job. If it's deployable on its own, it's a container. If it's a library inside something else, it's not; that's the next level down. We're aiming for a diagram that fits on one page, that the team would pin on the wall, and that answers 'what runs, what stores state, and how do they connect' in about ten seconds."*

If an Event Storming wall is available, this is where it earns its keep. Walk the wall and name the bounded contexts. Each bounded context typically becomes one container, or one container plus a datastore. The team has already done the hard work of deciding where the boundaries are; the C2 diagram pulls that decision into a single readable artefact.

In the running example, the C2 diagram (informed by the Event Storming session) looks like this:

- Customer web app: single-page application running in the buyer's browser. Calls the Commerce API over HTTPS.
- Warehouse web app: internal tooling for pickers and packers. Calls the Fulfilment API and prints labels.
- Support web app: internal tooling for support agents. Calls the Commerce API and the Support API.
- Commerce API: the primary HTTP API the customer web app talks to. Owns the Order bounded context. Publishes events onto the event bus. Reads and writes the order datastore.

- Payment worker: a long-running service that owns the Payment bounded context. Subscribes to checkout-submitted events; calls Stripe; calls the tax API; publishes payment-captured, payment-failed, and refund-issued events back onto the bus. Reads and writes the payment datastore.
- Inventory service: owns the Inventory bounded context. Subscribes to payment-captured events (to reserve stock) and order-cancelled events (to release stock). Exposes a stock-levels read API. Reads and writes the inventory datastore.
- Fulfilment API: owns the Fulfilment bounded context. Subscribes to stock-reserved events to create pick tasks; called by the warehouse web app to record picks, packs, and hand-offs. Publishes order-handed-to-carrier events.
- Delivery tracker: subscribes to carrier webhooks and republishes them as domain events (scanned-at-hub, out-for-delivery, parcel-delivered). Anti-corruption layer (a translation shim that keeps an external system's vocabulary out of yours) between the carrier and the rest of Pagebound.
- Support API: owns the Support bounded context. Called by the support web app. Reads and writes the support datastore. Subscribes to a few events for context (order-cancelled, payment-failed, parcel-lost) so agents see recent history.
- Notifications worker: subscribes to several domain events (payment-captured, parcel-delivered, return-requested) and sends the matching email via SendGrid. No datastore of its own; no domain invariant.
- Event bus: a message broker (Kafka, SNS/SQS, or RabbitMQ depending on the platform). Not an aggregate, not a bounded context. Shared infrastructure that carries domain events between containers.
- Order datastore, payment datastore, inventory datastore, fulfilment datastore, support datastore: five separate datastores, one per bounded context. Each is owned by exactly one service. No shared database.

That's ten containers plus five datastores. The connecting arrows show which services read and write which stores (each store has exactly one writer, a rule worth enforcing on the diagram), which services publish to and subscribe from the event bus, and which services call each other synchronously. The external systems from C1 (Stripe, the carrier, SendGrid, the tax API, the identity provider) appear on this diagram too, at the edges, so the Container diagram is self-contained.

The important thing the diagram makes visible that the Event Storming wall didn't: every bounded context owns its own datastore. The Event Storming session argued for bounded contexts on the basis of consistency boundaries. The C2 diagram makes the operational consequence of that decision visible: if each bounded context owns its own data, each container needs its own store, and cross-context data flows through events, not through shared tables. That's a foundational decision, and it belongs on the diagram where the whole team can see it.

What to say at the container boundary argument:

> *"If this is one container, it deploys as one unit, scales as one unit, and fails as one unit. If that's wrong for any of those three, it's two containers."*

What to watch for:

- The org-chart diagram. Containers land exactly on team ownership rather than technical responsibility. *"If Team A disappeared tomorrow, would this still be the correct container?"* If no, the diagram is an org chart, not an architecture.
- Components leaking into the container level. Someone starts drawing individual classes or packages inside a container box. *"That's C3. Park it and we'll decide in Phase 4 whether this container earns a zoom-in."*
- Multi-writer datastores. Two services drawn writing to the same database. This is *database integration*, one of the oldest integration styles and, unfortunately, one of the most common. It's a well-known anti-pattern (Fowler wrote about it in *Patterns of Enterprise Application Architecture* two decades ago, and it's still everywhere) because it couples the two services through a schema neither of them owns, makes migrations terrifying, and turns every shared table into a distributed transaction problem. But it does exist, and you need to decide what to do about it on the diagram. Three possibilities: *(1)* it's a genuine mistake and one of the services should stop writing; pink note it and schedule the fix. *(2)* The "one database" is really two logical stores sharing a physical engine, and the diagram should show two container boxes even if they run on the same PostgreSQL instance; draw it as two containers with a note explaining the co-location. *(3)* You've inherited database integration from a legacy system and you're not going to fix it this quarter; name it on the diagram as a deliberate, acknowledged compromise, ideally with a link to the migration plan. What you can't do is draw it cleanly and pretend it's fine.
- The hidden container. A scheduled job or cron that runs production-critical work and isn't on the diagram. *"Who runs the invoice generation? Is that on a schedule? Where does the schedule live?"* Schedules are containers too.
- Infrastructure masquerading as containers. A load balancer is usually not a container in C4 terms; it's deployment topology, and it belongs on the deployment view. A CDN isn't a container. Don't pollute C2 with things that will make the diagram stale the first time someone migrates a load balancer.
- Too many arrows. If the Container diagram has more than about twenty-five arrows, it's become unreadable. Either the system genuinely has too many connections (in which case the *system* has a problem, not just the diagram), or you're showing too many concerns at once. Consider drawing a second C2 focused on one flow.

Phase 4: Zoom into Components (C3), selective, 30 min

Component diagrams are optional. This phase exists to *decide* which containers deserve one, not to draw C3 for every container on the wall.

Open with:

> *"We are not going to draw C3 for every container. Most containers are simple enough that C2 plus the code is enough. We're going to look at each container on the wall and ask: does a developer opening this repo need a diagram to understand its internal shape, or is reading the code enough? If reading the code is enough, we don't draw C3. If the container is complicated enough that the shape gets lost in the code, we draw one."*

Go round the containers. For each one, ask:

- Does this container have more than one bounded concern inside it, or is it a single-responsibility service?
- Does a developer joining the team need a diagram to find their way around, or is the repo structure enough?
- Is the internal structure something the team argues about, or is it settled?

A useful rule of thumb: a container earns a C3 diagram if it contains three or more aggregates from the Event Storming session, or if it has significant internal complexity that isn't obvious from the top-level package layout.

In the running example, the Commerce API is the container most likely to earn a C3 diagram. It owns the Order bounded context, which contained several aggregates (Cart, Order, Promotion) and several subtle state-machine rules in the Event Storming session. A C3 diagram of the Commerce API would show components like:

- Cart: the aggregate that manages what a customer is considering buying. Line items, totals, discount codes. Throws away state when the cart converts to an order.
- Order Lifecycle: the aggregate that manages the order state machine (pending, confirmed, cancelled, delivered). The rules for legal transitions live here.
- Catalogue Client: the read model of available titles, stock status at a glance, and pricing. Backed by a cached snapshot of inventory so page loads don't hit Inventory on every request.
- Promotion Engine: the component that evaluates discount codes and loyalty offers at cart and checkout time.
- Event Publisher: a shared component that publishes domain events onto the event bus with consistent envelopes.
- HTTP Adapter: the component that exposes the HTTP endpoints and translates between HTTP and the domain model.

Six components, grouped by responsibility. Each one corresponds to a package or module in the codebase; each one would be recognisable to a developer opening the repo. The C3 diagram makes the shape legible without forcing someone to read every file.

The Payment worker probably doesn't earn a C3; it has one main aggregate (Payment), an anti-corruption layer for Stripe, and a thin event-handling layer. The code is simple enough that the container-level box plus the Event Storming wall is sufficient.

The Support API almost certainly doesn't earn a C3; it's CRUD over tickets with a few event subscriptions. Drawing C3 for it would be busywork.

The Inventory service might earn a C3 if the reservation logic is non-trivial: multi-warehouse reservations, time-boxed holds for abandoned carts, reconciliation with physical stock counts. If it's a single store with straightforward increment/decrement, skip it.

Decide explicitly, for each container, whether it earns a C3 or not, and write the decision on the whiteboard. *"Commerce API: yes. Payment worker: no. Inventory service: maybe, revisit next session. Support API: no."* The *maybe* is a valid answer; it means "come back to this if we later decide we need it."

What to watch for:

- The completionist. Someone insists every container needs a C3 diagram. *"Every container drawn at C3 is another diagram that needs maintaining. Which containers actually have the complexity?"* Saying no is a feature of this phase.
- The architect's ivory tower. One person wants to draw C3 diagrams for containers they don't actually work in. *"The person who owns the code owns the diagram. If you don't touch this container, you don't draw its C3."*
- Components that are really classes. A C3 component is a cluster of related classes, not one class. If the C3 diagram has thirty components, the level is wrong; that's a C4 Code diagram, and your IDE draws it for you.
- C3 components that cross bounded contexts. If a C3 "component" is actually two concerns in one box, split it. If it's doing work that belongs in another container, the container boundary is wrong; go back to C2.

Phase 5: Review and name-check, 20 min

Walk the completed diagrams with the group. Ask three questions in order:

> *"Does this match reality? If we pushed to production right now, would this diagram be correct? Anywhere it's wrong?"*

> *"What's missing? Is there anything running in production that isn't on this diagram? A cron, a one-off script, a manually-triggered job, an integration we forgot?"*

> *"Where did we drift from the Event Storming wall? Are the container boundaries we drew here the same as the bounded context boundaries we drew there? If they're different, why?"*

The third question is the one that earns the session its keep. Drift between the Event Storming wall and the C2 diagram is information: either the wall had a boundary wrong, or the act of drawing containers has surfaced a consequence nobody noticed. Both are worth naming. Neither is a disaster, but both deserve a note on the diagram and a follow-up.

Mark any discrepancies on the whiteboard in a different colour. These become the pink-note equivalent of the C4 session: open questions, follow-ups, things to investigate.

What to watch for:

- "Yes, it matches reality" said too quickly. Prompt for something specific: *"When was the last deploy that changed the shape of this? Did anything in that deploy move the lines we just drew?"*
- The missing scheduled job. Crons and schedules are the most commonly forgotten containers. Ask explicitly: *"What runs on a timer that isn't on this diagram?"*
- The forgotten integration. An external system the team uses often enough that nobody thinks of it as an integration any more. *"What do we call when we want to know the current exchange rate?"* is the kind of question that surfaces these.
- Silent drift from the Event Storming wall. If nobody can name a point where the diagrams diverge from the wall, either you've matched perfectly (unlikely on a first session) or the room isn't examining carefully. Name it: *"If we found nothing to reconcile, we probably didn't look hard enough. Anyone want to challenge a single boundary?"*

Phase 6: Wrap-up, tooling, owners, 15 min

The whiteboard drawings are the first-pass artefact. They are not the durable version. Before the session ends, pick the tool the team will commit to for the durable version, and pick the owner.

Say it directly:

> *"The whiteboard is going to be photographed and the photograph is going to decay. The durable version has to live somewhere the team actually looks. Before we leave this room, we are picking the tool, picking the owner, and picking the date the durable version will exist."*

Options (pick one; the session should close with a single answer):

- Structurizr: Simon Brown's own tool. DSL-based, strongest semantic match to C4 because it was built for C4. Paid for teams but has a free Lite version.
- draw.io / diagrams.net: free, widely used, renders in most wikis. No C4 semantics but has C4 shape libraries.
- Excalidraw: free, fast, great for collaborative whiteboarding. Low ceremony. Best if the team values "easy to update" more than "semantically precise."
- Mermaid with the C4 plugin: text-based, renders in most Markdown tools, version controls with the code. Best if the team lives in Markdown.
- PlantUML with C4-PlantUML: text-based, long-established, widely tooled.

Pick one. Pick an owner: a named person who will produce the durable version. Pick a deadline: typically within a week of the session. The owner is probably the tech lead or a senior developer; occasionally it's a technical writer if you have one. It is not "the team"; diagrams owned by the team as a whole are owned by nobody.

Finally, agree the maintenance cadence. C4 diagrams decay fast. Pick one of:

- Review at every architecture-affecting PR
- Review monthly
- Review at each retrospective
- Review whenever the next major change lands

Pick the one the team will actually do. Writing "monthly" on a plan the team will never execute is worse than writing "at the next big change" and honouring it.

What to watch for:

- No owner decision. The session closes with "we'll figure out the tool later." Don't let it. The durable version won't exist.
- Tool bikeshedding. Fifteen minutes arguing Excalidraw versus draw.io. Pick one, move on. The team can migrate later if the choice was wrong; the cost of migrating is lower than the cost of no diagram.
- "We'll update it when we need to." That's not a cadence; it's a hope. Pick a trigger.

A worked example

The running example in the phases above walks a three-hour session on Pagebound, the same system used as the running example in the Event Storming an Architecture post. In that earlier session, the team settled on five aggregates (Order, Payment, Inventory, Fulfilment, Delivery) plus a Notifications

adapter, and drew boundaries around them. In this session, those aggregates become five bounded contexts which become ten containers plus five datastores on the C2 diagram, one of which (the Commerce API) earns a C3 zoom-in showing six components. The session closes with the team picking Structurizr as the durable tool, the tech lead as the owner, and a cadence of "review at every architecture-affecting PR, plus a full re-walk at each quarterly planning retrospective."

The most useful moment of the session isn't any of the diagrams. It's Phase 5, when the group compares the C2 diagram against the Event Storming wall and notices that the Delivery aggregate has no separate container; it's been folded into the Fulfilment API because, in practice, Pagebound's "delivery" state is entirely reflected from the carrier's webhooks, with no persistent domain state of its own beyond a thin status record per parcel. Someone asks whether that's correct. The answer is *"probably, for now, because Delivery is almost pure projection, but if we ever add our own last-mile logistics or start holding returns-in-transit state, we'll split it."* That exchange gets captured as a note on the diagram: *"Delivery folded into Fulfilment API; split if we start owning any last-mile logistics."* Six months later, when Pagebound pilots a same-day courier partnership with its own tracking stack, that note is the thing that reminds the team to split cleanly instead of piling more into Fulfilment. The diagram didn't just document a decision; it documented the *boundary* of the decision, and flagged the trigger that would change it. That's what a living C4 diagram earns its keep with.

What Can Go Wrong

Named failure modes. Each has a symptom, a recovery move, and a threshold where you stop rather than limp through.

The org-chart diagram. The C2 diagram's container boundaries land exactly on team ownership lines. Billing is a container because the billing team exists, not because the domain says so. *Recovery:* Name it out loud. *"We're drawing the org chart. Let's put the teams aside and redraw from the domain; we'll argue ownership afterwards."* Go back to the Event Storming wall if one exists; use the bounded contexts as the reference, not the team structure. *Stop if:* A second attempt produces the same org-chart shape. The team boundaries and the domain boundaries have diverged in the real system, and that's an organisational problem a C4 session cannot fix. Capture the divergence as a finding and escalate it outside the session.

Drawing C3 for every container. The session is halfway through Phase 4 and the group is insisting every container needs a component diagram, including the ones that are obviously too simple. *Recovery:* Call the rule explicitly. *"A container earns a C3 if a developer needs a diagram to find their way around it. Let's walk the containers and pick the ones that pass the test."* Veto the rest. *Stop if:* The group refuses to accept "no C3 here" as an answer. That usually means someone in the room is treating C4 as a completionist exercise, not a communication artefact. End Phase 4 early; what C3 diagrams you have are better than the ones you'd draw under pressure.

The ASCII art temptation. The team decides the whiteboard snapshot *is* the canonical diagram. Someone takes a photo and pastes it into Confluence and declares the work done. *Recovery:* Block the photo-as-canon decision at the session wrap-up. *"The photo is the source material. The durable version has to be editable, version-controlled, and readable at full resolution. Which tool do we commit to before*

we leave?" *Stop if:* The group refuses to commit to a tool in the session. The durable version won't be created afterwards either. Book a follow-up session specifically to produce the durable version and name the owner before you leave.

The stale diagram. The session runs fine, the diagram gets drawn, and six months later it doesn't match the system any more. This isn't a session failure, it's a post-session failure, but it starts in the session when nobody owns the diagram. *Recovery:* Not in the session. In the session, the prevention is Phase 6: pick an owner and a cadence. After the session, every architecture-affecting PR should include a diagram update, and the owner should re-walk the diagram at least once a quarter. *Stop signal:* If a team runs a second C4 session less than six months after the first and has to throw out the first diagram entirely, the durable version wasn't maintained, which means the owner model failed, not the notation. Fix the owner before re-running.

The architect's ivory tower. One person is drawing and the rest of the room is watching. The diagram that emerges is one architect's mental model, lightly ratified by nodding. *Recovery:* Pair people up. Hand the marker to a developer, not the architect. *"You write the code in this container. You draw the box. We'll argue the edges."* The architect's job becomes reviewing, not authoring. *Stop if:* Pairing two different people over two phases still produces one-architect output. The session dynamic is broken; end early and reschedule with the architect explicitly briefed that their job is to listen.

Levels mismatch. Someone in the room keeps drawing components when the group is on the container diagram, or keeps drawing infrastructure on the container diagram. They're at a different zoom level from everyone else. *Recovery:* Stop the drawing. Ask the group, out loud: *"What level are we on?"* Get the answer. Say it. *"Components / deployment details go on the next / a separate diagram. For now, everything we draw is at container level."* It will need saying more than once. *Stop if:* The mismatched person cannot hold the level distinction across multiple prompts. They may not have internalised the C4 zoom model yet; they belong in a briefer orientation conversation first, not in the session.

The frozen wall. The Event Storming wall the session was meant to be drawing from turns out to be wrong somewhere foundational, and the team discovers it while drawing C2. *Recovery:* Pause the C4 work. Fix the wall with whoever can. If the fix is small, resume. If it's large, stop. *Stop if:* The Event Storming wall has significant gaps. Reschedule the C4 session after a follow-up Event Storming session. Drawing C4 on top of a broken wall produces a diagram that inherits the wall's mistakes.

Next Steps

The session ends; the work begins.

Facilitator's close-out (same day, 24 hours)

- Photographs of the whiteboard at each level, high-resolution, lit well enough that labels are legible.
- A short text summary listing every container, every datastore, every external system, and every C3 component that got drawn, in the same order the diagrams show them.
- A decision log: which level decisions were made (why C3 for this container and not that one), which tool was chosen, who owns the durable version, what the maintenance cadence is.
- The tool decision, the owner name, and the deadline for the durable version, all written somewhere visible: team channel topic, repo README, or pinned message.

The tech lead's week

The tech lead (or whoever the session named as the owner) carries the week after the session. This is where C4 diagrams most often die: unlike a sprint plan, nothing breaks immediately if the durable version doesn't get produced.

- Produce the durable version. Inside a week, the whiteboard photos become an editable diagram in whichever tool the session committed to. The durable version lives where the team actually looks: in the repo next to the code, in the team wiki, in Structurizr Cloud, wherever the team agreed.
- Walk the durable version with the team. A fifteen-minute review once the durable version exists. Every diagram should look *exactly* like the whiteboard, no sneaky additions, no "while I was drawing this I also redesigned a container." If the durable version doesn't match, it's not the artefact the team committed to.
- Wire it into the change process. Every PR that changes an architecture-affecting file (anything that adds a service, adds an integration, moves a container boundary, or changes an API shape between containers) should include a diagram update. If it doesn't, the diagram starts drifting from reality, and drift is terminal.
- Walk the diagrams with anyone who should have been in the room and wasn't. Adjacent tech leads, the security team, SRE. Their reactions surface missing containers and integrations the original session missed.
- Pin the C2 diagram somewhere physical if there's a team room. A4 is too small; A3 is the minimum. People glance at wall diagrams in passing and notice things they'd never notice in a wiki tab.

Ongoing

- Review the C1 diagram rarely; it changes only when the external shape of the system does, which is a big deal when it happens.
- Review the C2 diagram often: every meaningful architecture change, every new container, every deprecated integration.
- Review C3 diagrams per-container, as part of the normal code review for changes inside that container.
- Regenerate C4 Code diagrams from the IDE on demand. Don't maintain them.
- When two consecutive incidents, onboarding conversations, or architecture arguments surface *the same* drift between the diagram and reality, stop and re-run the relevant level. The diagram is trying to tell you something.

Where to go next in the Workshop series:

- Event Storming an Architecture: the strongest link in the Workshop series. An Architecture session finds the aggregates and bounded contexts; C4 draws them as containers and components. A C4 session run immediately after an Architecture session is a redrawing exercise, turning a wall of sticky notes into a durable artefact while the decisions are still hot. If you only learn two patterns from this series, learn these two and run them together.
- Process Level Event Storming: the event flows from a Process Level session feed directly into C4 dynamic views. One dynamic view per important scenario, drawn with the containers and

components from your C4 diagrams, is often the clearest way to explain "what happens when X?" to a new joiner or a security reviewer.

- Big Picture Event Storming: if you're running a C4 session at the enterprise level (the optional C0 System Landscape view) a Big Picture session is usually the correct predecessor. Big Picture finds the systems; C0 draws their relationships.
- Example Mapping: when a C3 component's rules are unclear, Example Mapping is the pattern that turns the unclear rules into rules-and-examples before any code gets written. A C3 component whose behaviour nobody can state is a component waiting for an Example Mapping session.
- Threat Modelling *(publishes later)*: the C2 Container diagram with trust boundaries drawn on it is one of the canonical inputs to a threat modelling session. The containers, the external systems, the arrows between them, and the data that crosses each arrow are exactly what the threat model needs as its starting picture.
- Architecture Decision Records *(publishes later)*: the decisions a C4 session surfaces (why this container and not two, why this datastore is shared, why this integration goes through an anti-corruption layer) are natural ADR candidates. The diagram shows *what* the architecture is; the ADRs explain *why* it is that way, and they compound over time in a way a diagram on its own can't.

Variants

Post-Architecture session (default). The canonical use case. Run immediately after an Event Storming an Architecture session, while the wall is still up and the bounded-context decisions are still hot. The session is a translation exercise: the wall becomes C1 plus C2 plus, selectively, C3. Three hours, 4–6 people, produces a durable artefact that survives the wall coming down. This is what most teams need, and the rest of this post describes it.

Greenfield (no Event Storming wall). A team is starting a new system and wants to commit a starting architecture to paper before writing any code. Same shape as the default session, but slower; the group is discovering and drawing simultaneously, so expect 4 hours rather than 3, and expect a follow-up session a fortnight later once the first build choices have shown where the diagram is wrong. Don't draw C3 in the first session; the components are guesses until code exists.

Onboarding-only (C1 + C2). A team has a working system but no diagrams, and the immediate pain is onboarding cost. Skip C3 entirely. Two hours, 3–4 people, produces a System Context and a Container diagram pinned to the team-room wall. Cheaper than the full session and the most common variant for established teams catching up on documentation debt.

Compliance-driven (C2 + deployment view). A security or audit request has landed and the team needs a diagram showing what crosses trust boundaries. Spend the bulk of the session on C2 with trust boundaries drawn explicitly, then add a deployment view mapping the containers onto the regions and infrastructure they run on. Three hours, with the SRE or platform engineer as a mandatory participant rather than an optional one.

Enterprise (C0 System Landscape). Multiple systems across an organisation, usually post-acquisition or in a multi-product company. Draw the optional C0 System Landscape view first (which systems exist, who owns each, how they relate) then run a separate C4 session per system that warrants it. A Big Picture Event Storming session is usually the right predecessor.

Remote. A digital whiteboard (Miro, Mural, FigJam, or Excalidraw) with the C4 shape libraries pinned, video call for the conversation. Slightly slower (the rhythm of *"draw a box, place an arrow"* is faster in person, especially at C2 where boundary arguments need to be visceral) but the structure transfers cleanly. Use one shared cursor: only the facilitator places shapes, prompted by the team, to keep the layout legible.

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-c4-modelling/.

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