COMMUNICATING ARCHITECTURE WITH THE C4 MODEL

How the C4 model brings clarity, structure, and shared understanding to software architecture

CONTENTS

IcePanel

ABOUT ICEPANEL

We started IcePanel in 2020 out of frustration with existing products.

- Products that slowed teams down instead of moving forward.
- Products that felt dull instead of delightful.
- Products you were told to use, not excited to use.
- Products optimized for sales, not built for us.

Communicating complex systems is harder than building them.

- Designing the future feels stuck in the past.
- Slow, fragmented, and full of miscommunication.
- One step forward, two steps back.

So we did what builders do and launched IcePanel. We're still early in our journey, focused on building the best tool for software architects.

If you care about architecture, you've likely seen Luca on LinkedIn, YouTube, or at a conference. He has a knack for explaining complex topics in a way that's easy to understand. A perfect fit when we were looking for someone to write an ebook about the C4 model. We've admired Luca's work for a long time and are excited to partner with him to bring this book to life.

Happy reading and stay chill!

CHAPTER 1

WHY THE C4 MODEL

WHY THE C4 MODEL

Every sociotechnical system demands a holistic perspective, one that integrates understanding of business requirements, architectural characteristics, and implementation readiness before a single line of code is written.

Effective software systems are not born from individual components in isolation, they emerge from a clear vision of how the system must behave, evolve, and support the business context.

A shared architectural model becomes essential, not only for alignment among stakeholders but also for steering design decisions with intent and precision.

Traditional modeling languages such as UML were designed with rigor and formality in mind, but their complexity often renders them impractical in fast-moving, agile environments.

For example, a UML deployment diagram may attempt to capture every server, database, and network connection in a system, but the level of detail quickly becomes overwhelming and hard to maintain. As a result, many teams abandon these models in favor of quick “boxes and lines” sketches, diagrams that are easy to draw but prone to inconsistency, ambiguous naming, and mixed levels of abstraction.

These compromises may serve short-term convenience, but ultimately, they weaken the shared understanding that architecture demands.

C4 MODEL

The C4 model addresses these shortcomings by offering a set of consistent, layered abstractions: Context, Containers, Components, and Code. These levels allow teams to zoom in and out of the system in a manner similar to how Google Maps lets you transition from a high-level view of a city to detailed street-level insights.

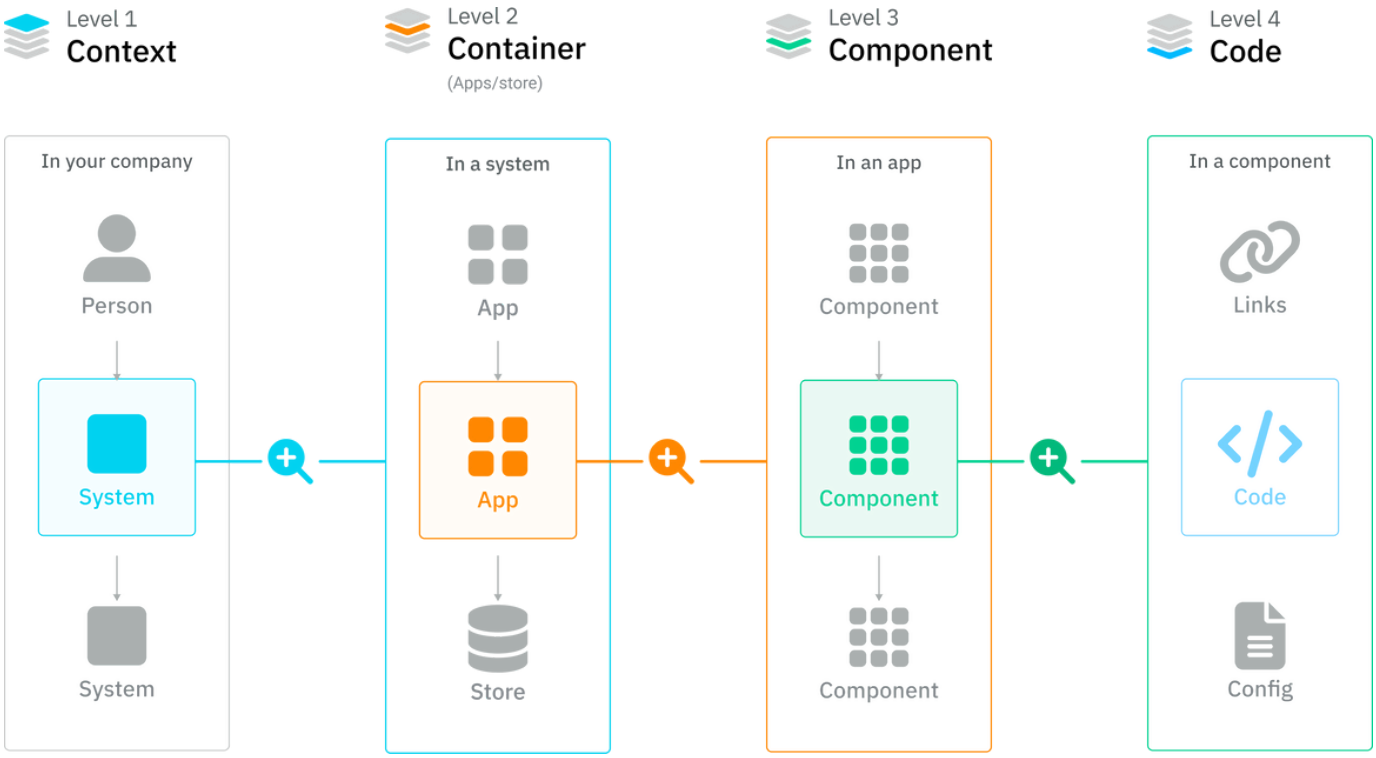
In the words of its creator, Simon Brown, the C4 model is about creating “maps of your code”, diagrams that reveal both where the system lives in its environment and how its internal pieces fit together.

This layered approach is not merely visual, it is communicative.

Executives and non-technical stakeholders can immediately grasp high-level context, such as who uses the system and why. Architects and developers can dive deeper into containers and components, exploring the structure and responsibilities of the system. When necessary, typically during design reviews or development planning, a code-level view such as a UML class diagram offers precision where precision matters.

By tailoring the level of abstraction to the audience, the C4 model ensures clarity for each stakeholder group without compromising consistency or traceability. It reduces ambiguity, promotes more effective documentation, and fosters an architectural vocabulary that spans business, design, and implementation.

THE LEVELS



WHAT THIS GUIDE COVERS

This guide introduces the C4 model as a practical framework for visualizing and communicating software architecture.

It explains how its four levels—Context, Containers, Components, and Code—help teams reason about systems at different levels of abstraction while maintaining consistency and clarity across all views.

Through the modernization of an e-commerce platform, the guide demonstrates how to apply the C4 model in practice, highlighting best practices, common pitfalls, and the benefits of using modeling tools like IcePanel to keep diagrams accurate, connected, and easy to evolve.

Together, these insights show how the C4 model turns architecture documentation into a living, shared understanding across teams.

CHAPTER 2

WHAT IS THE C4 MODEL

WHAT IS THE C4 MODEL

The C4 model is a lightweight framework for visualizing the architecture of software systems. Its purpose is not to replace detailed specifications or low-level diagrams, but to provide a clear, consistent way of communicating structure at different levels of abstraction.

The model is built on a simple idea: a software system can be understood as a set of “maps,” each offering a different level of detail.

At the highest level, you see how the system fits into its environment. As you zoom in, you uncover the internal building blocks, their responsibilities, and their relationships. This layered approach avoids the pitfalls of trying to express everything in a single diagram, while still keeping the different views consistent and connected.

THE LEVELS EXPLAINED

C4 model diagram levels	Scope	Audience
L1. Context The big picture business overview	Person, System	Business, Product, Architects & Developers
L2. Container (App/store) How apps & stores communicate	Actor, System, App, Store	Product, Architects & Developers
L3. Component Building blocks and interactions within an app/store	Actor, System, App, Store, Component	Architects & Developers
L4. Code The technical implementation	Code, Links, Config, Repos	Architects & Developers

The four layers of the model are:

- **System Context:** shows the system in its environment, highlighting users and external dependencies.
- **Containers:** identify the high-level technology building blocks (runnable and deployable units) of the system such as applications, databases, or message brokers.
- **Components:** describes how a container is internally structured into modules or controllers and how those collaborate.
- **Code:** offers a close-up of the implementation level, typically with class or function diagrams, where detail is needed.

This progression creates a structured way to “zoom in” from a broad view of the system to detailed implementation concerns. The value of the model lies not in producing exhaustive diagrams, but in ensuring that each level communicates clearly to its intended audience.

Executives and product stakeholders can focus on context and containers, while architects and developers can rely on components and code for technical design discussions.

By offering just enough formality to stay consistent, yet remaining lightweight enough to be practical, the C4 model bridges the gap between the informal ad-hoc sketches and the rigidity of traditional modelling languages.

CHAPTER 3

THE 4 LEVELS IN PRACTICE

THE 4 LEVELS IN PRACTICE

The strength of the C4 model lies in its ability to describe a system at different levels of abstraction, depending on the audience and the purpose. Each level focuses on a specific story and offers a diagram that complements the others without duplicating them.

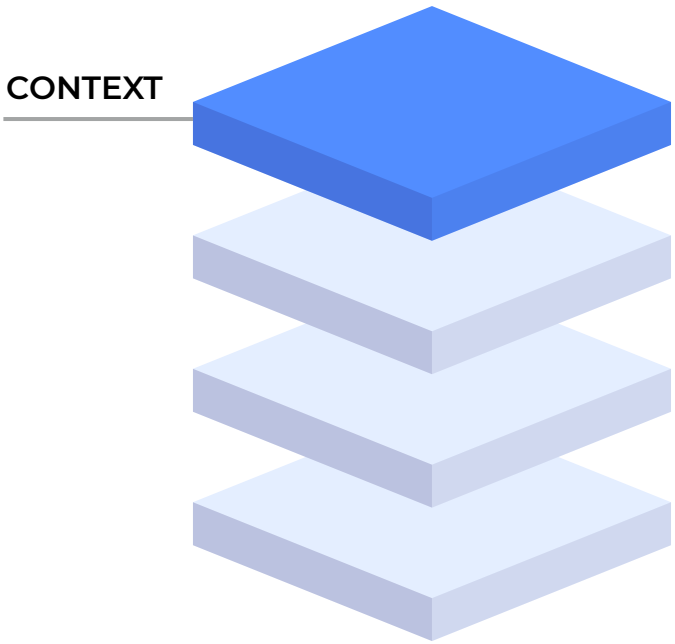
LEVEL 1: CONTEXT

The system context diagram represents the software system as a single box within its environment, showing the people, organizations, and external systems that interact with it.

At this level, the system is treated as a distinct product or service, often owned and maintained by a single team, delivering value to its users while potentially integrating with other systems.

For example, an online retail platform might interact with customers who browse and purchase products, warehouse staff who manage fulfilment, and a payment gateway such as Stripe.

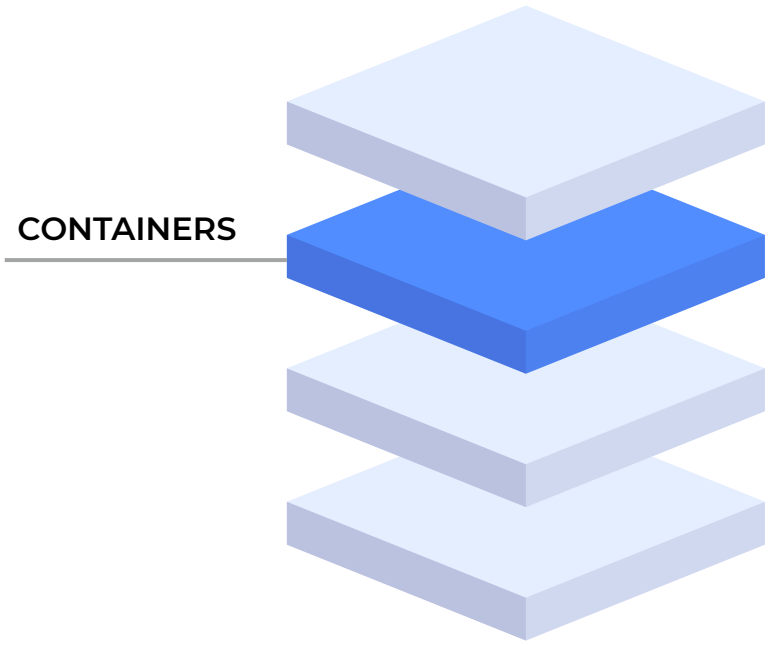
This view emphasizes scope and boundaries, clarifying who uses the system, what external systems it depends on, and how it fits into the broader environment, without delving into internal implementation details.



LEVEL 2: CONTAINERS

A container diagram zooms in to a single system to show the applications, services, and data stores that make up that system, representing the high-level technology choices and deployment units. Each container is a separately deployable and/or runnable application or service with its own responsibilities.

In the online retail example, containers might include a web application, a mobile app, an API backend, a relational database for product and order data, and a message broker to coordinate asynchronous tasks such as sending order confirmations. This view helps architects and technical stakeholders understand how the system is composed, how responsibilities are distributed, and how containers interact with each other and with external systems.

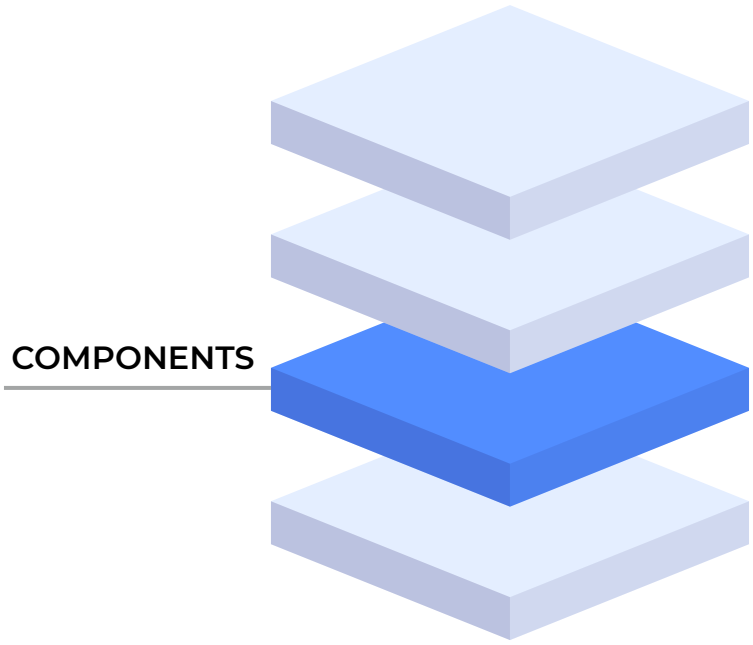


LEVEL 3: COMPONENTS

A component diagram zooms into a single container to show the major building blocks that collaborate to fulfill its responsibilities. Each component has a clearly defined role and interface, making the system easier to understand, maintain, and evolve.

For example, within the backend API container of an online retail platform, the Order Service might be divided into components such as the Order Controller, which handles incoming requests; the Order Processor, which manages business logic and validation; and the Payment Gateway Adapter, which communicates with external payment providers like Stripe.

This view is primarily intended for architects and developers, providing a clear map of the internal structure of a container while avoiding unnecessary detail for higher-level stakeholders.

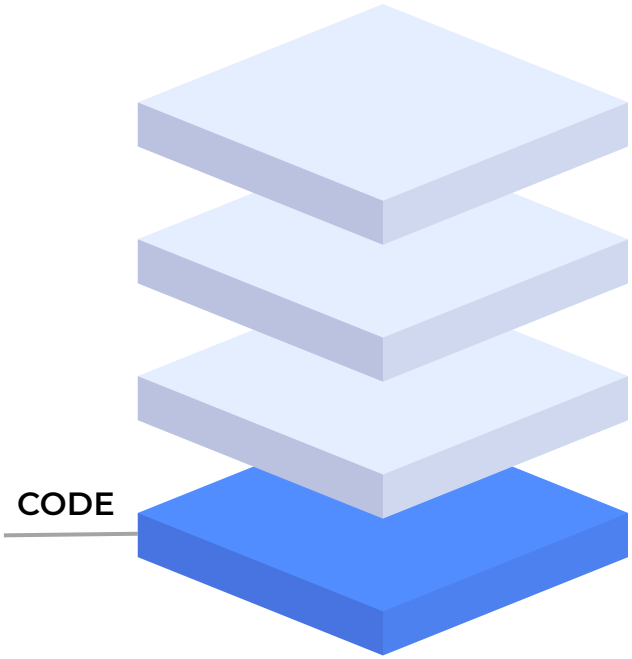


LEVEL 4: CODE

The code diagram zooms into a single component to show its internal implementation using classes, functions, or other programming constructs. It captures how the component fulfills its responsibilities, providing precision where it is needed for development or design discussions.

For example, within the Order Service component of an online retail platform, a code diagram might show classes such as Order, OrderItem, and Customer, along with the methods that orchestrate order validation, payment processing, and inventory updates.

This level is primarily intended for developers, offering insight into the system's implementation while maintaining alignment with higher-level diagrams. It is rarely necessary; many areas of a system can stop at the component level, with code diagrams created only where additional clarity is required. In modern practices, code-level diagrams are rarely used except for onboarding, design reviews, or highly regulated systems that demand precise documentation.



GUIDANCE ON COMPONENTS AND CODE LEVELS

It is important to note that Levels 3 and 4 are not always necessary for every system or every part of a system. In many cases, understanding the system down to the container level is sufficient for most discussions, planning, and decision-making. Teams may choose to create component diagrams only for areas that require additional clarity, such as complex business logic or critical services.

BEST PRACTICES AND TIPS

When using the C4 model, the goal is clarity and effective communication, not exhaustive documentation. The following guidelines can help teams get the most value from this approach:

- **Tailor the level of detail to your audience**
  Not every stakeholder needs to see every level. Executives and product managers typically benefit from system context and container diagrams, while developers and architects may need components and, occasionally, code-level views.

- **Invest time in intention and clarity**
  Before creating diagrams, think carefully about what you need to express. Gather requirements, understand your audience, and translate those into architecture.

  Without clarity of intention, it's difficult to design effective C4 diagrams across the different levels.

- **Use consistent naming and notation**
  Define clear conventions for naming systems, containers, and components.

  Consistent labels, symbols, and colors improve comprehension and reduce ambiguity across diagrams.

- [x] **Start with context and containers**
  Begin with the system context and container diagrams to establish boundaries, dependencies, and high-level responsibilities. Drill down to components only when more clarity is needed. Code-level diagrams are optional and should be reserved for complex areas or specific needs.

- [x] **Iterate and maintain diagrams**
  Architecture evolves over time. Treat diagrams as living documents, updating them to reflect major changes. Avoid spending excessive time perfecting diagrams that quickly become outdated.

- [x] **Leverage tooling and templates**
  Use diagramming tools or templates that support layering and consistency. This reduces effort and helps maintain a standard visual language across your documentation.

CHAPTER 4

THE C4 MODEL CHECKLIST

1. THE DIAGRAM HAS A CLEAR AND DESCRIPTIVE NAME

2. A SHORT DESCRIPTION EXPLAINS WHAT THE DIAGRAM REPRESENTS AND ITS PURPOSE.

3. ALL OBJECTS ARE NAMED CLEARLY, WITH ACRONYMS EXPANDED OR EXPLAINED.

4. OBJECT RESPONSIBILITIES ARE SELF-EXPLANATORY OR SUPPORTED BY DISPLAYED DESCRIPTIONS.

5. RELATIONSHIPS BETWEEN OBJECTS (CONNECTIONS) ARE LABELED TO INDICATE INTENT OR FLOW.

6 NOTATION AND SYMBOLS ARE EXPLAINED, IDEALLY THROUGH A LEGEND.

7 SHAPES, LINE STYLES, BORDERS, ARROWHEADS, ICONS, AND COLORS ARE USED CONSISTENTLY AND MEANINGFULLY.

8 OBJECT SIZES ARE APPROPRIATE AND PROPORTIONAL TO THEIR IMPORTANCE OR HIERARCHY.

9 THE DIAGRAM SHOWS THE RIGHT LEVEL OF DETAIL FOR THE INTENDED AUDIENCE.

10 THE INTENDED AUDIENCE CAN UNDERSTAND THE DIAGRAM WITHOUT ADDITIONAL EXPLANATION.

CHAPTER 5

DIAGRAMMING VS MODELLING

DIAGRAMMING VS MODELLING

How many times have you opened a diagram in Confluence or Miro only to realize it was outdated, inconsistent, or missing key changes?

It happens all the time.

Diagramming is great for explaining an idea in the moment, but as soon as the system evolves, they stop reflecting reality. Suddenly you're not just explaining architecture, you're explaining the gaps between the diagram and the real system.

Modelling, by contrast, builds a single source of truth.
A structured representation of systems, containers, components, and relationships. This model becomes the foundation for automatically updating multiple C4 diagrams, ensuring that changes propagate across all views without duplication or drift.

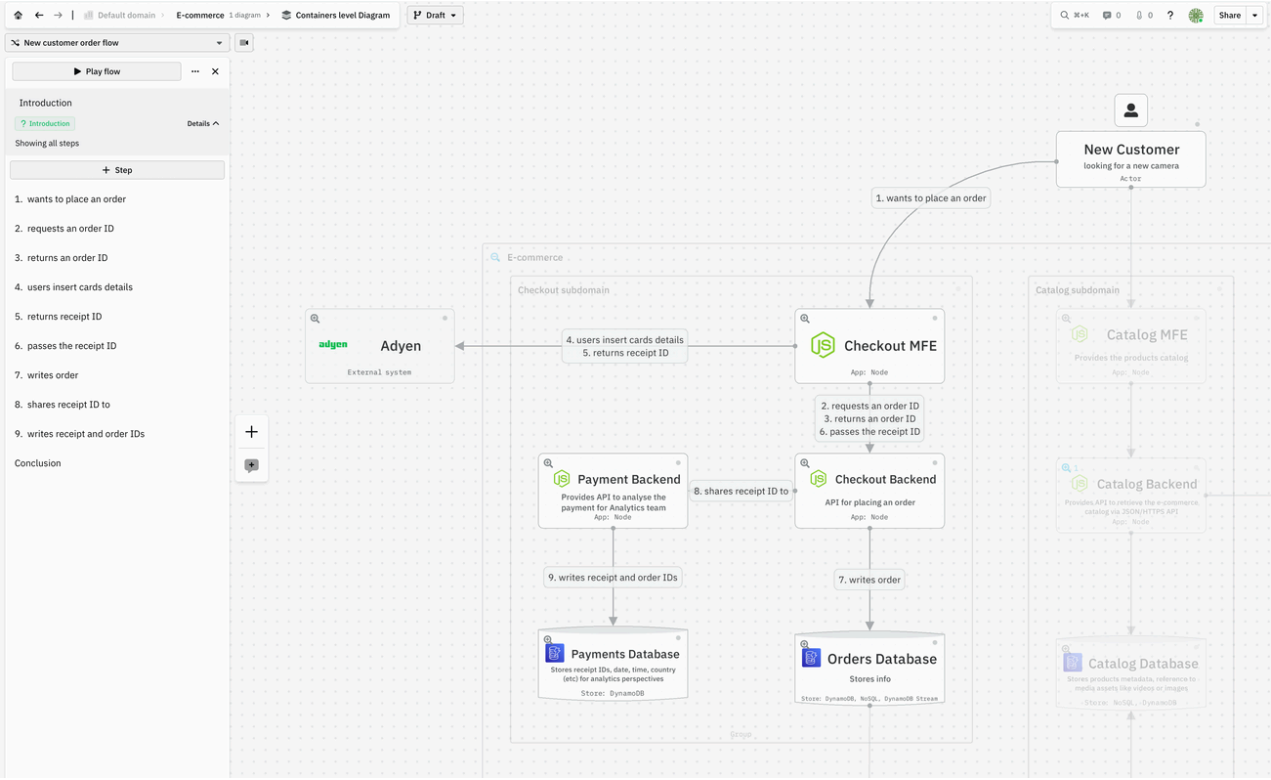
Moreover, modelling allows teams to scale many diagrams across the organization by enforcing consistency and ensuring that the objects in diagrams are up to date.

Imagine changing the name of an external system in the context diagram. In a pure diagramming tool, you must locate and update each diagram where that system appears manually. In a modelling tool, the change happens once in the model, and all diagrams reflect it instantly.

Moreover, the model lets teams query architecture, for example, identifying all components depending on a particular service or extracting insights on deprecated technology stacks.
Modelling does demand more upfront effort such as defining names, metadata, and structure, but it pays off with consistency, discoverability, and agility.

When applied to C4, modelling allows you to author your context diagram, then seamlessly refine into containers, components, and deployment or dynamic views, all connected to the same underlying architecture.

Tools like **IcePanel** embody this modelling-first mindset.
They allow architects to capture architecture as a connected models aligned with the C4 model, ensuring that diagrams remain in sync, dependencies are easier to track, and the architecture evolves as the system does.



Now let's move from the theory to **real examples** so you can see how the C4 model works in practice. I'll use IcePanel to walk through a system and show you how modelling makes architecture documentation reliable, consistent, and easy to evolve.

CHAPTER 6

AN E-COMMERCE MODERNIZATION

CONTEXT AND APPROACH

A growing e-commerce company faced mounting challenges with their legacy monolithic platform. Releases had become slow and cumbersome, making it difficult to respond quickly to customer needs or implement new features.

The system had grown over the years into a tightly coupled architecture, with complex dependencies between modules, making maintenance and scaling increasingly costly. The organization wanted to increase agility, reduce operational complexity, and empower teams to move faster, while ensuring stability and reliability for critical business operations.

After evaluating several approaches, the company decided to modernize using microservices, leveraging internal expertise in Node.js and deploying it on AWS.

A serverless-first architecture was chosen to accelerate releases, reduce the operational burden on the platform team, and enable teams to work autonomously within a federated organizational model. This approach also allowed developers to take advantage of battle-tested cloud infrastructure without needing deep platform expertise, providing both speed and safety in delivering new features.

In this example, we will use the C4 model to visualize the new system. We will explore different layers, from context down to components, while skipping the code-level view, which is rarely needed.

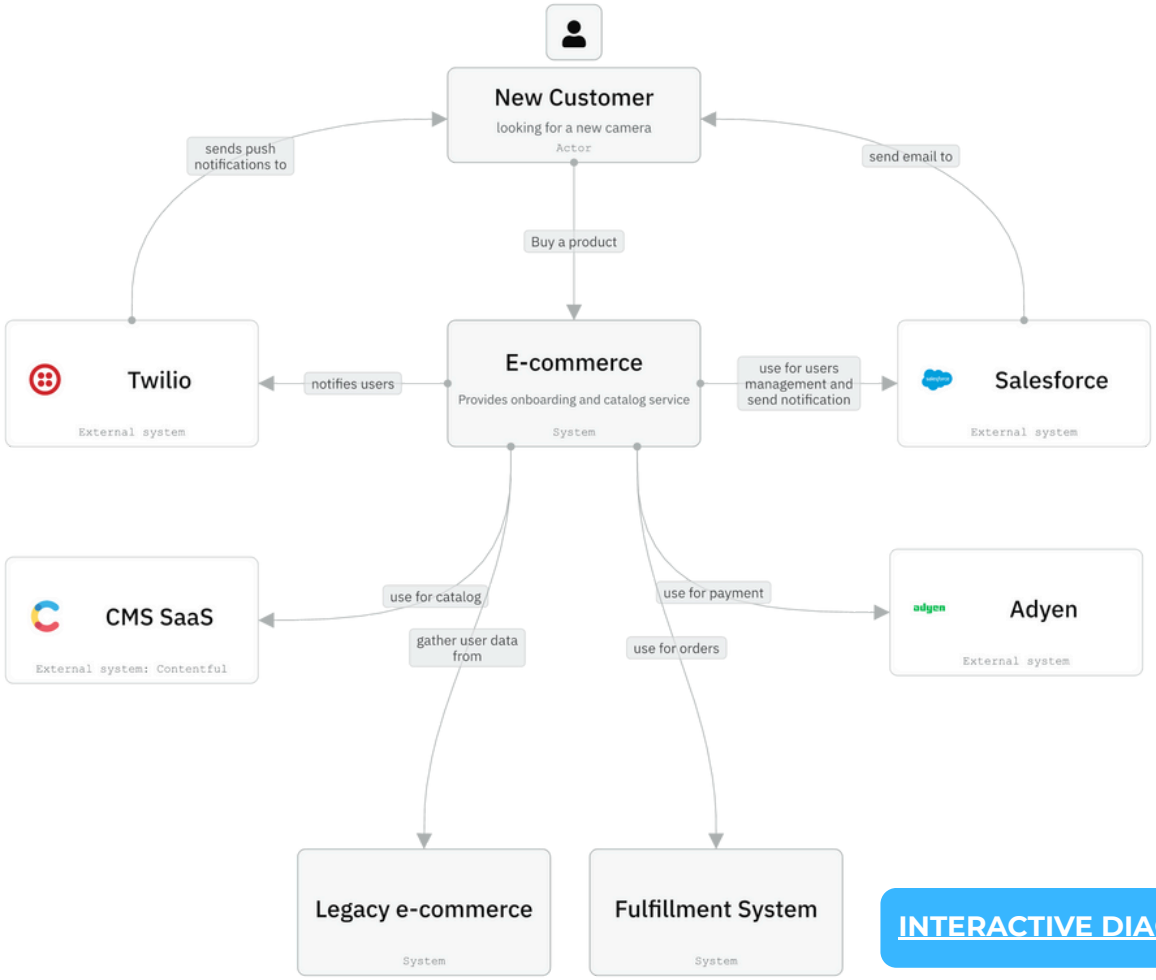
To illustrate the migration journey, part of the system will be shown as already modernized into microservices, while another part will remain in the monolith. Each layer will include concrete examples to provide clarity on how the C4 model can be applied in practice, helping readers understand not just what the architecture looks like, but how to communicate it effectively across different stakeholders.

THE CONTEXT LEVEL

CONTEXT LEVEL

The **system context level** is the highest level of abstraction in the C4 model. At this stage, the system is represented as a “black box” and the focus is not on its internals but on how it interacts with the world around it.
The goal is to show **who uses the system** and **what other systems it communicates with**, providing a clear boundary of responsibilities and dependencies.
At this level, diagrams usually include:

- **People (actors):** end users, customers, or roles that interact with the system.
- **Software systems:** external applications or platforms that the system integrates with, such as payment gateways, notification services, or CRMs.
- **The system itself:** the central box that represents the software we are building or modernizing, like in our case.



INTERACTIVE DIAGRAM

In our e-commerce modernization example, the diagram captures exactly these elements.

At the center, we find the new e-commerce platform, which represents the modernized part of the system built with microservices and micro-frontends.

Around it, several external systems define its environment: Salesforce for user management, Twilio for notifications, Adyen for payments, and a CMS SaaS for managing the product catalogue.
On the fulfilment side, the diagram shows the legacy fulfilment system, a monolithic application that continues to handle shipping and order processing.

On its side sits the legacy e-commerce platform, which remains in use for parts of the business not yet migrated.

By including these systems, the diagram provides a realistic picture of the transitional state: the new e-commerce system already interfaces with both modern external services and existing monolithic components.

AUDIENCE AND USE CASES

The system context level is designed for broad communication. It provides a view that is simple enough for non-technical stakeholders to follow, yet precise enough for architects to use as a foundation for deeper design.

- **Executives and business stakeholders** benefit from this view because it highlights scope and boundaries. They can see who interacts with the system, which external providers are critical to its operation, and how legacy systems remain part of the picture. This allows them to assess risks, plan budgets, and align technology choices with business strategy.

- **Architects and technical leads** use this view early in discussions to frame the system's boundaries and external dependencies before diving into containers or components. It ensures the entire team starts with the same understanding of what the system is and what it is not.

This level is most valuable when you need to step back and see the big picture, as Simon Brown puts it. In distributed systems, new features often span multiple domains, requiring conversations that align business and technical stakeholders toward the same goal. A context diagram provides a shared view. It is equally useful when evaluating new third-party integrations, since it shows how external systems fit within existing boundaries and responsibilities. Finally, this abstraction is particularly effective for onboarding new team members, giving them a clear understanding of the system's scope, its users, and its key dependencies before they dive into lower-level details.

In our case, the context diagram ties directly back to the business goals driving the modernization. The company's push for agility and faster releases is reflected in the choice of external services like Twilio, Salesforce, and Adyen, which reduce the need for building these capabilities in-house. The continued presence of legacy fulfilment and e-commerce systems shows how the organization is managing transition and risk, ensuring critical operations remain stable while new capabilities are delivered.

In other words, the boundaries drawn in the context diagram are not arbitrary: they mirror the business priorities of faster delivery, reduced operational burden, and a clear path to modernization without disrupting ongoing sales and fulfilment.

COMMON PITFALLS

Teams often run into a few recurring mistakes when creating system context diagrams. A frequent one is mixing internal containers with the system context, for example, showing APIs, databases, or services that belong to the internals of the platform.

This blurs the abstraction and makes the diagram harder to interpret.

Another pitfall is adding excessive technical detail, such as protocols or infrastructure choices, which distracts from the purpose of this level. Finally, inconsistent naming of actors and systems, for instance, alternating between “Customer” and “End User” in the same diagram, can create confusion and weaken the shared understanding the diagram is meant to provide. Specifically, this problem can be easily solved with modelling tools like IcePanel that provide a simple way to drag and drop existing elements in different C4 model layers.

PRACTICAL TIPS

To make the most of this level, aim to keep diagrams simple and readable, focusing only on people, systems, and their relationships. Label relationships clearly, but avoid technical jargon that only a subset of the audience will understand. Remember, these diagrams might be helpful in executive presentations, strategic planning sessions, and high-level roadmaps, where clarity on boundaries and dependencies is most important. Treat it as a communication tool first and foremost: its strength lies in aligning diverse stakeholders around a shared view of the system’s place in its environment.

THE CONTAINERS LEVEL

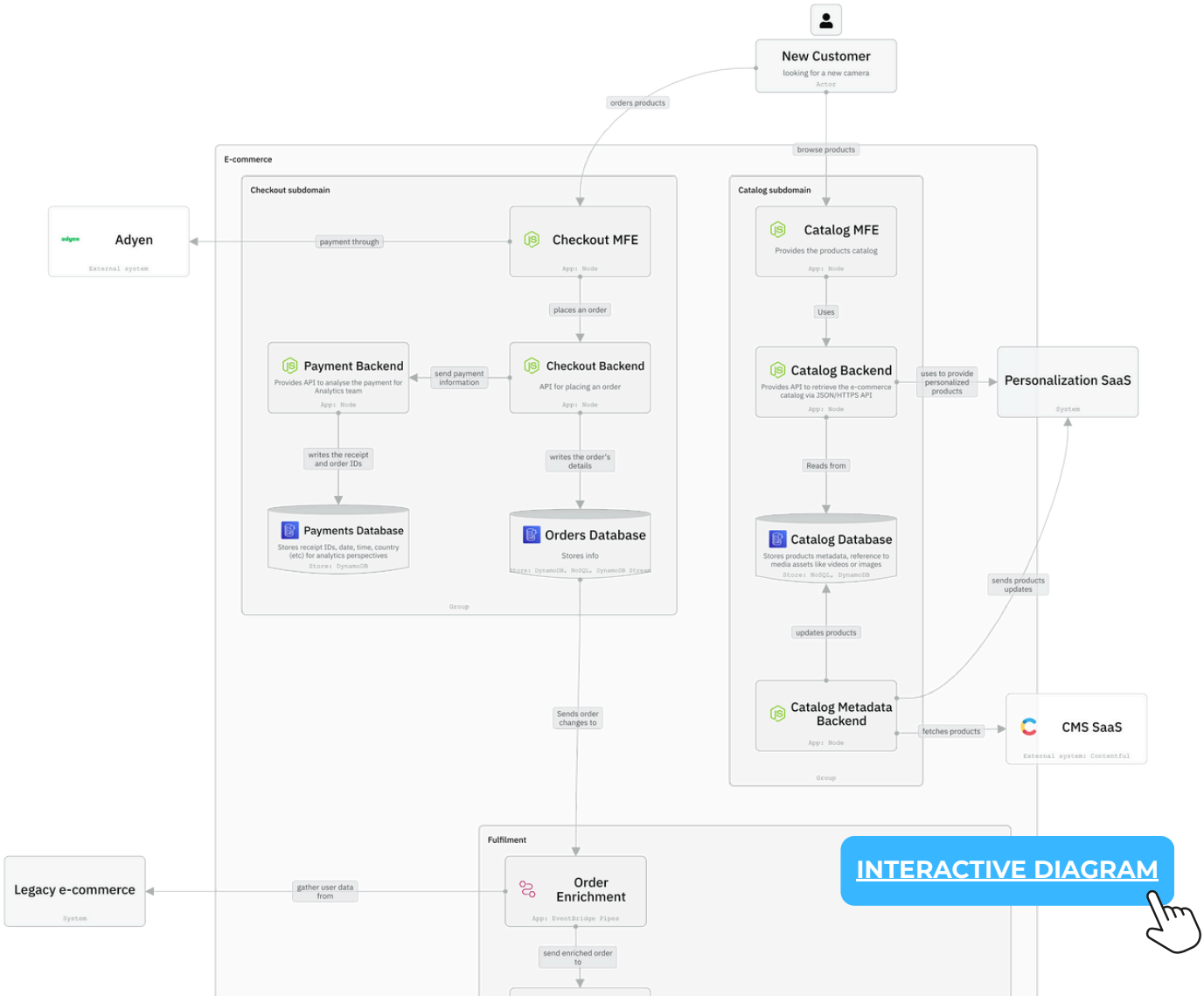
CONTAINERS LEVEL

When we step from the System Context into the **Containers level**, the C4 model asks us to represent the **major runtime building blocks** that make up our system. A container is any application or data store that requires its own runtime or execution environment. It can be a web application, a mobile app, a microservice, a database, or even an external SaaS dependency.

This definition is crucial: a **container is not a Docker container**.

In C4 terminology, the word “container” comes from the idea of “containing” code or data. Each container is something you can deploy, operate, and monitor independently.

This view is especially valuable because it balances **abstraction with technical concreteness**. A system context diagram might simply state that “the e-commerce system integrates with a payment provider,” but at the container level, we learn *how* this is done.



CATALOG SUBDOMAIN

The Catalog subdomain is responsible for surfacing products to customers. In C4, we represent this subdomain as a group of containers, each with a clear runtime boundary:

- **Catalog MFE (Micro Frontend)** A frontend application written in Node.js, deployed independently, and responsible for rendering the product catalog in the browser. In C4, this is represented as a container of type application, annotated with the technology (Node.js) and responsibility ("Provides the products catalog").

- **Catalog Backend** A backend-for-frontend (BFF)-style API that retrieves products and serves them to the frontend. Again, this is a separate container, deployed in its own runtime, exposing a JSON/HTTPS API.

- **Catalog Database** A persistent data store—modeled as a container because it runs independently and has a clear runtime (NoSQL + DynamoDB). In the C4 model, databases are first-class citizens: you treat them as containers because they require deployment, scaling, and operational considerations. We annotate them with their technology (MongoDB, DynamoDB) and their purpose ("Stores products metadata, references to media assets like videos or images").

- **Catalog Metadata Backend** A service responsible for synchronizing product information from an external CMS (Contentful) and pushing updates into the Catalog Database. It also forwards data to the Personalization SaaS. This is a textbook example of a microservice container: it runs independently, has a well-defined API surface, and a clear responsibility.

By modeling each of these as containers, we make visible not just the components of the catalog but also their interactions: the MFE depends on the Backend, which queries the Database; the Metadata Backend synchronizes data from external SaaS providers.

CHECKOUT SUBDOMAIN

The Checkout subdomain is responsible for order placement and payment processing. Again, each container is a runtime unit:

- **Checkout MFE** A frontend container providing the checkout user interface. It orchestrates calls to the backend and external payment provider.

- **Checkout Backend** The API for placing an order. Represented as a container because it is a deployable Node.js service exposing an HTTPS interface.

- **Payment Backend** A dedicated container for handling payment processing. It abstracts the integration with Adyen and persists receipts. In C4, it's best practice to separate such concerns into their own containers so they can evolve independently.

- **Orders Database** Another database container, implemented using DynamoDB and DynamoDB Streams, storing order details and emitting changes for downstream processes (e.g., fulfilment).

- **Payments Database** A DynamoDB table for storing receipt IDs, dates, amounts, and other payment metadata. Annotated as a data store container, with a clear responsibility and technology tag.

Notice how this view clarifies that the checkout flow is not one monolithic backend but a set of cooperating containers, each with their own runtime and responsibility.

WHY THE CONTAINERS LEVEL MATTERS

The container diagram is particularly useful for audiences who need a deeper technical understanding without drowning in implementation details:

- **Architects and senior engineers:** It helps them reason about system boundaries, identify ownership, and ensure that the architecture supports business capabilities like scalability, resilience, or compliance.

- **Developers new to the system:** It serves as a map, clarifying where responsibilities lie, what services exist, and how data flows between them. Instead of a flat diagram with dozens of boxes, containers provide structure through logical groupings.

- **Product owners and technical stakeholders:** By tying containers to subdomains and teams, they can better understand how responsibilities are divided and how external services fit into the system.

At this level, one of the key benefits is **making trade-offs explicit**. For example, persisting orders in DynamoDB with streams attached to EventBridge is not just a technical implementation choice; it expresses priorities like scalability, reliability, and eventual consistency. Similarly, introducing a buffer through SQS queues shows how the architecture anticipates high traffic without overwhelming downstream services.

CONTAINERS AS A FOUNDATION FOR TEAMS

Finally, the containers level has an organizational dimension. By structuring the e-commerce platform into subdomains and associating a container or group of containers with teams, we make the team-to-architecture mapping explicit. This is especially important in modern microservice and micro-frontend architectures, where Conway's Law teaches us that team structures shape the system's design.

Being able to point to a container and say "this is the responsibility of the Catalog team" clarifies ownership and avoids ambiguities. It also supports practices like Team Topologies, where boundaries between teams and services must be deliberately designed.

BEST PRACTICES AND GUIDANCE FOR MODELING CONTAINERS

When working at the containers level, it's easy to either oversimplify (drawing only a couple of boxes) or overcomplicate (turning the diagram into a wiring map). The following practices help strike the right balance:

- **Use deployability as your boundary** A container represents something that can be deployed, scaled, and operated independently. If you can run it in its own process, VM, container, or managed runtime, it's a candidate for a container in your diagram. This guideline keeps you from modeling at the wrong level of abstraction.

- **Make responsibilities explicit** Every container should state in plain language what it does. "Provides the products catalog," "Stores order details," or "Processes payments" are clear and accessible even to non-technical readers. Avoid purely technical labels such as "Service A" or "DB1."

- **Always annotate with technology** The C4 model encourages pairing responsibilities with technologies: Node.js – Provides the products catalog, DynamoDB – Stores order details. This helps readers quickly connect what something does with how it is implemented. Over time, this practice also makes migrations or technology choices transparent.

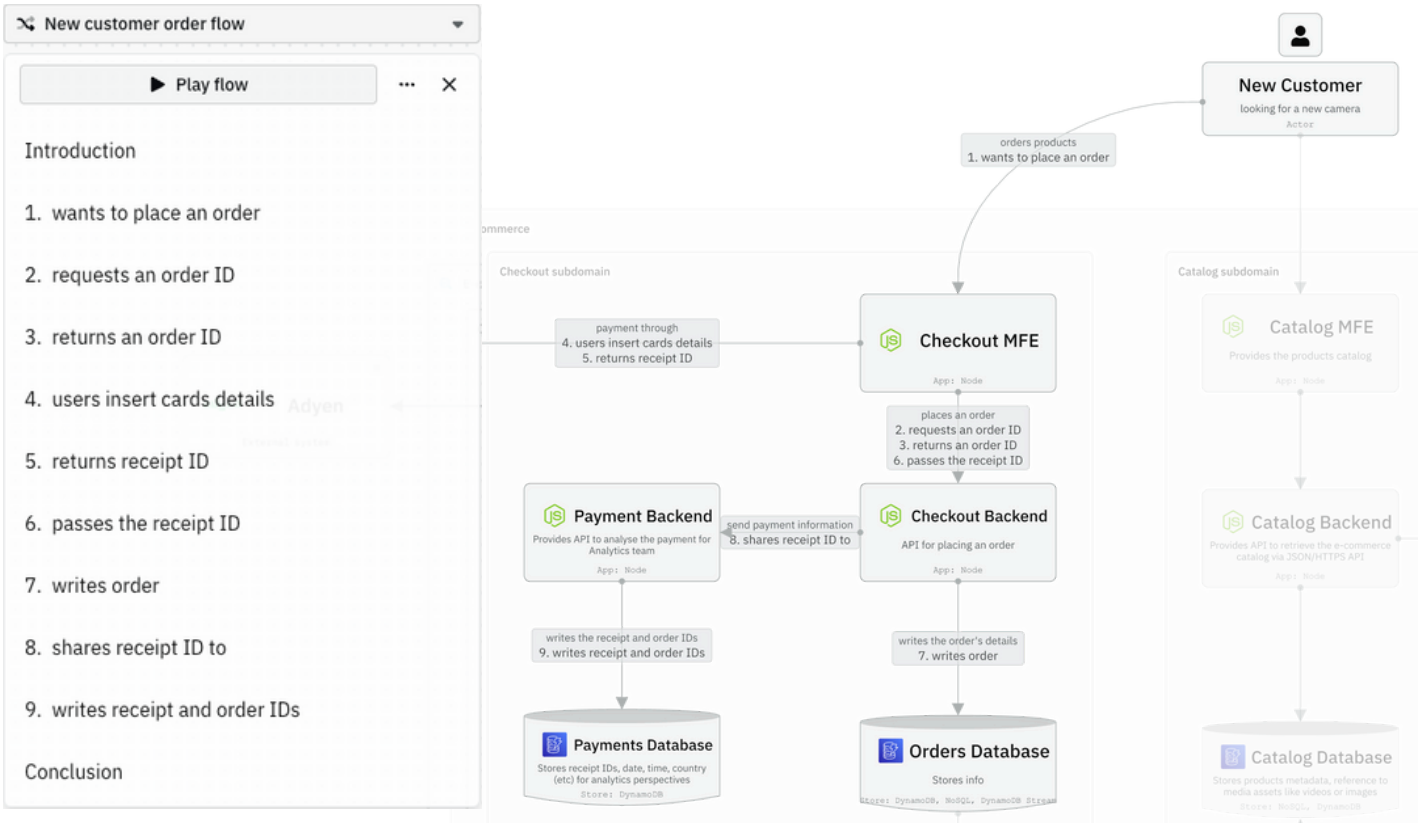
- **Treat databases as first-class citizens** In many diagramming notations, data stores are relegated to supporting roles. In C4, databases are containers because they have runtime behavior, scaling concerns, and operational impact. Modeling them explicitly surfaces trade-offs (e.g., NoSQL vs relational, streams vs queues) that often have architectural significance.

- **Show external systems with equal clarity** SaaS integrations (e.g., Adyen for payments, Contentful for CMS, or a personalization engine) should be modeled like any other container. Even though you don't build or deploy them, they affect availability, resilience, and team responsibilities.

- **Group containers by subdomain or team ownership** Grouping containers into subdomains (as in Catalog, Checkout, Fulfilment) clarifies the business capabilities behind the architecture. Mapping these groups to teams strengthens alignment between architecture and organization — a key principle in modern, distributed systems design

MORE THAN A STATIC PICTURE

One of the powerful features of modern tools, such as IcePanel in our case, is the ability to go beyond static diagrams. The **Flow** capability allows us to illustrate sequences of interactions, such as a customer browsing the catalog, placing an order, and processing payments. This interactive view brings together what is often split between architecture diagrams and sequence diagrams, enabling architects to "play" through a scenario step by step.



INTERACTIVE DIAGRAM

For example, the checkout flow starts with the customer invoking the Checkout Micro-Frontend, which in turn communicates with the Checkout Backend to generate order ID. Payment details are handled via Adyen, receipts are written into a Payments Database, and orders persisted in the Orders Database. From here, downstream processes such as fulfilment and customer notifications are triggered.

Explaining a diagram in a meeting using such features can drastically simplify discussions. Rather than walking through a static drawing, you can “play” the interactions and let the diagram tell the story. Even outside of meetings, simply sharing a link to a C4 model diagram with a new joiner allows them to walk through the system interactions at their own pace. This not only accelerates onboarding but also provides clarity without requiring someone to be physically present to explain the architecture.

THE COMPONENTS LEVEL

COMPONENTS LEVEL

The component level diagram in the C4 model zooms inside a container to illustrate the major building blocks that make up that container and how they collaborate. Each component is defined by its responsibilities and boundaries, rather than its technical implementation. In practice, a component might map to a module, a class, or a cohesive set of functions that collectively provide part of the container's behavior. The emphasis here is on showing how pieces fit together to deliver the container's purpose, without overwhelming detail.

WHEN TO USE IT

Not every container warrants a component-level breakdown. For simple services, a container-level view may be enough to communicate its responsibilities. However, **for complex or business-critical areas**, the component view can be invaluable. Examples include:

- An **order management system**, where multiple workflows (order validation, payment, shipping) interact in subtle ways.
- An **authentication service**, which may include components for credential storage, token generation, multi-factor validation, and auditing.
- A **catalog backend**, where personalization, categorization, and product details must be composed into a cohesive API.

The rule of thumb: use component diagrams where additional clarity helps reduce misunderstandings, speed onboarding, or highlight architectural trade-offs.

WHO IS INTERESTED

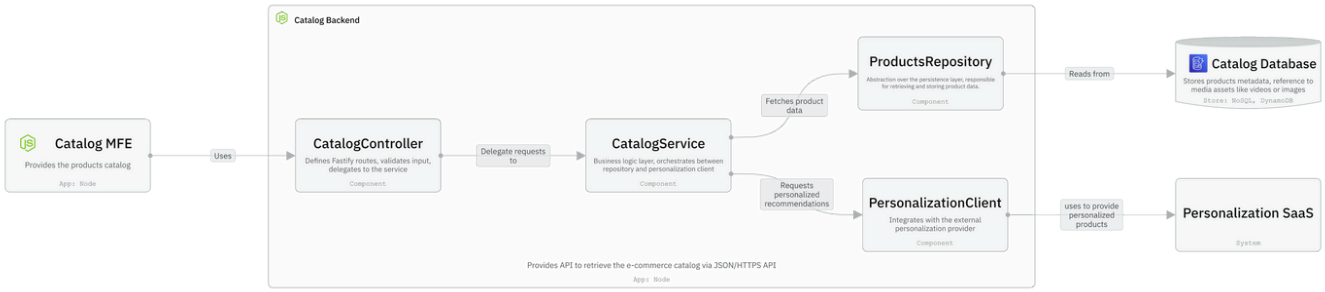
The audience for component-level diagrams is **primarily developers and architects.**

Developers benefit from understanding the internal structure of a service before diving into the codebase. Architects use these diagrams to reason about design choices—whether responsibilities are well-separated, dependencies well-managed, and boundaries clearly defined. Product owners or non-technical stakeholders usually don't need this level of detail.

EXAMPLE: CATALOG BACKEND

Consider a catalog backend service that powers the catalog UI of an e-commerce platform. This container must:

- Retrieve data from the product database.
- Interact with an external personalization system to tailor results.
- Expose APIs for different catalog views: newest products, bestsellers, personalized recommendations, and product detail pages.



INTERACTIVE DIAGRAM

At the component level, we might model the following building blocks:

- **CatalogController** Handles incoming API requests from the UI.
- **CatalogService** Orchestrates requests by delegating to the repository and personalization client, applying business rules, and shaping responses.
- **ProductRepository** Provides access to product data stored in the database.
- **PersonalizationClient** Integrates with the external personalization system.

This view highlights not just the existence of a container but the responsibilities and collaboration of its internal parts, making it easier to onboard new developers or reason about future changes.

BEST PRACTICES

To get the most out of component-level diagrams, it is important to strike a balance between detail and clarity. Components should be documented selectively, focusing only on those areas of the system where complexity makes additional structure valuable. The diagrams should remain conceptual, emphasizing responsibilities and relationships rather than drowning in implementation details.

Keeping the number of components per view manageable ensures that the diagram remains readable; if a service grows too large, it is better to create sub-diagrams than to crowd everything into one picture.

Naming conventions play a key role as well—aligning component names with domain concepts, such as *ProductRepository* or *CatalogService*, makes diagrams intuitive for both technical and business audiences.

Finally, component-level diagrams should be treated as guides to design intent rather than exact mirrors of the codebase. When deeper technical precision is required, supplementing the diagram with direct references to source repositories provides a way to stay up to date without overloading the visual representation.

THE CODE LEVEL

CODE LEVEL

The code level diagram is the most detailed step in the C4 model. It drills down into a single component to show the internal structure of classes, interfaces, methods, or functions. At this point, the diagram resembles a UML class diagram or a package view of the source code. The purpose is to give developers a concrete reference for how the internals of a component are organized.

For example, consider a *CheckoutService* component responsible for orchestrating the shopping cart and payment workflow. A code-level diagram might show the *OrderValidator*, *PaymentProcessor*, and *ReceiptGenerator* classes, highlighting their responsibilities and dependencies. This provides an explicit map of the moving parts within the component.

Despite its clarity, **the code level is rarely used**. In most cases, developers prefer to consult the source code directly, and maintaining these diagrams in sync with rapidly evolving codebases can become tedious. Still, there are moments when they are valuable—for instance, in safety-critical domains, in onboarding situations where internal complexity must be explained quickly, or when documenting frameworks and libraries for external consumers.

A practical way to make this level relevant without creating maintenance overhead is to link diagrams directly to the source code repository. Instead of trying to freeze the code structure in static images, reference the GitHub repository where the code lives.
This ensures the diagrams remain lightweight while giving readers a clear path to the latest, authoritative implementation details.
A simple annotation like “See GitHub repository: /checkout-service” keeps the reference alive and trustworthy.

In short, the code level is best used selectively, by architects and senior developers, when there is a strong need to explain internal details. When combined with links to living repositories, it can provide clarity without adding unnecessary maintenance burden.

CHAPTER 7

IN SUMMARY

IN SUMMARY

The C4 model gives us a structured way to tell the story of our software architecture—from the broad strokes of the system context down to the fine-grained details of code. By progressively refining the view, we can communicate effectively with different audiences: business stakeholders at the context level, developers and architects at the container and component levels, and occasionally engineers at the code level.

Across these diagrams, the key value lies in clarity.

Instead of overwhelming teams with either vague abstractions or low-level code, C4 provides just enough detail to illuminate responsibilities, boundaries, and interactions.

Combined with practices like Domain-Driven Design and explicit mapping of containers to teams, this approach helps create architectures that are both comprehensible and evolvable. Of course, diagrams are only as valuable as they are maintained.

In fast-moving organizations, static drawings quickly become obsolete, eroding trust. This is where modern tools come in. Platforms such as **IcePanel** extend the C4 model beyond paper or slide decks, enabling teams to keep diagrams synchronized with reality, explore system flows interactively, and simplify discussions in meetings or onboarding sessions. Features like dependency views, flows, and live links turn diagrams into living documentation rather than one-off deliverables.

If you are serious about improving the way your teams communicate architecture, now is the time to try out a tool that makes C4 practical. Whether you are scaling a microservices ecosystem, coordinating micro frontends, or simply onboarding new developers, having living, interactive diagrams can make the difference between confusion and clarity.

Start experimenting with C4 in your own systems and consider adopting tools like IcePanel to bring your diagrams to life. The earlier you make your architecture visible, shared, and continuously maintained, the more resilient and aligned your teams will become.