KompozeR
A Microservice Platform for Guided Configuration of Modular Shelving

Final Report for the Distributed Systems Course [2025/2026]

Valerio Giannini
valerio.giannini@studio.unibo.it

August 4, 2026

Kompo is a company that produces customizable modular furniture. Its catalog is organized into four product categories—Tondo, Quadro, Intelligente and Kube—which are not compatible with one another but share similar features, so that the logic for composing components stays homogeneous within each category. In the current purchasing process, customers must select the individual components on their own, already having a clear idea of the final result; this is complex and requires the knowledge of the product, forcing them to rely on the support of the company. KompozeR wants to make this process easier by letting customers interactively design and purchase modular shelving units through a visual configurator (CAD) that composes a product in an intuitive way avoiding incompatible combinations, keeps price and availability aligned with the catalog in real time, and notifies users when a saved configuration is affected by catalog changes. The system is built as a web application using the stack MEVN (MongoDB, Express, Vue.js, Node.js) and is deployed as a hexagonal architecture. Services communicate through synchronous REST for commands and queries, asynchronous Redis pub/sub for domain events (price and availability changes), and Socket.IO for real-time delivery of notifications and chatbot messages. Socket.IO also enables a collaborative configuration mode, where multiple users can work on the same shelving unit at the same time and see each other's changes in real time. To keep the core configuration workflow available even under failures, the data is stored on a MongoDB replica set that provides replication and automatic failover, so that the loss of a single node does not interrupt the service. The whole system can be deployed both with Docker Compose for local development and on Kubernetes, which handles scaling, health checks and self-healing of the running services.

1 Concept

1.1 Product type

KompozeR is a **web application** (a browser-based Single Page Application) that extends a classic e-commerce scenario with a domain-specific feature: a guided, visual *configurator* (referred to as the CAD service) that lets a customer compose a modular shelving unit out of individual components. The catalog is organised in four product categories (Tondo, Quadro, Intelligente and Kube), and the configurator enforces the domain constraint that components of different families cannot be mixed in the same configuration.

1.2 Application scenario

- **What** the software does: it lets users build a valid shelving configuration while seeing an up-to-date preview and price, automatically save configurations to their profile, add them to a cart, and place orders. It keeps saved configurations consistent with the catalog, notifying users in real time when a component they depend on changes price or becomes unavailable. A contextual chatbot assists users during configuration, and administrators manage the catalog and inspect order trends.
- **Where** the users are: users are geographically distributed and reach the system over the public Internet from their own devices. Servers are centralised (a single logical deployment), while clients are remote.
- **When and how frequently**: interaction is on-demand and session-based. A configuration session is a relatively long, interactive activity; administrative catalog updates happen occasionally but must propagate promptly to every affected user.
- **How they interact / devices**: through a web browser (desktop or mobile) talking to the SPA. Communication is mostly request/response over HTTP; real-time channels (WebSocket via Socket.IO) are used for notifications, the chatbot, and the collaborative CAD sessions, where several users edit the same configuration and see each other's changes in real time.
- **Data to store**: the system persists users and sessions, the component catalog, configurations and their bill of materials (BOM), carts, orders, notifications and subscriptions, and chat sessions. Each kind of data is owned and stored by the service responsible for it (database-per-service).
- **Roles**: the system distinguishes multiple roles — `GUEST`, `BASE` (authenticated customer), and `ADMIN` (catalog and back-office operator) — with different permissions.

1.3 Why distribution is needed

Distribution in KompozeR is a design consequence of the domain. The main motivations are:

- **Resource sharing**: a single update of the catalog and its price/availability information must be shared among many concurrent users and among several application contexts (configurator, cart, notifications, chatbot, reporting).
- **Evolvability and separation of concerns**: the problem naturally decomposes into distinct bounded contexts (authentication, catalog, configuration, cart, orders, notifications, chatbot, reporting). Designing each of them as an independently deployable service allows them to evolve, scale, and be tested in isolation.
- **Fault tolerance**: the configuration workflow is the core of the product. Its persistence is backed by a replicated MongoDB replica set that survives the loss of a node through automatic primary election, and the collaborative session state is protected through checkpoint/recovery.
- **Scalability**: the load is dominated by many independent, read-heavy configuration sessions. A stateless service tier behind a gateway can be scaled horizontally to absorb an increasing number of users and requests.

The deployment is logically centralised, and the interesting distributed-systems challenges are consistency of shared catalog data, event-driven propagation of changes, and fault tolerance of the core workflow.

2 Requirements Elicitation and Analysis

This chapter states *what* KompozeR must do, independently of how it is implemented. Requirements are derived from the stakeholders (customers, recurring registered customers, catalog administrators) and from a small set of representative scenarios: configuring a shelving unit, asking the chatbot for contextual help, saving and resuming a configuration, being notified of a price/availability change, administering the catalog, and consulting order trends. They are grouped into *functional*, *non-functional*, and *implementation* requirements, each with an identifier and an acceptance criterion used during validation (section 5).

2.1 Glossary

Component a single catalog item (e.g. a shelf, an upright), identified by a SKU and belonging to a product family.

Product family one of the four mutually incompatible catalogs (Tondo, Quadro, Intelligente and Kube); components of different families cannot be combined.

Configuration a work-in-progress or saved shelving design owned by the CAD context, with a state, a column plan, and a derived Bill of Materials.

BOM Bill of Materials: the list of components (with quantities) that a configuration resolves to, and the basis for its price.

Subscription a user's interest in a specific component (due to have it in the configuration), used to notify the user when that component changes price or availability.

Role the authorisation level of a principal: `GUEST`, `BASE` (authenticated customer), or `ADMIN`.

2.2 Functional requirements

FR1 — Visual configurator The system shall provide a visual configurator to compose a shelving unit within a selected product family. *Acceptance:* starting a session and adding compatible components yields a coherent configuration that can be persisted and retrieved.

FR2 — Live preview The system shall update the configuration preview after every change made in the configurator. *Acceptance:* adding, removing, or replacing a component is reflected in the returned configuration snapshot.

FR3 — Live price The system shall update the estimated price of the configuration after every change. *Acceptance:* the price returned for a configuration equals the sum derived from its current BOM and catalog prices.

FR4 — Add to cart The system shall allow a finalised configuration to be added to the cart. *Acceptance:* finalising a configuration produces a cart whose items match the configuration BOM (SKU, quantity, unit price).

FR5 — Contextual chatbot The system shall provide an integrated chatbot answering contextual questions about component prices. *Acceptance:* a price question about a known component returns the current catalog price for that component within the user's own session.

FR6 — Save configuration The system shall let authenticated users save a configuration under their profile. *Acceptance:* a saved configuration is associated with the owner and appears in that user's list.

FR7 — Resume configuration The system shall let authenticated users list and resume previously saved configurations. *Acceptance:* a user retrieves only their own configurations and can reopen one for editing.

FR8 — Change notifications The system shall notify authenticated users of price or availability changes affecting their active or saved configurations. *Acceptance:* an administrative change to a subscribed SKU results in exactly one notification delivered to each affected user, and to no other user.

FR9 — Family compatibility The system shall prevent composing a configuration that mixes components of different product families. *Acceptance:* attempting to add an incompatible component is rejected and the configuration is left unchanged.

FR10 — Catalog administration The system shall let administrators update catalog data, including component price and availability. *Acceptance:* an authorised admin update is persisted and becomes visible to catalog reads and triggers the corresponding domain event.

FR11 — Orders and reporting The system shall provide administrators with an order view and a synthetic sales-trend report. *Acceptance:* an authorised admin obtains order data and an aggregated trend over a selected period.

2.3 Non-functional requirements

NFR1 — Consistency of shared catalog Price and availability changes shall be propagated consistently to every context that depends on them (cart, notifications). *Acceptance:* after a catalog change, dependent contexts eventually reflect the new value, and each change is applied at most once (idempotent event handling).

NFR2 — Real-time delivery Notifications and chatbot replies shall be delivered to the interested user in near real time, without client polling. *Acceptance:* a subscribed, connected user receives a push within a short bound of the triggering event.

NFR3 — Fault tolerance of the core workflow The configuration context shall tolerate the loss of a single database node without data loss and without permanent unavailability. *Acceptance:* killing one replica member triggers automatic election of a new primary and writes resume.

NFR4 — Recoverability The collaborative configuration session state shall survive a service crash. *Acceptance:* after crashing and restarting the service, the last checkpointed session state is recovered.

NFR5 — Security / authorisation Administrative functionality shall be accessible only to authenticated users with an authorised role, and users shall only access their own resources. *Acceptance:* unauthenticated or `BASE` access to admin endpoints is rejected; cross-user access to configurations, orders, chat, and notifications is denied.

NFR6 — Data minimisation / privacy The system shall store only the data required to operate accounts, saved configurations, and application operations. *Acceptance:* persisted schemas contain no data beyond what the use cases require.

NFR7 — Maintainability / evolvability Configuration, catalog, notifications, and reporting shall remain independently deployable and separately owned. *Acceptance:* each context is a separate service with its own database and can be built, tested, and deployed independently.

NFR8 — Usability and responsiveness The interface shall give immediate feedback on relevant actions and be usable on desktop and mobile. *Acceptance:* configuration, saving, and price updates produce visible feedback; core flows work at desktop and mobile breakpoints.

2.4 Implementation requirements

IR1 — Distributed architecture The system shall be realised as multiple, independently deployable services rather than a monolith, to exercise distributed-systems concerns (course constraint).

IR2 — Containerisation Every component shall be packaged as a container and be reproducibly deployable via a declarative orchestration (Docker Compose for development, Kubernetes for a production-like setup).

IR3 — Technology baseline Backend services shall be implemented in Node.js/TypeScript with Express, and the frontend as a Vue 3 SPA.

IR4 — Persistence Persistent state shall be stored in MongoDB, one logical database per service, honouring the database-per-service constraint.

IR5 — Automated validation The system shall be covered by automated unit, integration, and end-to-end tests runnable from the command line.

2.5 Relevant Distributed System Features

This section motivates which distributed-system features are central to KompozeR and which are secondary.

Highly relevant.

- **Resource sharing** — *central.* A single authoritative catalog (prices, availability) is shared by many concurrent users and by several contexts (configurator, cart, notifications, chatbot). Much of the design revolves around sharing this resource without letting contexts read each other's private state.
- **Fault tolerance and dependability** — *central* for the core configuration workflow. Losing a configuration is unacceptable, so its persistence is replicated and must survive a node failure; the collaborative session state must be recoverable after a crash (NFR3, NFR4). For non-core contexts, availability is desirable but not critical.
- **Consistency / data management** — *central.* Catalog changes must reach dependent contexts reliably and be applied exactly once (NFR1). The system favours *eventual consistency* across contexts via events, while keeping strong consistency *within* the core context via a configurable write/read concern on the replica set.
- **Security and trust** — *central.* The system stores personal and order data, distinguishes roles, and must isolate users from one another (NFR5). Authentication and authorisation are enforced at the gateway and re-checked by services.
- **Evolvability / maintainability** — *central*, and a primary reason for choosing microservices (NFR7, IR1): each bounded context evolves and deploys independently.

Relevant but bounded.

- **Scalability** — *relevant.* The workload is many independent, read-heavy sessions; the stateless service tier behind the gateway can scale horizontally.
- **Transparency** — *relevant.* Location and access transparency are provided to clients: they only know the gateway and are unaware of service placement, replication of the CAD database, or which node is primary.
- **Openness / interoperability / heterogeneity** — *partially relevant.* Services interoperate through plain HTTP/JSON and Socket.IO, which keeps them loosely coupled and language-agnostic; however the stack is deliberately homogeneous (Node.js), so heterogeneity is not exploited.
- **Performance / concurrency / efficiency** — *relevant.* Perceived responsiveness of preview/price updates and real-time delivery matter (NFR2, NFR8), and many sessions run concurrently.

Secondary.

- **Geographic distribution** — the deployment is logically centralised (single cluster); components are distributed across containers/pods, not across datacenters or continents.
- **Economy / cost** — beyond running on a modest, single-cluster footprint, cost optimisation is not a design driver in this academic setting.

3 Design

3.1 Architecture

KompozeR adopts a **microservice** architectural style behind an **API Gateway**, with a **client–server** interaction between the browser SPA and the backend. This style was chosen as a direct consequence of the analysis: the domain decomposes itself naturally into bounded contexts (section 3.3), and the non-functional requirements of evolvability and independent scalability (section 2.5) are best served by independently deployable services, each owning its own data (*database-per-service*).

The system is composed of the following logical tiers:

- **Presentation**: a Single Page Application. It never talks to the services directly; every request goes through the gateway.
- **Gateway**: a single entry point that routes REST calls to the downstream services, proxies the WebSocket channel, enforces authentication and role checks, normalises identity headers, and centralises error handling. It also exposes a small number of Backend-for-Frontend (BFF) endpoints that aggregate data from several services for specific screens.

- **Domain services**: one service per bounded context — *authentication*, *catalog*, *CAD* (configuration), *cart*, *order*, *notification*, *chatbot*, and *reporting*. Each service is autonomous, owns its schema, and exposes a REST API; some of them also publish or consume asynchronous events.
- **Data and messaging**: a dedicated MongoDB database per service and a Redis instance used as an event bus (publish/subscribe) and cache.

Three communication styles coexist, each used where it fits best:

- **Synchronous REST** for commands and queries (the default interaction);
- **Asynchronous events** over Redis pub/sub for domain events that must be propagated to several contexts (catalog `PRICE_CHANGED` and `AVAILABILITY_CHANGED`);
- **Real-time push** over Socket.IO (WebSocket) for notifications and chatbot messages, so users receive updates without polling.

Figure 1 shows the resulting logical components and how the three communication styles map onto them.

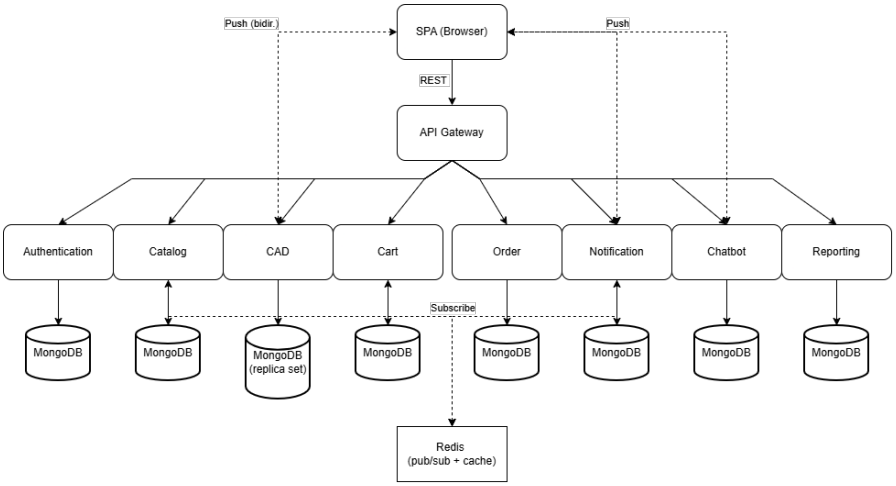

Figure 1: Logical component diagram. The browser SPA never talks to a domain service directly: every call goes through the API Gateway (solid arrows, synchronous REST), which fans requests out to the eight domain services; each service persists to its own MongoDB database (thin grey arrows), the CAD service's being a three-member replica set instead of a single instance. Dashed arrows show the asynchronous path: the catalog service publishes domain events on Redis pub/sub, consumed independently by the cart and notification services. Dotted arrows show the real-time push channel over Socket.IO: one-way from the notification and chatbot services to the browser, bidirectional for the CAD collaborative editor; the gateway proxies this channel end-to-end even though, for clarity, the diagram draws it directly between its logical endpoints.

3.2 Infrastructure

The system introduces the following infrastructural components:

- **Clients**: web browsers running the SPA.
- **Reverse proxy / static server**: an nginx instance serves the static SPA build and reverse-proxies `/api` (including the WebSocket upgrade) to the gateway, acting as the single public entry point of the application.
- **API Gateway**: one instance, the only component reachable by clients for API traffic.
- **Domain services**: eight stateless backend services (authentication, catalog, CAD, cart, order, notification, chatbot, reporting).
- **Databases**: one MongoDB deployment per service. Most contexts use a single-node MongoDB; the *CAD* context, being the core workflow, uses a **MongoDB replica set** for fault tolerance.

- **Message broker / cache**: a single **Redis** instance used both as a pub/sub event bus between services and as a cache.

Distribution over the network. All components run as containers within a single logical environment (a Docker Compose stack for development, a Kubernetes namespace for a production-like setup). Clients are remote and reach the system over the Internet; services, brokers and databases are in the same network but in separate containers/pods, so that each can fail, restart, and scale independently. The CAD replica set is spread over three separate members to tolerate the loss of a node.

Naming and discovery. Components find each other by logical name rather than by address. Within the container network, services are reached through their service name resolved by the platform DNS (Docker Compose's embedded DNS, or Kubernetes `Service` DNS), and each service is configured with the base URLs of its collaborators. Clients only ever know a single name than the gateway lets services move or scale without affecting the front end.

Figure 2 shows how these components are distributed and which ones run replicated.

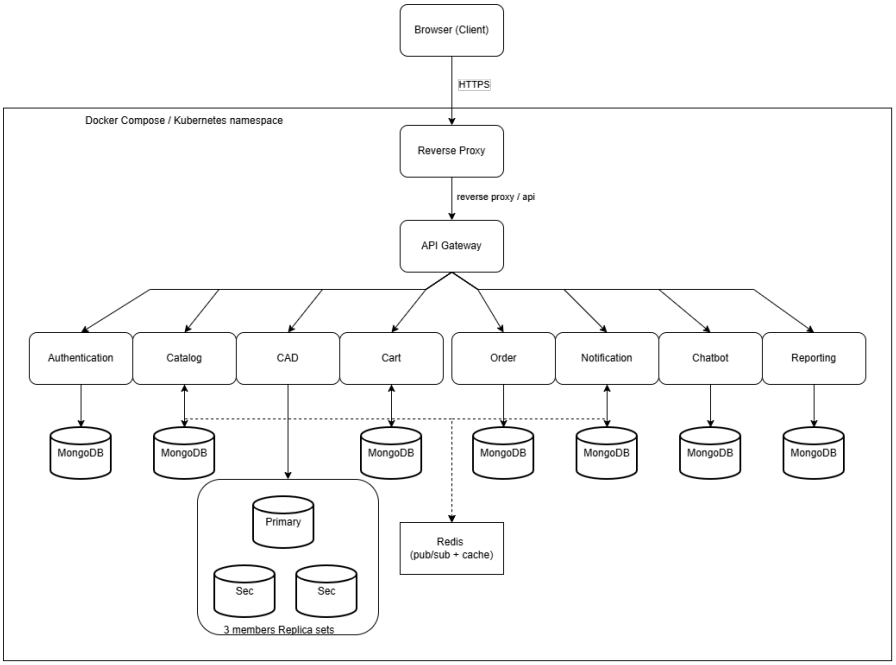

Figure 2: Infrastructural / deployment diagram. Every box other than the client is a separate container (Docker Compose) or pod (Kubernetes) distributed inside one logical network/namespace (thick outer boundary); the client is the only component outside it and reaches the system over the Internet. The reverse proxy is the single public entry point, serving the static SPA build and reverse-proxying `/api` (including the WebSocket upgrade) to the API Gateway, itself the only component that talks to the eight stateless domain services. Each service owns a single-instance MongoDB with a persistent volume, except the CAD service, whose database is instead a three-member replica set (one primary, two secondaries) spread over separate members, so that losing any one of them still leaves a majority able to serve reads and writes. A single Redis instance sits on the same network, reachable by every service, but only the catalog, cart and notification services currently use it, as the pub/sub event bus of section 3.3.

3.3 Modelling

Domain entities. Following *database-per-service*, every bounded context owns a MongoDB collection; entities are never shared across service boundaries, only their identifiers are.

Figure 3 summarises the entities owned by each context:

Authentication *User* (username, email, password hash, role $\in$ {`GUEST`, `BASE`, `ADMIN`}) and *Session* (token id, expiry, revocation flag), used to issue and invalidate JWTs.

Catalog *Component* (SKU, category/family, type, price, availability, dimensions, com-

patible SKUs, an optimistic-concurrency `version`).

CAD *Configuration* (owner, collaborators, lifecycle `status`, environment constraints, column plan, per-column design, derived BOM, `version`); plus two supporting collections used only for collaborative-session recovery (section 3.7): *CollabCheckpoint* (last known session snapshot and Lamport clocks) and *CollabEvent* (an append-only log of accepted edits since the last checkpoint).

Cart *Cart* (one per user: line items with SKU/quantity/price).

Order *Order* (owner, shipping information, line items snapshot, status $\in$ {`SUBMITTED`, `DONE`, `CANCELLED`}).

Notification *Notification* (recipient, type, related SKU, originating context), *Subscription* (a user's interest in a component).

Chatbot *ChatSession* and *ChatMessage* (role `USER`/`BOT`, content).

Reporting a read-only *Order* projection (status, total, submission date) used only for aggregation queries.

Mapping to infrastructural components. Each entity lives exclusively in its owning service's MongoDB database and is mutated only by that service; other contexts that need the data either *copy* the relevant identifiers/values at write time or *query it* over REST (e.g. the CAD service asks the catalog for current component rules whenever it derives a BOM, section 3.6). The one deliberate exception is the CAD collaborative session: while a session is open, its authoritative state lives in the memory of the single `cadService` instance that hosts it (not in MongoDB), because it changes many times per second and every participant needs the same in-process view; MongoDB is used only as a periodic, recoverable checkpoint of that in-memory state.

Domain events. Two contexts publish domain events on Redis pub/sub, both carrying an `eventId`:

`catalog:events` published by the catalog service whenever an admin update changes a component: `PRICE_CHANGED` (old/new price) or `AVAILABILITY_CHANGED` (old/new availability). Consumed by the cart service and by the notification service.

`cart:events` published by the cart service for every cart-affecting action (item added/removed, cart synchronised from a configuration, items removed/restored due to availability, prices updated, order requested). No other service currently subscribes to this channel; it exists for traceability/observability and as an extension point.

Messages exchanged. Three kinds of messages appear in the system, matching the three communication styles of section 3.1: *commands and queries* as synchronous REST request/response; *domain events* as asynchronous, messages on Redis pub/sub (no persistence of undelivered messages: a consumer that is down when an event is published misses it); and *real-time messages* over Socket.IO for notification, chatbot answers and fine-grained *operations* in the CAD collaborative editor (one message per field changed, carrying an operation id, a Lamport timestamp and the base version it was computed against).

System state. The overall state of KompozeR comprehends: account and session data (authentication); the catalog of components and their price and availability (catalog); in-progress and saved configurations, their BOM, and open collaborative sessions (CAD); carts (cart); placed orders (order); notifications and subscriptions, plus the ledger of already-processed catalog events (notification); chat transcripts (chatbot); and derived, read-only sales data (reporting). There is no single component holding a global view of this state: each context is authoritative for its own slice, and the gateway never persists anything itself.

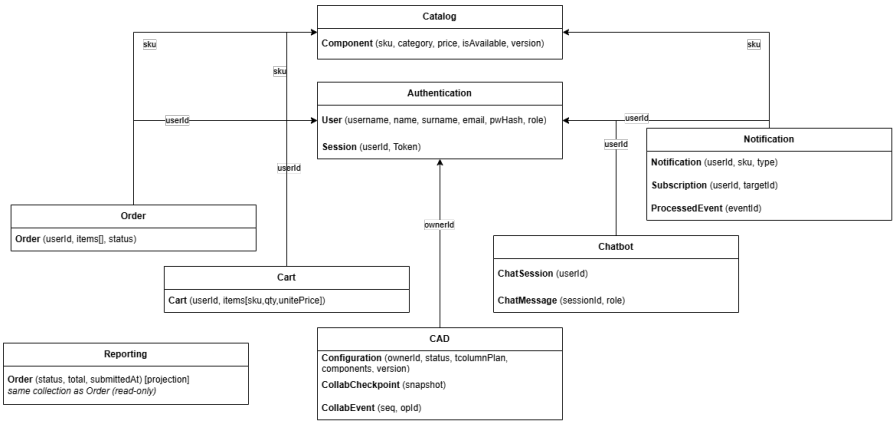

Figure 3: Entities owned by each bounded context (database-per-service). Solid boxes never share a database. The arrows illustrate the general pattern (Cart referencing User and Component by id only, never by foreign key or join).

3.4 Interaction

Components communicate through the three channels introduced in section 3.1, each with a different interaction pattern:

Request/reply over REST is the default: the SPA calls the gateway, the gateway forwards the call (adding trusted identity headers) to exactly one domain service and returns its JSON response synchronously. This is used for every command and query that has a single, well-identified owner.

Scatter/gather at the edge is used by the gateway's Backend-for-Frontend endpoints (e.g. the dashboard, the configurator screen), which fan a single client request out to several domain services in parallel and gather the results with `Promise.allSettled`: a failing or slow service degrades its own field of the response instead of failing the whole call, trading a little staleness/incompleteness for availability.

Publish/subscribe over Redis decouples the catalog (the producer of `PRICE_CHANGED` and `AVAILABILITY_CHANGED`) from the cart and notification services (the consumers). This is a genuinely asynchronous, at-most-once channel: Redis pub/sub does not queue messages for a consumer that is offline, so a service that is down when an event is published misses it. Every consumer applies the event idempotently, so re-processing is always safe.

Real-time push over Socket.IO is used one-way, from a service to a specific, already-identified client (a room per user id for notifications, a room per chat connection for the chatbot), and bidirectionally for the CAD collaborative editor, where it carries fine-grained *operations* instead of request/response payloads.

Replicated-state broadcast with conflict resolution is the pattern behind collaborative CAD editing: every accepted edit is broadcast to all members of the session's room, so each browser keeps a full, up-to-date replica of the configuration being edited, and conflicts between concurrent edits to the same field are resolved deterministically with Lamport clocks (section 3.5).

Health polling is used by the gateway to actively probe every downstream service's `/health` endpoint on demand, rather than services pushing their own liveness.

Figure 4 details the interaction triggered by an administrative catalog change, which mixes request/reply (the admin update), publish/subscribe (propagation to cart and notification), and push (delivery to the browser). Figure 5 details the interaction inside a collaborative CAD session, purely over Socket.IO.

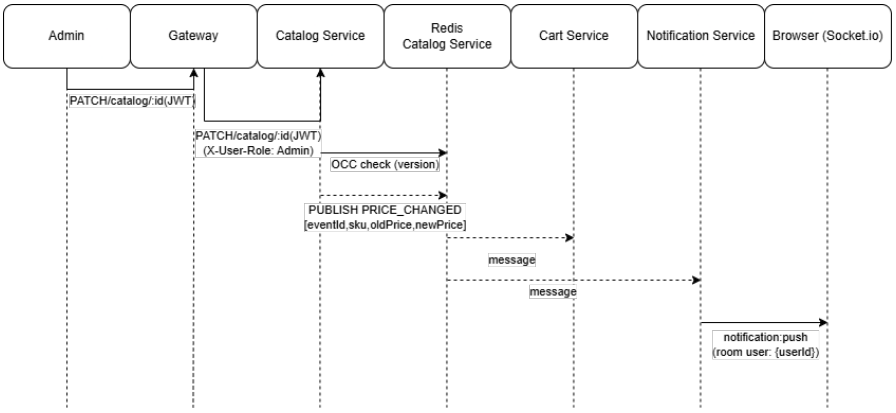

Figure 4: Propagation of a catalog change (e.g. an admin lowers a component's price) to every dependent context. The gateway turns the client call into a trusted internal request; the catalog service is the only writer and the only publisher; cart and notification consume the same event independently and idempotently; only the notification service ends up pushing to connected browsers.

3.5 Behaviour

Individual behaviour. Six of the eight domain services (authentication, catalog, cart, order, chatbot, reporting) behave as **stateless** request handlers: every REST call is handled by loading the relevant document(s) from that MongoDB's service, applying a pure use-case function, and writing the result back (no business state lives in the process itself) which is what lets any of them be restarted or (in principle) replicated without coordination. The **notification** service is stateless in the same sense but is additionally *event-driven*: its behaviour in response to a `catalog:events` message is fixed by `HandleCatalogEvent` (fig. 4) and it takes no action on its own initiative. The **CAD** service is the exception: alongside its stateless REST endpoints for solo editing, it hosts **stateful** in-memory collaborative sessions (section 3.3), whose behaviour in response to an incoming operation is the deterministic rule shown in fig. 5 (validate base version, resolve the conflict with a Lamport clock, mutate the shared snapshot, log the change).

Who updates state, when, how. Each context is the *only* writer of its own state, always synchronously, in direct response to a request it received. The are 2 exception, the notification service writes state in reaction to an event it did not originate, and also during a collaborative CAD session, the in-memory snapshot is updated by whichever participant's operation wins the Lamport/LWW comparison, with MongoDB updated only afterwards, asynchronously, by the periodic checkpoint.

State machine of a Configuration. The CAD service is also the only place with an explicit, named lifecycle: every configuration carries a `status` that is entirely *derived*

from which of its fields are populated, recomputed by the same rule whether the edit came through a regular REST call or through a collaborative operation (`applyStatusTransition`, shared by both paths so both keep the same semantics). Figure 6 shows the intended forward path and the reset behaviour:

3.6 Data and Consistency Issues

What is stored, where, why. Section 3.3 lists the data for each context; all of it is persisted, without exception, in its own MongoDB database rather than in process memory; that is because keeping it in memory would tie every context's state to the lifetime of a single process instance: a restart or a crash would either lose state a client expects to survive across requests or leave it inconsistent between instances, and neither the request handler nor the client has any way to recover it. Persisting to MongoDB instead makes that state outlive is what lets the services be restarted or replicated without coordination. The only exception, the CAD collaborative session, is described in section 3.7 because it *starts* as in-memory state and has to be reconciled with MongoDB.

Why documents. MongoDB (a document store) was chosen for every context, for reasons that hold uniformly across the domain: (i) most entities are naturally tree-shaped and read/written as a whole (a configuration with its column plan, per-column design and BOM; a cart with its line items; an order with a frozen shipping address and item list) so embedding them in one document avoids the joins a relational schema would need for the same read; (ii) *database-per-service* already rules out cross-service joins, so the usual argument for a shared relational schema does not apply; (iii) schema flexibility matters during development of a project with several evolving contexts; (iv) MongoDB's replica set gives the CAD context the replication and automatic failover required by NFR3 without introducing a second data-store technology.

Queries and concurrent access. Read patterns are simple and mostly key- or owner-scoped: by `sku`/category/text search (catalog, indexed), by `ownerId`/`collaborators` (CAD, compound-indexed with `updatedAt`), by `userId` (cart, one document per user; orders and notifications, indexed and paginated by date). Concurrent *reads* are unrestricted (MongoDB's default). Concurrent *writes* to the *same* document are the one place true races can occur, and the system handles them with two different mechanisms depending on the needs:

- **Whole-document optimistic concurrency control**, for catalog components and for a configuration edited solo: every document carries a `version` field; a write must supply the `version` it read (`expectedVersion`/`baseVersion`), and the update is only applied with a conditional `findOneAndUpdate` filtering on {`_id, version: expected`}; a mismatch means someone else updated first, and the request is rejected with a 409 (`VersionConflictError`) instead of silently overwriting a concurrent change (a classic *lost-update* prevention).
- **Field-level, last-writer-wins with a Lamport clock**, only inside an open collaborative CAD session, where whole-document OCC would be too heavy (every participant would constantly conflict with everyone else's most recent edit): each of the editable fields keeps its own logical clock, so two participants editing *different* fields never conflict, and two edits to the *same* field are ordered deterministically (section 3.5) instead of one simply failing.

No multi-document transactions are used anywhere: every write that must be atomic is expressed as a single conditional update to a single document, which keeps each service's persistence layer simple and effective for distributed transactions across bounded contexts.

Data shared between components. The catalog's price and availability is the one piece of state genuinely needed by several contexts, and it is shared through two different strategies: contexts that need the *current* value at the exact moment of an operation pull it synchronously over REST (CAD deriving a BOM, cart snapshotting a line item, both in section 3.6's spirit of "ask the owner, don't copy the truth"); contexts that need to *react* to a value that changed after the fact subscribe to the `catalog:events` stream (section 3.3) and update their own copy accordingly (notification computing impacted users). One boundary is intentionally bent for pragmatism: the notification service's impact resolver reads the cart and CAD services' MongoDB collections directly (a second connection into their databases) instead of calling their REST APIs, to efficiently answer "which users have this SKU in their cart/finalized configuration" (a deliberate trade-off against strict database-per-service encapsulation, acceptable here because the notification service only ever *reads* that data).

3.7 Fault-Tolerance

Replication. The only replicated data store is the CAD MongoDB replica set (`cadrs`), matching NFR3's decision to concentrate fault-tolerance investment on the core configuration workflow. In Kubernetes it runs as three data-bearing members (Primary-Secondary-Secondary, one preferred-primary member at priority 2, two at priority 1); in Docker Compose, two data-bearing members plus an arbiter (which only votes in elections and holds no data, a lighter-weight way to keep an odd number of voters for local development). The cadService connection is configured (section 3.6, env-driven) with write concern `majority`, read concern `majority` and read preference `primaryPreferred`: a write is acknowledged only once it is durable on a majority of members, and reads prefer the primary but can fall back to a secondary, making the consistency/availability trade-off explicit and tunable rather than hard-coded. Losing one member (a pod restart, a node failure) leaves a majority among the remaining two, so writes keep flowing and, if the lost member was the primary, the replica set elects a new one automatically among the survivors. Every other context's database is a single MongoDB instance with a persistent volume: durability against disk loss, but no failover against the loss of the node itself, a conscious scope decision (section 2.5) since none of those contexts is the product's core workflow.

Collaborative session recovery (checkpoint/replay). The CAD service's in-memory collaborative sessions (section 3.3) need their own mechanism, since they are not stored in the replica set while active. A background timer checkpoints every live session to MongoDB every 10 seconds (`CAD_CHECKPOINT_INTERVAL_MS`), and every accepted edit in between checkpoints is additionally appended to a per-session event log, so that at most

10 seconds of edits can ever be missing from durable storage at a given instant. Figure 7 shows what happens when the process crashes and restarts: before accepting any connection, it loads every persisted checkpoint, discards those whose TTL already expired, rebuilds the in-memory session from the checkpoint snapshot, and *replays* the event-log entries recorded after that checkpoint through the very same mutation function used for live edits. Connected clients are then told to resynchronise (`cad:collab:resync`) and re-fetch the snapshot.

Timeouts and health checks. Every inter-service REST call carries an explicit timeout rather than blocking indefinitely: 3 s (CAD → catalog, fetching pricing rules), 5 s (cart → catalog; gateway → any downstream service). Every service exposes a `GET /health`; the gateway actively aggregates all of them into one endpoint (3 s per-probe timeout, run in parallel), returning `200` if every service is up, `207` if some are degraded, `503` if all are down. Kubernetes complements this with per-pod *readiness* probes so the platform stops routing traffic to a pod that is not ready; no liveness probes are configured, so Kubernetes will not restart a pod that is up but stuck. The one place with an actual *retry* is the chatbot's call to the external LLM provider; the other synchronous inter-service calls (CAD/cart → catalog) are single-attempt, relying on their short timeout and the caller's own error handling rather than on retrying.

Error handling. Failures are handled at three levels. Within a service, use cases translate domain errors into fixed HTTP status codes through a shared error-mapping middleware (`404` not found, `409` version/business conflicts, `422` validation), so a client always gets a meaningful response instead of a stack trace. Across services, the gateway's Backend-for-Frontend endpoints isolate partial failure with `Promise.allSettled` (section 3.4): one failing service surfaces as an error *inside* its own field of the response, the others are unaffected. Inside the CAD collaborative session, operations are idempotent by `opId` (a retransmitted or duplicated operation is detected and acknowledged without being re-applied); as the notification service, the same idempotency principle is applied to catalog events via the `ProcessedEvent` ledger keyed by `eventId` (section 3.3).

3.8 Availability

Caching. Redis is present in the architecture (section 3.2) but, in the current implementation, only as the pub/sub event bus of section 3.3; there is no read-through cache in front of MongoDB. Every read, including a repeated catalog lookup inside a busy configuration session, goes to the owning service's database directly. This was a deliberate scope decision given the academic setting and the modest data volumes involved.

Load balancing. The stateless tier (every domain service and the gateway) is architecturally prepared for horizontal scaling: no service keeps per-request state between calls, so any number of replicas behind a Kubernetes `Service` could serve traffic interchangeably, with `kube-proxy` load-balancing across them and clients only ever knowing the

gateway's name. The reference deployment currently runs a single replica of each service (`replicas: 1`), so this capability is available but not exercised yet (again a scope choice). The gateway is exposed as a Kubernetes `NodePort`, the single externally reachable, load-balanced entry point for all client traffic; every internal service sits behind its own `ClusterIP` Service, invisible from outside the cluster.

Behaviour under network partitioning. The two data stores react differently, on purpose. The CAD replica set tolerates a partition that still leaves a majority reachable. A single-node MongoDB (every other context) has no such tolerance: if it becomes unreachable, that context is simply down until it recovers, an accepted trade-off, since none of those contexts is the core workflow. At the service level, the gateway's plain proxy routes have no built-in fallback: if a downstream service cannot be reached, the proxied call fails and the error propagates to the client.

3.9 Security

Authentication. The authentication service issues signed JWTs (HS256) on login, registration, and guest-session creation, embedding `userId`, `tokenId` and `role` in the payload; a matching `Session` document lets a token be revoked (logout) independently of its expiry. Signature verification, however, happens in exactly *one* place: the gateway. Every request carries `Authorization: Bearer <token>`; the gateway verifies it, and, only on success, replaces any client-supplied `X-User-Id`, `X-User-Role` and `X-Session-Id` headers with values derived from the verified token before forwarding the request downstream. Domain services never see the raw JWT and never verify a signature themselves; they trust the identity headers as set by the gateway. Registration, login and guest-session creation are the only endpoints reachable without a token. Socket.IO handshakes for the chatbot and CAD collaboration accept the token as a `?token=` query parameter as well, since browsers cannot attach custom headers to the WebSocket upgrade request.

Authorization. The system distinguishes three roles: `GUEST`, `BASE` (authenticated customer), `ADMIN` — and combines two complementary access-control styles. *Role-based* checks guard administrative operations: catalog writes and order-status changes require `X-User-Role: ADMIN`. *Ownership-based* checks guard access to a user's own resources, which matters more often than role in this domain: a cart, an order, notifications and chat sessions are scoped to their owning `userId`; a CAD configuration is accessible to its owner or to a collaborator explicitly added to it (`canAccessConfiguration`), and a collaborative session additionally requires knowing its short, randomly generated join code. Unauthenticated or `BASE` access to admin endpoints is rejected, and cross-user access to another user's resources is denied regardless of role.

Cryptography and hardening. Passwords are hashed with `bcrypt` at 12 salt rounds before storage, the raw password is never persisted or logged. JWTs are signed (not encrypted) with a secret provisioned as a Kubernetes `Secret` (`jwtSecret`, alongside the

MongoDB root credentials) and injected as an environment variable, never hard-coded. Security headers (`helmet`) are applied at the gateway, the single point all client traffic must cross. Two aspects are intentionally left minimal for this academic deployment rather than hardened: transport is plain HTTP end-to-end (no TLS termination is configured; a real deployment would add it at an ingress in front of the gateway), and input validation is hand-written per field (length/format checks, an email regex, a minimum password length) rather than schema-based, with no request rate limiting in front of the authentication endpoints, both reasonable targets for future hardening (section 9) rather than gaps that affect the design itself.

4 Implementation

Network protocols. Three protocols are used, matching the three communication styles of section 3.1. **HTTP/1.1** carries every synchronous REST call, both client-to-gateway and gateway-to-service. **WebSocket**: every Socket.IO server (CAD collaboration, notifications, chatbot, section 3.3) is configured with `transports: ['polling','websocket']`, so a client starts with HTTP long-polling and transparently upgrades to a persistent WebSocket connection where the network allows it, falling back gracefully otherwise. Internally, the **Redis protocol (RESP)** carries pub/sub domain events between the catalog, cart and notification services via `ioredis` clients (section 3.4). The chatbot service is the only component that talks to the outside world beyond the browser: it issues outbound **HTTPS** requests to the Mistral AI chat-completions API (`https://api.mistral.ai/v1/chat/completions`) to generate answers, with a template-based fallback if no API key is configured.

In-transit data representation. **JSON** is the only wire format used anywhere, REST request/response bodies, Redis event payloads, and every Socket.IO message, keeping the whole system technologically homogeneous and human-readable for debugging. A few conventions are applied consistently across services to avoid ambiguity: identifiers are **string UUIDs** used directly as MongoDB `_id`; monetary amounts are represented as **integer euro cents** rather than floating-point decimals, both in the catalog's source price and everywhere it is copied (cart line items, order snapshots, BOM items), specifically to avoid floating-point rounding errors in a domain full of price sums.

Database access. Every service talks to its own MongoDB exclusively through **Mongoose**, never the raw driver: domain entities are mapped to schemas in each service's `adapters/persistence` layer, and repositories expose only domain-shaped methods to the rest of the service (a port/adapter boundary). Queries are plain Mongoose finds/filters with indexes for the transactional read/write paths; the one place using MongoDB's **aggregation pipeline** is the reporting service (`MongoOrderTrendReader`), which groups orders by day and status (`$match`/`$group`/`$sum`) directly in the database rather than pulling raw documents and summing them in application code.

Authentication. Implemented with **JWT** (the `jsonwebtoken` library, HS256), issued by the authentication service on login/registration/guest-session creation and verified exclusively at the API gateway; passwords are hashed with **bcrypt** (`bcryptjs`, 12 salt rounds) and never stored or logged in clear text.

Authorisation. Implemented as plain **RBAC** via a small hand-written Express middleware duplicated in each service that needs it (no shared authorisation package), reading the gateway-issued `X-User-Role` header, complemented by ownership checks written directly in use-cases, acceptable given the small, fixed role set (`GUEST`/`BASE`/ `ADMIN`) and the fact that most access rules are simple ownership checks rather than complex attribute combinations.

4.1 Technological details

Backend. All nine backend components (the gateway and the eight domain services) are written in **TypeScript 5.4** on **Node.js 24** and follow the same internal layering (a light hexagonal/clean-architecture style): a `domain` layer (entities, value objects, port interfaces) with no framework dependency, a `useCases` layer implementing application logic against those ports, and an `adapters` layer (`http`, `persistence`, `httpClient`, `messaging`, `websocket` as needed) wiring the use-cases to **Express 4.18**, **Mongoose 8.4**, **ioredis** and **socket.io**. This uniform structure is what let unit tests fake every port and never depend on a real MongoDB (section 5). Cross-cutting libraries: `jsonwebtoken` and `bcryptjs` (authentication service), `helmet` and `cors` (gateway), `http-proxy-middleware` (gateway's REST/WS proxying).

Frontend. A **Vue 3.4** Single Page Application, written entirely with the **Composition API** (`<script setup lang="ts">`), bundled with **Vite 5**, and type-checked in strict mode with `vue-tsc`. State is managed with **Pinia** (a store per cross-cutting concern: `authStore`, `cartStore`, `notificationStore`), routing with **vue-router** (a single route table with per-route `meta` flags for auth/role guards, section 7). HTTP calls go through a small hand-written `fetch`-based client rather than a library such as axios, and real-time channels use **socket.io-client** through three dedicated wrappers, one per Socket.IO server (CAD collaboration, notifications, chatbot), mirroring the backend's separation. No CSS framework is used — components are styled with scoped `<style>` blocks over a small shared design-token stylesheet — and, unlike the backend, the frontend currently has no automated test suite (section 5).

External integration. The chatbot is the only feature depending on a third-party service: **Mistral AI**'s chat-completions endpoint (default model `mistral-small-latest`), called with a bounded retry and timeout policy (section 3.7) and gated by an API key read from an environment variable / Kubernetes secret — never hard-coded in source.

Containerisation. Every backend service and the frontend are packaged as Docker images built `FROM node:24-alpine`; the frontend's production image is a two-stage build

that compiles the Vite bundle and then serves it through a lightweight **nginx:alpine** image, which also acts as the reverse proxy for `/api` traffic (REST and WebSocket upgrades alike) as described in section 3.2 and detailed in section 6.

5 Validation

5.1 Automatic Testing

Testing is organised in three tiers:

Unit tests. Every one of the nine backend components (the gateway and the eight domain services) has its own, fully isolated Jest configuration and a `__tests__` folder, for a total of **44 test files**. Consistent with the hexagonal layering of section 4.1, these tests never touch a real MongoDB or Redis: each service provides a `__tests__/helpers/fakes.ts` with in-memory implementations of its domain ports so a use-case test real business logic against a deterministic double instead of infrastructure. Two kinds of targets are covered: *use-cases* and *pure domain functions*. HTTP adapters are also unit-tested, with `supertest` driving an Express app built purely from fakes (`buildTestApp.ts`), so a router test verifies status codes, validation and role/ownership rejection without any real dependency. Coverage is measured for an informative purpose, since there is no CI.

Domain-level concurrency tests. The most distributed-systems-relevant unit tests are cadService's `collabConcurrency.test.ts` and `collabCheckpointRecovery.test.ts`, which exercise `InMemoryCollabSessionService` (section 3.7) directly against fakes. The first covers twelve scenarios of the Lamport/LWW conflict rule: a higher Lamport timestamp wins, a tie is broken by actor id, independent fields converge independently, a duplicate `opId` is acknowledged without being re-applied, an old `baseVersion` is rejected, and the outcome is independent of the arrival order of two concurrent operations. The second test drives a full crash/recovery cycle: it checkpoints a session, creates a *brand-new* `InMemoryCollabSessionService` instance sharing the same checkpoint/event-log fakes (simulating a process restart), calls `recoverSessions()`, and asserts the rebuilt state (name, category, version, participants) is identical to the pre-crash state.

Integration tests ("e2e"). A separate, top-level `e2e/` package largely re-exercises the same business rules already covered by unit tests, but as black-box HTTP calls against the *real*, dockerised stack (real MongoDB/Redis instead of fakes). Its `globalSetup.js` polls the gateway's aggregated `/health` until the stack is up, obtains a guest JWT, and idempotently seeds an `ADMIN` dev user before the suite runs. Eight spec files (`auth`, `cad`, `cart`, `catalog`, `chatbot`, `notifications`, `order`, `reporting.integration.test.ts`), each tagged `[INT]`, cover registration/login/session/logout, configurator, cart/catalog/order CRUD, chatbot

Q&A and the admin-only reporting endpoint, with ownership isolation (403/404 on another user's resource) and role rejection asserted throughout (directly validating NFR5). Two things go beyond a mere replay with real infrastructure, since they involve behaviour unit tests cannot see by construction: the full configurator flow for both the TONDO and the INTELLIGENTE product family (environment → category → column plan → design → finalize, asserting the returned BOM) and catalog's `[DS]` case (`PUT /catalog/:id` against an old version → `409 VERSION_CONFLICT`) both exercise real persistence end to end.

How to run them. Every service exposes `npm test` and `npm run test:coverage`; the workspace root aggregates all nine with `npm run test:backend`, the black-box suite with `npm run test:e2e` (which requires `docker compose -f docker-compose.dev.yml up` to already be running), and `npm run test:baseline` chains both. There is currently **no continuous-integration pipeline** running any of this automatically on push or before merge.

5.2 Acceptance test

Two kinds of manual verification complement the automated suites, since neither is easily expressed as a Jest assertion.

Manual UI testing. The Vue frontend has no automated test suite (section 4.1), so every user-facing flow described in section 7 as the registration/guest access, the full configurator wizard for each of the four product families, the collaborative session (joining from a second browser tab/profile to see real-time synchronisation), cart/checkout, the notification bell and toast pushes, the chatbot, and the admin screens, was exercised manually against the Docker Compose stack.

Production-like deployment as a test. Following the hint that deployment automation doubles as a test of a production-like environment, the two distributed-systems properties that automated tests can only approximate with fakes were also demonstrated manually on the Kubernetes deployment (section 6): killing the current CAD MongoDB primary (`kubectl delete pod mongo-cad-0`) and observing the replica set elect a new primary and writes resume; and killing the running `cad-service` pod (`kubectl delete pod -l app=cad-service`) and observing an open collaborative session survive via checkpoint/replay, validating NFR4 against a real process crash and Kubernetes-driven restart, not just the `collabCheckpointRecovery` unit test.

6 Deployment

The system can be brought up in two ways: **Docker Compose**, for local development and for running the automated test suites of validation, and **Kubernetes** for a

production-like deployment. Both are declarative: bringing the system up or down is a single command.

6.1 Prerequisites

No Node.js version is declared in any `package.json` (no `engines` field, no `.nvmrc`); the authoritative version is fixed at the container level where every service and the frontend build `FROM node:24-alpine`, so it runs, whether or not Node is installed on the host. Consequently, the only host-level prerequisites are:

- **Docker** and **Docker Compose** (v2 syntax), for the Compose path.
- **Minikube** and **kubectl**, for the Kubernetes path.

6.2 Running with Docker Compose

Configuration. The file groups its variables by concern: shared credentials and infrastructure (`MONGO_ROOT_USER`, `MONGO_ROOT_PASSWORD`, `JWT_SECRET`, `REDIS_URL`), one MongoDB connection string per service (`*_MONGO_URI`), and one HTTP port per service. The chatbot's optional third-party integration (Mistral AI, section 4.1) is configured separately via `MISTRAL_API_KEY`/`MISTRAL_MODEL`: if unset, the chatbot falls back to template-based answers instead of failing.

Two Compose files, two purposes. `docker-compose.dev.yml` is the actively maintained development environment with the end-to-end suite of validation targets: it adds the `order-service`, gives every MongoDB container a named volume and a health check, exposes every database on a host port for inspection with a GUI client such as MongoDB Compass, and additionally runs the observability stack. Both manage the CAD context as a genuine three-member replica set initialised by a one-shot `mongo-cad-init` job that waits for all three members to be healthy.

Bringing it up. From `kompozer/`:

```
docker compose -f docker-compose.dev.yml up --build
```

Expected outcome: every container reports healthy, the CAD replica set finishes electing a primary within a few seconds, and the API gateway becomes reachable at `http://localhost:3000` (health check: `GET /health` returns the aggregated status of fault-tolerance). The frontend is served separately in this environment (`npm run dev:frontend` from the workspace root, a Vite dev server on `http://localhost:5173` proxying `/api` to the gateway); the automated test suites can then be run as described in section 5. Tearing down: `docker compose -f docker-compose.dev.yml down -v` (the `-v` also removes the named volumes for a full data reset).

6.3 Deploying to Kubernetes

Layout. All manifests live in `kompozer/k8s/`, numbered to convey dependency order and aggregated by a single `kustomization.yaml` (namespace `kompozer`). The gateway and the frontend are the only two `NodePort` Services while every other Service is a `ClusterIP`, unreachable from outside the cluster, matching the single-entry-point design of section 3.1.

Bringing it up. On a machine with Minikube:

```
minikube start --driver=docker --memory=6144 --cpus=4
(build every service image directly into Minikube's Docker store)
minikube image build ... (once per service/frontend image)
kubectl apply -k kompozer/k8s
```

Images are built locally and loaded straight into Minikube rather than pulled from a registry (every application `Deployment` sets `imagePullPolicy: Never`); only the infrastructure images (MongoDB, Redis, Loki, Promtail, Grafana) are pulled from public registries as usual. Expected outcome, checked with `kubectl rollout status statefulset/mongo-cad` and `kubectl wait --for=condition=complete job/mongo-cad-init`: the CAD replica set is initialised and every Deployment reaches its desired replica count; the application is then reachable via `minikube service frontend -n kompozer --url` (and, separately, `api-gateway/grafana` the same way). Tearing down: `kubectl delete -k kompozer/k8s`.

Demonstrating the distributed-systems properties. Because this deployment is closer to production than Compose, it is also where the fault-tolerance guarantees can be simulated against a real node/process failure rather than a fake: killing the current CAD primary (`kubectl delete pod mongo-cad-0`) shows automatic re-election and writes resuming (NFR3); killing the CAD service pod (`kubectl delete pod -l app=cad-service`) shows an open collaborative session recover via checkpoint/replay after Kubernetes restarts it (NFR4).

6.4 Observability

Both environments can additionally run a log-aggregation stack (**Grafana**, **Loki** and **Promtail**) scoped to development/troubleshooting rather than production use (in-memory index, filesystem storage, anonymous admin access). Promtail runs once per Docker host in Compose, and as a Kubernetes `DaemonSet` in the cluster deployment, shipping every container's logs to Loki; Grafana is pre-provisioned with Loki as a datasource, so logs from any of the nine backend components can be filtered and correlated from a single dashboard during manual testing.

7 User Guide

This chapter walks through KompozeR from the point of view of its three roles:`GUEST`, `BASE` and `ADMIN` following the navigation structure of the frontend. The interface itself is in Italian (for now).

7.1 Access

KompozeR is reached at the frontend's URL (section 6), which shows a single card with two tabs, "Accedi" (Log in) and "Registrati" (Register), plus a divider and a "Continua come ospite" (Continue as guest) button.

- **Registering** asks for Username, Nome (first name), Cognome (last name), Email and Password (minimum 8 characters); on success the new user is shown a confirmation dialog and returned to the login tab with their identifier pre-filled.
- **Logging in** accepts either the username or the email plus the password. A wrong password or an unknown identifier produce a clear inline error ("Password Errata" / "Utente Non Trovato").
- **Continuing as guest** skips registration entirely and opens a temporary `GUEST` session (FR1), useful for trying the configurator without commitment; a guest cannot save a configuration under a profile (FR6) or reach the catalog/admin screens.

Once logged in, a header bar appears at the top of every screen with navigation links (varying by role), a cart icon, a notification bell for authenticated non-guest users, the current username (or "Guest"), and an "Esci" (Log out) button.

7.2 Building a configuration

The configurator (route `/cad`) is the core of the product (FR1) and is organised as a five-step wizard, matching the `Configuration` lifecycle of section 3.5: "Ambiente" (environment), "Categoria" (product family), "Colonne" (columns), "Design" and "Finalizza" (finalise). A BOM panel and a live schematic are always visible on the side, updated after every step.

1. **Starting a configuration.** From "Configurazioni" (or directly from the configurator), enter a name, optionally pick a starting product family, and press "Crea e apri in CAD" (Create and open in CAD).
2. **Ambiente.** Enter the minimum and maximum width/height (mm) the finished unit must fit, then "Salva ambiente" (FR1).
3. **Categoria.** Choose one of the four mutually incompatible product families:Tondo, Quadro, Kube, Intelligente then "Salva categoria". Changing this later discards any column plan or design already produced, with a two-step confirmation dialog.

4. **Colonne.** Choose how many columns (1–8) and a width for each, picked from the widths actually available for the selected family, then "Salva piano colonne" (FR9: only compatible widths are offered, so an invalid combination cannot be composed in the first place).

5. **Design.** For each column, add or remove shelf levels one at a time ("+ Livello" / "- Ultimo"); only heights that keep the design physically valid are offered, and an explanatory reason is shown when a candidate height is unavailable (e.g. too close to another shelf, or exceeding the maximum height). For the Intelligente family, adding a level applies it to every column at once, automatically distinguishing outer columns ("BORDO") from inner ones ("INTERMEZZO"). The BOM panel and price update after every change (FR2, FR3).

6. **Finalizza.** Once the design is complete, "Finalizza e invia al carrello" (Finalise and send to cart) is enabled, moving the configuration to its terminal `FINALIZED` state and adding its BOM to the cart in one action (FR4).

7.3 Collaborating on a configuration

While editing, the owner can press "Collabora" to open a collaborative session: a short, unambiguous code is shown in a dialog with a "Copia codice" (copy) button.

Another user opens the same configuration's join box, types the code ("Entra con codice"), and immediately sees the same environment, category, column plan and design as the owner; from then on, edits from either participant appear on the other's screen in real time, resolved deterministically if both edit the same field at once. A status line ("Collaborazione attiva · N partecipanti") always shows who is connected; if the owner leaves, the session ends for everyone with an explicit notice, since the configuration itself only ever lives in the owner's account.

7.4 Saving and resuming configurations

The "Configurazioni" screen (route `/configurations`, `BASE` and `ADMIN` only) lists a logged-in user's own configurations with summary counters (total, drafts, ready to finalise, finalised, in progress) and, an "Apri in CAD" button to resume editing (FR6, FR7) and, once `FINALIZED`, a "Riordina" (reorder) button that adds its BOM straight to the cart again.

7.5 Cart and checkout

The cart (route `/cart`) lists every line item with its quantity (adjustable with +/- buttons) and running total, plus a "Svuota carrello" (empty cart) and a "Procedi al checkout" button (FR4). If a component in the cart becomes unavailable in the meantime, it is automatically removed and the user is told so, rather than being allowed to order something the catalog can no longer provide.

Checkout opens a "Dati spedizione" (shipping details) form: name, surname, email, country, city, postal code, address, phone, and an optional delivery note. Validated client-side before "Conferma checkout" finalises the order.

7.6 Notifications

Authenticated users see a bell icon with an unread-count badge in the header, updated both by polling and in real time as changes happen (NFR2). The "Notifiche" screen lists every notification (a price or availability change affecting a subscribed component, FR8) with a filter for unread-only and a button to mark one as read.

7.7 Chatbot

The chatbot (route `/chatbot`) is a dedicated chat screen (FR5). Messages are exchanged like in a normal mesasging app; a typing indicator shows while the bot is composing an answer, which can concern, for instance, the current price of a specific component by SKU.

7.8 Administration

`ADMIN` users additionally see three screens (FR10, FR11), hidden from every other role (section 7.9).

- **"Catalogo Admin"** (route `/admin/catalog`): add a new component (SKU, name, category, type, price, availability, description, dimensions, image, compatible category) through a modal form, or edit the price/availability of an existing one inline; both actions immediately trigger the catalog-change propagation.
- **"Gestione ordini"** (route `/admin/orders`): every order in the system, filterable by status, with actions to mark a `SUBMITTED` order as "DONE" or "CANCELLED" (guarded so a non-`SUBMITTED` order cannot be changed twice).
- **"Report ordini"** (route `/admin/reports`): a date-ranged sales dashboard (total orders, revenue, a per-day breakdown by status) computed with a MongoDB aggregation over the order history.

7.9 What each role can see

The header's navigation links and the router's own guards (section 4.1) enforce the same restriction consistently:

`GUEST` Configuratore, Chatbot only; no saved configurations, no cart history beyond the current session, no catalog browsing or admin screens.

`BASE` Configurazioni, Configuratore, Chatbot, plus cart and notifications; everything a registered customer needs, nothing administrative.

`ADMIN` every screen above, plus Catalogo, Catalogo Admin, Gestione ordini and Report; an admin's default landing screen is "Gestione ordini" rather than the configurator.

8 Self-evaluation

8.1 Valerio Giannini

Role. I am the sole author and developer of KompozeR. The project spans roughly three/four months of history (from the start of the scoping process in April to the final report revisions in July) and grew into the full microservice system documented here.

Strengths. Working alone on all layers forced the architecture to stay coherent end-to-end: the same hexagonal layering (`domain`/`useCases`/`adapters`) and the same conventions (UUID identifiers, cents-based pricing, OCC via a `version` field, section 4) are applied identically across all the backend components, because there was only one person deciding them. I am satisfied in particular with the collaborative-editing subsystem and with the CAD replica set failover, these are the two places where a genuinely distributed-systems mechanism, is implemented and automatically tested (section 5), including under an actual killed pod on Kubernetes. I also kept a disciplined commit history (consistent `FEAT`/`FIX`/ `TEST`/`DOC` prefixes throughout) and a real, if manually triggered, three-tier test suite rather than relying on manual testing alone.

Weaknesses. Being both the sole developer and the sole reviewer is itself a limitation: there was no second pair of eyes to catch design inconsistencies before they were written down, and several of the gaps weren't noticed in an early stage; most notably the absence of any continuous integration pipeline (tests are run manually, section 5), the complete lack of an automated test suite on the frontend, the single-attempt (no-retry) behaviour of most inter-service REST calls, and every Kubernetes Deployment fixed at one replica even though the stateless tier is designed to scale horizontally. Secret management for local/demo environments is also weaker than it should be and is the first thing I would fix outside the scope of this report. Time-boxing the project around the course deadline also meant prioritising getting all eight bounded contexts and both real-time subsystems working end-to-end over hardening any single one of them further; section 9 lists the concrete consequences of that trade-off.

9 Future works

Every functional requirement of section 2 is implemented and covered by the end-to-end suite of section 5, so there is no unimplemented feature to report; this chapter instead collects the concrete simplifications and gaps that sections 3 to 6 already flagged as deliberate scope decisions for the project but can be improved.

Process. The most impactful single addition would be a **continuous-integration pipeline** running `npm run test:baseline` and a frontend test suite on every push or pull request, since currently every test tier of section 5 is triggered manually. A close second is giving the **frontend an automated test suite** at all: component/unit tests for the larger views to replace the manual UI walkthrough for at least the core paths (configurator, checkout, collaborative session).

Reliability. Inter-service REST calls that are currently single-attempt should implement some retry mechanisms. Kubernetes **liveness** probes (only readiness probes exist today, section 3.7) would let the platform recover a pod that is up but stuck, not just one that has not started yet.

Performance and scalability. The stateless tier is architecturally ready to scale horizontally but every Kubernetes `Deployment` is currently pinned at one replica; the best practice would be to raising replica counts and adding a `HorizontalPodAutoscaler` on CPU/latency.

Security. Three items from section 3.9 are natural next steps: **TLS termination** (an ingress with a certificate, so traffic is no longer plain HTTP even inside the cluster boundary), **schema-based input validation** to replace the hand-written field checks currently duplicated per use-case, and **rate limiting** on the authentication endpoints to blunt credential-stuffing/brute-force attempts. More generally, secret management for anything beyond local development need to be more strict before any real deployment.

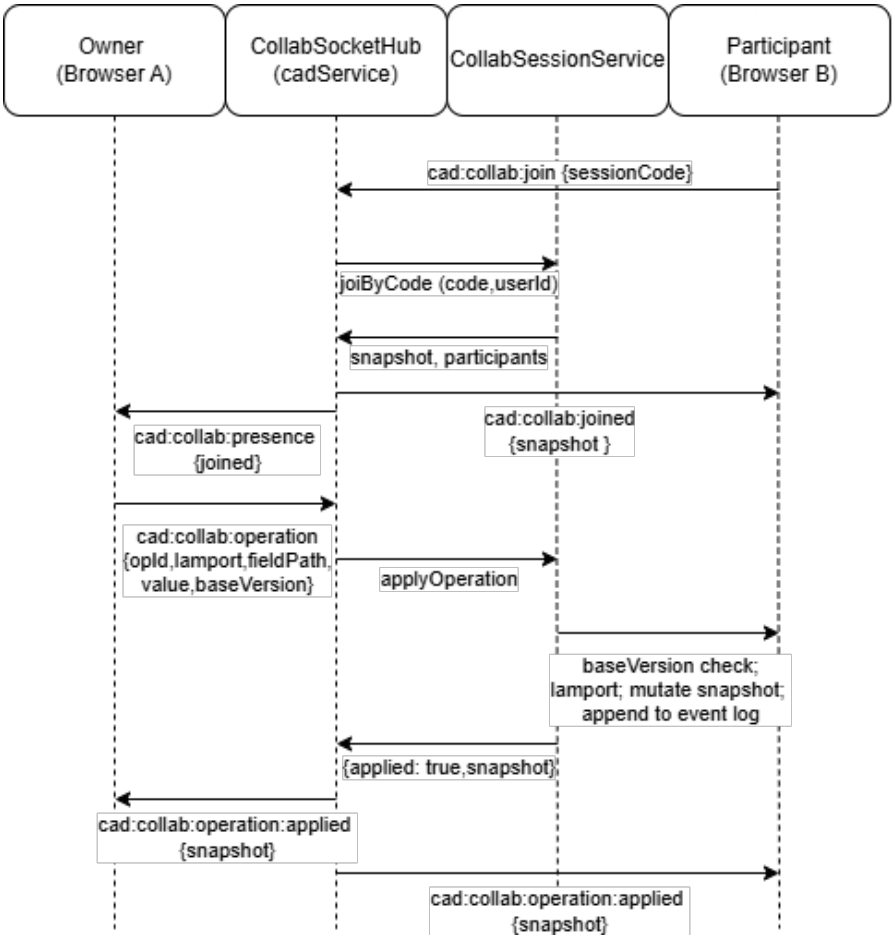

Figure 5: A collaborative CAD session: a participant joins with a short session code, then owner and participant exchange field-level operations broadcast to the whole room (`cad:collab:{sessionCode}`). The session service is the single authority for conflict resolution; both browsers converge to the same snapshot because every operation is applied through the same deterministic rule.

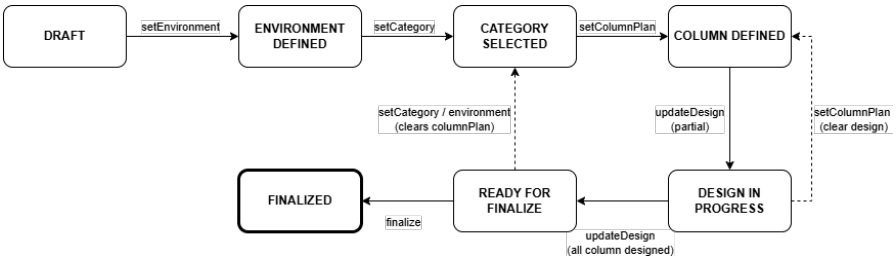

Figure 6: Lifecycle of a CAD `Configuration`. Solid arrows are the intended forward path (also enacted, field by field, by collaborative operations); dashed arrows summarise the reset rule that regresses status and discards downstream data whenever an earlier-stage field is rewritten. `FINALIZED` (bold border) is a terminal state.

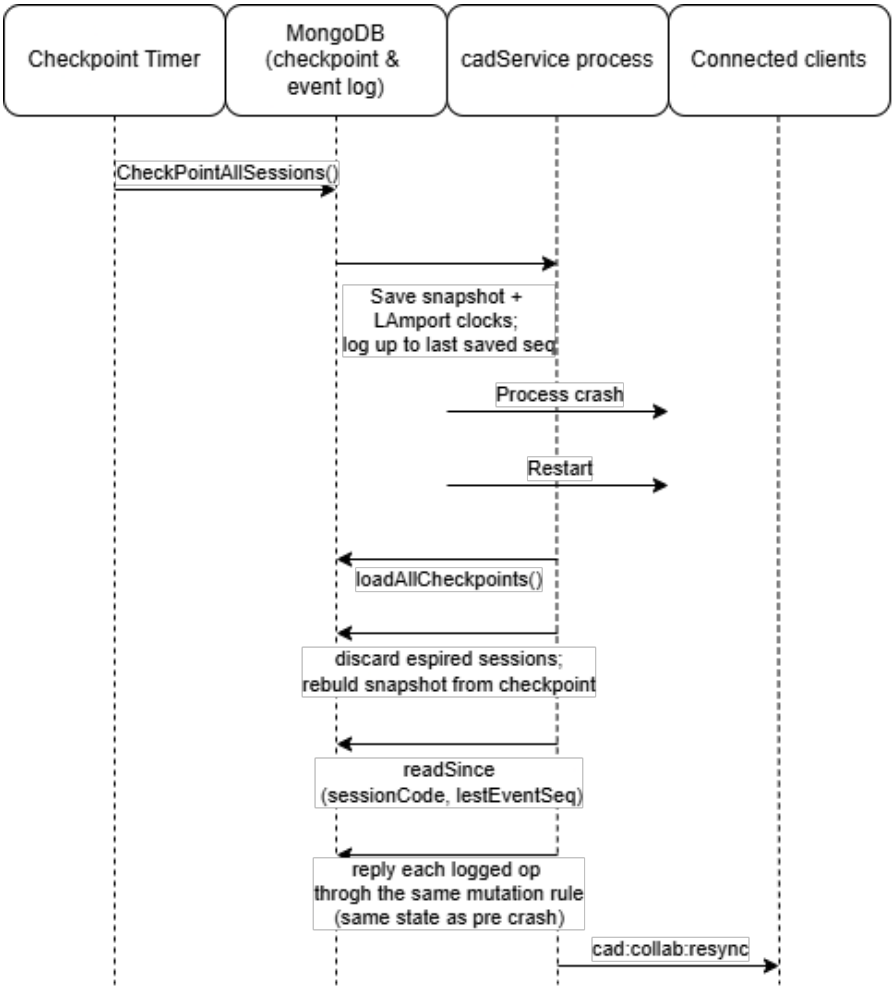

Figure 7: Checkpoint and crash recovery for a CAD collaborative session. The session lives in process memory while active; a periodic checkpoint plus an append-only event log since the last checkpoint let a restarted process rebuild exactly the pre-crash state, after which clients are told to resynchronise.

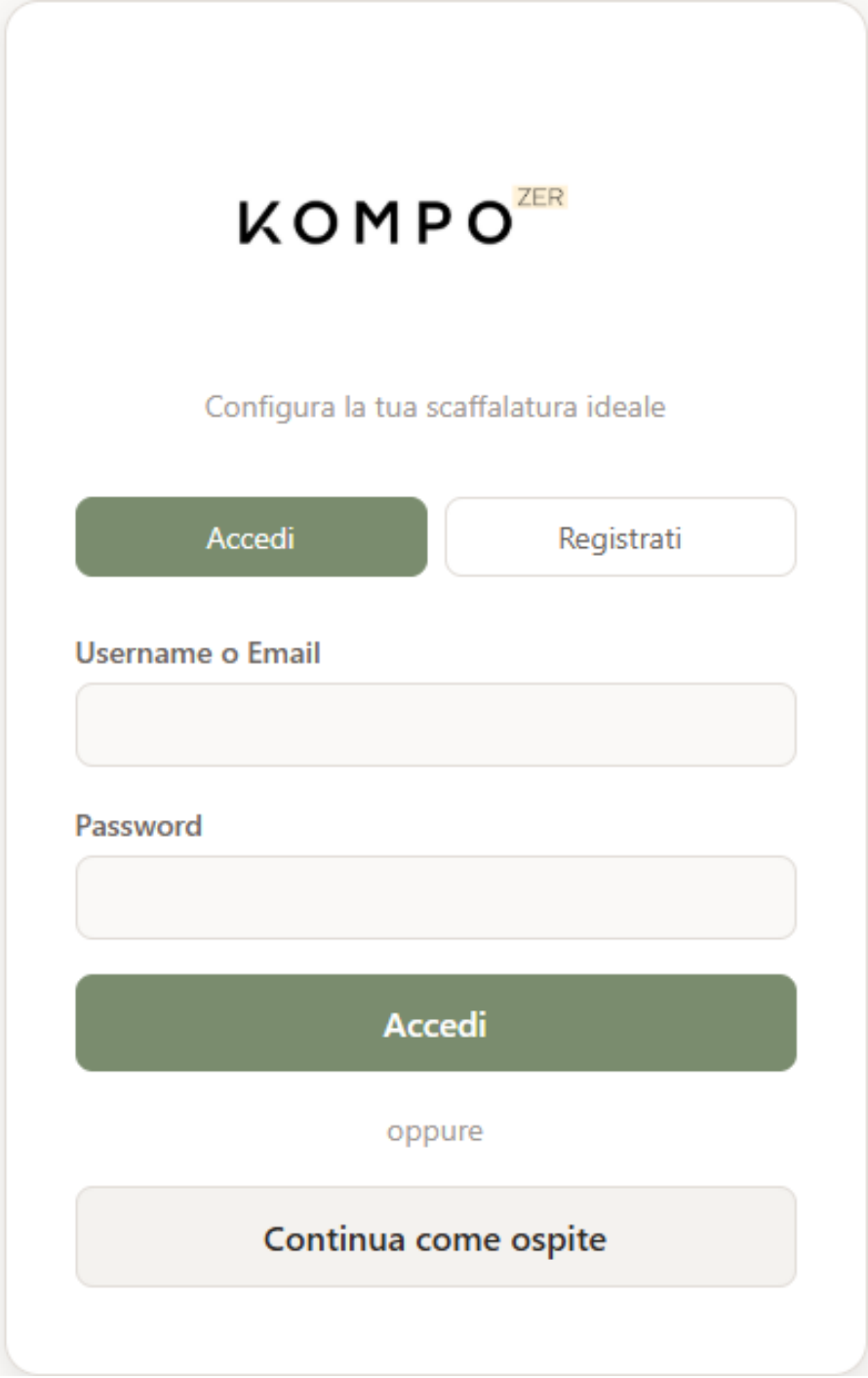

Figure 8: AuthView: card login/registration with Guest button.

Figure 9: AppHeader: Navigation ber for BASE user.

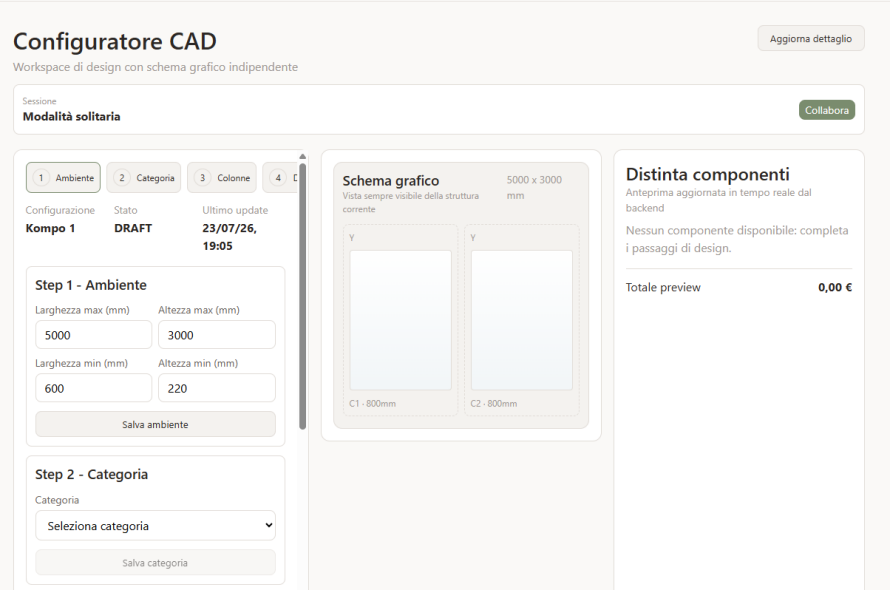

Figure 10: CadView: the five-step wizard with the BOM/preview panel.

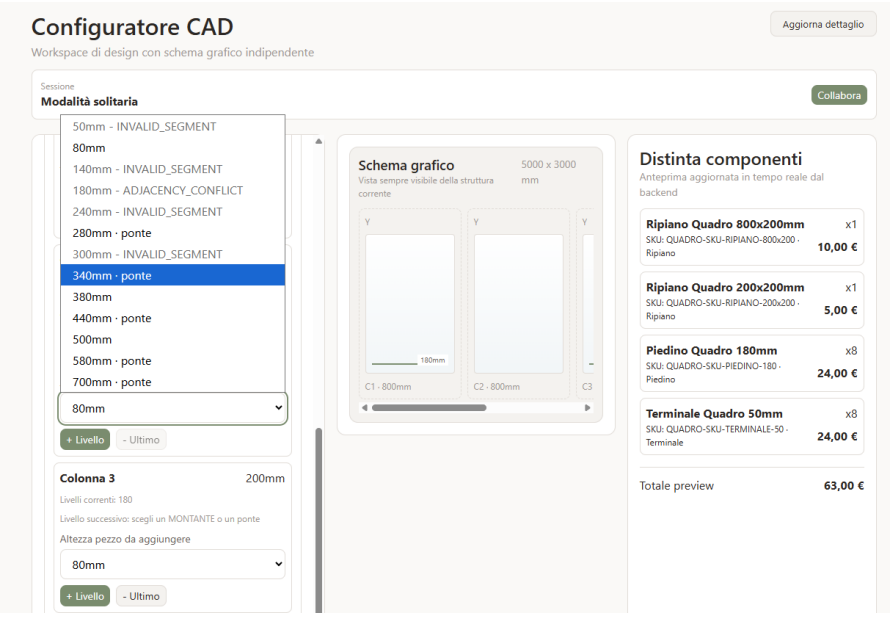

Figure 11: CadView step 4: level-by-level design with the reason codes.

Figure 12: CadView: the "Sessione collaborativa attiva" dialog with the session code.

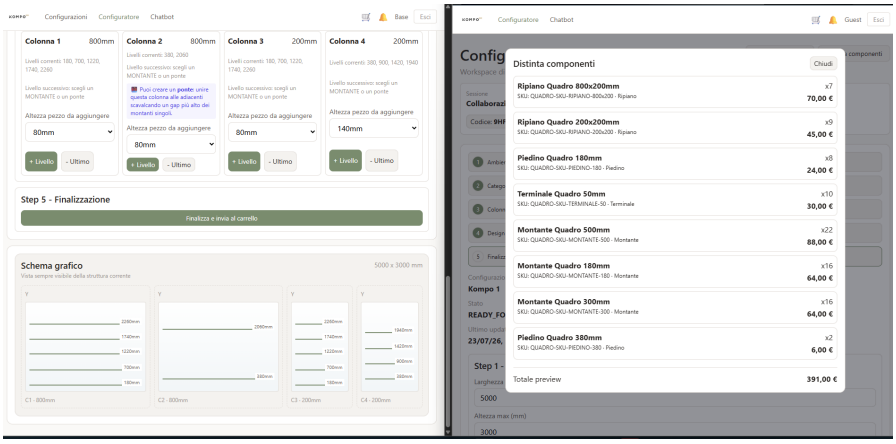

Figure 13: CadView: two browser windows showing the same configuration in sync.

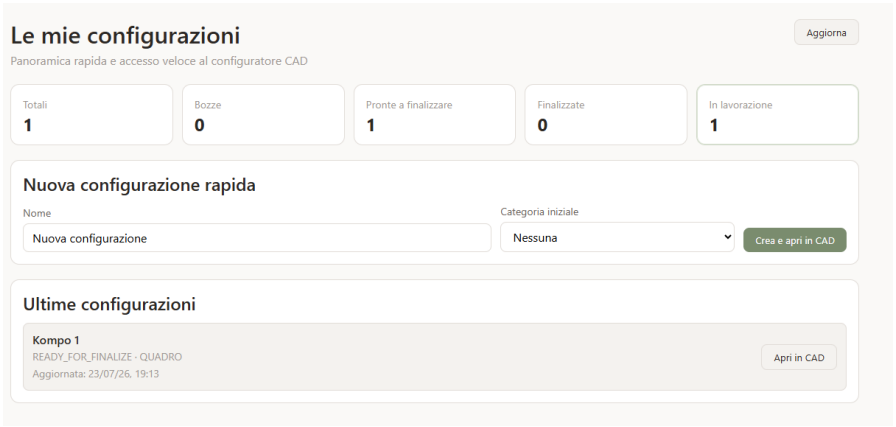

Figure 14: ConfigurationsView: the list of saved configurations with KPI cards.

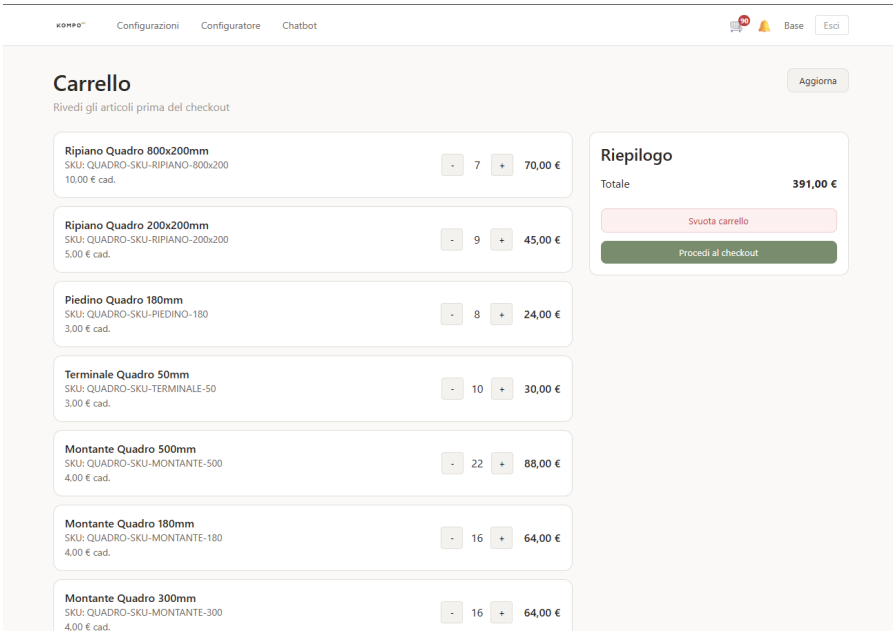

Figure 15: CartView: line items and the checkout summary panel

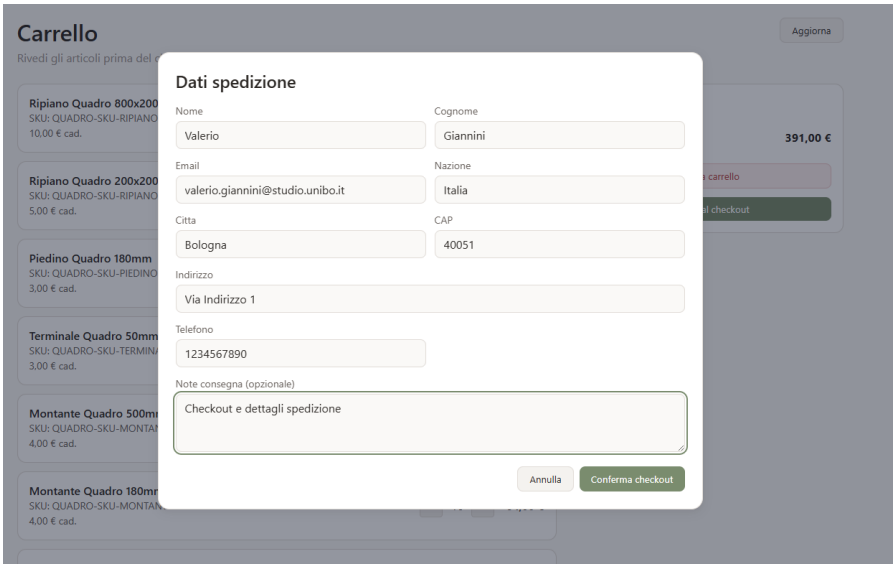

Figure 16: CartView: the shipping-details checkout modal.

Figure 17: NotificationsView — the notification list.

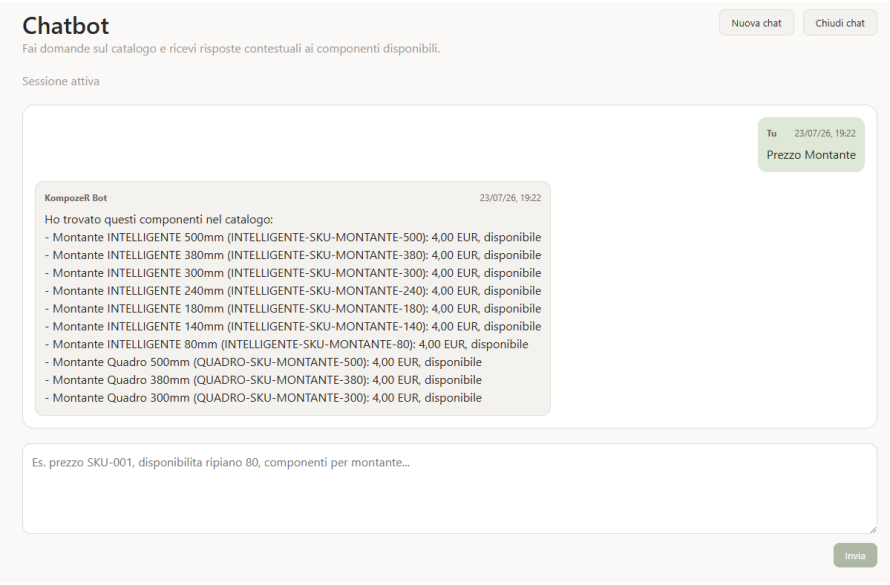

Figure 18: ChatbotView: example of a conversation.

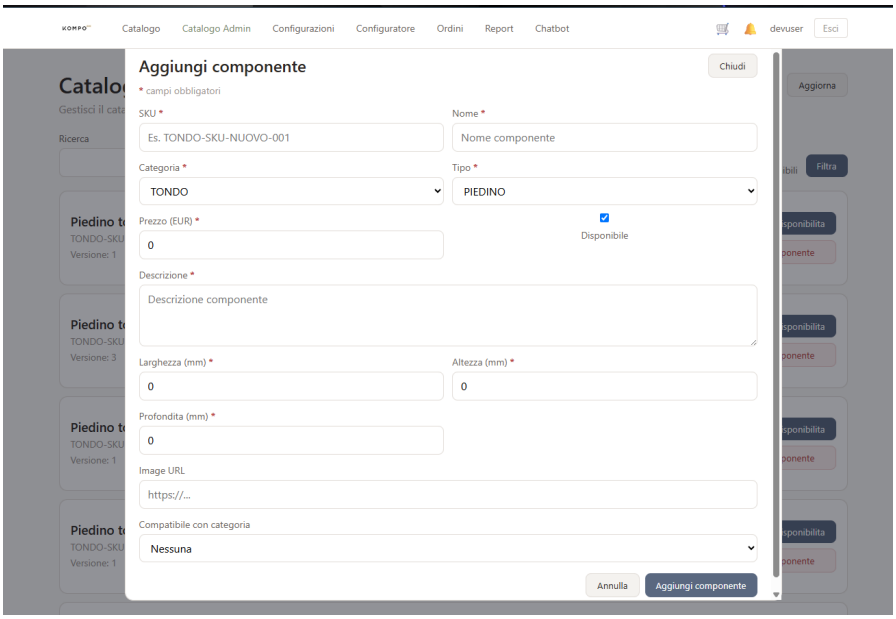

Figure 19: AdminCatalogView: the component list and the "add component" modal.

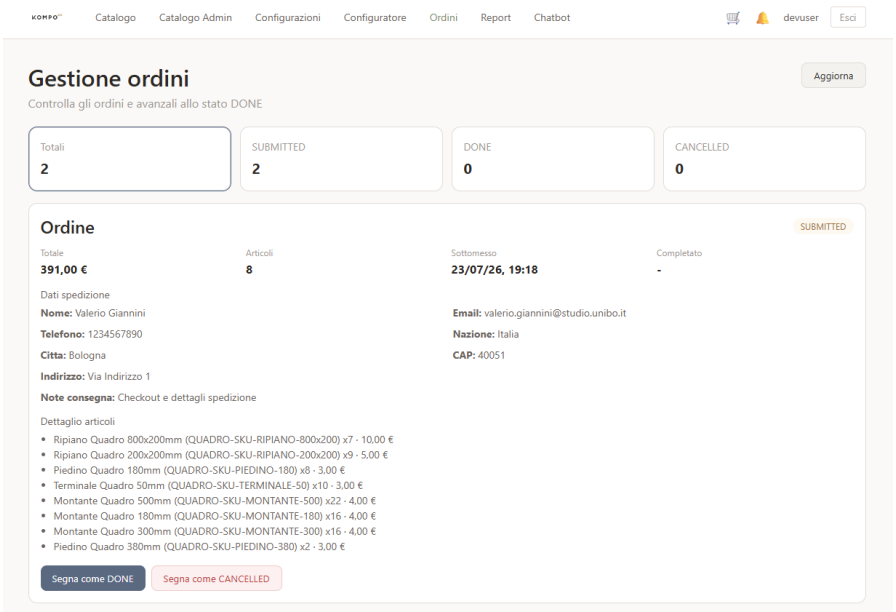

Figure 20: AdminOrdersView: the order list with status filters.

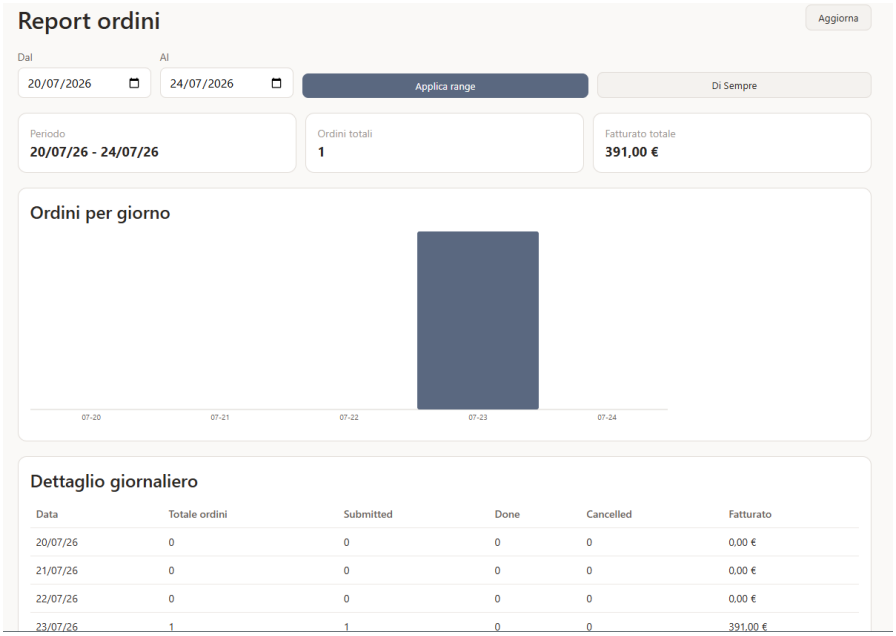

Figure 21: AdminReportsView: the KPI cards and the per-day chart.