QuickCart

PRODUCT & ENGINEERING DOSSIER · v1.0 · Sept 2026

The quick-buy marketplace for *everything*.

One search box. Five sourcing platforms. QuickCart finds any product across Blinkit, Zepto, BigBasket, Flipkart and Amazon, compares live price, stock and delivery ETA, and gets it to your door — fast.

Quick-commerce aggregation | MCP adaptor layer | Node.js + PostgreSQL | Best-price engine

WHAT THIS DOSSIER CONTAINS

Six documents, one system

Each document is self-contained and print-ready. Together they describe the full product — from the screens a shopper sees, to the Node.js services that fan out to five platforms and rank the results, to the cross-platform basket QuickCart fulfils as a single order.

DOCUMENT 01

UI Mockups — Home, List & Detail

High-fidelity phone mockups of the three core screens, annotated with the data each element is bound to.

DOCUMENT 02

User Flow (Mermaid)

Rendered flow diagrams: discovery → compare → checkout → track, including the missing-SKU capture loop.

DOCUMENT 03

System Architecture

How the mobile app talks to the Node backend and database; catalogue management, missing-SKU discovery, analytics and payments.

DOCUMENT 04

MCP Integrations

Every platform MCP (Blinkit, Zepto, BigBasket, Flipkart, Amazon), the tools each exposes, and exactly how far we can use them.

DOCUMENT 05

Price & Availability Engine

How the backend gathers, normalises and ranks best price, stock and ETA across sources — with the algorithm and code.

DOCUMENT 06

Orders & Delivery

One cross-platform basket → one internal order: procurement split across sources, saga orchestration, and last-mile delivery.

CROSS-CUTTING

Analytics & Payments

Event model, KPI dashboards and the payment/settlement flow live inside the Architecture document.

The idea in one picture

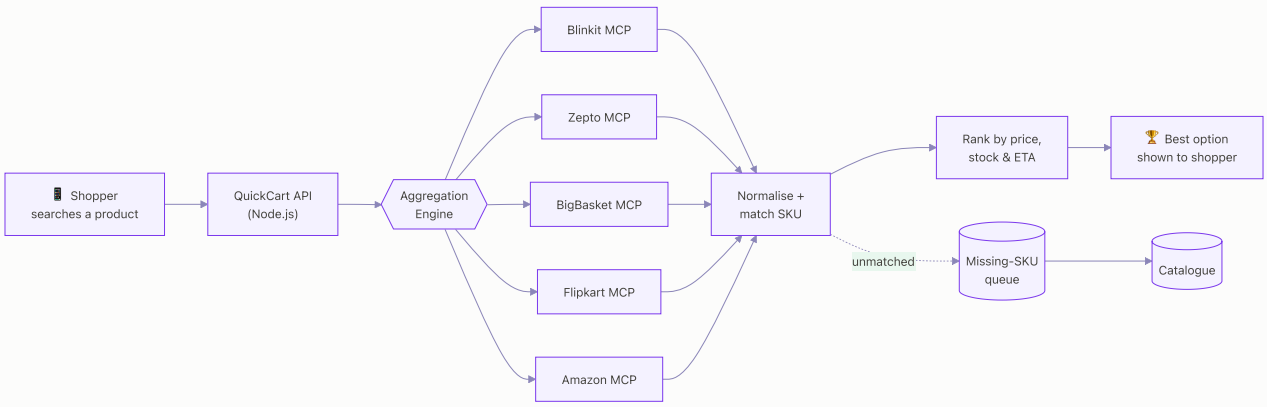


Figure 0.1 — QuickCart at a glance: one query fans out to five platform MCPs, results are normalised, matched and ranked, and gaps feed catalogue growth.

Snapshot targets

5
SOURCE PLATFORMS

<1.5s
SEARCH P95 LATENCY

10 min
MEDIAN DELIVERY ETA

98%
SKU MATCH COVERAGE GOAL

Targets are design goals for v1, not measured production numbers.

How to read this dossier

Open any document from the top navigation. Every page is styled for both screen and print — use `convert-to-pdf.sh` (or your browser's *Print* → *Save as PDF*) to export a pixel-accurate PDF with all diagrams and charts baked in.

DOCUMENT 01 · UI MOCKUPS

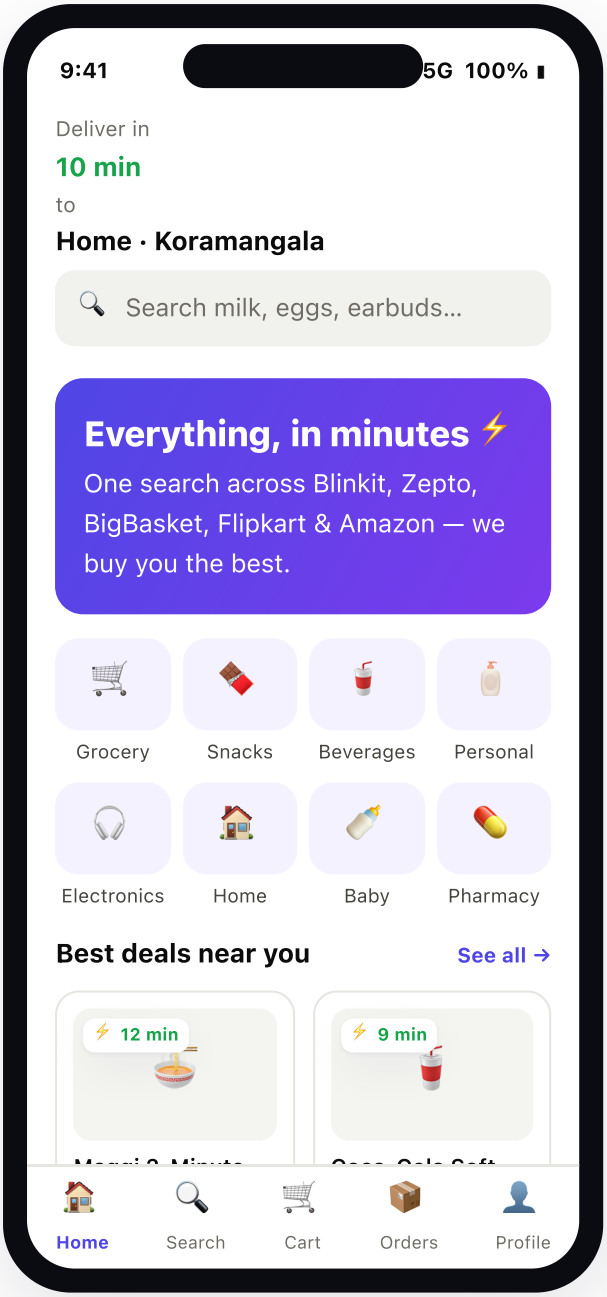
Home, List & Detail

High-fidelity mockups of the three core screens a shopper touches — and a map of exactly which backend field each pixel is bound to.

3	1	5→1
CORE SCREENS	SEARCH BOX, FIVE SOURCES	SOURCES COMPARED INLINE

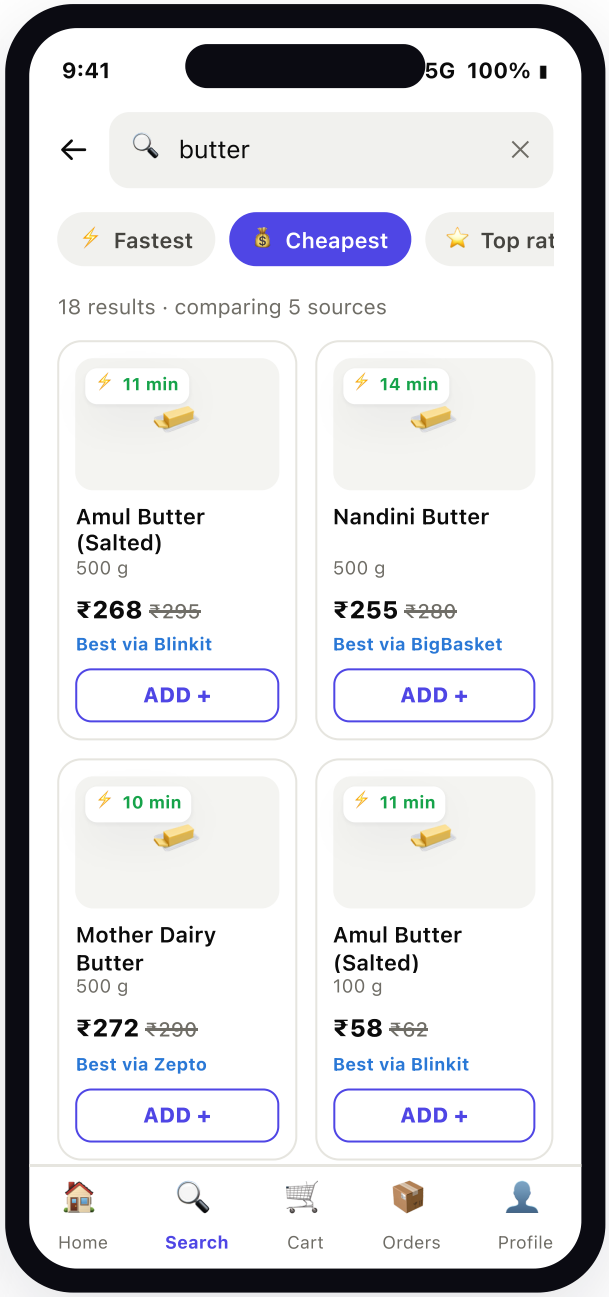
The product is deliberately simple on the surface: search or browse on **Home**, scan ranked results on the **List**, and see the five-source price/stock/ETA comparison on the **Detail** screen. Everything hard happens in the backend (Doc 05).

1 The three core screens



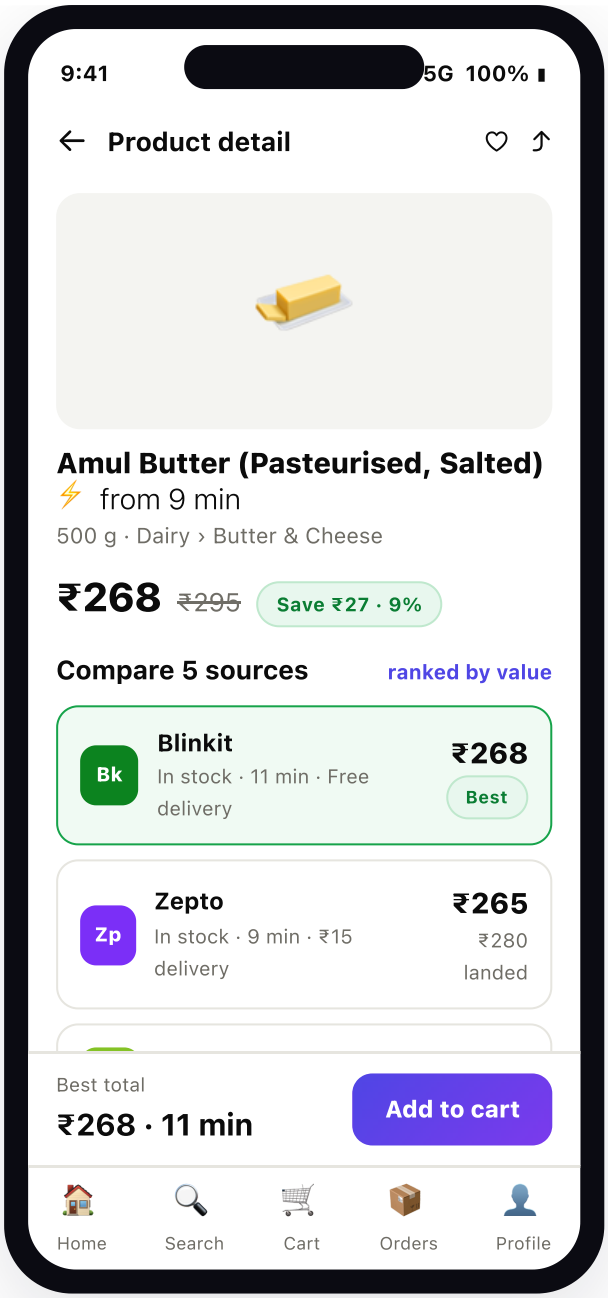
Home

Location · search · categories · deals



List — search “butter”

Ranked results · filter chips · cheapest source per item



Detail — Amul Butter 500g

Price · MRP · savings · five-source compare · add

Prices, ETAs and stock are illustrative. Screens shown at ~1× phone width.

2 What each element is bound to

Every visible value maps to a field the backend produces. The comparison block is the ranked `offers[]` array from the Price & Availability Engine (Doc 05), and each source's numbers come through its adaptor in MCP Integrations (Doc 04).

UI ELEMENT	BOUND TO	SOURCE
Delivery ETA badge ⚡ 11 min	`offer.etaMinutes` (per zone)	Winning source MCP → Doc 04
Struck-through MRP `₹295`	`offer.mrpPaise / 100`	Source listing → Doc 04
Live price `₹268`	`offer.pricePaise / 100` (effective, post-offer)	Doc 05 §Normalise
"You save ₹27 · 9%"	`mrpPaise − pricePaise` → %	Computed client-side
Source label "Best via Blinkit"	`response.best` (winning `sourceId`)	Doc 05 §Ranking
"Compare 5 sources" list	ranked `offers[]`, sorted by `score`	Doc 05 §Ranking
Filter chips (Fastest / Cheapest…)	ranking `weights` override	Doc 05 §Signals
"18 results · comparing 5 sources"	`sourcesReturned` / `sourcesQueried`	Doc 05 §Scatter-gather
Search box	`GET /v1/search?q=`	Doc 03 §API surface
"excluded" / out-of-stock row	`offer.inStock === false`	Doc 05 §Ranking (dropped from live set)

⚡

ETA is a promise, per zone

The badge reflects the hyperlocal ETA for the shopper's delivery zone — not a global figure — because quick-commerce stock and dispatch are store-specific.

🏆

"Best" ≠ cheapest

The winning source is chosen by the weighted score (price + ETA + stock + reliability), so free delivery and a 11-min ETA can beat a lower sticker price.

🔁

Re-validated at checkout

Numbers on Detail are cached (30–60s TTL). The winning offer is re-fetched live before payment so the shopper never pays a stale price.

3 Notes for engineering & design

> **These mockups are static HTML + CSS on purpose**
>
> No images or JavaScript are used — emoji stand in for product/category art — so the screens render *identically* in the browser and in the exported PDF. The live app is **React Native** (see Architecture, Doc 03); these frames define layout, hierarchy and data bindings, not the production component code.

Design tokens used

- Brand indigo #4f46e5
- Brand violet #7c3aed
- Quick green #16a34a — ETA & savings
- Ink #0b0b0b / muted #6f6d66

Source brand chips on the Detail screen use each platform's own colour (Blinkit green, Zepto violet, BigBasket lime, Flipkart blue, Amazon amber) purely to aid recognition; QuickCart chrome stays indigo/violet.

QuickCart

DOCUMENT 02 · USER FLOW

From search to doorstep

The complete shopper journey — discover, compare five sources, buy and track — plus the request lifecycle behind the scenes and the loop that turns a failed search into new catalogue.

4	1 tap	<1.5s
CORE JOURNEY STAGES	TO COMPARE 5 SOURCES	TO FIRST RANKED RESULTS

QuickCart collapses "check five apps" into one action. Everything below is written from the shopper's point of view first, then unpacked into the system behaviour that makes each step fast and reliable.

1 The end-to-end journey

A shopper opens the app, finds a product by browsing or searching, sees the best offer across all connected platforms, and either checks out in-app or is handed off to the source app. If nothing matches, the query is captured so the catalogue can grow — the shopper never hits a dead end.

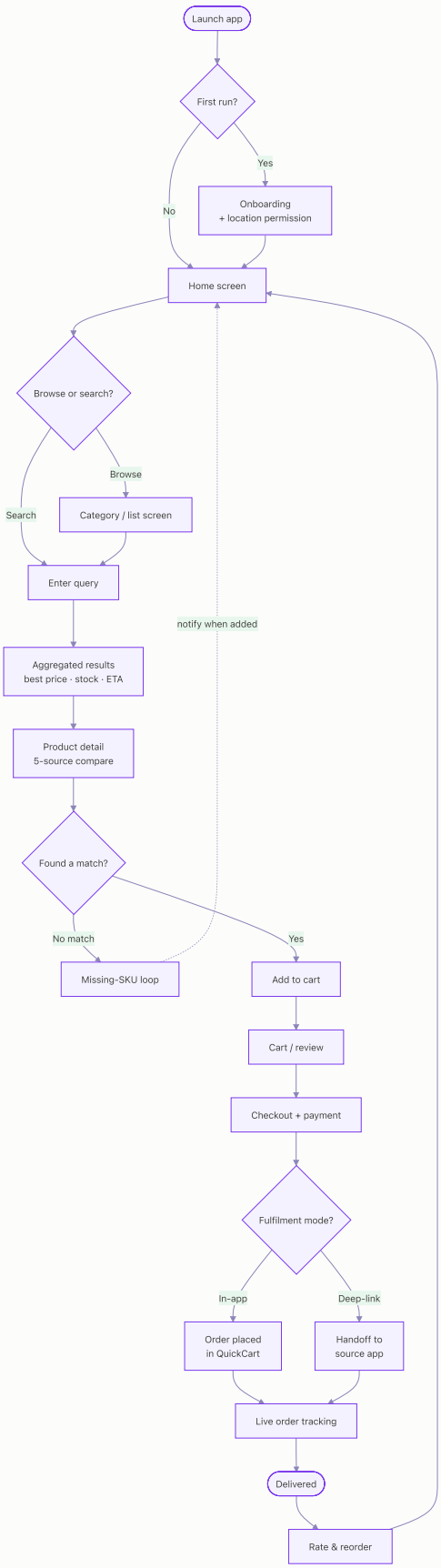


Figure 2.1 — The complete shopper journey. The "no match" branch feeds the missing-SKU loop (Figure 2.3) rather than dead-ending.

2 Search request lifecycle

Behind a single search, the API gateway checks a short-lived cache and, on a miss, fans out to all five platform MCPs in parallel. Results are normalised, matched to canonical SKUs and ranked before returning. A hard deadline guarantees the shopper sees *something* even if one source is slow.

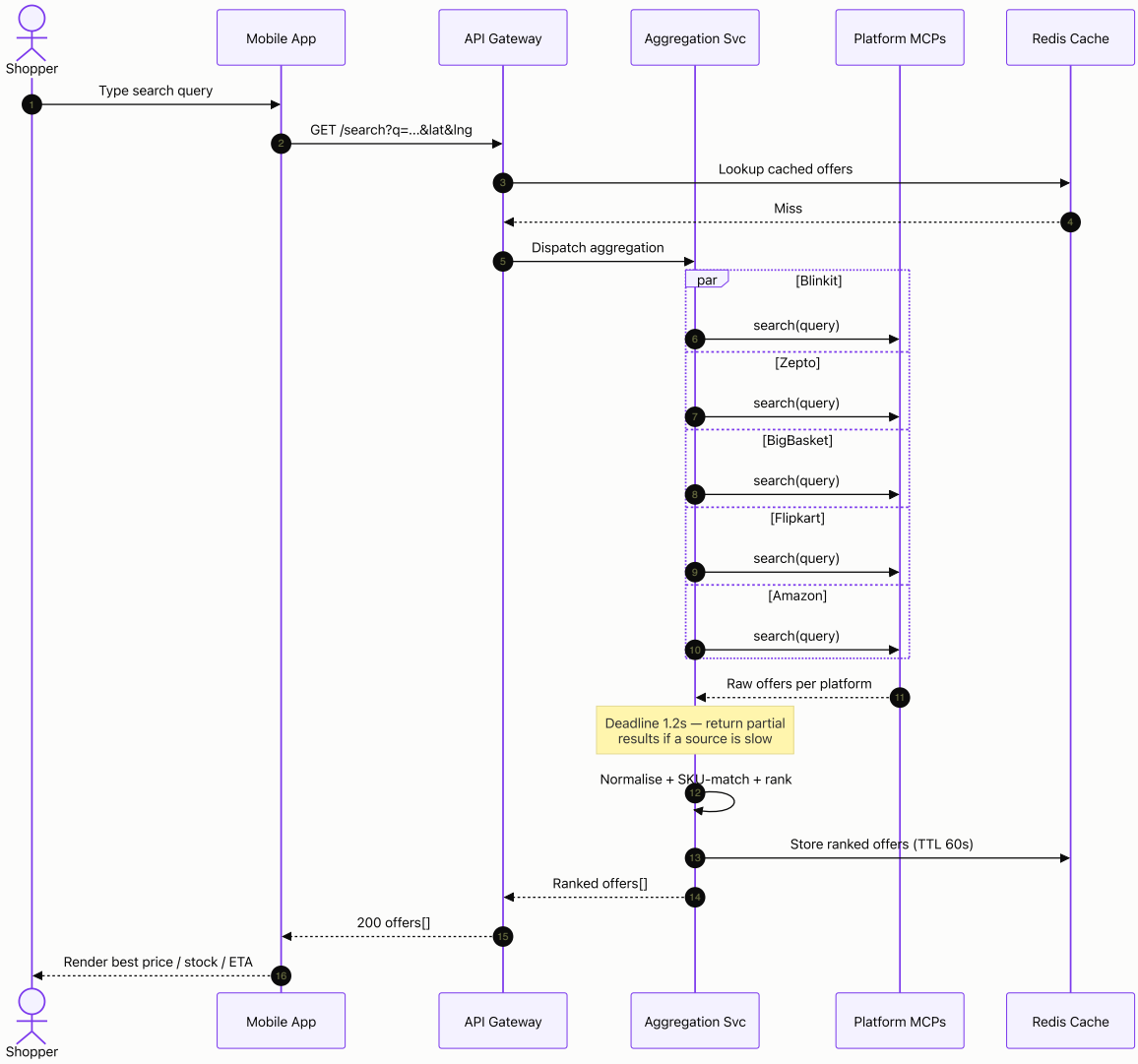


Figure 2.2 — Search request lifecycle. Scatter-gather to five MCPs with a 1.2s deadline; see Document 05 for the ranking algorithm.

Partial results are a feature, not an error

If a source misses the 1.2s deadline it is dropped from *this* response and refreshed asynchronously. The UI labels the offer set "showing 4 of 5 sources" rather than blocking on the slowest platform.

3 Missing-SKU capture loop

When a search returns nothing (or only weak matches), the query is never thrown away. It enters a queue that clusters similar misses, enriches them from platform browse endpoints and external product data, and either auto-creates a canonical catalogue entry or routes it to human review. The full data model lives in Document 03 — Missing-SKU Discovery.

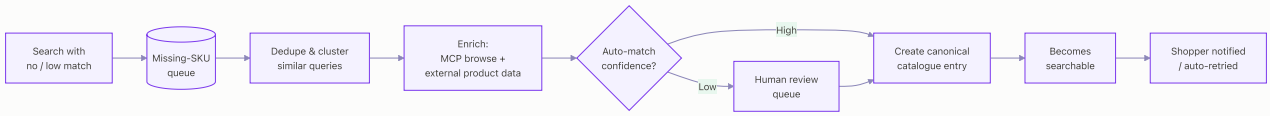


Figure 2.3 — Missing-SKU capture loop. Demand signal (real searches) drives catalogue growth; low-confidence matches get a human in the loop.

4 Order lifecycle

Once a shopper checks out, the order moves through a well-defined state machine owned by the Order service, with payment and refunds coordinated by the Payment service. Every transition emits an analytics event (see Document 03 — Analytics).

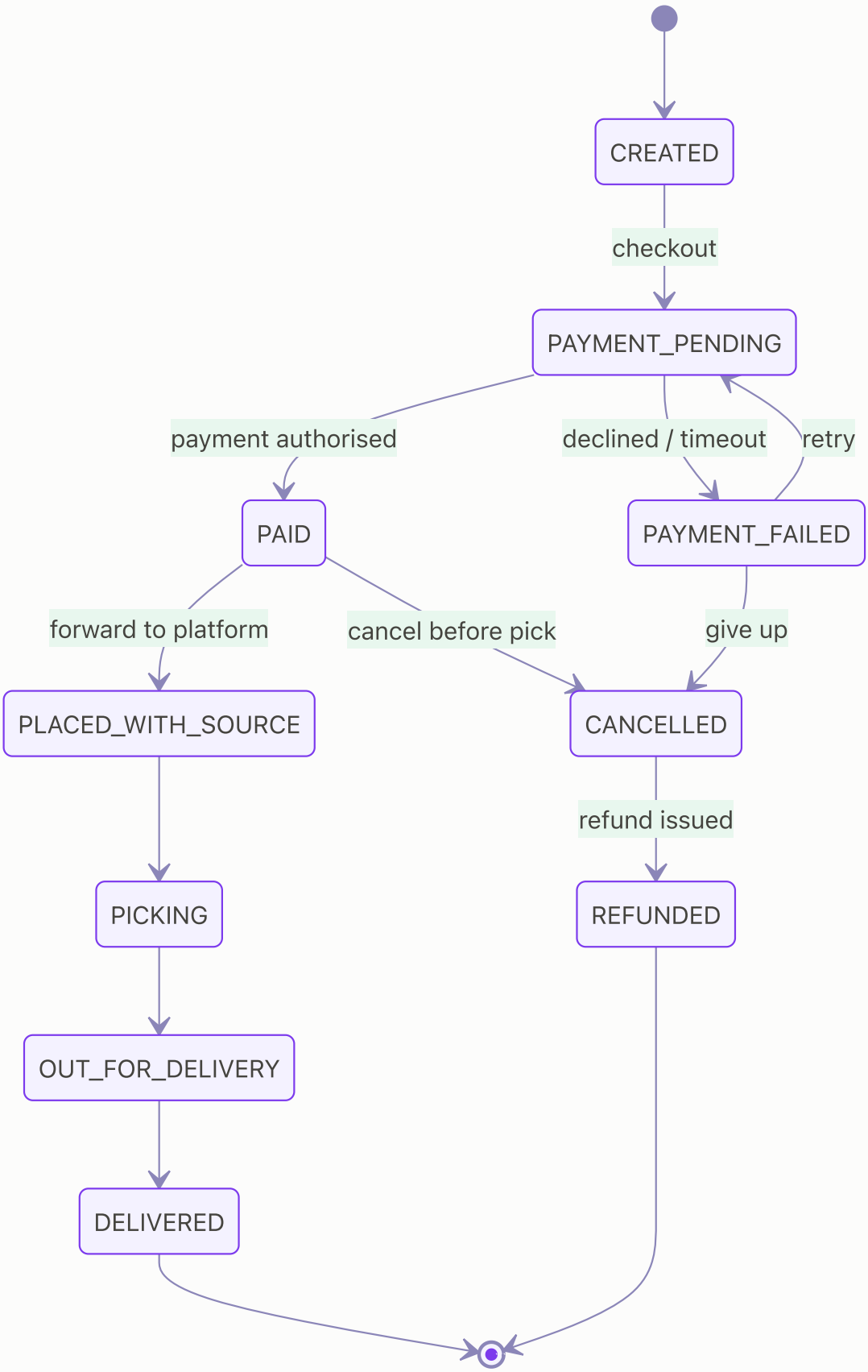


Figure 2.4 — Order state machine. PAYMENT_FAILED is recoverable; a cancellation after payment always routes through REFUNDED.

5 Who owns each stage

Journey stage → backend service

Search / compare → Aggregation service + Catalogue service (Doc 05, Doc 03) · **Source data & live stock** → Platform MCP adaptors (Doc 04) · **Cart / order state** → Order service (Doc 03) · **Payment / refund** → Payment service (Doc 03) · **No-match** → Missing-SKU pipeline (Doc 03).

6 Edge cases we handle

The happy path is easy; the value is in the failure paths. Each of these has an explicit branch in the flows above.

● No results

Nothing matched

Show related suggestions, log the query to the missing-SKU queue, and offer "notify me when available."

● Partial outage

A source is down or slow

Drop the source past the 1.2s deadline, badge results "4 of 5 sources," and backfill on the next refresh.

● Stale offer

Price or stock changed at checkout

Re-validate the winning offer before payment; if it moved, show a one-tap "accept new price" or re-rank.

● Coverage

Address outside all delivery zones

Detect at location step; fall back to standard-delivery sources (Amazon/Flipkart) or waitlist the pincode.

Payment

Payment failure & retry

Keep the cart, hold the order in `PAYMENT_PENDING`, allow a different method, and auto-expire after N minutes.

Ambiguity

Duplicate / ambiguous match

When two SKUs match the query closely, present a disambiguation card instead of guessing the wrong pack size.

DOCUMENT 03 · SYSTEM ARCHITECTURE

How QuickCart is built

The end-to-end system — from the React Native app on a shopper's phone, through the Node.js services and the `mcp-adaptor` aggregation layer, down to PostgreSQL, Redis and OpenSearch — plus catalogue management, missing-SKU discovery, analytics and payments.

8	3+	<1.5s	5
NODE.JS SERVICES	POSTGRESQL · REDIS · OPENSEARCH	SEARCH P95 LATENCY	MCP SOURCES

QuickCart is a **meta-marketplace**: it holds a canonical product catalogue but sources live price, stock and delivery-time from five platforms — Blinkit, Zepto, BigBasket, Flipkart and Amazon — through a single aggregation service, the `mcp-adaptor`. The architecture below is organised around that fan-out.

1 System context

A thin, presentation-only mobile client talks to a single **API Gateway / BFF**, which composes responses from a set of focused Node.js services. Only the `mcp-adaptor` service ever speaks to external platform MCPs; every other service is internal and stateless where possible.

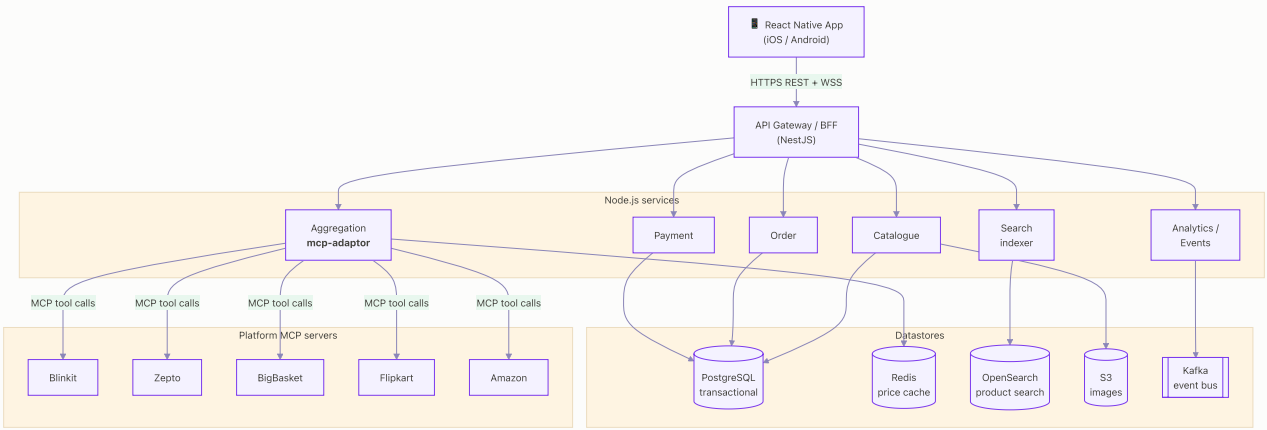


Figure 3.1 — System context. The app has one entry point (the gateway); only `mcp-adaptor` reaches external platforms.

Design principle — one aggregation boundary

Every external dependency (rate limits, outages, ToS constraints) is contained inside `mcp-adaptor`. The rest of the system treats "an offer" as a normalised internal object, so the product experience is unchanged whether a source is fast, slow or temporarily down.

2 How the mobile app connects to the backend

The client is a **React Native** app. It never talks to a database or to a platform MCP directly — it speaks only to the gateway over three channels:

🔗 HTTPS REST + JSON

The default. Versioned under `/v1`, cursor-paginated lists, ETag caching. A GraphQL BFF endpoint is optional for screens that need to compose many resources in one round-trip.

📡 WebSocket / SSE

Live order tracking (`WS /v1/orders/:id/track`) pushes courier location and status transitions without polling. Falls back to SSE where sockets are blocked.

🔐 Auth & transport security

OTP login → short-lived **JWT access token** + rotating **refresh token**. TLS 1.3 everywhere, certificate pinning in the app, secrets in a vault.

Connection & auth handshake

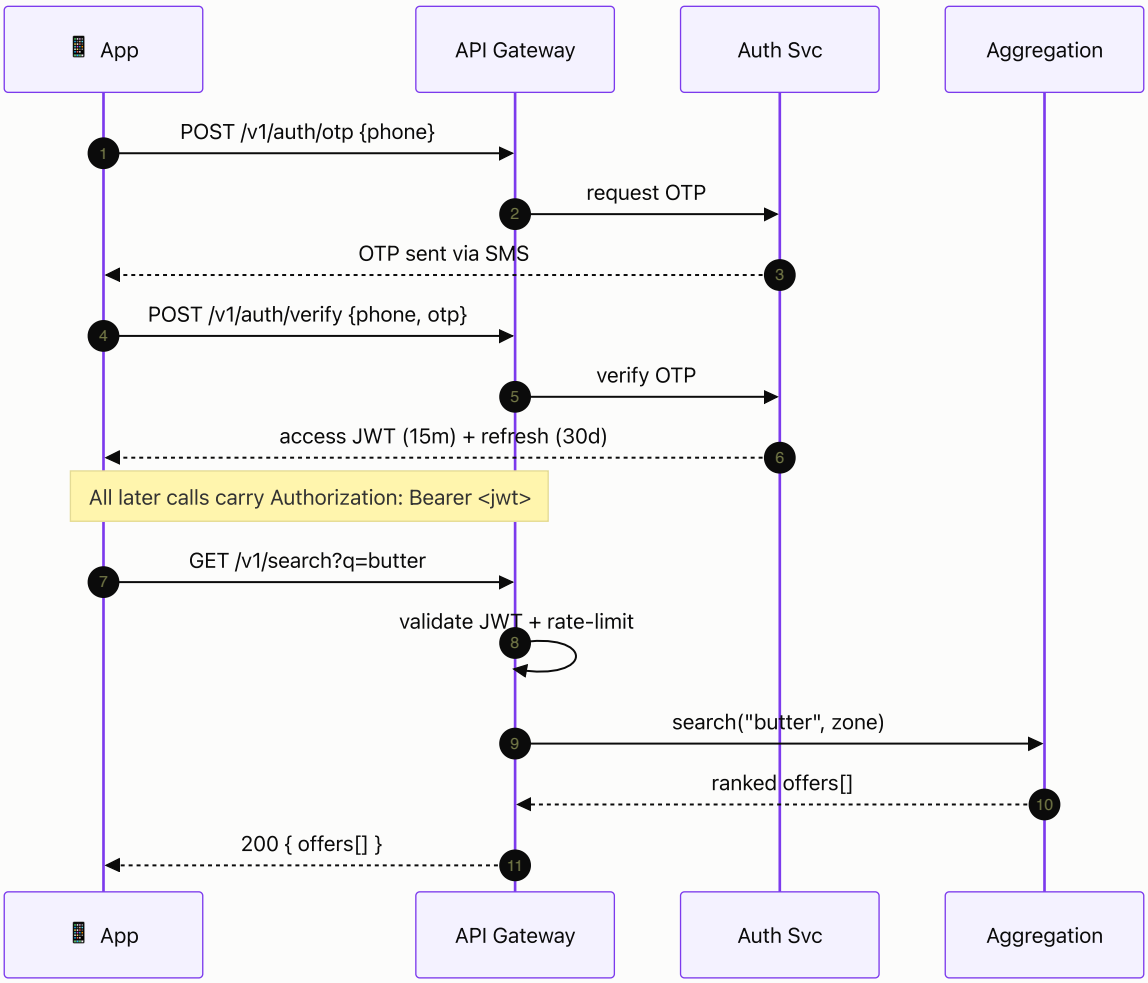


Figure 3.2 — OTP login issues a JWT pair; every subsequent request is validated at the gateway before fan-out.

Key API surface

METHOD & PATH	PURPOSE	NOTES
`POST /v1/auth/otp` · `/verify`	Passwordless login	Issues JWT access + refresh
`GET /v1/search?q=`	Search products	Returns canonical products + best offer
`GET /v1/products/:id/offers`	All source offers for one product	Ranked; cached in Redis (see Doc 05)
`POST /v1/cart` · `PATCH /v1/cart`	Cart management	Server-authoritative cart
`POST /v1/orders`	Place an order	Requires `Idempotency-Key` header
`GET /v1/orders/:id`	Order detail	Snapshot state
`WS /v1/orders/:id/track`	Live tracking	Status + courier position stream

An auth-guarded route

`services/gateway/routes/offers.ts`

```
import { Router } from "express";
import { authGuard } from "../middleware/authGuard";
import { aggregation } from "../clients/aggregation";

const router = Router();

// GET /v1/products/:id/offers  — ranked offers for one canonical product
router.get("/v1/products/:id/offers", authGuard, async (req, res, next) => {
  try {
    const { id } = req.params;
    const zone = req.user.deliveryZone;              // from JWT claims
    const offers = await aggregation.getRankedOffers(id, zone);
    res.set("Cache-Control", "private, max-age=30"); // matches Redis TTL
    res.json({ productId: id, offers });
  } catch (err) { next(err); }
});

export default router;
```

`services/gateway/middleware/authGuard.ts`

```
import jwt from "jsonwebtoken";

export function authGuard(req, res, next) {
  const header = req.headers.authorization ?? "";
  const token = header.startsWith("Bearer ") ? header.slice(7) : null;
  if (!token) return res.status(401).json({ error: "missing_token" });
  try {
    req.user = jwt.verify(token, process.env.JWT_PUBLIC_KEY, { algorithms:
["RS256"] });
    next();
  } catch {
    return res.status(401).json({ error: "invalid_token" });
  }
}
```

Resilience built into the client contract

Orders require an `Idempotency-Key` so a retried request never double-charges. The app keeps an offline read cache of the catalogue and the last cart, so browsing degrades gracefully on a flaky connection.

3 Backend services

Each service owns one responsibility and (where it is stateful) one datastore. They communicate synchronously through the gateway for reads and asynchronously over Kafka for events.

API Gateway / BFF · NestJS

Auth, rate-limiting, request validation, response composition, WebSocket fan-out. The only public surface.

Aggregation · `mcp-adaptor`

Scatter-gathers to the five platform MCPs, normalises fields, matches to canonical SKUs, ranks offers. Cache in Redis. *Detailed in Doc 05.*

Catalogue

Owns canonical products, brand/category taxonomy, images and the platform `source_map`. Writes to PostgreSQL + S3.

Search indexer

Consumes catalogue changes, builds the OpenSearch index (analyzers, synonyms, typo-tolerance), serves query→product resolution.

Order

Cart→order state machine, idempotent placement, source hand-off / managed fulfilment, live tracking events.

Payment

PSP integration (Razorpay), webhooks, refunds, settlement & reconciliation. PCI-minimised via tokenisation.

Analytics / Events

Ingests the client + server event stream into Kafka, powers funnels, KPIs and the missing-SKU signal.

Notification

Push (FCM/APNs), SMS and email for OTP, order updates and back-in-catalogue alerts.

Technology stack

LAYER	TECHNOLOGY	WHY
Mobile client	React Native + TypeScript	One codebase, native performance, OTA updates
API / services	Node.js 20, NestJS, TypeScript	Async I/O suits high-fan-out aggregation; strong typing
Transactional DB	PostgreSQL 16	ACID for orders/payments, JSONB for flexible attributes
Cache / hot data	Redis 7	Sub-ms price/stock cache, rate-limit counters, sessions
Search	OpenSearch	Typo-tolerant, synonym-aware product search
Object storage	S3 + CDN	Product images, served at the edge
Event bus	Kafka	Decouples analytics, search indexing, missing-SKU capture
Warehouse	ClickHouse / BigQuery	Funnels, KPIs, cohort analysis
Infra	Kubernetes, containers	Horizontal scale, rolling deploys, autoscaling
Payments	Razorpay (UPI/cards/wallets)	India-first PSP, UPI-native, webhooks

4 Data model

The catalogue is deliberately separated from the volatile per-source offer snapshots. A **canonical product** is stable; an **offer** (price/stock/ETA from one source) is short-lived and refreshed constantly.

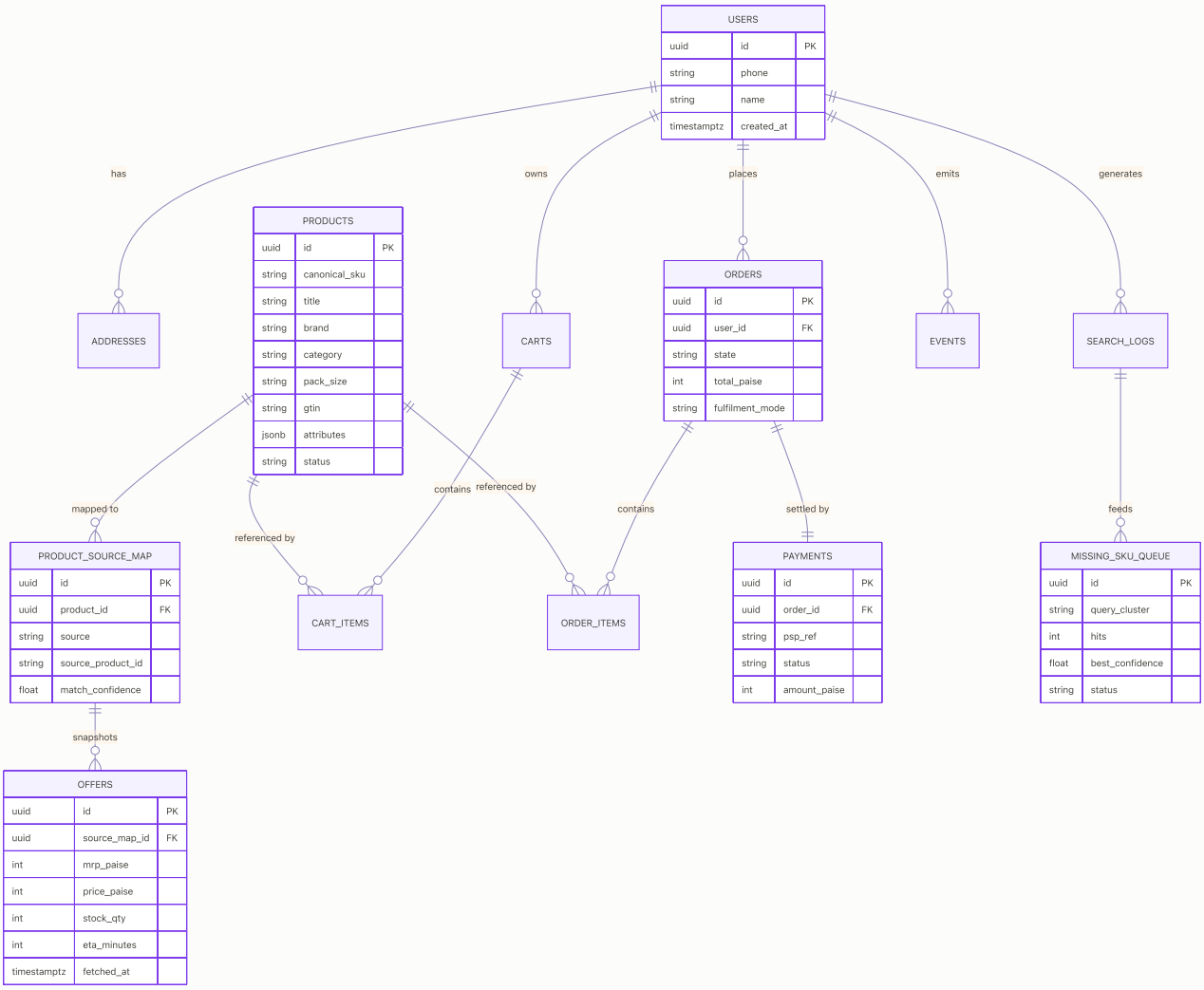


Figure 3.3 — Core relational model. Offers hang off the source-map, not the product, so a product can carry five live offers at once.

POSTGRESQL

Users, catalogue, carts, orders, payments — anything that must be consistent and durable.

REDIS

The *hot* offer cache (price/stock/ETA) with 30–60s TTLs, rate-limit counters, and pub/sub for tracking.

OPENSEARCH

The searchable projection of the catalogue — text queries resolve here to canonical product IDs.

5 Catalogue management

The catalogue is the backbone: every search result, comparison and order references a **canonical product**. Platform-specific listings are attached to it, never the other way round.

The canonical product model

FIELD	MEANING
`canonical_sku`	Stable internal identifier, e.g. `QC-GRO-AMUL-BUTTER-500G`
`title, brand, category`	Normalised display + taxonomy
`pack_size / unit`	Normalised pack (e.g. `500 g`) so "butter 500g" matches across sources
`gtin / ean`	Barcode — the strongest cross-source match key
`attributes` (JSONB)	Flexible per-category attributes (flavour, wattage, veg/non-veg…)
`images[]`	De-duplicated, re-hosted on our CDN
`status`	active · candidate · retired

Mapping platform products → canonical

Each source listing is linked through `PRODUCT_SOURCE_MAP` with a **match confidence**. Matching runs a cascade: exact **GTIN/EAN** → fuzzy **brand + normalised title + pack-size** → **embedding similarity** on title+image. High-confidence matches auto-link; the rest go to review.

canonical-product.example.json

```
{
  "canonical_sku": "QC-GRO-AMUL-BUTTER-500G",
  "title": "Amul Butter (Pasteurised, Salted)",
  "brand": "Amul",
  "category": "Dairy > Butter & Cheese",
  "pack_size": "500 g",
  "gtin": "8901020000123",
  "attributes": { "salted": true, "veg": true },
  "status": "active",
  "source_map": [
    { "source": "blinkit",   "source_product_id": "prod_88213", "confidence":
0.99 },
    { "source": "zepto",     "source_product_id": "z_44120",    "confidence":
0.97 },
```

```
    { "source": "bigbasket", "source_product_id": "BB_10029",   "confidence":
0.98 },
    { "source": "amazon",    "source_product_id": "B07XYZ",     "confidence":
0.91 }
  ]
}
```

Data governance

Prices and stock are never authored by us — they are *snapshots* attributed to a source and timestamped. Catalogue edits are versioned and moderated; auto-created products enter as `candidate` and are only promoted to `active` after validation, so a bad auto-match can't silently corrupt the catalogue.

6 Missing-SKU discovery

QuickCart grows its catalogue from real demand. When a shopper searches for something we can't map, that gap is a *signal* — the same loop shown in Document 02.

Where gaps come from

User searches
Queries with zero or low-confidence matches are logged to the missing-SKU queue.

Source browse
Scheduled category-listing walks via each platform MCP surface catalogue items we don't yet have.

Trending queries
External trend feeds flag rising products to pre-empt demand.

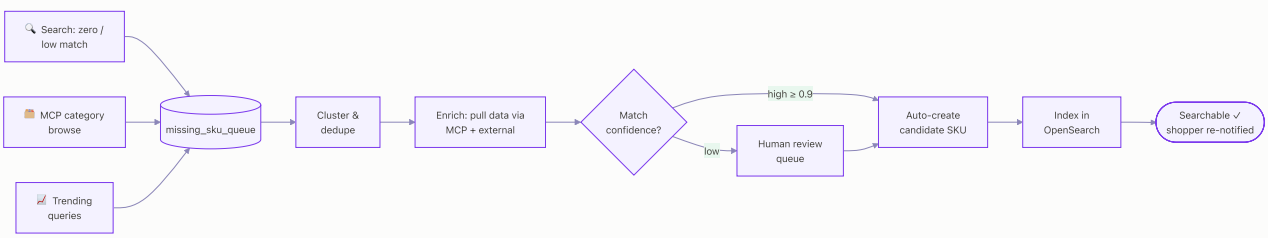


Figure 3.4 — The missing-SKU pipeline: capture → cluster → enrich → match → (review) → publish → searchable.

Pipeline health metrics

METRIC	TARGET	WHAT IT TELLS US
Queue size	< 2,000 open	Backlog of un-catalogued demand
Auto-match rate	≥ 70%	Share resolved without a human
Time-to-catalogue (p50)	< 24 h	Speed from first miss to searchable
Review precision	≥ 98%	Correctness of published entries

7 Analytics

Every meaningful action emits a typed event from the client SDK or a service. Events flow through Kafka into both a real-time layer (dashboards, missing-SKU signal) and the warehouse (funnels, cohorts, KPIs).

Event taxonomy

EVENT	FIRES WHEN	KEY PROPERTIES
`search_performed`	Query submitted	query, results_count, zone
`offers_viewed`	Compare screen opened	product_id, source_count
`source_selected`	An offer is picked	product_id, source, rank
`add_to_cart`	Item added	product_id, price_paise, source
`checkout_started`	Checkout opened	cart_value, item_count
`order_placed`	Order confirmed	order_id, total, fulfilment_mode
`order_delivered`	Delivery complete	order_id, actual_eta_min
`missing_sku_logged`	No/low match	query, best_confidence

Headline KPIs

₹4.8	9,200	14.2%	₹560
Cr/mo GMV	ORDERS / DAY	SEARCH→ORDER CONVERSION	AVG BASKET VALUE

Conversion funnel

Where shoppers drop off between searching and ordering. Bars are one blue ordinal ramp; magnitude is the bar length.

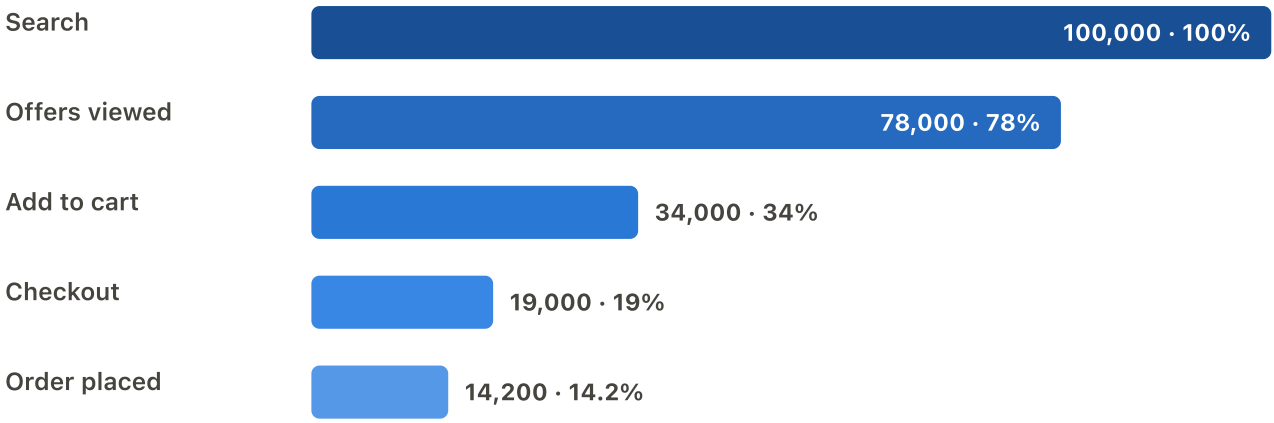


Figure 3.5 — Search-to-order funnel (illustrative weekly volumes). Biggest drop is offers-viewed → add-to-cart.

Source win-rate

How often each platform provides the *ranked-best* offer at the moment of purchase.

Blinkit Zepto BigBasket Flipkart Amazon

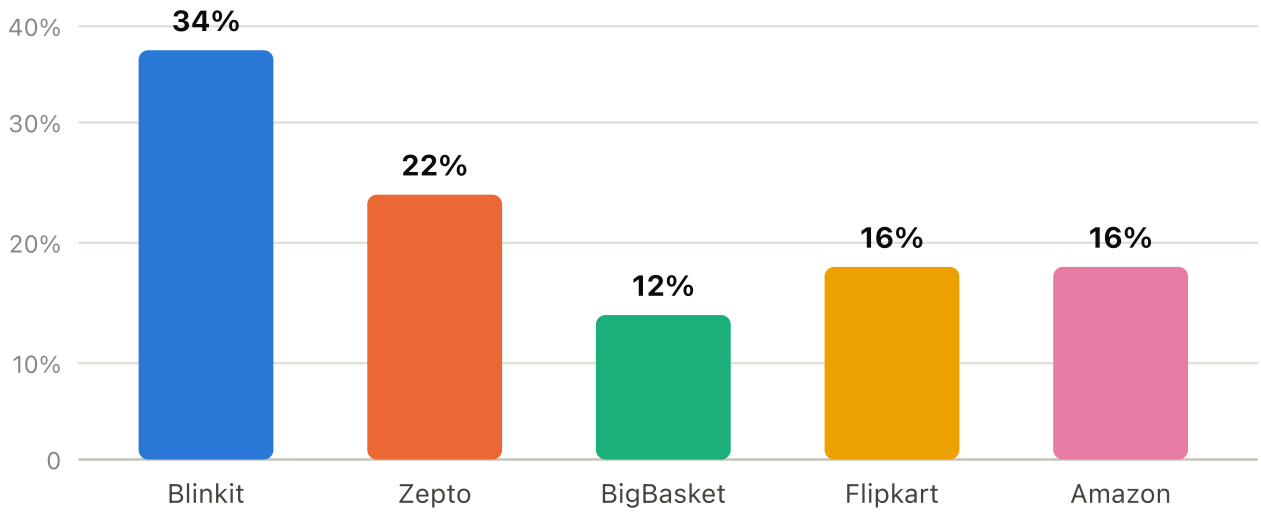


Figure 3.6 — Share of "best offer" by platform (illustrative). Quick-commerce players win on ETA; marketplaces win on price for long-tail goods.

GMV trend

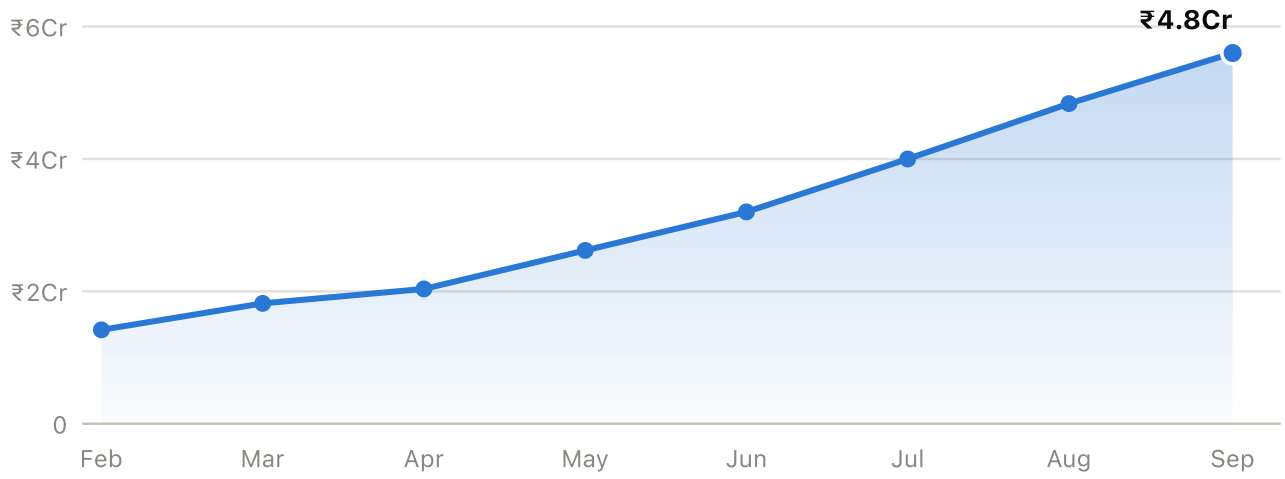


Figure 3.7 — Monthly GMV (illustrative). Single-series magnitude, one blue hue.

All figures on this page are illustrative design targets, not measured production data.

8 Payments

QuickCart supports two fulfilment-and-money models, chosen per source based on what that platform's MCP allows (see Document 04):

Managed checkout

QuickCart collects payment, places the order with the source on the shopper's behalf, and settles to the source (net of commission). One cart can even span multiple sources. Requires a source MCP that exposes `create_cart` / `checkout`.

Assisted hand-off

Where a source only exposes read tools, QuickCart deep-links into that platform with the basket pre-filled; the source collects payment. QuickCart earns via affiliate attribution. No card data touches us.

Payment methods

UPI | Cards (tokenised) | Netbanking | Wallets | Cash on delivery

Managed-checkout payment flow (UPI)

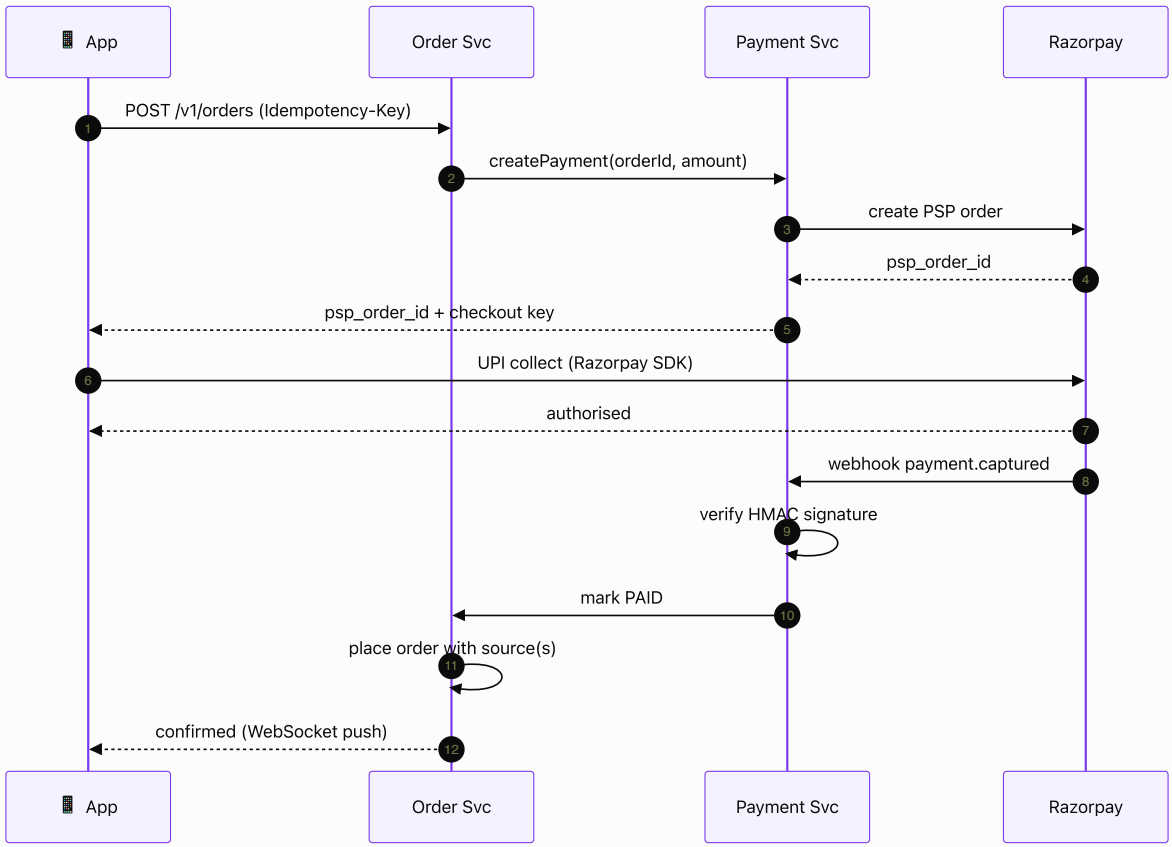


Figure 3.8 — Payment is confirmed by the server-verified webhook, never by the client's word alone.

Refunds, reconciliation & scope

- **Refunds/cancellations:** if a source rejects or an item is out of stock at pick time, the order is auto-cancelled and refunded through the PSP; partial refunds for partial fulfilment.

- **Reconciliation:** a daily job matches PSP settlements against orders and source invoices; mismatches raise a finance alert.
- **Idempotency:** both order placement and webhook handling are idempotent (keyed by `Idempotency-Key` / PSP event id) so retries never double-charge or double-fulfil.

> **Compliance — keep PCI scope minimal**
> QuickCart never stores raw card numbers. Cards are tokenised by the PSP (SAQ-A scope); webhooks are HMAC-verified; all money movement is logged immutably. UPI and tokenised cards keep the app out of full PCI-DSS scope. Personal data handling follows India's DPDP Act (consent, purpose limitation, deletion on request).

9 Scaling, reliability & security

SCALE

Stateless services autoscale on Kubernetes; Redis absorbs read fan-out; OpenSearch and Postgres use read replicas. Aggregation is I/O-bound and parallel.

RELIABILITY

Circuit breakers + timeouts on every MCP call; partial results are first-class (a down source is simply omitted). Graceful degradation, health checks, blue/green deploys.

SECURITY

JWT + refresh rotation, rate-limiting, WAF, cert pinning, secrets in a vault, least-privilege service roles, audit logs.

OBSERVABILITY

Structured logs, RED/USE metrics, distributed tracing across the gateway→service→MCP path, SLO alerting.

DATA PROTECTION

Encryption in transit + at rest, PII minimisation, DPDP-aligned consent & deletion, tokenised payments.

COST CONTROL

Aggressive caching (fewer MCP calls), request coalescing, and TTLs tuned per data volatility (price 30s, catalogue hours).

Read next

Document 04 — MCP Integrations details exactly which tools each platform exposes and how far we can use them; Document 05 — Price & Availability Engine covers the scatter-gather, normalisation and ranking algorithm behind every offer shown here.

DOCUMENT 04 · MCP INTEGRATIONS

The MCP adaptor layer

Five platform adaptors, one unified tool contract. This document catalogues every MCP tool QuickCart calls to fetch products, MRP, live stock, delivery ETA and offers — and states plainly how far each platform actually lets us go.

Blinkit · Zepto · BigBasket · Flipkart · Amazon

5	1	11	Read-first
PLATFORM ADAPTORS	UNIFIED TOOL SCHEMA	CANONICAL TOOLS	ACCESS POSTURE

1 What is MCP, and how QuickCart uses it

MCP (Model Context Protocol) is an open standard for exposing *tools* and *resources* to an agent runtime over JSON-RPC 2.0. A server advertises a list of callable tools (each with a JSON-Schema for its inputs), and a client invokes them by name over a transport — either `stdio` for a co-located process or streamable `HTTP+SSE` for a networked one. The protocol is transport-agnostic and schema-first, which is exactly what we want when talking to five very different retail backends.

QuickCart runs a single gateway service — `mcp-adaptor` — that fronts **five per-platform MCP servers**, one each for Blinkit, Zepto, BigBasket, Flipkart and Amazon. Every adaptor implements the *same* normalised tool contract (§4), so the Aggregation service can fan a query out to all five with identical call sites and merge the results without special-casing any platform. Each adaptor is a thin translator: it maps our canonical tool call onto whatever surface the platform genuinely offers, then maps the platform's native response back into QuickCart's canonical schema.

Put differently: MCP is our *internal integration boundary*. It is not something these retailers publish — it is how *we* choose to wrap them so a new source is "just another adaptor speaking the same eleven tools."

> **Reality check — these are QuickCart-built adaptors, not vendor MCPs**
> As of 2026, **none of Blinkit, Zepto, BigBasket, Flipkart or Amazon publishes an official public Model Context Protocol server**. Everything in this document is a QuickCart-built MCP adaptor that wraps each platform's *actual* available surface — an official affiliate/seller API where one exists, or a partner commerce API granted under a business agreement. Where a platform offers no sanctioned data access, that capability is marked out of scope. We do not treat unauthorised scraping as a production data source (see §7).

2 Adaptor topology

The Aggregation service never speaks to a retailer directly. It speaks MCP to the `mcp-adaptor` gateway, which routes each canonical tool call to the right per-platform server. Each server owns the credentials, rate-limit budget, native request shape and response-normalisation for exactly one platform.

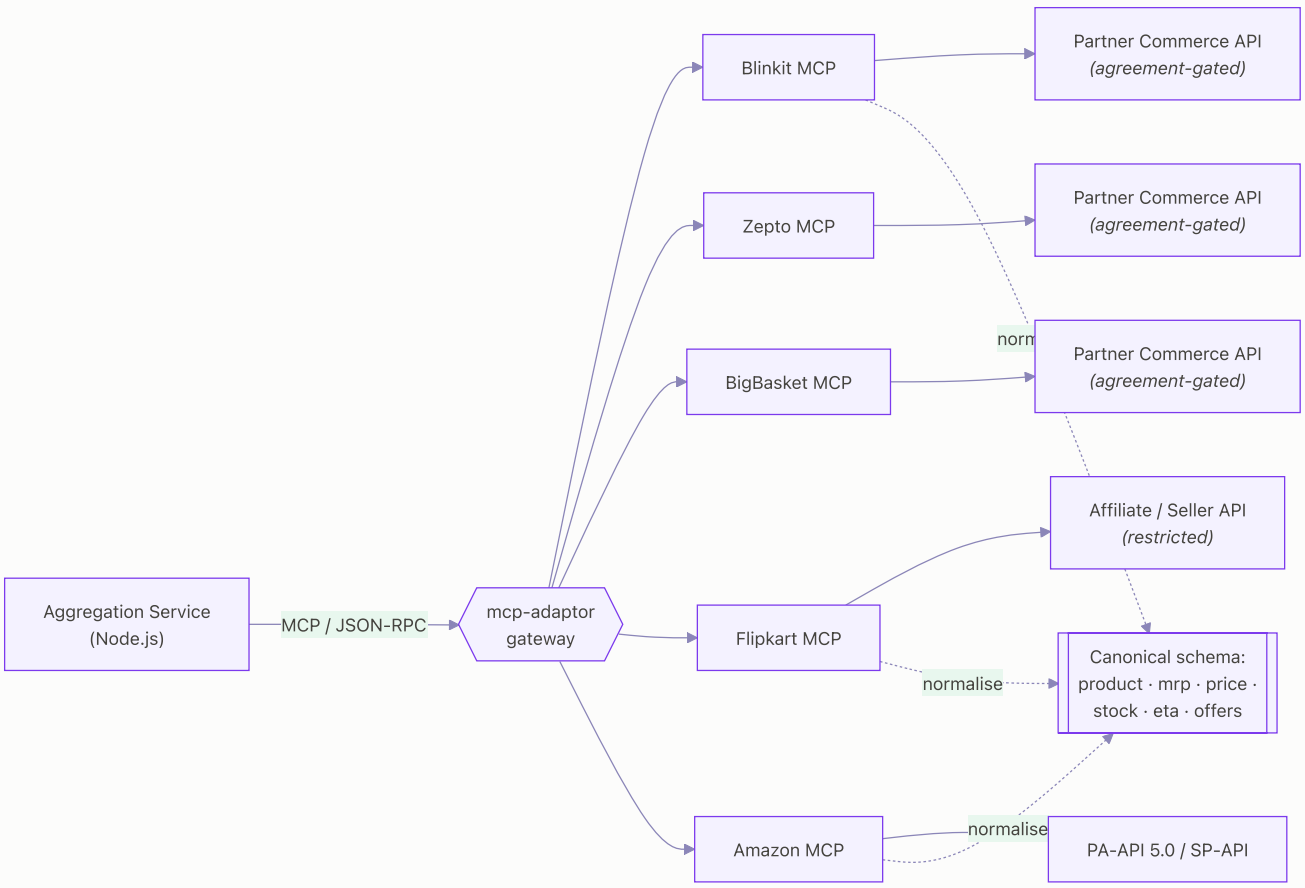


Figure 4.1 — One gateway, five adaptors. Each adaptor translates a native backend into the same canonical schema so the Aggregation service stays platform-agnostic.

3 The unified tool contract

Every adaptor implements the same eleven tools. Read tools are mandatory; write tools are optional and only wired up where a platform's commerce API genuinely permits programmatic cart/checkout. All price fields are in ₹ (INR); all availability and ETA calls are *location-scoped* because quick-commerce stock is hyperlocal.

TOOL	PARAMETERS	RETURNS (NORMALISED)	NOTES
`search_products`	query, location, limit	`[{sourceProductId, title, brand, packSize, image, price, inStock, etaMinutes}]`	Read. Ranked snippets; drives the search fan-out.
`get_product`	sourceProductId	Full product: title, brand, attributes, images, description	Read. Detail-screen hydrate.
`get_price`	sourceProductId, location	`{mrp, sellingPrice, effectivePrice, currency}`	Read. `effectivePrice` nets applicable auto-offers.
`check_stock`	sourceProductId, location	`{inStock, quantityAvailable, maxQty}`	Read. Location-scoped; shortest TTL (~1 min).
`get_delivery_estimate`	sourceProductId, location	`{minutes, slot, deliveryFee}`	Read. Real-time; drives the ETA badge.
`get_offers`	sourceProductId, location	`[{code, type, value, minCart, expiresAt}]`	Read. Coupons & auto-applied discounts.
`list_categories`	—	Category tree	Read. Seeds catalogue & browse.
`list_products_by_category`	categoryId, cursor	Paged product list	Read. Powers the missing-SKU crawl (Doc 03 §5).
`create_cart`	location	`{cartId}`	Write — only where a partner commerce API allows it.
`add_to_cart`	cartId, sourceProductId, qty	`{cartId, lineItems, subtotal}`	Write — partner-gated; otherwise deep-link handoff.
`checkout`	cartId, address, paymentToken	`{orderId, status, trackingUrl}`	Write — rarely available; most platforms are handoff-only.

`get_offers — input JSON-Schema`

```
{
  "name": "get_offers",
  "description": "Coupons and auto-applied discounts for a product at a location",
  "inputSchema": {
    "type": "object",
    "required": ["sourceProductId", "location"],
    "properties": {
      "sourceProductId": { "type": "string" },
      "location": {
        "type": "object",
        "required": ["lat", "lng"],
        "properties": {
          "lat": { "type": "number" },
          "lng": { "type": "number" },
          "pincode": { "type": "string" }
        }
      }
    }
  }
}
```

`get_offers — normalised response`

```
{
  "sourceId": "blinkit",
  "sourceProductId": "blk_88213",
  "offers": [
    { "code": "WELCOME50", "type": "FLAT", "value": 50, "minCart": 199, "expiresAt": "2026-09-30T18:30:00Z" },
    { "code": null, "type": "AUTO_PERCENT", "value": 10, "appliesTo": "item" }
  ],
  "fetchedAt": "2026-09-08T10:14:22Z",
  "ttlSeconds": 120
}
```

4 Per-platform adaptors

Same eleven tools, five very different realities. The strength of each platform maps directly to what it is: quick-commerce players are unbeatable on hyperlocal stock and ETA; the marketplaces are richer on catalogue depth, MRP and reviews. Access mechanisms and limits below are illustrative of each platform's real posture, not published MCP specs.

Blinkit Blinkit adaptor

Underlying access: partner commerce API, agreement-gated. No public product API — access requires a signed data/commerce partnership.

Supports: `search_products`, `get_product`, `get_price`, `check_stock`, `get_delivery_estimate`, `get_offers`, `list_categories`, `list_products_by_category`. Hyperlocal stock & ETA are its standout signals.

Write / checkout: typically `handoff` — deep-link into the Blinkit app/PDP; programmatic `checkout` only under a commerce agreement.

Auth: OAuth2 client-credentials + per-partner API key · **Rate:** quota per agreement · **Freshness:** stock ~1 min, ETA real-time.

Zepto Zepto adaptor

Underlying access: partner commerce API, agreement-gated. Same posture as Blinkit — a business relationship is the gate, not a public developer portal.

Supports: the full read set; excellent sub-15-minute ETA and dark-store-level stock. `get_offers` reflects Zepto Pass / cart-level promos where surfaced by the API.

Write / checkout: `handoff` by default; cart APIs only if the partnership includes commerce endpoints.

Auth: OAuth2 + API key · **Rate:** per-agreement quota · **Freshness:** stock ~1 min, ETA real-time.

BigBasket BigBasket adaptor

Underlying access: partner/commerce API, agreement-gated. Slotted + express (BB Now) fulfilment, so ETA can be a delivery *slot* as well as minutes.

Supports: full read set; deepest grocery catalogue and reliable MRP/pack-size data; `get_delivery_estimate` returns either express minutes or a slot window.

Write / checkout: `handoff` by default; slot booking via API only under agreement.

Auth: OAuth2 + API key · **Rate:** per-agreement quota · **Freshness:** price ~5 min, stock ~2 min, slots cached to slot expiry.

Flipkart Flipkart adaptor

Underlying access: Flipkart *Affiliate API* (historically public, now **restricted/approval-only**) and the Seller/Marketplace API for merchants. Product read is possible for approved affiliates; deep catalogue but no hyperlocal minute-ETA.

Supports: `search_products`, `get_product`, `get_price` (MRP + selling price), `get_offers`, ratings/reviews, images. `get_delivery_estimate` returns standard delivery days for a pincode, not minutes.

Write / checkout: `affiliate deep-link` buy — no programmatic consumer checkout.

Auth: Affiliate ID + token · **Rate:** affiliate quotas · **Freshness:** price ~15 min; must honour affiliate attribution.

Amazon Amazon adaptor

Underlying access: Product Advertising API (PA-API 5.0) for Associates and, for our own listings, **SP-API**. The richest read surface of the five — full product detail, MRP (`SavingBasis`), Buy Box price, images and reviews summary.

Supports: `search_products` (SearchItems), `get_product` (GetItems), `get_price`, `get_offers`, images, ratings. `get_delivery_estimate` is delivery days / Prime eligibility, not minutes.

Write / checkout: `affiliate deep-link` (Add-to-Cart / associate links) — no programmatic consumer checkout via PA-API.

Auth: PA-API access/secret keys + Associate tag · **Rate:** TPS tied to shipped-items revenue (throttles hard when low) · **Freshness:** price ~15 min; strict display & attribution terms.

Reading the cards

Two families, two shapes of value:

- **Quick-commerce** (Blinkit, Zepto, BigBasket) → hyperlocal *stock + minute-level ETA*; access is *partnership-gated*; checkout is usually handoff.
- **Marketplaces** (Flipkart, Amazon) → deep catalogue, reliable *MRP + reviews*; access via *affiliate/seller* programs; buy is an attributed *deep-link*.

The Price Engine (Doc 05) weights each source's signals by exactly these strengths.

5 Capability matrix

What each adaptor can actually return today, given the access mechanisms above. This is the single most important table in the document — the Aggregation service degrades gracefully around every ◑ and ✕.

PLATFORM	SEARCH	DETAILS	MRP	SELL PRICE	LIVE STOCK	ETA	OFFERS	IMAGES	REVIEWS	BROWSE	ADD CART	CHECKOUT	ORD
Blinkit	✓	✓	✓	✓	✓	✓	✓	✓	✕	✓	◑	◑	◑
Zepto	✓	✓	✓	✓	✓	✓	✓	✓	✕	✓	◑	◑	◑
BigBasket	✓	✓	✓	✓	✓	◑	✓	✓	◑	✓	◑	◑	◑
Flipkart	✓	✓	✓	✓	◑	◑	✓	✓	✓	✓	✕	✕	✕
Amazon	✓	✓	✓	✓	◑	◑	✓	✓	✓	✓	◑	✕	✕

✓ Fully supported ◑ Partial / conditional (hover for the caveat) ✕ Not available via sanctioned access

Quick-commerce write/checkout cells are ◑ because they depend on a commerce partnership; marketplace checkout is ✕ because affiliate/PA-API programs deliberately exclude programmatic consumer purchase.

6 To what extent can we use them?

The honest answer differs per capability, not just per platform. Four axes govern everything:

📖 **Read vs write**
All five give sanctioned *read* access to product, price and (for quick-commerce) stock/ETA. *Write* — cart and checkout — is the exception, available only under a commerce partnership; the default consumer path is a deep-link handoff.

📍 **Per-location queries**
Stock, ETA and often price are **location-scoped**. Quick-commerce answers are per dark-store; a query without lat/lng/pincode is meaningless. We cache per `(sourceProductId, geohash)`.

⏱ **Rate limits & quotas**
PA-API TPS scales with our shipped-items revenue and throttles aggressively when low; affiliate and partner APIs carry per-agreement quotas. The gateway enforces a token bucket per adaptor and sheds load to cache.

🕒 **Data freshness**
We cache with capability-specific TTLs: **price ~2–5 min**, **stock ~1 min**, **ETA real-time (no cache)**, catalogue/images hours. Every response carries `fetchedAt` so the UI can show staleness.

WHAT WE CAN RELY ON

- Product search & detail across all five.
- MRP + selling price on all five (marketplaces are most reliable).
- Live hyperlocal stock & minute-level ETA from Blinkit & Zepto (BigBasket express).
- Offers/coupons where the API surfaces them.
- Attributed deep-link buy to every platform (universal fallback).

WHAT NEEDS A PARTNERSHIP / IS OUT OF SCOPE

- Programmatic cart & checkout on quick-commerce — requires a commerce agreement.
- Any consumer checkout on Flipkart/Amazon via affiliate/PA-API — excluded by program terms.
- Live per-unit quantity on marketplaces (only an availability message).
- Order status for orders we didn't place.
- **Scraping** of any platform — out of scope on ToS/legal grounds.

Legal & ToS boundary
Production access to each platform is contingent on the relevant **affiliate, seller or commerce partnership** and its display/attribution terms (e.g. Amazon Associates & PA-API display rules, affiliate link attribution, cached-price display windows). **Unauthorised scraping is explicitly out of scope** — it violates platform terms, is legally risky, and is not a data source QuickCart ships on. Every adaptor must degrade to a deep-link handoff when a sanctioned data path is unavailable.

7 Field mapping

Each adaptor's job ends here: collapse a native payload into QuickCart's canonical fields. Native field names below are **illustrative** — exact for Amazon PA-API, representative for the partner APIs whose schemas aren't public.

CANONICAL FIELD	AMAZON (PA-API)	FLIPKART (AFFILIATE)	BLINKIT / ZEPTO / BIGBASKET (ILLUSTRATIVE)
`sourceProductId`	`ASIN`	`productId`	`sku_id` / `product_id`
`title`	`ItemInfo.Title.DisplayValue`	`productBaseInfo.title`	`name`
`mrp`	`Offers.Listings.SavingBasis.Amount`	`maximumRetailPrice.amount`	`mrp`
`sellingPrice`	`Offers.Listings.Price.Amount`	`flipkartSpecialPrice.amount`	`selling_price`
`effectivePrice`	Price – applied `Promotions`	Price – `discountPercentage`	`final_price` (post auto-offer)
`inStock`	`Offers.Listings.Availability.Type`	`inStock`	`availability.in_stock`
`quantityAvailable`	— (message only)	— (flag only)	`availability.qty`
`deliveryEstimate.minutes`	— (delivery days)	— (delivery days)	`eta_minutes`
`deliveryEstimate.deliveryFee`	Prime / shipping	`shippingCharges`	`delivery_charge`
`image`	`Images.Primary.Large.URL`	`imageUrls['200x200']`	`image_url`
`rating`	`CustomerReviews.StarRating`	`productBaseInfo.rating.average`	— (usually absent)

8 Sample MCP call

A single `search_products` invocation against the Blinkit adaptor over JSON-RPC, and the normalised list it returns. Note the adaptor has already collapsed Blinkit's native payload into our canonical fields — the caller never sees a Blinkit-shaped object.

JSON-RPC request → blinkit MCP

```
{
  "jsonrpc": "2.0",
  "id": 42,
  "method": "tools/call",
  "params": {
    "name": "search_products",
    "arguments": {
      "query": "amul butter 500g",
      "location": { "lat": 12.9716, "lng": 77.5946, "pincode": "560001" },
      "limit": 5
    }
  }
}
```

JSON-RPC result ← blinkit MCP (normalised)

```
{
  "jsonrpc": "2.0",
  "id": 42,
  "result": {
    "content": [{ "type": "json", "json": {
      "sourceId": "blinkit",
      "products": [
        {
          "sourceProductId": "blk_88213", "title": "Amul Butter (Pasteurised)",
          "brand": "Amul", "packSize": "500 g",
          "mrp": 290, "sellingPrice": 278, "effectivePrice": 265,
          "inStock": true, "quantityAvailable": 14,
          "deliveryEstimate": { "minutes": 9, "deliveryFee": 0 },
          "image": "https://cdn.example/blk_88213.jpg"
        }
      ],
      "fetchedAt": "2026-09-08T10:14:22Z"
    } }]
  }
}
```

On the Aggregation side, the same call is one line via the MCP client — the gateway hides transport and auth:

aggregation/sources.ts

```
import { mcp } from "./mcpAdaptorClient";

// Fan a single canonical tool out to one source; the adaptor normalises the shape.
export async function searchSource(sourceId, query, location) {
  const res = await mcp.call(sourceId, "search_products", {
    query, location, limit: 5
  }, { timeoutMs: 800 });       // per-source deadline; see Doc 05
  return res.products;                    // already in canonical schema
}
```

How these five per-source results are merged, SKU-matched and ranked into a single best-price answer is the subject of Document 05 — Price & Availability Engine.

Where this fits

The `mcp-adaptor` gateway sits inside the backend described in Document 03 — Architecture, called by the Aggregation service; the `list_products_by_category` crawl also feeds the missing-SKU pipeline there. Merge & ranking of everything these tools return is covered in Document 05.

DOCUMENT 05 · PRICE & AVAILABILITY ENGINE

Best price, live stock & fastest ETA

How the backend fans out to five platform MCPs, normalises very different responses into one shape, matches them to a canonical SKU, and ranks them into the single "best buy" a shopper sees — with the algorithm, the caching strategy and the Node.js code.

5→1	1.2s	30–60s	4
SOURCES RANKED TO ONE PICK	FAN-OUT DEADLINE	OFFER CACHE TTL	RANKING SIGNALS

This is the heart of QuickCart: turning five inconsistent, hyperlocal, constantly-changing price lists into one trustworthy answer — *fast enough to feel instant* and *fresh enough to trust at checkout*. It builds on the MCP adaptor layer (Doc 04) and the catalogue in Doc 03.

1 The problem in one line

A shopper types `butter`. Five platforms answer with five different product IDs, price shapes, pack sizes, stock signals and delivery promises — some in rupees, some in paise, some quoting MRP, some the offer price, some silent on stock. The engine must reconcile all of it into a ranked list **in under 1.5 seconds**, and re-verify the winner before money moves.

Gather

Scatter-gather the query to all five MCPs in parallel, under a hard deadline. Slow sources are dropped, not waited on.

Normalise & match

Coerce every response into one `Offer` shape and attach it to a canonical SKU via the source-map.

Rank

Score each offer on price, ETA, stock and source reliability; return the sorted list and the winner.

2 Aggregation pipeline

Every search or product-detail view runs the same pipeline. It is cache-first: a warm zone+SKU key returns in single-digit milliseconds; a miss triggers the fan-out below.

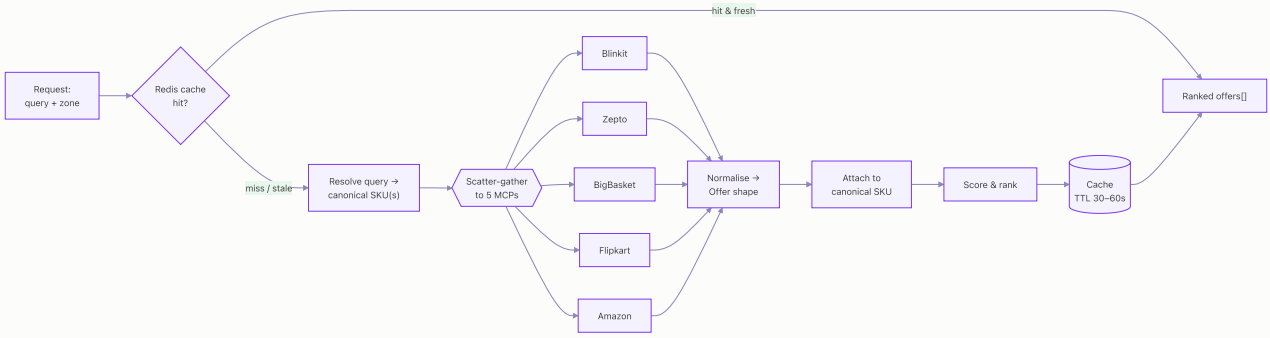


Figure 5.1 — Cache-first aggregation. A miss fans out to five MCPs, each result is normalised, matched and ranked, then cached with a short TTL.

Why cache-first with a short TTL

Quick-commerce price and stock move fast, so TTLs are short (30s for price/stock, up to 60s for ETA). But identical searches cluster hard (everyone wants *milk* at 8am), so even a 30s cache slashes MCP call volume and cost while keeping data effectively live. Stale-while-revalidate serves the cached value instantly and refreshes in the background.

3 Normalising five different responses

Each adaptor already maps its platform onto the canonical tool contract (Doc 04), but the engine still normalises the *values*: money to integer paise, MRP vs effective price after offers, pack size to a comparable unit, and stock to a single signal. This makes offers directly comparable.

The canonical `Offer`

types/offer.ts

```
export interface Offer {
  source:        SourceId;        // "blinkit" | "zepto" | "bigbasket" |
"flipkart" | "amazon"
  productId:     string;          // canonical QuickCart SKU
  sourceProductId: string;
  mrpPaise:      number;          // list / maximum retail price
  pricePaise:    number;          // effective price after platform offers
  unitPricePaise:number;          // per-100g / per-unit, for like-for-like
compare
  inStock:       boolean;
  stockQty:      number | null;   // null = unknown but purchasable
  etaMinutes:    number | null;   // delivery estimate for this zone
  deliveryFeePaise: number;
  rating:        number | null;
  fulfilment:    "managed" | "handoff";  // see Doc 03 §Payments
  fetchedAt:     string;          // ISO timestamp — drives freshness
  deepLink:      string;          // for assisted hand-off checkout
}
```

Field mapping across sources

CANONICAL FIELD	BLINKIT / ZEPTO / BIGBASKET	FLIPKART	AMAZON
`pricePaise`	offer price × 100	`final_price`	`Offers.Listing.Price.Amount`
`mrpPaise`	MRP field × 100	`mrp`	`SavingBasis` / list price
`inStock` / `stockQty`	hyperlocal stock for the store serving the zone	seller availability	`Availability.Type`
`etaMinutes`	10–20 min (dark-store ETA)	derived from delivery-date promise	same-day / next-day → minutes
`deliveryFeePaise`	zone fee, free over threshold	shipping fee	Prime = 0, else shipping

> **MRP is a claim, not a constant**
> Different sellers list slightly different MRPs for the same GTIN. QuickCart shows the *source's own* MRP alongside its effective price so "you save" is always internally consistent, and flags implausible MRPs (e.g. price > MRP) rather than displaying a negative discount.

4 Scatter-gather with a deadline

The fan-out uses `Promise.allSettled` wrapped in a per-source timeout so one slow platform can never hold up the response. Failed or timed-out sources are simply omitted and the result is labelled "showing N of 5 sources." Circuit breakers trip a repeatedly-failing source out of rotation.

services/aggregation/gather.ts

```
import { mcp } from "./mcpClient";
import { normalise } from "./normalise";
import { SOURCES, DEADLINE_MS } from "./config";

// Wrap any promise so it rejects after `ms` instead of hanging the fan-out.
function withTimeout<T>(p: Promise<T>, ms: number): Promise<T> {
  return Promise.race([
    p,
    new Promise<T>((_, reject) =>
      setTimeout(() => reject(new Error("deadline_exceeded")), ms)),
  ]);
```

```
}

// Fan out one query to every source; return only the offers that came back in
time.
export async function gatherOffers(sku: string, zone: Zone):
Promise<GatherResult> {
  const settled = await Promise.allSettled(
    SOURCES.map(async (source) => {
      if (breaker.isOpen(source)) throw new Error("circuit_open");
      const raw = await withTimeout(
        mcp.call(source, "get_offer", { sourceSku: map(sku, source), zone }),
        DEADLINE_MS,                        // 1200 ms
      );
      return normalise(source, raw);   // → Offer
    }),
  );

  const offers  = settled.filter(isFulfilled).map((s) => s.value);
  const misses  = settled.filter(isRejected);
  misses.forEach((m, i) => breaker.record(SOURCES[i], m.reason));

  return { offers, sourcesQueried: SOURCES.length, sourcesReturned:
offers.length };
}
```

Partial results are first-class

Returning 4 of 5 offers in 1.2s beats returning 5 in 6s. The UI shows a "4 of 5 sources" badge and the missing source refreshes asynchronously into the cache, so the next shopper sees the full set.

5 The ranking algorithm

"Best" is not just "cheapest." QuickCart scores each in-stock offer on four normalised signals and sorts by the weighted total. Weights are tunable per surface — a "Fastest" toggle raises the ETA weight; a "Cheapest" toggle raises the price weight (these are the filter chips in Doc 01).

Signals & default weights

SIGNAL	NORMALISATION (0–1, HIGHER = BETTER)	DEFAULT WEIGHT
Effective price (incl. delivery fee)	`minLandedPrice / thisLandedPrice`	**0.45**
Delivery ETA	`1 − clamp(eta / 120min)`	**0.30**
Stock confidence	in-stock & ample = 1 · low = 0.6 · unknown = 0.4	**0.15**
Source reliability	rolling success + rating, per source & zone	**0.10**

services/aggregation/rank.ts

```ts
const W = { price: 0.45, eta: 0.30, stock: 0.15, reliability: 0.10 };

const landed  = (o: Offer) => o.pricePaise + o.deliveryFeePaise;
const stockScore = (o: Offer) =>
  !o.inStock ? 0 : o.stockQty == null ? 0.4 : o.stockQty > 5 ? 1 : 0.6;

export function rankOffers(offers: Offer[], w = W): Ranked[] {
  const live = offers.filter((o) => o.inStock);
  if (!live.length) return [];

  const minLanded = Math.min(...live.map(landed));

  const scored = live.map((o) => {
    const priceScore = minLanded / landed(o);                 // 1 = cheapest
    const etaScore   = 1 - Math.min((o.etaMinutes ?? 120) / 120, 1);
    const score =
      w.price * priceScore +
      w.eta   * etaScore +
      w.stock * stockScore(o) +
      w.reliability * reliability(o.source, o.zone);
    return { ...o, score, priceScore, etaScore };
  });

  return scored.sort((a, b) => b.score - a.score);        // winner = [0]
}
```

Tie-breaks & guard-rails

Equal scores break toward faster ETA, then higher reliability. An offer whose `fetchedAt` is older than its TTL is refreshed before it can win. The winner is re-validated at checkout (§7) so the shopper is never charged a stale price.

6 Worked example — "Amul Butter 500g"

Five offers come back for one canonical SKU. Landed price = price + delivery fee; the winner maximises the weighted score, not merely the lowest sticker price.

SOURCE	PRICE	DELIVERY	LANDED	ETA	STOCK	SCORE	
Blinkit	₹268	₹0	**₹268**	11 min	ample	**0.97**	● Best
Zepto	₹265	₹15	₹280	9 min	ample	0.94	
BigBasket	₹262	₹0	₹262	95 min	ample	0.88	
Amazon	₹259	₹40	₹299	1 day	ample	0.61	
Flipkart	—	—	—	—	out	—	excluded

Blinkit wins despite not being the cheapest sticker price: zero delivery fee makes its *landed* price lowest, and an 11-minute ETA scores far higher than BigBasket's 95 minutes. Values illustrative.

What the API returns

`GET /v1/products/QC-GRO-AMUL-BUTTER-500G/offers → 200`

```
{
  "productId": "QC-GRO-AMUL-BUTTER-500G",
  "sourcesQueried": 5,
  "sourcesReturned": 4,
  "best": "blinkit",
  "offers": [
    { "source":"blinkit", "pricePaise":26800, "mrpPaise":29500,
      "deliveryFeePaise":0, "etaMinutes":11, "inStock":true,
      "fulfilment":"managed", "score":0.97, "fetchedAt":"2026-09-08T12:00:05Z"
},
    { "source":"zepto", "pricePaise":26500, "deliveryFeePaise":1500,
      "etaMinutes":9, "inStock":true, "score":0.94 }
    /* …bigbasket, amazon… */
```

```
  ]
}
```

This is the exact array the Detail screen (Doc 01) renders as the "Compare 5 sources" block, and whose `best` drives the source label on every list card.

7 Freshness & checkout re-validation

Speed and freshness pull in opposite directions; QuickCart resolves the tension by tiering data by volatility and by **always re-checking the winner before payment**.

PRICE & STOCK

TTL 30s, stale-while-revalidate. Served from Redis; background refresh keeps the hot set current.

DELIVERY ETA

TTL 60s per zone — changes slower than price but is still hyperlocal.

CATALOGUE & IMAGES

TTL hours. Stable canonical data; invalidated on catalogue edits via Kafka.

Re-validate before charging

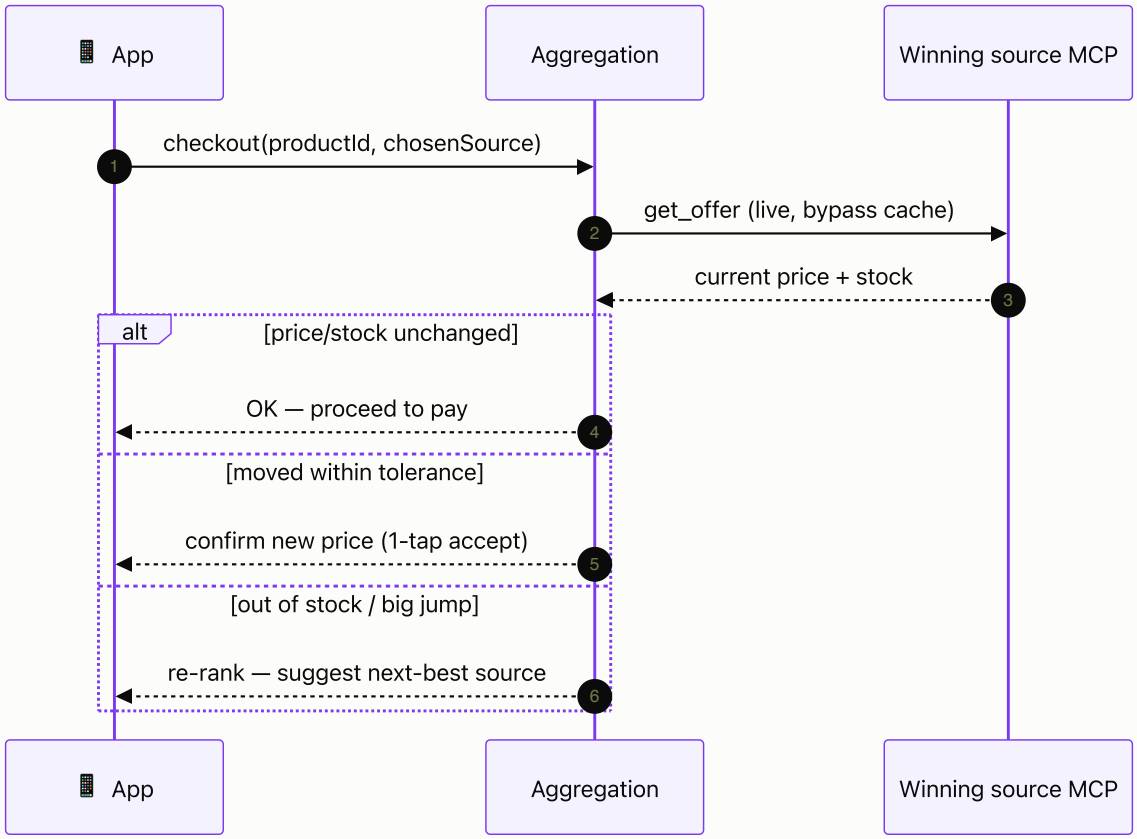


Figure 5.2 — The winning offer is re-fetched live at checkout; the shopper is never charged a cached price that has since moved.

8 Where the latency goes

The 1.5s p95 budget for a cold search, broken down. The fan-out dominates — which is exactly why it runs in parallel under a deadline rather than sequentially.

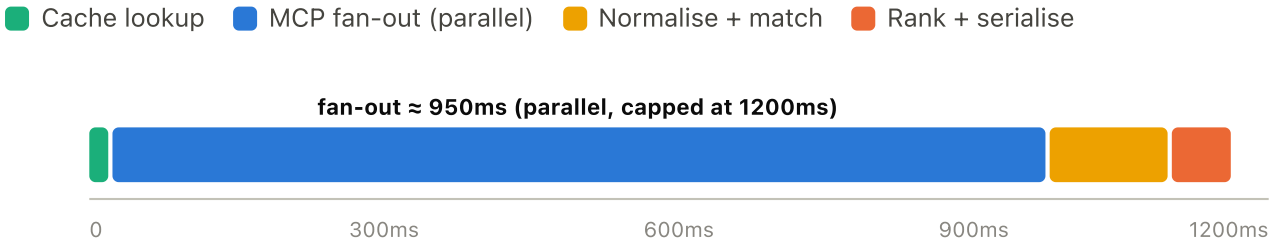


Figure 5.3 — Cold-search latency budget (illustrative). Warm cache hits skip everything but the ~20ms lookup.

9 How it lives in the backend

The engine is the `Aggregation` service (the `mcp-adaptor` of Doc 03). It is stateless and horizontally scalable — all shared state lives in Redis — so it scales with search traffic.

CONCERN	MECHANISM
Deduplicate concurrent identical searches	Request coalescing / single-flight per (sku, zone) key
Protect slow / failing sources	Per-source circuit breaker + exponential backoff
Respect platform rate limits	Token-bucket per source inside each MCP adaptor (Doc 04)
Keep the hot set warm	Background refresh of top-N zone+SKU keys via Kafka triggers
Observe correctness	Emit `offers_ranked` events → win-rate & price-accuracy dashboards (Doc 03 §Analytics)

Read alongside

Doc 04 defines the `get_offer` tool and per-platform limits this engine calls; Doc 03 shows where the Aggregation service sits; Doc 02 shows the same fan-out as a user-facing sequence.

QuickCart

Dossier · Document 06

DOCUMENT 06 · ORDERS & DELIVERY

One basket, many platforms, one order

How a shopper adds products sourced from Blinkit, Zepto, BigBasket, Flipkart and Amazon into a single basket — and QuickCart, as merchant of record, creates one internal order, procures each line from the best source, and delivers it as one experience.

Documents 04 and 05 explained how QuickCart *finds* the best price, stock and ETA for a single product across five platforms. This document explains what happens when a shopper wants **several** products at once — and those products live on different platforms. The answer is the piece that turns QuickCart from a comparison tool into a marketplace: a **cross-platform cart**, **one internal order**, and an **orchestrator** that procures and delivers everything under a single QuickCart receipt.

1 cart	1 payment	1 delivery	MoR
N SOURCE PLATFORMS	ONE QUICKCART CHARGE	TRACKED END-TO-END	MERCHANT OF RECORD

1 The model: QuickCart is the merchant of record

A price-comparison app stops at "Blinkit is cheapest — tap to open Blinkit." QuickCart goes further: **the shopper transacts once, with QuickCart.** Behind that single order, QuickCart procures each line item from whichever platform is cheapest and in stock, then delivers the whole basket. The shopper never creates five accounts, pays five times, or tracks five parcels.

Procurement of any given line happens in one of two modes — the same two modes introduced in the Architecture document (§9, Payments):

● Mode A

Managed checkout

Where a platform exposes a partner *commerce* API (programmatic cart + checkout), QuickCart places the sub-order automatically via the `mcp-adaptor` layer. Fast, hands-off, fully tracked. Realistically available on marketplace sources (Amazon/Flipkart via seller/affiliate flows) and on quick-commerce sources *only* where a partner agreement is in place.

● Mode B

Assisted fulfillment

Where a source is **handoff-only** (no programmatic checkout — see Doc 04), a QuickCart picker/agent buys the item in-store or in-app on the shopper's behalf, *or* the line is re-sourced to a different platform that *can* be placed programmatically and is still in stock at a comparable price.

Why "merchant of record" matters

Because the shopper's contract is with QuickCart, QuickCart owns the money flow (one authorisation, one capture, per-source settlement), the fulfillment promise (one combined ETA), and the support relationship (one place to raise a refund). Each platform becomes a *supplier*, not a checkout the user is bounced to.

2 Building a cross-platform basket

A QuickCart cart is a flat list of **lines**. Every line references a **canonical SKU** (the platform-independent product from the Catalogue, Doc 03 §6) plus a chosen `sourceId`. By default the source is the ranked winner from the Price Engine (Doc 05); the shopper can override it ("I'd rather get the butter from Zepto — it's faster"). Because lines carry their own source, one cart naturally spans multiple platforms.

`POST /v1/cart — a mixed-source basket`

```
{
  "cartId": "crt_9f2a1c",
  "userId": "usr_512",
  "pincode": "560103",
  "lines": [
    {
      "lineId": "ln_1",
      "canonicalSku": "AMUL-BUTTER-500G",
      "sourceId": "bigbasket",
      "qty": 1,
      "substitutable": true,
      "priceSnapshot": { "mrp": 295, "effectivePrice": 268, "etaMin": 14,
```

```
"deliveryFee": 0 }
    },
    {
      "lineId": "ln_2",
      "canonicalSku": "MAGGI-MASALA-70G-PACK12",
      "sourceId": "blinkit",
      "qty": 1,
      "substitutable": true,
      "priceSnapshot": { "mrp": 168, "effectivePrice": 152, "etaMin": 9,
"deliveryFee": 15 }
    },
    {
      "lineId": "ln_3",
      "canonicalSku": "BOAT-AIRDOPES-311PRO",
      "sourceId": "amazon",
      "qty": 1,
      "substitutable": false,
      "priceSnapshot": { "mrp": 4490, "effectivePrice": 1299, "etaMin": 1440,
"deliveryFee": 0 }
    }
  ],
  "reoptimiseAtCheckout": true
}
```

CART-LINE FIELD	MEANING
`canonicalSku`	Platform-independent product id from the Catalogue — the thing the shopper actually wants.
`sourceId`	Chosen platform for this line. Defaults to the Price-Engine winner; user-overridable.
`qty`	Units requested.
`substitutable`	May QuickCart re-source or substitute if this source goes out of stock? Shopper controls it per line.
`priceSnapshot`	Price/ETA captured when added — *advisory*. Re-verified at checkout (Doc 05).
`reoptimiseAtCheckout`	Cart-level flag: re-run ranking at checkout so a line auto-moves to a better source if one appeared.

> **The snapshot is not a promise**
>
> Prices and stock move minute-to-minute. The snapshot is what the shopper saw; the **authoritative** price and availability are re-checked at checkout, before any money is captured (see §9 and Doc 05, "Caching & freshness").

3 From cart to one internal order

At checkout, QuickCart authorises a single payment *hold* for the basket total, creates one **Order**, and hands it to the **Order Orchestrator**. The orchestrator splits the order into **sub-orders — one per source** — and routes each sub-order by what that source can actually do.

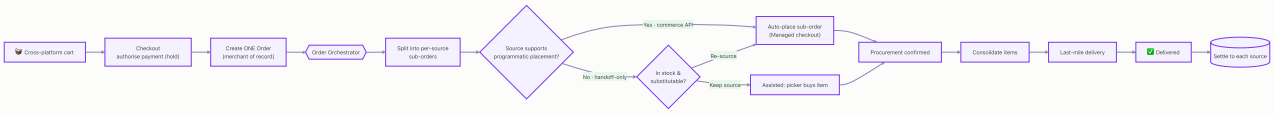


Figure 6.1 — One cart becomes one order, which fans out into per-source sub-orders, each routed by the source's capability, then re-consolidated for a single delivery.

4 Order & sub-order data model

One `ORDER` owns one `PAYMENT` and fans out to many `SUB_ORDER`s — exactly one per source. Each sub-order owns its items and, once fulfilled, a per-source `SETTLEMENT`. This shape lets QuickCart confirm, cancel, refund and settle a *single source* without touching the rest of the basket.

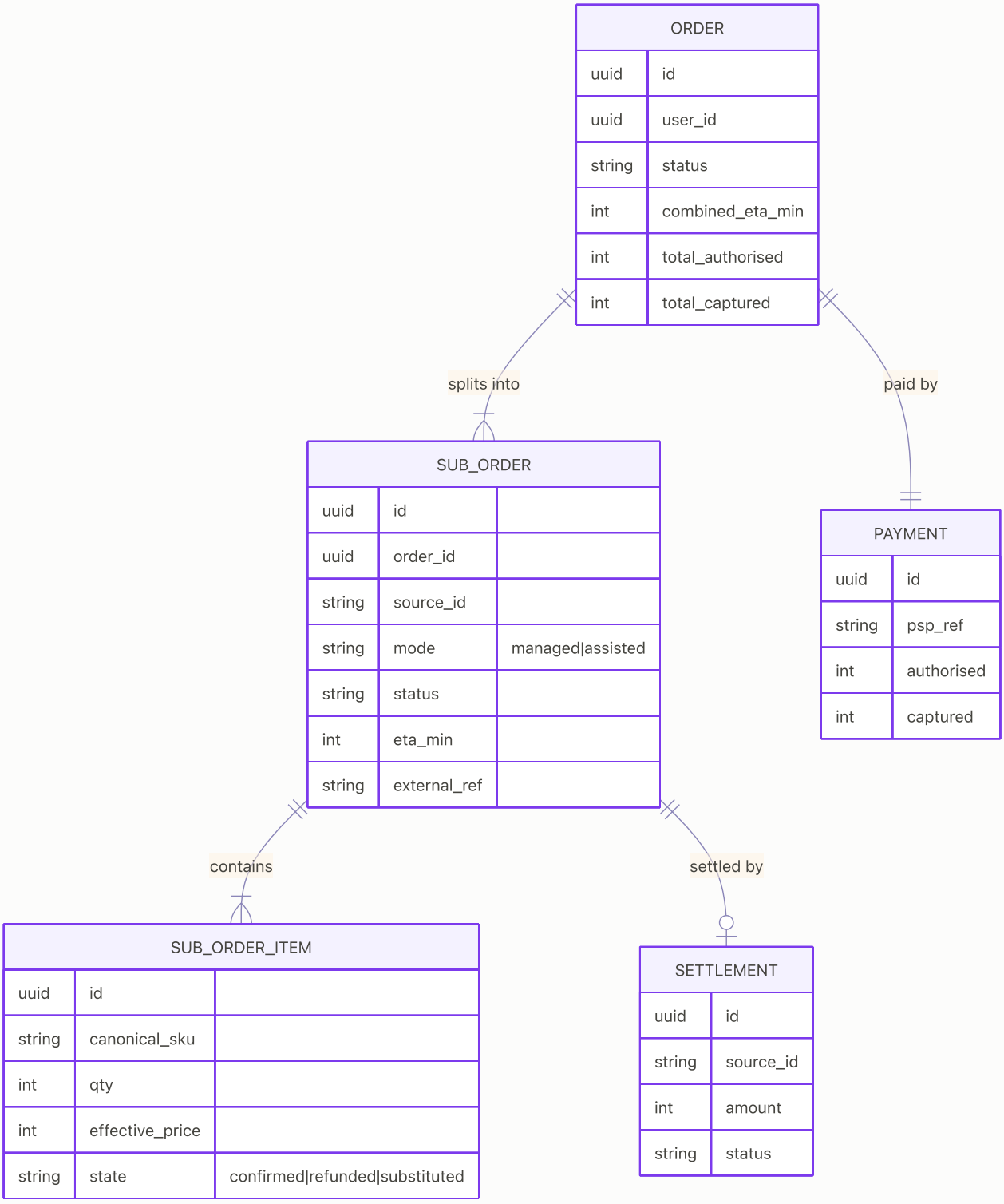


Figure 6.2 — Order → sub-orders (one per source) → items, with a single payment and per-source settlements. Illustrative fields.

Combined ETA

The order's `combined_eta_min` is the **max** of its sub-order ETAs when the basket is delivered together (pooled). If a slow line (e.g. an Amazon item at 1–2 days) would hold up fast grocery lines, QuickCart offers a **split delivery** instead — see §7.

5 Orchestration as a saga

The hard part: five sources are independent systems with no shared database and no shared transaction. QuickCart cannot "BEGIN … COMMIT" across Blinkit and Amazon. So the orchestrator runs a **saga** — a sequence of local steps, each with a **compensating action** that undoes it if a later step fails.

1. **Authorise, don't capture.** Place a hold on the full basket total. Compensation: void the hold.
2. **Re-verify per source.** Right before placing, re-check stock & price for each sub-order (a fast line can vanish between "added to cart" and "checkout").
3. **Place each sub-order** — managed (commerce API) or assisted (picker task). Compensation: cancel the placed sub-order / cancel the picker task where still possible.
4. **On failure, compensate the failed line only** — re-source it to an in-stock alternative if it's substitutable, otherwise mark that line for refund. The rest of the basket proceeds.
5. **Capture the confirmed subtotal.** Money actually taken = value of confirmed lines + delivery fee; the held remainder is released.

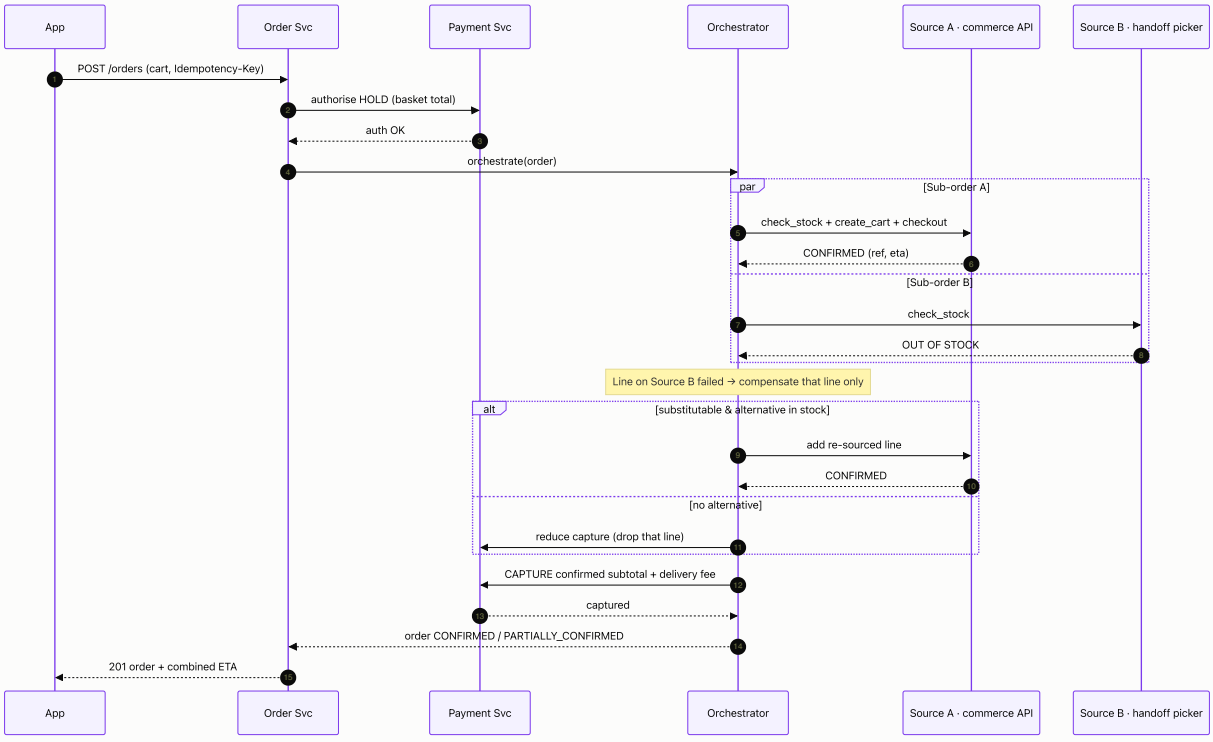


Figure 6.3 — The saga in action: parallel placement, a per-line compensation when one source is out of stock, then capture of only the confirmed amount.

`services/orders/orchestrator.js`

```
// Saga: authorise → re-verify → place per source → compensate on failure → capture
const { mcp } = require('../mcp-adaptor/client');
const payment = require('../payment/psp');
```

```
const pickerQueue = require('../fulfillment/picker-queue');

// Sources that expose a partner commerce API (programmatic placement).
// Quick-commerce sources are assisted-only unless a partner agreement is
enabled.
const PLACEABLE = new Set(['amazon', 'flipkart']);

async function orchestrate(order) {
  const saga = new Saga();
  try {
    // 1) hold funds for the whole basket (authorise, do NOT capture yet)
    const auth = await payment.authorise(order.paymentId,
order.totalAuthorised);
    saga.add(() => payment.void(auth.id));                // compensation

    // 2) one sub-order per source
    const subOrders = groupBySource(order.lines);

    // 3) place sub-orders in parallel, isolated from each other
    const settled = await Promise.allSettled(
      subOrders.map((so) => placeSubOrder(so, order, saga))
    );

    const confirmed = settled.filter(r => r.status === 'fulfilled').map(r =>
r.value);
    const failed    = subOrders.filter((_, i) => settled[i].status ===
'rejected');

    // 4) re-source or refund each failed line — never fail the whole basket
    for (const so of failed) {
      const alt = await resourceLines(so.lines, order.pincode);   // find in-
stock alternative
      if (alt) confirmed.push(await placeSubOrder(alt, order, saga));
      else     order.markLinesRefunded(so.lines);
    }

    // 5) capture ONLY what was actually confirmed
    const captureAmount = subtotal(confirmed) + deliveryFee(order);
    await payment.capture(auth.id, captureAmount);

    order.status = failed.length ? 'PARTIALLY_CONFIRMED' : 'CONFIRMED';
    await dispatchToLastMile(order, confirmed);           // pooled / direct /
split — see §7
    return order;
  } catch (err) {
    await saga.compensate();          // void hold + cancel any placed sub-
orders
    order.status = 'CANCELLED';
    throw err;
```

```
  }
}

async function placeSubOrder(so, order, saga) {
  // stock + price re-check immediately before placing (Doc 05)
  const stock = await mcp.call(so.sourceId, 'check_stock', { items: so.lines,
location: order.pincode });
  if (!stock.allAvailable) throw new OutOfStock(so.sourceId, stock.missing);

  if (PLACEABLE.has(so.sourceId)) {
    const cart   = await mcp.call(so.sourceId, 'create_cart', { items: so.lines
});
    const placed = await mcp.call(so.sourceId, 'checkout',    { cartId:
cart.cartId, address: order.address });
    saga.add(() => mcp.call(so.sourceId, 'cancel_order', { ref: placed.ref }));
    return { sourceId: so.sourceId, mode: 'managed', ref: placed.ref, lines:
so.lines, etaMin: placed.etaMin };
  }

  // handoff-only source → create an assisted picker task
  const task = await pickerQueue.enqueue({ sourceId: so.sourceId, lines:
so.lines, pincode: order.pincode });
  saga.add(() => pickerQueue.cancel(task.id));
  return { sourceId: so.sourceId, mode: 'assisted', taskId: task.id, lines:
so.lines, etaMin: task.etaMin };
}

class Saga {
  constructor() { this.undos = []; }
  add(fn) { this.undos.push(fn); }
  async compensate() { for (const undo of this.undos.reverse()) { try { await
undo(); } catch (_) {} } }
}

module.exports = { orchestrate };
```

Idempotency

Order creation carries an `Idempotency-Key` (Doc 03 §3). A retried submit returns the same order instead of double-charging or double-placing — essential when a flaky mobile network makes the app resend a checkout.

6 Fulfillment & delivery models

Once sub-orders are confirmed, the items still have to physically reach the shopper. QuickCart picks one of three delivery shapes per order, based on geography, source type and the ETA promise.

Model A

Direct

The source delivers straight to the shopper (its own rider), and QuickCart just relays live tracking. Best when a single source covers most of the basket, or for marketplace parcels (Amazon/Flipkart) on their own logistics.

Model B

Pooled / consolidated

A QuickCart rider does a **multi-pickup** run across the sources' dark stores, optionally via a consolidation hub, then makes **one drop**. Best for several quick-commerce lines going to the same address within minutes of each other.

Model C

Split

Items arrive in separate deliveries when combining would make everything slower — e.g. groceries now, the Amazon gadget tomorrow. The shopper sees each parcel tracked under the same order.

Figure 6.4 — Pooled delivery: one rider, multi-pickup across sources, one consolidated drop to the shopper.

How the model is chosen

The dispatcher scores `batchable` sub-orders by pickup proximity and ETA spread. If all confirmed lines are within a tight radius and time window → **pooled**. If one line is far slower → offer **split**. If one source dominates and delivers itself → **direct**.

7 Order lifecycle state machine

The order moves through a single state machine; each sub-order runs its own parallel mini-lifecycle (`PENDING → PLACED → PICKED → HANDED_OVER`, or `… → FAILED → RESOURCED`). The order aggregates them: it is `CONFIRMED` only when every line is sourced, and `PARTIALLY_CONFIRMED` when some lines were dropped and refunded.

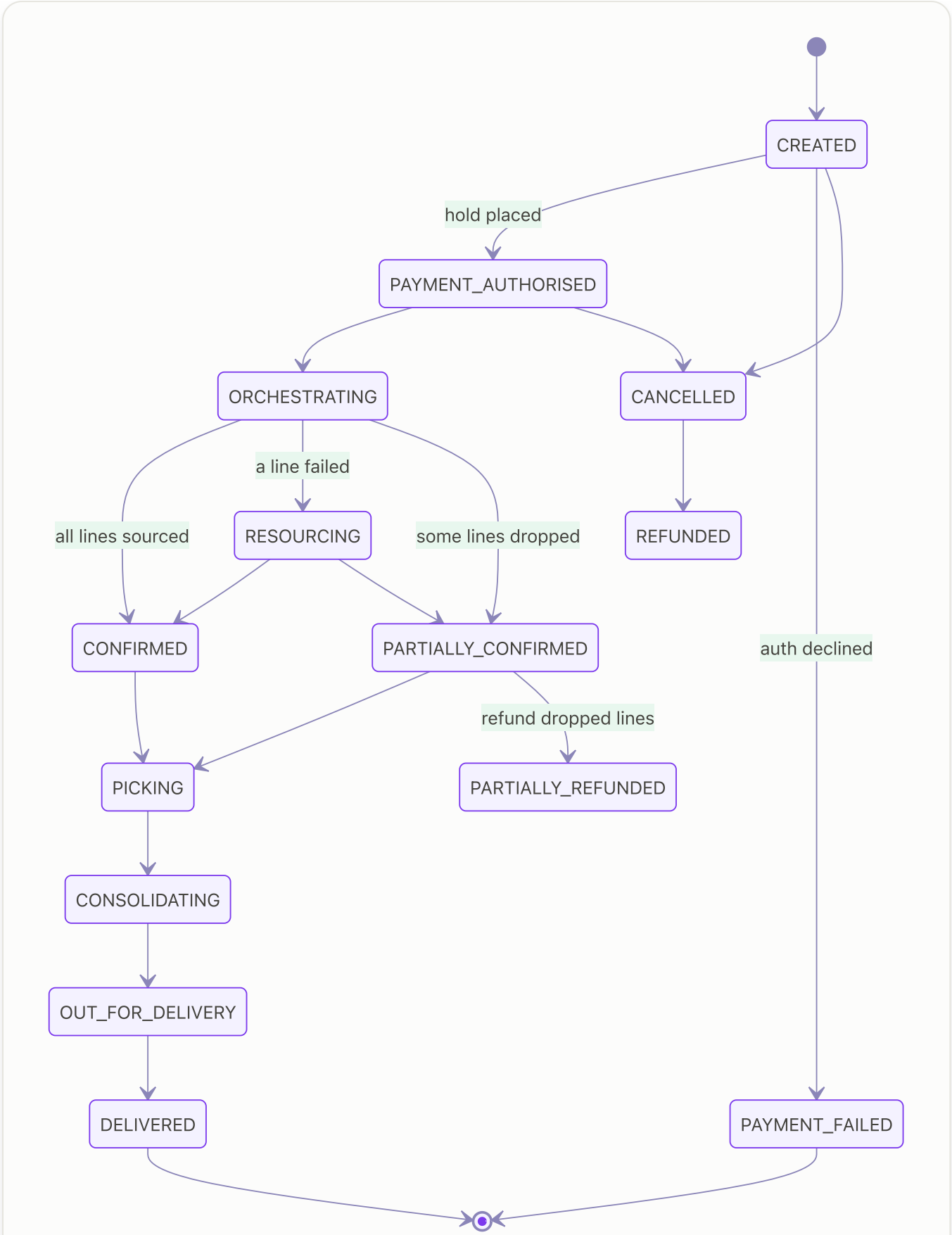


Figure 6.5 — Order lifecycle. Payment is held on entry and captured only after orchestration settles which lines are real.

8 Money flow

One shopper-facing charge; many supplier payouts. The hold-then-capture pattern is what makes partial baskets safe: QuickCart never captures money for a line it couldn't actually source.

EVENT	WHAT HAPPENS TO MONEY
Checkout	**Authorise** (hold) the full basket total on the shopper's UPI/card via the PSP. No funds move yet.
All lines confirmed	**Capture** the full total. Order → `CONFIRMED`.
Some lines dropped	**Capture** only the confirmed subtotal + delivery fee; the held remainder is released automatically. Order → `PARTIALLY_CONFIRMED`.
Per-source procurement	QuickCart pays each platform through its own supplier account; recorded as a `SETTLEMENT` row per sub-order for reconciliation.
Cancel before pickup	Void the hold (if not captured) or issue a full refund (if captured). Cancel placed sub-orders where the source allows it.
Single-line refund	Partial refund of just that line's `effectivePrice`; other lines and the delivery fee are unaffected.
Delivery fee	One fee per order (not per source). Pooled deliveries absorb multi-pickup cost; QuickCart's margin sits between supplier price and shopper price + fee.

> **Capture only after re-verification**
> Funds are **captured only after** stock and price are re-verified at checkout (Doc 05). If a confirmed line's price moved up beyond a tolerance, the shopper is re-prompted before capture — never silently charged more than shown.

9 Edge cases we handle

Out of stock at fulfillment

Re-source the line to an in-stock alternative (if `substitutable`), otherwise drop & refund just that line. Basket proceeds.

Partial availability

Shopper chooses the policy up front: **all-or-nothing** (cancel if anything is missing) or **best-effort** (ship what's available).

One source is slow

Offer a **split delivery** so fast grocery lines aren't held hostage by a 1–2 day marketplace parcel.

Price moved up at checkout

Re-confirm with the shopper before capture; re-rank the line in case another source is now cheaper.

Handoff-only, no picker coverage

If the pincode has no assisted-fulfillment coverage, re-source the line; if impossible, disallow that source for the address before checkout.

Duplicate submit

`Idempotency-Key` returns the same order — no double charge, no double placement.

Full cancellation

Run saga compensation: void/refund payment and cancel every placed sub-order or picker task where the source permits.

Single-line refund post-delivery

Damaged/wrong item → refund that line only, reconcile the dispute against the specific source's settlement.

Source outage mid-order

Circuit breaker (Doc 05) trips the source; substitutable lines re-source, others refund — order still completes.

Related documents

Doc 04 defines which sources support programmatic placement (managed checkout) versus handoff-only (assisted fulfillment). **Doc 05** defines how each line's source is chosen and re-verified. **Doc 03** holds the Payment service, order data stores and the services referenced here.