Enfinitus Newtech – Backend Services Requirements Specification

The backend is composed of coupled micro-services that communicate through a Kafka event stream and a few well-defined REST / GraphQL calls, yet persist their own data in isolated schemas of a shared PostgreSQL database.

Because every service owns its schema and emits outbox events, each one can be upgraded or scaled without touching the others while still meeting exactly-once consistency.

Enfinitus Newtech Backend – Service Catalog

1. **Pricing** – imports ENET curves and grid charges, publishes tariff snapshots.
2. **Contract & Import** – validates sign-ups, runs credit checks, emits **ContractTable** and **CustomerTable**.
3. **Customer Management** – maintains the single customer record, contracts and meter reads.
4. **Energy Management** – imports meter data, aggregates demand, calculates demand forecasts, calculates open positions and P/L.
5. **Billing** – issues invoice, matches payments, exports data to Finance.
6. **Direct-Sales Settlement** – calculates partner commissions from billed revenue.
7. **Market Communication Gateway** – hands structured JSON to a clearing-house API, which delivers all GPKE, MaBiS and BKV messages over the required protocols.
8. **Messaging Service** – sends e-mails, SMS and push notifications in response to backend events.

Pricing Service

Objectives

- Separate price-list API per sales channel (web funnel, call-centre, partners) delivering optimised prices based on channel-specific margin logic.
- Working price and fix fee for every tariff accurately reflect DSO grid fee and concession tax at ZIP-code, city, and district level.
- Daily price refresh driven by the standard load-profile–weighted wholesale price from the hourly price forward curve (HPFC).
- Architecture supports continuous improvement: pricing script lives in a Git repository, and GitHub CI/CD builds and ships a new OCI image which the scheduler can pull without downtime.

Key Constraints & Rules

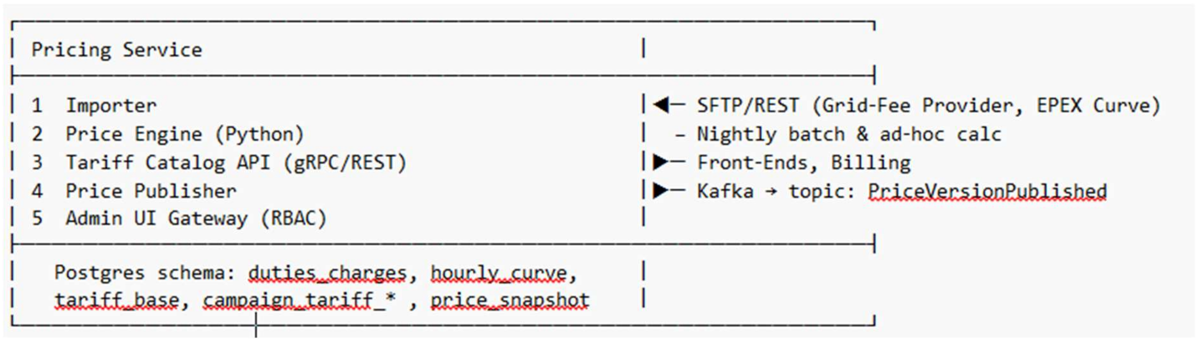
- Uniqueness – Combination of *ZIP Code* + *City* + *City District* + *TariffID* + *Date* must be unique inside a campaign table.
- Price Integrity – When inserting rows the system must validate that Working Price lies within the minimum/maximum margin defined in the Tariff Data Table.
- Versioning – Creating a new CampaignID generates an immutable price version; existing rows are never overwritten, ensuring a complete price history.

Technical Description:

1. The data basis for the applicable working prices for each unique combination ZIP-Code, City, City district and ensures we are charging the right fees and taxes on behalf of the grid provider. Each sales channel than gets its specific margin and, thus, pricing table.

2. Given the margin and additional constraints, a pricing algorithm calculates the optimal price for each ZIP code, city, city district and channel.
3. The algorithms are written in Python and packaged into a lightweight OCI image (pricing-engine:<tag>).
4. A dedicated Kubernetes CronJob (01:00 CET) or an on-demand Job spins up a pod, runs the scripts inside a Docker container, then terminates.
5. When the calculation is finished, the container POSTs the generated campaign tables to an internal endpoint POST /internal/pricing-results (gRPC fallback available). The Pricing Service persists the payload and immediately emits a PriceVersionPublished event.
6. Each sales channel, direct marketeers, and website should have its own API, since each should have its own pricing to achieve maximum revenue.

Pricing Architecture:



```
┌───────────────────────────────────────────────────────────────────┐
│ Pricing Service                                  │
├───────────────────────────────────────────────────────────────────┤
│ 1  Importer                                      │◄─ SFTP/REST (Grid-Fee Provider, EPEX Curve)
│ 2  Price Engine (Python)                         │  – Nightly batch & ad-hoc calc
│ 3  Tariff Catalog API (gRPC/REST)                │►─ Front-Ends, Billing
│ 4  Price Publisher                               │►─ Kafka → topic: PriceVersionPublished
│ 5  Admin UI Gateway (RBAC)                       │
├───────────────────────────────────────────────────────────────────┤
│   Postgres schema: duties_charges, hourly_curve,  │
│   tariff_base, campaign_tariff_* , price_snapshot │
└───────────────────┴───────────────────────────────────────────────┘
```

Pricing data schemas:

Table	Purpose	Key Columns	Operational Notes
Tariff Data Table	Holds master data for every tariff product. The parameters here apply **globally**, independent of time or location.	1. **TariffID** – primary key 2. **TariffName** (string) 3. **Min Contract Duration** (months) 4. **Minimum Margin / Maximum Margin** – pricing corridor the engine must respect 5. **Consumption Range** – target annual-consumption band (e.g. 0-5 MWh) 6. **Billing Method / Monthly Instalments** (e.g. monthly, yearly) 7. **End-Of-Contract Rule** – renewal or cancellation logic	The pricing engine reads this table first. Every downstream price record must reference a valid **TariffID** and stay inside the defined margin corridor.
Campaign Table	Stores the **concrete** working price for dynamic tariffs (price	1-4. **ZIP Code, City, City District, Date** – forms a unique geo-	Updated as often as required (typically nightly).

Dynamic Tariff	can change daily/weekly).	time key 5. **TariffID** – FK to Tariff Data Table 6. **Working Price** (ct/kWh) 7. **Fix Fee** (€/month) 8-9. **Variable Bonus / Fixed Bonus** – campaign incentives 10. **CampaignID** – groups all rows belonging to one price version	Sales front-ends query the current price from here.
Campaign Table Fix12 Tariff	Fixed-price product with a 12-month guarantee. Same schema as the Dynamic table.	Same as above	Separate table avoids mixing fixed and dynamic price versions.

Contract Service & Import Process

Objectives

Upon completion of the workflow each switching request results in:

- A fully **Contract** (row in **ContractTable** with status Active) linked to a validated tariff and campaign snapshot.
- A definitive entry of each customer with its corresponding market location (MaLo) data is stored in **CustomerMaLo** (a row in **CustomerMaLo**) containing all market-location data required for billing and balancing.
- A correspondingly updated or newly created **customer** master record.
- A complete, immutable **audit trail** in **ContractEvent** allowing any regulator or internal auditor to replay the process end-to-end.

These outcomes guarantee that downstream services—Billing, Customer Management, and Energy Management—receive consistent and dependable data without manual rework.

Key Constraints & Rules

- **24 Hour Switching SLA** – A signed contract must be submitted to the Distribution Grid Operator via GPKE ANMELDUNG (*Lieferantenwechsel Anmeldung/ Supplier Switch Apllication*) within 24 h.
- **Data Completeness** – A MaLo (market location) record with valid BDEW code and meter number is mandatory before the GPKE message is sent.
- Only customers with a **positive respond message from the GPKE process** are imported to activated customers
- **Audit Trail** – All status changes are immutable and timestamped in a contract event log.

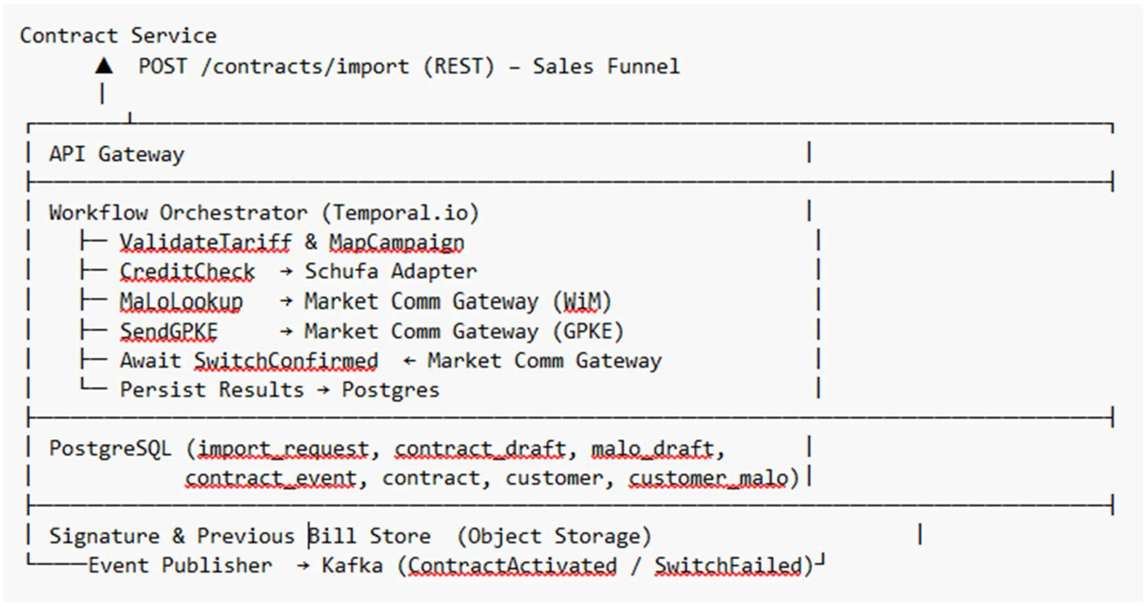
Technical Description

1. **Import Request Logged** – Sales funnel sends an authenticated POST /contracts/import with customer data, selected **Campaign ID**, and a link to the signed contract. The raw JSON payload persisted in the import_request table (idempotency key, timestamp). An

optional previous-bill upload is stored in object storage and its URL is added to the same row.

2. **Contract Draft Creation** – Using the ***CustomerID*** and ***Campaign ID***, the service generates a unique **ContractID** and inserts a row in **ContractDraftTable** (status =Draft, pricing snapshot columns working_price, fix_fee).
3. **Validation & Enrichment** – The Workflow Orchestrator an annual-consumption sanity check; results are patched into the existing **ContractDraftTable** row and a corresponding **Contract_Event** (type =ValidationPassed/Failed) is appended.
4. **Market-Location Mapping** – The customer specific MaLo data (BDEW code, meter number, previous provider) is retrieved via the Market Communication Gateway and written to **Malo_Draft**, linked by **ContractId**.
5. **Manual Oversight & Audit** – Ops edits to **Malo_Draft**, are stored as new versions; every change triggers a **Contract_Event** (type =ManualEdit) and must be confirmed via an admin flag in the same table.
6. GPKE Switch & Activation – After sending ANMELDUNG the service waits for **SwitchConfirmed**. Upon success, it migrates the draft rows: **ContractDraft** → **Contract_Table** (status =Active) and **MaloDraft** → **CustomerMalo**; it also inserts the customer (if new) into the customer table. A final **ContractEvent** (type =Activated) is written and **ContractActivated** is published; failures move the draft to **SwitchFailed**

Contract Service & Import Process Architecture:



```
Contract Service
      ▲  POST /contracts/import (REST) – Sales Funnel
      |
┌─────┴──────────────────────────────────────────────────────────────┐
| API Gateway                                           |
├────────────────────────────────────────────────────────────────────┤
| Workflow Orchestrator (Temporal.io)                   |
|    ├─ ValidateTariff & MapCampaign                     |
|    ├─ CreditCheck   → Schufa Adapter                   |
|    ├─ MaLoLookup    → Market Comm Gateway (WiM)        |
|    ├─ SendGPKE      → Market Comm Gateway (GPKE)       |
|    ├─ Await SwitchConfirmed ← Market Comm Gateway      |
|    └─ Persist Results → Postgres                       |
├────────────────────────────────────────────────────────────────────┤
| PostgreSQL (import_request, contract_draft, malo_draft, |
|            contract_event, contract, customer, customer_malo)|
├────────────────────────────────────────────────────────────────────┤
| Signature & Previous Bill Store  (Object Storage)             |
└────Event Publisher  → Kafka (ContractActivated / SwitchFailed)┘
```

Contract Service & Import Process data schemas:

Table	Purpose	Key Columns (examples)	Operational Notes
ContractDraft	Temporary holding area for every new contract as soon as it is imported. Holds the full pricing snapshot and	funnelId, customerId, **ContractId**, tariffId, workingPrice, fixFee, expectedConsumption, address fields, contractDraftDate, desiredContractChangeDate, expectedContractEndDate, schufaStatus, iban, sepaMandate, otherBillingInfo	Created by the POST /contracts/import call (status =Draft). Updated in place during validation, then either migrated to **contract** on success or flagged SwitchFailed.

	customer meta-data while the switch is in progress.		
MaloDraft	Staging table for market-location (MaLo) details that are required to launch the supplier-switch. Lets Ops amend data before it becomes final.	customerId, contractId, marketLocationIdentifier, hasOwnMSB (flag), meterNumber, previousProviderCode, previousAnnualConsumption, possibleSupplierChangeDate, schufaScoreAccepted, maloDraftStatus	Written after MaLo lookup via the Market Communication Gateway. Any manual edits create new row versions; admin approval toggles maloDraftStatus = Approved.
CustomerMalo	Definitive MaLo record once the GPKE switch is confirmed. Used by downstream services (Billing, Energy Mgmt) to link contracts and meters to the grid.	customerId, contractId, marketLocationIdentifier, hasOwnMSB, meterNumber, previousProviderCode, previousAnnualConsumption, possibleSupplierChangeDate, schufaScoreAccepted, changeProcessStatus	

Customer Management Service

Objectives

- A single authoritative **CustomerTable** row per end-customer, enriched with current address and contact data.
- Every active contract reflected in **ContractTable** and linked via **ContractRefTable** so that the total number of contracts is always consistent.
- All meter readings for each customer consolidated in **MeterReadingTable**, guaranteeing no orphaned meters.
- Up-to-date pricing snapshots and special conditions recorded in **CustomerPricingTable**
- A complete, append-only **CustomerEventHistory** timeline for regulatory audits and customer support inquiries.
- All artefacts are stored exclusively in the PostgreSQL database—no additional object storage is required by this service.

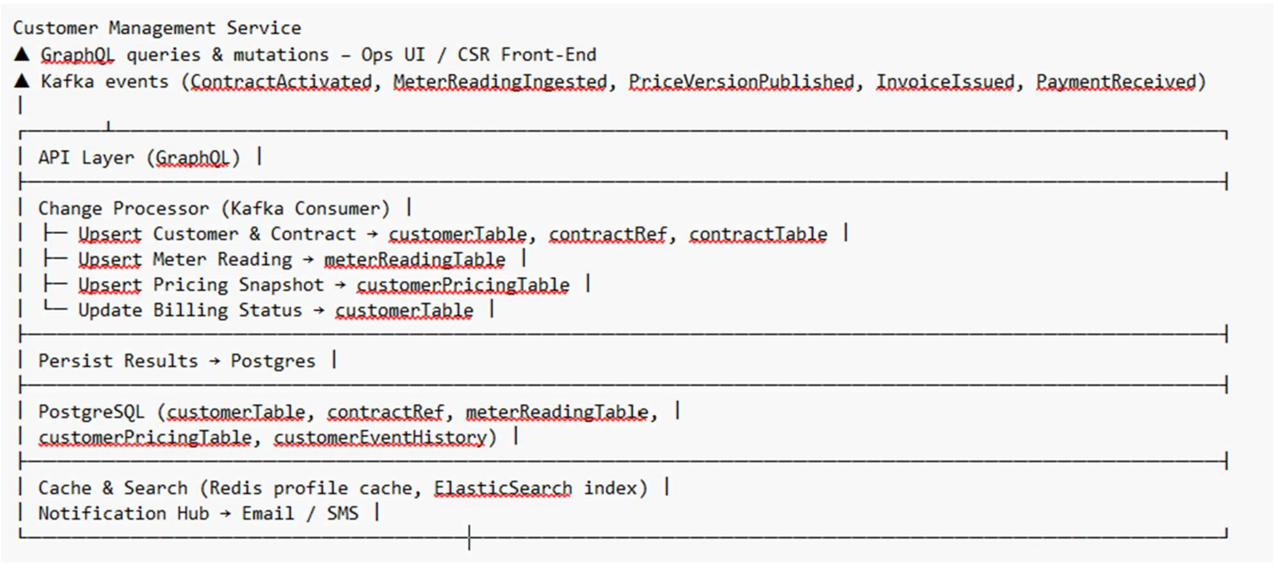
Key Constraints & Rules

- **Single Source of Truth** – Each **CustomerId** is unique across all services.
- **GDPR Compliance** – Personal data must be **deletable** on request within 30 days.
- **Meter–Contract Consistency** – Every active contract must reference at least one meter; orphaned meters trigger alerts.

Technical Description:

1. **Contract Intake** – The service subscribes to ContractActivated. For each event it inserts the new row into **CustomerTable** (if not present) and creates a **ContractRefTable** link to the owning customer.
2. **Meter Reading Processing –** Listens to **MeterReadingIngested**. Validates that the meter belongs to the customer via **CustomerMaloTable**, then stores the reading in **MeterReadingTable** and updates rolling consumption aggregates.
3. **Pricing Snapshot Management** – When **PriceVersionPublished** or special-pricing updates arrive, resolves the active tariff for each contract and inserts a record into **CustomerPricingTable** if the working_price or fixed_fee changed.
4. **Profile Maintenance** – CSR or self-service API calls mutate name, address or contact fields; writes persisted to **CustomerTable** and an entry is appended to **CustomerEventHistory** (type = ProfileUpdated).
5. **Accounts Receivable Tracking** – Consumes InvoiceIssued and PaymentReceived events. Updates billingStatus and arBalance columns in customer_table so support agents have real-time payment status.
6. **Search Index Sync** – After any write to customer_table or contract_table the change-processor pushes a document into an ElasticSearch index, powering the Customer Base 2.0 UI.
7. **Notification Hub** – On key events (SwitchCompleted, PaymentOverdue) the service triggers emails or SMS via the Notification Hub.
8. **GDPR and Compliance** – A **DeleteCustomer** request anonymises PII columns in **CustomerTable** and cascades to **ContactInfo** while retaining surrogate keys for financial audit.

Customer Management Architecture:



```
Customer Management Service
▲ GraphQL queries & mutations – Ops UI / CSR Front-End
▲ Kafka events (ContractActivated, MeterReadingIngested, PriceVersionPublished, InvoiceIssued, PaymentReceived)
|
┌───────┴──────────────────────────────────────────────────────────────────────────────┐
| API Layer (GraphQL) |
├──────────────────────────────────────────────────────────────────────────────────────┤
| Change Processor (Kafka Consumer) |
| ├─ Upsert Customer & Contract → customerTable, contractRef, contractTable |
| ├─ Upsert Meter Reading → meterReadingTable |
| ├─ Upsert Pricing Snapshot → customerPricingTable |
| └─ Update Billing Status → customerTable |
├──────────────────────────────────────────────────────────────────────────────────────┤
| Persist Results → Postgres |
├──────────────────────────────────────────────────────────────────────────────────────┤
| PostgreSQL (customerTable, contractRef, meterReadingTable, |
| customerPricingTable, customerEventHistory) |
├──────────────────────────────────────────────────────────────────────────────────────┤
| Cache & Search (Redis profile cache, ElasticSearch index) |
| Notification Hub → Email / SMS |
└──────────────────────────────┼───────────────────────────────────────────────────────┘
```

Customer Management – Data Schemas

Table	Purpose	Key Columns (examples)	Operational Notes
Customer Table	Authoritative master record for each end-customer (B2C or B2B).	customerId PK, salutation, firstName, lastName, companyName?, addressStreet, addressPostalCode, addressCity, addressCountry, email, phone, contactPrefsJson, schufaScorePct, nrContracts GENERATED*, arBalanceCent, billingStatus enum(Current,Overdue,Dunning,Blocked), createdAt, updatedAt, deletedAt?	*nrContracts is a generated column counting ACTIVE ContractRef rows; helps enforce **Single-Source-of-Truth** rule. PII columns are encrypted at rest; on DeleteCustomer request they are anonymised and deletedAt is stamped (kept ≤ 30 days until purge).
Contract Table	Stores every *active* retail contract.	contractId PK, tariffId, campaignId, workingPriceCtKWh, fixFeeEURMonth, expectedConsumptionKWh, contractStartDate, contractEndDate, billingInterval enum(Monthl	Physical FK to **ContractRefTable** enforces that a contract belongs to at least one customer; triggers forbid delete while active.

		y,Quarterly,Annual), contractStatus enum(Active,Suspended, Terminated,SwitchInProgress), cancellationStatus enum(None,Requested,Confirmed), createdAt	
Contract RefTable	Maintains 1-N mapping between customers and contracts (landlord/tenant scenarios).	contractRefId PK, customerId FK, contractId FK, relationType enum(Owner, CoTenant,Former), activeFlag, createdAt	Drives nrContracts generated column; ON DELETE CASCADE to keep referential integrity.
Customer PricingTable	Historic price snapshots incl. special bonuses.	pricingId PK, contractId FK, campaignId, priceValidFrom, workingPriceCtKWh, fixFeeEURMonth, variableBonusCtKWh, fixedBonusEUR, createdAt	Inserted only when price components change (checked hash-diff) – avoids write amplification.
MeterReadingTable	Consolidated meter reads per customer.	readingId PK, customerId FK, contractId FK, meterNumber, readingTimestamp, readingKWh, readingSource enum(IoTGateway,Upload,Estimate), importedAt	FK (meterNumber) REFERENCES **CustomerMaloTable**; a deferred constraint ensures meter belongs to contract owner, satisfying *Meter–Contract Consistency*. UNIQUE (meterNumber, readingTimestamp) prevents duplicates.
Customer EventHistory	Append-only audit/event log for support & compliance.	eventId PK, customerId FK, eventTimestamp, eventType enum(ProfileUpdated,ContractActivated,MeterReadingIngested,PriceChanged,PaymentReceived,GDPRDelete), payloadJson, performedBy, correlationId	Stored directly in Postgres JSONB column (payloadJson); no external object store, meeting "PostgreSQL-only" artefact requirement. Retention = unlimited.

Note on Storage: All artefacts remain inside the PostgreSQL cluster—no S3/object storage dependency inside this service. Binary artefacts (e.g. invoice PDFs) are referenced but stored by Billing Service.

Energy Management Service

Objectives

- Ingest Data from ENET – capture hourly demand forecasts (SLP and day-ahead) and historic consumption series for every customer market location.
- Compute Current & Future Demand – aggregate real-time consumption and generate rolling forecasts (intraday, day-ahead, week-ahead, year-ahead) per grid area.
- Calculate Open Position – net contracted demand against hedged volumes and maintain a live, hourly-updated open-position ledger (MWh & EUR).
- Regulatory Reporting – automatically compile and submit all mandatory reports to the balancing responsible party (BKV) and MaBiS balancing-group authorities within statutory deadlines.
- Expose Insights to Front-Ends – stream the open position, forecast curves and alerts to trader dashboards and operations front-ends via Kafka topics and a GraphQL resolver.
- Update Financial Statements – derive energy P/L, imbalance costs and feed monthly & year-to-date results into the ProfitLoss table for Finance & B

Key Constraints & Rules

- Data Freshness – ENET demand forecasts and historic consumption must be ingested and persisted within 15 minutes of publication.
- Open-Position Discipline – Recalculate the net open position hourly; if unhedged exposure exceeds ±20 % of forecast demand (or > 5 MWh), an automated hedge order must be emitted.
- Regulatory Deadlines –
- BKV schedule nominations no later than 14:30 CET D-1.
- MaBiS balancing-group reports submitted no later than T + 7 working days.
- Financial Integrity – Post a P/L snapshot to the ProfitLoss table daily by 02:00 CET and the final month-close entry within 2 business days.
- Reliability – Service availability ≥ 99.7 % monthly, with automated retries on ENET/TSO API failures (exponential back-off, max 6 h)

Technical Description

The service consumes **all forecast inputs exclusively from ENET**:

- **SLP profiles** (static, yearly) for standard load categories.
- **Intraday, day-ahead, week-ahead and year-ahead demand curves** via ENET's REST endpoints.

Historic and near-real-time **meter data** are received from DSO streams. After each ingestion cycle the service:

1. Normalises ENET forecast payloads into the SlpDemandProfile, DayAheadDemandForecast and WeeklyDemandForecast tables.
2. Aggregates realised consumption and calculates **current demand** plus rolling **future demand** horizons.

3. Nets contracted volume against hedged trades to produce the **OpenPosition** (MWh & EUR) and stores it.
4. Emits Kafka events: forecast.intraday, forecast.weekly, position.open, pl.statement for traders, Finance and dashboards.
5. Generates and dispatches BKV schedule nominations (D-1 14:30 CET cut-off) and MaBiS balancing-group reports (T + 7 wd).
6. Writes daily P/L snapshots to ProfitLoss and posts month-close entries for Finance.

Reliability mechanisms include automatic retries on ENET API failures with exponential back-off (max 6 h) and compensation logic when DSOs deliver late actuals.

Energy Management Service Architecture

```
Energy Management Service
▲ GraphQL queries & subscriptions - Trader UI / Monitoring Dashboard
▲ Kafka events (ENETForecastImported, ConsumptionRealised, ContractOnboarded)

┌─────────────────────────────────────────────────────────────────────────────┐
│ API Layer (GraphQL + gRPC)                                                  │
└─────────────────────────────────────────────────────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────────────────────────────────────────┐
│ Ingestion Pipeline (Kafka Consumer ×3)                                      │
│ ├─ Upsert ENET Forecast → weeklyDemandForecastTable, dayAheadForecastTable  │
│ ├─ Upsert Historic Consumption → realisedConsumptionTable                   │
│ └─ Upsert Contract Load Profiles (SLP)                                      │
└─────────────────────────────────────────────────────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────────────────────────────────────────┐
│ Demand Aggregator (Stream Job)                                              │
│ ├─ Hourly aggregate of forecast & contracts                                 │
│ └─ Emits DemandAggregated event (kWh/h)                                     │
└─────────────────────────────────────────────────────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────────────────────────────────────────┐
│ Position Manager (Stream Job)                                               │
│ ├─ Joins AggregatedDemand with RealisedConsumption                          │
│ ├─ Calculates Open Position (buy/sell MWh, €/MWh)                           │
│ ├─ Publishes OpenPositionCalculated event                                   │
│ └─ Sends live position via GraphQL Subscription to Trader UI                │
└─────────────────────────────────────────────────────────────────────────────┘
          │
          ├──────────────┐
          ▼              ▼
┌────────────────────┐ ┌──────────────────────────────────────────────────────┐
│ BKV / MaBiS Reporter│ │ Financial Posting Worker                            │
│ • Weekly balancing  │ │ • Captures P&L deltas                               │
│ • EDIFACT messages  │ │ • Posts journal lines to Finance topic              │
└────────────────────┘ └──────────────────────────────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────────────────────────────────────────┐
│ Persist Results → Postgres                                                  │
└─────────────────────────────────────────────────────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────────────────────────────────────────┐
│ PostgreSQL (core tables)                                                    │
│  weeklyDemandForecastTable     │ realisedConsumptionTable                   │
│  dayAheadForecastTable         │ demandAggregatedTable                      │
│  openPositionTable             │ profitLossTable                            │
│  procurementTable              │ bkvSubmissionLog                           │
└─────────────────────────────────────────────────────────────────────────────┘
          │
          ▼
┌─────────────────────────────────────────────────────────────────────────────┐
│ Cache & Search (Redis price cache, ElasticSearch index)                     │
```

Energy Management Data Schemas

Table	Purpose	Key Columns	Operational Notes
SlpDemandProfile	Static load-profiles by category & hour.	profileCategory, hourOfYear, consumptionKWh	Loaded once per year from ENET.

WeeklyDemandForecast	Aggregated 168-h forecast for grid area.	forecastWeek (YYYY-WW) PK, gridArea, forecastKWh, basis enum(SLP,TSO,DSO), version, createdAt	New version inserted on every recalculation.
DayAheadDemandForecast	24-h rolling forecast with price signal.	forecastDate, hour (0-23), gridArea, demandKWh, forecastPriceEURMWh, version	Joined with HPFC for margin planning.
RealisedConsumption	Actual load measured by DSOs.	datetime, gridArea, consumptionKWh, source, ingestedAt	Drives forecast-error analysis.
OpenPosition	Quantity still to hedge.	timestamp PK, qtyMWh, avgPriceEURMWh, status enum(Open,Hedged), hedgeOrderId?	Trader UI subscribes.
ProfitLoss	P&L snapshot for finance.	plId PK, period (YYYY-MM), purchasedQtyMWh, purchasedCostEUR, soldQtyMWh, soldRevEUR, imbalanceCostEUR, marginEUR	Written at month-close and whenever trades settle.

Billing Service

Objectives

- Bill every active contract on time and without errors.
- Keep invoices, credit notes, and journals audit-proof and immutable.
- Expose up-to-date billing status and A/R balance for each customer in near real-time.
- Automate payments (SEPA), matching, and dunning to minimise overdue balances.
- Store all billing data—including PDFs—only in PostgreSQL while meeting GDPR and tax-retention rules.

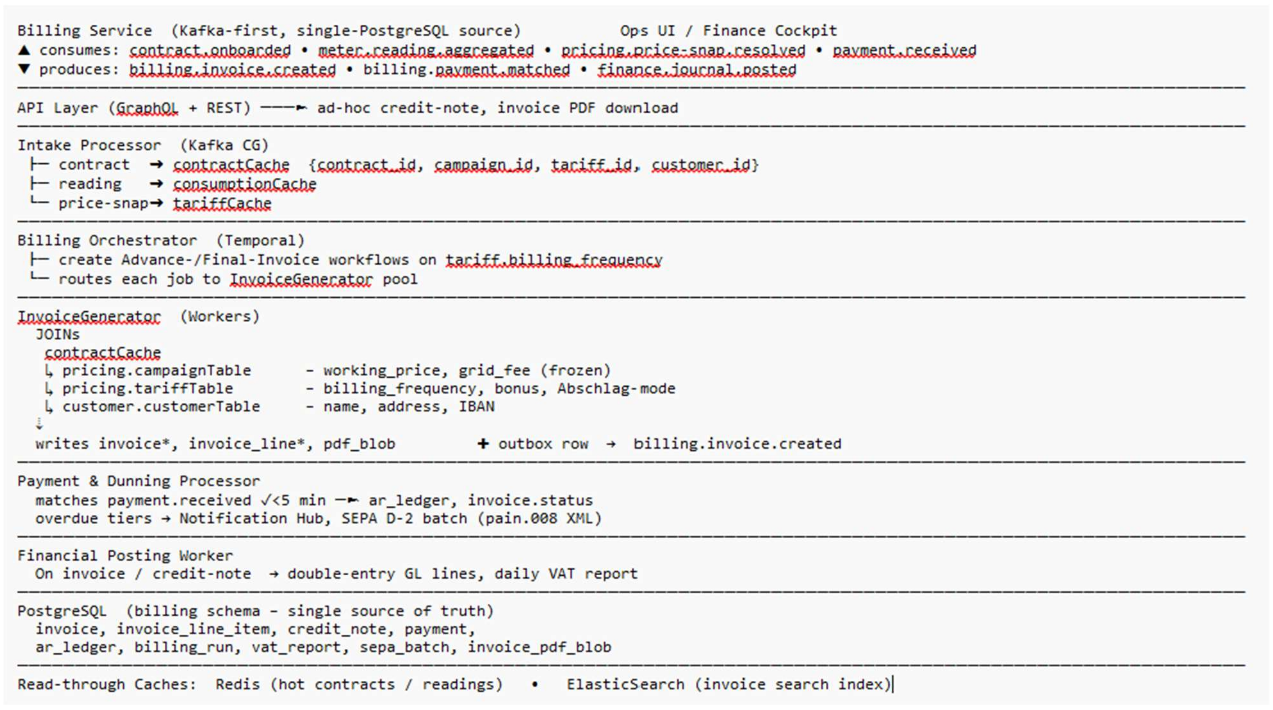
Key Constraints & Rules

- **PostgreSQL is the sole source of truth** – all other stores are read-through caches only.
- **Exactly-once events** – InvoiceCreated, PaymentReceived, and JournalPosted carry unique IDs enforced by DB constraints.
- **Monotonic numbering** – invoice & credit-note numbers increase within each calendar year and are never reused.
- **VAT-aware rounding** – line items to €0.01, totals "round half up," always reconcile.
- **Tariff immutability** – any price change gets a new priceVersionId; past invoices never re-rate in place.
- **SEPA lead time D-2** – direct-debit files are generated two banking days before collection.
- **Retention & GDPR** – financial records kept ≥ 10 years; PII anonymised within 30 days on DeleteCustomer.
- **Service SLOs** – p95 PDF render < 2 s, p99 payment match < 5 min, monthly uptime ≥ 99.7 %.

Technical Description

1. Contract **snapshot cache** – when contract.onboarded arrives the Intake-Processor stores contract_id, campaign_id, tariff_id, customer_id in Redis and the billing.contract_cache table; this gives the InvoiceGenerator fast joins to campaignTable, tariffTable and customerTable inside Postgres.
2. Price look-up – at rating time the InvoiceGenerator joins
3. contract_cache → pricing.campaignTable (key: campaign_id + invoice_period_date) to pull the effective working-price ct/kWh and grid-fee €/month that were frozen when the contract was sold.
4. Tariff metadata – the same query joins pricing.tariffTable (key: tariff_id) to read billing_frequency_months, advance_payment_mode (ABSCHLAG_MONTHLY, ABSCHLAG_QUARTERLY,ABSCHLAG_YEARLY) and any fixed bonuses; this drives when the Temporal scheduler fires MonthlyBillingWorkflow, QuarterlyBillingWorkflow, or YearlyBillingWorkflow.
5. Customer enrichment – customer name, address and IBAN are fetched on demand from customerTable (FK customer_id) so that invoice PDFs contain correct recipient data and SEPA files have the mandate reference.
6. Rating algorithm – energy-line items = kWh × working-price; grid-fee line = monthly grid fee from campaign; VAT rounding follows § 14 UStG. All numbers are written to invoice & invoice_line_item in one ACID TX together with an outbox row.
7. Outbox → Kafka – Debezium streams the outbox row to billing.invoice.created; consumers (Settlement, Finance) can trust the prices because they stem from immutable campaignTable rows.
8. Advance-payment (Abschlag) schedule – if the tariff says "12 monthly Abschläge + annual reconciliation" the Orchestrator instantly spawns 12 pro-forma invoices (type =ADVANCE) using the same campaign prices, and sets due_date = start_of_month + tariff.grace_days.
9. SEPA generation – the Payment-Processor groups ADVANCE invoices with method = SEPA and creates a pain.008 in sepa_batch exactly D-2 before the due_date; mandate details come from customerTable.
10. Re-rating safety – if Pricing later publishes a new campaignTable row, existing invoices are untouched; only new invoice periods will point to the new campaign_id / price_version_id (tariff immutability rule).
11. Single-store guarantee – all joins (campaignTable, tariffTable, customerTable) happen inside the shared PostgreSQL cluster; Redis and Elastic remain read-through caches so legal record-keeping and GDPR deletion stay centralised.

Billing Service Architecture


```
Billing Service  (Kafka-first, single-PostgreSQL source)          Ops UI / Finance Cockpit
▲ consumes: contract.onboarded • meter.reading.aggregated • pricing.price-snap.resolved • payment.received
▼ produces: billing.invoice.created • billing.payment.matched • finance.journal.posted
──────────────────────────────────────────────────────────────────────────────────────────────────────────
API Layer (GraphQL + REST) ───► ad-hoc credit-note, invoice PDF download
──────────────────────────────────────────────────────────────────────────────────────────────────────────
Intake Processor  (Kafka CG)
 ├─ contract   ➜ contractCache  {contract_id, campaign_id, tariff_id, customer_id}
 ├─ reading    ➜ consumptionCache
 └─ price-snap➜ tariffCache
──────────────────────────────────────────────────────────────────────────────────────────────────────────
Billing Orchestrator  (Temporal)
 ├─ create Advance-/Final-Invoice workflows on tariff.billing_frequency
 └─ routes each job to InvoiceGenerator pool
──────────────────────────────────────────────────────────────────────────────────────────────────────────
InvoiceGenerator  (Workers)
  JOINs
   contractCache
   ↳ pricing.campaignTable      - working_price, grid_fee (frozen)
   ↳ pricing.tariffTable        - billing_frequency, bonus, Abschlag-mode
   ↳ customer.customerTable     - name, address, IBAN
  ↓
  writes invoice*, invoice_line*, pdf_blob        ✚ outbox row  →  billing.invoice.created
──────────────────────────────────────────────────────────────────────────────────────────────────────────
Payment & Dunning Processor
  matches payment.received ✓<5 min ─► ar_ledger, invoice.status
  overdue tiers → Notification Hub, SEPA D-2 batch (pain.008 XML)
──────────────────────────────────────────────────────────────────────────────────────────────────────────
Financial Posting Worker
  On invoice / credit-note  → double-entry GL lines, daily VAT report
──────────────────────────────────────────────────────────────────────────────────────────────────────────
PostgreSQL  (billing schema - single source of truth)
  invoice, invoice_line_item, credit_note, payment,
  ar_ledger, billing_run, vat_report, sepa_batch, invoice_pdf_blob
──────────────────────────────────────────────────────────────────────────────────────────────────────────
Read-through Caches:  Redis (hot contracts / readings)   •   ElasticSearch (invoice search index)
```

Billing Service data schemas

Table name	Primary key	Key columns (✓ = FK)	Purpose / notes
contract_cache	contract_id (UUID)	✓customer_id, ✓campaign_id, ✓tariff_id, contract_start, contract_end, payment_method, iban_masked	Normalised snapshot of the contract as sold; joined at billing time so the rating logic never reaches across schemas.
invoice	invoice_id (UUID)	✓customer_id, ✓contract_id, ✓campaign_id, ✓tariff_id, price_version_id, invoice_no, invoice_type (FINAL,ADVANCE,CREDIT), invoice_date, period_start, period_end, gross_total_eur, net_total_eur, vat_total_eur, billing_status (OPEN,PAID,DUNNING)	Immutable header; campaign_id + price_version_id freeze the working-price & grid fee that applied when the invoice was created.
invoice_line_item	line_id (UUID)	✓invoice_id, charge_type (ENERGY,GRID,TAX,BONUS,FIXED_FEE), quantity_kwh, unit_price_ct,	Detailed rating; sums must equal header

		line_net_eur, vat_rate_pct, line_vat_eur	totals (checked by DB trigger).
invoice_pdf_blob	invoice_id (PK & FK)	pdf_data (BYTEA), sha256_hash, generated_ts	Stores rendered PDF inside Postgres → single-datastore rule.
credit_note	credit_note_id (UUID)	✓customer_id, ✓original_invoice_id, credit_no, credit_date, gross_total_eur, reason_code	Negative invoice; also inserted into **invoice** with invoice_type = CREDIT.
payment	payment_id (UUID)	✓customer_id, ✓invoice_id, txn_ref_ext, method (SEPA,BANK_TX,REVERSAL), amount_eur, received_ts, match_status (MATCHED,SUSPENSE)	Raw cash movements imported from EBICS or bank CSV.
suspense_account	suspense_id (UUID)	✓payment_id, reason_text, resolved_ts	Holds unmatched payments pending manual intervention.
ar_ledger	ledger_entry_id (UUID)	✓customer_id, fiscal_year_month, doc_type (INV,CRN,PAY), doc_id, gl_account, debit_eur, credit_eur, posting_ts	Double-entry ledger for AR, revenue, VAT; feeds Finance.
billing_run	run_id (UUID)	run_type (MONTHLY,MOVE_OUT,RE_RATING), started_ts, finished_ts, contracts_billed, invoices_created, status	Tracks each batch for monitoring, retry & SLO metrics.
vat_report	vat_report_id (UUID)	period_year, period_month, vat_7_eur, vat_19_eur, generated_ts	Daily VAT totals handed to Finance by 09:00 CET.
sepa_batch	batch_id (UUID)	collection_date, generated_ts, pain_008_xml (BYTEA), status (PENDING,SENT,ACKED)	Direct-debit batches created D-2; XML kept for audit.

Market Communication Gateway – API Integration

Objectives

- Send every GPKE, MaBiS and BKV message **indirectly** by handing structured JSON to an accredited clearing-house API, which converts, signs and delivers over AS2/SMTP/SFTP. Our gateway only shapes the data, persists it, and tracks the provider's acknowledgement

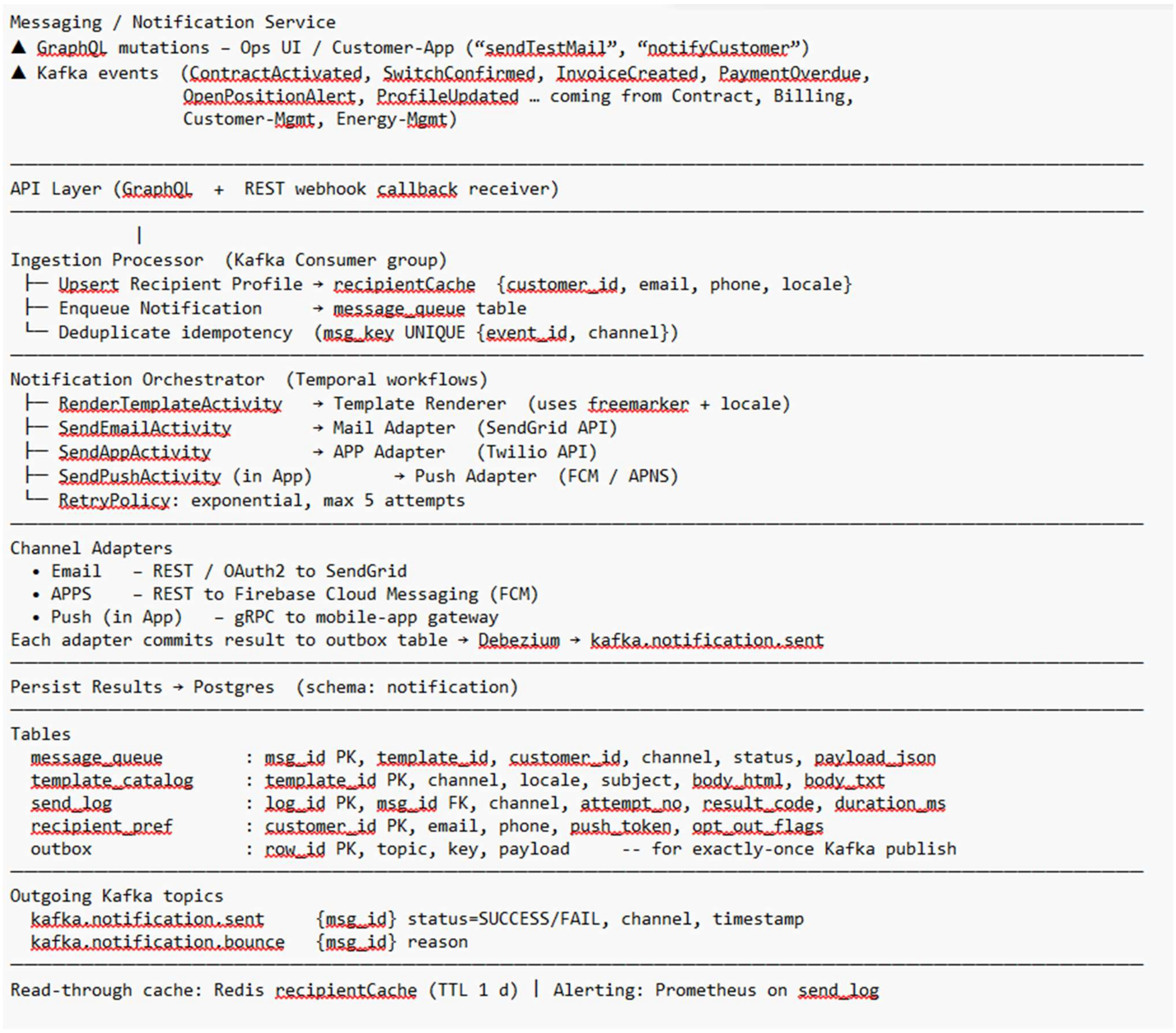
Key Constraints & Rules

- **Provider-API contract** – we speak REST/HTTPS, backward-compatible.
- **Exactly-once** – {message_type, market_location_id, sequence} is UNIQUE in gateway.outbox; retries are idempotent.
- **Deadline assurance** – GPKE < 24 h, BKV D-1 14:30 CET, MaBiS T+7 wd; provider SLA = "deliver or NACK within 30 min", monitored via webhook.
- **Immutable audit** – original JSON, provider ticket-ID, and final EDIFACT hash are stored as BYTEA in PostgreSQL for ≥ 10 years; PII encrypted at rest.
- **Failure isolation** – if provider API is down, messages queue and retry with exponential back-off; no service-to-service DB writes are blocked

Technical Description

1. **Event intake** – contract.activated, energy.bkv.schedule.ready, meter.mabis.dataset.ready land on Kafka; Change-Processor stores them in gateway.message_prepared.
2. **Outbox commit** – same TX inserts into gateway.outbox; Debezium sends gateway.message.prepared topic (traceable).
3. **Send worker** – picks rows in PENDING, POSTs JSON to the **clearing-house REST endpoint**; stores returned provider_ticket_id, sets status SENT.
4. **Webhook listener / poller** – receives provider callbacks (DELIVERED, NACK, VALIDATION_ERROR), updates gateway.message.status and emits gateway.message.acked Kafka event.
5. **Internal consumers** – Contract Service updates switch progress, Energy-Mgmt records BKV acceptance, Ops UI shows live status—all by subscribing to the ACK topic.
6. **Observability & SLOs** – Prometheus tracks queued > 30 min or nack_rate > 1 %; alert pushes to PagerDuty. Postgres WAL replication gives RPO = 0, RTO ≤ 30 min.

Market Communication Gateway Service Architecture



```
Messaging / Notification Service
▲ GraphQL mutations – Ops UI / Customer-App (“sendTestMail”, “notifyCustomer”)
▲ Kafka events   (ContractActivated, SwitchConfirmed, InvoiceCreated, PaymentOverdue,
                 OpenPositionAlert, ProfileUpdated … coming from Contract, Billing,
                 Customer-Mgmt, Energy-Mgmt)

────────────────────────────────────────────────────────────────────────────────────────
API Layer (GraphQL  +  REST webhook callback receiver)
────────────────────────────────────────────────────────────────────────────────────────
             |
Ingestion Processor  (Kafka Consumer group)
 ├─ Upsert Recipient Profile → recipientCache  {customer_id, email, phone, locale}
 ├─ Enqueue Notification     → message_queue table
 └─ Deduplicate idempotency  (msg_key UNIQUE {event_id, channel})
────────────────────────────────────────────────────────────────────────────────────────
Notification Orchestrator  (Temporal workflows)
 ├─ RenderTemplateActivity   → Template Renderer  (uses freemarker + locale)
 ├─ SendEmailActivity        → Mail Adapter  (SendGrid API)
 ├─ SendAppActivity          → APP Adapter   (Twilio API)
 ├─ SendPushActivity (in App)        → Push Adapter  (FCM / APNS)
 └─ RetryPolicy: exponential, max 5 attempts
────────────────────────────────────────────────────────────────────────────────────────
Channel Adapters
  • Email    – REST / OAuth2 to SendGrid
  • APPS     – REST to Firebase Cloud Messaging (FCM)
  • Push (in App)   – gRPC to mobile-app gateway
Each adapter commits result to outbox table → Debezium → kafka.notification.sent
────────────────────────────────────────────────────────────────────────────────────────
Persist Results → Postgres  (schema: notification)
────────────────────────────────────────────────────────────────────────────────────────
Tables
  message_queue       : msg_id PK, template_id, customer_id, channel, status, payload_json
  template_catalog    : template_id PK, channel, locale, subject, body_html, body_txt
  send_log            : log_id PK, msg_id FK, channel, attempt_no, result_code, duration_ms
  recipient_pref      : customer_id PK, email, phone, push_token, opt_out_flags
  outbox              : row_id PK, topic, key, payload     -- for exactly-once Kafka publish
────────────────────────────────────────────────────────────────────────────────────────
Outgoing Kafka topics
  kafka.notification.sent     {msg_id} status=SUCCESS/FAIL, channel, timestamp
  kafka.notification.bounce   {msg_id} reason
────────────────────────────────────────────────────────────────────────────────────────
Read-through cache: Redis recipientCache (TTL 1 d) | Alerting: Prometheus on send_log
```

Market Communication data schema

Table	Primary key	Main columns*	Purpose / notes
recipient_pref	customer_id (UUID)	email, phone, push_token, locale (de-DE,en-US…), opt_out_email BOOL, opt_out_sms BOOL, opt_out_push BOOL, updated_ts	One row per customer; refreshed on every customer.profile.updated event and cached in Redis.
template_catalog	template_id (UUID)	channel (EMAIL,SMS,PUSH,SLACK), locale, version_no, subject, body_html, body_txt, valid_from, valid_to, created_ts	Versioned message templates; hot-reloaded by the renderer.
message_queue	msg_id (UUID)	✓customer_id, ✓template_id, channel, status	Work-queue row inserted by Intake

		(PENDING,SENT,DELIVERED,BOUNCED,FAILED), payload_json (merged data model), scheduled_ts, attempts, next_retry_ts, provider_msg_id, created_ts	Processor or API; Orchestrator picks PENDING rows.
send_log	log_id (UUID)	✓msg_id, attempt_no, channel, provider_msg_id, result_code (202,4xx,5xx), http_status, duration_ms, sent_ts	Immutable per-attempt telemetry; drives Prometheus metrics and SLA reports.
outbox	row_id (bigserial)	aggregate_id (= msg_id), topic, key, payload_json, created_ts, processed BOOL DEFAULT false	Exactly-once bridge: same TX as message_queue/send_log; Debezium streams rows to Kafka (notification.sent, notification.bounce).