- **Envelope: CloudEvents** (mandatory).
- **Interface & docs: AsyncAPI** (mandatory).
- **Payload schema & evolution: Avro** in a **Schema Registry** (Kafka is possible or do we care and use PeeGeeQ and **EventCatalog**).
- **Catalog: EventCatalog** (eventcatalog.dev) and/or **Backstage** with an AsyncAPI plugin, generated from source control on CI.
- **Tracing & governance:** W3C **traceparent**, **correlation/causation IDs**, automated **compatibility checks** in CI, and a hard **versioning/deprecation policy**.
- **Bi-temporal:** model **validTime** and **systemTime** in your payload metadata and storage; treat corrections as new events, never edits.

In any case a good place to start is CLoudEvents (which in any case has been a recommended CTO pattern).

1) Event description “standard”

Use **CloudEvents** as the *envelope* for cross-team interoperability. It gives us a stable set of headers (id, source, type, subject, time, datacontenttype, dataschema) and has first-class Java SDKs and bindings (HTTP, Kafka, NATS, AMQP).

Add these fields consistently:

- `traceparent` (consider W3C Trace Context) → end-to-end tracing
- `correlationId` and `causationId` → saga debugging
- `schemaVersion` (payload version)
- `partitionKey` (explicit, as we know don’t just rely on topic defaults)
- **Bi-temporal:** `validTime` (business effective time) and rely on the event’s `time` (or an explicit `recordedTime`) as system time; if we need nanosecond ordering add a `sequence` per aggregate.

Naming: `com.stm..InvoicePaid.v1` for `type` . Keep `v` only when we break compatibility.

2) Interface definition & discoverability

Use **AsyncAPI** (YAML) to define channels/topics, message schemas (referencing payload schemas), bindings (Kafka, HTTP, MQTT), and security. This is the contract we review at design time and publish to your catalog.

- Each **bounded context** gets its own AsyncAPI file (or monorepo folder).
- Reference payload schemas by URL/registry id, not inline JSON blobs.
- Generate docs and client stubs from AsyncAPI to keep producers honest.

3) Payload schema: pick one and govern it

Use **Avro** (common in Kafka land) or **Protobuf** (great tooling/perf). JSON Schema is okay for REST/web, but we’ll regret it at high throughput.

- Stand up a **Schema Registry**. Enforce **backward compatibility** by default.
- CI rule: **no merge** unless your change passes registry compatibility checks.
- Avoid “map<string, any>” in your core events. That’s how catalogues rot over time.

Evolution rules that work:

- Add optional fields → OK.
- Remove/rename/repurpose fields → **New version** (v2 type + new subject/topic).
- Enum widening → OK; narrowing → **breaks**.
- Never change semantics without changing the event type name.

4) Event Catalog (make it self-maintaining)

Don’t make a wiki. It will die.

- Keep **AsyncAPI** + payload schemas **in the same repo** as the producer code (or a contracts repo per domain).
- On CI, **generate** a browsable site: **EventCatalog** (simple, purpose-built) or **Backstage** (heavier, more flexible).
- Every merge publishes the updated catalog; every artifact links back to source and schemas.
- Add **usage analytics** (who consumes which event) by scraping consumer configs or registry references.

Minimum catalog content per event:

- Human description, invariants, example payloads (real redacted samples).
- Producer service, owning team, escalation channel.
- Retention/compaction policy, expected frequency/volume, partitioning key strategy.
- Version history & deprecation window.

5) CQRS & event shapes

- Separate **Domain Events** (inside a bounded context) and **Integration Events** (published for others). The latter are more stable and usually “flattened”.
- Don’t publish Commands. Commands are API calls into your domain.
- Snapshots are an internal optimization—**don’t** catalog them for integration.

6) Bi-temporal modeling (the pragmatic way)

- Every event has **system time** (when it was recorded). That’s CloudEvents `time` or an explicit `recordedTime` .
- Add **validTime** (business effective time) in the payload metadata.
- Corrections? Emit a **new event** with the corrected validTime and a `supersedes` reference to the prior event id.
- Your **read models** (projections) maintain both a “present as of system time” view and, if needed, a “travel to valid time T” view. That’s a storage concern; don’t push this complexity to every consumer unless they ask for it.

7) Versioning & lifecycle we can actually have a chance of enforcing

- **Type name carries the major version:** `CustomerMoved.v2` .
- Topics/channels include major version for high-blast events.

- **Deprecation policy:** 90 days (or your reality) with dual-publishing (v1 & v2) + weekly reminders to consumers.
- **Contract linting**: run an AsyncAPI linter + custom rules (naming, metadata presence, partition keys) in CI.
- **Breaking changes require a migration plan** in the PR (who's impacted, by when).

8) Runtime concerns we should standardise

- **Idempotency:** deterministic event ids (aggregateId + sequence) or store-and-forward with outbox; consumers store processed ids.
- **Partitioning:** pick a stable business key. No key → no ordering → pain.
- **PII:** payload classification + field-level encryption or tokenization. Catalogue must flag PII-bearing events. We don't have it I think but it's a good practice and going to be part of GRAS definitely
- **Retention:** compact by key for state-like streams; time-based for audit streams. Document it in the catalog.

9) Minimal Java reference pattern

Publish CloudEvents to Kafka with Avro payload

```
// build.gradle: cloudevents-core, cloudevents-kafka, kafka-clients, avro, your registry serializer

CloudEvent event = CloudEventBuilder.v1()
    .withId(UUID.randomUUID().toString())
    .withSource(URI.create("urn:myco:billing:invoicing-service"))
    .withType("com.myco.billing.InvoicePaid.v1")
    .withSubject(invoiceId)
    .withTime(OffsetDateTime.now())
    .withExtension("traceparent", traceContext.getTraceparent())
    .withExtension("correlationid", correlationId)
    .withExtension("validtime", validTime.toString())
    .withExtension("schemaversion", "1")
    .withData("application/avro", avroBytes) // payload already serialized
    .build();

ProducerRecord<String, byte[]> record =
    KafkaMessageFactory.createWriter("billing.invoice-paid.v1").writeBinary(event, invoiceId);

producer.send(record);
```

Avro schema snippet (payload only)

```
{
  "type":"record","name":"InvoicePaid","namespace":"com.myco.billing",
  "fields":[
    {"name":"invoiceId","type":"string"},
    {"name":"amount","type":{"type":"bytes","logicalType":"decimal","precision":18,"scale":2}},
    {"name":"currency","type":"string"},
    {"name":"customerId","type":"string"},
    {"name":"validTime","type":{"type":"long","logicalType":"timestamp-millis"}},
    {"name":"supersedesEventId","type":["null","string"],"default":null}
  ]
}
```

10) Governance & automation (don’t skip this)

- **Pre-commit hooks**: validate AsyncAPI and schema references.
- **CI jobs**:
    - AsyncAPI lint + render docs → publish to catalog site.
    - Schema compatibility check against Registry (fail on break).
    - Contract impact report (which consumers subscribe to this topic?).
- **Runtime policy**: reject events missing required CloudEvents extensions via a stream gatekeeper (e.g., a Kafka Streams processor or a sidecar).

11) Anti-patterns (we’ll pay for these later)

- Free-form JSON events with “flexible” fields. That’s a schema, just undocumented.
- Stuffing bi-temporal logic into *every* consumer. Keep it in read models.
- Reusing the same event type across bounded contexts (“Enterprise Event”). No.
- Publishing command-shaped events like `CreateOrder` . Use an API for commands.
- Versioning by silently changing payloads without changing the type. Consumers will hate us .
- A Confluence page as “the catalogue”. It will be outdated next quarter.

What to do plan for

1. Pick **CloudEvents + AsyncAPI + Avro/Protobuf + Schema Registry**.
2. Create a **contracts repo** with one sample event, AsyncAPI, and CI to generate an **Event Catalog** site.
3. Add **lint & compatibility checks** to producer pipelines.
4. Define **versioning + deprecation** policy in writing.
5. Add **traceparent, correlationId, causationId, validTime** to your envelope conventions.
6. Retrofit one high-value domain first; prove the migration, then scale.

OTC derivatives are exactly where event-driven, CQRS, and bi-temporal event stores are valuable. For a trade-processing pipeline covering **capture** → **validation** → **enrichment** → **lifecycle**. We look at Solace PeeGeeQ, CloudEvents, Avro and Java.

Non-negotiables for this domain

- **CloudEvents envelope** for interoperability; **Avro** payloads in a eventually in a **Schema Registry**.
- **Partition key = tradeId (we didn't talk about UTI / Unique Swap Identifier actually - Archana?).** We need ordering per trade so
- i. UUID v1 (time-based UUID) Part of the official UUID RFC (4122). Embeds a timestamp + node id (MAC) + clock sequence.

2. UUID v7 (proposed / emerging standard) Draft standard in the IETF (successor to UUID v1/v4). Purely time-ordered, with a millisecond timestamp in the high bits, and randomness for uniqueness.

Explicitly designed for modern event sourcing / DB workloads.

Libraries exist in Java now (e.g. com.github.f4b6a3.uuid)..

- **Bi-temporal Two clocks everywhere:** `systemTime` (recorded) and `validTime` (business effective; usually executionTimestamp or lifecycleEffectiveTime).
- **Capture Facts not states:** publish events like `TradeCaptured` , `TradeValidated` , `TradeEnrichmentApplied` , `TradeLifecycleApplied` . No “isValid=true” mush.
- **Immutability of course as per PeeGeeQ concepts:** corrections are **new events** that **supersede** earlier ones; never edits.
- **Reference-data reproducibility:** include `refDataSnapshotId` / `asOfVersion` in enrichment/lifecycle events?

STM Event taxonomy (we should keep it boring and strict)

STM Capture (transaction and instruction)

- `TradeCaptured.v1` Facts at execution time: instrument, parties, economic terms, `executionTimestamp` , `captureSystem` , raw Trade IDs, fund admin / sales / trader, desk??
- `TradeCaptureCorrected.v1` (does it happen? rare but real) Contains `supersedesEventId` + corrected fields. `validTime` is the executionTimestamp being corrected.

Level 0 Validation

- `TradeValidated.v1` (pass) / `TradeRejected.v1` (fail) Include `rulesRun[]` , `failedRuleCodes[]` , `blocking=true/false` . Rejections route to a **quarantine** topic; only ops can release.

Level 1 Valiration and Enrichment (reference data, static/dynamic)

- `TradeEnrichmentApplied.v1` Adds book/accounting, legal entity identifiers, netting set, clearing eligibility, settlement calendar adjustments, comp curve IDs, etc, etc, `refDataSnapshotId` .
- `TradeEnrichmentSuperseded.v1` when ref data is re-run against the same trade (e.g., Did Archana say this happens in Markit ? corporate action back-dated and so forth).

Lifecycle (post-trade events that change economics/positions)

Normalize ALL of these to one canonical:

- `TradeLifecycleApplied.v1` with `lifecycleType` ∈{ `Amend` , `Terminate` , `IndexFixing` , `Fee` , `Novation` , `Allocation` , `Compression` , `CollateralizationEffect` , `Backload` , `Clear` , `Unclear` , `Exercise` , `Knockout` …} Each carries `lifecycleEffectiveTime` (validTime) + delta payload (what changed) + `sourceSystem` .
- `TradeLifecycleReversed.v1` for operational reversals (rare).
- If we must publish specialized types (e.g., `TradeNovated`), make them **aliases** of the canonical with a stricter schema.

Cross-cutting, e.g. iQube events?

- `ReportGenerated.v1` (regulatory/confirmation artifacts with hashes, not the doc)
- `PositionProjected.v1` (if we share projections—usually internal only)
- `OpsInstructionIssued.v1` (settlement/collateral calls kicked off)

Topics & retention (sometimes called the two-stream pattern)

- **Audit stream (append-only):** `trades.events.v1` All events, infinite(ish) retention. Source of truth for replay, forensics.
- **State stream (compacted):** one per aggregate flavor:
  - `trades.by-id.v1` (compacted; last known snapshot per trade)
  - Optional: `positions.by-book.v1` , `exposure.by-counterparty.v1` (materialized via streams/jobs)
- **Quarantine:** `trades.rejected.v1` (time-retained; ops tooling subscribes)

Why both? Audit supports **time travel** and bi-temporal queries; compacted topics give us fast warm starts and cheap read models. Advanced feature and Lusic does this as standard pattern.

Bi-temporal handling that won’t kill us

- Put `validTime` in the payload metadata; `systemTime` is the CloudEvents `time` (and/or `recordedTime` extension).
- **Corrections**: new event with same `validTime` , later `systemTime` , `supersedesEventId` .
- Read models store a **timeline** per trade: a log ordered by `systemTime` but queryable by `validTime` .
- For positions/P&L, maintain:
  - **As-of system time** views (what ops saw at T).
  - **As-at valid time** views (economic reality on trade date). Use windowed stores to re-project when late events arrive.

Reference data discipline

Include on enrichment/lifecycle:

- `refDataSnapshotId` (monotonic ID from your refdata service)
- `curveSetId` , `calendarVersion` , `legalEntityVersion` , etcv Consumers can re-price deterministically. If these change ex-post, publish a new `TradeEnrichmentApplied` **with same validTime** but higher `systemTime` .

Governance & interoperability details

- **CloudEvents extensions** we should standardize:
  - `traceparent` , `correlationId` , `causationId`
  - `partitionKey` (tradeId until UTI/USI minted; then switch—dual-publish for a period)

    - `schemaVersion` , `recordedTime` if we want it explicit
    - `supersedesEventId` where applicable
- **Versioning**: break the payload → bump the **event type** (`.v2`) and the **topic** (`…v2`). Dual-publish during a fixed deprecation window.
- **PII/reg data**: mask or field-encrypt CP names in the **public** integration events; keep full details in internal-only streams. Catalog must flag PII.

CQRS/read models we will need

- **Trade State** (per trade): last capture + validations + cumulative enrichments + lifecycle projections → forms the golden trade JSON used by downstreams.
- **Positions** (per book/CCY/product): can be materialized from lifecycle deltas; compacted plus periodic checkpoints. Lusid works like this actually.
- **Custody Obligations** (settlement schedule): synthesized from trade state + calendars + lifecycle events.
- **Reg Reporting Feeds** (EMIR/UK EMIR/CFTC/MiFIR): derive reportable fields with lineage back to event ids; emit `ReportGenerated` with hash + regulator ack ids.

In PeeGeeQ each could be a **separate projection** with its own store and re-projection mechanism.

PeeGeeQ Idempotency, ordering, and replay

- **Event id** = `${tradeId}:${sequence}` (sequence is a monotonic int per trade).
- Producers enforce one-at-least with the **outbox**; consumers store processed ids per partition.
- Late/out-of-order: keep a **grace window** (e.g., 48h) and a **delta compactor** that can re-order within a trade's stream using sequence + validTime.

Error flows to iQube, STM-Captue and STM Event Store (PeeGeeQ)

- Validation errors → `TradeRejected` to quarantine with `failedRuleCodes` .
- Enrichment faults (e.g., missing LEI) → either `TradeRejected` (blocking) or `TradeEnrichmentApplied` with `qualityFlags` so consumers can decide.
- Poison pills → send the raw event to `trades.deadletter.v1` with error metadata and the original headers.

Event Catalog structure (AsyncAPI + schemas)

- Repos by bounded context: `trade-capture-contracts` , `trade-validation-contracts` , `trade-lifecycle-contracts` .
- Each has:

  - `/asyncapi/trades-events.yaml` (channels `trades.events.v1` , …)
  - `/schemas/TradeCaptured.avsc` , `/schemas/TradeLifecycleApplied.avsc`
  - CI: validate AsyncAPI, check schema compatibility, generate **EventCatalog** site, publish.
- Catalog entries must show: **example payloads** (use real redacted events), **partitioning**, **retention/compaction**, **owners**, **SLA**, **PII flags**, **version history**, and **downstream consumers**.

Minimal schemas focus on KISS for the POC (Avro snippets)

TradeCaptured

```
{
 "type":"record","name":"TradeCaptured","namespace":"com.acme.trade",
 "fields":[
  {"name":"tradeId","type":"string"},
  {"name":"executionTimestamp","type":{"type":"long","logicalType":"timestamp-millis"}},
  {"name":"productType","type":"string"},        // e.g., IRS, CDS, NDF
  {"name":"economic","type":{
    "type":"record","name":"EconomicTerms","fields":[
      {"name":"notional","type":"double"},
      {"name":"currency","type":"string"},
      {"name":"payLeg","type":["null",{"type":"record","name":"PayLeg","fields":[
        {"name":"fixedRate","type":["null","double"],"default":null},
        {"name":"floatingIndex","type":["null","string"],"default":null}
      ]}],"default":null}
    ]
  }},
  {"name":"parties","type":{"type":"record","name":"Parties","fields":[
    {"name":"partyA","type":"string"},
    {"name":"partyB","type":"string"}
  ]}},
  {"name":"salesDesk","type":"string"},
  {"name":"captureSystem","type":"string"},
  {"name":"validTime","type":{"type":"long","logicalType":"timestamp-millis"}}, // usually = executionTimestamp
  {"name":"raw","type":["null","bytes"],"default":null}
 ]
}
```

TradeLifecycleApplied

```
{
 "type":"record","name":"TradeLifecycleApplied","namespace":"com.acme.trade",
 "fields":[
  {"name":"tradeId","type":"string"},
  {"name":"lifecycleType","type":{"type":"enum","name":"LifecycleType",
    "symbols":["Amend","Terminate","IndexFixing","Fee","Novation","Allocation","Compression","Clear","Unclear","Exercise"
  {"name":"delta","type":{"type":"map","values":["null","string","double","long","boolean"]}},
  {"name":"lifecycleEffectiveTime","type":{"type":"long","logicalType":"timestamp-millis"}},
  {"name":"refDataSnapshotId","type":"string"},
  {"name":"supersedesEventId","type":["null","string"],"default":null},
  {"name":"validTime","type":{"type":"long","logicalType":"timestamp-millis"}}
 ]
}
```

CloudEvents envelope (Java send)

```
CloudEvent evt = CloudEventBuilder.v1()
  .withId(tradeId + ":" + sequence)
  .withSource(URI.create("urn:acme:fo:trade-capture"))
  .withType("com.acme.trade.TradeCaptured.v1")
  .withSubject(tradeId)
  .withTime(OffsetDateTime.now()) // systemTime
  .withExtension("traceparent", traceCtx.getTraceparent())
  .withExtension("correlationid", correlationId)
  .withExtension("partitionkey", tradeId)
  .withExtension("schemaversion", "1")
  .withExtension("validtime", executionTime.toString())
  .withData("application/avro", avroBytes)
  .build();
```

Kafka Streams pattern for bi-temporal state (Java)

- **KStream** from `trades.events.v1` → groupBy `tradeId` → aggregate to a **timeline store** keyed by `tradeId` with a list ordered by `(systemTime, sequence)`.
- Build two KTables:
  - **As-of (system time)**: last event by `systemTime` → compacted state.
  - **As-at (valid time)**: custom query that binary-searches timeline by `validTime`.

Sketch:

```
KStream<String, TradeEvent> events = builder.stream("trades.events.v1", Consumed.with(Serdes.String(), tradeEventSerde));

KTable<String, TradeTimeline> timeline = events
  .groupByKey()
  .aggregate(TradeTimeline::empty,
    (tradeId, evt, agg) -> agg.add(evt), // keeps ordered by systemTime; handle supersedes
    Materialized.<String, TradeTimeline, KeyValueStore<Bytes, byte[]>>as("trade-timeline-store")
      .withKeySerde(Serdes.String())
      .withValueSerde(timelineSerde));

KTable<String, TradeState> asOf = timeline.mapValues(TradeTimeline::toLatestState);
```

Orchestration vs choreography talked a lot with Amrit last year

- Use **choreography** inside the trade domain (validation/enrichment/lifecycle are decoupled).

- Use **sagas** for cross-domain flows with external acks (clearing, confirmations, regulatory submissions). Persist saga state; publish `…AwaitingAck` / `…AckReceived` events.

Operational realities that we can eventually support: Jim, Nasir,

- **UTI/USI creation**: publish `TradeIdentifierAssigned.v1` when obtained; consumers update keys. Dual-publish using both provisional `tradeId` and final `UTI` during migration window.
- **Backfills**: dedicated replay service reading from audit stream, honoring partitions and throttling, able to slice by `validTime` or `systemTime` .
- **Reconciliation**: nightly job that compares materialized trade state vs. upstream FO blotter and downstream confirmations; emits `ReconciliationDiscrepancyFound.v1` .
- **Latency SLOs**: per stage (capture→validated, validated→enriched, enriched→lifecycle projected). Put these SLOs and current p95 in the catalog.

Security & compliance

- Field-level encryption for CP names/identifiers on integration topics; keys managed by KMS.
- Full payloads stored internally for audit; catalog flags PII and regulatory fields.
- Immutable **audit** + **who** published (service identity) + sig/hash for non-repudiation.