NoTIP

NO TESTS IN PRODUCTION

Specifica tecnica - Data *Consumer*$_G$

Versione	1.1.0
Data Modifica	2026-04-17
Utilizzo	Esterno

Abstract dei contenuti

Specifica tecnica del microservizio notip-data-*consumer*$_G$: architettura interna, value object di dominio, definizione dei port, design di dettaglio degli adapter, schema del database e metodologie di testing.

Changelog$_G$

Versione	Data	Autori	*Verificatore*$_G$	Descrizione
1.1.0	2026-04-17	Francesco Marcon	Leonardo Preo	Aggiornamento contenuti a seguito del colloquio tecnico di PB del 17/04/2026
1.0.0	2026-04-13	Leonardo Preo (responsabile)		Approvazione per ingresso in *baseline*$_G$ PB
0.4.1	2026-04-08	Matteo Mantoan	Francesco Marcon	Uniformazione titoli e intestazioni: solo prima lettera maiuscola
0.4.0	2026-04-08	Francesco Marcon	Valerio Solito	Aggiunta di logging avanzato e monitoraggio per il servizio Data-*Consumer*$_G$ per migliorare la tracciabilità e la diagnosi dei problemi
0.3.0	2026-03-24	Francesco Marcon	Alessandro Contarini	Aggiornamento con le modifiche effettuate al servizio per gestione coerente degli stati del *Gateway*$_G$
0.2.0	2026-03-24	Francesco Marcon	Alessandro Contarini	Integrazione sezione pattern a aggiunta port
0.1.0	2026-03-09	Francesco Marcon	Alessandro Contarini	Aggiunta sezione pattern
0.0.1	2026-02-28	Francesco Marcon	Alessandro Contarini	Creazione del documento prima bozza del documento

Indice

1. Introduzione

Questo documento illustra l'architettura interna e le scelte implementative del microservizio `notip-data-consumer`$_G$. Sviluppato in *Go*$_G$, questo componente opera esclusivamente nel *backend*$_G$ e ha la duplice responsabilità di consumare i flussi telemetrici da *NATS*$_G$ *JetStream*$_G$ per persisterli in batch sotto forma di blob opachi in *TimescaleDB*$_G$, e di tracciare la liveness dei *gateway*$_G$ *IoT*$_G$ tramite un meccanismo di *heartbeat*$_G$ in-memory. Espone un server HTTP con due *endpoint*$_G$: `/metrics` (*Prometheus*$_G$) e `/healthz` (health check con verifica DB).

Per l'esatta struttura dei *payload*$_G$ e i contratti delle interfacce, il codice sorgente costituisce la Single Source of Truth.

2. Dipendenze e configurazione

2.1. Variabili d'ambiente

Tutte le variabili sono caricate all'avvio da `internal/config`. La mancanza di una variabile obbligatoria causa un crash immediato. Il caricatore utilizza un pattern di accumulo dell'errore: la prima lettura fallita invalida tutte le successive senza ulteriori *branch*$_G$.

Campo	Variabile d'ambiente	Default	Obbligatorio
`NATSUrl`	`NATS_URL`	—	Sì
`NATSTlsCa`	`NATS_TLS_CA`	—	Sì
`NATSTlsCert`	`NATS_TLS_CERT`	—	Sì
`NATSTlsKey`	`NATS_TLS_KEY`	—	Sì
`NATSConsumerDurableName`	`NATS_CONSUMER_DURABLE_NAME`	`data-consumer`$_G$`-telemetry`	No
`NATSConnectTimeoutSeconds`	`NATS_CONNECT_TIMEOUT_SECONDS`	`10`	No
`DBHost`	`DB_HOST`	—	Sì
`DBPort`	`DB_PORT`	`5432`	No
`DBName`	`DB_NAME`	—	Sì
`DBUser`	`DB_USER`	—	Sì
`DBPasswordFile`	`DB_PASSWORD_FILE`	—	Sì
`DBMaxConns`	`DB_MAX_CONNS`	`10`	No
`DBMinConns`	`DB_MIN_CONNS`	`2`	No
`DBSSLMode`	`DB_SSL_MODE`	`"require"`	No
`DBSSLRootCert`	`DB_SSL_ROOT_CERT`	`""`	Condizionale[1]
`GatewayBufferSize`	`GATEWAY_BUFFER_SIZE`	`1000`	No
`HeartbeatTickMs`	`HEARTBEAT_TICK_MS`	`10000`	No
`HeartbeatGracePeriodMs`	`HEARTBEAT_GRACE_PERIOD_MS`	`120000`	No
`AlertConfigRefreshMs`	`ALERT_CONFIG_REFRESH_MS`	`30000`	No
`AlertConfigDefaultTimeoutMs`	`ALERT_CONFIG_DEFAULT_TIMEOUT_MS`	`60000`	No
`AlertConfigMaxRetries`	`ALERT_CONFIG_MAX_RETRIES`	`10`	No

[1] Obbligatorio quando `DB_SSL_MODE` è `verify-ca` o `verify-full`

`AlertConfigInitialBackoffMs`	`ALERT_CONFIG_INITIAL_BACKOFF_MS`	`1000`	No
`AlertConfigMaxBackoffMs`	`ALERT_CONFIG_MAX_BACKOFF_MS`	`30000`	No
`MetricsAddr`	`METRICS_ADDR`	`":9090"`	No

`Config` espone il metodo `GetDatabaseDSN() (string, error)`: legge la password dal *Docker*$_G$ secret file indicato da `DBPasswordFile`, rimuove gli spazi/newline finali, e costruisce il DSN nel formato `postgres://user:pass@host:port/dbname?sslmode=<DBSSLMode>`. Quando `DBSSLMode` è `verify-ca` o `verify-full` e `DBSSLRootCert` è non vuoto, aggiunge anche il parametro `sslrootcert=<DBSSLRootCert>` alla query string.

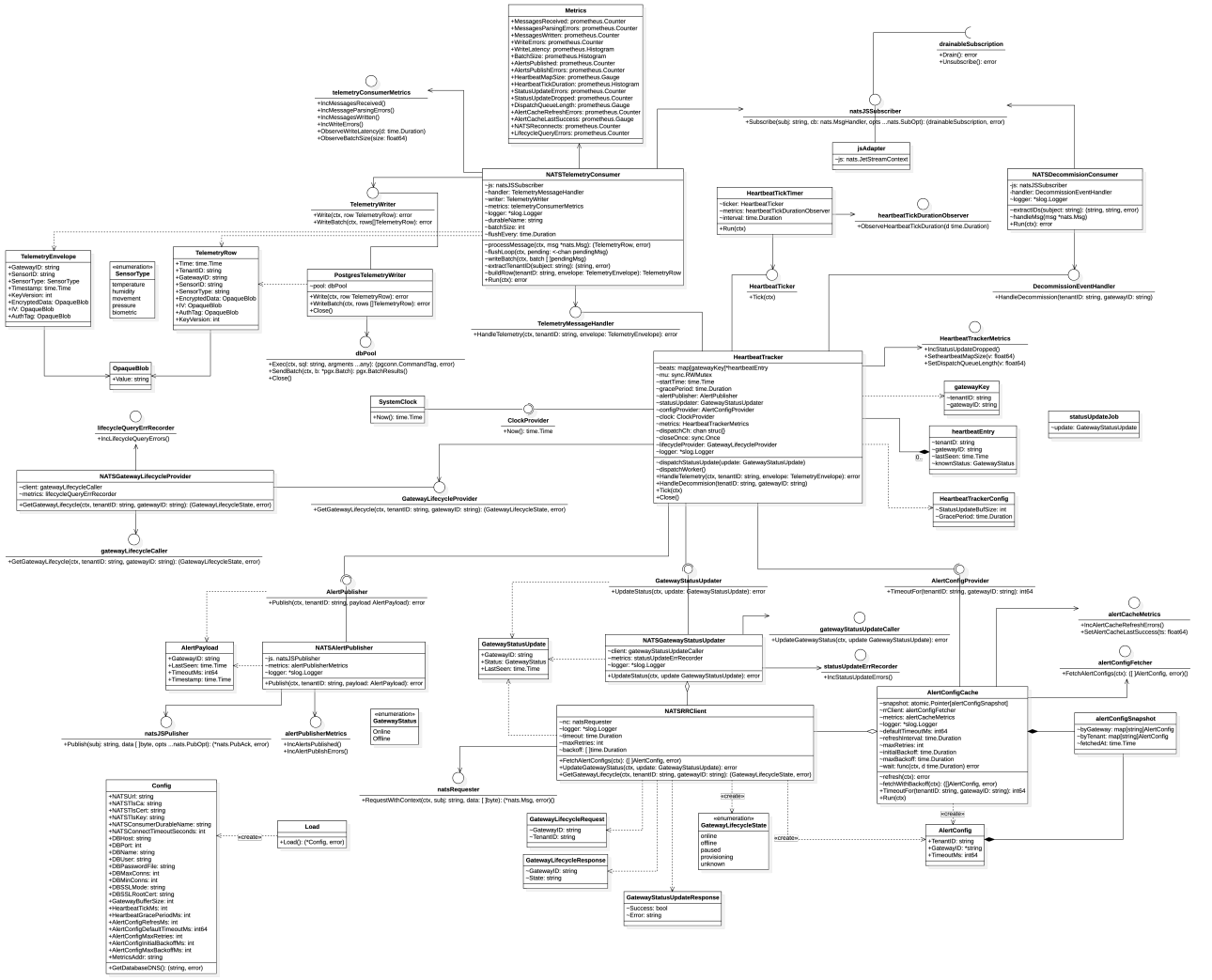
2.2. Sequenza di avvio

I passi bloccanti interrompono il processo in caso di fallimento. Il processo definisce tre costanti compile-time: `natsRRTimeout = 5s`, `telemetryBatchSize = 100`, `telemetryFlushInterval = 500ms`.

Step	Componente	Azione	Bloccante?
0	`slog`	Imposta *JSON*$_G$ handler su stderr come logger di default	No
1	`config.Load`	Carica e valida la configurazione da env; costruisce il DSN leggendo la password dal *Docker*$_G$ secret file	Sì
2	`Metrics`	Crea le metriche *Prometheus*$_G$ con `prometheus`$_G$`.DefaultRegisterer`	No
3	*NATS*$_G$	Connessione a *NATS*$_G$ con *mTLS*$_G$ (`RootCAs`, `ClientCert`); configura timeout e handler di riconnessione	Sì
4	*JetStream*$_G$	Acquisisce il contesto *JetStream*$_G$ dalla connessione *NATS*$_G$	Sì
5	`pgxpool`	Parse DSN, configura `MaxConns`/`MinConns`, crea il pool *TimescaleDB*$_G$; crea il context di segnale (`SIGTERM`, `SIGINT`)	Sì
5b	`migrations`	Applica le migrazioni SQL al database via `migrations.Apply`	Sì
6	Driven adapter	Istanzia `NATSRRClient`, `AlertConfigCache`, `NATSAlertPublisher`, `NATSGatewayStatusUpdater`, `NATSGatewayLifecycleProvider`, `PostgresTelemetryWriter`, `SystemClock`	No
7	`HeartbeatTracker`	Costruisce il servizio con tutte le dipendenze (incluso `lifecycleProvider`) tramite `HeartbeatTrackerConfig`; avvia la *goroutine*$_G$ `dispatchWorker`	No
8	Driving adapter	Istanzia `HeartbeatTickTimer`, `NATSDecommissionConsumer`, `NATSTelemetryConsumer`	No

9	*Prometheus*$_G$ HTTP	Avvia il server HTTP su `MetricsAddr` in una *goroutine*$_G$ di background; `/healthz` verifica la raggiungibilità del DB via `pool.Ping`	No
10	`AlertConfigCache.Run`	Avvia *goroutine*$_G$: fetch iniziale con backoff, poi loop di refresh periodico; usa i default se tutti i retry falliscono	No
11	`HeartbeatTickTimer`	Avvia *goroutine*$_G$ di background	No
12	`NATSDecommissionConsumer`	Avvia *goroutine*$_G$ di background (errori loggati)	No
13	`NATSTelemetryConsumer.Run`	Avvia il *consumer*$_G$ *JetStream*$_G$ (blocca il *main*$_G$ *goroutine*$_G$)	Sì
—	Signal handler	Su SIGTERM/SIGINT: cancella il root context → *consumer*$_G$ *telemetria*$_G$ si ferma → *consumer*$_G$ decommission si ferma → tick timer si ferma → cache alert si ferma → metrics server shutdown → `tracker.Close()` drena il canale di dispatch → pool chiude → *NATS*$_G$ drain	—

3. Architettura logica



3.1. Pattern architetturale: architettura esagonale

Il servizio adotta l'**architettura esagonale** (Ports & Adapters). La logica di dominio, contenuta interamente in `HeartbeatTracker`, dipende esclusivamente da interfacce (*port*) e mai da implementazioni concrete.

Il «composition root» in `cmd/consumer`$_G$`/main`$_G$`.go`$_G$ istanzia l'intero grafo di dipendenze tramite constructor injection.

3.2. Layout dei package

```
notip-data-consumer_G/
├── cmd/consumer_G/          Composition root, graceful shutdown
├── internal/
│   ├── config/              Config struct, loader da env, GetDatabaseDSN
│   ├── domain/
│   │   ├── model/           Value object puri (nessuna import infrastrutturale)
│   │   └── port/            Definizioni di tutti i driving/driven port
│   ├── service/             HeartbeatTracker
│   ├── adapter/
│   │   ├── driven/          PostgresTelemetryWriter, NATSAlertPublisher,
│   │   │                    NATSGatewayStatusUpdater, NATSGatewayLifecycleProvider,
│   │   │                    AlertConfigCache, NATSRRClient, SystemClock
```

```
│   │   └── driving/        NATSTelemetryConsumer, NATSDecommissionConsumer,
│   │                       HeartbeatTickTimer
│   └── metrics/            Prometheus_G metric handles e narrow-interface methods
├── tests/
│   └── integration/        Test di integrazione Testcontainers_G (NATS_G mTLS_G, TimescaleDB_G)
└── migrations/             SQL migrations TimescaleDB_G
```

3.3. Strati architetturali

Strato	Package	Contenuto

Dominio	`internal/domain/model`
	`internal/domain/port`
	`internal/service`
		Value object puri, definizioni dei port, `HeartbeatTracker`. Nessun package di questo strato contiene importazioni infrastrutturali.

Driving Adapter	`internal/adapter/driving`	Traduce eventi esterni in chiamate ai driving port. Tre adapter: messaggi *NATS*$_G$ telemetrici (con batch-processing), eventi di decommission *NATS*$_G$, tick periodico del timer.
Driven Adapter	`internal/adapter/driven`	Implementazioni dei driven port verso risorse esterne: *TimescaleDB*$_G$, *NATS*$_G$ *JetStream*$_G$, Management API (via *NATS*$_G$ RR), clock di sistema.

4. Definizione dei port

4.1. Driven port

Interfacce invocate dagli adapter (o dal dominio), implementate dagli adapter verso risorse esterne.

TelemetryWriter

DRIVEN PORT

Confine di persistenza per i record telemetrici. Isola il dominio dalla tecnologia di storage, consentendo la sostituzione del *backend*$_G$ senza impatto sulla logica applicativa.

Metodo	Responsabilità
`Write`	Persiste un singolo record telemetrico opaco
`WriteBatch`	Persiste un batch di record in un'unica operazione di rete

AlertPublisher

DRIVEN PORT

Confine di pubblicazione degli alert infrastrutturali. Disaccoppia la logica di rilevazione offline dal meccanismo di distribuzione (*JetStream*$_G$).

Metodo	Responsabilità
`Publish`	Pubblica un alert *gateway*$_G$-offline per un *tenant*$_G$ specifico

GatewayStatusUpdater

DRIVEN PORT

Confine di notifica delle transizioni di stato. Isola il dominio dal protocollo *NATS*$_G$ Request-Reply verso il Management API.

Metodo	Responsabilità
`UpdateStatus`	Notifica una transizione di stato online/offline al Management API

AlertConfigProvider

DRIVEN PORT

Confine di accesso alla configurazione degli alert. Permette al dominio di richiedere il timeout senza conoscere la sorgente. Lookup: override *gateway*$_G$ → default *tenant*$_G$ → default di sistema.

Metodo	Responsabilità
`TimeoutFor`	Restituisce il timeout offline configurato per uno specifico *gateway*$_G$ (ms)

GatewayLifecycleProvider

DRIVEN PORT

Interrogato da `HeartbeatTracker.Tick` immediatamente prima di emettere un alert offline, assicurandosi dello stato amministrativo del *gateway*$_G$. In caso di errore, il chiamante deve procedere *fail-open* (emettere comunque l'alert) per evitare di mascherare eventi offline reali quando il Management API non è raggiungibile.

Metodo	Responsabilità
`GetGatewayLifecycle`	Restituisce lo stato amministrativo corrente di un *gateway*$_G$

ClockProvider

DRIVEN PORT

Astrazione del clock di sistema. Consente l'iniezione di un clock deterministico nei test, eliminando la dipendenza diretta da `time.Now()` nella logica di dominio e di servizio.

Metodo	Responsabilità
`Now`	Restituisce il timestamp corrente

4.2. Driving port

Interfacce implementate dal dominio, invocate dagli adapter per immettere eventi nel sistema.

TelemetryMessageHandler

DRIVING PORT

Punto di ingresso per gli eventi telemetrici decodificati. L'adapter *NATS*$_G$ invoca questo port dopo aver estratto il `tenantID` dal subject e deserializzato l'envelope.

Metodo	Responsabilità
`HandleTelemetry`	Aggiorna l'*heartbeat*$_G$ del *gateway*$_G$ e gestisce le transizioni di stato (prima comparsa, recovery da offline)

DecommissionEventHandler **DRIVING PORT**

Punto di ingresso per gli eventi di decommission *gateway*$_G$. Rimuove il *gateway*$_G$ dalla mappa di *heartbeat*$_G$ per prevenire falsi alert su *gateway*$_G$ dismessi.

Metodo	Responsabilità
`HandleDecommission`	Rimuove un *gateway*$_G$ dalla mappa di liveness

HeartbeatTicker **DRIVING PORT**

Punto di ingresso per il tick periodico di liveness. Invocato dal timer adapter a intervalli configurabili.

Metodo	Responsabilità
`Tick`	Valuta la liveness di tutti i *gateway*$_G$ tracciati e genera alert per quelli offline

5. Design di dettaglio

5.1. Value object del dominio (`internal/domain/model`)

Tutti i tipi sono data struct, senza logica e senza importazioni infrastrutturali.

Tipo	Descrizione e campi
`OpaqueBlob`	Named type con campo `Value string` (blob base64). Il tipo rende visibile nel type system l'invariante della *pipeline*$_G$ opaca: qualunque tentativo di decodifica del contenuto è considerato una violazione di tipo. Espone `MarshalJSON` / `UnmarshalJSON` custom che serializzano `Value` come stringa *JSON*$_G$ plain, preservando il *payload*$_G$ base64 invariato attraverso qualsiasi round-trip *JSON*$_G$.
`TelemetryEnvelope`	Wire format di un messaggio *NATS*$_G$ su `telemetry.data.{tenantId}.{gwId}`. Campi con *JSON*$_G$ tag: `GatewayID string` (`gatewayId`), `SensorID string` (`sensorId`), `SensorType SensorType` (`sensorType`), `Timestamp time.Time` (`timestamp`), `KeyVersion int` (`keyVersion`), `EncryptedData OpaqueBlob` (`encryptedData`), `IV`$_G$ `OpaqueBlob` (`iv`$_G$), `AuthTag OpaqueBlob` (`authTag`). `TenantID` non è nel body *JSON*$_G$: è estratto dal subject *NATS*$_G$ dall'adapter.
`SensorType`	Named string type. Vincola `TelemetryEnvelope.SensorType` ai cinque valori definiti nel contratto AsyncAPI: `SensorTypeTemperature = "temperature"`, `SensorTypeHumidity = "humidity"`, `SensorTypeMovement = "movement"`, `SensorTypePressure = "pressure"`, `SensorTypeBiometric = "biometric"`.
`TelemetryRow`	Record normalizzato scritto su *TimescaleDB*$_G$. Aggiunge `TenantID string` (dal subject *NATS*$_G$) e `Time time.Time` (partition key dell'ipertabella) a `TelemetryEnvelope`. I tre `OpaqueBlob` sono passati invariati.
`AlertPayload`	*Payload*$_G$ pubblicato su `alert.{tenantId}.gw_offline`. Campi con *JSON*$_G$ tag: `GatewayID string` (`gatewayId`), `LastSeen time.Time` (`lastSeen`), `TimeoutMs int64` (`timeoutMs`), `Timestamp time.Time` (`timestamp`).
`AlertConfig`	`TenantID string`; `GatewayID *string` (nil = default *tenant*$_G$); `TimeoutMs int64`.
`GatewayStatusUpdate`	*Payload*$_G$ per la chiamata *NATS*$_G$ RR su `internal.mgmt.gateway`$_G$`.update-status`. Campi con *JSON*$_G$ tag: `GatewayID string` (`gateway_id`), `Status GatewayStatus` (`status`), `LastSeenAt time.Time` (`last_seen_at`).
`GatewayStatus`	Enum: `Online = "online"`, `Offline = "offline"`.
`GatewayStatusUpdateResponse`	Risposta *JSON*$_G$ del Management API per la chiamata RR su `internal.mgmt.gateway`$_G$`.update-status`. Campi

	con *JSON*$_G$ tag: `Success bool` (`success`), `Error string` (`error,omitempty` — presente solo quando `Success` è false).
`GatewayLifecycleState`	Stato amministrativo del *gateway*$_G$ impostato dagli operatori nel Management API. Distinto da `GatewayStatus` (stato runtime osservato). Costanti: `LifecycleOnline = "online"`, `LifecycleOffline = "offline"`, `LifecyclePaused = "paused"` (sopprime gli alert offline), `LifecycleProvisioning = "provisioning`$_G$`"`, `LifecycleUnknown = "unknown"` (locale per errori RR, mai trasmesso).
`GatewayLifecycleRequest`	*Payload*$_G$ *JSON*$_G$ per la chiamata RR su `internal.mgmt.gateway`$_G$`.get-status`. Campi con *JSON*$_G$ tag: `GatewayID string` (`gateway_id`), `TenantID string` (`tenant_id`).
`GatewayLifecycleResponse`	Risposta *JSON*$_G$ del Management API per la chiamata RR su `internal.mgmt.gateway`$_G$`.get-status`. Campi con *JSON*$_G$ tag: `GatewayID string` (`gateway_id`), `State GatewayLifecycleState` (`state`).

5.1.1. Tipi interni al package `service`

Tipo	Descrizione e campi
`gatewayKey`	Chiave composita per la mappa *heartbeat*$_G$. Usa una struct invece di una stringa formattata per evitare allocazioni e ambiguità di collision. Campi unexported: `tenantID string`, `gatewayID string`.
`heartbeatEntry`	Entry per *gateway*$_G$ tracciato nella mappa *heartbeat*$_G$. Tipo unexported, manipolato esclusivamente da `HeartbeatTracker`. Campi: `tenantID string`, `gatewayID string`, `lastSeen time.Time`, `knownStatus GatewayStatus`.
`statusUpdateJob`	Wrapper di `GatewayStatusUpdate` per il canale di dispatch asincrono. Campo: `update GatewayStatusUpdate`.

5.2. HeartbeatTracker (`internal/service`)

Unico servizio di dominio. Implementa i tre driving port `TelemetryMessageHandler`, `DecommissionEventHandler` e `HeartbeatTicker`. Gli status update sono inviati in modo asincrono tramite un canale a capacità limitata processato da una *goroutine*$_G$ worker dedicata, mantenendo `Tick` e `HandleTelemetry` non-bloccanti.

5.2.1. Interfaccia metrica ristretta

`HeartbeatTrackerMetrics` — sottoinsieme delle metriche emesse dal tracker:

Metodo	Ritorno
`IncStatusUpdateDropped()`	—
`SetHeartbeatMapSize(v float64)`	—
`SetDispatchQueueLength(v float64)`	—

5.2.2. Campi

Campo	Tipo	Note
`clock`	`ClockProvider`	Driven port iniettato
`alertPublisher`	`AlertPublisher`	Driven port iniettato
`statusUpdater`	`GatewayStatusUpdater`	Driven port iniettato
`configProvider`	`AlertConfigProvider`	Driven port iniettato
`lifecycleProvider`	`GatewayLifecycleProvider`	Driven port iniettato; interrogato in `Tick` prima di ogni alert offline
`metrics`	`HeartbeatTrackerMetrics`	Driven port iniettato
`logger`	`*slog.Logger`	
`startTime`	`time.Time`	Registrato alla costruzione via `clock.Now()`
`gracePeriod`	`time.Duration`	Derivata da `HeartbeatGracePeriodMs`; soppressione alert post-avvio
`mu`	`sync.RWMutex`	Read-lock per snapshot, write-lock per mutazione di `beats`
`beats`	`map[gatewayKey]*heartbeatEntry`	Chiave struct composita `{tenantID, gatewayID}`
`dispatchCh`	`chan statusUpdateJob`	Coda asincrona bounded per status update
`done`	`chan struct{}`	Chiuso alla terminazione di `dispatchWorker`
`closeOnce`	`sync.Once`	Garantisce che `Close()` esegua una sola volta

5.2.3. HeartbeatTrackerConfig

Struct di configurazione che raggruppa i parametri scalari per mantenere la firma del costruttore entro i limiti del linter.

Campo	Tipo	Note
`StatusUpdateBufSize`	`int`	Capacità del canale asincrono di dispatch degli status update
`GracePeriod`	`time.Duration`	Soppressione alert offline per questa durata dopo l'avvio

5.2.4. Costruttore

```
NewHeartbeatTracker(
  clock             ClockProvider,
  alertPublisher    AlertPublisher,
  statusUpdater     GatewayStatusUpdater,
  configProvider    AlertConfigProvider,
  lifecycleProvider GatewayLifecycleProvider,
  metrics           HeartbeatTrackerMetrics,
  cfg               HeartbeatTrackerConfig,
) *HeartbeatTracker
```

Inizializza la mappa *heartbeat*$_G$ vuota, crea il canale di dispatch con capacità `cfg.StatusUpdateBufSize` e avvia la *goroutine*$_G$ `dispatchWorker`.

5.2.5. Metodi pubblici

Metodo	Firma	Implementa
`HandleTelemetry`	`(ctx context.Context, tenantID string, envelope TelemetryEnvelope) error`	`TelemetryMessageHandler`
`HandleDecommission`	`(tenantID string, gatewayID string)`	`DecommissionEventHandler`
`Tick`	`(ctx context.Context)`	`HeartbeatTicker`
`Close`	`()`	—

`HandleTelemetry`

1. Deriva la chiave composita `gatewayKey{tenantID, envelope.GatewayID}`.
2. Acquisisce write-lock.
3. Se l'entry **non esiste**: crea una nuova entry con `lastSeen = clock.Now()` e `knownStatus = Online`; aggiorna la metrica della dimensione della mappa; esegue dispatch di un status update `Online`; ritorna.
4. Se l'entry **esiste**: aggiorna `lastSeen = clock.Now()`.
5. Se l'entry aveva `knownStatus = Offline`: passa a `Online` e inoltra uno status update `Online`.
6. **Non** scrive su *TimescaleDB*$_G$ — è responsabilità dell'adapter driving.

`HandleDecommission`

Acquisisce write-lock. Cancella l'entry per `gatewayKey{tenantID, gatewayID}`. Aggiorna la metrica della dimensione della mappa. Nessun altro side effect.

`Tick`

Approccio a tre fasi:

1. **Grace period check:** se `clock.Now()` è prima di `startTime + gracePeriod`, ritorna immediatamente.
2. **Fase 1 — RLock snapshot:** acquisisce read-lock, copia tutte le entry in uno slice locale (value copy), rilascia read-lock.
3. **Fase 2 — I/O fuori dal lock:** per ogni entry nello snapshot: se `knownStatus == Offline` salta; recupera il timeout via `configProvider.TimeoutFor`; se non scaduto salta; **lifecycle gate**: chiama `lifecycleProvider.GetGatewayLifecycle` — se lo stato è `LifecyclePaused` salta (nessun alert, nessun status update); in caso di errore procede *fail-open* (logga `slog.Warn` e continua) per evitare di mascherare offline reali quando il Management API non è raggiungibile; pubblica alert via `alertPublisher.Publish`; esegue dispatch `Offline` via il canale asincrono.
4. **Fase 3 — WLock**: acquisisce write-lock; rilegge l'entry reale dalla mappa; imposta `knownStatus = Offline` incondizionatamente. La re-validazione su lastSeen è stata rimossa: confrontare con lo snapshot lasciava `knownStatus = Online` mentre il sistema esterno era già stato notificato Offline, causando la mancata rilevazione dell'evento offline successivo. Se una *telemetria*$_G$ è arrivata durante la Fase 2, il prossimo `HandleTelemetry` rileva la transizione`Offline→Online` e gestisce il recovery

`Close`

Chiude `dispatchCh` (idempotente via `sync.Once`). Attende che `dispatchWorker` termini di drenare la coda e chiuda `done`.

5.2.6. Metodi privati

Metodo	Comportamento
`dispatchStatusUpdate(update GatewayStatusUpdate)`	Invio non-bloccante su `dispatchCh`; se il canale è pieno, scarta e incrementa `StatusUpdateDropped`.
`dispatchWorker()`	*Goroutine*$_G$ di background; processa serialmente i `statusUpdateJob` da `dispatchCh`; chiude `done` alla terminazione.

5.3. Adapter driven (`internal/adapter/driven`)

Ogni adapter definisce un'interfaccia metrica ristretta con i soli metodi che effettivamente emette. La struct `Metrics` soddisfa tutte queste interfacce.

5.3.1. PostgresTelemetryWriter

Implementa `TelemetryWriter`. Scrive `TelemetryRow` verbatim su *TimescaleDB*$_G$ — i tre `OpaqueBlob` (`EncryptedData`, `IV`$_G$, `AuthTag`) sono scritti as-is, senza alcuna decodifica. Dipende dall'interfaccia `dbPool` (soddisfatta da `*pgxpool.Pool`).

Campo	Tipo
`pool`	`dbPool`

Interfaccia `dbPool`:

Metodo	Firma
`Exec`	`(ctx context.Context, sql string, arguments ...any) (pgconn.CommandTag, error)`
`SendBatch`	`(ctx context.Context, b *pgx.Batch) pgx.BatchResults`
`Close`	`()`

Metodo	Firma	Note
`Write`	`(ctx context.Context, row TelemetryRow) error`	Singolo INSERT via `pool.Exec`
`WriteBatch`	`(ctx context.Context, rows []TelemetryRow) error`	Costruisce un `pgx.Batch` con un INSERT per riga; singolo round-trip via `pool.SendBatch`; fallisce al primo errore; no-op su slice vuoto
`Close`	`()`	Rilascia tutte le connessioni del pool

5.3.2. NATSAlertPublisher

Implementa `AlertPublisher`. Serializza `AlertPayload` in *JSON*$_G$ e pubblica su `alert.{tenantId}.gw_offline` via *JetStream*$_G$. Il subject è assemblato al momento della pubblicazione. Dipende dall'interfaccia `natsJSPublisher` (soddisfatta da `nats`$_G$`.JetStreamContext`).

Campo	Tipo
`js`	`natsJSPublisher`
`metrics`	`alertPublisherMetrics`
`logger`	`*slog.Logger`

Interfaccia `natsJSPublisher`:

Metodo	Firma
Publish	(subj string, data []byte, opts ...nats$_G$.PubOpt) (*nats$_G$.PubAck, error)

Interfaccia `alertPublisherMetrics`:

Metodo	Ritorno
IncAlertsPublished()	—
IncAlertPublishErrors()	—

Metodo	Firma
Publish	(ctx context.Context, tenantID string, payload$_G$ AlertPayload) error

Il context è accettato per compliance con l'interfaccia ma non propagato a `js.Publish`, la cui API sincrona non supporta cancellazione per-chiamata.

5.3.3. NATSGatewayStatusUpdater

Implementa `GatewayStatusUpdater`. Delega la meccanica *NATS*$_G$ RR all'interfaccia `gatewayStatusUpdateCaller` (soddisfatta da `NATSRRClient`). Incrementa la metrica di errore sui fallimenti.

Interfaccia `gatewayStatusUpdateCaller`:

Metodo	Firma
UpdateGatewayStatus	(ctx context.Context, update GatewayStatusUpdate) error

Interfaccia `statusUpdateErrRecorder`:

Metodo	Ritorno
IncStatusUpdateErrors()	—

Campo	Tipo
client	gatewayStatusUpdateCaller
metrics	statusUpdateErrRecorder
logger	*slog.Logger

Metodo	Firma
UpdateStatus	(ctx context.Context, update GatewayStatusUpdate) error

5.3.4. AlertConfigCache

Implementa `AlertConfigProvider`. Mantiene uno snapshot atomicamente sostituibile delle configurazioni di alert, aggiornato periodicamente via *NATS*$_G$ RR (`internal.mgmt.alert-configs.list`). Le letture in `TimeoutFor` sono lock-free tramite `atomic.Pointer[alertConfigSnapshot]`. Il costruttore inizializza con uno snapshot vuoto, rendendo `TimeoutFor` sicuro da chiamare immediatamente.

Lookup in `TimeoutFor`: override *gateway*$_G$ → default *tenant*$_G$ → `defaultTimeoutMs`.

Dipende dall'interfaccia `alertConfigFetcher` (soddisfatta da `NATSRRClient`).

Campo	Tipo	Note
`snapshot`	`atomic.Pointer[alertConfigSnapshot]`	Sostituito atomicamente ad ogni refresh
`rrClient`	`alertConfigFetcher`	Fetch configurazioni dal Management API
`metrics`	`alertCacheMetrics`	Contatore errori e timestamp ultimo refresh
`logger`	`*slog.Logger`	
`defaultTimeoutMs`	`int64`	Fallback in assenza di configurazione
`refreshInterval`	`time.Duration`	Default: 2 minuti
`maxRetries`	`int`	Default: 10; con backoff esponenziale
`initialBackoff`	`time.Duration`	Default: 1 s; configurabile via `ALERT_CONFIG_INITIAL_BACKOFF_MS`
`maxBackoff`	`time.Duration`	Cap del backoff; configurabile via `ALERT_CONFIG_MAX_BACKOFF_MS` (default 30 s)
`wait`	`func(ctx context.Context, d time.Duration) error`	Sleep context-aware iniettato alla costruzione; ritorna `ctx.Err()` alla cancellazione

Interfaccia `alertConfigFetcher`:

Metodo	Firma
`FetchAlertConfigs`	`(ctx context.Context) ([]AlertConfig, error)`

Interfaccia `alertCacheMetrics`:

Metodo	Ritorno
`IncAlertCacheRefreshErrors()`	—
`SetAlertCacheLastSuccess(ts float64)`	—

Metodo	Firma	Note
`TimeoutFor`	`(tenantID, gatewayID string) int64`	Lock-free; legge dall'atomic pointer
`Run`	`(ctx context.Context)`	Bloccante; fetch iniziale con backoff (`1s → 30s cap`, max `maxRetries`), poi loop di refresh
`refresh`	`(ctx context.Context) error`	Fetch e swap atomico dello snapshot

`fetchWithBackoff`	`(ctx context.Context) error`	Retry con backoff esponenziale; incrementa metrica ad ogni tentativo fallito
`snapshotCounts`	`() (int, int)`	Ritorna `(len(byGateway), len(byTenant))`; usato per logging dopo ogni refresh riuscito

Snapshot interno `alertConfigSnapshot` (immutabile dopo la costruzione):

- `byGateway map[string]AlertConfig` — chiave: `"tenantID/gatewayID"`
- `byTenant map[string]AlertConfig` — chiave: `tenantID`
- `fetchedAt time.Time`

5.3.5. NATSRRClient

Helper infrastrutturale condiviso (non un port). Incapsula i meccanismi di *NATS*$_G$ Request-Reply (timeout, serializzazione, gestione errori). Aggregato da `AlertConfigCache` (via `alertConfigFetcher`), `NATSGatewayStatusUpdater` (via `gatewayStatusUpdateCaller`) e `NATSGatewayLifecycleProvider` (via `gatewayLifecycleCaller`). Dipende dall'interfaccia `natsRequester` (soddisfatta da `*nats`$_G$`.Conn`).

Campo	Tipo	Note
`nc`	`natsRequester`	
`logger`	`*slog.Logger`	
`timeout`	`time.Duration`	Applicato per-tentativo
`maxRetries`	`int`	Default: 3
`backoff`	`[]time.Duration`	Default: `[1s, 2s, 4s]`
`sleep`	`func(ctx context.Context, d time.Duration) error`	Sleep context-aware; ritorna `ctx.Err()` se cancellato

Interfaccia `natsRequester`:

Metodo	Firma
`RequestWithContext`	`(ctx context.Context, subj string, data []byte) (*nats`$_G$`.Msg, error)`

Metodo	Firma	Note
`FetchAlertConfigs`	`(ctx context.Context) ([]AlertConfig, error)`	Body nil verso `internal.mgmt.alertconfigs.list`; deserializza la risposta *JSON*$_G$
`UpdateGatewayStatus`	`(ctx context.Context, update GatewayStatusUpdate) error`	Serializza l'update in *JSON*$_G$, invia verso `internal.mgmt.gateway`$_G$`.update-status`; deserializza `GatewayStatusUpdateResponse` e ritorna errore se `success` è false

`GetGatewayLifecycle`	`(ctx context.Context, tenantID string, gatewayID string) (GatewayLifecycleState, error)`	Serializza `GatewayLifecycleRequest` in *JSON*$_G$, invia verso `internal.mgmt.gateway`$_G$`.get-status`; deserializza `GatewayLifecycleResponse`; valida la risposta tramite `validateGatewayLifecycleResponse` (verifica che `gateway_id` non sia vuoto, che corrisponda all'ID atteso, e che `state` sia uno dei valori ammessi); ritorna `LifecycleUnknown` + errore in caso di risposta non valida

Ogni metodo delega a `requestWithRetry`: 1 tentativo iniziale più fino a `maxRetries` retry aggiuntivi (totale `maxRetries + 1` tentativi), timeout per-tentativo derivato da `timeout`, backoff esponenziale dalla slice `backoff`, con verifica della cancellazione del context tra un tentativo e l'altro. Su ogni tentativo fallito (eccetto l'ultimo) emette `slog.Warn` con subject, numero tentativo, delay ed errore. All'esaurimento ritorna un errore aggregato che concatena tutti gli errori individuali separati da `"; "` nella forma `"exhausted retries (N): err1; err2; ..."`.

5.3.6. SystemClock

Implementa `ClockProvider`. Adapter stateless che delega a `time.Now()`. Esiste esclusivamente per soddisfare l'interfaccia in produzione, permettendo ai test di iniettare un clock controllato.

5.3.7. NATSGatewayLifecycleProvider

Implementa `GatewayLifecycleProvider`. Delega la meccanica *NATS*$_G$ RR all'interfaccia `gatewayLifecycleCaller` (soddisfatta da `NATSRRClient`). Incrementa la metrica di errore sui fallimenti e restituisce l'errore al chiamante (la politica fail-open è responsabilità del chiamante).

Interfaccia `gatewayLifecycleCaller`:

Metodo	Firma
`GetGatewayLifecycle`	`(ctx context.Context, tenantID string, gatewayID string) (GatewayLifecycleState, error)`

Interfaccia `lifecycleQueryErrRecorder`:

Metodo	Ritorno
`IncLifecycleQueryErrors()`	—

Campo	Tipo
`client`	`gatewayLifecycleCaller`
`metrics`	`lifecycleQueryErrRecorder`

Metodo	Firma

`GetGatewayLifecycle`	`(ctx context.Context, tenantID string, gatewayID string) (GatewayLifecycleState, error)`

5.4. Adapter driving (`internal/adapter/driving`)

`NATSTelemetryConsumer` e `NATSDecommissionConsumer` dipendono dall'interfaccia condivisa `natsJSSubscriber`. L'interfaccia `drainableSubscription` astrae `*nats`$_G$`.Subscription` per disaccoppiare i test dal tipo concreto *NATS*$_G$ e per supportare la chiamata `Drain()` richiesta dai durable *consumer*$_G$ in shutdown.

Interfaccia `drainableSubscription`:

Metodo	Firma
`Drain`	`() error`
`Unsubscribe`	`() error`

Interfaccia `natsJSSubscriber`:

Metodo	Firma
`Subscribe`	`(subj string, cb nats`$_G$`.MsgHandler, opts ...nats`$_G$`.SubOpt) (drainableSubscription, error)`

`jsAdapter` — struct interna che funge da wrapper `nats`$_G$`.JetStreamContext` e implementa `natsJSSubscriber`. I costruttori pubblici di entrambi i *consumer*$_G$ accettano `nats`$_G$`.JetStreamContext` e lo «avvolgono» in un `jsAdapter`; i costruttori interni accettano `natsJSSubscriber` per permettere l'iniezione nei test.

5.4.1. NATSTelemetryConsumer

Sottoscrive `telemetry.data.>` come durable *consumer*$_G$ *JetStream*$_G$. I messaggi sono bufferizzati e scritti in batch per il *throughput*$_G$.

Per ogni messaggio *NATS*$_G$:

1. Incrementa la metrica `MessagesReceived`.
2. `processMessage` esegue: estrazione del `tenantID` dal subject; parse del body *JSON*$_G$ in `TelemetryEnvelope`; chiamata a `TelemetryMessageHandler.HandleTelemetry`; costruzione della `TelemetryRow`.
3. Il risultato è inviato al canale pending per il batch.
4. `flushLoop` accumula i pending e chiama `WriteBatch` al raggiungimento di `batchSize`, allo scadere di `flushEvery`, o alla cancellazione del context (flush finale).
5. Dopo `WriteBatch`: *ACK*$_G$ su successo; NAK con delay 5s per errori transitori; Term per errori permanenti di parsing (*NATS*$_G$ non invia nuovamente).

Interfaccia `natsJSSubscriber`:

Metodo	Firma
`Subscribe`	`(subj string, cb nats`$_G$`.MsgHandler, opts ...nats`$_G$`.SubOpt) (*nats`$_G$`.Subscription, error)`

Campo	Tipo
`js`	`natsJSSubscriber`

handler	TelemetryMessageHandler
writer	TelemetryWriter
metrics	telemetryConsumerMetrics
logger	*slog.Logger
durableName	string
batchSize	int
flushEvery	time.Duration

Interfaccia `telemetryConsumerMetrics`:

Metodo	Ritorno
IncMessagesReceived()	—
IncMessageParsingErrors()	—
IncMessagesWritten()	—
IncWriteErrors()	—
ObserveWriteLatency(d time.Duration)	—
ObserveBatchSize(size float64)	—

Metodo	Firma	Note
Run	(ctx context.Context) error	Bloccante; alla cancellazione del context esegue un flush finale del *buffer*$_G$ e chiama sub.Drain() per processare i messaggi in-flight prima della chiusura
processMessage	(ctx context.Context, msg *nats$_G$.Msg) (TelemetryRow, error)	Orchestrazione parse → handle → build; ritorna permanentError o errore transitorio
flushLoop	(ctx context.Context, pending <-chan pendingMsg)	Batch con tre trigger di flush
writeBatch	(ctx context.Context, batch []pendingMsg)	Separa errori permanenti dalle righe valide; chiama WriteBatch; *ACK*$_G$/NAK/Term per messaggio
extractTenantID	(subject string) (string, error)	Parsing del subject *NATS*$_G$
buildRow	(tenantID string, envelope TelemetryEnvelope) TelemetryRow	Mapping envelope + tenantID → row
enqueuePending	(ctx context.Context, pending chan<- pendingMsg, pm pendingMsg)	Se ctx è cancellato prima dell'enqueue: Term per permanentError, NakWithDelay(5s) per errori transienti

Tipi interni: permanentError — arricchisce un error per segnalare fallimenti non eseguibili nuovamente. pendingMsg — accoppia un'interfaccia msgAcknowledger (soddisfatta da *nats$_G$.Msg) con la TelemetryRow decodificata e l'eventuale errore.

5.4.2. NATSDecommissionConsumer

Sottoscrive `gateway`$_G$`.decommissioned.>` via *JetStream*$_G$ come durable *consumer*$_G$ (durable name: `"data-consumer`$_G$`-decommission-listener"`) con *ACK*$_G$ manuale. Per ogni messaggio estrae `tenantID` e `gatewayID` dal subject (attesi esattamente 4 segmenti dot-separated) e invoca `DecommissionEventHandler.HandleDecommission`. Per errori di parsing viene richiamato il metodo Term(). Alla cancellazione del context chiama `sub.Drain()` (non `Unsubscribe()`) per processare i messaggi in-flight prima della chiusura — necessario con durable *consumer*$_G$ per evitare perdita di messaggi.

Campo	Tipo
`js`	`natsJSSubscriber`
`handler`	`DecommissionEventHandler`
`logger`	`*slog.Logger`

Metodo	Firma
`Run`	`(ctx context.Context) error`
`handleMsg`	`(msg *nats`$_G$`.Msg)`
`extractIDs`	`(subject string) (tenantID string, gatewayID string, err error)`

5.4.3. HeartbeatTickTimer

Possiede un `time.Ticker` con intervallo `HeartbeatTickMs`. A ogni scatto invoca `HeartbeatTicker.Tick(ctx)`. Si ferma alla cancellazione del context. Emette *telemetria*$_G$ sulla durata del tick tramite un'interfaccia metrica ristretta, in modo che la salute dello scheduler sia osservabile senza accoppiamento alla struct `Metrics` concreta.

Interfaccia `heartbeatTickDurationObserver`:

Metodo	Ritorno
`ObserveHeartbeatTickDuration(d time.Duration)`	—

Campo	Tipo
`ticker`	`HeartbeatTicker`
`metrics`	`heartbeatTickDurationObserver`
`interval`	`time.Duration`

Metodo	Firma
`Run`	`(ctx context.Context)`

5.5. Metriche *Prometheus*$_G$ (`internal/metrics`)

La struct `Metrics` è costruita una volta nel composition root tramite `New(reg prometheus`$_G$`.Registerer)`, che accetta un registrer esplicito (anziché il global default) per isolare le registrazioni nei test. Iniettata agli adapter tramite interfacce metriche ristrette.

Campo	Tipo e Nome *Prometheus*$_G$	Descrizione
`MessagesReceived`	Counter `notip_consumer_messages_received_total`	Messaggi *NATS*$_G$ dequeued

`MessageParsingErrors`	Counter `notip_consumer_message_parsing_errors_total`	Messaggi scartati definitivamente (Term) per *JSON*$_G$ malformato o subject errato
`MessagesWritten`	Counter `notip_consumer_messages_written_total`	Scritture *TimescaleDB*$_G$ riuscite
`WriteErrors`	Counter `notip_consumer_write_errors_total`	Scritture *TimescaleDB*$_G$ fallite
`WriteLatency`	Histogram `notip_consumer_write_duration_seconds`	Durata scrittura su *TimescaleDB*$_G$
`BatchSize`	Histogram `notip_consumer_batch_size`	Numero di record telemetrici per ogni operazione di flush batch
`AlertsPublished`	Counter `notip_consumer_alerts_published_total`	Alert *gateway*$_G$-offline pubblicati
`AlertPublishErrors`	Counter `notip_consumer_alert_publish_errors_total`	Errori di pubblicazione alert
`HeartbeatMapSize`	Gauge `notip_consumer_heartbeat_map_size`	*Gateway*$_G$ attualmente tracciati nella mappa *heartbeat*$_G$
`HeartbeatTickDuration`	Histogram `notip_consumer_heartbeat_tick_duration_seconds`	Tempo di esecuzione del ciclo di Tick completo
`StatusUpdateErrors`	Counter `notip_consumer_status_update_errors_total`	Errori *NATS*$_G$ RR status update
`StatusUpdateDropped`	Counter `notip_consumer_status_update_dropped_total`	Status update scartati (canale pieno)
`DispatchQueueLength`	Gauge `notip_consumer_dispatch_queue_length`	Numero di job attualmente in coda nel canale asincrono
`AlertCacheRefreshErrors`	Counter `notip_consumer_alert_cache_refresh_errors_total`	Errori refresh cache configurazioni alert
`AlertCacheLastSuccess`	Gauge `notip_consumer_alert_cache_last_success_timestamp`	Timestamp Unix dell'ultimo fetch configurazioni riuscito

`NATSReconnects`	Counter `notip_consumer_nats_reconnects_total`	Riconnessioni *NATS*$_G$
`LifecycleQueryErrors`	Counter `notip_consumer_lifecycle_query_errors_total`	Query lifecycle state fallite (per singola query *gateway*$_G$ lifecycle)

La struct `Metrics` soddisfa tutte le narrow metric interface tramite i metodi: `IncMessagesReceived`, `IncMessageParsingErrors`, `IncMessagesWritten`, `IncWriteErrors`, `ObserveWriteLatency(d time.Duration)`, `ObserveBatchSize(size float64)`, `IncAlertsPublished`, `IncAlertPublishErrors`, `SetHeartbeatMapSize(v float64)`, `ObserveHeartbeatTickDuration(d time.Duration)`, `IncStatusUpdateErrors`, `IncStatusUpdateDropped`, `SetDispatchQueueLength(v float64)`, `IncAlertCacheRefreshErrors`, `SetAlertCacheLastSuccess(ts float64)`, `IncNATSReconnects`, `IncLifecycleQueryErrors()`.

5.6. Decisioni implementative

▸ **Tick a tre fasi con re-validazione**

Il ciclo di *heartbeat*$_G$ opera in tre fasi distinte:
1. acquisizione di uno snapshot in sola lettura della mappa;
2. operazioni di I/O (pubblicazione alert, dispatch status update) senza lock mantenuto;
3. riacquisizione del write-lock con re-validazione dello stato. Se durante la fase I/O è arrivato un nuovo messaggio dal *gateway*$_G$ (`lastSeen` avanzato), la transizione a offline viene annullata. Minimizza la contesa sul lock e previene falsi positivi causati dalla latenza delle operazioni di rete.

▸ **Grace period all'avvio**

Per `HeartbeatGracePeriodMs` dall'avvio, `Tick` non emette alert. Consente ai *gateway*$_G$ di stabilire la connessione e inviare il primo messaggio prima che il meccanismo di liveness diventi attivo, evitando alert spurii durante il warm-up.

▸ **Batch write con flush periodico**

I record vengono accumulati in un *buffer*$_G$ interno e scritti in batch con un singolo round-trip di rete. Il flush avviene al raggiungimento della soglia dimensionale (`telemetryBatchSize = 100`) o allo scadere del timer (`telemetryFlushInterval = 500ms`). Gli *ACK*$_G$ *NATS*$_G$ sono emessi solo dopo la scrittura riuscita del batch, garantendo semantica at-least-once delivery.

▸ **Dispatch asincrono non-bloccante per status update**

Le chiamate a `GatewayStatusUpdater.UpdateStatus` sono accodate in un canale a capacità limitata (`GatewayBufferSize`) e processate da una *goroutine*$_G$ worker dedicata. Il path del tick di *heartbeat*$_G$ e di `HandleTelemetry` non attendono mai il completamento della chiamata RR. Se il canale è pieno, l'aggiornamento è scartato deterministicamente e

`status_update_dropped_total` viene incrementato. La perdita è accettabile: il ciclo successivo genererà un nuovo aggiornamento se la condizione persiste.

‣ **Interfacce metriche ristrette per adapter**

Ogni adapter definisce un'interfaccia metrica minimale con i soli metodi necessari. La struct `Metrics` soddisfa tutte queste interfacce, ma ogni adapter è accoppiato esclusivamente ai propri contatori. Semplifica la creazione di mock nei test e impedisce a un adapter di acquisire visibilità sulle metriche di altri componenti.

‣ **Snapshot atomico per AlertConfigCache**

La cache delle configurazioni di alert usa `atomic.Pointer[alertConfigSnapshot]` per garantire letture lock-free in `TimeoutFor`. Il refresh periodico costruisce un nuovo snapshot immutabile e lo sostituisce atomicamente. In caso di errore nel refresh, l'ultimo snapshot valido rimane in uso senza degradare la disponibilità del servizio.

‣ **Struct key per la mappa *heartbeat*$_G$**

`gatewayKey` usa una struct come chiave della mappa invece di una stringa `"tenantID/gatewayID"` formattata. Elimina l'allocazione per la costruzione della chiave stringa e previene ambiguità di collision (es. `"a/b/c"` e `"a/b" + "c"`).

‣ **Lifecycle gate con politica fail-open**

Prima di emettere un alert offline, `Tick` interroga il Management API per lo stato amministrativo del *gateway*$_G$. Se lo stato è `LifecyclePaused`, sia l'alert sia lo status update vengono soppressi: il *gateway*$_G$ è intenzionalmente in pausa e non costituisce un evento anomalo. Se la query RR fallisce (Management API irraggiungibile), il sistema procede comunque con l'alert (*fail-open*): è preferibile un falso positivo operatore-visibile a mascherare un offline reale. L'errore è loggato come `slog.Warn` e incrementa `lifecycle_query_errors_total`.

‣ ***mTLS*$_G$ per autenticazione *NATS*$_G$**

L'autenticazione *NATS*$_G$ usa *mTLS*$_G$ (`RootCAs`, `ClientCert`) al posto del file `.creds`. Ogni servizio dispone di un certificato client firmato dalla CA interna (step-ca), garantendo mutual authentication e consentendo la revoca per certificato senza rotazione di credenziali condivise.

6. Schema database

```
CREATE TABLE IF NOT EXISTS telemetry (
    time           TIMESTAMPTZ NOT NULL,
    tenant_id      TEXT        NOT NULL,
    gateway_id     TEXT        NOT NULL,
    sensor_id      TEXT        NOT NULL,
    sensor_type    TEXT        NOT NULL,
    encrypted_data TEXT        NOT NULL,  -- Rule Zero: mai decodificato server-side
    ivG             TEXT        NOT NULL,  -- Rule Zero: mai decodificato server-side
    auth_tag       TEXT        NOT NULL,  -- Rule Zero: mai decodificato server-side
    key_version    INTEGER     NOT NULL
);

SELECT create_hypertable('telemetry', 'time',
    chunk_time_interval => INTERVAL '1 day',
    if_not_exists       => TRUE);

CREATE INDEX IF NOT EXISTS idx_telemetry_tenant_gateway_time
    ON telemetry (tenant_id, gateway_id, time DESC);
```

Le colonne `encrypted_data`, `ivG`, `auth_tag` sono memorizzate come `TEXT` (stringhe base64) e non vengono mai decodificate server-side — Rule Zero.

7. Metodologie di testing

Il race detector *Go*$_G$ (`-race`) viene eseguito in job dedicati quando esplicitamente previsto dalla *pipeline*$_G$ CI; non costituisce un requisito implicito di tutte le esecuzioni di test (unità e integrazione).

7.1. Test di unità

Le dipendenze sono sostituite da mock o stub iniettati tramite constructor. Il clock è sempre un `FakeClock` controllato per rendere i test deterministici.

`HeartbeatTracker`

Caso di test	Postcondizione verificata
`HandleTelemetry` — primo messaggio da *gateway*$_G$ sconosciuto	Entry inserita in `beats`; `knownStatus = Online`; `dispatchCh` riceve un job `Online`; metrica `HeartbeatMapSize` incrementata
`HandleTelemetry` — messaggio da *gateway*$_G$ con `knownStatus = Offline`	`dispatchCh` riceve un job `Online`; `knownStatus` aggiornato a `Online`
`HandleTelemetry` — messaggio da *gateway*$_G$ già `Online`	`dispatchCh` non riceve nessun job
`HandleDecommission` — *gateway*$_G$ presente in mappa	Entry rimossa; metrica `HeartbeatMapSize` decrementata; nessun altro side effect
`HandleDecommission` — *gateway*$_G$ assente in mappa	Nessun errore; nessun side effect
`Tick` — grace period attivo	`alertPublisher.Publish` non chiamato
`Tick` — timeout superato, nessun aggiornamento intervenuto	`alertPublisher.Publish` chiamato; `dispatchCh` riceve job `Offline`; `knownStatus` aggiornato a `Offline` dopo Fase 3
`Tick` — re-validazione: *heartbeat*$_G$ arrivato durante Fase 2 simulata	Transizione annullata; `alertPublisher.Publish` non chiamato; `knownStatus` rimane `Online`
`Tick` — *gateway*$_G$ già `knownStatus = Offline`	Nessun alert duplicato; `alertPublisher.Publish` non chiamato
`Tick` — lifecycle gate: stato `LifecyclePaused`	`lifecycleProvider.GetGatewayLifecycle` restituisce `LifecyclePaused`; `alertPublisher.Publish` non chiamato; nessun dispatch `Offline`; `knownStatus` rimane `Online`
`Tick` — lifecycle gate: errore RR (fail-open)	`lifecycleProvider.GetGatewayLifecycle` restituisce errore; `alertPublisher.Publish` chiamato comunque; `IncLifecycleQueryErrors` invocato

`dispatchStatusUpdate` — canale pieno	Job scartato; `IncStatusUpdateDropped` invocato
`Close` — drain del canale	Tutti i job in `dispatchCh` prima della chiusura sono processati; `done` chiuso

`NATSGatewayLifecycleProvider`

Caso di test	Postcondizione verificata
`GetGatewayLifecycle` — client restituisce `LifecyclePaused`	Stato `LifecyclePaused` propagato al chiamante; `IncLifecycleQueryErrors` non invocato
`GetGatewayLifecycle` — client restituisce `LifecycleOnline`	Stato `LifecycleOnline` propagato al chiamante; nessun errore
`GetGatewayLifecycle` — client restituisce errore *NATS*$_G$	Errore propagato; `IncLifecycleQueryErrors` invocato esattamente una volta; stato restituito è `LifecycleUnknown`

`NATSRRClient` — `GetGatewayLifecycle`

Caso di test	Postcondizione verificata
`GetGatewayLifecycle` — risposta valida con stato `LifecyclePaused`	Stato `LifecyclePaused` restituito; nessun errore
`GetGatewayLifecycle` — errore *NATS*$_G$	Errore restituito; stato è `LifecycleUnknown`
`GetGatewayLifecycle` — risposta *JSON*$_G$ malformata	Errore di unmarshal restituito; stato è `LifecycleUnknown`
`GetGatewayLifecycle` — stato non valido nella risposta (`"broken"`)	Errore `"invalid gateway`$_G$ `lifecycle state"` restituito; stato è `LifecycleUnknown`
`GetGatewayLifecycle` — `gateway_id` nella risposta non corrisponde all'atteso	Errore `"gateway_id mismatch"` restituito; stato è `LifecycleUnknown`

`AlertConfigCache`

Caso di test	Postcondizione verificata
`TimeoutFor` — override per *gateway*$_G$ presente nello snapshot	Restituisce `TimeoutMs` specifico del *gateway*$_G$
`TimeoutFor` — solo default *tenant*$_G$ presente	Restituisce `TimeoutMs` del *tenant*$_G$
`TimeoutFor` — nessuna configurazione presente	Restituisce `defaultTimeoutMs`
`TimeoutFor` su snapshot vuoto (costruzione senza fetch)	Restituisce `defaultTimeoutMs`; nessun panic

`NATSTelemetryConsumer`

Caso di test	Postcondizione verificata
`extractTenantID` — subject `telemetry.data.tenant1.gw1`	Restituisce `"tenant1"`
`extractTenantID` — subject malformato (es. `telemetry.data.tenant1`)	Restituisce errore
`buildRow` — mapping da `TelemetryEnvelope` con `tenantID`	`TenantID` propagato correttamente; campi `OpaqueBlob` passati intatti senza accesso a `Value`
`processMessage` — body *JSON*$_G$ non valido	Ritorna `permanentError`; handler non invocato
`writeBatch` — errore permanente per un messaggio nel batch	Messaggio Term()“d; gli altri messaggi validi del batch sono *ACK*$_G$()”d

`NATSDecommissionConsumer`

I test di unità usano l'interfaccia `drainableSubscription` e uno stub `fakeSubscription` per disaccoppiare i test dal tipo concreto `*nats`$_G$`.Subscription`, necessario dopo l'introduzione di `sub.Drain()` per i *consumer*$_G$ durevoli.

Caso di test	Postcondizione verificata
`Run` — context cancellato	`sub.Drain()` chiamato; `Run` restituisce `nil`
`extractIDs` — subject valido `gateway`$_G$`.decommissioned.t1.gw-1`	`tenantID` e `gatewayID` estratti correttamente
`extractIDs` — subject con numero di token errato	Restituisce errore
`handleMsg` — subject valido	`HandleDecommission` invocato con i parametri corretti; messaggio *ACK*$_G$()“d
`handleMsg` — subject malformato	`HandleDecommission` non invocato; messaggio Term()“d

7.2. Test di integrazione

I test di integrazione avviano infrastruttura reale tramite *Testcontainers*$_G$-*go*$_G$: *NATS*$_G$ *JetStream*$_G$ con *mTLS*$_G$ (certificati effimeri generati a runtime) e *TimescaleDB*$_G$. I test *Prometheus*$_G$ non richiedono container: usano `httptest.NewServer` con `promhttp.HandlerFor` su un registry isolato.

`PostgresTelemetryWriter`

Caso di test	Infrastruttura	Verifica
`Write` — singola riga	*TimescaleDB*$_G$	Record presente in `telemetry`; campi `OpaqueBlob` invariati rispetto all'input (Rule Zero)
`WriteBatch` — batch di più righe	*TimescaleDB*$_G$	Tutti i record persistiti; order preservato; campi `OpaqueBlob` invariati (Rule Zero)

`NATSTelemetryConsumer`

Caso di test	Infrastruttura	Verifica
Messaggio *NATS*$_G$ valido → *TimescaleDB*$_G$	*NATS*$_G$ + *TimescaleDB*$_G$	Record su `telemetry`; campi `OpaqueBlob` invariati; *ACK*$_G$ emesso
Messaggio *NATS*$_G$ con *JSON*$_G$ malformato	*NATS*$_G$ + *TimescaleDB*$_G$	Nessun record su DB; messaggio Term()“d (`NumPending = 0`, `NumAckPending = 0`)

`NATSDecommissionConsumer`

Caso di test	Infrastruttura	Verifica
Evento decommission valido	*NATS*$_G$	`HandleDecommission` invocato con `tenantID` e `gatewayID` estratti dal subject
Tre eventi per *tenant*$_G$/*gateway*$_G$ distinti	*NATS*$_G$	Tutti e tre gli eventi processati; nessuno perso
Subject con token extra (malformato)	*NATS*$_G$	Messaggio Term()“d (`NumPending = 0`); handler non invocato
`Run` — context cancellato	*NATS*$_G$	`Run` restituisce `nil` entro 3 secondi dalla cancellazione

`NATSAlertPublisher`

Caso di test	Infrastruttura	Verifica
Publish su subject corretto	*NATS*$_G$	Messaggio ricevuto su `alert.{tenantId}.gw_offline`; *payload*$_G$ *JSON*$_G$ corrispondente all’input
Isolamento multi-*tenant*$_G$	*NATS*$_G$	Subscriber del *tenant*$_G$ A non riceve l’alert del *tenant*$_G$ B e viceversa

`NATSRRClient`

Caso di test	Infrastruttura	Verifica
`FetchAlertConfigs` — risposta valida	*NATS*$_G$	Configs deserializzate correttamente; *gateway*$_G$-specific e *tenant*$_G$-level distinti
`FetchAlertConfigs` — nessun responder	*NATS*$_G$	Errore di timeout restituito
`UpdateGatewayStatus` — risposta valida	*NATS*$_G$	*Payload*$_G$ della richiesta ricevuto dal mock responder con i campi corretti
`UpdateGatewayStatus` — nessun responder	*NATS*$_G$	Errore di timeout restituito
`GetGatewayLifecycle` — risposta valida	*NATS*$_G$	Stato `LifecyclePaused` restituito; *payload*$_G$ della richiesta verificato (`gateway_id`, `tenant_id`)

`GetGatewayLifecycle` — nessun responder	*NATS*$_G$	Errore di timeout restituito
`GetGatewayLifecycle` — risposta con stato non valido (`"invalid-state"`)	*NATS*$_G$	Errore restituito; stato è `LifecycleUnknown`

`AlertConfigCache`

Caso di test	Infrastruttura	Verifica
Fetch iniziale popola la cache	*NATS*$_G$	`TimeoutFor` restituisce il valore *gateway*$_G$-specific; fallback a *tenant*$_G$-level; fallback a default per *tenant*$_G$ sconosciuto
Refresh periodico aggiorna la cache	*NATS*$_G$	Dopo aggiornamento del mock responder, `TimeoutFor` riflette i nuovi valori entro il ciclo di refresh
Nessun responder disponibile	*NATS*$_G$	Cache rimane sul default; metrica `IncAlertCacheRefreshErrors` incrementata

`NATSGatewayLifecycleProvider`

Caso di test	Infrastruttura	Verifica
Stato `LifecyclePaused` dal Management API	*NATS*$_G$	Stato `LifecyclePaused` restituito; *payload*$_G$ della richiesta verificato (`gateway_id`, `tenant_id`); nessun errore metrica
Stato `LifecycleOnline` dal Management API	*NATS*$_G$	Stato `LifecycleOnline` restituito; nessun errore
Nessun responder (timeout)	*NATS*$_G$	Errore restituito; `IncLifecycleQueryErrors` invocato esattamente una volta

`HeartbeatTracker`

Caso di test	Infrastruttura	Verifica
Ciclo completo Online → Offline → Online	*NATS*$_G$	Alert offline pubblicato su stream `ALERTS` con *payload*$_G$ corretto; dispatch `Offline` e secondo dispatch `Online` (recovery) emessi
Grace period sopprime gli alert	*NATS*$_G$	Nessun alert sul *JetStream*$_G$ durante la finestra di grace, anche con clock avanzato oltre il timeout
Decommission rimuove il *gateway*$_G$	*NATS*$_G$	Dopo `HandleDecommission`, nessun alert su `Tick`; `HeartbeatMapSize = 0`

Lifecycle gate: stato `LifecyclePaused` via *NATS*$_G$ reale	*NATS*$_G$	Nessun alert sul *JetStream*$_G$; nessun dispatch `Offline`; mock responder su `get-status` verificato

***Prometheus*$_G$ — *endpoint*$_G$ `/metrics`**

Caso di test	Infrastruttura	Verifica
Tutti i metric name esposti	— (httptest)	Tutti i nomi attesi presenti nell'output text-format; `Content-Type` corretto
Valore counter riflette gli incrementi	— (httptest)	Dopo 3 chiamate a `IncMessagesReceived`, scrape riporta `...messages_received_total 3`
Valore gauge riflette `Set`	— (httptest)	Dopo `SetHeartbeatMapSize(7)`, scrape riporta `...heartbeat_map_size 7`

***Pipeline*$_G$ E2E**

Caso di test	Infrastruttura	Verifica
Flusso dati completo: *NATS*$_G$ → DB → alert → status	*NATS*$_G$ + *TimescaleDB*$_G$	*Telemetria*$_G$ persistita su *TimescaleDB*$_G$ (Rule Zero verificata sui blob); *gateway*$_G$ Online; dopo timeout alert offline su stream `ALERTS`; dispatch `Offline` emesso