NoTIP

NO TESTS IN PRODUCTION

Specifica tecnica - Simulator *backend*$_G$ & *CLI*$_G$

Versione	1.1.0
Data Modifica	2026-04-17
Utilizzo	Esterno

Abstract dei contenuti

Specifica tecnica dei *microservizi*$_G$ notip-simulator-*backend*$_G$ e notip-simulator-*cli*$_G$: architettura interna (Ports & Adapters), simulazione di *gateway*$_G$ e sensori, generazione dati, integrazioni *NATS*$_G$ *mTLS*$_G$, schema del database locale, interfaccia a riga di comando per il controllo operativo dell'ambiente di simulazione e metodologie di testing.

Changelog$_G$

Versione	Data	Autori	*Verificatore*$_G$	Descrizione
1.1.0	2026-04-17	Francesco Marcon	Leonardo Preo	Correzioni a seguito del colloquio tecnico di PB
1.0.0	2026-04-11	Leonardo Preo (responsabile)		Approvazione per ingresso in *baseline*$_G$ PB
0.5.0	2026-04-10	Francesco Marcon	Valerio Solito	Stesura delle metodologie di testing multi-livello per *backend*$_G$ e *CLI*$_G$
0.4.0	2026-04-10	Valerio Solito	Francesco Marcon	Definizione dell'architettura della *CLI*$_G$
0.3.0	2026-04-09	Francesco Marcon	Mario De Pasquale	Definizione API *REST*$_G$, *payload*$_G$ e integrazione *NATS*$_G$ *mTLS*$_G$ per il *backend*$_G$
0.2.0	2026-04-09	Francesco Marcon	Mario De Pasquale	Stesura architettura logica Ports & Adapters e componenti del simulator-*backend*$_G$
0.1.0	2026-04-08	Valerio Solito	Francesco Marcon	Definizione della configurazione di base dei servizi
0.0.1	2026-04-08	Francesco Marcon	Valerio Solito	Prima stesura del documento

Indice

Indice delle tabelle

1. Introduzione al Documento

Il presente documento raccoglie le specifiche tecniche di due *microservizi*$_G$ distinti: `notip-simulator-backend`$_G$ e `notip-simulator-cli`$_G$. Entrambi i componenti appartengono al sottosistema di simulazione della piattaforma NoTIP e sono progettati per operare in stretta sinergia: il *backend*$_G$ costituisce il motore di simulazione, mentre la *CLI*$_G$ rappresenta il piano di controllo operativo che consente agli operatori di pilotarlo.

La scelta di riunire le specifiche dei due servizi in un unico documento è motivata dal loro accoppiamento funzionale: la *CLI*$_G$ non possiede utilità autonoma al di fuori del contesto del *backend*$_G$, essendo interamente dedicata alla traduzione di comandi umani in chiamate HTTP verso le sue API *REST*$_G$. Trattarle come entità documentali separate produrrebbe una frammentazione che non rifletterebbe la reale dipendenza architetturale tra i due servizi.

Il documento è strutturato in due parti principali:

- **Parte I — `notip-simulator-backend`$_G$**: architettura esagonale, ciclo di vita dei *gateway*$_G$ simulati, integrazioni *NATS*$_G$ *mTLS*$_G$, schema SQLite e *pipeline*$_G$ di testing del *backend*$_G$.
- **Parte II — `notip-simulator-cli`$_G$**: struttura del binario *Go*$_G$, modelli di dominio, client HTTP, strato dei comandi Cobra, REPL interattiva e strategia di testing della *CLI*$_G$.

2. Parte I — notip-simulator-*backend*$_G$

2.1. Introduzione

Questa sezione illustra l'architettura interna e le scelte implementative del microservizio `notip-simulator-backend`$_G$. Sviluppato in *Go*$_G$, questo componente è responsabile della simulazione di dispositivi fisici (*Gateway*$_G$ e Sensori *BLE*$_G$) su larga scala. Il simulatore replica fedelmente il comportamento dell'hardware reale all'interno della piattaforma: si interfaccia con il *Provisioning*$_G$ Service per il processo di *onboarding*$_G$ e l'ottenimento del materiale crittografico, cifra i dati telemetrici localmente tramite *AES-256*$_G$*-GCM*$_G$, si connette al cluster *NATS*$_G$ in *mTLS*$_G$ per l'invio asincrono della *telemetria*$_G$ e rimane in ascolto su *JetStream*$_G$ per ricevere comandi dal Cloud o gestire eventi di *decommissioning*$_G$. L'architettura è altamente concorrente, associando ogni *gateway*$_G$ virtuale a una *goroutine*$_G$ dedicata gestita da un registro centrale.

2.2. Dipendenze e Configurazione

2.2.1. Variabili d'ambiente

Tutte le variabili d'ambiente necessarie per il funzionamento del microservizio sono elencate di seguito. Un'eventuale mancanza o configurazione errata delle variabili obbligatorie comporterà un errore fatale all'avvio del microservizio:

Campo	Variabile d'ambiente	Default	Obbligatorio
ProvisioningUrl	`PROVISIONING_URL`	-	Sì
NATSUrl	`NATS_URL`	-	Sì
NATSCACertPath	`NATS_CA_CERT_PATH`	-	Sì
NATSTLSCertPath	`NATS_TLS_CERT`	-	No
NATSTLSKeyPath	`NATS_TLS_KEY`	-	No

SQLitePath	SQLITE_PATH	/data/simulator.db	No
HttpAddr	HTTP_ADDR	:8090	No
MetricsAddr	METRICS_ADDR	:9090	No
DefaultSendFreq	DEFAULT_SEND_FREQUENCY_MS	5000	No
GatewayBufferSize	GATEWAY_BUFFER_SIZE	1000	No
RecoveryMode	RECOVERY_MODE	false	No

Tabella 1: Variabili d'ambiente del microservizio notip-simulator-*backend*$_G$

Nota: `NATS_TLS_CERT` *e* `NATS_TLS_KEY` *non sono obbligatori in assoluto, ma sono vincolati tra loro: devono essere entrambi valorizzati o entrambi vuoti. Se le variabili* `DEFAULT_SEND_FREQUENCY_MS` *o* `GATEWAY_BUFFER_SIZE` *vengono omesse o impostate a valori invalidi, il sistema* `config` *non va in panico ma applica automaticamente i fallback hardcoded di 5000 e 1000.*

2.2.2. Sequenza di avvio

I passi bloccanti interrompono l'avvio del microservizio, pertanto è necessario assicurarsi che le dipendenze esterne (*Provisioning*$_G$, *NATS*$_G$) siano raggiungibili. La sequenza di avvio è la seguente:

Step	Componente	Azione	Bloccante?
0	`config.Load()`	Carica e valida le variabili d'ambiente dal sistema operativo.	Sì
1	`SQLiteStore`	Inizializza la connessione SQLite (`modernc.org/sqlite`) ed esegue le migrazioni embeddate per lo schema dati.	Sì
2	`Adapters`	Istanzia il connettore *NATS*$_G$ *mTLS*$_G$ caricando il CertPool della CA, il client HTTP di *Provisioning*$_G$ e l'Encryptor AES-*GCM*$_G$.	Sì
3	`GatewayRegistry`	Inizializza il registro centrale thread-safe che orchestra i `GatewayWorker`.	Sì
4	`RestoreAll`	Se `RECOVERY_MODE` è true, interroga il DB locale e riavvia i *gateway*$_G$ pre-esistenti.	No
5	`DecommissionListener`	Sottoscrizione a *JetStream*$_G$ su `gateway`$_G$`.decommissioned.>` per cleanup locale in tempo reale.	Sì
6	`Metrics Server`	Avvia il listener HTTP per l'esposizione delle metriche *Prometheus*$_G$ (es. `:9090`).	No
7	`API Server`	Avvia il `http.ServeMux` con gli handler *REST*$_G$ per il controllo delle simulazioni.	Sì

Tabella 2: Sequenza di avvio del microservizio notip-simulator-*backend*$_G$

2.3. Architettura Logica

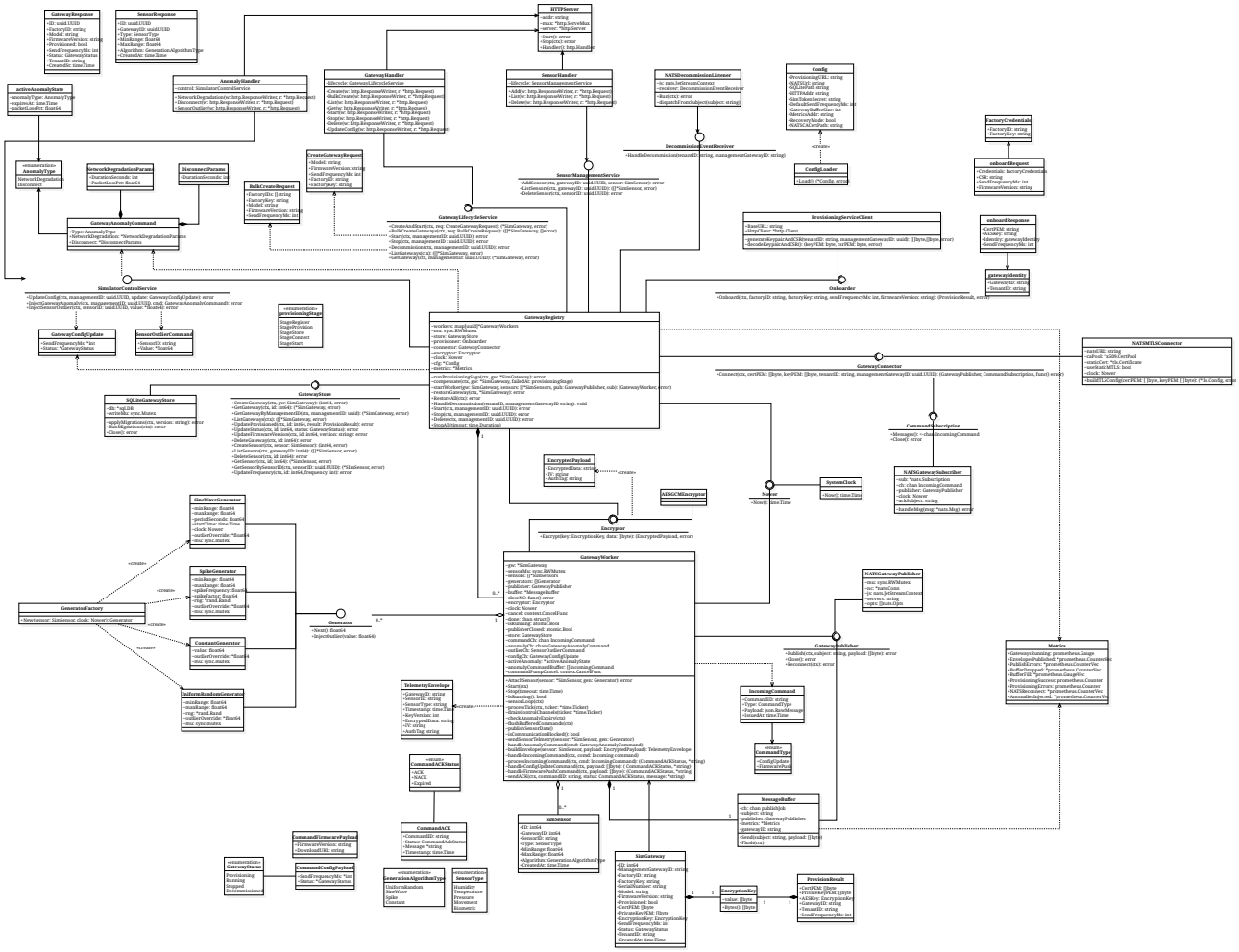
Il servizio adotta un'architettura **Ports and Adapters (Architettura Esagonale)**. La logica di business (generazione dati, gestione anomalie e ciclo di vita) è isolata al centro (Domain/App) e non dipende da framework infrastrutturali. Le comunicazioni verso l'esterno avvengono

tramite interfacce (Ports) implementate dagli adapter (SQLite, *NATS*$_G$, HTTP). L'interazione tra il Core applicativo e il sistema di persistenza (SQLite) è mediata dal *Repository*$_G$ Pattern tramite l'interfaccia GatewayStore. Questa scelta è stata presa per evitare di accoppiare ulteriormente la logica di business allo schema del database. A tal proposito il *Repository*$_G$ Pattern ci ha permesso di trattare la persistenza come un dettaglio implementativo. In questo modo, se si dovesse cambiare e scegliere di migrare a *PostgreSQL*$_G$ o a un database NoSQL, sarà sufficiente scrivere un nuovo Adapter che implementi l'interfaccia GatewayStore, lasciando il Core del simulatore intatto. Questo isolamento è anche fondamentale per iniettare mock durante i test d'unità per la logica di dominio.

2.3.1. Layout dei moduli

Essendo il microservizio strutturato per isolare la logica applicativa dalle dipendenze infrastrutturali, di seguito è riportata la struttura interna e rigorosa dei pacchetti:

```
notip-simulator-backend_G/
├── internal/
│   ├── adapters/              Implementazioni tecniche e Adapter Driven:
│   │   ├── http/              Handler REST_G, DTO_G mapper e client Provisioning_G
│   │   ├── nats_G/             Connettore mTLS_G, Pub/Sub_G JetStream_G e Listener
│   │   ├── sqlite/            Persistenza locale SQLite (store.go_G)
│   │   ├── clock.go_G          Implementazione di utilità del tempo di sistema
│   │   └── encryptor.go_G      Implementazione dell'algoritmo AES-256_G-GCM_G
│   ├── app/                   Core orchestrativo: app.go_G, buffer_G.go_G, registry.go_G,
worker.go_G
│   ├── config/                Caricamento env vars e defaults
│   ├── domain/                Modelli di business (Gateway_G, Sensor) ed Errori
│   ├── generator/             Algoritmi matematici e Factory (sine, spike, ecc.)
│   ├── health/                Endpoint_G di Liveness e Readiness
│   ├── metrics/               Definizione metriche Prometheus_G
│   ├── migrations/            Script SQL embeddati per lo schema
│   └── ports/                 Definizione interfacce Driving (API) e Driven
├── tests/                     Test di integrazione intra-service e mock
└── main_G.go_G                 Entrypoint dell'applicazione
```



Architettura Logica del Simulator *Backend*$_G$.

2.3.2. Strati Architetturali

Strato	Package	Contenuto
Presentation	`adapters/http`	Handler HTTP *REST*$_G$ per la creazione, l'avvio, l'arresto e la configurazione delle anomalie dei *gateway*$_G$ simulati.
Business	`app`, `generator`, `domain`	Logica applicativa: esecuzione dei tick temporali nei worker, calcolo dei dati, cifratura *payload*$_G$.
Persistence	`adapters/sqlite`	Accesso al database locale SQLite per conservare lo stato dei *gateway*$_G$ e dei sensori creati via API.
Integration	`adapters/nats`$_G$, `adapters/http`	Scambio dati verso l'infrastruttura Cloud (*Provisioning*$_G$ via *REST*$_G$, *Telemetria*$_G$ e Comandi via *JetStream*$_G$).

Tabella 3: Strati architetturali del microservizio notip-simulator-*backend*$_G$

2.3.3. Decisioni Architetturali

Per garantire le performance, la resilienza e la manutenibilità del simulatore senza appesantire l'infrastruttura, sono state prese le seguenti decisioni architetturali chiave:

- **Modello di Concorrenza (1 *Goroutine*$_G$ = 1 *Gateway*$_G$):** Ogni *gateway*$_G$ simulato è gestito da un `GatewayWorker` isolato eseguito in una propria *goroutine*$_G$ dedicata. Questo approccio mappa 1:1 il comportamento, garantendo che la congestione o il crash di un *gateway*$_G$ virtuale non impatti l'esecuzione degli altri. La sincronizzazione e il ciclo di vita sono orchestrati in sicurezza tramite `context.Context` e `sync.RWMutex` nel registro centrale.
- **Gestione della Backpressure (Drop-Oldest):** Per scongiurare l'esaurimento della memoria (Out-Of-Memory) o il blocco permanente delle *goroutine*$_G$ in caso di prolungata irraggiungibilità di *NATS*$_G$, è stato implementato il componente `MessageBuffer`. Questo canale a capacità limitata adotta una politica «Drop-Oldest»: in caso di saturazione, espelle silenziosamente il pacchetto telemetrico più vecchio per fare spazio al nuovo, privilegiando sempre il dato più recente.
- **Persistenza Locale e Serializzazione:** L'utilizzo di SQLite (`modernc.org/sqlite`) garantisce la persistenza per la funzionalità di riavvio a caldo (`RecoveryMode`). Per prevenire corruzioni o errori di `database is locked` derivanti dalle decine di worker *goroutine*$_G$ concorrenti, l'adapter impone un vincolo ferreo di una singola connessione attiva (`SetMaxOpenConns(1)`) e serializza programmaticamente tutte le operazioni di mutazione tramite un lock esclusivo (`writeMu sync.Mutex`).
- ***Pipeline*$_G$ Crittografica Opaca (AES-*GCM*$_G$):** Rispettando il vincolo architetturale della piattaforma NoTIP, il *payload*$_G$ telemetrico non viene mai esposto in chiaro sul broker di messaggistica. Il `GatewayWorker` utilizza l'`AESGCMEncryptor` locale per sigillare i dati prima dell'invio. La `EncryptionKey` è gestita come un *Value Object* immutabile che impedisce l'accesso diretto ai byte della chiave, garantendo che i campi `EncryptedData`, `IV`$_G$ e `AuthTag` viaggino come blob Base64 totalmente opachi.
- ***Buffer*$_G$ Comandi durante Anomalie (Defer-and-Flush):** Quando un `GatewayWorker` è in stato di anomalia attiva (es. disconnessione totale), i comandi *JetStream*$_G$ in ingresso non vengono né scartati né elaborati immediatamente. L'`anomalyCommandBuffer` — uno slice in memoria locale al worker — li accoda preservando l'ordine di arrivo. Al termine dell'anomalia, `flushBufferedCommands` li processa sequenzialmente, inviando l'*ACK*$_G$ per ciascuno. Questo approccio replica il comportamento reale di un dispositivo che riprende l'attività dopo un'interruzione, evitando sia la perdita silenziosa dei comandi sia elaborazioni concorrenti non sicure.
- **Motore di Generazione (Strategy & Factory Pattern):** Per la produzione delle misure, il sistema adotta una combinazione di Strategy e Factory Pattern. La necessità era gestire diversi algoritmi (Seno, Spike, ecc.) in modo estensibile. Senza questi pattern, avremmo dovuto usare dei cicli if/else o switch pesanti nel loop di invio, rendendo difficile l'aggiunta di nuovi sensori. Lo Strategy Pattern rende gli algoritmi intercambiabili come "spine" (Generator), mentre la Factory centralizza la loro creazione. Questa scelta è superiore a una semplice istanziazione manuale perché rispetta l'Open/Closed Principle: possiamo aggiungere nuovi tipi di sensori creando nuovi file, senza mai dover modificare il codice del motore di simulazione.

- **Gestione Ciclo di Vita:** Il coordinamento della terminazione dei *gateway*$_G$ è affidato all'`DecommissionListener`. Il problema era come notificare a più componenti (Worker, Database, Tickers) che un *gateway*$_G$ è stato rimosso dal cloud senza che il client *NATS*$_G$ dovesse conoscere tutti i moduli del sistema. Un approccio procedurale (chiamate dirette) avrebbe creato un accoppiamento stretto e fragile. La realizzazione di interfacce dedicate risolve questo problema permettendo ai moduli di iscriversi all'evento di terminazione in modo indipendente.
- **Protezione del Materiale Crittografico (Value Object Pattern):** La chiave crittografica viene incapsulata in un'entità immutabile (`EncryptionKey`) che ne impedisce l'accesso diretto ai byte. Il problema era evitare la fuga accidentale del materiale sensibile durante operazioni di routine come il logging di sistema o la serializzazione dei dati. Gestire la chiave come un semplice tipo primitivo (stringa o `[]byte`) avrebbe reso il sistema vulnerabile ad esposizioni involontarie nei log di debug o nelle risposte API in caso di disattenzione. Il Value Object risolve questa criticità applicando un principio di **defensive design**: il segreto è protetto all'interno di una struttura che sovrascrive i metodi per trasformare in stringa, rendendo la chiave accessibile solo tramite chiamate esplicite e controllate. Questo garantisce che la sicurezza non dipenda dalla costante attenzione dello sviluppatore, ma sia integrata nativamente nella struttura del codice.

2.3.4. Relazioni tra Componenti

Di seguito viene sintetizzata la catena delle dipendenze interne, che evidenzia la corretta applicazione dell'inversione del controllo (Ports & Adapters):

Componente Core	Relazione	Porta / Adapter
`GatewayHandler`	Gestisce ciclo di vita via	`GatewayRegistry`
`SensorHandler`	Gestisce entità via	`GatewayRegistry`
`AnomalyHandler`	Inietta alterazioni via	`GatewayRegistry`
`GatewayRegistry`	Delega a	`ProvisioningServiceClient` (HTTP)
`GatewayRegistry`	Crea ed orchestra	`GatewayWorker` (*Goroutine*$_G$)
`GatewayRegistry`	Persiste lo stato su	`GatewayStore` / SQLite (Driven)
`GatewayRegistry`	Richiede connessioni a	`NATSMTLSConnector` (Driven)
`GatewayWorker`	Cifra tramite	`AESGCMEncryptor`
`GatewayWorker`	Interroga	`Generator` (`SineWave`, ecc.)
`GatewayWorker`	Accoda in	`MessageBuffer`
`MessageBuffer`	Pubblica su	`NATSGatewayPublisher` (*NATS*$_G$)
`NATSGatewaySubscriber`	Inoltra i comandi cloud a	`GatewayWorker` (Core)
`NATSDecommissionListener`	Notifica	`GatewayRegistry`

Tabella 4: Relazioni funzionali interne

2.4. Design di Dettaglio

2.4.1. Moduli del microservizio

Modulo	Responsabilità
`GatewayRegistry`	Entry-point per tutte le chiamate API. Gestisce una mappa thread-safe di `GatewayWorker` associati al proprio management ID. Apre e chiude i contesti e delega al database.
`GatewayWorker`	*Goroutine*$_G$ isolata per singolo *gateway*$_G$. Gestisce un `time.Ticker` interno, acquisisce dati dai sensori, li cifra in AES-*GCM*$_G$ e li accoda per la pubblicazione. Elabora anche i comandi in ingresso.
`MessageBuffer`	Sistema di code (channel) con capienza limitata per assorbire i picchi di rete o *NATS*$_G$ lento, applicando una politica **drop-oldest** per non bloccare la simulazione.
`Generator`	Interfaccia comune per gli algoritmi matematici. Implementa metodi per produrre il sample successivo e un meccanismo di override (`InjectOutlier`).

Tabella 5: Responsabilità dei moduli applicativi

2.4.2. Entità

Entità	Campi principali
`gateways`	`id`, `management_gateway_id`, `factory_id`, `factory_key`, `model`, `firmware_version`, `provisioned`, `cert_pem`, `private_key_pem`, `encryption_key`, `send_frequency_ms`, `status`, `tenant_id`, `created_at`
`sensors`	`id`, `gateway_id`, `sensor_id`, `type`, `min_range`, `max_range`, `algorithm`, `created_at`

Tabella 6: Entità persistenti (SQLite)

2.4.3. *Endpoint*$_G$ API HTTP

L'API server (configurato nel file `server.go`$_G$) espone gli *endpoint*$_G$ per pilotare la simulazione. Le risposte *JSON*$_G$ (`GatewayResponse`) omettono sempre per sicurezza il materiale crittografico sensibile (chiavi AES, chiavi private *PEM*$_G$). Inoltre, grazie a `PRAGMA foreign_keys(1)`, le operazioni di `DELETE` su un *gateway*$_G$ innescano un `ON DELETE CASCADE` su SQLite, rimuovendo automaticamente i sensori associati. Di seguito il dettaglio completo delle rotte.

2.4.3.1. Health

Rotta	`GET /health`
Descrizione	Liveness probe. Ritorna HTTP 200 con `{"status": "ok"}` per confermare che il server HTTP è in ascolto.

Tabella 7: *Endpoint*$_G$ GET /health

2.4.3.2. Gateways

Rotta	`POST /sim/gateways`
Descrizione	Crea un *gateway*$_G$ locale, esegue l'*onboarding*$_G$ crittografico con il Cloud e avvia il worker.

Body Request	`{"factoryId": "...", "factoryKey": "...", "model": "...", "firmwareVersion": "...", "sendFrequencyMs": 5000}`
Response	`GatewayResponse` *JSON*$_G$. HTTP 201 Created.

Tabella 8: *Endpoint*$_G$ POST /sim/gateways

Rotta	**`POST /sim/gateways/bulk`**
Descrizione	Creazione massiva di N *gateway*$_G$. Utile per test di carico.
Body Request	`{"factoryIds": ["sim-001", "sim-002", ...], "factoryKey": "...", "model": "...", "firmwareVersion": "...", "sendFrequencyMs": 5000}`
Response	Oggetto *JSON*$_G$ contenente gli array `gateways` ed `errors`. HTTP 201 Created in caso di successo totale; HTTP 207 Multi-Status in caso di fallimenti parziali.

Tabella 9: *Endpoint*$_G$ POST /sim/gateways/bulk

Nota: L'operazione di creazione massiva è limitata internamente a un massimo di 10 esecuzioni concorrenti tramite semaforo a canale, per prevenire l'esaurimento delle risorse.

Rotta	**`GET /sim/gateways`**
Descrizione	Recupera la lista di tutti i *gateway*$_G$ locali presenti nel DB simulatore.
Response	Array di `GatewayResponse` (senza materiale crittografico). HTTP 200.

Tabella 10: *Endpoint*$_G$ GET /sim/gateways

Rotta	**`GET /sim/gateways/{id}`**
Descrizione	Recupera il dettaglio di un singolo *gateway*$_G$ partendo dal suo *UUID*$_G$ (Management ID).
Response	Singolo `GatewayResponse` *JSON*$_G$. HTTP 200. HTTP 404 se non trovato.

Tabella 11: *Endpoint*$_G$ GET /sim/gateways/{id}

Rotte	**`POST /sim/gateways/{id}/start`** **`POST /sim/gateways/{id}/stop`**
Descrizione	Avvia la *goroutine*$_G$ (worker) di un *gateway*$_G$ fermo, o la arresta chiudendo il Context.
Response	HTTP 204 No Content in caso di successo.

Tabella 12: *Endpoint*$_G$ POST Lifecycle (Start / Stop)

Rotta	**`DELETE /sim/gateways/{id}`**
Descrizione	Arresta il worker (se attivo) e rimuove in modo permanente il *gateway*$_G$ e i suoi sensori dal database locale.
Response	HTTP 204 No Content.

Tabella 13: *Endpoint*$_G$ DELETE /sim/gateways/{id}

2.4.3.3. Sensori

Rotta	**`POST /sim/gateways/{id}/sensors`**
Descrizione	Aggiunge un nuovo sensore (generatore dati) a un *gateway*$_G$ esistente.

Body Request	`{"type": "temperature", "minRange": 20.0, "maxRange": 25.0, "algorithm": "sine_wave"}`
Response	`SensorResponse` *JSON*$_G$ contenente l'`id` (*UUID*$_G$ v4 generato localmente dal *backend*$_G$). HTTP 201 Created.

Tabella 14: *Endpoint*$_G$ POST /sim/gateways/{id}/sensors

Rotta	**`GET /sim/gateways/{id}/sensors`**
Descrizione	Restituisce tutti i sensori logici associati al *gateway*$_G$ indicato.
Response	Array di `SensorResponse` *JSON*$_G$. HTTP 200.

Tabella 15: *Endpoint*$_G$ GET /sim/gateways/{id}/sensors

Rotta	**`DELETE /sim/sensors/{sensorId}`**
Descrizione	Elimina permanentemente il sensore dal database SQLite.
Response	HTTP 204 No Content.

Tabella 16: *Endpoint*$_G$ DELETE /sim/sensors/{sensorId}

2.4.3.4. Anomalie

Rotta	**`POST /sim/gateways/{id}/anomaly/network-degradation`**
Descrizione	Inietta una perdita pacchetti temporanea. Il worker applicherà la `packet_loss_pct` prima dell'invio. Se `packet_loss_pct` non è fornito dal client, l'handler applica un default di 0.3 (30%). Se il valore in ingresso è > 1.0, il worker lo normalizza automaticamente dividendolo per 100.
Body Request	`{"duration_seconds": 30, "packet_loss_pct": 0.5}`
Response	HTTP 204 No Content.

Tabella 17: *Endpoint*$_G$ Network Degradation

Rotta	**`POST /sim/gateways/{id}/anomaly/disconnect`**
Descrizione	Simula un down totale di rete. Nessun pacchetto verrà emesso dal *gateway*$_G$ per la durata impostata.
Body Request	`{"duration_seconds": 60}`
Response	HTTP 204 No Content.

Tabella 18: *Endpoint*$_G$ Disconnect

Rotta	**`POST /sim/sensors/{sensorId}/anomaly/outlier`**
Descrizione	Forza il generatore matematico del sensore a restituire il valore esatto fornito per il primissimo sample utile, sovrascrivendo la logica matematica.
Body Request	`{"value": 999.99}`
Response	HTTP 204 No Content.

Tabella 19: *Endpoint*$_G$ Sensor Outlier

2.4.4. Integrazioni Cloud (HTTP e *JetStream*$_G$)

Target / Subject	Tipologia	Responsabilità
`POST /api/provision/onboard`	HTTP *REST*$_G$	Invia il *CSR*$_G$ crittografico al *Provisioning*$_G$ e riceve Certificato X.509 + Chiave AES. Errori HTTP 401/409 sono mappati sul dominio.
`telemetry.data.{tenantId}.{gwId}`	*JetStream*$_G$ Pub	Pubblica la *telemetria*$_G$ in tempo reale cifrata *AES-256*$_G$-*GCM*$_G$.
`command.gw.{tenantId}.{gwId}`	*JetStream*$_G$ Sub	Ricezione asincrona di comandi cloud (config, firmware). Scarta comandi scaduti (> 60s).
`command.ack`$_G$`.{tenantId}.{gwId}`	*JetStream*$_G$ Pub	Invia l'esito (*ACK*$_G$, NACK, Expired) dell'elaborazione di un comando.
`gateway`$_G$`.decommissioned.>`	*JetStream*$_G$ Sub	Ascolta l'eliminazione cloud dei *gateway*$_G$ per innescare un cleanup del db locale.

Tabella 20: Integrazioni di rete Cloud-Simulator

2.4.4.1. Struttura dei *Payload*$_G$ *JetStream*$_G$

I messaggi scambiati sul broker *NATS*$_G$ seguono queste strutture *JSON*$_G$ esatte definite nel dominio:

- ***Telemetria***$_G$ **(`TelemetryEnvelope`)**: `{"gatewayId": "...", "sensorId": "...", "sensorType": "...", "timestamp": "...", "keyVersion": 1, "encryptedData": "...", "iv`$_G$`": "...", "authTag": "..."}`.
- **Comandi in ingresso (`IncomingCommand`)**: Il campo `payload`$_G$ dipende dal `type`. Per `config_update` accetta `send_frequency_ms` e `status`. Per `firmware_push` accetta `firmware_version` e `download_url`.
- **Esito Comandi (`CommandACK`)**: `{"command_id": "...", "status": "ack`$_G$`|nack|expired", "message": "...", "timestamp": "..."}`.

2.4.5. Metriche Operative

Il microservizio espone internamente le proprie metriche in formato *Prometheus*$_G$ sulla porta configurata (`METRICS_ADDR`). L'uso estensivo di `GaugeVec` e `CounterVec` permette il monitoraggio granulare della «salute» di ciascun *gateway*$_G$ simulato e l'identificazione di colli di bottiglia verso *NATS*$_G$.

Nome Metrica	Tipo *Prometheus*$_G$	Descrizione / Help
`notip_sim_gateways_running`	Gauge	Numero di worker (`GatewayWorker`) attualmente in esecuzione.

`notip_sim_buffer_fill_level`	GaugeVec (per `gateway_id`)	Livello attuale di occupazione del canale del `MessageBuffer`.
`notip_sim_buffer_dropped_total`	CounterVec (per `gateway_id`)	Messaggi scartati per saturazione della coda (Drop-Oldest).
`notip_sim_envelopes_published_total`	CounterVec (per `gateway_id`)	Tentativi di pubblicazione *NATS*$_G$ conclusi con successo.
`notip_sim_publish_errors_total`	CounterVec (per `gateway_id`)	Errori intercettati durante la pubblicazione *NATS*$_G$.
`notip_sim_nats_reconnects_total`	CounterVec (per `gateway_id`)	Riconnessioni forzate eseguite dal connettore *NATS*$_G$ *mTLS*$_G$.
`notip_sim_provisioning_success_total`	Counter	*Onboarding*$_G$ completati con successo verso il *Provisioning*$_G$ Cloud.
`notip_sim_provisioning_errors_total`	Counter	Errori o rifiuti durante il processo di *provisioning*$_G$ HTTP.
`notip_sim_anomalies_injected_total`	CounterVec (per `type`)	Anomalie iniettate via API (classificate per tipo di anomalia).

Tabella 21: Metriche *Prometheus*$_G$ esposte dal simulatore

2.4.6. Errori

Errore Dominio	Status HTTP	Causa
`ErrGatewayNotFound`	404	Il *gateway*$_G$ richiesto non esiste nel DB locale o nel registro.
`ErrSensorNotFound`	404	L'*UUID*$_G$ del sensore fornito per l'iniezione dell'anomalia o per l'eliminazione non esiste nel DB locale.
`ErrInvalidFactoryCredentials`	401	Le credenziali fornite sono rifiutate dal *Provisioning*$_G$ Service.
`ErrGatewayAlreadyProvisioned`	409	Tentativo di onboard su un *gateway*$_G$ già attivo nel cloud.
`ErrInvalidSensorRange`	400	Configurazione sensore errata (`MinRange >= MaxRange`).
`ErrGatewayAlreadyRunning`	409	Tentativo di avviare (`/start`) un *gateway*$_G$ il cui worker è già in esecuzione.

Tabella 22: Mappatura Errori di Dominio

2.4.7. Flussi di Esecuzione

Per comprendere l'orchestrazione interna del microservizio, vengono delineati i flussi delle operazioni principali, tracciando le chiamate attraverso gli strati esagonali dell'architettura.

2.4.7.1. Flusso di *Provisioning*$_G$ e Avvio

1. Il client effettua una chiamata `POST /sim/gateways`.
2. Il `GatewayHandler` converte il *payload*$_G$ in *DTO*$_G$ di dominio e invoca il `GatewayRegistry`.
3. Il Registry orchestra l'operazione delegando l'`Onboard` (`ProvisioningServiceClient`).
4. Il Client genera una coppia di chiavi **RSA a 2048 bit** e crea un file ***CSR***$_G$.
5. Il *CSR*$_G$ viene inviato tramite HTTP POST al *Provisioning*$_G$ Service in Cloud.
6. Alla ricezione del certificato X.509 e della **Chiave AES in Base64**, questa viene convertita nell'entità protetta `EncryptionKey`.
7. Il `GatewayStore` SQLite salva l'entità completa (comprese le chiavi *PEM*$_G$ locali).
8. Il Registry crea un nuovo `GatewayWorker`, istanzia una connessione *NATS*$_G$ isolata con i nuovi certificati, e avvia la *goroutine*$_G$ di background.

2.4.7.2. Flusso di Bootstrap e Recovery

1. All'avvio del microservizio, l'applicazione inizializza il `GatewayStore` (SQLite) e il `GatewayRegistry`.
2. Il sistema verifica la variabile di configurazione `RecoveryMode`. Se è `false` (es. in ambienti *CI/CD*$_G$ o di test effimeri), il processo di ripristino viene saltato.
3. Se `RecoveryMode` è `true`, il Registry invoca il metodo `RestoreAll`, interrogando il database locale per estrarre tutti i `Gateway`$_G$ con stato `Provisioned == true` (incluse le chiavi crittografiche e i certificati *PEM*$_G$).
4. Per ogni *gateway*$_G$ recuperato, viene interpellata la tabella per estrarre la configurazione dei sensori logici associati.
5. Il Registry ristabilisce una connessione *NATS*$_G$ *mTLS*$_G$ dedicata per il dispositivo e istanzia nuovamente il relativo `GatewayWorker`.
6. Invocando il metodo `Start`, le *goroutine*$_G$ del worker (`sensorLoop` e il flush del `MessageBuffer`) vengono riavviate. Questo garantisce la resilienza ai crash, permettendo al simulatore di riprendere l'emissione telemetrica in totale isolamento.

2.4.7.3. Flusso di Pubblicazione *Telemetria*$_G$

1. Il `GatewayWorker` esegue ciclicamente operazioni non bloccanti basate sul `time.Ticker` della frequenza configurata.
2. Per ogni `SimSensor` associato, viene invocata l'interfaccia `Generator.Next()` (es. onda sinusoidale o outlier forzato).
3. Il valore generato viene arricchito con l'unità di misura specifica del sensore (`°C` per temperature, `%` per humidity, `hPa` per pressure, `m/s` per movement, `bpm` per biometric). Questo oggetto *JSON*$_G$ interno (es. `{"value": 25.5, "unit": "°C"}`) viene poi passato all'`AESGCMEncryptor`.
4. L'Encryptor genera un ***IV***$_G$ **univoco di 12 byte**, cifra i dati utilizzando il materiale crittografico locale e appende l'AuthTag di validazione.
5. La `TelemetryEnvelope` finale (con campi offuscati in Base64) viene spinta nel `MessageBuffer`.
6. Il *Buffer*$_G$ tenta l'invio a *JetStream*$_G$; in caso di congestione o offline, scarta l'elemento più vecchio in coda per far spazio al nuovo, aggiornando le metriche *Prometheus*$_G$ (`notip_sim_buffer_dropped_total`).

2.4.7.4. Flusso di *Decommissioning*$_G$

1. L'adapter `NATSDecommissionListener` rimane in perenne ascolto sul subject wildcard `gateway`$_G$`.decommissioned.>`.
2. Ricevuto l'evento, estrae *UUID*$_G$ e TenantID dalla rotta *NATS*$_G$.

3. Tramite l'interfaccia (Porta) `DecommissionEventReceiver`, inoltra la richiesta al Core applicativo chiamando il metodo `HandleDecommission`. Il `GatewayRegistry` effettua una validazione di sicurezza: verifica che il `TenantID` ricevuto dal subject *NATS*$_G$ corrisponda esattamente a quello salvato localmente per il *gateway*$_G$ bersaglio; in caso di mismatch, l'evento viene ignorato loggando un warning (slog.Warn).
4. Il `GatewayRegistry`, che implementa tale interfaccia, riceve la chiamata. Acquisisce un lock di scrittura (`RWMutex`), cerca il worker corrispondente, lo arresta inviando un segnale al `context.CancelFunc` e lo disconnette da *NATS*$_G$.
5. I dati persistenti vengono eliminati dallo `Store` locale in SQLite in via definitiva.

2.4.7.5. Flusso di Buffering e Flush dei Comandi durante Anomalia

1. Il `NATSGatewaySubscriber` stabilisce una sottoscrizione *JetStream*$_G$ sul subject `command.gw.{tenantId}.{gwId}`. Per garantire che nessun comando venga perso durante disconnessioni fisiche o riavvii, la sottoscrizione è configurata come **durable push *consumer*$_G$** utilizzando il `durable_name` univoco `gw-{gwId}`, come prescritto dai contratti AsyncAPI. I comandi ricevuti vengono consegnati al `GatewayWorker` tramite il canale `commandCh`.
2. La sottoscrizione *JetStream*$_G$ è configurata con un limite massimo di riconsegna (`MaxDeliver(3)`). Se il canale interno di smistamento verso il `GatewayWorker` risulta saturo, il subscriber esegue esplicitamente un `msg.Nak()` (Negative Acknowledgement). Questo istruisce il broker *NATS*$_G$ a ritentare la consegna in un secondo momento, realizzando un pattern di backpressure nativo senza perdita di comandi.
3. `handleIncomingCommand` verifica se `activeAnomaly != nil`.
4. Se un'anomalia è attiva, il comando viene accodato nell'`anomalyCommandBuffer` (slice locale al worker) senza essere elaborato e senza inviare alcun *ACK*$_G$ al broker.
5. Ad ogni tick del `time.Ticker`, `checkAnomalyExpiry` confronta il timestamp corrente con la scadenza dell'anomalia.
6. Allo scadere dell'anomalia, `activeAnomaly` viene azzerato e `flushBufferedCommands` viene invocato.
7. Per ciascun comando nel *buffer*$_G$, nell'ordine originale di arrivo, viene eseguita l'elaborazione completa e inviato il corrispondente *ACK*$_G$ sul subject `command.ack`$_G$`.{tenantId}.{gwId}`.
8. Il *buffer*$_G$ viene svuotato tramite re-slice a lunghezza zero, conservando la capacità allocata per ottimizzare le future accodamenti.

2.4.8. Ciclo di Vita e Graceful Shutdown

L'applicativo gestisce i segnali del sistema operativo (es. `SIGTERM`) per orchestrare uno spegnimento controllato. Il coordinatore (`app.go`$_G$) impone un timeout di **10 secondi** per lo svuotamento delle connessioni in volo dei server HTTP (API e Metriche) e concede un ulteriore timeout di **5 secondi** ai `GatewayWorker` concorrenti per inviare l'ultimo pacchetto telemetrico prima della chiusura definitiva dei socket *NATS*$_G$.

2.5. Metodologie di Testing

Il microservizio adotta una strategia di testing multi-livello progettata per validare il corretto coordinamento dei worker concorrenti, la rigorosa segregazione dei dati e la validità della generazione matematica. I test di unità si avvalgono di *fake adapter* in memoria per isolare la logica dall'infrastruttura; i test di integrazione utilizzano istanze reali (SQLite temporanei, container *NATS*$_G$ via `testcontainers`$_G$`-go`$_G$).

2.5.1. Test di Unità

Generatori Matematici e Crittografia

Caso di test	Postcondizione verificata
`SineWave` — valori entro i limiti	Tutti i campioni generati ricadono nell'intervallo `[MinRange, MaxRange]` per l'intera durata del ciclo
`Spike` — picco e ritorno	Il valore anomalo viene emesso esattamente una volta; i campioni successivi tornano all'algoritmo di base
`UniformRandom` — distribuzione nei limiti	I valori generati rispettano i bound configurati; nessun campione supera `MaxRange` o scende sotto `MinRange`
`Constant` — valore fisso	Tutti i campioni restituiscono invariabilmente il valore configurato indipendentemente dall'iterazione
Cifratura *AES-256*$_G$*-GCM*$_G$ — struttura *payload*$_G$	Il *payload*$_G$ prodotto contiene un *IV*$_G$ generato casualmente, il ciphertext e l'AuthTag; il campo `keyVersion` è popolato correttamente
Cifratura *AES-256*$_G$*-GCM*$_G$ — chiave non conforme	Una chiave con lunghezza diversa da 32 byte provoca un errore esplicito senza produrre output parziale

Core Applicativo — Worker e Registry

Caso di test	Postcondizione verificata
`GatewayWorker` — ciclo di vita start/stop	Il worker avvia la *goroutine*$_G$ di emissione, risponde ai comandi di stop e termina senza *goroutine*$_G$ in fuga
`GatewayRegistry` — accesso concorrente	Operazioni concorrenti di lettura e scrittura tramite `RWMutex` non producono data race rilevabili con `-race`
`MessageBuffer` — politica Drop-Oldest in overflow	Quando il *buffer*$_G$ è pieno, il messaggio più vecchio viene scartato e il nuovo viene accodato correttamente
`MessageBuffer` — Drop-Oldest su rete assente	In assenza di connettività *NATS*$_G$, il *buffer*$_G$ accumula fino al limite e applica Drop-Oldest senza bloccare
Comandi *JetStream*$_G$ — scarto obsoleti	Un comando con timestamp superiore a 60 secondi viene scartato con esito NACK o Expired senza essere elaborato
Comandi *JetStream*$_G$ — elaborazione valida	Un comando recente viene applicato al worker corretto e l'*ACK*$_G$ viene inviato al broker
Comandi *JetStream*$_G$ — buffering durante anomalia e flush	Un comando ricevuto mentre è attiva un'anomalia viene accodato nell'`anomalyCommandBuffer` senza generare *ACK*$_G$; al termine dell'anomalia il comando viene elaborato e il corrispondente *ACK*$_G$ viene emesso

Handler HTTP e Configurazione

Caso di test	Postcondizione verificata

Handler — validazione input e mapping errori	Input non conformi producono risposte 400; risorse mancanti producono 404; conflitti producono 409; errori interni producono 500
Handler — trasformazione *DTO*$_G$	I campi del dominio vengono serializzati nelle chiavi *JSON*$_G$ contrattualmente attese dal client *CLI*$_G$
Metriche *Prometheus*$_G$ — popolamento isolato	I counter e gli histogram vengono incrementati correttamente senza interferenze tra test paralleli
Configurazione — valori di fallback	Le variabili d'ambiente assenti attivano i default definiti; una variabile obbligatoria mancante causa errore all'avvio

2.5.2. Test di Integrazione

Persistenza e Migrazioni (SQLite)

Caso di test	Infrastruttura	Verifica
Migrazioni — esecuzione e idempotenza	SQLite temporaneo	Le migrazioni vengono applicate correttamente; una seconda esecuzione è idempotente e non altera lo schema
Migrazioni — rimpiazzo schema	SQLite temporaneo	La validazione a basso livello via `PRAGMA table_info` conferma che le colonne attese siano presenti dopo il rimpiazzo
Vincoli relazionali — `ON DELETE CASCADE`	SQLite temporaneo	L'eliminazione di un *gateway*$_G$ rimuove automaticamente tutti i sensori logici associati

Message Brokering (*NATS*$_G$ *JetStream*$_G$)

Caso di test	Infrastruttura	Verifica
Connessione *mTLS*$_G$ al broker	`testcontainers`$_G$-`go`$_G$	La connessione al broker *NATS*$_G$ con certificati generati *on-the-fly* viene stabilita correttamente
Pubblicazione *telemetria*$_G$ cifrata	`testcontainers`$_G$-`go`$_G$	I *payload*$_G$ prodotti dal worker vengono pubblicati sul subject corretto e ricevuti da un subscriber di controllo
Sottoscrizione eventi `decommission`	`testcontainers`$_G$-`go`$_G$	I *payload*$_G$ malformati vengono scartati silenziosamente; i *payload*$_G$ validi avviano il decommission del *gateway*$_G$

***Provisioning*$_G$ e Recovery**

Caso di test	Infrastruttura	Verifica

Provisioning$_G$ — successo	`httptest.Server`	Il flusso completo produce un *gateway*$_G$ provisionato con certificato e chiave AES persistiti nello Store
Provisioning$_G$ — credenziali rifiutate (HTTP 401)	`httptest.Server`	Il client riceve l'errore 401 e lo propaga senza lasciare dati parziali nello Store
`RecoveryMode` — riavvio worker	SQLite temporaneo	Al bootstrap, i *gateway*$_G$ già presenti nello Store vengono estratti e i rispettivi worker vengono riavviati in isolamento

2.5.3. Obiettivi di Copertura Funzionale

Le attività di test automatizzate garantiscono che il microservizio sia verificato almeno rispetto ai seguenti scenari:

- Copertura minima dell'**80%** del codice di dominio e del core applicativo.
- Assenza di data race nel `GatewayRegistry` verificata con il flag `-race` su scenari di accesso concorrente.
- Persistenza atomica tramite transazioni SQLite durante le migrazioni, incluso il caso di interruzione parziale.
- Corretta estrazione e riavvio dei *gateway*$_G$ esistenti in `RecoveryMode` dopo caduta del processo.
- Tolleranza di fronte a *NATS*$_G$ congestionato o assente, con applicazione verificabile della politica Drop-Oldest.

3. Parte II — notip-simulator-*cli*$_G$

3.1. Introduzione

Il `notip-simulator-cli`$_G$ è un binario *Go*$_G$ autonomo (`sim-cli`$_G$) che funge da centro di controllo per il `notip-simulator-backend`$_G$. È progettato per essere eseguito come container *Docker*$_G$ effimero (con esecuzione di singolo comando) o direttamente da terminale tramite shell. La sua unica responsabilità è tradurre comandi leggibili dall'operatore in chiamate HTTP verso le API *REST*$_G$ del *backend*$_G$ simulatore.

La *CLI*$_G$ **non contiene logica di business**. Non persiste stato, non comunica con *NATS*$_G$ e non interagisce con alcun database. Ogni operazione corrisponde a una singola richiesta HTTP (o a una coppia di richieste quando è necessaria la risoluzione dell'*UUID*$_G$ del *gateway*$_G$).

3.2. Dipendenze e Configurazione

3.2.1. Variabili d'ambiente

Campo	Variabile d'ambiente	Default	Note
SimulatorURL	`SIMULATOR_URL`	`http://simulator:8090`	Base URL del *backend*$_G$. Letta una sola volta all'avvio del processo; non viene riletta durante l'esecuzione.

Tabella 23: Variabili d'ambiente del microservizio notip-simulator-*cli*$_G$

3.2.2. Rilevamento TTY

All'avvio, la *CLI*$_G$ verifica se `os.Stdout` è un terminale interattivo tramite `golang`$_G$`.org/x/term.IsTerminal`. Se **non** lo è (output rediretto, *pipeline*$_G$ CI), PTerm, modulo di *Go*$_G$, disabilita globalmente stile e colori, garantendo output pulito e analizzabile da strumenti automatici. Il controllo viene eseguito una sola volta in `cmd.init`$_G$`()` e non è riconfigurabile a runtime.

Questa scelta progettuale ha due implicazioni dirette:

- In ambienti non interattivi, lo spinner animato viene sostituito da un `noopSpinner`, prevenendo la creazione di *goroutine*$_G$ PTerm che causerebbero problemi.
- Le tabelle e i messaggi di output mantengono la stessa struttura sia in modalità TTY che non-TTY, ma senza codici ANSI di colore nella seconda.

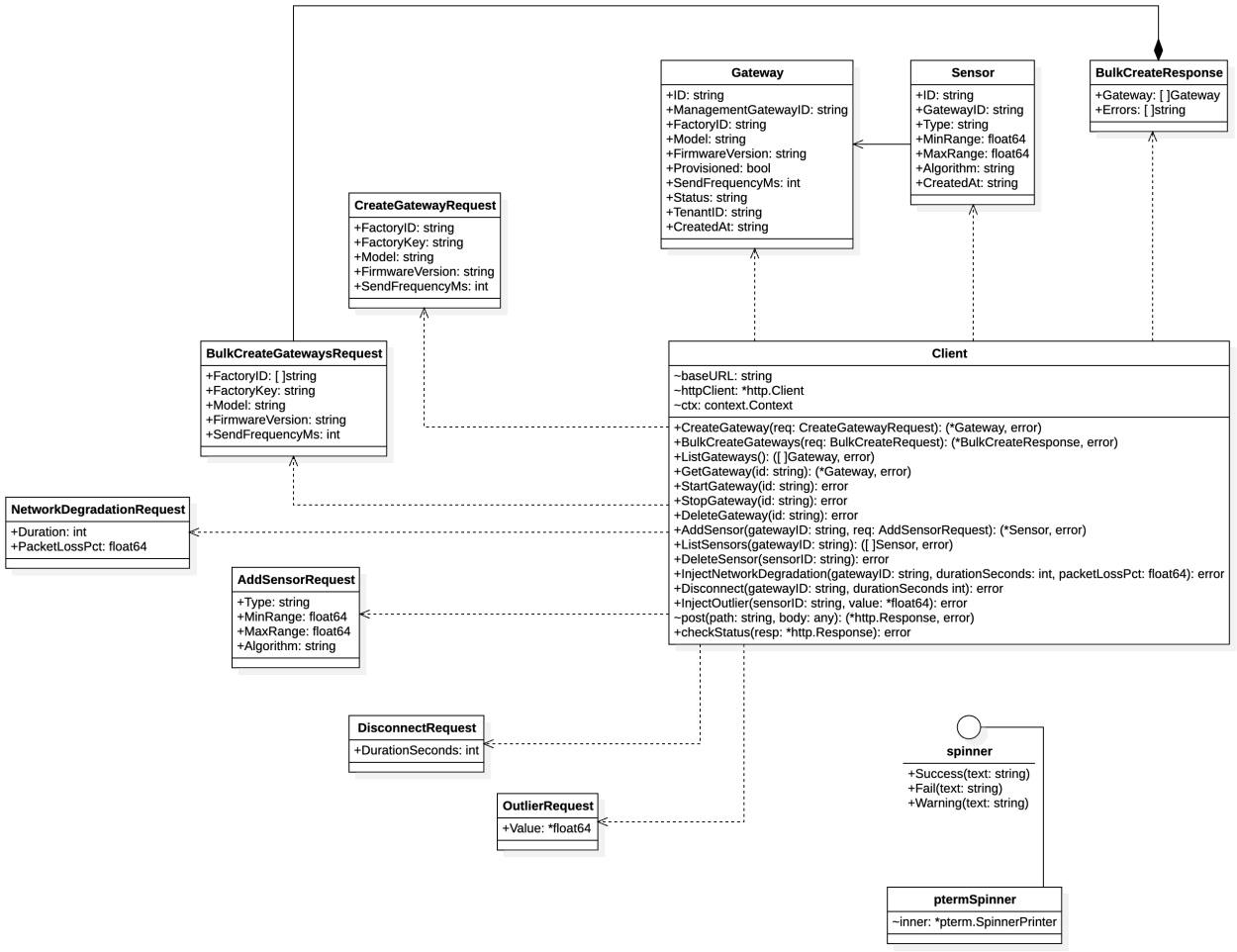
3.3. Scelte Architetturali

Di seguito sono descritte le scelte progettuali principali adottate nella *CLI*$_G$ e la loro motivazione.

Pattern	Motivazione e comportamento
Builder / propagazione del contesto	`Client.WithContext(ctx)` restituisce una copia superficiale del client legata a un nuovo `context.Context`. Ogni comando passa `cmd.Context()` tramite questo metodo, assicurando che la gestione dei segnali di Cobra si propaghi alle richieste HTTP in corso.
Identificatori *UUID*$_G$-first	Tutti gli identificatori di *gateway*$_G$ e sensori sono stringhe *UUID*$_G$ nell'intera API. `resolveGatewayID` chiama `GetGateway(uuid`$_G$`)` e restituisce `gw.ID` (stringa). I comandi sensore accettano *UUID*$_G$ direttamente — non viene eseguito alcun parsing di ID numerici.
Validazione *backend*$_G$-first	La maggior parte dei vincoli numerici e di dominio (es. durata positiva, range packet loss) sono trattati come regole contrattuali del *backend*$_G$. La *CLI*$_G$ impone solo i flag obbligatori e il parsing numerico, senza aggiungere una validazione semantica ampia prima dell'invio delle richieste.

Tabella 24: Pattern architetturali adottati nel notip-simulator-*cli*$_G$

3.4. Architettura Logica



3.4.1. Layout dei Pacchetti

```
notip-simulator-cli_G/
├── main_G.go_G                    Entry point; execute/exit iniettabili per i test
├── main_test.go_G                 Test dell'entry point per il path di errore
├── cmd/
│   ├── root.go_G                  Comando Cobra root, init_G TTY, reset flag,
SIMULATOR_URL
│   ├── gateways.go_G              Sottocomandi gateway_G (list, get, create, bulk, start,
stop, delete)
│   ├── sensors.go_G               Sottocomandi sensore (add, list, delete); risoluzione
UUID_G→ID
│   ├── anomalies.go_G             Sottocomandi anomalia (disconnect, network-
degradation, outlier)
│   ├── shell.go_G                 Modalità REPL interattiva (editor_G di riga su TTY,
fallback bufio)
│   ├── spinner.go_G               Interfaccia spinner, ptermSpinner, noopSpinner,
startSpinner
│   ├── commands_test.go_G         Test di integrazione per tutti i comandi CLI_G (mock
HTTP server)
│   ├── shell_test.go_G            Test della REPL shell
│   ├── spinner_test.go_G          Test dell'astrazione spinner
│   ├── anomalies_test.go_G        Test di validazione flag anomalie
│   └── request_mapping_test.go_G  Test di mapping payload_G richieste
└── internal/
```

```
    └── client/
        ├── client.go_G                Client HTTP, modelli di dominio, tipi richiesta/
risposta
        ├── client_test.go_G           Test unitari del client HTTP (httptest)
        └── request_construction_test.go_G   Test di costruzione delle richieste
```

3.5. Design di Dettaglio

3.5.1. Modelli di Dominio (`internal/client`)

Tutti i tipi sono struct di dati puri con tag *JSON*$_G$. Non sono presenti metodi eccetto su `Client`.

3.5.1.1. `Gateway`$_G$ — value object

Rispecchia il *DTO*$_G$ `GatewayResponse` restituito dal *backend*$_G$ simulatore. `ID` è l'identificatore pubblico del *gateway*$_G$ (stringa *UUID*$_G$) usato in tutti i path API. `ManagementGatewayID` è un *UUID*$_G$ aggiuntivo del piano di management presente su alcuni *backend*$_G$.

Campo	Tipo	Tag *JSON*$_G$	Note
`ID`	`string`	`id`	Identificatore pubblico (*UUID*$_G$); usato in tutti i path *gateway*$_G$ e sensore.
`ManagementGatewayID`	`string`	`managementGatewayId,omitempty`	*UUID*$_G$ del piano di management; opzionale.
`FactoryID`	`string`	`factoryId`	
`Model`	`string`	`model`	
`FirmwareVersion`	`string`	`firmwareVersion`	
`Provisioned`	`bool`	`provisioned`	
`SendFrequencyMs`	`int`	`sendFrequencyMs`	Intervallo di emissione *telemetria*$_G$ in ms.
`Status`	`string`	`status`	Stato runtime (es. `"online"`, `"offline"`).
`TenantID`	`string`	`tenantId`	
`CreatedAt`	`string`	`createdAt`	Stringa ISO-8601; non viene effettuato il parsing.

Tabella 25: Campi della struct *Gateway*$_G$

3.5.1.2. `Sensor` — value object

Rispecchia il *DTO*$_G$ `SensorResponse` restituito dal *backend*$_G$.

Campo	Tipo	Tag *JSON*$_G$	Note
`ID`	`string`	`id`	Identificatore pubblico (*UUID*$_G$); usato nei path di delete sensore e outlier.
`GatewayID`	`string`	`gatewayId`	*UUID*$_G$ del *gateway*$_G$ padre.
`Type`	`string`	`type`	Stringa del tipo di sensore.

`MinRange`	`float64`	`minRange`	
`MaxRange`	`float64`	`maxRange`	
`Algorithm`	`string`	`algorithm`	Algoritmo di generazione dati.
`CreatedAt`	`string`	`createdAt`	Stringa ISO-8601; non viene effettuato il parsing.

Tabella 26: Campi della struct Sensor

3.5.2. Tipi di Richiesta (`internal/client`)

Le struct di richiesta sono serializzate in *JSON*$_G$ e inviate come corpo delle richieste HTTP. I campi marcati `omitempty` vengono omessi dal *payload*$_G$ *JSON*$_G$ quando assumono il valore zero.

3.5.2.1. `CreateGatewayRequest`

Payload$_G$ dedicato all'*endpoint*$_G$ `POST /sim/gateways`.

Campo	Tipo	Tag *JSON*$_G$	Req.	Note
`FactoryID`	`string`	`factoryId`	Sì	
`FactoryKey`	`string`	`factoryKey`	Sì	
`Model`	`string`	`model,omitempty`	No	
`FirmwareVersion`	`string`	`firmwareVersion,omitempty`	No	
`SendFrequencyMs`	`int`	`sendFrequencyMs,omitempty`	No	Default: 1000 ms via flag *CLI*$_G$.

Tabella 27: Campi di CreateGatewayRequest

3.5.2.2. `BulkCreateGatewaysRequest`

Payload$_G$ dedicato all'*endpoint*$_G$ `POST /sim/gateways/bulk`.

Campo	Tipo	Tag *JSON*$_G$	Req.	Note
`FactoryIDs`	`[]string`	`factoryIds`	Sì	
`FactoryKey`	`string`	`factoryKey`	Sì	
`Model`	`string`	`model,omitempty`	No	
`FirmwareVersion`	`string`	`firmwareVersion,omitempty`	No	
`SendFrequencyMs`	`int`	`sendFrequencyMs,omitempty`	No	Default: 1000 ms via flag *CLI*$_G$.

Tabella 28: Campi di BulkCreateGatewaysRequest

3.5.2.3. `BulkCreateResponse`

Risposta di `POST /sim/gateways/bulk`. HTTP 201 indica successo totale; HTTP 207 indica errori parziali.

Campo	Tipo	Tag *JSON*$_G$	Note
`Gateways`	`[]Gateway`$_G$	`gateways`	*Gateway*$_G$ creati con successo.
`Errors`	`[]string`	`errors`	Slice parallela: stringa vuota all'indice i significa che il *gateway*$_G$ i è stato creato con successo.

Tabella 29: Campi di BulkCreateResponse

3.5.2.4. `AddSensorRequest`

Payload$_G$ dedicato all'*endpoint*$_G$ `POST /sim/gateways/{id}/sensors`. Il parametro `{id}` è l'*UUID*$_G$ del *gateway*$_G$.

Campo	Tipo	Tag *JSON*$_G$	Req.	Note
`Type`	`string`	`type`	Sì	`temperature`, `humidity`, `pressure`, `movement`, `biometric`.
`MinRange`	`float64`	`minRange`	Sì	
`MaxRange`	`float64`	`maxRange`	Sì	
`Algorithm`	`string`	`algorithm`	Sì	`uniform_random`, `sine_wave`, `spike`, `constant`.

Tabella 30: Campi di AddSensorRequest

3.5.2.5. `NetworkDegradationRequest`

Payload$_G$ dedicato all'*endpoint*$_G$ `POST /sim/gateways/{id}/anomaly/network-degradation`.

Campo	Tipo	Tag *JSON*$_G$	Note
`DurationSeconds`	`int`	`duration_seconds`	Obbligatorio da policy flag *CLI*$_G$; la *CLI*$_G$ non impone `> 0`, il *backend*$_G$ valida la semantica.
`PacketLossPct`	`float64`	`packet_loss_pct,omitempty`	Frazione 0–1; omesso quando 0; il *backend*$_G$ applica il default 0.3.

Tabella 31: Campi di NetworkDegradationRequest

3.5.2.6. `DisconnectRequest`

Payload$_G$ dedicato all'*endpoint*$_G$ `POST /sim/gateways/{id}/anomaly/disconnect`.

Campo	Tipo	Tag *JSON*$_G$	Note
`DurationSeconds`	`int`	`duration_seconds`	Obbligatorio da policy flag *CLI*$_G$; il *backend*$_G$ valida la semantica.

Tabella 32: Campi di DisconnectRequest

3.5.2.7. `OutlierRequest`

Payload$_G$ dedicato all'*endpoint*$_G$ `POST /sim/sensors/{sensorId}/anomaly/outlier`.

Campo	Tipo	Tag *JSON*$_G$	Note
`Value`	`*float64`	`value,omitempty`	Puntatore opzionale; omesso quando nil. Il campo è assente dal *JSON*$_G$ quando il flag `--value` non è stato esplicitamente impostato; il *backend*$_G$ applica il proprio fallback.

Tabella 33: Campi di OutlierRequest

3.5.3. HTTP Client (`internal/client`)

3.5.3.1. Struct `Client`

Il singolo client HTTP per tutte le interazioni con il *backend*$_G$ simulatore. Mantiene una base URL, un `*http.Client` standard con timeout fisso di 30 secondi e un `context.Context` per la cancellazione per-richiesta.

Campo	Tipo	Note
`baseURL`	`string`	Impostato alla costruzione; mai mutato.
`httpClient`	`*http.Client`	Condiviso tra tutti i metodi; timeout 30 s.
`ctx`	`context.Context`	Contesto per-richiesta; sostituito da `WithContext`.

Tabella 34: Campi della struct Client

Le costanti interne definiscono i prefissi dei path: `pathGateways` (`"/sim/gateways/"`), `pathSensors` (`"/sim/sensors/"`), `defaultTimeout` (30 s).

Il costruttore `New(baseURL string) *Client` crea un `Client` con un nuovo `*http.Client` (timeout 30 s) e `context.Background()`. Il metodo `WithContext(ctx context.Context) *Client` restituisce una copia superficiale con `ctx` impostato; se `ctx` è nil, ricade su `context.Background()`.

3.5.3.2. Metodi *Gateway*$_G$

Metodo	HTTP / Path	Body	Ritorna
`ListGateways`	GET `/sim/gateways`	—	`([]Gateway`$_G$`, error)`
`GetGateway`	GET `/sim/gateways/{uuid`$_G$`}`	—	`(*Gateway`$_G$`, error)`
`CreateGateway`	POST `/sim/gateways`	`CreateGatewayRequest`	`(*Gateway`$_G$`, error)`
`BulkCreateGateways`	POST `/sim/gateways/bulk`	`BulkCreateGatewaysRequest`	`(*BulkCreateResponse, error)`
`StartGateway`	POST `/sim/gateways/{uuid`$_G$`}/start`	nessuno	`error`
`StopGateway`	POST `/sim/gateways/{uuid`$_G$`}/stop`	nessuno	`error`
`DeleteGateway`	DELETE `/sim/gateways/{uuid`$_G$`}`	—	`error`

Tabella 35: Metodi HTTP del Client per i *Gateway*$_G$

3.5.3.3. Metodi Sensori

Metodo	HTTP / Path	Note
`AddSensor(gatewayID, req)`	POST `/sim/gateways/{gatewayID}/sensors`	Usa *UUID*$_G$ *gateway*$_G$.
`ListSensors(gatewayID)`	GET `/sim/gateways/{gatewayID}/sensors`	Usa *UUID*$_G$ *gateway*$_G$.
`DeleteSensor(sensorID)`	DELETE `/sim/sensors/{sensorID}`	Usa *UUID*$_G$ sensore.

Tabella 36: Metodi HTTP del Client per i Sensori

3.5.3.4. Metodi Anomalie

Metodo	HTTP / Path	Note
`InjectNetworkDegradation(gatewayID, durationSeconds, packetLossPct)`	POST `/sim/gateways/{uuid`$_G$`}/anomaly/network-degradation`	Usa *UUID*$_G$ *gateway*$_G$.
`Disconnect(gatewayID, durationSeconds)`	POST `/sim/gateways/{uuid`$_G$`}/anomaly/disconnect`	Usa *UUID*$_G$ *gateway*$_G$.
`InjectOutlier(sensorID, value *float64)`	POST `/sim/sensors/{sensorID}/anomaly/outlier`	Usa *UUID*$_G$ sensore.

Tabella 37: Metodi HTTP del Client per le Anomalie

3.5.3.5. Helper Privati

Il metodo privato `post(path string, body any) (*http.Response, error)` serializza il `body` in *JSON*$_G$ se non nil, imposta `Content-Type: application/json`$_G$ e usa `http.NewRequestWithContext` con `c.ctx`. La funzione `checkStatus(resp *http.Response) error` permette di restituire nil per i codici 2xx; per qualsiasi altro codice legge il corpo della risposta e restituisce un errore formattato.

3.5.4. Strato dei Comandi (`cmd`)

I comandi della *CLI*$_G$ sono implementati come variabili di pacchetto non esportate di tipo `*cobra.Command` e registrate tramite le funzioni `init`$_G$`()`. L'architettura di questo strato è **stateless**: ogni singola invocazione di un comando istanzia un nuovo `*client.Client`, evitando qualsiasi condivisione di stato tra le esecuzioni.

3.5.4.1. Comando Radice (`root`)

La variabile `rootCmd` rappresenta il punto di ingresso principale dell'applicativo (`sim-cli`$_G$). La sua configurazione e il suo ciclo di vita sono gestiti come segue:

- Inizializzazione: La variabile di pacchetto `simulatorURL` viene popolata leggendo la variabile d'ambiente `SIMULATOR_URL` durante la fase di `init`$_G$`()`.
- Esecuzione: La funzione esportata `Execute() error` agisce da wrapper per `rootCmd.Execute()` e viene invocata direttamente dall'entry point nel `main`$_G$`()`.
- Gestione dello Stato: La funzione `resetAllCommandFlags(c *cobra.Command)` percorre ricorsivamente l'albero dei comandi per ripristinare ogni flag al proprio valore di default dichiarato, forzando lo stato `Changed = false`. Questo meccanismo di pulizia è essenziale e obbligatorio prima di ogni iterazione della REPL shell, in quanto previene la contaminazione dei parametri tra un comando e il successivo.

3.5.4.2. Sottocomandi `gateways`

Il comando padre `gatewaysCmd` (invocato come `sim-cli`$_G$ `gateways`) funge da raggruppamento logico per sette sottocomandi operativi. L'intera configurazione e registrazione di questo gruppo avviene all'interno della funzione `init`$_G$`()` del file `cmd/gateways.go`$_G$.

Sottocomando	Argomenti / Flag	Comportamento
`gateways list`	Nessun flag.	Elenca tutti i *gateway*$_G$ come tabella PTerm con colonne: ID, *UUID*$_G$, Status

		(color-coded), Model, Freq (ms), *Tenant*$_G$.
`gateways get <uuid`$_G$`>`	1 argomento posizionale: *UUID*$_G$ *gateway*$_G$.	Mostra una tabella chiave–valore verticale con: ID, *UUID*$_G$, Factory ID, Model, Firmware, Status, Provisioned, Send Freq (ms), *Tenant*$_G$, Created At.
`gateways create`	`--factory-id` (req.), `--factory-key` (req.), `--model` (req.), `--firmware` (req.), `--freq` int default 1000 (req.).	In caso di successo, mostra il *gateway*$_G$ creato come tabella.
`gateways bulk`	`--factory-id` (req., ripetibile — un'occorrenza per *gateway*$_G$), `--factory-key` (req.), `--model` (req.), `--firmware` (req.), `--freq` int default 1000 (req.).	HTTP 207 (parziale) non è un errore a livello comando: viene mostrato uno stato `Warning` con il conteggio dei fallimenti; i *gateway*$_G$ creati con successo vengono comunque renderizzati.
`gateways start <uuid`$_G$`>`	1 argomento posizionale: *UUID*$_G$ *gateway*$_G$.	Avvia il worker del *gateway*$_G$.
`gateways stop <uuid`$_G$`>`	1 argomento posizionale: *UUID*$_G$ *gateway*$_G$.	Arresta il worker del *gateway*$_G$.
`gateways delete <uuid`$_G$`>`	1 argomento posizionale: *UUID*$_G$ *gateway*$_G$.	Rimuove permanentemente il *gateway*$_G$ e i suoi sensori.

Tabella 38: Sottocomandi gateways e loro flag

3.5.4.3. Sottocomandi `sensors`

Il comando padre `sensorsCmd` (invocato come `sim-cli`$_G$ `sensors`) gestisce le operazioni relative ai sensori ed è composto da tre sottocomandi registrati nel file `cmd/sensors.go`$_G$ durante la fase di `init`$_G$`()`.

Un aspetto implementativo rilevante di questo gruppo è la validazione preventiva degli identificatori. Per garantire la coerenza delle operazioni, viene utilizzata la funzione di supporto `resolveGatewayID(c *client.Client, input string) (string, error)`. Tale funzione invoca `c.GetGateway(input)` e ne restituisce il campo `gw.ID`; questo comporta l'esecuzione deliberata di una richiesta HTTP aggiuntiva verso il *backend*$_G$ al solo scopo di validare l'esistenza e la correttezza dell'*UUID*$_G$ del *gateway*$_G$ prima di procedere con la logica del comando.

Sottocomando	Argomenti / Flag	Comportamento
`sensors add <gateway`$_G$`-uuid`$_G$`>`	`--type` (req.), `--min` float64 default 0 (req.), `--max` float64 default 100 (req.), `--algorithm` (req.).	Chiama sempre prima `GetGateway` per validare e risolvere l'ID *gateway*$_G$ (2 richieste HTTP totali).
`sensors list <gateway`$_G$`-uuid`$_G$`>`	1 argomento posizionale.	Chiama `resolveGatewayID` (1 richiesta `GetGateway` aggiuntiva). Mostra tabella PTerm con colonne: ID, *UUID*$_G$, Type, Min, Max, Algorithm.

`sensors delete <sensor-uuid`$_G$`>`	1 argomento posizionale: *UUID*$_G$ sensore.	Nessuna validazione oltre la presenza dell'argomento.

Tabella 39: Sottocomandi sensors e loro flag

3.5.4.4. Sottocomandi `anomalies`

Il comando padre `anomaliesCmd` (invocato come `sim-cli`$_G$ `anomalies`) costituisce il raggruppamento logico per tutte le operazioni di iniezione di guasti e manipolazione del comportamento simulato (come disconnessioni forzate, degrado della rete e generazione di valori *outlier*). I tre sottocomandi che lo compongono sono registrati e configurati all'interno della funzione `init`$_G$`()` del file `cmd/anomalies.go`$_G$.

Sottocomando	Argomenti / Flag	Comportamento
`anomalies disconnect <gateway`$_G$`-uuid`$_G$`>`	`--duration` int default 0 (req.).	Il *backend*$_G$ impone la validità semantica della durata.
`anomalies network-degradation <gateway`$_G$`-uuid`$_G$`>`	`--duration` int default 0 (req.), `--packet-loss` float64 default 0 (opz.).	Quando `--packet-loss` non è fornito (`Changed == false`), il valore 0 viene passato a `InjectNetworkDegradation`; il tag `omitempty` lo esclude dal *JSON*$_G$ e il *backend*$_G$ applica il proprio default di 0.3.
`anomalies outlier <sensor-uuid`$_G$`>`	`--value` float64 default 0 (opz.).	La presenza del flag è controllata esplicitamente via `cmd.Flags().Changed("value")`. Solo quando il flag è stato impostato esplicitamente, il valore è convertito in `*float64` e incluso nella richiesta. Quando omesso, `OutlierRequest.Value` è nil e il campo è assente dal *JSON*$_G$.

Tabella 40: Sottocomandi anomalies e loro flag

3.5.4.5. Comando `shell`

Il comando `sim-cli`$_G$ `shell` avvia una sessione interattiva, consentendo l'esecuzione sequenziale di molteplici comandi senza la necessità di riavviare il processo o il container ospite. Il ciclo di vita della sessione si articola nelle seguenti fasi:

1. Inizializzazione visiva: Viene stampato un banner di benvenuto. In modalità *styled* (interattiva) utilizza i componenti `BigText` e `DefaultBox` della libreria PTerm. In modalità *raw-output*, il rendering grafico del `BigText` viene omesso per mantenere l'output pulito, preservando unicamente il testo informativo.
2. Selezione della modalità di input: Viene effettuato un controllo su `stdin` e `stdout`. Se l'ambiente è interattivo, la *CLI*$_G$ entra in modalità *raw* e istanzia `term.NewTerminal` come *editor*$_G$ di riga, offrendo un'esperienza utente avanzata (inclusa la cronologia e l'editing dei comandi). In caso contrario, effettua un fallback su un approccio standard leggendo gli stream riga per riga tramite `bufio.Reader`.
3. Loop di elaborazione: Per ogni riga in ingresso non vuota, viene applicata la seguente logica:

- I token `exit` o `quit` innescano la terminazione pulita (stampando «Goodbye!») e restituiscono `nil`.
- Il token `shell` (come primo comando) viene intercettato ed emette un avviso, bloccando attivamente l'annidamento non voluto di sessioni REPL multiple.
- Per qualsiasi altro input, la shell effettua la pulizia dello stato chiamando `resetAllCommandFlags(rootCmd)`, applica i nuovi parametri tramite `rootCmd.SetArgs(args)` e delega l'azione a `rootCmd.Execute()`. Eventuali errori restituiti dai sottocomandi vengono intercettati e stampati a schermo tramite `pterm.Error.Println`, ma **non** causano la terminazione della sessione.

4. Gestione terminazione: La ricezione di un segnale `io.EOF` (in qualsiasi modalità di input) conclude la sessione in modo controllato.

3.5.4.6. Interfaccia `spinner`

L'astrazione del feedback visivo di caricamento è definita all'interno di `cmd/spinner.go`$_G$ ed è utilizzata uniformemente da tutti gli handler dei comandi. Tale scelta architetturale disaccoppia l'esecuzione logica dal rendering grafico a terminale.

Tipo	Metodi	Comportamento
Interfaccia `spinner`	`Success(text)`, `Fail(text)`, `Warning(text)`	Contratto comune per i due adattatori.
`noopSpinner`	Tutti no-op.	Usato in raw-output mode e nei test per evitare la creazione di *goroutine*$_G$ PTerm.
`ptermSpinner`	Delega a `*pterm.SpinnerPrinter`.	Gestisce il caso nil di `inner` (PTerm può restituire nil su fallimento di `Start`).

Tabella 41: Interfaccia spinner e implementazioni

La factory `startSpinner(text string) spinner` restituisce `noopSpinner` quando `pterm.RawOutput` è true; altrimenti avvia uno spinner PTerm e restituisce `ptermSpinner`.

3.5.4.7. Funzioni di supporto

Funzione	Comportamento
`mustMarkRequired(cmd, flagName)`	Chiama `cmd.MarkFlagRequired`; in caso di errore scrive su stderr e chiama `exitProcess(1)`. `exitProcess` è una variabile di pacchetto (default `os.Exit`) per permettere l'iniezione nei test.
`statusStyle(status string) string`	Restituisce lo status avvolto in verde PTerm per `"online"`/`"connected"`, rosso per `"offline"`/`"disconnected"`, stringa plain altrimenti. No-op in raw-output mode.
`gatewayUUID(gw Gateway`$_G$`) string`	Restituisce `gw.ManagementGatewayID` se non vuoto; altrimenti ricade su `gw.ID`. Usato da `gateways list`, `gateways get` e `printGatewayTable` per popolare la colonna *UUID*$_G$.
`printGatewayTable(gateways)`	Mostra una tabella PTerm con colonne: ID, *UUID*$_G$, Status, Model, Freq (ms). No-op su slice vuota. A differenza di `gateways list`, questo helper non include la colonna *Tenant*$_G$.

`printSensorTable(sensors)`	Mostra una tabella PTerm con colonne: ID, *UUID*$_G$, Type, Min, Max, Algorithm. No-op su slice vuota. Entrambe le colonne ID e *UUID*$_G$ mostrano `Sensor.ID` (la stringa *UUID*$_G$), poiché nel modello attuale non esiste una chiave numerica separata.

Tabella 42: Funzioni di supporto nel package cmd

3.5.5. Entry Point (`main`$_G$`.go`$_G$)

Variabile	Tipo	Default	Scopo
`execute`	`func() error`	`cmd.Execute`	Iniettabile per i test; `main`$_G$ chiama `execute()`.
`osExit`	`func(int)`	`os.Exit`	Iniettabile per i test; chiamata con codice 1 in caso di errore.

Tabella 43: Variabili iniettabili dell'entry point

`main`$_G$`()` chiama `execute()` e chiama `osExit(1)` se restituisce un errore non nil. L'iniettabilità permette a `main_test.go`$_G$ di verificare il comportamento del path di errore senza avviare un sottoprocesso.

3.6. Metodologie di Testing

Il binario *CLI*$_G$ è verificato attraverso una suite di test automatizzati suddivisa in test di unità (focalizzati sui componenti UI e sul client HTTP) e test di integrazione (focalizzati sull'albero dei comandi e sulla traduzione dei flag in *payload*$_G$ HTTP).

Durante l'esecuzione dei test (`TestMain`), l'output formattato e i colori vengono disabilitati globalmente (`pterm.DisableOutput()`, `pterm.DisableStyling()`). Questo forza la modalità `RawOutput`, garantendo che i componenti visivi (come lo spinner animato) vengano istanziati come oggetti `noop` (no-operation).

3.6.1. Test di Unità

Le dipendenze esterne non sono presenti. I test dell'interfaccia utente (UI) usano mock dell'`os.Stdout` e pattern di iniezione delle funzioni, mentre i test del client HTTP isolano la logica di serializzazione e deserializzazione.

Componenti UI e Utility (`cmd`)

Caso di test	Postcondizione verificata
`mustMarkRequired` — successo	Il flag specificato viene correttamente annotato come obbligatorio in Cobra
`mustMarkRequired` — flag mancante	Scatta l'errore e viene invocata la funzione iniettata `exitProcess(1)` (comportamento di `os.Exit`)
`startSpinner` in `RawOutput` mode	Restituisce un'istanza `noopSpinner`; le chiamate a `Success`, `Fail` e `Warning` non generano side effect
`ptermSpinner` con istanza `inner` nil	Le chiamate ai metodi di completamento non causano panic (safety check)
`printPrompt` / `printWelcomeBanner`	In modalità raw stampano output plain text scansionabile; in modalità styled usano i widget grafici PTerm

`printWelcomeBanner` — errore di render	Se il renderer PTerm restituisce un errore, il banner viene silenziosamente omesso senza causare panic

Shell con Line-*Editor*$_G$ (`cmd`)

Caso di test	Postcondizione verificata
`RunShell` con line-*editor*$_G$ — EOF	Alla fine dello stream di input il loop REPL termina in modo pulito senza panic
`RunShell` con line-*editor*$_G$ — errore di lettura	Un errore `Read` sul terminale viene propagato e il comando REPL restituisce un errore non nil
`RunShell` con line-*editor*$_G$ — errore di ripristino terminale	Se `Restore` fallisce dopo `MakeRaw`, il comando restituisce comunque l'errore di ripristino
`RunShell` con line-*editor*$_G$ — errore `MakeRaw` tra comandi	Se `MakeRaw` fallisce dopo l'esecuzione di un sottocomando, l'errore viene propagato e la shell si arresta
`shell` — esecuzione con line-*editor*$_G$ attivo	Il comando letto dal line-*editor*$_G$ viene eseguito correttamente da Cobra
`shell` — fallback a reader classico se line-*editor*$_G$ non disponibile	Se `MakeRaw` fallisce (modalità raw non disponibile), `runShellWithLineEditor` restituisce un errore e la shell passa automaticamente alla modalità `bufio.Reader`
`shell` — copertura delle funzioni iniettabili di default	Le funzioni iniettabili di default (`shellStdout`, `shellNewEditor`) restituiscono valori non nil senza errori

Client HTTP e Costruzione Richieste (`internal/client`)

Caso di test	Postcondizione verificata
`CreateGateway` / `AddSensor` — serializzazione base	Il *JSON*$_G$ risultante ha le chiavi previste contrattualmente dal *backend*$_G$ (es. `duration_seconds` e non `duration`; `minRange` e non `min`)
`CreateGateway` — omitempty su campi opzionali	Se `Model`, `FirmwareVersion` o `SendFrequencyMs` sono zero-valued, le relative chiavi sono assenti dal *JSON*$_G$
`InjectNetworkDegradation` — default `packet_loss_pct`	Se il valore è 0, la chiave *JSON*$_G$ viene omessa per demandare l'applicazione del default (`0.3`) al *backend*$_G$
`InjectOutlier` — puntatore nil per valore omesso	Se il puntatore `value` è `nil`, la chiave scompare dal *JSON*$_G$
Risposta con *JSON*$_G$ malformato (es. decodifica lista *gateway*$_G$)	Restituisce un errore di decodifica invece di terminare brutalmente
Risposta HTTP 400 (Bad Request)	L'errore restituito include esplicitamente sia lo status code che il corpo testuale inviato dal *backend*$_G$ (es. errori di validazione ID)
Gestione HTTP 404 (Not Found)	I metodi restituiscono un errore mappato per mancata risorsa, non mascherato da unparseable struct

`ListGateways` — URL base non valido	Un URL su cui non può essere fatto il parsing passato al client produce un errore di costruzione della richiesta, senza panic
`AddSensor` — errore di encoding *payload*$_G$	Se la serializzazione *JSON*$_G$ del *payload*$_G$ fallisce, viene restituito un errore prima di contattare il server
`WithContext` — `nil` come argomento	Il client usa automaticamente `context.Background()` come fallback senza panic
`WithContext` — context cancellato	La richiesta HTTP in-flight viene interrotta e viene restituito un errore di cancellazione

3.6.2. Test di Integrazione

I test di integrazione eseguono l'intera *pipeline*$_G$ della *CLI*$_G$: dal parsing della linea di comando (`Cobra`) alla generazione delle richieste verso un server HTTP locale (`httptest.Server`) istanziato per l'occasione e assegnato alla variabile `simulatorURL`. La funzione di supporto `resetAllFlags(rootCmd)` assicura che lo stato sticky dei flag di Cobra venga pulito prima di ogni esecuzione, evitando inquinamento tra i test.

Mapping dei Comandi e *Payload*$_G$

Questi test verificano che la combinazione dei flag da linea di comando generi l'esatto *payload*$_G$ *JSON*$_G$ previsto dal *backend*$_G$.

Caso di test	Infrastruttura	Verifica
`gateways create` — tutti i flag valorizzati	`httptest`	Il mock riceve una POST `/sim/gateways` con `Content-Type: application/json`$_G$ e i campi `factoryId`, `model`, ecc., mappati correttamente
`sensors add` — traduzione nomi dei flag	`httptest`	I flag *CLI*$_G$ `--min` e `--max` generano le chiavi *JSON*$_G$ esatte `minRange` e `maxRange`
`anomalies network-degradation` — omissione flag opzionale	`httptest`	Senza il flag `--packet-loss`, la chiave non è presente nel body *JSON*$_G$ generato

Esecuzione Comandi *Gateway*$_G$ e Sensor

Caso di test	Infrastruttura	Verifica
`gateways bulk` — successo parziale (HTTP 207)	`httptest`	Il comando gestisce il 207 senza uscire con codice di errore (exit 1), parsando la slice di `errors` mista e stampando warning e successi assieme
`gateways create` — flag `--model` mancante	Cobra (Locale)	Il mock server non viene mai invocato; Cobra rileva la violazione e blocca l'esecuzione ritornando errore

`sensors add` — identificatore *gateway*$_G$ non valido (NaN)	`httptest`	La pre-validazione `GetGateway` via HTTP fallisce (404); il comando si arresta senza mai chiamare l'*endpoint*$_G$ POST del sensore
`sensors list` — errore durante il lookup *gateway*$_G$	`httptest`	Se la risoluzione *UUID*$_G$/ID del *gateway*$_G$ fallisce, il comando restituisce l'errore senza invocare l'*endpoint*$_G$ della lista sensori
`sensors add` — fallimento dopo lookup *gateway*$_G$ riuscito	`httptest`	Se il lookup del *gateway*$_G$ ha successo ma la POST del sensore ritorna un errore server, il comando lo propaga correttamente
`gateways get` — *gateway*$_G$ inesistente (404)	`httptest`	Il comando rileva l'errore HTTP 404 e restituisce un log informativo di errore per l'operatore

Esecuzione Shell

Caso di test	Infrastruttura	Verifica
`shell` — invocazione comando `quit` / `exit`	`os.Pipe` (stdin)	Il loop di elaborazione si interrompe e il comando ritorna `nil` in modo pulito chiudendo la *CLI*$_G$
`shell` — ricezione segnale `io.EOF`	`os.Pipe` (stdin chiuso)	Il loop rileva la fine dello stream e termina gracefully senza panics
`shell` — tentativo di annidamento (comando `shell`)	`os.Pipe` (stdin)	Il loop intercetta il token `shell`, stampa un avviso e previene l'apertura annidata di un nuovo *editor*$_G$
`shell` — comando interno fallito (es. server error 500)	`httptest` + `os.Pipe`	L'errore del comando (`gateways list`) viene intercettato da `pterm.Error` e stampato, **senza** causare l'interruzione della sessione di shell interattiva
`anomalies outlier` — successo senza flag `--value`	`httptest`	L'outlier viene iniettato omettendo la chiave `value` dal *JSON*$_G$; il *backend*$_G$ applica il valore di default
`gateways list` — tabella vuota senza output	Locale	Se la slice dei *gateway*$_G$ è vuota, la funzione di rendering non produce righe su stdout
`sensors list` — tabella vuota senza output	Locale	Se la slice dei sensori è vuota, la funzione di rendering non produce righe su stdout

Risoluzione *UUID*$_G$ *gateway*$_G$ (`gatewayUUID`)	`httptest`	Dato un identificativo numerico, il helper risolve e restituisce l'*UUID*$_G$ corretto del *gateway*$_G$ tramite GET

3.6.3. Obiettivi di copertura funzionale

Le attività di test automatizzate garantiscono che il binario *CLI*$_G$ sia verificato almeno rispetto ai seguenti scenari:

- Verifica che ogni sottocomando produca la richiesta HTTP corretta (metodo, path, body) verso il *backend*$_G$.
- Corretta gestione di HTTP 207 nel comando `gateways bulk`: i *gateway*$_G$ creati con successo vengono renderizzati e i fallimenti parziali generano un warning senza interrompere l'esecuzione.
- Comportamento della REPL shell in modalità TTY e non-TTY: corretto funzionamento del line *editor*$_G$, gestione di `io.EOF`, prevenzione dell'annidamento shell.
- Verifica che i flag sticky tra iterazioni REPL siano eliminati da `resetAllCommandFlags`.
- Verifica che in ambienti non-TTY (`pterm.RawOutput = true`) nessuna *goroutine*$_G$ spinner venga creata, garantendo assenza di data race rilevabili con `-race`.
- Propagazione corretta del `context.Context` di Cobra alle richieste HTTP in-flight per la gestione della cancellazione via segnale.