Smart irrigation leveraging Computing Continuum

SIMONE GUIDATO, Università degli studi di Roma Tor Vergata, Italia
LEONARDO BECCARINI, Università degli studi di Roma Tor Vergata, Italia

Progettiamo, implementiamo e deployiamo un sistema di smart irrigation su **computing continuum** (Edge–Fog–Cloud) composto da microservizi accoppiati tramite **MQTT** e **HTTP/gRPC**. I dati grezzi dei sensori vengono aggregati sull'Edge, le decisioni di irrigazione sono calcolate sul Fog integrando ET_o **(Hargreaves–Samani)** e previsioni meteo, mentre sul Cloud eseguiamo persistenza time-series e composizione per la Dashboard. Per i flussi critici adottiamo **QoS 1** con idempotenza applicativa (dedup a TTL) per ottenere un effetto exactly-once pur in presenza di semantica at-least-once. La topologia è orchestrata con **Kubernetes/Minikube** (label per ruolo e tc netem per latenza controllata) e con **AWS Elastic Beanstalk** per i servizi Cloud.

ACM Reference Format:
Simone Guidato and Leonardo Beccarini. 2025. Smart irrigation leveraging Computing Continuum . 1, 1 (October 2025), 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduzione

Il *computing continuum* distribuisce l'elaborazione tra **Edge–Fog–Cloud** per ridurre latenza e traffico, contenere i costi e preservare la località dei dati: vicino alle sorgenti si filtra e si reagisce, nel cloud si storicizza e si presenta. Nel nostro caso di agricoltura 3.0, i *sensor simulator* generano misure di umidità aggregate all'**Edge**. Abbiamo supposto che ad ogni sensore corrispondesse anche un attuatore che gestisce l'irrigazione.
Il **Fog** esegue un *irrigation controller* che integra misure e meteo per stimare il budget idrico e decidere *quando* e *quanto* irrigare, attivando gli attuatori via gRPC e chiudendo il ciclo con i feedback; il **Cloud** offre persistenza time–series, un modello eventi unificato e un *gateway* per la dashboard. La messaggistica MQTT (QoS mirato + idempotenza applicativa) e il deployment riproducibile su Kubernetes/Minikube ed Elastic Beanstalk rendono osservabili e misurabili gli effetti della latenza lungo l'intera filiera.

2 Architettura

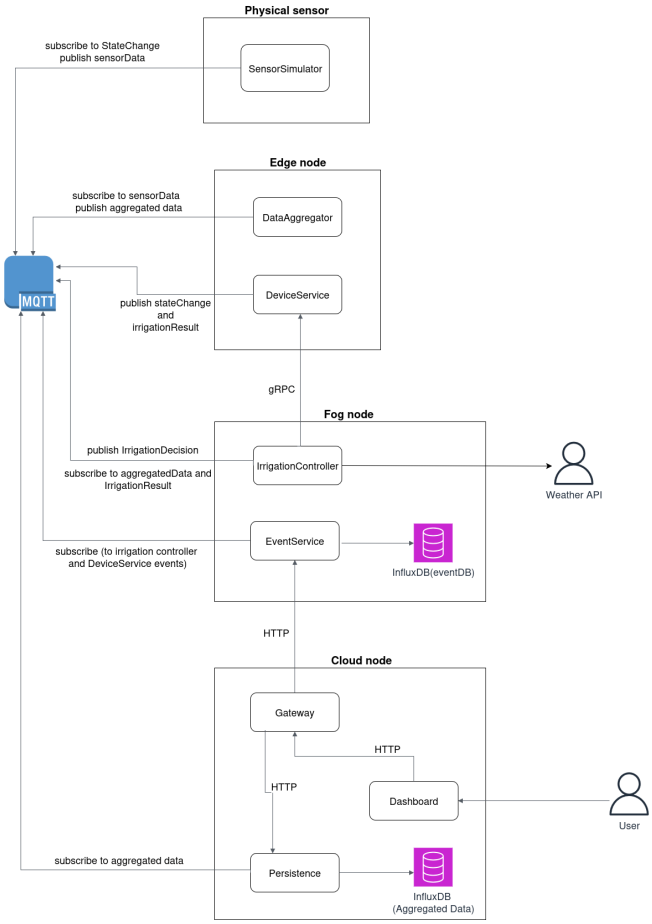

Fig. 1. Diagramma architetturale

Il sistema è composto da 7 microservizi più il simulatore dei sensori IoT, così ripartiti:

- **Physical Sensor**:
  (1) sensor simulator
- **Edge node**:
  (1) aggregator
  (2) device
- **Fog Node**:
  (1) irrigation controller
  (2) event
- **Cloud Node**
  (1) gateway
  (2) persistence
  (3) dashboard

Authors' Contact Information: Simone Guidato, Università degli studi di Roma Tor Vergata, Roma, Italia; Leonardo Beccarini, Università degli studi di Roma Tor Vergata, Roma, Italia.

ACM XXXX-XXXX/2025/10-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Come broker per la comunicazione asincrona con code di messaggi è stato usato un broker MQTT (RabbitMQ + plugin MQTT) [7]. Come supporto per la persistenza è stato usato InfluxDB [3] (avendo a che fare con serie temporali), sia per la persistenza degli eventi di irrigazione sia per la persistenza dei dati aggregati di umidità del suolo.

Analizziamo nel dettaglio ogni componente.

2.1 MQTT broker

Principale canale di comunicazione del sistema, ci permette di avere disaccoppiamento temporale e spaziale tra produttori e consumatori, offrendo un modello publish/subscribe a topic gerarchici. I topic sono:

- **sensor/data/{field}/{sensor}**: *Produttore*: sensor simulator *Consumatore*: aggregator
- **sensor/aggregated/{field}/{sensor}**: *Produttore*: aggregator *Consumatori*: persistence, irrigation-controller
- **event/StateChange/{field}/{sensor}**: *Produttore*: device *Consumatore*: sensor simulator, event
- **event/irrigationResult/{field}/{sensor}**: *Produttore*: device *Consumatore*: irrigation-controller, event

2.1.1 Semantica di consegna e idempotenza. Applichiamo una semantica ibrida:

- *sensor/data/** a **Qos 0** (at-most-once) per flussi di dati grezzi ad alta frequenza, dove una riconsegna o perdita occasionale non ha impatto
- *sensor/aggregated/** ed entrambi i topic *event/** a **QoS 1** (at-least-once).

Per neutralizzare i duplicati dovuti alla semantica at-least once, i *consumer* mantengono un set delle *chiavi di idempotenza*. Se una chiave ricompare entro una **finestra di deduplicazione** (TTL), il messaggio viene scartato. In questo modo si ottiene un effetto *exactly-once* a livello applicativo.

Chiavi d'idempotenza (per topic):

- *sensor/aggregated/**: SHA-256 del payload JSON
- *event/StateChange/**: SHA-256 del payload JSON
- *event/irrigationResult/**: ticket_id lato controller; SHA-256(topic+payload JSON) lato event-service

2.1.2 Schema payload dei messaggi. **AggregatedSensorData**

```
{ "field_id":"field_1","sensor_id":"sensor_2","moisture":42.1,"a
↪  ggregated":true,"timestamp":"2025-09-30T23:42:51Z"
↪  }
```

StateChange

```
{ "field_id":"field_1","sensor_id":"sensor_2","new_state":"On","
↪  duration_s":120,"timestamp": "..."
↪  }
```

IrrigationResult

```
{ "field_id":"field_1","sensor_id":"sensor_2","ticket_id":"155e7bde-
↪  2625-4eed-9d90-77adfa021e6b","status":"OK","mm_applied":3.5,"sta
↪  rted_at":"...","timestamp":"..."
↪  }
```

2.2 Physical sensor node

Questo nodo ha il compito di simulare i sensori IoT che sono la sorgente di dati del nostro sistema.

2.2.1 Sensor Simulator. Pubblica periodicamente sul topic sensor/data/{field}/{sensor} e si sottoscrive a event/StateChange/{field}/{sensor} per applicare i comandi provenienti dal servizio device (accensione/spegnimento).
All'avvio inizializza il valore di umidità utilizzando SoilGrids [4] e successivamente evolve il valore con una dinamica interna (decadimento quando l'attuatore del sensore è spento + aumento quando è acceso).

2.3 Edge node

2.3.1 Aggregator. Consuma i campioni ad alta frequenza prodotti dagli emulatori e, per ciascuna coppia {field,sensor}, calcola una **media mobile** su finestra scorrevole. Ogni Δt emette un *punto aggregato* su sensor/aggregated/{field}/{sensor} con valore medio e timestamp. In questo modo riduce rumore e burstiness e fornisce al controller un segnale più stabile su cui prendere decisioni.

2.3.2 Device. Microservizio responsabile di gestire l'attuazione delle decisioni di irrigazione. Espone un endpoint gRPC per ricevere dal controller delle irrigazioni comandi parametrici (field, sensor, duration). Traduce tali comandi in:

- event/StateChange/{field}/{sensor}: comando che cambia lo stato degli attuatori
- event/irrigationResult/{field}/{sensor}: per comunicare l'esito dell'irrigazione, con la quantità di acqua effettivamente erogata.

Per assicurarsi che il sensore sia attivo per la durata dell'irrigazione utilizza un meccanismo di liveness implicita. Il sistema si sottoscrive al flusso grezzo dei dati generati dal sensore e registra il timestamp dell'ultimo dato ricevuto. Durante un'irrigazione, se non vengono ricevuti dati per un intervallo superiore a una soglia configurabile, il sensore viene considerato offline: l'irrigazione viene contrassegnata come fallita e viene pubblicato l'esito, indicando la quantità d'acqua erogata fino a quel momento.

2.4 Fog node

2.4.1 Irrigation controller. Centro decisionale del sistema, per ogni sensore recupera dalla OpenWeather API [6]le variabili meteorologiche giornaliere (temperatura minima/massima e precipitazioni) e stima localmente ET_0 con la formula semplificata di Hargreaves–Samani [2](con R_a trattato come costante per ottenere mm/giorno):

$$ET_0 = 0.0023\,(T_{mean} + 17.8)\sqrt{T_{max} - T_{min}}\,R_a$$

Sulla base di ET_0 e della pioggia prevista calcola a mezzanotte locale il **budget idrico giornaliero** $B_d = B_{base} + k \cdot \max(0, ET_0 - P)$ e ne mantiene il rimanente per sensore. All'arrivo di un dato aggregato, se l'umidità del terreno scende sotto una soglia, determina una dose preliminare $\hat{D} = 5 + 0.5 \cdot \max(0, ET_0 - P)$ e applica un clamp $D = \min(\hat{D}, B_{rem})$ per non superare il budget residuo.

Converte quindi i millimetri da erogare in durata in minuti, utilizzando l'area coperta dal sensore e il flusso di erogazione; l'avvio avviene solo se il sensore non è *busy*. Se l'irrigazione è attuabile invoca via gRPC il device service, crea una richiesta aggiungendo un Id univoco (*ticket_id*) e memorizza l'entry in una mappa di decisioni *pending*, impostando un **TTL pari alla durata dell'irrigazione più un margine configurabile**.

Parametri principali (configurabili): `MOISTURE_THRESHOLD_PCT` (soglia), `MOISTURE_GUARDS` (isteresi), `DAILY_BASE_MM`, `DAILY_ETO_COEFF` (calcolo di B_d), `IRR_PENDING_MARGIN` (margine sul TTL), `TZ` (fuso orario per il calcolo a mezzanotte).

Il controller si sottoscrive a `event/irrigationResult/*`, recupera l'entry *pending* e la elimina; **detrae dal budget del sensore i mm effettivamente applicati** solo se riceve un risultato coerente entro il TTL.

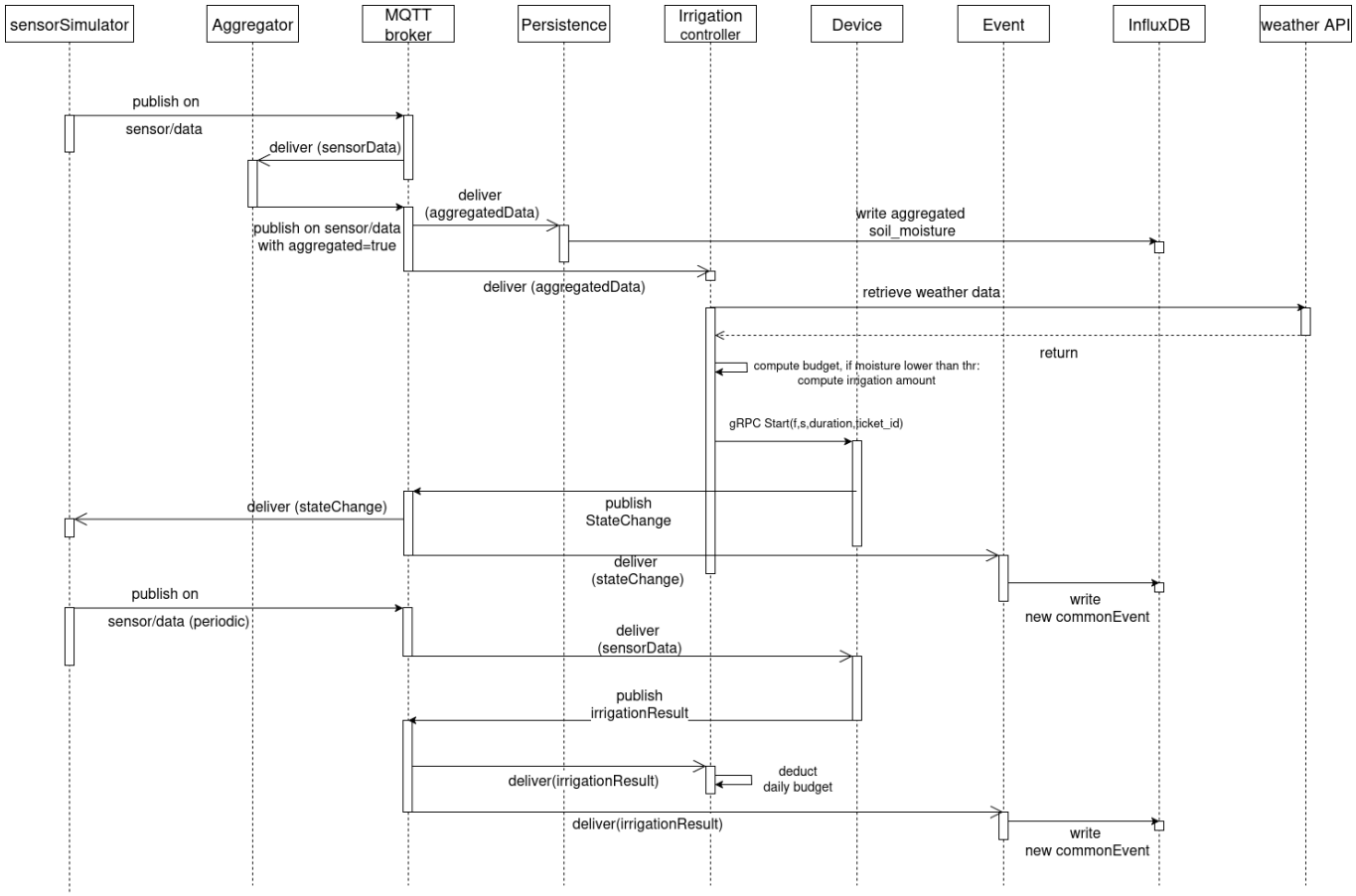

Fig. 2. Sequence diagram raw data –> result

2.4.2 Event. Questo servizio si occupa della persistenza degli eventi che occorrono nel sistema. In particolare si iscrive ai topic event/StateChange/* e event/irrigationResult/*. Questi due eventi sono decodificati in un modello unificato **CommonEvent** che espone metadati (tipo di evento, sorgente) più un dizionario di attributi: per *StateChange* sono valorizzati lo stato di destinazione e la durata della transizione, mentre per gli *irrigationResult* sono registrati esito, millimetri effettivamente erogati e motivazione, con severità elevata a *warning* in caso di *FAIL*.
Gli eventi normalizzati sono tradotti in *point* Influx aggregati nella misura unica `system_event`, proiettando nei *tag* i metadati stabili `(event_type, source_service, field_id, sensor_id, severity)` e mantenendo nei *field* gli attributi osservati (ad es. `mm_applied, status, duration`), con timestamp pari all'istante logico dell'evento.
La scrittura avviene in modo asincrono e batch verso InfluxDB.
Il servizio espone, inoltre, l'endpoint di query /events/irrigation/latest?limit=&minutes= che restituisce, per gli ultimi eventi di irrigazione, la quantità erogata e il relativo timestamp (UTC) per sensore.

2.5 Cloud Node

2.5.1 Persistence. Il *persistence service* realizza la persistenza dei dati di umidità del terreno aggregati: si sottoscrive al topic sensor/aggregated/*, mappa ogni osservazione in un *point* Influx con tag field_id, sensor_id e `aggregated`. con il field numerico moisture. **Il timestamp del point coincide con quello del dato aggregato, se presente, altrimenti viene valorizzato all'istante corrente**.
Il servizio mantiene una cache dell'ultimo valore per coppia campo/sensore. Inoltre espone un endpoint HTTP /data/latest, il quale prova ad estrarre gli ultimi valori di moisture tramite una query Flux; in assenza di risultati o in caso di errore effettua il fallback sulla cache.

2.5.2 Gateway. Il gateway espone l'endpoint /dashboard/data che **compone** i dati provenienti da due servizi interni:

(1) ultimi valori di umidità dal *persistence service*
(2) ultimi esiti di irrigazione dall'*event service*

Le due chiamate sono **eseguite in parallelo** con **timeout indipendenti**; i risultati vengono **normalizzati** (nomenclature/tipi) e unificati in un unico payload adatto alla Dashboard. Se una delle due fonti non risponde entro il proprio timeout, la risposta viene **degradata in modo parziale** includendo solo i dati disponibili (con campi null o placeholder per quelli mancanti).

Su ciascun upstream è interposto un **circuit breaker** autonomo, implementato come una piccola macchina a stati: in *Closed* le chiamate passano e i fallimenti sono conteggiati; al superamento della soglia d'errore il breaker entra in *Open* e **rifiuta temporaneamente** nuove richieste verso quell'upstream; scaduto l'intervallo di apertura, passa in *HalfOpen* ed effettua una **singola sonda**. Se la sonda ha esito positivo il breaker ritorna in *Closed*, altrimenti torna in *Open*. L'uso è esplicito nel codice: prima della chiamata `Allow()`, al termine Success() o Fail() a seconda dell'esito. Questo meccanismo **limita i retry inutili** e consente all'altro upstream di completare, mantenendo la reattività dell'endpoint.

Parametri tipici: error_threshold (es. 5 fallimenti su finestra mobile), open_for (es. 30 s), request_timeout (per-upstream), halfopen_probe=1.

2.5.3 Dashboard. Servizio di presentazione: effettua una singola richiesta con timeout a `/dashboard/data` e visualizza, *per sensore*, l'**ultimo valore di umidità** disponibile e gli **ultimi esiti di irrigazione** (quantità e istante). Il payload viene normalizzato (chiavi/tipi, gestione di null) e trasformato in un modello uniforme per la UI. Oltre alla vista per-sensore, la pagina espone *statistiche trasversali* sull'**ultimo snapshot** (min/max/media dei valori di umidità correnti).

Degrado controllato:

- se mancano gli ultimi *valori di umidità*, mostra solo gli *esiti di irrigazione* disponibili (con placeholder per i valori assenti);
- se mancano gli *esiti*, mostra i *valori correnti* e le statistiche di snapshot, omettendo la timeline delle irrigazioni;
- se uno degli upstream non risponde entro il proprio timeout o è in *Open* (circuit breaker), la risposta viene composta **parzialmente** con i dati disponibili e indicatori non intrusivi per i campi mancanti.

3 Deployment

Per riprodurre un'architettura *edge–fog–cloud* abbiamo adottato un approccio ibrido: **Minikube** per orchestrare i livelli *edge* e *fog* in locale (cluster multinodo con latenza controllata via *tc netem*) e **AWS Elastic Beanstalk** per il livello *cloud*. Queste scelte permettono di:

(1) isolare i piani di esecuzione;
(2) modellare la rete con *tc netem*;
(3) semplificare il deployment cloud con un PaaS Docker-based.

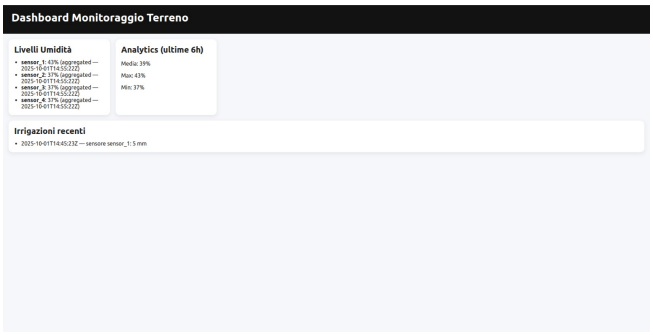

Fig. 3. Vista della dashboard

3.1 Topologia logica e ambienti Kubernetes

Gli ambienti *edge* e *fog* sono eseguiti in un unico cluster Minikube **multinodo**; sono definiti i namespace: `sensors`, `edge`, `fog`, rabbitmq. Namespace distinti separano responsabilità e configurazioni tra livelli e rendono più semplice lo scheduling con nodeSelector e label. In più, Secret/ConfigMap restano nel proprio ambito e i servizi sono raggiunti via DNS (<svc>.<ns>.svc.cluster.local).

Viene creato un cluster con 6 nodi (1 control-plane + 5 worker) e si applicano label di ruolo per vincolare il posizionamento dei pod tramite `nodeSelector`:

- `role=rabbitmq` per il broker,
- `role=sensors` per i simulatori di sensori IoT,
- `role=fog` per irrigation-controller, event-service e InfluxDB,
- `role=edge1` e `role=edge2` per aggregator e device-service.

La topologia del sistema è statica ed è composta da 4 sensori, due nodi edge (responsabili di due sensori ciascuno) e un singolo nodo fog.

3.1.1 Emulazione latenza. Per simulare le diverse condizioni di collegamento nel continuum si usano *initContainer* con `tc netem` nei manifest:

- *Edge*: ogni deployment applica un ritardo (es. `DELAY=10ms`, `JITTER=3ms`) prima di avviare il container applicativo;
- *Fog*: `k8s/fog/irrigation-controller.yaml` imposta, ad esempio, `DELAY=20ms`, `JITTER=6ms`.

Questa modellazione permette di osservare gli effetti end-to-end (propagazione degli eventi sensoriali, decisioni di irrigazione, feedback esecutivi) sotto latenza controllata.

3.1.2 Messaggistica e servizi. Nel namespace `rabbitmq` sono definiti:

- **RabbitMQ** con plugin MQTT abilitato;
- un deployment **ngrok** per pubblicare la porta MQTT (1883) verso l'esterno quando richiesto.

Nel namespace `sensors`:

- quattro **sensor-simulator**.

Nel namespace `edge`:

- **aggregator-1/2**,
- **device-service-1/2**.

Nel namespace `fog`:

- **InfluxDB** per la persistenza degli eventi,
- **event-service** con **Cloudflare Tunnel** che crea un endpoint pubblico verso il fog.

3.2 Deployment cloud con AWS Elastic Beanstalk

Il livello *cloud* ospita i componenti **Gateway**, **Persistence** e **Dashboard** impacchettati come single-container Docker su Elastic Beanstalk. EBS è un servizio PaaS che provvisiona e orchestra automaticamente un ambiente applicativo Docker su EC2 (scaling, load balancer, health check, log/metriche integrati), permettendo di caricare un artifact zip contenente Dockerfile e sorgenti.

In particolare sono state istanziate tre applicazioni: EBS—*Gateway*, *Persistence*, *Dashboard*, ognuna in un environment Docker single-container. Ogni environment gestisce il lifecycle dell'applicazione a lui corrispondente (deploy, rollback, aggiornamenti e scaling).
La configurazione avviene via variabili d'ambiente (es. INFLUX_URL/ORG/BUCKET/TOKEN, EVENT_URL); tutti gli environment sono nella stessa VPC (configurazione di default).

3.2.1 InfluxDB su EC2. : per il servizio *persistence* utilizziamo un'istanza EC2 dedicata con InfluxDB: i dati aggregati vengono scritti su questo endpoint.
L'accesso è regolato dai *Security Group*: sull'istanza EC2 è configurata una regola *inbound* TCP sulla porta `8086` con **sorgente** impostata al Security Group dell'environment EBS di *persistence*.

3.2.2 Connettività cloud–fog. Per collegare i servizi EBS al cluster Minikube:

- **Verso il fog**: il deployment event-quick-tunnel pubblica l'event-service tramite **Cloudflare Tunnel** [1], generando un URL pubblico; lo script `deploy_minikube.sh` include un estratto per recuperare tale URL e (opzionalmente) impostarlo come variabile d'ambiente nell'ambiente EBS (es. EVENT_URL).
- **Verso il broker MQTT**: **ngrok** [5] nel namespace `rabbitmq` espone la porta 1883 verso Internet; i servizi cloud consumano i topic MQTT puntando all'endpoint TCP esterno, mantenendo inalterate le stesse variabili già usate in locale.

4 Risultati

In questa sezione presentiamo le misure raccolte con *Prometheus* e visualizzate in *Grafana*. Abbiamo utilizzato il *Blackbox Exporter* (probe HTTP) per stimare la latenza end-to-end dei microservizi e i dashboard di *RabbitMQ* per il throughput del piano dati MQTT.

4.1 Metodologia di misura

Misuriamo:

- **RTT HTTP** (*round-trip time*) dei servizi interni tramite il tempo di risposta del probe.
- **RTT p95** su finestre scorrevoli, per cogliere la coda superiore della latenza.
- **Jitter** come variabilità (deviazione standard/range su finestra) dell'RTT.
- **Metriche RabbitMQ** (*messages published/routed/delivered*, *delivered with manual ack*) per verificare stabilità e assenza di backlog con QoS 1.

4.2 Latenza e jitter

Le misure evidenziano il previsto ordinamento *fog* > *edge* in termini di latenza e variabilità, coerente con i parametri di `tc netem`. Sull'*edge* la p95 si colloca tipicamente tra 7–13 ms; sul *fog* tra 30–35 ms con picchi intorno a 40 ms. Il jitter sul *fog* risulta più ampio per l'effetto combinato di ritardo, variabilità di rete e overhead di piattaforma (overlay Kubernetes, scheduling, DNS interno).

Fig. 4. RTT p95 sui servizi *edge* (aggregator-1/2). Le variazioni a gradino corrispondono a cambi dei parametri `tc netem`.

Fig. 5. RTT p95 sul servizio *fog* (event-service). Valori tipici 30–35 ms in linea con delay 20 ms + jitter.

Fig. 6. RTT medio nel tempo sul *fog*. La maggiore variabilità riflette jitter e overhead applicativo.

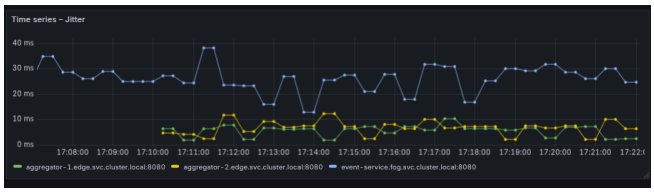

Fig. 7. Jitter (variabilità RTT) per *edge* e *fog*. La banda più ampia sul *fog* è coerente con la configurazione `netem`.

4.3 Piano dati MQTT (RabbitMQ)

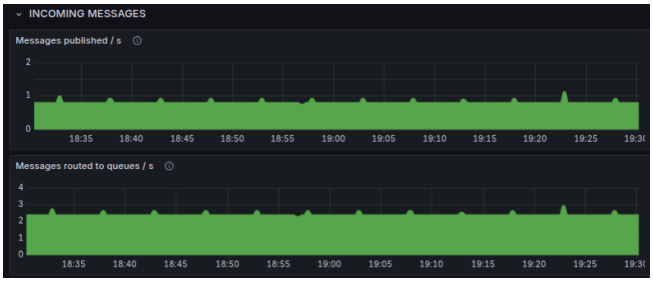

Fig. 8. RabbitMQ — *Incoming messages*: pubblicati e instradati al secondo. Serie stabile, assenza di backlog.

Si nota come i producer emettano messaggi ad un ritmo costante (circa 1 msg/s), *message routed to queues* è più alto perché lo stesso topic viene consumato da più servizi.

Fig. 9. RabbitMQ — *Outgoing messages*: consegne e consegne con *manual ack*/s. Pattern coerente con QoS 1.

message delivered ricalca *message routed to queues*, i messaggi entrano in coda alla stessa velcità con cui vengono smaltiti.
message delivered with manual ack mostra il comportamento atteso con QoS 1 (ack esplicito).

5 Conclusioni

I risultati ottenuti confermano la credibilità della simulazione del computing continuum: la latenza risulta più bassa all'edge rispetto al fog, in linea con il comportamento atteso e con la configurazione imposta tramite *tc netem*. Il protocollo MQTT si è dimostrato stabile, con valori di *delivered* pressoché coincidenti con *routed* e un meccanismo di acknowledgement coerente con il livello di QoS 1.
Nel complesso,questa implementazione ha validato l'architettura proposta, mostrando che l'infrastruttura fog-edge è in grado di gestire correttamente la comunicazione e il flusso dei dati.

5.1 Miglioramenti

Sarebbe stato interessante misurare la latenza end-to-end, dal momento in cui il dato viene emesso a quando il risultato dell'irrigazione è presente sulla dashboard.
L'attuale approccio, basato su *tc netem*, permette di modellare delay e jitter ma non tiene conto della congestione o della competizione per le risorse di rete, fattori che possono influire in modo significativo sul comportamento reale del sistema.
Inoltre, la topologia del sistema è statica e il carico di lavoro limitato, riducendo la variabilità e quindi la significatività statistica delle misurazioni. Di fatto, si potrebbe anche valutare l'impatto di topologie dinamiche, con nodi fog che entrano o escono dalla rete, per testare la resilienza e l'adattabilità del sistema.
Un ulteriore sviluppo potrebbe riguardare l'integrazione di un modello di *Machine Learning* per la predizione della quantità di irrigazione necessaria, basato su dati storici ambientali (ad esempio umidità del suolo, temperatura, e tassi di evaporazione).
In questo modo, il sistema potrebbe anticipare i bisogni idrici, ottimizzare l'uso dell'acqua e ridurre la latenza percepita tra l'acquisizione dei dati e l'azione. Tale componente predittiva renderebbe il sistema più intelligente e autonomo.

References

[1] Cloudflare, Inc. 2025. *Cloudflare Tunnel Documentation.* https://developers.cloudflare.com/cloudflare-one/connections/connect-networks/ Accessed 2025-10-06.
[2] George H. Hargreaves and Zohrab A. Samani. 1985. Reference Crop Evapotranspiration from Temperature. *Applied Engineering in Agriculture* 1, 2 (1985), 96–99. doi:10.13031/2013.26773
[3] InfluxData, Inc. 2025. *InfluxDB 2.x HTTP API.* https://docs.influxdata.com/influxdb/v2/api/ Accessed 2025-10-06.
[4] ISRIC — World Soil Information. 2025. *SoilGrids REST API.* https://docs.soilgrids.org/ Accessed 2025-10-06.
[5] ngrok, Inc. 2025. *ngrok Tunnels Documentation.* https://ngrok.com/docs/ Accessed 2025-10-06.
[6] OpenWeather Ltd. 2025. *OpenWeather API (One Call).* https://openweathermap.org/api/one-call-3 Accessed 2025-10-06.
[7] VMware, Inc. 2025. *RabbitMQ MQTT Plugin Guide.* https://www.rabbitmq.com/mqtt.html Accessed 2025-10-06.