La tua bolletta AWS non è uno strumento di observability: come monitorare l'utilizzo dei token per Amazon Bedrock

26 Agosto 2026 - 14 min. read

Every single day
And every word you say
Every game you play
Every night you stay
I'll be watching you

The Police, I'll be watching you

Un team che gestisce workload di intelligenza artificiale generativa dovrebbe poter rispondere a questa domanda in qualsiasi momento della giornata: **quanti token stiamo consumando e quanto ci costano?**

Se la risposta a questa domanda è "fammi controllare il billing", allora questo articolo fa al caso tuo.

Di recente ho imparato, nel modo peggiore possibile, che la billing console è uno strumento di reconciliation, non di observability. È in ritardo, è aggregata e, in casi rari ma molto reali, *può anche sbagliare.*

Quanto sbagliata? Nel mio caso, di un fattore mille.

Mi sono svegliato (beh, tecnicamente, avevo appena finito di pranzare) davanti a una bolletta da 58.000 dollari per un workload che sarebbe dovuto costare una cinquantina di dollari. L'unico motivo per cui sono riuscito a dimostrare l'errore, a farlo correggere in pochi giorni e a far sistemare il bug sottostante per tutti è che avevo una

telemetria dei token mantenuta dall'applicazione, che non dipendeva dal metering di AWS.

Ma niente spoiler: vediamo cos'è successo e come sono sopravvissuto.

L'incidente

Per prima cosa, un po' di contesto. Magari sapete già che mi piace testare nuovi servizi e condividere quello che imparo: a febbraio 2026, da Community Builder (adesso sono Hero!), stavo sperimentando con agent autonomi su Amazon Bedrock usando OpenClaw, confrontando diversi foundation model: DeepSeek v3.2, Amazon Nova Lite e, più avanti, Kimi K2.5 di Moonshot AI, arrivato nel catalogo Bedrock giusto in tempo.

Il pricing di Kimi sembrava ottimo: $0.60 per milione di input token e $3.00 per milione di output token. Ma la libreria Node.js di OpenClaw non andava d'accordo con l'implementazione del tool-calling di Kimi sulle API native di Bedrock Converse: ottenevo chiamate fallite, risposte incoerenti e azioni che semplicemente non venivano eseguite, il tutto senza errori evidenti.

Ho iniziato a fare debug della libreria, ma alla fine ho fatto ricorso a un altro metodo: leggere la documentazione (una tecnica di debug sorprendentemente efficace), che mi ha portato a trovare un workaround: **Project Mantle**, il motore di distributed inference di Bedrock annunciato al re:Invent 2025, che espone endpoint con API **OpenAI-compatible** come sostituto drop-in, senza dover modificare il codice applicativo.

Basta puntare il proprio client OpenAI all'endpoint **https://bedrock-mantle..api.aws/v1**, generare un access token per Bedrock, e tutto funziona. Ed è stato proprio così: sono passato a Mantle il 17 febbraio, e i problemi di tool-calling sono spariti.

Ovviamente, trattandosi di esperimenti, ho mantenuto un approccio da "cittadino del cloud" responsabile, con controlli giornalieri del billing nell'account di management dell'organizzazione. Ho impostato un budget alarm da $100 sull'account sandbox e un forecast alarm, ovviamente. Il 23 febbraio, la bolletta mensile dell'intera organizzazione era di circa $140, comodamente coperta dai crediti AWS, con gli esperimenti su Bedrock che contribuivano ai costi con i soliti $3-5 al giorno.

Poi, il 24 febbraio, subito dopo pranzo, sono arrivate due notifiche di AWS Budgets contemporaneamente: la spesa reale era di **$56.265,59**, mentre la previsione di fine

mese era di **$70.161,62**, senza alcuna crescita graduale né alcun alert intermedio. La bolletta è passata da $140 a oltre $56.000 dall'oggi al domani, per poi assestarsi oltre i $58K, tasse incluse.

Dopo il fisiologico attacco di panico e il conseguente mezzo infarto, ho fatto quello che facciamo in beSharp quando un cliente ci chiama con una bolletta spaventosa: ho iniziato a misurare.

Individuare il problema

Il breakdown del billing per us-east-1 mostrava il problema, ma in modo sottile, e c'è voluto un attimo per crederci davvero. Gli input token di Kimi K2.5 attraverso Mantle erano **77.228.206 unità da 1K token**, fatturate a $0,0006 per 1K, per un totale di **$46.336,92**. Gli output token erano 195.679 unità da 1K token a $0,003, per un totale di altri $587,04.

77 milioni di migliaia di token. Cioè, 77 **miliardi** di token: una quantità di testo che non potresti, da solo, far passare attraverso un workflow di agenti interattivi in una settimana, nemmeno provandoci apposta.

La telemetria lato applicazione mi ha salvato, perché OpenClaw traccia autonomamente l'utilizzo dei token, a partire dai campi usage restituiti dall'API. I suoi numeri per Kimi K2.5 via Mantle erano di circa **72,1 milioni di token su 547 messaggi**. Non 72 milioni di migliaia di token, ma 72 milioni, e basta.

Per essere sicuro che i numeri di OpenClaw fossero affidabili, ho verificato l'utilizzo dei token anche sugli altri modelli che avevo usato accedendo alle API native di Bedrock:

Modello	Percorso API	Telemetria lato app	Billing console	Corrispondenza?
DeepSeek v3.2	Bedrock nativa (Converse)	~61,8M token / 844 messaggi	61.910 x 1K token	Sì
Amazon Nova Lite	Bedrock nativa (Converse)	Coerente	Coerente	Sì
Kimi K2.5	Mantle (OpenAI-compatible)	~72,1M token / 547 messaggi	77.228.206 x 1K token	**Scarto di ~1000x**

Era chiaro che qualcosa nel billing di Mantle trattava i token come unità da 1K, moltiplicando la mia bolletta per 1000; quindi, invece di $46,34, il mio debito era di

$46.336,92. Si trattava di una semplice virgola fuori posto che mi stava costando quanto un'auto nuova (e nemmeno delle più economiche!).

La bolletta gonfiata era il bug, ma è stata l'assenza di segnali "premonitori" a peggiorare la situazione:

- **Le metriche dei token su CloudWatch non venivano popolate** per le invocazioni a Mantle: le metriche di utilizzo dei token che normalmente compaiono per Bedrock risultavano vuote per il nuovo endpoint.
- **I dati di billing sono stati scaricati tutti insieme**, saltando il mio budget alarm da $100 e notificandomi solo quando la spesa era già a $56K.
- **Cost Explorer alla fine ha mostrato che l'anomalia era iniziata il 17 febbraio**, esattamente il giorno del passaggio a Mantle, ma non è stato aggiornato se non una settimana dopo.

Ogni rete di sicurezza che avevo era a valle del metering di AWS; quando i dati sono arrivati in ritardo, tutte le mie difese sono diventate inutili. Questa è la modalità di **correlated failure** che dovrebbe preoccuparti: budget alarm, forecast alarm, anomaly detection e Cost Explorer condividono tutti la stessa fonte di dati a monte.

Come si è risolto

Ho costruito un report con il confronto dei numeri visti prima: i numeri relativi alle API native erano corretti, mentre i costi di Mantle erano gonfiati e il fattore 1000x era costante in tutto il dataset. Ho aperto un ticket di supporto (anche con il piano di supporto base è possibile aprire un ticket relativo al billing) e ho descritto il mio caso con tutti i relativi conti. La conversazione è iniziata sabato; entro martedì successivo il billing era corretto e il bug era stato risolto a livello del service team, per me e per chiunque altro.

Due cose hanno reso possibile questo risultato: **i dati** e la **buona fede**. Non stavo chiedendo ad AWS di fidarsi delle mie sensazioni sulla bolletta; stavo mostrando numeri provenienti da un sistema di misurazione indipendente.

Per essere onesto, Project Mantle è bello dal punto di vista ingegneristico: si tratta di un'infrastruttura di inferenza distribuita che risolve problemi di migrazione reali nel tempo necessario per modificare un file di configurazione. Dopo un po' di tempo dal

mio incidente, AWS ha pubblicato una documentazione dedicata al monitoraggio dell'endpoint **bedrock-mantle**.

Gli early adopter trovano i bug; fa parte del gioco, la cosa importante è che li trovi tu, non il tuo CFO.

La versione completa della storia è sul mio blog personale; qui voglio concentrarmi sull'approccio che consente di scovare questo tipo di problema prima ancora di dare un'occhiata a Cost Explorer.

I token sono la nuova unità fatturabile

Abbiamo passato 15 anni a costruire l'observability per le classiche unità di costo del cloud: instance hour, GB-month e request. Impostiamo alert, tag, assegnandoli a team e centri di costo.

La GenAI introduce una nuova unità fatturabile, il token, con alcune proprietà scomode:

- **Il consumo lo decide il modello**: la lunghezza dell'output varia, esistono i reasoning token, e anche gli agent loop rendono tutto più difficile da stimare. Dobbiamo anche tenere presente che due richieste simili possono comportare costi molto diversi.
- **Un APM tradizionale non è progettato per tracciare l'utilizzo dei token**: una richiesta che costa $0,002 e un'altra che ne costa $0,40 possono condividere lo stesso profilo di latenza e lo stesso status code HTTP 200.
- **I moltiplicatori si nascondono ovunque.** Cache read e write, burndown rate rispetto alla quota TPM, differenze di pricing tra modelli, cross-region inference: mappare "cosa ha fatto la mia app" su "quanto pago" non è affatto banale.
- **Gli agenti amplificano tutto.** Un agent autonomo bloccato in un loop non genera errori. Consuma silenziosamente token con uno 0% di error rate finché qualcosa (si spera un alarm, magari il limite della tua carta di credito) non lo ferma.

Dobbiamo trattare l'utilizzo dei token con lo stesso approccio che usiamo per qualsiasi altro workload in produzione: raccogliere metriche lato provider, impostare la telemetria applicativa, e fare reconciliation finanziaria. Costruiamolo, layer per layer.

Layer 1: la visibilità AWS-native (necessaria, non sufficiente)

Tutto ciò che è presente in questo layer proviene direttamente da AWS. Si tratta dello strato fondamentale, che costa poco e andrebbe sempre utilizzato, anche se condivide il destino del metering di AWS. Le componenti fondamentali sono:

Metriche di CloudWatch nel namespace AWS/Bedrock. Bedrock pubblica automaticamente **Invocations**, **InvocationLatency**, **InputTokenCount** e **OutputTokenCount** per ciascun modello. Sono la baseline: vanno impostati allarmi anche sui conteggi dei token, e non solo su quelli delle invocazioni, perché il volume e il costo dei token sono correlati solo debolmente.

Metriche operative Bedrock più recenti. Bedrock adesso mette a disposizione anche le metriche **TimeToFirstToken** per le API in streaming e **EstimatedTPMQuotaUsage**, che riflettono come le richieste consumano davvero la quota di token per minuto. La metrica sulla quota è particolarmente interessante: se il consumo stimato della quota si discosta in modo significativo dal numero di token rilevati dall'applicazione, c'è qualcosa che non va da qualche parte, e vale la pena indagare, potrebbe esserci un bug da qualche parte.

Model invocation logging. È disabilitato di default, perché Bedrock scriverebbe metadati, richieste e risposte di ogni invocazione su CloudWatch e/o S3, inclusi i conteggi di input e output token per ogni singola invocazione. È la cosa più vicina a un audit trail lato provider, e CloudWatch Logs Insights permette di interrogare i dati, vedere quali sessioni presentano il maggiore utilizzo di token e raggruppare le richieste in base alla dimensione dell'input. Va valutata con attenzione l'adozione di queste metriche, in quanto può generare una notevole mole di dati, oltre a potenziali dati PII o, nel peggiore dei casi, dati sensibili.

CloudWatch generative AI observability. La dashboard preconfezionata Model Invocations fornisce il conteggio dei token per modello, i trend giornalieri, i throttle e una tabella delle invocazioni già pronta, senza dover costruire nulla.

Application inference profile e cost allocation tag. Se più applicazioni o team condividono lo stesso account Bedrock, è possibile creare profili di inferenza e applicare tag a ciascuno di essi. L'utilizzo di Bedrock senza tag si traduce in un'unica voce su Cost Explorer, e determinare "chi ha speso quanto?" diventa una sessione di scavi archeologici nei dati.

AWS Budgets e Cost Anomaly Detection. Se non li hai ancora sul tuo account AWS, entra in console adesso e abilitali e affiancali con soglie intermedie: non passare da $100 al panico e interiorizza anche il loro limite: scattano sui dati di billing, e i dati di billing per un layer API nuovo di zecca potrebbero essere in ritardo, raggruppati o, una volta ogni tanto, sbagliati per via di bug.

Una nota: se stai provando nuovi servizi, verifica che le metriche vengano effettivamente popolate prima di fidarti di loro. Nel mio caso, l'endpoint Mantle non ha restituito alcuna metrica sui token per giorni. Una dashboard vuota assomiglia moltissimo a un workload tranquillo e in salute.

Layer 2: le convenzioni OpenTelemetry GenAI (il tuo testimone indipendente)

Questo è il layer che mi ha salvato, ma in una forma più corretta e definita. L'applicazione vede ogni risposta della API, e ogni risposta può contenere un blocco usage. Una volta catturato, strutturato e inviato ad un backend, è possibile avere un sistema di accounting dei token che non dipende dal provider.

È possibile implementare logging custom (essenzialmente quello che fa il tracking integrato di OpenClaw), ma l'industria sta convergendo verso l'**OpenTelemetry GenAI semantic convention**. È uno standard sviluppato dall'OTel GenAI SIG che definisce un vocabolario comune per la telemetria degli LLM. Questo rende più semplice raccogliere metriche e span uniformi, qualunque sia la piattaforma che stai usando: una chiamata a Bedrock Converse, un agent LangChain o un endpoint compatibile con OpenAI avranno sempre una rappresentazione comune.

Al centro di tutto, ogni chiamata al modello diventa uno span che porta con sé attributi come:

```
gen_ai.provider.name         = "aws.bedrock"
gen_ai.request.model         = "moonshot.kimi-k2.5"
gen_ai.operation.name        = "chat"
gen_ai.usage.input_tokens    = 3412
gen_ai.usage.output_tokens   = 287
gen_ai.response.finish_reason = "tool_use"
```

In aggiunta a questi attributi, sono presenti anche le metriche: **gen_ai.client.token.usage** e **gen_ai.client.operation.duration**.

Da questi due è possibile derivare tutto ciò che conta: token per sessione, costo per feature (conteggi di token moltiplicati per la tua tabella di pricing), latenza p95 per modello e, cosa cruciale, una cifra aggiornata di continuo su "quanto dovrebbe essere fatturato".

La bellezza architetturale sta nel decoupling: è possibile esportare le metriche verso un **OpenTelemetry Collector**, che fa da fan-out verso i backend: CloudWatch (via AWS Distro for OpenTelemetry), uno stack self-hosted, Langfuse, o tutti insieme, senza toccare il codice applicativo.

Tre note pratiche dal mondo reale:

1. **L'auto-instrumentation fornisce l'80% delle funzionalità.** Librerie come OpenLLMetry e OpenLIT strumentano gli SDK LLM più popolari ed emettono span **gen_ai.*** automaticamente. Usando un endpoint OpenAI-compatible (come Mantle, o LiteLLM come gateway), l'instrumentation per OpenAI si ottiene cambiando solo l'URL base.
2. **Emetti una metrica di costo, non solo il conteggio dei token.** Un piccolo pezzo di logica nella pipeline che moltiplica l'utilizzo dei token per il prezzo del modello trasforma la telemetria in una dashboard di spesa quasi in tempo reale. La mia intuizione dei "$3-5 al giorno" era una versione mentale di questo
3. Attenzione alla maturità.

   La maggior parte delle convention **gen_ai.*** è ancora in sviluppo, quindi i nomi degli attributi possono cambiare tra una versione e l'altra. Gli attributi core sull'utilizzo dei token sono la parte più stabile.

Con questo layer attivo, il mio problema sarebbe stato rilevato lo stesso giorno (sempre che i dati di billing non fossero in ritardo e raggruppati). Un confronto programmato tra **sum(gen_ai.usage.input_tokens)** e i numeri riportati dal billing mi avrebbe avvisato il 18 febbraio, non il 24, perché una divergenza di 1000x tra due misurazioni della stessa cosa è l'allarme più rumoroso che si possa avere.

Layer 3: Langfuse (rendere i dati sui token esplorabili)

Le metriche indicano che si stanno utilizzando molti token. Usando gli agenti, però, occorre anche sapere in che modo e perché si stanno usando, e questo è un problema di tracing.

Una piattaforma di observability specifica per gli LLM si guadagna il proprio posto in questo caso. È possibile usare **Langfuse**, anche **in versione self-hosted**, mantenendo i prompt e le completion all'interno dell'account. Anche in questo caso vale lo stesso ragionamento fatto per il model invocation logging: occorre tenere in considerazione eventuali PII e dati sensibili.

Langfuse si comporta come un backend OpenTelemetry: espone un endpoint OTLP, ingerisce nativamente gli span `gen_ai.*`, e li trasforma in una UI appositamente costruita. Oltre alle trace grezze, si hanno:

- **Trace complete degli agent**: ogni step di una sessione come una gerarchia di span, con prompt, completion, tool call e utilizzo dei token per singolo step.
- **Cost tracking con model pricing**: Langfuse mantiene i prezzi per modello (e permette di definirne di custom, utile il giorno in cui arriva un nuovo modello), quindi ogni trace porta con sé un importo in dollari.
- **Dimensioni di attribuzione**: vengono mantenuti gli user ID, i session ID, i tag e i metadati, in modo che la spesa in token possa essere suddivisa per cliente, feature o esperimento.
- **Individuazione di loop e anomalie**: i pattern patologici degli agenti (una sessione che consuma 50.000 token per un task che normalmente ne richiede 3.000, retry storm, tool loop fuori controllo) sono visivamente evidenti in un trace tree e banalmente interrogabili.

Sul fronte del deployment, Langfuse gira benissimo su ECS o su Kubernetes e, per la maggior parte dei team, il sizing richiesto è modesto. Se il self-hosting non è un vincolo, la versione managed cloud elimina anche l'effort di manutenzione.

Mettiamo tutto insieme

Vediamo il quadro completo, per un'applicazione su ECS/EKS/Lambda che chiama Bedrock:

1. **Application layer:** chiamate agli SDK LLM instrumentate con le convenzioni OTel GenAI (auto-instrumentation dove possibile). Ogni invocazione genera uno span

relativo all'utilizzo dei token e un data point per la metrica.

2. **Collector layer:** un OpenTelemetry Collector (sidecar, DaemonSet o Gateway) riceve OTLP e funge da fan-out: le trace verso Langfuse, le metriche verso CloudWatch via ADOT (o verso Prometheus/Grafana managed).
3. **Provider layer:** model invocation logging di Bedrock abilitato verso CloudWatch Logs; le metriche native **AWS/Bedrock** e le dashboard di generative AI observability; application inference profile con cost allocation tag.
4. **Financial layer:** AWS Budgets configurato con soglie diverse, Cost Anomaly Detection e Cost Explorer per l'attribuzione dei costi.
5. **Il reconciliation loop:** un job che confronta periodicamente l'utilizzo dei token lato applicazione con i dati del provider. Può anche confrontare l'utilizzo fatturato con quello di Cost Explorer. Una divergenza oltre la soglia di tolleranza può far scattare un allarme.

Fidati, ma verifica

La mia storia è finita bene perché avevo i dati per dimostrare che il bug era reale, e la parte bella è che la correzione è stata resa disponibile a tutti. Project Mantle è un'aggiunta solida a Bedrock, e continuo a usarlo. Gli early adopter trovano i bug: è la tassa che paghiamo per giocare con i tool nuovi e, onestamente, continuo a pagarla volentieri.

La lezione va oltre un singolo bug in un layer API. Nella token economy, **la misurazione fornita dal tuo provider è una misura, non una verità assoluta**, e può risultare in ritardo, incompleta o errata. I team che gestiscono i workload AI in sicurezza sono quelli che misurano in modo indipendente, fanno reconciliation continua e sono in grado di dimostrare con i numeri i fatti quando qualcosa non torna.

La bolletta dovrebbe essere una conferma di quanto la telemetria sta già indicando. Se ci sono discrepanze e sorprese, in un senso o nell'altro, l'observability ha un buco.

State già misurando l'uso dei token nei workload di GenAI? Ti è mai capitato di riscontrare una discrepanza tra la misura dell'applicazione e la bolletta? Faccelo sapere nei commenti, e se vuoi una mano a progettare il layer di observability per i tuoi workload AI, sai dove trovarci!

About Proud2beCloud

Proud2beCloud è il blog di beSharp, APN Premier Consulting Partner italiano esperto nella progettazione, implementazione e gestione di infrastrutture Cloud complesse e servizi AWS avanzati. Prima di essere scrittori, siamo Solutions Architect che, dal 2007, lavorano quotidianamente con i servizi AWS. Siamo innovatori alla costante ricerca della soluzione più all'avanguardia per noi e per i nostri clienti. Su Proud2beCloud condividiamo regolarmente i nostri migliori spunti con chi come noi, per lavoro o per passione, lavora con il Cloud di AWS. Partecipa alla discussione!

Damiano Giorgi

Ex sistemista on-prem, pigro e incline all'automazione di task noiosi. Alla ricerca costante di novità tecnologiche e quindi passato al cloud per trovare nuovi stimoli.L'unico hardware a cui mi dedico ora è quello del mio basso; se non mi trovate in ufficio o in sala prove provate al pub o in qualche aeroporto!
