Ars Digitale
Engineering Notes Series

# PERFORMANCE ENGINEERING NOTES

Scalability · Latency · Throughput · Observability

**Alessandro Fabri**

2026 Edition
info@ars-digitale.com

# Note di Performance Engineering

Alessandro Fabri

Ars Digitale

# Note di Performance Engineering

# Note di Performance Engineering

Un riferimento tecnico strutturato per l’ingegneria delle prestazioni applicative e di sistema

**Alessandro Fabri**

2026 Edition

**Contatto**: info@ars-digitale.com

**Web**: https://www.ars-digitale.com

# Copyright



Versione: 1.0
Lingua: Italiano

**Contatto**: info@ars-digitale.com

**Web**: https://www.ars-digitale.com

2026 Edition

# Prefazione

Questa guida è pensata come un riferimento tecnico strutturato per l'ingegneria delle prestazioni, con particolare attenzione a chiarezza, precisione e utilità pratica.

Copre il comportamento applicativo e di sistema sotto carico, diagnostica, colli di bottiglia, accodamento, concorrenza, comportamento runtime e prestazioni a livello di risorsa.

Ogni capitolo è progettato sia come parte di un percorso coerente sia come riferimento autonomo per l'analisi tecnica.

# Informazioni su questo libro

Questo libro è stato progettato come un riferimento pratico e strutturato per l'ingegneria delle prestazioni.

L'obiettivo è combinare:

- precisione tecnica
- chiarezza concettuale
- ragionamento a livello di sistema
- utilità operativa
- valore duraturo come riferimento

L'edizione EPUB è ottimizzata per la lettura digitale e la navigazione per capitoli.

**Contatto**: info@ars-digitale.com

**Web**: https://www.ars-digitale.com

🌎 **Lingua:** Italiano

---

# Note di Performance Engineering

Questo indice fornisce la struttura completa **italiana (IT)** della guida **Note di Performance Engineering**.

La guida può essere letta in modo sequenziale oppure utilizzata come riferimento tecnico.

Questa documentazione pubblicata copre attualmente il **corpo teorico principale della guida**.

---

## Struttura della guida



---

# 1.1 – Fondamenti

Questa sezione introduce i concetti fondamentali necessari per ragionare sulle prestazioni applicative e dei sistemi.

Fornisce un modello concettuale utilizzato attraverso tutta la guida.

Definisce i principi fondamentali utilizzati nell'ingegneria delle prestazioni per l'analisi del comportamento dei sistemi sotto carico.

## Indice



---

## 1.1.1 Throughput, latenza, concorrenza

### Definizione

Queste sono le tre dimensioni principali utilizzate per descrivere le prestazioni di un sistema.

- **Throughput**: Quantità di lavoro eseguito nell'unità di tempo; numero di richieste elaborate per unità di tempo (es. richieste al secondo)
- **Latenza**: tempo necessario per completare una richiesta (tempo di risposta)
- **Concorrenza**: numero di richieste in elaborazione nello stesso momento

Questi concetti sono fondamentali nell'ingegneria delle prestazioni e sono utilizzati in tutta la guida per descrivere il comportamento dei sistemi.

---

### Relazione

Queste grandezze non sono tra loro indipendenti.

Per un sistema stabile:

- aumentare il throughput aumenta tipicamente la concorrenza
- aumentare la concorrenza tende ad aumentare la latenza
- la latenza influisce direttamente su quante richieste rimangono "in flight"

Questa relazione è centrale per comprendere come i sistemi si comportano sotto carico.

---

### Intuizione pratica

Un sistema può essere visto come una pipeline di elaborazione:

- **Input**: le richieste entrano
- **Execution**: vengono elaborate
- **Output**: escono

In ogni momento:

- alcune richieste sono in elaborazione (concorrenza)
- nuove richieste arrivano (throughput)

- ogni richiesta richiede tempo per essere completata (latenza)

Questo modello mentale aiuta a ragionare su flusso, accumulo e ritardi nei sistemi reali.

---

### Esempio

Se un sistema elabora:

- `100` richieste al secondo (100 Req./sec.)
- ogni richiesta richiede `200 ms` (0.2 s)

quindi, in media:

- circa `20` richieste sono `in flight` in ogni dato momento

Questa relazione è formalizzata dalla **Legge di Little**:

→ 1.2.1 Legge di Little

---

### Interpretazione pratica

Throughput, latenza e concorrenza formano un sistema chiuso.

Modificare uno di essi impatta necessariamente gli altri.

Per esempio:

- ridurre la latenza riduce la concorrenza a parità di throughput
- aumentare il throughput aumenta la concorrenza se la latenza rimane costante
- alta concorrenza aumenta la probabilità di accodamento e contesa

Questo è un elemento chiave per diagnosticare problemi prestazionali.

---

## 1.1.2 Tempo di servizio vs tempo di risposta

### Definizione

A livello di risorsa, il tempo di risposta è composto da due parti:

- **tempo di servizio (S)**: tempo impiegato a svolgere il lavoro effettivo
- **tempo di attesa (Wq)**: tempo trascorso in attesa prima di essere elaborato

Questa distinzione è fondamentale nell'analisi delle prestazioni.

---

### Relazione

Il tempo di risposta (Response Time):

- include sia `esecuzione` che `attesa`
- esso aumenta quando si formano code

Anche se il tempo di servizio rimane costante:

- il tempo di risposta può aumentare significativamente a causa dell'attesa

Questo è uno dei motivi principali per cui i sistemi degradano sotto carico.

---

### Significato pratico

Un sistema lento spesso non è tale perché il lavoro da compiere è "dispendioso", ma perché il lavoro da compiere è in attesa di risorse disponibili.

All’aumentare del carico:

- le code crescono
- l’attesa domina
- il tempo di risposta degrada

Questa scomposizione è formalizzata come:

→ 1.2.3 Tempo di servizio vs tempo di risposta

---

### Interpretazione pratica

Separare il `tempo di servizio` dal `tempo di risposta` consente:

- identificare se il sistema è limitato dalla CPU o dalle code
- distinguere tra costo di elaborazione e contesa sulle risorse
- comprendere se l’ottimizzazione deve agire sull’esecuzione o sull’attesa

In molti sistemi reali, i problemi di latenza sono causati principalmente dall’accodamento piuttosto che dal calcolo.

---

## 1.1.3 Sistemi sotto carico

### Definizione

Un sistema sotto carico elabora un flusso continuo di richieste in ingresso.

Il carico è tipicamente espresso come:

- richieste al secondo
- utenti concorrenti
- transazioni al secondo

Il carico definisce le condizioni operative in cui le prestazioni devono essere valutate.

---

### Comportamento

All’aumentare del carico:

- l’utilizzo delle risorse aumenta
- le code iniziano a formarsi
- la latenza aumenta
- il throughput alla fine si stabilizza o degrada

Questi effetti non sono lineari e dipendono dal design del sistema e dai vincoli delle risorse.

---

### Osservazione chiave

I sistemi non degradano in modo lineare.

A basso carico:

- le prestazioni sono stabili

Vicino alla saturazione:

- piccoli aumenti di carico possono causare importanti incrementi in termini di latenza

Questo comportamento non lineare è una caratteristica chiave dei sistemi reali.

---

### Interpretazione pratica

Comprendere il comportamento del sistema sotto carico è essenziale per:

- capacity planning
- test delle prestazioni
- diagnosi dei problemi di latenza

Esso puo' spiegare le ragioni del perché i sistemi possono apparire stabili nei test ma fallire con un carico di produzione leggermente più elevato.

---

## 1.1.4 Saturazione e colli di bottiglia

### Saturazione

Una risorsa è satura quando è occupata per la maggior parte o per tutto il tempo.

Esempi tipici:

- CPU al 100% (o quasi...)
- pool di thread completamente utilizzato
- pool di connessioni esaurito

La saturazione indica che una risorsa non può gestire ulteriore domanda senza subire una degradazione.

---

### Collo di bottiglia

Il collo di bottiglia (bottleneck) è la risorsa che limita il throughput del sistema.

Caratteristiche:

- massimo utilizzo
- code più lunghe
- contributo dominante al tempo di risposta

Il collo di bottiglia determina la capacità complessiva del sistema.

---

### Significato pratico

Migliorare risorse che non sono problematiche (colli di bottiglia) ha poco o nessun effetto.

I miglioramenti delle prestazioni richiedono:

- identificare il collo di bottiglia
- ridurne la domanda o aumentarne la capacità

Questo è un principio chiave nell'ingegneria delle prestazioni.

---

### Interpretazione pratica

Nei sistemi complessi:

- più risorse possono sembrare limitanti
- ma tipicamente solo una limita il throughput in un dato momento

Identificare correttamente il collo di bottiglia è essenziale per evitare ottimizzazioni inefficaci.

---

## 1.1.5 Perché i sistemi rallentano

### Meccanismi comuni

Il degradamento delle prestazioni è solitamente indotto da un numero limitato di fattori:

- accodamento dovuto alla saturazione
- contesa su risorse condivise
- uso inefficiente delle risorse
- dipendenze esterne che diventano lente

Questi meccanismi spesso interagiscono e si amplificano a vicenda.

---

### Effetto dell'accodamento

Quando l'utilizzo di una risorsa si avvicina ai sui limiti:

- il tempo di attesa aumenta rapidamente
- il tempo di risposta è dominato dall'accodamento

Questo comportamento è strettamente correlato all'utilizzo e agli effetti di accodamento:

→ 1.2.2 Legge di Utilizzazione

---

### Effetti di amplificazione

Alcuni pattern amplificano i problemi di prestazioni:

- i retry aumentano il carico su sistemi già saturi
- i timeout portano a lavoro duplicato
- dipendenze a cascata propagano i ritardi

Questi effetti possono trasformare un carico moderato in un degrado severo.

---

### Interpretazione pratica

Il degrado delle prestazioni è raramente causato da un solo, singolo fattore.

Piuttosto, esso emerge da:

- interazioni tra componenti
- accumulo del tempo di attesa
- cicli di feedback sotto carico

Da qui deriva la possibilità di una diagnosi efficace.

---

### Conclusione pratica

La maggior parte dei problemi di prestazioni non è causata da una singola operazione problematica o lenta, ma da:

- interazioni tra componenti
- accumuli dei tempi di attesa
- condizioni di sovraccarico

Comprendere questi meccanismi è essenziale prima di applicare formule o eseguire test.

---

### Idea chiave

Le prestazioni di un sistema sono determinate dalle interazioni tra carico di lavoro, risorse e concorrenza.

La comprensione di queste interazioni è il fondamento dell'ingegneria delle prestazioni.

---

◀ Note di Performance Engineering | ▲ Index | 1.2 – Metriche e formule di base ▶

# 1.2 – Metriche e formule di base

Questo documento presenta un riferimento sintetico delle principali formule utilizzate nella **performance engineering applicativa + di sistema**.

Queste formule formalizzano i concetti introdotti in:

→ 1.1 Fondamenti

Esse dovrebbero esser lette come complemento al modello concettuale, non in modo isolato.

Forniscono la base quantitativa utilizzata per ragionare sul comportamento dei sistemi, validare ipotesi e interpretare i risultati dei test prestazionali.

## Indice



---

## Notazione (tipica)

| Symbol | Definition |
|---|---|
| `X` or `λ` | **throughput** / tasso di arrivo (richieste al secondo) |
| `R` or `W` | **tempo di risposta** / tempo nel sistema (secondi) |
| `S` | **tempo di servizio** su una risorsa (secondi per richiesta) |
| `U` | **utilizzazione** di una risorsa (0–1) |
| `L` | **concorrenza media** / richieste in flight (conteggio) |
| `V` | **numero medio di visite** a una risorsa per richiesta |
| `D` | **domanda di servizio** su una risorsa (secondi per richiesta) |

Questa notazione è utilizzata in modo coerente in tutta la guida e consente di applicare le formule in modo uniforme in contesti differenti.

---

## 1.2.1 Legge di Little (concorrenza a livello di sistema)

### Definizione

Questa legge mette in relazione la **concorrenza** media con il **throughput** e il **tempo nel sistema**.

### Formula

$L = \lambda \cdot W$

### Dove

- `L` = numero medio di richieste nel sistema (in-flight / concorrenza)
- `λ` = tasso di arrivo / throughput (richieste/s)
- `W` = tempo medio nel sistema (s) (spesso il tempo medio di risposta end-to-end)

### Significato pratico

Se si conosce il `throughput` e il `tempo medio di risposta`, si puo' stimare il numero di richieste che sono, contemporaneamente, "in flight" sul sistema.

Questo rende la Legge di Little uno degli strumenti più utili per ragionare sul carico e sulla concorrenza di un sistema.

### Esempio

Se `λ = 200 req/s` e `W = 0.15 s`:

$L = 200 \cdot 0.15 = 30$

In media ci sono circa **30** richieste in flight.

---

### Interpretazione pratica

La Legge di Little collega tre grandezze osservabili:

- throughput
- latenza
- concorrenza

Questo consente di:

- stimare la concorrenza a partire da misurazioni
- validare il comportamento del sistema
- rilevare incoerenze nelle metriche

Questa legge è estensivamente utilizzata nella performance engineering, nel capacity planning e nella diagnostica dei sistemi.

---

## 1.2.2 Legge di Utilizzazione (tempo occupato a livello di risorsa)

### Definizione

L'utilizzazione è la **frazione di tempo** in cui una *singola risorsa* è occupata durante un intervallo fisso di tempo (tipicamente 1 secondo).
Essa misura la "percentuale di tempo occupato".

### Formula

$U = X \cdot S$

### Dove

- `U` = utilizzazione (0–1)
- `X` = throughput osservato da quella risorsa (req/s)

- `S` = tempo medio di servizio su quella risorsa (s/req)

### Risorsa

Una **singola unità di servizio**, ad es. core CPU, thread/worker, connessione DB, ecc.

### Esempio

Un worker DB gestisce `50 req/s`, ogni query richiede `10 ms = 0.01 s`:

$U = 50 \cdot 0.01 = 0.5 \Rightarrow 50\%$

Interpretazione: la risorsa è occupata **0.5 secondi per secondo**.

---

### Interpretazione pratica

L'utilizzazione è un indicatore chiave della saturazione di una risorsa.

Quando l'utilizzazione si avvicina a 1:

- l'accodamento (Queueing) aumenta
- la latenza cresce in modo non lineare
- la stabilità del sistema diminuisce

Questo la rende uno dei segnali più importanti nella diagnosi dei colli di bottiglia (bottlenecks).

---

## 1.2.3 Tempo di servizio vs tempo di risposta (accodamento)

### Definizione

Il tempo di risposta (Response Time) su una risorsa include:

- il tempo di servizio (lavoro effettivo)
- il tempo di coda (attesa)

### Formula

$R = S + W_q$

### Dove

- `R` = tempo di risposta sulla risorsa
- `S` = tempo di servizio
- `W_q` = tempo di attesa in coda

### Significato pratico

Quando l'utilizzazione si avvicina alla saturazione, l'accodamento (Queueing) cresce in modo non lineare e **domina** il tempo di risposta, causando **latenza long-tail**.

---

### Interpretazione pratica

Questa formula spiega perché i sistemi rallentano sotto carico anche quando il costo computazionale non cambia.

In molti sistemi reali:

- il tempo di servizio rimane relativamente stabile
- il tempo di attesa aumenta rapidamente

Di conseguenza:

- il tempo di risposta è dominato dall'accodamento
- la latenza diventa imprevedibile

Questo è un punto chiave nella diagnosi dei problemi prestazionali.

---

## 1.2.4 Domanda di servizio (visite × tempo di servizio)

### Definizione

Servizio totale richiesto a una risorsa per richiesta, tenendo conto di visite multiple.

### Formula

$D = V \cdot S$

### Dove

- `D` = domanda di servizio sulla risorsa (s)
- `V` = visite medie alla risorsa per richiesta
- `S` = tempo di servizio per visita (s)

### Esempio

Una richiesta esegue `V = 3` query DB, ciascuna richiede `S = 5 ms = 0.005 s`:

$D = 3 \cdot 0.005 = 0.015$ s = 15 ms

---

### Interpretazione pratica

La domanda di servizio rappresenta il lavoro totale richiesto a una risorsa per ogni richiesta.

È particolarmente utile per:

- identificare le risorse maggiormente utilizzate
- stimare i limiti di capacità
- comprendere il comportamento in scalabilità

Ridurre la domanda di servizio è spesso più efficace che aumentare la capacità grezza.

---

## 1.2.5 Throughput

### Definizione

Richieste completate per unità di tempo.

### Formula

**Formula:** `X = N / T`

### Dove

- `N` = numero di richieste completate
- `T` = finestra di osservazione (secondi)

---

### Interpretazione pratica

Il `throughput` è uno degli indicatori principali delle prestazioni di un sistema.

Riflette la capacità del sistema di elaborare lavoro.

Tuttavia, il throughput deve sempre essere interpretato insieme a:

- latenza
- tasso di errore

- utilizzazione delle risorse

Un throughput elevato, da solo, non garantisce un comportamento accettabile del sistema.

---

## 1.2.6 Tasso di errore

### Definizione

Frazione di richieste che falliscono (timeout, 5xx, ecc.).

### Formula

**Formula:** `ErrorRate = (N_err / N_total) × 100%`

---

### Interpretazione pratica

Il tasso di errore riflette l'affidabilità del sistema sotto carico.

Un aumento del tasso di errore indica spesso:

- condizioni di sovraccarico
- esaurimento delle risorse
- instabilità

Il tasso di errore dovrebbe sempre essere monitorato insieme a latenza e throughput.

---

## 1.2.7 Percentili (p50, p95, p99)

### Definizione

Il percentile `p`-esimo è il valore al di sotto del quale ricade **il p% delle osservazioni**.

- `p50` ≈ mediana ("richiesta tipica")
- `p95` = soglia per il 5% più lento
- `p99` = soglia per l'1% più lento

I percentili catturano la **distribuzione** e il **comportamento della coda** meglio delle medie.

---

### Interpretazione pratica

I percentili sono essenziali per comprendere l'esperienza reale dell'utente.

In molti sistemi:

- la latenza media appare accettabile
- la latenza di coda (p95/p99) è significativamente peggiore

Questa differenza è critica per la valutazione del sistema e la definizione degli SLO.

---

#### 1.2.7.1 Come calcolare un percentile (campione ordinato)

Date `N` valori ordinati in modo crescente:

$v_1 \le v_2 \le \ldots \le v_N$

Calcola la posizione teorica:

**Formula:** `P = (p / 100) × (N + 1)`

- Se `P` è un intero → percentile = `v_P`

- In caso contrario, poni `k = floor(P)` e `δ = P - k` (parte frazionaria), quindi interpola:

Percentile($p$) $\approx v_k + \delta \cdot (v_{k+1} - v_k)$

> Nota: le definizioni di percentile variano leggermente tra i diversi strumenti. Questo metodo è un approccio comunemente utilizzato.

---

#### 1.2.7.2 Interpretazione vs media (perché le code contano)

- Se `p50` è molto più basso della media, la distribuzione è **asimmetrica a destra** (poche richieste lente gonfiano la media).
- Se `p95` o `p99` è molto al di sopra della media, hai **latenza long-tail**.

Un pattern tipico:

- la media sembra "accettabile"
- `p95/p99` sono negativi

→ l'esperienza utente è degradata per una frazione non trascurabile di utenti e gli SLO sono a rischio.

---

#### Interpretazione pratica

I percentili mettono in evidenza comportamenti che le medie nascondono.

Sono essenziali per:

- definire gli obiettivi di livello di servizio (SLO)
- rilevare problemi di latenza di coda
- comprendere il comportamento nel caso peggiore

Ignorare i percentili porta spesso a conclusioni scorrette sulle prestazioni del sistema.

---

## 1.2.8 CDF empirica (soglia → percentuale)

#### Definizione

Data una soglia `t`, la funzione di distribuzione cumulativa empirica (CDF) indica la frazione di campioni pari o inferiori a `t`.

#### Formula

**Formula:** `F(t) = count(x_i ≤ t) / N`

#### Significato pratico

La CDF risponde alla domanda: "Se il mio SLO è `200 ms`, quale % di richieste lo rispetta?"

I percentili rispondono alla domanda inversa: "Quale soglia corrisponde al 95% delle richieste?"

---

#### Interpretazione pratica

CDF e percentili sono viste complementari degli stessi dati.

- CDF: data una soglia → quale frazione la rispetta
- Percentile: data una frazione → quale soglia le corrisponde

Entrambi sono utili per l'analisi delle prestazioni e la validazione degli SLO.

---

## 1.2.9 Latenza long-tail (che cos'è)

### Definizione

Una piccola frazione di richieste (es. 5% o 1%) è **molto più lenta** della maggioranza.

---

### Perché la coda "domina"

- Gli SLO sono tipicamente definiti su `p95/p99`, quindi le code determinano il pass/fail.
- Nei sistemi distribuiti, la dipendenza più lenta determina spesso la latenza end-to-end.
- Gli eventi di coda sono frequentemente guidati da **contesa/accodamento**.

---

### Cause comuni (alto livello)

- saturazione del thread pool / connection pool (accodamento)
- contesa su lock / punti caldi di sincronizzazione
- query DB lente, indici mancanti, attese su lock
- retry + timeout che amplificano la latenza di coda
- hot key nelle cache / carico non uniforme sugli shard
- pause GC / pressione di memoria (stop-the-world)
- jitter di rete / perdita di pacchetti / ritrasmissioni
- picchi di I/O disco, compactions, flush fsync/wal

---

### Interpretazione pratica

La latenza long-tail è uno degli aspetti più critici delle prestazioni di un sistema.

Spiega perché:

- le metriche medie possono apparire accettabili
- l'esperienza utente è comunque degradata

Gestire la latenza di coda è spesso più importante che migliorare la prestazione media.

---

## 1.2.10 Checklist rapida (cosa misurare nei test)

- Latenza: `p50/p90/p95/p99`
- Throughput: `RPS/TPS`
- Tasso di errore: `timeouts/5xx`
- Utilizzazione: CPU, memoria, DB, pool
- Lunghezze delle code: thread pool, connection pool, backlog dei messaggi
- Tempi delle dipendenze: DB/Redis/API esterne

---

### Interpretazione pratica

Queste metriche costituiscono il set minimo richiesto per comprendere il comportamento del sistema durante i test prestazionali.

Esse consentono di:

- identificare i colli di bottiglia
- rilevare instabilità
- correlare il carico di lavoro con il comportamento del sistema

Misurare solo un sottoinsieme di queste metriche porta spesso a un'analisi incompleta o fuorviante.

---

### Idea chiave

Le formule non sono astrazioni isolate.

Sono strumenti utilizzati per spiegare il comportamento osservato e validare i modelli del sistema.

La loro valutazione è un elemento essenziale della performance engineering.

---

◀ 1.1 – Fondamenti | ▲ Index | 1.3 – Lavoro di un performance engineer ▶

# 1.3 – Lavoro di un performance engineer

Questa sezione descrive che cosa sia, in pratica, la performance engineering e come venga applicata ai sistemi reali.

## Indice



---

## 1.3.1 Che cos'è la performance engineering (in pratica)

### Definizione

La performance engineering è la disciplina che consiste nel comprendere, misurare e controllare il modo in cui un sistema si comporta sotto carico.

Essa non si limita al performance testing, né a specifici strumenti o tecnologie.

Essa si riferisce piuttosto ad una metodologia complessiva di ragionamento sui sistemi sotto carico o, eventualmente, sotto stress.

Essa si concentra sul comportamento complessivo del sistema e non su metriche isolate o su singoli componenti.

---

### Prestazioni e requisiti non funzionali

La performance engineering non si concentra su una singola proprietà.

Quando un sistema viene esercitato sotto carico, diventa visibile un sottoinsieme di **requisiti non funzionali (NFR)**:

- latenza e throughput (prestazioni)
- scalabilità (verticale e orizzontale)
- stabilità e resilienza sotto stress
- utilizzo delle risorse ed efficienza
- limiti di capacità

Queste proprietà non sono tra loro indipendenti.

Emergono tutte insieme man mano che il sistema viene portato ai suoi limiti.

Il carico agisce come una **forcing function** che rivela il modo in cui il sistema si comporta.

Un sistema che appare perfettamente equilibrato a basso carico può mostrare un comportamento completamente diverso quando viene stressato.

---

### Che cosa osserva realmente la performance engineering

Sotto carico, un sistema rivela:

- come il lavoro attraversa i suoi componenti
- come vengono consumate le risorse

- dove compaiano contese (contention)
- dove si formano le code (queueing)
- quali sono i limiti che vengono raggiunti per primi

Questo richiede:

- comprendere il modello del sistema (→ 1.1 Fondamenti)
- misurare le metriche chiave (→ 1.2 Metriche e formule di base)
- identificare i fattori limitanti

L'obiettivo evidentemente non è solo quello di osservare il comportamento del sistema, ma anche di spiegarlo.

---

### Non solo testing

La performance engineering viene spesso ridotta al solo **load testing**.

In pratica, la fase di testing è solo una parte del lavoro.

I test vengono utilizzati per:

- esporre il comportamento del sistema
- validare ipotesi
- riprodurre problemi

Ma la performance engineering comprende anche:

- analizzare il design del sistema
- investigare problemi di produzione
- dimensionare le risorse (heap, pool, thread, connessioni)
- spiegare il comportamento osservato

Il Testing senza l'analisi produce dati senza comprensione.

---

### Prospettiva pratica

Negli scenari reali, il lavoro coinvolge tipicamente:

- preparare e calibrare gli ambienti di test
- Interpretare i requisiti non funzionali (NFRs)
- individuare e definire scenari (significativi) di test rispetto agli NFRs
- validare il comportamento con casi d'uso controllati
- applicare carico o stress per far emergere i problemi (spesso in white-box)
- identificare e correggere i colli di bottiglia
- dimensionare i componenti del sistema (CPU, memoria, pool, limiti di concorrenza)
- mettere a punto configurazioni e parametri (Tuning)
- eseguire benchmark per stabilire punti di riferimento
- eseguire test di lunga durata (soak / endurance) per validare la stabilità nel tempo

Queste attività non sono isolate.

Fanno parte di un processo continuo orientato alla comprensione delle possibilità e dei limiti del sistema.

---

### Idea chiave

La performance engineering non consiste (solamente) nel rendere un sistema più veloce e prestazionale.

Comprende invece un insieme ti attività e tasks, atti al comprendere come un sistema si comporta sotto carico di lavoro, e all'assicurarsi che esso rimanga:

- prevedibile

- stabile
- scalabile

La maggior parte dei problemi non è causata da una singola operazione “lenta”, ma da:

- interazioni tra componenti
- accumulo dei tempi di attesa
- saturazione di risorse condivise

Questi meccanismi costituiscono, insieme, il nucleo della performance engineering.

---

## 1.3.2 Workflow tipico

La performance engineering è un processo iterativo in cui il sistema viene progressivamente esercitato, analizzato, stabilizzato e compreso sotto livelli crescenti di carico.

L’obiettivo non è solo rilevare problemi, ma costruire un modello affidabile di come il sistema si comporta in condizioni realistiche (e limite) di produzione.

---

### 1.3.2.1 Preparazione e calibrazione dell’ambiente

- verificare e allineare l’ambiente di test alle caratteristiche della produzione (per quanto possibile)
- verificare le configurazioni (CPU, memoria, pool, connessioni)
- garantire osservabilità (metriche, log, trace)

Obiettivo:

- stabilire una baseline affidabile
- garantire la ripetibilità dei risultati

Senza calibrazione, le misurazioni sono difficili (o impossibili) da interpretare e i confronti diventano quantomeno inaffidabili.

---

### 1.3.2.2 Definizione dei casi d’uso e modellazione del workload

Prima di applicare carico al sistema, il workload deve essere definito.

Un sistema non viene testato in isolamento, ma attraverso le richieste che elabora.

Questo richiede l’identificazione precisa di:

- i percorsi critici utente e di sistema
- le operazioni tipiche (read, write, batch, background job)
- la frequenza relativa di ciascuna operazione
- i pattern di concorrenza

Un workload realistico include:

- un mix di casi d’uso (Use Cases)
- una distribuzione pesata (es. percentuali di traffico)
- diversi tipi di richieste e diversi costi

La definizione del workload è uno dei passaggi più critici e va fatto in stretta collaborazione con chi definisce i requisiti non funzionali (NFRs).

Un workload scorretto porta a conclusioni fuorvianti o addirittura del tutto inutili.

---

#### Requisiti non funzionali (NFR)

In parallelo con la definizione del workload, i **requisiti non funzionali** devono essere chiariti.

Essi definiscono ciò che viene considerato un **comportamento accettabile del sistema**.

Esempi tipici:

- obiettivi di throughput (es. 30 req/s)
- livelli di concorrenza (es. 500 utenti concorrenti)
- obiettivi di latenza (es. p95 < 200 ms)
- soglie del tasso di errore
- vincoli sull'utilizzo delle risorse

Gli NFR possono essere:

- definiti esplicitamente dagli stakeholder
- definiti solo parzialmente
- mancanti o incoerenti

In tutti i casi, devono essere:

- riesaminati
- validati
- resi misurabili

---

#### Implicazione pratica

Workload e NFR devono essere allineati.

Per ciascun caso d'uso:

- il carico atteso deve essere definito
- il comportamento accettabile deve essere noto

Altrimenti:

- i risultati non possono essere valutati
- i test non possono essere considerati né riusciti né falliti

Una definizione scorretta del workload o l'assenza di NFR porta a risultati tecnicamente corretti, ma non azionabili.

---

### 1.3.2.3 Test iniziali di carico / stress (scoperta dei problemi)

La prima fase di "load test" mira a far emergere una baseline di riferimento ed eventuali problemi principali.

Obiettivi tipici:

- identificare colli di bottiglia evidenti
- rilevare errori funzionali sotto carico
- far emergere instabilità (timeout, crash, saturazione)

Questa fase è spesso:

- esplorativa
- iterativa
- parzialmente white-box (usando visibilità interna)

L'obiettivo è la scoperta, non la precisione.

---

### 1.3.2.4 Analisi e identificazione dei colli di bottiglia

Una volta che i problemi siano emersi, il sistema deve essere analizzato nel dettaglio.

Questo comporta:

- correlare le metriche (latenza, throughput, utilizzazione)
- identificare dove viene speso il tempo
- localizzare i punti di saturazione e le code

Domande tipiche:

- quale risorsa è satura?
- dove si accumula la latenza?
- che cosa limita il throughput?

Questo passaggio si basa su:

→ 1.1 Fondamenti
→ 1.2 Metriche e formule di base

---

#### 1.3.2.5 Correzioni e validazione iterativa

Dopo avere identificato i colli di bottiglia, debbono essere applicati i correttivi.

Queste possono includere:

- modifiche al codice
- aggiornamenti di configurazione
- aggiustamenti delle risorse (scalabilità verticale/orizzontale)

Ogni correzione deve essere validata rieseguendo i test.

Questo crea un ciclo iterativo:

- **Test → Analizza → Correggi → Testa** di nuovo

L'obiettivo è stabilizzare progressivamente il sistema.

---

#### 1.3.2.6 Validazione intermedia (baseline stabile)

Prima di passare ai test ulteriori e di lunga durata, il sistema deve raggiungere una baseline stabile.

Questo significa:

- nessun errore critico sotto il carico atteso
- comportamento prevedibile
- latenza e tassi di errore sotto controllo

Questa fase garantisce che:

- i problemi principali siano risolti
- i risultati siano riproducibili

---

#### 1.3.2.7 Validazione di lunga durata (soak / endurance)

Una volta ci si sia assicurati che il sistema è stabile, esso deve essere investigato sulla durata.

Questa fase valuta il comportamento del sistema sotto un carico di lavoro sostenuto nel tempo.

Obiettivi tipici:

- rilevare memory leak lenti
- osservare l'accumulo di risorse (thread, connessioni, buffer)
- identificare degradazioni prestazionali sulla durata
- validare la stabilità di lungo periodo

Questa fase è essenziale perché alcuni problemi:

- non compaiono immediatamente

- emergono soltanto dopo esercizio prolungato

I risultati di questa fase hanno un impatto diretto su:

- dimensionamento del sistema
- capacity planning
- configurazione di runtime

---

### 1.3.2.8 Dimensionamento e definizione della capacità

Sulla base delle osservazioni precedenti, e anche a partire da eventuali test unitari successivi alla fase di stabilizzazione della baseline, i componenti del sistema vengono dimensionati.

Questa fase include:

- configurazione di heap e memorie
- thread pool e connection pool
- limiti di concorrenza
- dimensionamento dell'infrastruttura
- clustering

L'obiettivo è definire:

- quanto carico il sistema può gestire
- in quali condizioni esso rimane stabile
- quali margini eventuali sono richiesti

Il dimensionamento deve basarsi sul comportamento osservato, non su ipotesi.

---

### 1.3.2.9 Tuning

Una volta definito il dimensionamento, il tuning rifinisce il comportamento del sistema.

Aree tipiche:

- parametri del garbage collector
- scheduling dei thread e dimensionamento dei pool
- impostazioni del database e delle connessioni
- strategie di caching

Il tuning mira a:

- ridurre la latenza
- migliorare la stabilità
- ottimizzare l'utilizzo delle risorse

Spesso è iterativo e dipendente dal contesto specifico.

---

### 1.3.2.10 Verifica e regressione

Dopo la fase di tuning, il sistema deve essere nuovamente validato.

Questo include:

- rieseguire gli scenari chiave
- verificare che i miglioramenti siano effettivi
- assicurare che non vengano introdotte regressioni

Questa fase garantisce coerenza e affidabilità.

---

#### 1.3.2.11 Benchmarking e punti di riferimento

Infine, vengono stabiliti i benchmark.

Essi forniscono:

- metriche prestazionali di riferimento
- punti di confronto tra versioni
- validazione rispetto alle aspettative

I benchmark non sono obiettivi in sé.

Sono utilizzati per:

- comprendere il comportamento del sistema
- seguirne l'evoluzione nel tempo

---

#### Idea chiave

La performance engineering si sviluppa secondo un ciclo iterativo:

- **definisci il workload → testa → analizza → correggi → valida → ottimizza**

L'obiettivo non è solo migliorare le prestazioni, ma comprendere i limiti del sistema e garantire un comportamento prevedibile sotto carico.

---

## 1.3.3 Black-box vs white-box

La performance engineering può essere affrontata da due prospettive complementari:

- **black-box** (osservazione esterna)
- **white-box** (osservazione interna)

Entrambe sono necessarie per comprendere il comportamento del sistema sotto carico di lavoro.

---

#### 1.3.3.1 Approccio black-box

In un approccio black-box, il sistema viene osservato dall'esterno.

Viene misurato solo il comportamento visibile esternamente:

- tempo di risposta
- throughput
- tasso di errore

L'implementazione interna non viene presa in considerazione.

---

#### Ciò che fornisce

L'osservazione black-box consente di:

- validare il comportamento del sistema dal punto di vista dell'utente
- misurare le prestazioni end-to-end
- rilevare errori visibili sotto carico

Essa risponde a domande quali:

- Il sistema è sufficientemente veloce?
- Gestisce il carico atteso?
- Fallisce sotto stress?

---

### Limiti

Il solo black-box non può spiegare:

- dove eventualmente viene piu spesso speso del tempo
- quale risorsa è satura
- perché le prestazioni degradano

Mostra i sintomi, non le cause.

---

### 1.3.3.2 Approccio white-box

In un approccio white-box, viene osservato il comportamento interno del sistema.

Questo include:

- utilizzazione delle risorse (CPU, memoria, disco, rete)
- thread pool e connection pool
- code interne
- tempi a livello di componente

L'osservazione white-box fornisce un livello di **introspezione nell'esecuzione del sistema**.

In molti casi, questo include visibilità vicina al livello del codice:

- tempi a livello di metodo
- call path e flussi di esecuzione
- hotspot (metodi lenti o eseguiti frequentemente)
- pattern di allocazione e comportamento della memoria
- contesa sui lock e punti di sincronizzazione

---

### Ciò che fornisce

L'osservazione white-box consente di:

- identificare i colli di bottiglia
- comprendere dove viene speso il tempo
- rilevare contesa (contentio) e accodamento (queueing)
- analizzare la saturazione delle risorse

Essa risponde a domande quali:

- Quale componente è lento?
- Dove si accumula la latenza?
- Che cosa limita il throughput?
- Quale parte dell'esecuzione è responsabile del rallentamento?

---

### Limiti

Il solo white-box non garantisce:

- un comportamento end-to-end corretto
- un'esperienza utente accettabile

Un sistema può apparire internamente efficiente ma fallire comunque sotto condizioni di workload reale.

---

### 1.3.3.3 Osservabilità e strumentazione

L'osservabilità fornisce i dati necessari per l'analisi white-box.

Essa include tipicamente:

- metriche di sistema e applicative (es. utilizzo CPU, latenza, throughput)
- log (eventi, errori, cambiamenti di stato)
- trace (flusso delle richieste tra componenti)
- application performance monitoring (APM)

Queste fonti forniscono visibilità continua sul comportamento del sistema.

---

#### Artifact diagnostici

Oltre all'osservabilità continua, un'analisi più profonda si basa spesso su artifact diagnostici.

Questi vengono tipicamente raccolti on demand e forniscono uno snapshot dello stato del sistema.

Esempi comuni includono:

- thread dump (stati dei thread, lock, contesa)
- heap dump (uso della memoria, retention degli oggetti, leak)
- snapshot di profiling (profiling CPU e allocazioni)
- core dump (analisi dei guasti a livello di processo)

Questi artifact consentono di:

- ispezionare lo stato interno dell'esecuzione
- identificare thread bloccati e deadlock
- analizzare memory leak e retention path
- investigare in dettaglio anomalie prestazionali

Sono in genere più pesanti e intrusivi rispetto agli strumenti di osservabilità, e vengono utilizzati in modo selettivo durante la diagnostica.

---

#### 1.3.3.4 Combinare entrambi gli approcci

Una performance engineering efficace richiede la combinazione di entrambe le prospettive.

Workflow tipico:

- usare il black-box per rilevare i problemi
- usare il white-box per spiegarli
- validare nuovamente i miglioramenti con il black-box

Questo crea un ciclo di feedback:

- **osserva → analizza → correggi → valida**

---

#### Idea chiave

L'osservazione **black-box** rivela che esiste un problema.

L'osservazione **white-box** spiega perché esiste.

Entrambe sono necessarie per comprendere e controllare il comportamento del sistema sotto carico.

---

## 1.3.4 Load testing vs diagnostica

Il load testing e la diagnostica vengono spesso confusi.

Essi servono scopi differenti e operano a livelli differenti.

Entrambi sono necessari per comprendere il comportamento del sistema sotto carico di lavoro.

---

#### 1.3.4.1 Load testing

Il load testing applica un workload controllato al sistema.

Viene utilizzato per:

- osservare il comportamento in condizioni specifiche
- misurare latenza, throughput e tassi di errore
- validare ipotesi su capacità e scalabilità

Il load testing opera principalmente a livello **black-box**:

- le richieste vengono generate esternamente
- le risposte vengono misurate esternamente

---

#### Ciò che fornisce

Il load testing risponde a domande quali:

- Il sistema può gestire il carico atteso?
- Cosa accade quando il carico aumenta?
- Quando le prestazioni degradano?
- Qual è il throughput massimo sostenibile?

---

#### Limiti

Il solo load testing non spiega:

- perché il sistema rallenta
- quale componente è responsabile
- come vengono utilizzate internamente le risorse

Rivela il comportamento, ma non le cause.

---

#### 1.3.4.2 Diagnostica

La diagnostica investiga il comportamento interno del sistema.

Viene utilizzata per:

- identificare i colli di bottiglia
- comprendere i percorsi di esecuzione
- analizzare l'utilizzo delle risorse
- spiegare i problemi prestazionali osservati

La diagnostica opera a livello **white-box**:

- vengono analizzate metriche interne
- vengono ispezionati trace e percorsi di esecuzione
- possono essere raccolti artifact diagnostici

---

#### Ciò che fornisce

La diagnostica risponde a domande quali:

- Dove viene speso il tempo?
- Quale risorsa è satura?
- Quale componente è responsabile della latenza?
- Che cosa causa il degrado delle prestazioni?

---

### Strumenti e tecniche

La diagnostica si basa tipicamente su:

- metriche, log e trace
- application performance monitoring (APM)
- thread dump e heap dump
- profiling e analisi dell'esecuzione

---

### Limiti

La diagnostica senza load testing può non cogliere:

- condizioni di workload reale
- interazioni tra componenti
- comportamento sotto stress

Può spiegare un problema, ma non necessariamente riprodurlo.

---

### 1.3.4.3 Relazione tra load testing e diagnostica

Il load testing e la diagnostica devono essere combinati.

Workflow tipico:

- applicare carico per esporre il comportamento
- usare la diagnostica per analizzare lo stato interno
- applicare correzioni
- validare di nuovo con il load testing

Questo crea un ciclo:

- osserva → spiega → correggi → valida

---

### Idea chiave

Il load testing rivela che un problema esiste.

La diagnostica spiega perché esiste.

Nessuno dei due è sufficiente da solo.

La comprensione del comportamento del sistema richiede entrambi.

---

## 1.3.5 Ciò che conta davvero (e ciò che non conta)

La performance engineering coinvolge un insieme esteso di strumenti, metriche e tecniche.

Tuttavia, non tutte possono avere lo stesso livello di importanza in contesti eterogenei.

Capire che cosa conta è essenziale per evitare di sprecare effort e trarre conclusioni errate.

---

### Ciò che conta

Gli aspetti più importanti sono:

- **comprendere il comportamento del sistema sotto carico**
- **identificare i colli di bottiglia e i fattori limitanti**
- **utilizzare workload realistici e NFR validati**
- **ragionare sulle interazioni tra componenti**

- **misurare e interpretare correttamente i risultati**

La performance engineering riguarda principalmente:

- costruire un modello mentale del sistema
- validare quel modello attraverso osservazioni
- raffinarlo tramite iterazione

---

### Ciò che non conta (quanto sembra)

Alcuni aspetti vengono spesso enfatizzati eccessivamente:

- strumenti e framework
- metriche isolate senza contesto
- scenari di test sintetici o irrealistici
- micro-ottimizzazioni senza impatto a livello di sistema
- risultati di un singolo test presi in isolamento

Questi elementi possono essere utili, ma non sono sufficienti.

---

### Fraintendimenti comuni

Compaiono frequentemente diversi fraintendimenti:

- “Se eseguo un load test, comprendo il sistema”
- “Se la CPU è bassa, il sistema è sano”
- “Se la latenza media è accettabile, il sistema va bene”
- “Più hardware risolverà il problema”

Queste assunzioni portano spesso a conclusioni scorrette.

---

### Pensiero a livello di sistema

Le prestazioni emergono dalle interazioni:

- tra componenti
- tra workload e risorse
- tra concorrenza e accodamento

Concentrarsi su una singola parte del sistema è raramente sufficiente.

Ciò che serve è una visione globale.

---

### Implicazione pratica

Una performance engineering efficace richiede:

- porre le domande corrette
- validare le ipotesi
- correlare segnali multipli
- iterare sulla base delle evidenze

Strumenti, test e metriche supportano questo processo, ma non lo sostituiscono.

---

### Idea chiave

La performance engineering non consiste nel raccogliere dati.

Riguarda il comprendere che cosa i dati significhino.

L'obiettivo non è produrre numeri, ma spiegare il comportamento del sistema e prendere decisioni informate.

---

◀ 1.2 – Metriche e formule di base | ▲ Index | 01-04-types-of-performance-tests ▶

## 1.4 – Tipi di test prestazionali

Questo capitolo introduce le principali categorie di test prestazionali utilizzate nella performance engineering.

Ogni tipo di test prestazionale risponde a una diversa domanda sul comportamento del sistema sotto carico.

Nel loro insieme, essi aiutano a valutare prestazioni, stabilità, scalabilità, recupero e capacità del sistema in modo controllato e misurabile.

## Indice



---

## 1.4.1 Scopo del performance testing

### Definizione

Il performance testing, come già ricorsato nei precedenti paragrafi, valuta come un sistema si comporta in condizioni di workload controllato.

Esso fornisce dati misurabili su:

- latenza
- throughput
- tasso di errore
- utilizzo delle risorse

(→ 1.2 Metriche e formule di base)

Il performance testing non è quindi soltanto un'attività di misurazione, ma anche un'attività di validazione.

Viene sfruttato per comparare il comportamento atteso (definito nei NFRs) con il comportamento osservato in condizioni di workload definite.

---

### Ruolo nella performance engineering

Il performance testing non riguarda soltanto la misurazione dei risultati.

Viene utilizzato per:

- validare il comportamento del sistema nelle condizioni attese
- far emergere colli di bottiglia e limitazioni
- supportare il capacity planning
- validare decisioni architetturali

Esso fornisce anche un framework controllato per confrontare:

- versioni dello stesso sistema
- diverse configurazioni
- cambiamenti infrastrutturali
- scelte di tuning

Senza test controllati, le discussioni sulle prestazioni restano spesso basate su ipotesi piuttosto che su evidenze.

---

### Il workload come modello

Un workload di test rappresenta un modello semplificato dell'utilizzo reale.

Esso definisce:

- tasso di arrivo (richieste al secondo)
- concorrenza (numero di utenti o richieste attive)
- pattern delle richieste (distribuzione, mix di operazioni)

(→ 1.2.1 Legge di Little (concorrenza a livello di sistema))

Un workload non è lo specchio esatto dell'utilizzazione reale di produzione in sé.

È un'approssimazione pratica dei pattern di utilizzo più rilevanti.

Per questa ragione, il valore di un test prestazionale dipende fortemente da quanto realistico sia il modello di workload.

---

### Condizioni controllate

Un test prestazionale è significativo soltanto se le condizioni di esecuzione sono ben definite e controllate.

Questo include:

- la definizione del workload
- la durata del test
- l'ambiente in cui esso viene eseguito
- le metriche raccolte durante l'esecuzione

Se queste condizioni non sono chiare i risultati, pur sempre numerici, saranno scarsamente o del tutto privi di valore conoscitivo e proiettivo.

Il controllo delle condizioni iniziali è uno di quei parametri che trasforma un test da semplice esercizio a indispensabile attività ingegneristica.

---

Il performance testing è dunque il punto d'ingresso a molti dei concetti sviluppati nel proseguio di questo documento.

Come pratica complessiva di test esso fa emergere:

- effetti di accodamento e saturazione (→ 1.5 Comportamento del sistema sotto carico)
- limiti di concorrenza (→ 1.6 Concorrenza e parallelismo)
- effetti di runtime e memoria (→ 1.7 Runtime e modello di memoria)
- saturazione delle risorse (→ 1.8 Prestazioni a livello di risorsa)

Per questa ragione, il design dei test dovrebbe sempre essere collegato ad una conoscenza approfondita e complessiva del sistema.

---

### Significato pratico

Un buon test prestazionale non risponde soltanto a:

- “Quanto è veloce il sistema?”

Esso aiuta anche a rispondere a:

- “In quali condizioni il sistema rimane stabile?”
- “Che cosa cambia all’aumentare del carico?”
- “Quale limite viene raggiunto per primo?”
- “Quale tipo di degradazione compare?”

Queste domande sono essenziali nella performance engineering perché collegano la misurazione all’interpretazione.

---

#### Idea chiave

I test prestazionali sono esperimenti controllati.

Sono progettati per osservare il comportamento del sistema in specifiche condizioni di workload.

Il loro valore non risiede soltanto nelle misurazioni che producono, ma soprattutto nella comprensione che forniscono.

---

## 1.4.2 Load testing

#### Definizione

Il **load testing** valuta il comportamento del sistema sotto workload standard o tipico.

È il modo più comune e più diretto per validare che un sistema si comporti in modo accettabile in condizioni operative normali.

---

#### Obiettivo

- verificare che il sistema soddisfi i requisiti prestazionali
- validare obiettivi di latenza e throughput
- osservare l’utilizzo delle risorse in condizioni normali

Il load testing risponde alla domanda se il sistema si comporti correttamente nell’intervallo operativo che ci si aspetta supporti.

---

#### Caratteristiche

- il workload è stabile e controllato
- il sistema opera entro il suo intervallo atteso
- l’attenzione è rivolta al comportamento in regime stazionario

Lo scopo non è portare il sistema ai limiti, ma stabilire se esso si comporti correttamente sotto un carico (di produzione) per cui è stato progettato.

---

#### Esempio

Un sistema progettato per:

- 200 richieste al secondo
- latenza p95 < 300 ms

Un load test verifica che questi obiettivi siano soddisfatti.

Può anche verificare che:

- il tasso di errore rimanga basso

- il throughput rimanga stabile
- l'utilizzo delle risorse rimanga entro limiti accettabili

---

### Valore diagnostico

Il load testing fornisce una baseline:

- distribuzione normale della latenza
- utilizzo tipico delle risorse
- throughput atteso

Questa baseline è essenziale per il confronto con gli altri test.

Senza una baseline affidabile, è difficile determinare se il comportamento osservato nei test di stress, spike, soak o capacity sia anomalo o semplicemente normale per il sistema sotto analisi.

---

### Limiti del load testing

Il solo load testing non determina:

- la capacità massima del sistema
- i punti di rottura del sistema
- la stabilità di lungo periodo del runtime
- il comportamento di recupero dopo cambiamenti bruschi del carico

Un sistema può superare un load test e fallire comunque sotto sovraccarico, esecuzione prolungata o rapidi burst di traffico.

Per questa ragione, il load testing è necessario ma non sufficiente.

---

### Interpretazione pratica

Il load testing è il punto di riferimento per il resto dell'analisi prestazionale.

Esso definisce il normale comportamento operativo del sistema e permette di interpretare i test successivi nel loro contesto.

Se il sistema si comporta già male sotto il carico standard, ha poco valore passare immediatamente a tipi di test più avanzati.

---

### Idea chiave

Il load testing risponde a: *"Il sistema si comporta correttamente sotto il carico atteso?"*

Esso stabilisce la baseline rispetto alla quale tutti gli altri test prestazionali possono essere interpretati.

---

## 1.4.3 Stress testing

### Definizione

Lo **stress testing** valuta il comportamento del sistema oltre la sua capacità attesa.

Viene utilizzato per osservare che cosa accade quando il sistema viene spinto fuori dal suo intervallo operativo previsto.

---

### Obiettivo

- identificare i limiti del sistema

- osservare il comportamento sotto sovraccarico
- rilevare i modi di fallimento

Lo stress testing concerne principalmente il comportamento al limite del sistema e la degradazione delle capacità lavorative sotto carico eccedente gli standard previsti.

---

### Caratteristiche

- il workload aumenta oltre i livelli normali
- il sistema si avvicina o raggiunge la saturazione

(→ 1.8 Prestazioni a livello di risorsa)

Il sovraccarico può essere applicato progressivamente o mantenuto a un livello chiaramente eccessivo.

In entrambi i casi, l'obiettivo è far emergere il modo in cui il sistema si comporta quando la domanda supera la capacità.

---

### Effetti osservabili

- la latenza aumenta rapidamente
- il throughput si appiattisce o diminuisce
- il tasso di errore aumenta

(→ 1.5.3 Degradazione non lineare)
(→ 1.5.4 Collasso del throughput)

Effetti aggiuntivi possono includere:

- accumulo di code
- amplificazione dei timeout
- esaurimento dei pool
- utilizzo instabile delle risorse
- sovraccarico guidato dai retry

---

### Valore diagnostico

Lo stress testing rivela:

- colli di bottiglia
- punti di saturazione
- stabilità del sistema sotto pressione

È particolarmente utile per comprendere se la degradazione sia graduale, brusca, recuperabile o instabile.

Due sistemi con risultati simili nei load test possono comportarsi in modo molto diverso sotto stress.

---

### Comportamento in rottura

Un aspetto importante dello stress testing non è soltanto se e quando il sistema fallisca, ma come fallisca.

Domande rilevanti includono:

- La latenza aumenta prima che compaiano errori?
- Gli errori compaiono gradualmente o improvvisamente?
- Il throughput si appiattisce prima di collassare?
- Il sistema recupera quando il carico viene ridotto?

Queste domande contano operativamente perché il sovraccarico è uno scenario realistico nei sistemi di produzione.

### Distinzione dal capacity testing

Lo stress testing e il capacity testing sono collegati, ma differenti.

- lo **stress testing** si concentra sul comportamento in sovraccarico e sui modi di fallimento
- il **capacity testing** si concentra sul massimo carico sostenibile che soddisfa ancora i requisiti

Lo stress testing continua quindi oltre l'intervallo operativo accettabile per esaminare degradazione e rottura.

---

### Interpretazione pratica

Lo stress testing è utile quando la domanda ingegneristica non è soltanto:

- "Quanto carico può supportare il sistema?"

ma anche:

- "Che cosa accade dopo che non può più supportare il carico?"
- "Degrada in modo graduale?"
- "Può recuperare in modo pulito?"

Queste sono domande essenziali per resilienza e robustezza operativa.

---

### Idea chiave

Lo stress testing risponde a: *"Che cosa accade quando il sistema viene spinto oltre i suoi limiti?"*

Esso rivela come il sistema degrada, come fallisce e quanto sovraccarico può tollerare prima di diventare instabile.

---

## 1.4.4 Spike testing

### Definizione

Lo **spike testing** valuta il comportamento del sistema sotto aumenti improvvisi di carico.

A differenza del load testing o dello stress testing graduale, lo spike testing si concentra sulle transizioni rapide piuttosto che su condizioni operative stabili.

---

### Obiettivo

- osservare la reazione a cambiamenti bruschi del workload
- valutare elasticità e recupero
- rilevare instabilità transitoria

Lo spike testing è particolarmente rilevante per sistemi esposti a traffico bursty, picchi da campagne, domanda guidata da eventi o brevi impennate di attività.

---

### Caratteristiche

- il workload aumenta rapidamente ed in pochissimo tempo
- il sistema deve adattarsi velocemente

La caratteristica distintiva non è soltanto il volume di carico, ma la velocità con cui il carico cambia.

Un sistema può gestire un carico elevato quando esso viene raggiunto gradualmente, ma comportarsi male quando lo stesso carico arriva improvvisamente.

---

### Effetti osservabili

- picchi temporanei di latenza
- accumulo di code
- potenziali errori durante la transizione

(→ 1.5 Comportamento del sistema sotto carico)

Effetti aggiuntivi possono includere:

- risposta ritardata dello scaling
- esaurimento transitorio delle connessioni
- cascade temporanee di timeout
- recupero lento dopo il burst

---

### Valore diagnostico

Lo spike testing rivela:

- sensibilità al traffico bursty
- comportamento di accodamento sotto carico improvviso
- capacità di recupero dopo lo spike

Questo tipo di testing è prezioso perché molti sistemi sono ottimizzati per condizioni di regime stazionario ma restano fragili durante transizioni brusche.

---

### Comportamento di recupero

La parte più importante dello spike testing è spesso ciò che accade dopo lo spike.

Domande rilevanti includono:

- Il sistema ritorna rapidamente alla latenza normale?
- Le code si svuotano in modo controllato?
- Le risorse vengono rilasciate correttamente?
- Il sistema resta degradato dopo che lo spike è passato?

Un sistema che sopravvive allo spike ma recupera lentamente può comunque essere operativamente debole.

---

### Interpretazione pratica

Lo spike testing è particolarmente utile per sistemi che sono:

- esposti esternamente a traffico bursty
- dipendenti da auto-scaling o comportamento elastico
- sensibili all'accumulo di code
- soggetti a cambiamenti di domanda guidati da eventi

In questi casi, il carico medio è spesso meno importante dei picchi di breve periodo e della reazione del sistema ad essi.

---

### Idea chiave

Lo spike testing risponde a: *"Come reagisce il sistema a cambiamenti improvvisi di carico?"*

Esso valuta non soltanto la resistenza ai burst, ma anche la capacità di recuperare in modo pulito dopo di essi.

---

## 1.4.5 Soak testing

### Definizione

Il **soak testing** valuta il comportamento del sistema su un periodo esteso sotto carico sostenuto.

Talvolta viene chiamato anche endurance testing.

Il suo scopo è far emergere problemi che non compaiono in test di breve durata.

---

### Obiettivo

- rilevare problemi di lungo periodo
- osservare la stabilità nel tempo
- identificare degradazione graduale

Il soak testing riguarda meno la prestazione di picco e più la coerenza, l'accumulo e la deriva.

---

### Caratteristiche

- il workload è costante o varia lentamente
- la durata del test è lunga (ore o giorni)

La dimensione chiave è il tempo.

Alcuni sistemi si comportano correttamente per minuti ma degradano dopo ore a causa di effetti di accumulo.

---

### Effetti osservabili

- crescita della memoria
- leak di risorse
- degradazione delle prestazioni nel tempo

(→ 1.7 Runtime e modello di memoria)

Sintomi aggiuntivi di lunga durata possono includere:

- accumulo di thread
- leakage di connessioni
- code in lento aumento
- crescita dell'overhead del GC
- squilibrio della cache o retention incontrollata

---

### Valore diagnostico

Il soak testing rivela:

- slow memory leak
- esaurimento delle risorse
- instabilità di lungo periodo

È spesso l'unico modo affidabile per validare se il sistema rimanga sano ed operabile durante attività prolungata.

Questo è essenziale per sistemi di produzione che devono funzionare in continuo.

---

### Degradazione dipendente dal tempo

Il soak testing è importante perché alcune rotture non sono basate su soglie, ma sul tempo.

Esempi includono:

- memoria trattenuta lentamente nel tempo
- pool non completamente rilasciati
- task in background che accumulano deriva
- pattern di retry che aumentano lentamente la pressione
- cache che crescono senza eviction efficace

Questi problemi possono non comparire in load test o stress test di breve durata.

---

### Valore operativo

Un sistema che si comporta bene per dieci minuti ma degrada dopo sei ore non è stabile.

Il soak testing contribuisce quindi direttamente a:

- validazione per la messa in produzione
- fiducia nel runtime
- valutazione dell'affidabilità di lungo periodo
- dimensionamento dell'infrastruttura e del runtime

Esso aiuta anche a validare che il monitoraggio rimanga significativo su lunghi periodi di operatività.

---

### Interpretazione pratica

Il soak testing è particolarmente importante per sistemi con:

- lunghi uptime
- elaborazione in background
- runtime con gestione della memoria
- architetture ricche di connessioni
- pool di risorse che cambiano lentamente nel tempo

In tali sistemi, i risultati prestazionali di breve durata non sono sufficienti a garantire la stabilità reale.

---

### Idea chiave

Il soak testing risponde a: *"Il sistema rimane stabile nel tempo?"*

Esso valida il comportamento di lunga durata e rivela problemi causati da accumulo, deriva e degradazione lenta.

---

## 1.4.6 Capacity testing

### Definizione

Il **capacity testing** determina il workload massimo che un sistema può gestire soddisfacendo i requisiti prestazionali.

Viene utilizzato per identificare il limite operativo pratico del sistema in condizioni accettabili.

---

### Obiettivo

- identificare il throughput massimo sostenibile
- determinare limiti operativi sicuri
- supportare il capacity planning

Il capacity testing è quindi direttamente collegato a pianificazione, dimensionamento, forecasting e decisioni operative.

---

### Metodo

- eventuali test unitari per baseline dimensionale
- aumentare gradualmente il workload
- monitorare latenza, throughput ed errori
- identificare il punto in cui le prestazioni degradano

L'aumento del carico dovrebbe essere controllato e misurabile.

Questo permette di localizzare il limite del sistema con maggiore precisione rispetto a uno stress test puramente esplorativo.

---

### Interpretazione

Il limite di capacità viene raggiunto quando:

- la latenza supera soglie accettabili
- il tasso di errore aumenta
- il throughput non scala più

(→ 1.2 Metriche e formule di base)
(→ 1.5 Comportamento del sistema sotto carico)

In pratica, il limite non è sempre un singolo valore esatto.

Può essere meglio compreso come un intervallo in cui il comportamento accettabile inizia a deteriorarsi.

---

### Che cosa rivela il capacity testing

Il capacity testing rivela:

- il carico sostenibile più elevato sotto criteri di accettazione definiti
- il margine tra carico atteso e carico massimo accettabile
- la relazione tra domanda crescente e comportamento degradato
- il punto in cui ulteriore carico non produce più throughput utile

Queste informazioni sono essenziali per decisioni ingegneristiche e di pianificazione.

---

### Relazione con il capacity planning

Il capacity testing è uno dei principali input del capacity planning.

Esso aiuta a rispondere a domande quali:

- Quanto traffico può supportare l'attuale sistema?
- Quanto headroom è disponibile?
- Quando sarà necessario scalare?
- Quale componente vincola per primo la capacità?

Questo rende il capacity testing particolarmente utile per forecasting e preparazione operativa.

---

### Distinzione dallo stress testing

Il capacity testing non consiste nel forzare il fallimento per il fallimento stesso.

Consiste nell'identificare il carico più elevato che soddisfa ancora requisiti definiti.

- il **capacity testing** si ferma al limite accettabile o vicino a esso
- lo **stress testing** continua oltre tale limite per esaminare il comportamento in sovraccarico

La distinzione conta perché molte decisioni di business e ingegneristiche dipendono da un'operatività sicura, non dal fallimento totale.

---

### Significato pratico

La capacità non è soltanto un numero.

Essa dipende da:

- mix del workload
- livello di concorrenza
- obiettivi di latenza
- tasso di errore accettabile
- vincoli sulle risorse

Per questa ragione, ogni valore di capacità deve sempre essere interpretato nel contesto del workload e dei criteri di accettazione utilizzati durante il test.

---

### Interpretazione pratica

Il capacity testing è più utile quando l'obiettivo ingegneristico è rispondere a:

- "Qual è l'intervallo operativo sicuro?"
- "Quanto headroom abbiamo?"
- "Quando dobbiamo scalare?"
- "Che cosa vincola la crescita futura?"

Esso è quindi una delle forme di performance testing più orientate alle decisioni.

---

### Idea chiave

Il capacity testing risponde a: *"Fino a che punto il sistema può scalare prima di degradare?"*

Esso identifica il massimo intervallo operativo sostenibile, non soltanto il punto di fallimento.

---

◀ 1.3 – Lavoro di un performance engineer | ▲ Index | 1.5 – Comportamento del sistema sotto carico ▶

# 1.5 – Comportamento del sistema sotto carico

Questo capitolo analizza il comportamento dei sistemi all'aumentare del carico di lavoro (workload) e in prossimità dei loro limiti di capacità.

Esso si concentra sui principali meccanismi che possono causare degradazione sotto carico, inclusi **saturazione**, **accodamento**, **perdita di throughput** e **amplificazione della tail latency**.

Questi concetti sono centrali nella performance engineering poiché analizzano il perché i sistemi possano apparire stabili a basso carico e diventino instabili in prossimità dei loro limiti capacitivi.

## Indice



---

## 1.5.1 Carico vs capacità

### Definizione

Un sistema opera sotto un carico di lavoro, ma possiede una capacità ben definita.

- **Carico**: la quantità di lavoro applicata al sistema (es. richieste al secondo, utenti concorrenti)
- **Capacità**: la quantità massima di lavoro che il sistema può gestire rimanendo stabile

Comprendere la relazione tra carico e capacità è fondamentale nella performance engineering.

Essa definisce l'inviluppo operativo del sistema e determina quando il comportamento sia prevedibile e quando inizi la degradazione.

---

### Comportamento del sistema

A basso carico:

- le risorse sono sottoutilizzate
- il tempo di risposta è stabile
- il throughput aumenta linearmente con il carico

All'aumentare del carico:

- l'utilizzazione delle risorse cresce
- la contesa inizia a comparire
- il tempo di risposta aumenta

Quando il carico si avvicina alla capacità:

- si formano code
- la latenza aumenta rapidamente
- il comportamento del sistema diventa meno prevedibile

Questa transizione è uno degli aspetti più importanti dell'analisi delle prestazioni.

Un sistema raramente passa direttamente da "stabile" a "problematico".

Di solito attraversa una regione di crescente instabilità e ridotta efficienza.

### La capacità non è un valore fisso

La capacità è spesso fraintesa come un insieme ristretto di valori.

In realtà, essa dipende da:

- composizione del workload (casi d'uso e distribuzione)
- configurazione delle risorse (CPU, memoria, pool)
- stato del sistema (cold vs warm, effetti della cache)
- dipendenze esterne (database, servizi)

Un sistema può gestire:

- 100 req/s per richieste semplici
- ma solo 20 req/s per richieste complesse

La capacità è quindi sempre contestuale.

Deve essere compresa in relazione a uno specifico workload, ambiente e criteri di accettazione.

---

### Capacità effettiva

La capacità deve essere definita sotto vincoli ben precisi.

Criteri tipici:

- latenza entro limiti accettabili (es. p95)
- tasso di errore sotto soglia
- utilizzo stabile delle risorse

Il carico massimo che soddisfa queste condizioni è la **capacità effettiva**.

Questa è la capacità che conta operativamente.

Un massimo teorico che produce latenza inaccettabile o instabilità non è utile nella pratica.

---

### Implicazione pratica

La capacità non può essere assunta a priori.

Deve essere:

- misurata sotto workload realistico
- validata tramite testing
- monitorata nel tempo

Aumentare il carico oltre la capacità effettiva conduce a:

- rapida degradazione
- comportamento instabile
- potenziale rottura del sistema

Può anche ridurre la capacità del sistema di recuperare rapidamente dopo un sovraccarico.

---

### Collegamento con i concetti precedenti

La relazione tra carico, latenza e concorrenza è formalizzata da:

→ 1.2.1 Legge di Little

All'aumentare del carico:

- la concorrenza aumenta

- il tempo di attesa cresce
- il tempo di risposta degrada

Questa relazione costituisce uno dei fondamenti per comprendere il comportamento sotto carico.

---

### Interpretazione pratica

Carico e capacità non dovrebbero mai essere trattati come etichette astratte.

Essi determinano:

- se il sistema opera con headroom
- se è probabile che compaia accodamento (queueing)
- quanto margine esista prima che appaia instabilità

Nella performance engineering, sapere che un sistema “funziona” non è sufficiente.

Ciò che conta è sapere sotto quali condizioni di carico esso rimanga stabile e quanto sia vicino alla sua capacità effettiva.

---

### Idea chiave

Un sistema non si rompe quando raggiunge la capacità.

Inizia a degradare prima di quel punto.

L’obiettivo della performance engineering è identificare:

- dove si trovino i limiti di capacità
- come il sistema si comporti in prossimità di essi
- quanto margine sia richiesto

---

## 1.5.2 Saturazione e accodamento

### Definizione

La **saturazione** si verifica quando una risorsa è occupata per la maggior parte o per tutto il tempo.

L’**accodamento** (queueing) si verifica quando il lavoro in ingresso non può essere elaborato immediatamente e deve essere messo in attesa: in coda.

Questi due fenomeni sono strettamente correlati.

Essi sono tra i più importanti meccanismi alla base della degradazione delle prestazioni nei sistemi reali.

---

### Saturazione della risorsa

Una risorsa diventa satura quando:

- la sua utilizzazione si avvicina al limite
- ha poco o nessun tempo di inattività

Esempi tipici:

- CPU vicina al 100%
- thread pool completamente occupato
- connection pool esaurito

A questo punto:

- le nuove richieste non possono essere elaborate immediatamente
- devono attendere

La saturazione non significa necessariamente problema.

Significa che il sistema ha perso margine di elaborazione e non è più in grado di assorbire ulteriore lavoro senza ritardo.

---

### Formazione della coda

Quando le richieste di lavoro arrivano più velocemente di quanto possano essere elaborate:

- si forma una coda
- il tempo di attesa aumenta

Questo influisce sul tempo di risposta:

- il tempo di servizio rimane lo stesso
- il tempo di attesa cresce

→ 1.2.3 Tempo di servizio vs tempo di risposta

L'accodamento è quindi la conseguenza visibile di una capacità di elaborazione insufficiente su una determinata risorsa.

---

### Effetto non lineare

L'accodamento non cresce linearmente.

All'aumentare dell'utilizzazione:

- il tempo di attesa cresce lentamente all'inizio
- poi aumenta rapidamente
- infine domina il tempo di risposta

Piccoli aumenti di carico possono causare grandi aumenti di latenza.

Questo spiega perché i sistemi spesso appaiano stabili per lungo tempo e poi degradino improvvisamente vicino alla soglia di saturazione.

---

### Collegamento con l'utilizzazione

L'utilizzazione svolge un ruolo centrale:

→ 1.2.2 Legge di Utilizzazione

Quando l'utilizzazione si avvicina al proprio limite:

- la probabilità di attesa aumenta
- le code crescono
- la latenza diventa instabile

Il punto importante non è che una risorsa sia "occupata", ma che quando essa sia persistentemente occupata, il lavoro in ingresso inizi ad accumularsi.

---

### Implicazioni pratiche

L'accodamento è spesso la causa principale della degradazione delle prestazioni.

I sintomi includono:

- aumento improvviso del tempo di risposta
- tail latency elevata (p95, p99)
- code crescenti (thread, connessioni, richieste)

Anche se:

- la CPU non è completamente saturata
- la latenza media sembra accettabile

l'accodamento può comunque essere la fonte dominante del ritardo.

Questo è particolarmente comune nei sistemi con pool condivisi, operazioni bloccanti o colli di bottiglia sulle dipendenze.

---

### Esempio

Un sistema gestisce richieste con:

- tempo di servizio = 10 ms

A basso carico:

- le richieste vengono elaborate immediatamente
- tempo di risposta ≈ 10 ms

All'aumentare del carico:

- le richieste iniziano ad attendere
- il tempo di risposta diventa:
  10 ms (servizio) + tempo di attesa

Ad alto carico:

- il tempo di attesa domina
- il tempo di risposta aumenta rapidamente

Questo esempio vuole illustrare il perché la crescita della latenza sotto carico sia spesso causata più dall'attesa che dal lavoro in sé stesso.

---

### Interpretazione pratica

La saturazione è la condizione.

L'accodamento (queueing) è la conseguenza.

Il sistema non rallenta perché ogni richiesta richiede più computazione, ma perché più richieste stanno competendo per le stesse risorse limitate.

Questa distinzione è essenziale:

- ottimizzare il tempo di servizio può aiutare
- ma ridurre l'accodamento è spesso ancora più importante

---

### Idea chiave

La saturazione non rompe immediatamente il sistema.

Introduce accodamento.

L'accodamento aumenta il tempo di attesa.

Il tempo di attesa domina il tempo di risposta.

Questo è il meccanismo principale alla base della degradazione delle prestazioni sotto carico.

---

## 1.5.3 Degradazione non lineare

### Definizione

Le prestazioni del sistema non degradano linearmente all'aumentare del carico.

Piuttosto, la degradazione segue un andamento non lineare, specialmente in prossimità dei limiti di capacità.

Ciò significa che la relazione tra carico e tempo di risposta è spesso inizialmente regolare e poi fortemente instabile vicino alla saturazione.

---

### Comportamento lineare vs non lineare

A carico basso o moderato:

- il throughput aumenta proporzionalmente al carico
- la latenza rimane relativamente stabile

In questa regione, il sistema appare prevedibile.

---

Quando il carico si avvicina alla capacità:

- piccoli aumenti di carico producono grandi aumenti di latenza
- la variabilità aumenta
- il comportamento diventa instabile

Questo segna la transizione verso la degradazione non lineare.

Il sistema non si comporta più in modo proporzionale alla domanda.

Inizia a reagire in modo sproporzionato al lavoro aggiuntivo.

---

### Causa radice

La degradazione non lineare è causata principalmente da:

- effetti di accodamento (→ 1.5.2 Saturazione e accodamento)
- elevata utilizzazione delle risorse
- contesa tra richieste

All'aumentare dell'utilizzazione:

- il tempo di attesa cresce in modo sproporzionato
- il tempo di risposta viene dominato dai ritardi piuttosto che dal servizio

Questo spiega perché la degradazione spesso acceleri improvvisamente invece di crescere gradualmente.

---

### Effetti osservabili

I sintomi tipici includono:

- rapido aumento della latenza p95 e p99
- ampliamento del divario tra latenza media e tail latency
- aumento della varianza nei tempi di risposta
- errori intermittenti o timeout

Questi effetti spesso compaiono improvvisamente.

Il sistema può sembrare sano subito prima di entrare in una regione di grave instabilità.

---

### Intuizione fuorviante

È comune assumere:

- “Se il sistema gestisce 80 req/s, dovrebbe gestire 100 req/s con latenza leggermente più alta”

In realtà:

- le prestazioni possono rimanere stabili fino a un certo punto
- poi degradare bruscamente oltre quel punto

Spesso non esiste una transizione graduale.

Questo costituisce uno degli errori più comuni nel capacity planning e nelle aspettative prestazionali.

---

### Esempio

Un sistema si comporta come segue:

- fino a 70 req/s → latenza stabile (~100 ms)
- a 80 req/s → la latenza aumenta a 150 ms
- a 90 req/s → la latenza salta a 400 ms
- a 100 req/s → il sistema diventa instabile

La degradazione non è proporzionale al carico.

Gli ultimi incrementi di carico hanno un effetto molto maggiore rispetto a quelli precedenti.

---

### Implicazione pratica

Il capacity planning deve tenere conto del comportamento non lineare.

Operare un sistema vicino ai suoi limiti conduce a:

- latenza imprevedibile
- prestazioni instabili
- esperienza utente scadente

I sistemi dovrebbero operare con un ragionevole margine di sicurezza al di sotto della capacità.

Quel margine non è opzionale.

È ciò che permette al sistema di assorbire la normale variabilità senza entrare in un comportamento instabile.

---

### Collegamento con i concetti precedenti

La degradazione non lineare è l’effetto visibile di:

- utilizzazione crescente (→ 1.2.2 Legge di Utilizzazione)
- accodamento crescente (→ 1.5.2 Saturazione e accodamento)

È quindi una conseguenza a livello di sistema di meccanismi già introdotti nelle sezioni precedenti.

---

### Interpretazione pratica

La degradazione non lineare spiega perché i sistemi non dovrebbero essere gestiti troppo vicino al loro massimo teorico.

Un adeguato margine operativo può fare la differenza tra:

- prestazioni stabili
- degradazione imprevedibile

Questo spiega anche perché il solo utilizzo medio delle risorse sia spesso fuorviante nella valutazione della sicurezza in produzione.

---

#### Idea chiave

La degradazione delle prestazioni non è graduale.

Accelera man mano che il sistema si avvicina ai propri limiti.

Comprendere questa non linearità è essenziale per evitare di gestire sistemi troppo vicino ai loro limiti di capacità.

---

## 1.5.4 Collasso del throughput

#### Definizione

Il **collasso del throughput** si verifica quando l'aumento del carico non aumenta più il throughput e può perfino ridurlo.

Invece di scalare con la domanda, il sistema diventa meno efficiente man mano che il carico aumenta.

Questo è uno dei segnali più chiari che il sistema stia operando oltre la propria capacità effettiva.

---

#### Comportamento atteso vs collasso

In condizioni normali:

- l'aumento del carico aumenta il throughput
- fino a quando il sistema si avvicina ai limiti di capacità

Tuttavia, oltre un certo punto:

- il throughput smette di aumentare
- può stabilizzarsi o diminuire
- la latenza aumenta significativamente

Questo è il cosiddetto collasso del throughput.

Più lavoro in ingresso non si traduce in altrettanto lavoro completato.

---

#### Cause radice

Il collasso del throughput è tipicamente causato da:

- accodamento eccessivo
- contesa su risorse condivise
- thrashing delle risorse (CPU, memoria, I/O)
- amplificazione dei retry
- scheduling o locking inefficienti

Quando il sistema va in sovraccarico:

- si spende più tempo nel gestire la contesa che nel fare lavoro utile
- la capacità di elaborazione effettiva diminuisce

Questa è la ragione chiave per cui maggiore domanda può produrre minore output.

---

#### Contributo dell'accodamento

Quando le code crescono:

- le richieste attendono più a lungo
- le risorse del sistema restano occupate
- le nuove richieste aggiungono pressione senza aumentare il lavoro completato

L'accodamento può quindi:

- aumentare la latenza
- ridurre il throughput effettivo

Questo è particolarmente visibile quando il sistema trascorre sempre più tempo a gestire l'arretrato invece di fare reale progresso.

---

### Contesa e thrashing

Ad alto carico:

- i thread competono per risorse condivise
- i lock diventano hotspot
- il context switching aumenta
- la località della cache degrada

In casi estremi:

- il sistema trascorre più tempo a coordinare che a elaborare

Questo conduce a una riduzione del throughput.

Il sistema rimane attivo, ma la sua attività diventa sempre più improduttiva.

---

### Amplificazione dei retry

I fallimenti sotto carico spesso innescano retry.

Questo crea carico aggiuntivo:

- le richieste fallite vengono ritentate
- viene generato più lavoro
- la pressione aumenta ulteriormente

Questo loop di feedback può:

- accelerare il collasso
- rendere difficile il recupero

Il comportamento dei retry non è quindi soltanto una risposta ai sintomi, ma anche una frequente causa del peggioramento del sovraccarico.

---

### Effetti osservabili

I sintomi tipici includono:

- throughput che si stabilizza o diminuisce nonostante l'aumento del carico
- forte aumento della latenza
- aumento dei tassi di errore (timeout, 5xx)
- comportamento instabile o oscillante

A questo stadio, il sistema può sembrare occupato ma non sta più scalando in modo utile.

---

### Esempio

Un sistema si comporta come segue:

- 50 req/s → 50 req/s di throughput
- 80 req/s → 80 req/s di throughput
- 100 req/s → 90 req/s di throughput
- 120 req/s → 70 req/s di throughput

L'aumento del carico riduce il throughput effettivo.

Questo è un indicatore diretto del fatto che il sovraccarico stia "danneggiando" il lavoro utile.

---

### Implicazione pratica

Il collasso del throughput indica che il sistema sta operando oltre la propria capacità effettiva.

A questo punto:

- aggiungere più carico peggiora le prestazioni
- il sistema può diventare instabile

La mitigazione richiede:

- ridurre il carico
- rimuovere i colli di bottiglia
- migliorare l'efficienza delle risorse

In molti casi, la prima azione correttiva non è l'ottimizzazione ma la protezione: rate limiting, admission control o controllo dei retry.

---

### Collegamento con i concetti precedenti

Il collasso del throughput è il risultato di:

- degradazione non lineare (→ 3.5.3 Degradazione non lineare)
- saturazione e accodamento (→ 3.5.2 Saturazione e accodamento)

Può quindi essere compreso come uno stadio avanzato del comportamento in sovraccarico.

---

### Interpretazione pratica

Un sistema non elabora sempre più lavoro quando gliene viene applicato di aggiuntivo.

A un certo punto, il lavoro aggiuntivo diventa distruttivo anziché produttivo.

Riconoscere questa transizione è essenziale nella performance engineering, perché segna la differenza tra carico elevato e sovraccarico.

---

### Idea chiave

Oltre un certo punto, il carico aggiuntivo riduce la capacità del sistema di elaborare richieste.

Comprendere il collasso del throughput è essenziale per evitare condizioni di sovraccarico.

---

## 1.5.5 Amplificazione della tail latency

### Definizione

L'**amplificazione della tail latency** si riferisce all'aumento sproporzionato dei tempi di risposta ad alto percentile (es. p95, p99) sotto carico.

Mentre la latenza media può apparire accettabile, un sottoinsieme di richieste diventa significativamente più lento.

Questo effetto costituisce uno dei più importanti indicatori di un'esperienza utente degradata e d'instabilità nascosta.

---

### Percentili vs media

La latenza media nasconde la variabilità.

I percentili rivelano la distribuzione:

- p50 rappresenta la richiesta tipica
- p95 e p99 rappresentano le richieste più lente

Sotto carico:

- la latenza media può aumentare moderatamente
- la tail latency può aumentare drasticamente

→ 1.2.7 Percentili

Per questa ragione, le sole medie non sono sufficienti per valutare la reale qualità prestazionale.

---

### Cause radice

L'amplificazione della tail latency è guidata principalmente da:

- ritardi di accodamento
- contesa su risorse condivise
- distribuzione disomogenea del workload
- variabilità delle dipendenze (es. database, servizi esterni)

Anche piccoli ritardi in alcuni componenti possono:

- propagarsi attraverso il sistema
- amplificare la latenza end-to-end

La tail latency è quindi spesso un effetto emergente, non soltanto locale.

---

### Effetto nei sistemi distribuiti

Nei sistemi con più componenti:

- una richiesta dipende spesso da più servizi
- la latenza complessiva dipende dal componente più lento

All'aumentare del numero di dipendenze:

- la probabilità di una richiesta lenta aumenta
- la tail latency diventa più pronunciata

Questa è una delle ragioni per cui la tail latency sia particolarmente importante nelle architetture distribuite.

---

### Sotto carico

All'aumentare del carico:

- le code crescono
- la contesa aumenta
- la variabilità si espande

Questo conduce a:

- un ampliamento del divario tra media e p95/p99

- tempi di risposta imprevedibili per un sottoinsieme di utenti

Il sistema può quindi apparire per lo più stabile pur producendo comunque un'esperienza inaccettabile per una frazione significativa di richieste.

---

### Effetti osservabili

I sintomi tipici includono:

- latenza media stabile con p95/p99 degradati
- risposte lente intermittenti
- timeout che colpiscono solo una frazione di richieste

Questo può risultare fuorviante:

- il sistema appare "per lo più a posto"
- ma l'esperienza utente è degradata

Questo spiega perché le metriche di coda siano essenziali nel performance testing e nel monitoraggio in produzione.

---

### Esempio

Un sistema mostra:

- latenza media = 120 ms
- latenza p95 = 180 ms (accettabile)
- latenza p99 = 1200 ms (problematica)

La maggior parte delle richieste è veloce, ma una piccola percentuale è molto lenta.

In molti sistemi user-facing, questa piccola percentuale è sufficiente a creare insoddisfazione visibile o violazioni degli SLO.

---

### Implicazione pratica

La valutazione delle prestazioni deve considerare la **tail latency**.

Affidarsi alle medie può:

- nascondere problemi critici
- sottostimare l'impatto sugli utenti

I sistemi dovrebbero essere progettati e testati per:

- controllare il comportamento di coda
- limitare la variabilità sotto carico

Questo è particolarmente importante per sistemi distribuiti, API e applicazioni interattive.

---

### Collegamento con i concetti precedenti

L'amplificazione della tail latency è una conseguenza di:

- accodamento (→ 1.5.2 Saturazione e accodamento)
- degradazione non lineare (→ 1.5.3 Degradazione non lineare)
- interazioni e dipendenze di sistema

Essa è quindi una delle manifestazioni più visibili dello stress del sistema sotto carico.

---

### Interpretazione pratica

Le prestazioni non sono definite dalla richiesta media.

Sono definite dalla prevedibilità dei tempi di risposta, specialmente per le richieste più lente.

Un sistema con latenza media accettabile ma comportamento p95/p99 scarso non è realmente stabile dal punto di vista dell'utente o operativo.

---

### Idea chiave

Le prestazioni non sono definite dalla richiesta media.

Sono definite da come il sistema si comporta per le richieste più lente.

Controllare la tail latency è essenziale per sistemi prevedibili e affidabili.

---

◀ 01-04-types-of-performance-tests | ▲ Index | 1.6 – Concorrenza e parallelismo ▶

# 1.6 – Concorrenza e parallelismo

Questo capitolo introduce concorrenza e parallelismo come concetti fondamentali nella performance engineering dei sistemi e delle applicazioni.

Esso introduce lo scheduling del lavoro, come interagiscano task multipli e perché overhead di coordinamento, contesa e sincronizzazione diventino spesso fattori limitanti sotto carico.

Concorrenza e parallelismo sono essenziali per la scalabilità, ma introducono anche complessità, overhead e punti di rottura che influenzano direttamente latenza, throughput e stabilità del sistema.

## Indice



---

## 1.6.1 Concorrenza vs parallelismo

### Definizione

**Concorrenza** e **parallelismo** sono concetti correlati ma distinti.

Essi sono spesso confusi, ma descrivono aspetti differenti del comportamento del sistema.

Comprendere la distinzione è essenziale perché un sistema può gestire molte attività contemporaneamente da un punto di vista strutturale senza eseguire realmente molte attività simultaneamente a livello hardware.

---

### Concorrenza

La **concorrenza** si riferisce alla capacità di un sistema di gestire più task durante uno stesso intervallo di tempo.

Questi task:

- possono non essere eseguiti esattamente nello stesso momento
- possono essere “interleaved”
- condividono risorse di sistema

La concorrenza riguarda:

- struttura
- coordinamento
- gestione di più operazioni “in flight”

Essa è quindi principalmente interessata a come il lavoro viene organizzato e schedulato.

---

### Parallelismo

Il **parallelismo** si riferisce all'esecuzione di più task nello stesso istante.

Questo richiede:

- più unità di elaborazione (es. core CPU)
- vera esecuzione simultanea

Il parallelismo riguarda:

- esecuzione
- utilizzo dell'hardware
- svolgere più lavoro nello stesso istante

Esso è quindi principalmente interessato all'esecuzione simultanea.

---

### Differenza chiave

- **Concorrenza** = gestire molti task
- **Parallelismo** = eseguire molti task simultaneamente

Un sistema può essere:

- concorrente ma non parallelo (single core, task "interleaved")
- parallelo ma non altamente concorrente (pochi task di lunga durata)

Questa distinzione conta perché le proprietà di scalabilità di un sistema dipendono non solo da quanto lavoro esista, ma anche da come tale lavoro venga coordinato e schedulato.

---

### Relazione con le prestazioni

La concorrenza influisce su:

- quante richieste possono essere in esecuzione
- come vengono condivise le risorse
- come sorge la contesa

Il parallelismo influisce su:

- quanto velocemente il lavoro possa essere eseguito
- quanto efficacemente venga utilizzato l'hardware

Entrambi influenzano:

- throughput
- latenza
- scalabilità

Nella pratica, aggiungere concorrenza senza sufficiente parallelismo può aumentare attesa e contesa, mentre aggiungere parallelismo senza un buon controllo della concorrenza può sprecare risorse o esporre problemi di coordinamento.

---

### Intuizione pratica

Un sistema concorrente:

- può accettare molte richieste
- può comunque elaborarle sequenzialmente o con parallelismo limitato

Un sistema parallelo:

- può elaborare più richieste nello stesso momento
- ma può comunque soffrire di contesa o overhead di coordinamento

Per questa ragione, concorrenza e parallelismo non dovrebbero essere trattati come automaticamente benefici.

Il loro valore dipende da come interagiscono con workload, risorse condivise e vincoli di esecuzione.

---

### Collegamento con i concetti precedenti

La concorrenza aumenta:

- il numero di richieste in flight (→ 1.2.1 Legge di Little)

Questo conduce a:

- condivisione delle risorse
- potenziale accodamento (→ 1.5.2 Saturazione e accodamento)

Questa è una delle principali ragioni per cui la concorrenza diventa un tema centrale nella performance engineering e non soltanto una questione di programmazione.

---

### Interpretazione pratica

La concorrenza è spesso necessaria per supportare molte operazioni simultanee, specialmente nei sistemi di rete e guidati da I/O.

Tuttavia, la concorrenza aumenta anche la probabilità di:

- interazioni su stato condiviso
- accumulo di code
- contesa sui lock
- overhead di coordinamento

Il parallelismo può aumentare il throughput, ma solo se viene realmente eseguito lavoro utile anziché lavoro bloccato o serializzato.

---

### Idea chiave

La concorrenza determina quanti task siano attivi.

Il parallelismo determina quanti task vengano eseguiti nello stesso momento.

Le prestazioni dipendono da entrambi, e da come interagiscono con le risorse del sistema.

---

## 1.6.2 Thread e modello di esecuzione

### Definizione

Il **modello di esecuzione** definisce come il lavoro venga eseguito all'interno di un sistema.

Nella maggior parte dei sistemi, il lavoro viene svolto da **thread**, che vengono eseguiti all'interno di un **processo**.

Il modello di esecuzione determina come le richieste vengano mappate sulle unità di esecuzione, come venga gestita l'attesa e come vengano consumate le risorse di sistema sotto carico.

---

### Processi e thread

Un **processo** è un ambiente di esecuzione isolato:

- possiede il proprio spazio di memoria
- contiene risorse (file, socket, memoria)

Un **thread** è un'unità di esecuzione all'interno di un processo:

- più thread condividono la stessa memoria del processo
- i thread eseguono task in concorrenza

Nella maggior parte delle applicazioni:

- un processo ospita più thread
- i thread gestiscono le richieste in ingresso

Questo modello a memoria condivisa rende i thread efficienti per la comunicazione, ma introduce anche la complessità dello stato condiviso.

---

### Thread

Un thread:

- esegue istruzioni
- consuma tempo CPU
- può bloccarsi in attesa (es. I/O, lock)

Più thread permettono a un sistema di:

- gestire più richieste
- sovrapporre computazione e attesa
- aumentare la concorrenza

Tuttavia, i thread non sono gratuiti.

Ogni thread aggiuntivo introduce overhead di memoria, overhead di scheduling e complessità di coordinamento.

---

### Ciclo di vita del thread

Un thread attraversa tipicamente diversi stati:

- **running** (in esecuzione attiva)
- **runnable** (pronto a essere eseguito, in attesa di CPU)
- **waiting** / blocked (in attesa di una risorsa o di un evento)

Le prestazioni sono influenzate da come i thread si spostano tra questi stati.

Un sistema con molti thread in stato "runnable" o "blocked" può apparire attivo, ma espletare un progresso utile limitato.

Comprendere gli stati dei thread è quindi essenziale nella diagnosi dei problemi di concorrenza.

---

### Stack e memoria

Ogni thread possiede il proprio **stack**:

- memorizza chiamate di metodo e variabili locali
- cresce e si riduce durante l'esecuzione

Implicazioni:

- più thread → maggiore utilizzo di memoria (uno stack per thread)
- catene di chiamata profonde → maggiore utilizzo dello stack
- l'esaurimento dello stack può portare a rotture

Questo è particolarmente rilevante nei sistemi ad alta concorrenza.

Il numero di thread influisce quindi non solo sullo scheduling, ma anche sull'impronta di memoria e sulla stabilità.

## Modelli di esecuzione

Sistemi differenti utilizzano **modelli di esecuzione** differenti.

I modelli comuni includono:

---

### Un thread per richiesta

Ogni richiesta viene gestita da un thread dedicato.

Caratteristiche:

- modello semplice
- facile da comprendere
- le operazioni bloccanti sono dirette

Limiti:

- elevato utilizzo di memoria con molti thread
- scalabilità limitata sotto condizioni di alta concorrenza

Questo modello è concettualmente semplice, ma spesso si comporta male quando la concorrenza diventa molto elevata o quando il blocking è frequente.

---

### Thread pool

Un numero fisso di thread gestisce le richieste in ingresso.

Le richieste vengono accodate e assegnate ai thread disponibili.

Caratteristiche:

- concorrenza controllata
- overhead ridotto rispetto a thread non limitati

Limiti:

- accodamento quando tutti i thread sono occupati
- potenziale saturazione del pool

Questo modello è ampiamente utilizzato perché fornisce utilizzo controllato delle risorse, ma introduce una coda esplicita e quindi un limite di capacità visibile.

---

### Modello event-driven / asincrono

Il lavoro viene gestito usando operazioni **non bloccanti** e **event loop**.

Caratteristiche:

- pochi thread possono gestire molte richieste concorrenti
- efficiente per workload I/O-bound

Limiti:

- modello di programmazione più complesso
- richiede gestione accurata dei flussi asincroni

Questo modello riduce il numero di thread bloccati, ma sposta la complessità su coordinamento, callback, gestione dello stato e design non bloccante.

---

## Prospettiva Java (esempio)

In Java, un modello di esecuzione comune utilizza thread pool.

Per esempio:

```
ExecutorService executor = Executors.newFixedThreadPool(10);

executor.submit(() -> {
    // task logic
});
```

Le richieste vengono:

- inviate a una coda
- eseguite da un numero limitato di thread

Se tutti i thread sono occupati:

- i task attendono nella coda
- la latenza aumenta

Per una spiegazione dettagliata dei thread in Java, vedi:

→ https://ars-digitale.github.io/java-21-study-guide/en/module-07/threads/

Questo esempio è semplice, ma evidenzia un'idea chiave: risorse di esecuzione limitate introducono naturalmente accodamento quando la domanda supera la capacità di elaborazione immediata.

---

### Bloccante vs non bloccante

I thread possono:

- **bloccarsi** (attendere I/O, lock, risorse esterne)
- **rimanere attivi** (lavoro CPU-bound)

Il blocking riduce la concorrenza effettiva:

- i thread sono occupati ma non progrediscono
- meno thread sono disponibili per nuovo lavoro

Gli approcci non bloccanti mirano a:

- ridurre l'attesa inattiva
- migliorare l'utilizzo delle risorse

La distinzione è importante perché un alto numero di thread non significa necessariamente alto throughput.

Se i thread trascorrono la maggior parte del tempo in attesa, la concorrenza è presente, ma l'esecuzione produttiva è limitata.

---

### Implicazioni pratiche

Il modello di esecuzione determina:

- come venga gestita la concorrenza
- come vengano utilizzate le risorse
- come compaia l'accodamento

Effetti tipici includono:

- saturazione del thread pool → accodamento delle richieste
- operazioni bloccanti → throughput ridotto
- troppi thread → overhead di context switching

Il modello di esecuzione determina anche dove i colli di bottiglia diventino visibili: nelle code, nei pool, nei thread bloccati o negli event loop.

---

### Collegamento con i concetti precedenti

Il comportamento dei thread impatta direttamente:

- accodamento (→ 1.5.2 Saturazione e accodamento)
- latenza sotto carico
- capacità effettiva del sistema

Esso influenza anche la rapidità con cui un sistema passa da un comportamento stabile alla saturazione quando la concorrenza aumenta.

---

### Interpretazione pratica

Scegliere un modello di esecuzione non è solo una decisione di programmazione.

È una decisione prestazionale.

Il modello influisce su:

- consumo di memoria
- overhead di scheduling
- latenza in condizioni di attesa
- scalabilità sotto workload reale

Un design facile da implementare può non essere quello che si comporta meglio sotto carico sostenuto.

---

### Idea chiave

Il modello di esecuzione definisce come il lavoro venga schedulato ed elaborato.

I thread non sono gratuiti.

Il modo in cui vengono utilizzati determina:

- quanto lavoro possa essere gestito
- quanto efficientemente vengano utilizzate le risorse
- come il sistema si comporti sotto carico

---

## 1.6.3 Contesa e sincronizzazione

### Definizione

La **contesa** si verifica quando più thread competono per la stessa risorsa.

La **sincronizzazione** è il meccanismo usato per coordinare l'accesso alle risorse condivise.

Questi concetti sono centrali per comprendere la degradazione delle prestazioni nei sistemi concorrenti.

Essi collegano correttezza e prestazioni: gli stessi meccanismi che proteggono lo stato condiviso possono anche diventare la fonte di attesa e di ridotta scalabilità.

---

### Risorse condivise

Nei sistemi concorrenti, i thread condividono spesso risorse come:

- strutture di memoria (oggetti, cache)
- lock e monitor
- thread pool e code
- connessioni a database
- canali I/O

Quando l'accesso non è coordinato, può verificarsi **corruzione** dei dati.

Quando l'accesso è coordinato, può comparire **contesa**.

Questo rende la sincronizzazione necessaria, ma non gratuita.

---

### Sincronizzazione

La sincronizzazione garantisce che le risorse condivise siano accessibili in modo sicuro.

I meccanismi comuni includono:

- lock (mutex, monitor)
- sezioni sincronizzate
- semafori
- operazioni atomiche

La sincronizzazione garantisce correttezza, ma introduce overhead.

Tale overhead può derivare da:

- attesa
- serializzazione dell'esecuzione
- memory barrier aggiuntive
- costi di coordinamento tra thread

---

### Contesa

La **contesa** sorge quando più thread tentano di accedere simultaneamente alla stessa risorsa.

Quando si verifica contesa:

- i thread possono bloccarsi o attendere
- l'esecuzione viene ritardata
- il throughput si riduce

Più thread competono:

- maggiore è il tempo di attesa
- minore è il parallelismo effettivo

Un sistema altamente concorrente può quindi comportarsi come un sistema parzialmente serializzato se molto del suo lavoro dipende dalle stesse risorse condivise.

---

### Contesa sui lock

Una forma comune di contesa coinvolge i lock.

Quando un thread detiene un lock:

- gli altri thread devono attendere
- può formarsi una coda di thread in attesa

Gli effetti includono:

- aumento della latenza
- riduzione del throughput
- potenziali colli di bottiglia

La contesa sui lock è particolarmente problematica quando le sezioni critiche sono lunghe, frequentemente accedute o collocate su hot path di esecuzione.

---

### Contesa vs utilizzazione

Elevata contesa può verificarsi anche quando l'utilizzazione della CPU è moderata.

Per esempio:

- molti thread sono in attesa di un lock
- la CPU è parzialmente inattiva
- il sistema appare sottoutilizzato ma è in realtà vincolato

Questa è una fonte comune di diagnosi fuorvianti.

Essa spiega perché un utilizzo basso o moderato della CPU non significhi necessariamente che il sistema abbia capacità disponibile.

---

### Sincronizzazione fine-grained vs coarse-grained

La sincronizzazione può essere:

- **coarse-grained** (pochi lock, grandi sezioni critiche)
- **fine-grained** (molti lock, sezioni critiche più piccole)

Compromessi:

- **coarse-grained** → più semplice ma maggiore contesa
- **fine-grained** → più scalabile ma più complessa

La scelta tra i due modelli dipende dalle caratteristiche del workload, dai pattern di accesso e dal costo della complessità aggiuntiva di design.

---

### Prospettiva Java (esempio)

In Java, la sincronizzazione può essere implementata usando blocchi `synchronized`:

```
synchronized (lock) {
    // critical section
}
```

Oppure lock espliciti:

```
Lock lock = new ReentrantLock();

lock.lock();
try {
    // critical section
} finally {
    lock.unlock();
}
```

Se molti thread tentano di entrare nella stessa sezione critica:

- la contesa aumenta
- i thread si bloccano
- le prestazioni degradano

Questo esempio evidenzia come un meccanismo di correttezza possa diventare un vincolo di scalabilità sotto carico.

---

### Sintomi della contesa

Indicatori tipici includono:

- aumento del tempo di risposta sotto carico
- **basso utilizzo CPU con alta latenza**

- thread in stati blocked o waiting
- lunghe code su risorse condivise

Questi sintomi spesso compaiono prima della saturazione totale e possono essere scambiati per altri problemi di risorse se non analizzati con attenzione.

---

### Implicazioni pratiche

La contesa limita la scalabilità.

Anche con:

- CPU sufficiente
- memoria adeguata

Un sistema può non scalare se:

- i thread trascorrono tempo in attesa invece di trovarsi in esecuzione

Ridurre la contesa ha spesso un impatto maggiore dell'ottimizzazione delle singole operazioni.

Questo è particolarmente vero per sistemi in cui le prestazioni siano vincolate dall'accesso condiviso piuttosto che dalla computazione pura.

---

### Collegamento con i concetti precedenti

La contesa contribuisce a:

- accodamento (→ 1.5.2 Saturazione e accodamento)
- degradazione non lineare (→ 1.5.3 Degradazione non lineare)
- collasso del throughput (→ 1.5.4 Collasso del throughput)

La contesa è quindi sia un fenomeno locale di sincronizzazione sia un meccanismo prestazionale a livello di sistema.

---

### Interpretazione pratica

La concorrenza aumenta le opportunità di sovrapposizione utile, ma aumenta anche la competizione per le risorse condivise.

La sfida pratica non è semplicemente aggiungere più thread, ma garantire che la concorrenza aggiuntiva produca lavoro utile anziché attesa aggiuntiva.

---

### Idea chiave

La concorrenza introduce la necessità di sincronizzazione.

La sincronizzazione introduce contesa.

La contesa limita le prestazioni.

Comprendere e controllare la contesa è essenziale per sistemi scalabili.

---

## 1.6.4 Problemi comuni di concorrenza

La concorrenza introduce complessità.

Quando più thread interagiscono, assunzioni scorrette o scarso coordinamento possono condurre a specifiche classi di problemi.

Questi problemi compaiono spesso sotto carico e possono influenzare severamente prestazioni e correttezza.

Molti di essi sono difficili da riprodurre in test superficiali perché dipendono da timing, scheduling o pressione sulle risorse.

---

### 1.6.4.1 Race conditions

#### Definizione

Una **race condition** si verifica quando più thread accedono a dati condivisi senza adeguata sincronizzazione, e il risultato dipende dal timing.

L'esito non è quindi deterministico e può variare da un'esecuzione all'altra.

---

#### Esempio

Due thread aggiornano un contatore condiviso:

- Thread A legge valore = 10
- Thread B legge valore = 10
- Thread A scrive 11
- Thread B scrive 11

Risultato atteso: 12
Risultato reale: 11

Il valore finale dipende dall'ordine in cui operazioni non sincronizzate vengono eseguite.

---

#### Impatto

- risultati errati
- stato del sistema incoerente
- bug difficili da riprodurre

Le race condition possono anche corrompere assunzioni interne in modi che compaiono solo più tardi sotto carico.

---

#### Rilevanza prestazionale

Le race condition possono non causare sempre errori visibili, ma:

- richiedono spesso sincronizzazione aggiuntiva
- fix impropri possono introdurre contesa

Questa è una delle ragioni per cui correttezza e prestazioni non possano essere trattate come questioni completamente separate nei sistemi concorrenti.

---

### 1.6.4.2 Deadlock

#### Definizione

Un **deadlock** si verifica quando due o più thread si attendono indefinitamente l'un l'altro.

Ogni thread detiene una risorsa e attende un'altra risorsa detenuta dall'altro thread.

Di conseguenza, il progresso si arresta completamente.

---

#### Esempio

- Thread A detiene il lock L1 e attende L2
- Thread B detiene il lock L2 e attende L1

Nessuno dei due può procedere ulteriormente.

Questo pattern di attesa circolare è la caratteristica distintiva del deadlock.

---

#### Impatto

- il sistema si blocca
- le richieste non vengono mai completate
- le risorse rimangono bloccate

I deadlock sono particolarmente gravi perché trasformano risorse attive in risorse permanentemente bloccate.

---

#### Rilevazione

- i thread rimangono bloccati
- i thread dump mostrano attesa circolare

I deadlock sono spesso rilevati tramite analisi dei thread piuttosto che tramite metriche prestazionali generali.

---

### 1.6.4.3 Livelock

#### Definizione

Un **livelock** si verifica quando i thread non sono bloccati ma cambiano continuamente stato in risposta reciproca senza fare progresso.

A differenza del deadlock, l'attività continua, ma il lavoro utile no.

---

#### Esempio

Due thread ritentano ripetutamente un'operazione:

- entrambi rilevano un conflitto
- entrambi ritentano nello stesso momento
- il conflitto persiste

Il sistema rimane attivo, ma il comportamento conflittuale continua indefinitamente.

---

#### Impatto

- la CPU viene utilizzata
- nessun lavoro utile viene completato

I livelock possono quindi sembrare elaborazione attiva anche se il progresso effettivo è pari a zero.

---

### 1.6.4.4 Starvation

#### Definizione

La **starvation** si verifica quando alcuni thread non riescono a ottenere risorse per un periodo prolungato.

Altri thread continuano a eseguire mentre alcuni vengono di fatto ignorati.

Ciò significa che il sistema sta operando progresso, ma non in modo equo o prevedibile per tutto il lavoro.

---

### Cause

- scheduling non equo
- thread ad alta priorità che dominano l'esecuzione
- monopolizzazione delle risorse

La starvation è particolarmente problematica quando un sottoinsieme di richieste sperimenta latenza estrema mentre il resto del sistema appare funzionale.

---

### Impatto

- alcune richieste sperimentano latenza molto elevata
- il sistema appare parzialmente funzionale
- la tail latency aumenta

Questo rende la starvation particolarmente rilevante sia dal punto di vista prestazionale sia da quello dell'esperienza utente.

---

## 1.6.4.5 Esaurimento del thread pool

### Definizione

L'**esaurimento del thread pool** si verifica quando tutti i thread di un pool sono occupati e i task in ingresso devono attendere.

Questo è uno dei colli di bottiglia legati alla concorrenza più comuni nei sistemi reali.

---

### Cause

- operazioni bloccanti all'interno dei thread
- dimensione insufficiente del pool
- task di lunga durata

Queste cause possono esistere indipendentemente oppure rafforzarsi a vicenda sotto carico crescente.

---

### Effetti

- la coda delle richieste cresce
- la latenza aumenta
- il throughput può degradare

Se la saturazione continua, l'esaurimento del thread pool può anche contribuire a timeout, retry e instabilità nei componenti upstream.

---

### Collegamento con i concetti precedenti

L'esaurimento del thread pool è un esempio diretto di:

- saturazione (→ 1.5.2 Saturazione e accodamento)
- degradazione non lineare (→ 1.5.3 Degradazione non lineare)

Esso costituisce quindi una delle più chiare espressioni pratiche dei comportamenti di sistema introdotti nel capitolo precedente.

### Idea chiave

I problemi di concorrenza non sono soltanto problemi di correttezza.

Sono anche problemi prestazionali.

Molte degradazioni delle prestazioni sono causate da:

- contesa
- blocking
- fallimenti di coordinamento

Comprendere questi problemi è essenziale per diagnosticare sistemi reali.

---

◀ 1.5 – Comportamento del sistema sotto carico | ▲ Index | 01-07-runtime-and-memory-model ▶

## 1.7 – Runtime e modello di memoria

Questo capitolo spiega come i “managed runtime” organizzano la memoria, allocano gli oggetti, recuperano memoria non più utilizzata e si comportano in situazione di memoria “sotto pressione”.

Ci si concentra sui meccanismi di runtime e di memoria che influenzano direttamente latenza, stabilità e throughput sotto carico.

Comprendere questi meccanismi è essenziale perché molti problemi di performance non sono causati solo da limiti di CPU o I/O, ma dal modo in cui la memoria viene allocata, mantenuta e recuperata nel tempo.

## Indice



---

## 1.7.1 Struttura della memoria (heap, stack)

### Modelli di gestione della memoria

Sistemi diversi utilizzano strategie di gestione della memoria diverse.

Due approcci comuni sono:

- **gestione manuale della memoria**
  La memoria è allocata e liberata esplicitamente dal programmatore (es. C, C++)
- **memoria gestita**
  La memoria è allocata automaticamente e recuperata dal runtime (es. Java, .NET)

Questa guida si concentra sui **sistemi a memoria gestita**, dove:

- gli oggetti sono allocati dinamicamente
- la memoria è recuperata automaticamente da uno o piu’ thread dedicati delle rispettive macchine virtuali (garbage collection)

Questa distinzione è importante perché il comportamento delle performance cambia significativamente a seconda che il ciclo di vita della memoria sia controllato direttamente dal programmatore o indirettamente dal runtime.

---

### Definizione

La memoria è organizzata in diverse regioni con ruoli ben distinti.

Le due aree più importanti per il discorso sulle performance sono:

- **heap**
- **stack**

Queste due regioni supportano aspetti diversi dell'esecuzione del programma e hanno implicazioni di performance molto diverse.

---

### Heap

L'heap è un'area di memoria condivisa utilizzata per l'allocazione dinamica.

Nei runtime gestiti (come Java):

- gli oggetti sono allocati sull'heap
- la memoria è gestita dal runtime
- la garbage collection recupera gli oggetti non utilizzati

Implicazioni:

- l'utilizzo della memoria cresce con il tasso di allocazione
- la garbage collection impatta le performance
- l'accesso condiviso può introdurre contesa

L'heap quindi non è solo un'area di storage, ma una sezione centrale rispetto al comportamento del runtime sotto carico.

---

### Stack

Ogni thread ha il proprio stack.

Lo stack memorizza:

- chiamate di metodo (call frame)
- variabili locali
- valori intermedi

Caratteristiche:

- privato per ogni thread
- cresce e si riduce durante l'esecuzione
- tipicamente molto più piccolo dello heap

Poiché lo stack è privato del thread, l'accesso è semplice ed efficiente, ma il numero di thread influisce direttamente sull'utilizzo totale della memoria dello stack.

---

### Heap vs stack

| Aspetto | Heap | Stack |
|---|---|---|
| Scope | Condiviso tra thread | Privato per thread |
| Allocazione | Dinamica (oggetti) | Automatica (chiamate metodo) |
| Durata | Gestita dal runtime | Legata all'esecuzione metodo |
| Performance | Più complessa | Molto veloce |
| Impatto memoria | Globale | Per thread |

---

### Interazione con i thread

Ogni thread:

- ha il proprio stack
- condivide l'heap

Questo crea un modello in cui:

- l'esecuzione è isolata per thread (stack)
- i dati sono condivisi tra thread (heap)

Questa interazione è una fonte di:

- contesa (oggetti condivisi)
- overhead di coordinamento

Spiega anche perché concorrenza e comportamento al livello della memoria sono strettamente correlati nei sistemi gestiti dal runtime.

---

### Implicazioni sulle performance

Heap:

- allocazione eccessiva → aumento dell'attività GC
- heap grande → cicli di garbage collection più lunghi
- accesso condiviso → potenziale contesa

Stack:

- molti thread → maggiore utilizzo totale della memoria (uno stack per thread)
- catene di chiamata profonde → aumento dell'utilizzo dello stack
- stack overflow → fallimento in casi estremi

Queste implicazioni diventano particolarmente importanti quando il sistema è sotto carico sostenuto o ad alta concorrenza.

---

### Interpretazione pratica

Heap e stack non sono solo dettagli implementativi.

Influenzano:

- come i dati sono condivisi
- come il lavoro viene eseguito
- come la memoria cresce sotto concorrenza
- dove appare l'overhead del runtime

Un sistema con molti thread e allocazioni frequenti stressa entrambe le regioni in modo diverso: lo stack tramite il numero di thread e la profondità delle chiamate, l'heap tramite creazione e retention degli oggetti.

---

### Idea chiave

L'heap memorizza dati condivisi.

Lo stack supporta l'esecuzione.

Le performance dipendono da come questi due interagiscono sotto carico.

---

### Collegamento con concetti precedenti

Il comportamento della memoria impatta direttamente:

- l'esecuzione dei thread (→ 1.6.2 Threads and execution model)
- la contesa (→ 1.6.3 Contention and synchronization)
- la latenza sotto carico (→ 1.5 System behavior under load)

Per questo motivo il modello di runtime e memoria non può essere analizzato separatamente dalla concorrenza e dal comportamento del sistema.

---

## 1.7.2 Allocazione e ciclo di vita degli oggetti

### Definizione

Nei sistemi a memoria gestita, gli oggetti sono creati dinamicamente e vivono per un certo periodo di tempo prima di essere recuperati dal runtime.

Il modo in cui gli oggetti sono allocati e quanto a lungo vivono ha un impatto diretto sulle performance.

Il comportamento di allocazione quindi non è solo una questione di memoria, ma anche una questione di latenza e stabilità.

---

### Allocazione

L'allocazione è il processo di creazione di nuovi oggetti in memoria.

Nella maggior parte dei runtime gestiti:

- l'allocazione avviene sull'heap
- è progettata per essere veloce ed efficiente
- avviene molto frequentemente nelle applicazioni tipiche

Esempi di allocazione:

- creazione di oggetti request
- costruzione di strutture dati
- elaborazione di risultati intermedi

Nei sistemi ad alto throughput, l'allocazione è spesso continua e strettamente legata all'intensità del carico di lavoro.

---

### Tasso di allocazione

Il **tasso di allocazione** è la quantità di memoria allocata per unità di tempo.

È un fattore chiave di performance.

Un alto tasso di allocazione significa:

- più oggetti creati
- maggiore churn di memoria
- maggiore pressione sul runtime

Anche se le allocazioni individuali sono veloci, grandi volumi impattano il sistema.

Questo è uno dei motivi per cui "allocazione veloce" non significa automaticamente "basso overhead di memoria."

---

### Ciclo di vita degli oggetti

Gli oggetti non vivono tutti per la stessa durata.

Categorie tipiche includono:

- **oggetti a vita breve**
  creati e scartati rapidamente (es. dati temporanei di request)
- **oggetti a vita media**
  sopravvivono per un certo tempo durante l'elaborazione
- **oggetti a vita lunga**
  rimangono in memoria per periodi estesi (es. cache, stato condiviso)

Comprendere la durata di vita degli oggetti è essenziale per ragionare sul comportamento della memoria.

Questa caratteristica determina quanta memoria rimane attiva nel tempo e come il runtime deve organizzare il lavoro di recupero.

---

### Pattern di allocazione

I sistemi reali tendono a mostrare pattern come:

- molti oggetti a vita breve per request
- oggetti a vita lunga occasionali
- burst di allocazione sotto carico

Questi pattern determinano:

- utilizzo della memoria
- comportamento della garbage collection
- stabilità delle performance

I pattern di allocazione sono spesso più informativi degli eventi di allocazione isolati, perché il runtime reagisce al comportamento aggregato nel tempo.

---

### Impatto sulle performance

L'allocazione in sé è solitamente veloce.

L'impatto principale deriva da:

- aumento dell'utilizzo della memoria
- pressione sulla garbage collection

Un alto tasso di allocazione può portare a:

- cicli di garbage collection più frequenti
- aumento della latenza
- pause imprevedibili

Il punto importante è che il costo della memoria è spesso indiretto: il sistema paga non solo per creare oggetti, ma per gestire le conseguenze della creazione di molti oggetti.

---

### Sotto carico

Con l'aumentare del carico:

- più richieste vengono elaborate
- più oggetti vengono creati
- il tasso di allocazione aumenta

Questo amplifica:

- la pressione di memoria
- l'attività di garbage collection
- la variabilità della latenza

Un sistema stabile a basso carico può quindi diventare sensibile alla memoria con l'aumentare del volume di richieste, anche se la logica di ogni richiesta rimane invariata.

---

### Interazione con la concorrenza

L'allocazione è spesso eseguita da più thread.

Questo può portare a:

- contesa sulle strutture di memoria
- aumento dell'overhead di coordinamento
- pattern di utilizzo della memoria non uniformi

Nei sistemi ad alta concorrenza:

- il tasso di allocazione cresce con la concorrenza
- la memoria diventa un collo di bottiglia condiviso

Questo è uno dei modi in cui concorrenza e comportamento della memoria si rafforzano a vicenda sotto carico.

---

### Implicazioni pratiche

Per ragionare sulle performance è importante considerare:

- quanti oggetti sono creati per request
- quanto a lungo vivono
- come il tasso di allocazione cambia sotto carico

Comprendere l'allocazione è essenziale per:

- spiegare il comportamento della latenza
- identificare colli di bottiglia
- prevedere i limiti del sistema

Aiuta anche a distinguere tra problemi causati dal calcolo e problemi causati dal churn di memoria.

---

### Interpretazione pratica

L'allocazione è spesso invisibile a livello di codice perché è facile da scrivere e generalmente poco costosa per operazione.

Tuttavia, a livello di sistema, l'allocazione ripetuta cambia il carico di lavoro del runtime.

Un design che crea grandi quantità di oggetti temporanei può funzionare correttamente, ma comunque imporre una pressione significativa sul sottosistema della memoria.

---

### Collegamento con i concetti successivi

Allocazione e durata di vita degli oggetti influenzano direttamente:

- il comportamento della garbage collection (→ sezione successiva)
- la pressione di memoria
- la latenza sotto carico

Costituiscono quindi la base causale degli effetti di runtime descritti nel resto di questo capitolo.

---

### Idea chiave

Le performance dipendono da quanta memoria viene allocata e da quanto a lungo viene mantenuta.

I pattern di allocazione modellano il comportamento del sistema sotto carico.

---

## 1.7.3 Garbage collection (concettuale)

### Definizione

La garbage collection (GC) è il processo attraverso il quale un runtime gestito recupera memoria che non è più in uso.

Invece di richiedere una deallocazione esplicita, il runtime:

- identifica oggetti non utilizzati
- libera la loro memoria
- rende disponibile spazio per nuove allocazioni

La garbage collection è uno dei meccanismi distintivi dei runtime gestiti e uno dei principali modi in cui il comportamento della memoria diventa visibile nell'analisi delle performance.

---

### Principio di base

Un oggetto è eleggibile per la "collezione" quando non è più raggiungibile (puntato) da altri elementi del programma.

Questo significa:

- nessun riferimento attivo punta ad esso
- non può essere acceduto dal programma

Il runtime periodicamente:

- scansiona i riferimenti agli oggetti
- identifica oggetti non raggiungibili
- recupera la loro memoria

Questo modello permette una gestione automatica della memoria, ma implica anche che il lavoro di recupero debba essere eseguito durante l'esecuzione del programma.

---

### Ciclo allocazione e recupero

L'utilizzo della memoria segue un ciclo:

1. gli oggetti sono allocati
2. gli oggetti diventano inutilizzati
3. la garbage collection recupera la memoria

Questo ciclo si ripete continuamente durante l'esecuzione.

Il runtime alterna quindi allocazione di nuova memoria e recupero di memoria vecchia, con un comportamento complessivo guidato dal tasso di allocazione e dai pattern di retention.

---

### Prospettiva Java (esempio)

In Java, l'allocazione di oggetti è frequente ed economica.

Per esempio:

```
for (int i = 0; i < 1_000_000; i++) {
    String s = new String("test");
}
```

Questo codice crea un grande numero di oggetti a vita breve.

In un runtime gestito:

- questi oggetti sono allocati rapidamente sullo heap
- diventano non raggiungibili poco dopo la creazione
- la garbage collection li recupera

Se tali pattern di allocazione si verificano sotto carico:

- l'attività GC aumenta
- la pressione di memoria cresce
- la latenza può diventare instabile

L'impatto dipende non da una singola allocazione, ma dal **tasso di allocazione nel tempo**.

Per questo il comportamento della memoria deve essere analizzato come un pattern, non come un'operazione isolata.

### Esempio: retention degli oggetti

Gli oggetti che rimangono referenziati non vengono raccolti.

```
List<String> cache = new ArrayList<>();

while (true) {
    cache.add(new String("data"));
}
```

In questo caso:

- gli oggetti sono allocati continuamente
- non vengono mai rilasciati
- l'utilizzo della memoria cresce nel tempo

Questo porta a:

- aumento della pressione di memoria
- cicli di garbage collection più costosi
- potenziale instabilità del sistema

Questo esempio illustra la differenza tra churn temporaneo di allocazione e retention persistente.

### Costo della garbage collection

La garbage collection non è gratuita.

Introduce overhead:

- tempo CPU per analizzare la memoria
- pause durante la raccolta (a seconda della strategia/policy di GC)

Il costo dipende da:

- tasso di allocazione
- numero di oggetti attivi
- dimensione della memoria

In altre parole, il costo GC dipende non solo da quanta memoria esiste, ma da quanta memoria è attiva ed ancora raggiungibile.

---

### Effetto stop-the-world

Alcune fasi (di alcune policy) della garbage collection possono sospendere l'esecuzione dell'applicazione.

Durante queste pause:

- i thread applicativi sono temporaneamente in stand-by

- nessun lavoro applicativo viene eseguito

Anche pause brevi possono:

- aumentare la latenza
- influenzare i tempi di risposta in coda (p95, p99)

Questo è uno dei motivi per cui i problemi GC appaiono spesso prima nell'analisi della latenza basata su percentili piuttosto che nelle medie.

---

### Comportamento generazionale (concettuale)

La maggior parte dei runtime moderni utilizza un approccio generazionale.

Basato sull'osservazione:

- la maggior parte degli oggetti ha vita breve
- pochi oggetti hanno durata di vita prolungata

La memoria è organizzata in modo tale che:

- gli oggetti a vita breve siano raccolti frequentemente
- gli oggetti a vita lunga siano raccolti meno spesso

Questo migliora l'efficienza perché recuperare molti oggetti a vita breve è solitamente più economico che scansionare ripetutamente memoria a lunga retention.

---

### Sotto carico

Con l'aumentare del carico:

- il tasso di allocazione aumenta
- la garbage collection viene eseguita più frequentemente

Questo può portare a:

- maggiore utilizzo della CPU
- pause più frequenti
- aumento della variabilità della latenza

Sotto carico importante, la GC può quindi passare da meccanismo di manutenzione in background a parte visibile del comportamento delle performance del sistema.

---

### Interazione con il ciclo di vita degli oggetti

Il comportamento della garbage collection dipende da:

- quanti oggetti sono creati
- quanto a lungo essi vivono

Pattern tipici:

- molti oggetti a vita breve → raccolte frequenti
- molti oggetti a vita lunga → raccolte più pesanti

Per questo allocazione e retention devono essere analizzate insieme: il numero di oggetti da solo non è sufficiente.

---

### Effetti osservabili

I problemi di garbage collection appaiono spesso come:

- picchi di latenza

- latenza di coda (degrado p95/p99)
- pause periodiche
- aumento dell'utilizzo CPU senza causa evidente

Questi sintomi sono spesso intermittenti, il che rende i problemi legati alla GC difficili da diagnosticare senza correlare segnali di memoria e latenza.

---

### Implicazioni pratiche

L'analisi delle performance deve considerare:

- tasso di allocazione
- distribuzione della durata di vita degli oggetti
- frequenza e costo dei cicli GC

L'ottimizzazione tipicamente si concentra su:

- comprensione dei pattern di allocazione
- riduzione della creazione inutile di oggetti
- controllo della pressione di memoria

Il tuning del collector può aiutare, ma di solito è più efficace capire in anticipo perché il runtime è sotto pressione.

---

### Interpretazione pratica

La garbage collection non è un bug o un'anomalia.

È un meccanismo necessario del runtime.

La domanda sulle performance non è se la GC esiste, ma se il suo costo di funzionamento rimane compatibile con il carico di lavoro e gli obiettivi di latenza del sistema.

---

### Collegamento con concetti precedenti

La garbage collection è direttamente collegata a:

- allocazione (→ 1.7.2 Allocazione e ciclo di vita degli oggetti)
- struttura della memoria (→ 1.7.1 Struttura della memoria)
- latenza di coda (→ 1.5.5 Tail latency amplification)

È quindi sia un meccanismo di runtime sia un contributore a livello di sistema alla variabilità delle performance.

---

### Idea chiave

La garbage collection abilita la gestione automatica della memoria ma introduce variabilità.

Le performance dipendono da quanto efficientemente la memoria viene recuperata.

---

## 1.7.4 Pressione di memoria e performance

### Definizione

La pressione di memoria si riferisce allo stress posto sul sistema della memoria quando allocazione, retention e recupero interagiscono sotto carico.

Non riguarda solo quanta memoria viene utilizzata, ma come la memoria sia gestita e si comporta nel tempo.

La pressione di memoria è quindi una condizione dinamica, non semplicemente una misura statica dell'occupazione dello heap.

---

### Cosa crea pressione di memoria

La pressione di memoria è guidata da una combinazione di fattori:

- alto tasso di allocazione
- grande numero di oggetti attivi
- lunga durata di vita degli oggetti
- recupero inefficiente della memoria

Questi fattori si rafforzano a vicenda e determinano quanto lavoro il runtime deve svolgere per mantenere la memoria utilizzabile.

---

### Allocazione vs retention

Due pattern diversi possono creare pressione:

- **alto tasso di allocazione**
  molti oggetti sono creati e rapidamente scartati
- **alta retention**
  gli oggetti rimangono in memoria per lunghi periodi

Questi pattern creano pressione in modi diversi.

Un alto tasso di allocazione aumenta il churn e la frequenza di raccolta.

Un'alta retention aumenta la quantità di memoria che rimane attiva e deve essere scansionata o preservata.

---

### Esempio: alto tasso di allocazione

```
for (int i = 0; i < 1_000_000; i++) {
    String s = new String("test");
}
```

Caratteristiche:

- molti oggetti a vita breve
- allocazione frequente
- garbage collection frequente

Effetti:

- aumento dell'attività GC
- overhead CPU
- potenziali picchi di latenza

Questo esempio evidenzia una pressione guidata dal churn piuttosto che dalla retention a lungo termine.

---

### Esempio: retention della memoria

```
List<String> cache = new ArrayList<>();

while (true) {
    cache.add(new String("data"));
}
```

Caratteristiche:

- gli oggetti sono mantenuti

- l’utilizzo della memoria cresce continuamente

Effetti:

- aumento dell’utilizzo dell’heap
- cicli di garbage collection più pesanti
- instabilità o fallimento finale

Questo esempio evidenzia una pressione guidata dalla memoria trattenuta piuttosto che dalla sola frequenza di allocazione temporanea.

---

### Sotto carico

Con l’aumentare del carico del sistema:

- più richieste sono elaborate
- più oggetti sono creati
- più oggetti sono trattenuti

Questo porta a:

- aumento del tasso di allocazione
- aumento dell’utilizzo della memoria
- aumento dell’attività GC

La pressione di memoria amplifica:

- la variabilità della latenza
- la latenza di coda

Per questo il degrado legato alla memoria diventa spesso più visibile quando il sistema passa da carico moderato a carico sostenuto elevato.

---

### Interazione con la garbage collection

La garbage collection risponde alla pressione di memoria.

Sotto pressione:

- le raccolte diventano più frequenti
- le pause possono aumentare
- l’utilizzo della CPU cresce

In casi estremi:

- la GC domina l’esecuzione
- il lavoro utile diminuisce

Quando questo accade, il runtime sta spendendo una quota significativa del suo sforzo di lavoro nella gestione stessa della memoria invece che nell’elaborazione del lavoro applicativo.

---

### Sintomi osservabili

La pressione di memoria appare spesso come:

- picchi di latenza senza un chiaro collo di bottiglia CPU
- degrado della latenza di coda (p95, p99)
- pause periodiche
- aumento della frequenza GC
- crescita dell’utilizzo della memoria nel tempo

Questi sintomi sono particolarmente importanti perché possono essere scambiati per lentezza generica se il comportamento della memoria non viene analizzato direttamente.

---

### Intuizione pratica

Un sistema può apparire:

- poco carico (CPU moderata)
- ma comunque lento

Questo spesso indica:

- pressione di memoria
- overhead legato alla GC

Questo è uno dei motivi principali per cui la sola CPU non è sufficiente per valutare la salute del sistema.

---

### Modello semplificato

Il comportamento del sistema può essere approssimato come:

- tasso di allocazione ↑ → attività GC ↑
- retention ↑ → utilizzo della memoria ↑
- attività GC ↑ → variabilità della latenza ↑

Queste relazioni non sono lineari.

Dipendono dalla strategia del runtime, dalla forma del carico di lavoro, dalla durata di vita degli oggetti e dalla quantità di dati attivi.

---

### Implicazioni pratiche

Per gestire la pressione di memoria:

- comprendere i pattern di allocazione
- identificare gli oggetti a lunga vita
- monitorare il comportamento GC
- correlare metriche di memoria con la latenza

L'ottimizzazione dovrebbe concentrarsi su:

- ridurre allocazioni non necessarie
- controllare la durata di vita degli oggetti
- evitare retention non limitata

In molti casi, la soluzione più efficace non è il tuning del collector, ma la riduzione del lavoro di memoria che il runtime è costretto a eseguire.

---

### Collegamento con concetti precedenti

La pressione di memoria contribuisce a:

- degrado non lineare (→ 1.5.3 Non-linear degradation)
- collasso del throughput (→ 1.5.4 Throughput collapse)
- amplificazione della latenza di coda (→ 1.5.5 Tail latency amplification)

È quindi un ponte diretto tra gli interni del runtime e il comportamento visibile del sistema sotto carico.

---

### Interpretazione pratica

La pressione di memoria spiega perché un sistema può degradare anche quando non è evidentemente limitato dalla CPU o bloccato esternamente.

Un runtime sotto stress al livello della memoria può apparire attivo, ma produrre latenza crescente, throughput ridotto e comportamento instabile.

Questo rende la pressione di memoria una delle cause nascoste più importanti nel degrado delle performance dei runtime gestiti.

---

### Idea chiave

La pressione di memoria deriva dall'interazione tra allocazione, retention e garbage collection sotto carico.

Comprendere questa interazione è essenziale per spiegare problemi di latenza e stabilità nei sistemi reali.

---

◀ 1.6 – Concorrenza e parallelismo | ▲ Index | 01-08-resource-level-performance ▶

## 1.8 – Performance a livello di risorse

Questo capitolo investiga come le risorse fondamentali di sistema si comportano sotto carico e come esse possano vincolare le performance.

Ci si concentra su CPU, I/O, network e sulle modalità in cui possano emergere colli di bottiglia quando una delle risorse si satura prima delle altre.

Comprendere la performance a livello di risorse è essenziale perché il degrado del sistema è spesso il risultato visibile di limiti delle risorse piuttosto che della sola logica applicativa.

## Indice



---

## 1.8.1 Comportamento della CPU

### Definizione

La **CPU** è il componente responsabile dell'esecuzione delle istruzioni.

Le performance della CPU sono determinate non solo da quanto velocemente le istruzioni vengono eseguite, ma da come l'esecuzione viene schedulata tra carichi di lavoro concorrenti.

Questa distinzione è importante perché il degrado legato alla CPU è spesso causato da pressione di scheduling, accodamento e contese, piuttosto che solo dal costo computazionale.

---

### Utilizzo della CPU vs saturazione

L'**utilizzo della CPU** rappresenta quanto della capacità della CPU venga usato.

Un utilizzo elevato non è necessariamente indice di un eventuale problema.

La **saturazione della CPU** si verifica quando:

- c'è più lavoro di quanto la CPU possa eseguire
- i thread sono pronti a eseguire ma non possono essere schedulati immediatamente

Distinzione chiave:

- **alto utilizzo** → la CPU è occupata
- **saturazione** → la CPU è sovraccarica

Un sistema può quindi mostrare un elevato utilizzo della CPU e continuare comunque a comportarsi in modo accettabile, finché il lavoro eseguibile non si accumula più velocemente di quanto la CPU possa elaborarlo.

---

### Scheduling e run queue

I thread non eseguono in modo continuo.

Sono schedulati dal sistema operativo.

Ad ogni momento:

- alcuni thread sono in **esecuzione**
- alcuni sono **in attesa** di eseguire (run queue)

Quando il numero di thread eseguibili supera il numero dei core CPU disponibili:

- i thread si accumulano nella run queue
- i ritardi di scheduling aumentano

Questo impatta direttamente la latenza (→ 1.5 System behavior under load) e può essere investigato usando le relazioni di concorrenza (→ 1.2.1 Little's Law (system-level concurrency)).

La run queue è quindi un segnale critico di pressione della CPU, perché mostra non solo che la CPU è occupata, ma che c'è del lavoro che è in attesa di essere eseguito.

---

### Comportamento osservabile (esempio)

Un sistema sotto pressione della CPU mostra un numero crescente di thread eseguibili.

```
$ vmstat 1
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 7  0      0  12000  45000 300000    0    0     2     1 1200 3000 90  8  2  0  0
 8  0      0  11000  45000 300000    0    0     1     2 1300 3200 92  6  2  0  0
```

Interpretazione:

- run queue ( `r` ) alta → thread in attesa della CPU
- CPU idle ( `id` ) vicino a zero → nessuna capacità disponibile
- utilizzo della CPU ( `us + sy` ) vicino alla saturazione

Questo indica che ci sono thread pronti ad eseguire ma che non possono essere schedulati immediatamente per mancanza di core disponibili (→ 1.6 Concurrency and parallelism).

Il punto importante è che la saturazione della CPU non è definita solo da valori percentuali, ma dalla presenza di lavoro eseguibile che non può progredire immediatamente.

---

### Impatto sulle performance

Quando la CPU diventa satura:

- i ritardi di scheduling aumentano
- il tempo di risposta aumenta
- il throughput può stabilizzarsi o diminuire

Questo effetto è non lineare (→ 1.5.3 Non-linear degradation).

Con l'aumentare della saturazione della CPU, l'applicativo può spendere (progressivamente) più tempo ad attendere di essere schedulato per l'esecuzione piuttosto che a svolgere lavoro utile.

---

### Interazione con la concorrenza

La concorrenza aumenta il numero di thread attivi.

Con la crescita della concorrenza:

- più thread competono per la CPU
- la lunghezza della run queue aumenta
- l'overhead di scheduling aumenta

Oltre un certo punto:

- aggiungere thread riduce le performance invece di migliorarle (→ 1.6 Concurrency and parallelism).

Questo è il motivo per cui aggiungere più lavoro concorrente non produce sempre un throughput migliore.

Se il tempo CPU diventa la risorsa limitante, la concorrenza si trasforma in pressione di scheduling.

---

### Implicazioni pratiche

Per ragionare sul comportamento della CPU:

- distinguere utilizzo da saturazione
- osservare i thread eseguibili, non solo la %CPU
- correlare le metriche CPU con la latenza (→ 1.2 Core metrics and formulas)

I problemi CPU spesso non riguardano il puro utilizzo, ma la **contesa per l'esecuzione**.

È quindi possibile che un sistema appaia "pienamente occupato" senza essere instabile, oppure che appaia solo moderatamente occupato mostrando già ritardi di scheduling.

---

### Interpretazione pratica

L'analisi della CPU dovrebbe concentrarsi sulla capacità del sistema di tenere il passo con il lavoro eseguibile.

Una CPU occupata non è automaticamente un problema.

Una CPU satura diventa un problema quando i task eseguibili si accumulano, la latenza aumenta e il throughput non scala più con la domanda in ingresso.

---

### Idea chiave

**Le performance della CPU sono limitate dallo scheduling**.

Quando i thread non possono essere schedulati immediatamente, la latenza aumenta anche se il sistema appare pienamente utilizzato.

---

## 1.8.2 I/O e disco

### Definizione

Le **operazioni di I/O** implicano lettura da o scrittura verso dispositivi di storage.

A differenza delle operazioni CPU, l'I/O è tipicamente più lento e spesso bloccante.

Questo significa che molti problemi di performance che coinvolgono l'I/O sono dominati dal tempo di attesa piuttosto che dal calcolo attivo.

---

### Latenza vs throughput

Le performance dell'I/O hanno due dimensioni chiave:

- **latenza** → tempo per completare una singola operazione
- **throughput** → numero di operazioni per unità di tempo

Un throughput alto non garantisce una bassa latenza.

Un sistema può muovere una grande quantità di dati complessiva mentre le singole richieste sperimentano comunque tempi di attesa significativi.

---

### Comportamento bloccante

Molte operazioni di I/O sono bloccanti:

- un thread avvia un'operazione
- attende fino al suo completamento

Durante questo tempo:

- il thread non esegue lavoro utile
- può mantenere risorse (lock, connessioni)

Questo è uno dei motivi principali per cui i colli di bottiglia di I/O spesso si propagano in pressione sui thread pool, accodamento e riduzione della concorrenza effettiva.

---

### Effetti di accodamento

Quando più richieste eseguono I/O:

- le operazioni si accodano al livello del dispositivo
- il tempo di attesa aumenta

Con l'aumentare della lunghezza della coda:

- la latenza aumenta
- la variabilità aumenta (→ 1.5 System behavior under load)

Questo può essere espresso come ritardo di accodamento (→ 1.2.3 Service time vs response time (queueing)).

Il punto importante è che il costo dell'I/O non è limitato alla durata dell'operazione in sé.

Include anche il tempo speso ad aspettare che le operazioni precedenti siano completate.

---

### Comportamento osservabile (esempio)

Un sistema sotto pressione di I/O mostra tempi di attesa crescenti.

```
$ iostat -x 1
Device            r/s     w/s   await   %util
sda              120      80     35.0    95.0
sda              130      90     42.0    98.0
```

Interpretazione:

- `await` alto → le richieste spendono un tempo significativo in attesa
- `%util` vicino al 100% → il dispositivo è saturo
- latenza crescente indica accumulo di coda

Questo riflette effetti di accodamento (→ 1.2 Core metrics and formulas).

Il valore `await` crescente è particolarmente importante, perché spesso rivela che il dispositivo non è semplicemente occupato, ma sempre più incapace di assorbire il lavoro in ingresso senza ritardo aggiuntivo.

---

### Impatto sulle performance

Quando l'I/O diventa un collo di bottiglia:

- la latenza delle richieste aumenta
- il throughput può degradare
- i thread spendono più tempo ad attendere che ad eseguire

Questo può ridurre la capacità effettiva del sistema anche quando l'utilizzo della CPU rimane moderato.

Un sistema può quindi essere limitato dall'I/O senza apparire limitato dalla CPU.

---

### Interazione con la concorrenza

Più richieste concorrenti portano a:

- più operazioni di I/O
- code sul dispositivo più lunghe
- latenza aumentata

Aumentare la concorrenza non migliora le performance se il dispositivo è saturo (→ 1.6 Concurrency and parallelism).

Oltre un certo punto, concorrenza aggiuntiva aumenta solo l'attesa e peggiora il tempo di risposta.

---

### Implicazioni pratiche

Per ragionare sul comportamento dell'I/O:

- concentrarsi sulla latenza ( `await` ), non solo sul throughput
- identificare l'accumulo di coda
- correlare l'attesa di I/O con la latenza applicativa (→ 1.5 System behavior under load)

I problemi di I/O sono spesso fraintesi perché il throughput può rimanere accettabile mentre la latenza degrada significativamente.

---

### Interpretazione pratica

Le performance dell'I/O dovrebbero essere valutate come un sistema di attesa.

La domanda centrale non è solo quante operazioni al secondo il dispositivo possa supportare, ma quanto a lungo le operazioni attendano quando il carico di lavoro si intensifica.

Un sottosistema di storage che si comporta bene a bassa concorrenza può degradare bruscamente quando le richieste iniziano ad accumularsi.

---

### Idea chiave

**Le performance dell'I/O sono dominate dal tempo di attesa**.

Quando le code crescono, la latenza aumenta e la responsività del sistema degrada.

---

## 1.8.3 Comportamento della rete

### Definizione

Le performance di **rete** sono determinate dal trasferimento di dati tra sistemi.

Dipendono sia dalla latenza sia dalla larghezza di banda.

Nei sistemi distribuiti, il comportamento della rete è spesso un contributore principale al tempo di risposta end-to-end, specialmente quando le richieste attraversano più servizi.

---

### Latenza e round trip

La comunicazione di rete richiede spesso scambi multipli.

Ogni scambio introduce:

- ritardo di trasmissione
- ritardo di propagazione
- ritardo di elaborazione

Round trip multipli amplificano la latenza totale (→ 1.5 System behavior under load).

Questo è particolarmente importante nelle catene di richieste in cui ogni chiamata di servizio dipende dalla risposta della precedente.

Anche piccoli ritardi possono accumularsi significativamente attraverso molteplici hop di rete.

---

### Limitazioni di larghezza di banda

La larghezza di banda definisce la quantità di dati che possono essere trasferiti per unità di tempo.

Quando la larghezza di banda è limitata:

- payload grandi richiedono più tempo per essere trasferiti
- il throughput diventa vincolato

La larghezza di banda quindi conta soprattutto quando la quantità di dati trasferiti diventa abbastanza grande da dominare il tempo di comunicazione.

La latenza, al contrario, conta anche per payload piccoli quando sono richiesti molti round trip.

---

### Amplificazione sotto carico

Con l'aumentare del carico:

- più richieste sono inviate sulla rete
- la contesa aumenta
- si possono formare code nei buffer

Questo porta a:

- aumento della latenza
- ritardi di pacchetti o ritrasmissioni (→ 1.5.5 Tail latency amplification)

Sotto carico, la variabilità della rete diventa particolarmente importante perché ritardi occasionali possono influenzare solo una parte del traffico degradando comunque l'esperienza utente complessiva.

---

### Comportamento osservabile (esempio)

Un sistema sotto pressione di rete mostra accumulo di connessioni e di code.

```
$ ss -s
Total: 1200
TCP:   900 (estab 850, timewait 30)

Transport Total     IP        IPv6
*         1200      -         -
RAW       0         0         0
UDP       50        40        10
TCP       870       800       70
```

Interpretazione:

- grande numero di connessioni stabilite → alta concorrenza
- accumulo di connessioni può indicare elaborazione lenta o ritardi di rete

Un numero crescente di connessioni aperte può indicare che le richieste non stanno completando abbastanza rapidamente, o perché i servizi downstream sono lenti o perché il sistema non è in grado di elaborare efficientemente il lavoro di rete.

---

### Impatto sulle performance

I vincoli di rete portano a:

- aumento del tempo di risposta
- maggiore variabilità
- ritardi a cascata tra servizi

Nelle architetture distribuite, questi ritardi spesso si propagano e si amplificano perché una singola interazione di rete lenta può ritardare molte operazioni dipendenti.

---

### Interazione con il design del sistema

I sistemi distribuiti amplificano gli effetti della rete:

- più servizi introducono più hop di rete
- la latenza si accumula attraverso le chiamate (→ 1.5 System behavior under load)

Un sistema con molti confini di servizio può quindi soffrire di latenza indotta dalla rete anche quando ogni singola chiamata appare relativamente poco costosa.

---

### Implicazioni pratiche

Per ragionare sul comportamento della rete:

- considerare il numero di round trip
- osservare i pattern di connessione
- correlare l'attività di rete con la latenza

È anche importante distinguere tra:

- comportamento limitato dalla larghezza di banda
- comportamento limitato dalla latenza
- ritardo indotto dalle dipendenze

Questi sono problemi correlati ma non identici.

---

### Interpretazione pratica

Le performance di rete non riguardano solo quanto velocemente si muovano i byte.

Riguardano anche quanto spesso i sistemi comunicano, quante dipendenze sono coinvolte e come i ritardi in un componente influenzano gli altri.

In molte architetture a servizi, ridurre round trip non necessari può migliorare la latenza più efficacemente che aumentare semplicemente la larghezza di banda.

---

### Idea chiave

**Le performance di rete sono guidate dalla latenza e dai pattern di comunicazione**.

Sotto carico, piccoli ritardi si accumulano e impattano significativamente il tempo di risposta.

---

## 1.8.4 Saturazione delle risorse e colli di bottiglia

### Definizione

Un **collo di bottiglia** (bottleneck) è la risorsa che limita le performance del sistema.

La saturazione si verifica quando quella risorsa opera a capacità piena o in intervalli prossimi alla propria capacità limite.

Questo è il punto in cui carico di lavoro aggiuntivo non si traduce più in throughput utile proporzionale.

---

### Identificare la risorsa limitante

In ogni momento, le performance del sistema sono vincolate da una risorsa dominante:

- CPU
- I/O
- rete
- memoria (indirettamente tramite GC → 1.7 Runtime and memory model)

Identificare questa risorsa è essenziale.

Senza identificare la reale risorsa limitante, gli sforzi di ottimizzazione spesso prendono di mira i sintomi piuttosto che le cause.

---

### Principio del singolo collo di bottiglia

Anche nei sistemi complessi:

- le performance sono tipicamente limitate da un vincolo primario

Migliorare risorse non limitanti ha poco effetto.

Questo principio è una delle ragioni per cui la performance engineering deve rimanere orientata al sistema.

Molte risorse possono apparire attive, ma solo una, di solito, determina il limite di capacità corrente.

---

### Effetti a cascata

Quando una risorsa diventa satura:

- le code si accumulano
- la latenza aumenta
- i componenti upstream rallentano

Questo può propagarsi attraverso il sistema (→ 1.5 System behavior under load).

Un collo di bottiglia locale può quindi diventare un problema esteso all'intero sistema, poiché i ritardi si diffondono a chiamanti, worker, pool e servizi dipendenti.

---

### Interazione tra risorse

Le risorse non sono indipendenti:

- I/O lento aumenta il tempo di attesa dei thread → influenza lo scheduling della CPU (→ 1.8.1 CPU behavior)
- i ritardi di rete aumentano la durata delle richieste → aumentano l'utilizzo della memoria (→ 1.7 Runtime and memory model)
- la saturazione della CPU ritarda l'elaborazione → aumenta la dimensione delle code (→ 1.2.1 Little's Law (system-level concurrency))

Questa interazione spiega perché i colli di bottiglia spesso si spostano o appaiono accoppiati al variare delle condizioni di carico di lavoro.

Il fattore limitante può cambiare quando una parte del sistema viene migliorata o quando cambia la composizione del carico di lavoro.

---

### Pattern osservabili

Segni comuni di colli di bottiglia:

- CPU vicina alla saturazione con run queue alta
- latenza I/O in aumento con elevato utilizzo del dispositivo
- ritardi di rete con conteggio di connessioni crescenti

Questi pattern sono utili perché collegano sintomi a livello di sistema con comportamenti specifici delle risorse.

Aiutano a ridurre l'ambiguità diagnostica.

---

### Impatto sul comportamento del sistema

Quando viene raggiunto un collo di bottiglia:

- il throughput smette di aumentare
- la latenza cresce rapidamente
- il sistema diventa instabile sotto ulteriore carico

Questo corrisponde a:

- degrado non lineare (→ 1.5.3 Non-linear degradation)
- collasso del throughput (→ 1.5.4 Throughput collapse)

In questa fase, domanda aggiuntiva spesso peggiora la situazione invece di aumentare l'output utile.

---

### Implicazioni pratiche

Per analizzare le performance:

- identificare la risorsa satura
- correlare le metriche di risorsa con la latenza
- concentrare l'ottimizzazione sul fattore limitante

Una diagnosi corretta dipende quindi dalla comprensione non solo di quali risorse siano occupate, ma di quale di esse stia attualmente determinando il comportamento dell'intero sistema.

---

### Interpretazione pratica

L'analisi dei colli di bottiglia è il ponte tra osservazione e azione.

Lo scopo non è semplicemente raccogliere metriche di CPU, I/O o rete, ma determinare quale risorsa stia vincolando il lavoro utile nel punto operativo corrente.

Una volta identificata quella risorsa, l'ottimizzazione diventa significativa.

---

#### Idea chiave

**Le performance del sistema sono limitate dal suo collo di bottiglia**.

Comprendere quale risorsa sia satura è essenziale per spiegare e migliorare il comportamento sotto carico.

---

◀ 01-07-runtime-and-memory-model | ▲ Index | 01-09-common-performance-problems ▶

## 1.9 – Problemi comuni di performance

Questo capitolo descrive problemi comuni di performance che appaiono nei sistemi reali sotto carico.

Questi problemi non appartengono a categorie isolate. Spesso interagiscono, si rafforzano a vicenda e diventano visibili come crescita della latenza, perdita di throughput, instabilità o degrado in coda.

Lo scopo di questo capitolo è collegare sintomi ricorrenti ai meccanismi sottostanti già introdotti nei capitoli precedenti.

## Indice



---

## 1.9.1 Inefficienza CPU-bound

#### Definizione

Un'inefficienza CPU-bound si verifica quando il sistema spende eccessivo tempo CPU svolgendo un lavoro che potrebbe essere ridotto, ottimizzato o addirittura evitato.

Questo non significa necessariamente che il sistema sia sempre CPU-saturo.

Significa che il tempo CPU disponibile viene consumato in modo inefficiente, riducendo la quantità di lavoro utile che il sistema può svolgere prima di raggiungere la saturazione.

---

#### Cause tipiche

- algoritmi inefficienti (es. complessità non necessaria)
- calcoli ripetuti
- mancanza di caching per operazioni costose
- trasformazioni di dati eccessive

Queste cause sono comuni perché l'inefficienza CPU emerge spesso da codice funzionalmente corretto ma strutturalmente dispendioso.

Nella performance engineering, l'inefficienza è maggiormente impattante quando si riscontra in hot path o in operazioni altamente ripetitive.

---

### Esempio

```
public int countMatches(List<String> items, String target) {
    int count = 0;
    for (String s : items) {
        if (s.toLowerCase().equals(target.toLowerCase())) {
            count++;
        }
    }
    return count;
}
```

Interpretazione:

- chiamate ripetute a `toLowerCase()` creano lavoro non necessario
- il tempo CPU aumenta con la dimensione dell'input
- calcolo evitabile negli hot path

Il problema non è solo il costo del loop in sé, ma la trasformazione ripetuta di valori che potrebbero essere normalizzati una sola volta invece che a ogni confronto.

---

### Meccanismo

L'inefficienza CPU-bound spreca capacità di esecuzione.

Viene consumato più tempo CPU del necessario per produrre lo stesso risultato.

Con la crescita del carico di lavoro:

- l'utilizzo della CPU aumenta prima
- il lavoro eseguibile si accumula prima
- il throughput utile raggiunge prima il suo limite

Questo trasforma codice inefficiente in un collo di bottiglia al livello di sistema, quando il volume delle richieste aumenta.

---

### Impatto sotto carico

- aumento dell'utilizzo della CPU
- riduzione del throughput
- saturazione della CPU anticipata

Questo porta a ritardi di scheduling (→ 1.8.1 CPU behavior) e a crescita non lineare della latenza (→ 1.5.3 Non-linear degradation).

In termini pratici, il sistema raggiunge il proprio limite CPU prima del previsto, lasciando meno margine per burst o crescita concorrente del traffico.

---

### Sintomi osservabili

I sintomi tipici includono:

- alto utilizzo della CPU sotto carico moderato
- latenza in aumento con l'aumentare del volume di richieste
- throughput che si appiattisce prima del previsto
- tempo CPU significativo speso in operazioni ripetute o evitabili

Questi sintomi spesso appaiono prima della saturazione totale della CPU e inizialmente possono sembrare un generico problema di scalabilità.

---

### Implicazioni pratiche

- ottimizzare gli hot path
- evitare lavoro ripetuto
- ridurre la complessità algoritmica

È anche importante identificare quali inefficienze contano davvero al livello del sistema.

Un'operazione inefficiente eseguita una volta può essere irrilevante.

La stessa inefficienza eseguita milioni di volte diventa un collo di bottiglia.

---

### Interpretazione pratica

L'inefficienza CPU è una delle ragioni più comuni per cui un sistema non riesce a scalare nonostante hardware apparentemente adeguato.

Il problema non è la mancanza di CPU in termini assoluti, ma il cattivo utilizzo della CPU disponibile.

L'ottimizzazione è quindi tanto più preziosa quanto più aumenta la quantità di lavoro utile svolto, per unità di tempo CPU.

---

### Idea chiave

L'inefficienza CPU riduce la quantità di lavoro utile che il sistema può svolgere prima di raggiungere la saturazione.

---

## 1.9.2 Allocazione eccessiva e churn di memoria

### Definizione

L'allocazione eccessiva si verifica quando il sistema crea un gran numero di oggetti a vita breve, aumentando il churn di memoria e la pressione sul runtime.

Questo è un problema comune nei managed runtime, dove l'allocazione è spesso poco costosa per operazione, ma che diventa molto dispendiosa, in aggregato, quando viene eseguita eccessivamente e sotto carico.

---

### Esempio

```
for (Order o : orders) {
    result.add(new ReportRow(o.getId(), o.getAmount(), o.getStatus()));
}
```

Interpretazione:

- molti oggetti sono creati per iterazione
- gli oggetti hanno vita breve
- il tasso di allocazione aumenta

Se questo pattern appare in codice eseguito frequentemente, il volume totale di allocazione può diventare significativo anche quando ogni singolo oggetto resta poco impattante.

---

### Meccanismo

- un alto tasso di allocazione aumenta il churn di memoria
- la garbage collection viene eseguita più frequentemente

(→ 1.7.2 Allocation and object lifecycle)
(→ 1.7.3 Garbage collection)

Il sistema soffre quindi non solo nella fase di creazione degli oggetti, ma per tracciarli, eliminarli e gestire, in generale, gli effetti sul runtime di un frequente turnover della memoria.

---

### Impatto sotto carico

- aumento dell'attività GC
- overhead CPU per la gestione della memoria
- variabilità della latenza

Questo contribuisce alla pressione sulla memoria (→ 1.7.4 Memory pressure and performance).

Con l'aumentare del carico, l'overhead legato all'allocazione diventa spesso più visibile attraverso pause, jitter e allargamento dei percentili di latenza.

---

### Sintomi osservabili

I sintomi tipici includono:

- aumento della frequenza della garbage collection
- picchi periodici di latenza
- gap crescente tra latenza media e latenza di coda
- utilizzo moderato della CPU con tempi di risposta instabili
- comportamento della memoria che degrada con l'aumentare del throughput

Questi sintomi sono particolarmente comuni nei sistemi che allocano pesantemente nei percorsi di elaborazione delle richieste.

---

### Implicazioni pratiche

- ridurre la creazione non necessaria di oggetti
- riutilizzare oggetti quando appropriato
- analizzare i pattern di allocazione

È anche importante distinguere tra:

- allocazione necessaria
- allocazione evitabile
- allocazione trattenuta che avrebbe dovuto invece esser temporanea

Questa distinzione aiuta a determinare se il problema sia churn, retention o entrambi.

---

### Interpretazione pratica

L'allocazione eccessiva è spesso invisibile in code review perché il codice rimane semplice e corretto.

Il suo effetto diventa visibile solo a runtime, quando la creazione ripetuta di oggetti cambia il comportamento della GC e la pressione di memoria.

Un sistema può quindi apparire logicamente efficiente e comunque comportarsi male perché crea troppo traffico di memoria transiente.

---

### Idea chiave

Il churn di memoria aumenta l'overhead del runtime e introduce variabilità della latenza.

---

## 1.9.3 Contesa e hot spot di sincronizzazione

### Definizione

La contesa (contention) si verifica quando più thread competono per la stessa risorsa, forzando un accesso serializzato.

Un hot spot di sincronizzazione è una parte del sistema in cui questa competizione diventa concentrata e ritarda ripetutamente l'esecuzione.

Questi hot spot sono particolarmente problematici perché riducono il parallelismo effettivo esattamente dove ci si aspetta che la concorrenza possa aiutare.

---

### Esempio

```
public class Counter {
    private int value = 0;

    public synchronized void increment() {
        value++;
    }
}
```

Interpretazione:

- l'accesso è serializzato attraverso la sincronizzazione
- solo un thread progredisce alla volta
- il throughput è limitato dalla sezione critica

Il problema non è che la sincronizzazione esista, ma che un percorso condiviso e frequentemente acceduto possa diventare il punto limitante per l'intero sistema.

---

### Meccanismo

- i thread si bloccano mentre aspettano il lock
- la contesa aumenta con la concorrenza

(→ 1.6 Concurrency and parallelism)

Quando più thread competono per la stessa sezione sincronizzata:

- il tempo di attesa cresce
- il parallelismo effettivo diminuisce
- più tempo viene speso nel coordinamento che nel progresso

Questo fa sì che il sistema si comporti come se il suo livello di concorrenza fosse inferiore a quanto il numero di thread suggerisca.

---

### Impatto sotto carico

- aumento del tempo di attesa
- riduzione del throughput
- aumento della latenza

Questo porta a effetti di accodamento (→ 1.5 System behavior under load).

Sotto carico più elevato, gli hot spot di sincronizzazione diventano spesso visibili come crescita della latenza senza crescita proporzionale della CPU, perché i thread sono in attesa invece di eseguire lavoro.

---

### Sintomi osservabili

I sintomi tipici includono:

- latenza in aumento con utilizzo moderato della CPU
- molti thread bloccati o in attesa
- scalabilità ridotta con l'aumentare della concorrenza
- throughput limitato da una piccola sezione critica
- percorsi di codice con uso intensivo di lock che appaiono negli hot path di esecuzione

Questi sintomi sono spesso fuorvianti perché il sistema può apparire solo parzialmente utilizzato pur essendo già vincolato.

---

#### Implicazioni pratiche

- minimizzare lo stato mutabile condiviso
- ridurre la dimensione della sezione critica
- usare pattern di concorrenza più scalabili

È anche importante identificare se il collo di bottiglia sia causato da:

- scope del lock
- frequenza di accesso
- sezioni critiche lunghe
- sincronizzazione non necessaria

Cause diverse richiedono soluzioni diverse.

---

#### Interpretazione pratica

I problemi di contention sono spesso fraintesi come lentezza generica.

In realtà, il problema centrale è la serializzazione: molti thread sono presenti, ma solo pochi stanno progredendo nel lavoro utile.

La performance engineering quindi non si preoccupa soltanto d'aggiungere concorrenza, ma deve soprattutto assicurarsi che la concorrenza presente non collassi in attesa.

---

#### Idea chiave

**La contesa converte lavoro parallelo in esecuzione serializzata**.

---

### 1.9.4 Colli di bottiglia dovuti a blocking e attesa

#### Definizione

Il blocking si verifica quando un thread aspetta che un'operazione esterna sia completata, impedendogli di svolgere lavoro utile.

Questo include l'attesa di:

- I/O
- risposte di rete
- lock
- servizi esterni
- altri eventi coordinati

Il blocking è spesso necessario, ma diventa un collo di bottiglia quando troppe risorse di esecuzione sono occupate ad attendere invece che a progredire.

---

### Esempio

```
public String fetchData() throws Exception {
    Thread.sleep(50); // simulate blocking call
    return "data";
}
```

Interpretazione:

- il thread è inattivo durante l'attesa
- le risorse rimangono allocate
- la concorrenza non si traduce in throughput

Il thread esiste, ma non sta facendo avanzare lavoro utile durante il periodo di blocco.

---

### Meccanismo

- i thread spendono tempo ad aspettare invece che ad eseguire
- i thread pool possono saturarsi

(→ 1.6 Concurrency and parallelism)

Quando più thread si bloccano:

- meno thread rimangono disponibili per nuovo lavoro
- l'accodamento appare al livello del modello di esecuzione
- la latenza cresce anche se la CPU non è pienamente utilizzata

Questo è il motivo per cui i colli di bottiglia da blocking spesso coesistono con un utilizzo moderato della CPU.

---

### Impatto sotto carico

- aumento della latenza
- riduzione del throughput
- esaurimento dei thread

Questo amplifica accodamento e saturazione (→ 1.5 System behavior under load).

Sotto carico sostenuto, il comportamento bloccante crea spesso un loop di feedback in cui le richieste in coda aspettano thread che, a loro volta, stanno aspettando operazioni lente.

---

### Sintomi osservabili

I sintomi tipici includono:

- molti thread in stati di attesa o bloccati
- code di richieste in crescita
- CPU moderata con throughput scarso
- latenza in aumento durante operazioni heavy di I/O o heavy di dipendenze
- thread pool che appaiono pieni senza corrispondente lavoro produttivo

Questi sintomi sono particolarmente comuni nei servizi che mescolano concorrenza delle richieste con chiamate downstream sincrone.

---

### Implicazioni pratiche

- ridurre le operazioni bloccanti
- usare pattern asincroni o non bloccanti quando appropriato
- dimensionare con attenzione i thread pool

È anche utile distinguere tra:

- blocking inevitabile
- blocking evitabile
- blocking collocato in percorsi di esecuzione ad alta frequenza

Questa distinzione aiuta a identificare dove sia necessario un redesign.

---

### Interpretazione pratica

Il blocking riduce la concorrenza effettiva.

Un sistema può avere molti thread, ma se una grand parte di essi è in attesa, il sistema si comporta come se avesse molta meno capacità di esecuzione.

Questo è il motivo per cui i problemi di blocking sono spesso problemi del modello di esecuzione prima di diventare problemi di pura risorsa.

---

### Idea chiave

Il blocking riduce la concorrenza effettiva e limita il throughput del sistema.

---

## 1.9.5 Accumulo di code ed effetti di saturazione

### Definizione

L'accumulo di code si verifica quando il lavoro in ingresso supera la capacità di elaborazione, causando l'attesa delle richieste prima che siano elaborate.

Questo è uno dei problemi di performance più comuni e più importanti, perché il queueing trasforma un sovraccarico magari moderato in una latenza rapidamente crescente.

---

### Meccanismo

- il tasso di arrivo supera la capacità di servizio
- le code crescono nel tempo

Questo può essere descritto usando Little's Law (→ 1.2.1 Little's Law (system-level concurrency)).

Mentre la domanda in ingresso continua e l'elaborazione rimane limitata, l'attesa si accumula e il tempo di risposta inizia a includere un ritardo di coda sempre più grande.

---

### Impatto sotto carico

- il tempo di attesa aumenta
- il tempo di risposta aumenta
- la latenza diventa instabile

Questo porta a degrado non lineare (→ 1.5.3 Non-linear degradation) e a limiti di throughput.

Una volta che l'accodamento diventa dominante, il sistema può deteriorarsi molto rapidamente anche se l'aumento originario del carico era relativamente piccolo.

---

### Sintomi osservabili

- lunghezze di coda crescenti
- tempi di risposta in aumento
- throughput stabile o in diminuzione

Altri sintomi possono includere:

- burst di errori di timeout
- ampliamento della latenza p95/p99
- recupero ritardato dopo sovraccarico temporaneo

Questi effetti spesso indicano che il sistema sta operando vicino o oltre la sua capacità effettiva.

---

### Implicazioni pratiche

- controllare la concorrenza
- aumentare la capacità della risorsa che è collo di bottiglia
- ridurre il tasso di arrivo se necessario

È anche importante determinare dove si stia formando la coda:

- thread pool
- connection pool
- dispositivo
- buffer di rete
- servizio downstream

La posizione della coda spesso rivela il vero collo di bottiglia.

---

### Interpretazione pratica

L'accumulo di code non è solo un dettaglio operativo.

Spesso è il meccanismo diretto attraverso cui il sovraccarico diventa visibile agli utenti.

Un sistema può ancora funzionare, ma una volta che il lavoro inizia ad attendere in modo sistematico, la crescita della latenza diventa inevitabile.

---

### Idea chiave

**Le code crescono quando la domanda supera la capacità, determinando la latenza**.

---

## 1.9.6 Amplificazione delle dipendenze e latenza a cascata

### Definizione

L'amplificazione delle dipendenze si verifica quando la latenza in un componente si propaga e aumenta la latenza attraverso il sistema.

Questo problema è particolarmente importante nei sistemi distribuiti, dove una richiesta spesso dipende da più chiamate downstream prima di potersi completare.

---

### Meccanismo

- le richieste dipendono da più servizi downstream
- i ritardi si accumulano attraverso le chiamate
- componenti lenti influenzano l'intero sistema

Anche quando ogni singolo ritardo è piccolo, l'effetto totale può diventare significativo una volta che più dipendenze, retry o catene di chiamate seriali siano coinvolte.

---

### Esempio

```
public Response process() {
    Data a = serviceA.call();
    Data b = serviceB.call();
    return combine(a, b);
}
```

Interpretazione:

- la latenza totale dipende da più dipendenze
- la dipendenza più lenta domina il tempo di risposta

Nei sistemi reali, questo effetto diventa più forte quando le richieste dipendono da molti servizi, database remoti o operazioni sincrone concatenate.

---

### Impatto sotto carico

- amplificazione della latenza attraverso i servizi
- aumento della variabilità
- degrado della latenza di coda

(→ 1.5.5 Tail latency amplification)

Sotto carico, l'amplificazione delle dipendenze diventa spesso più severa perché sistemi downstream lenti trattengono thread, richieste e code upstream per periodi più lunghi.

---

### Sintomi osservabili

I sintomi tipici includono:

- aumenti improvvisi di latenza senza saturazione locale della CPU
- degrado del comportamento p95/p99 causato dalla variabilità downstream
- catene di richieste che diventano più lente mentre una dipendenza rallenta
- instabilità che si diffonde da un servizio a un altro
- retry e timeout che aumentano la pressione attraverso il sistema

Questi sintomi sono spesso difficili da interpretare senza correlare il comportamento attraverso più componenti.

---

### Implicazioni pratiche

- minimizzare il numero di dipendenze sincrone
- usare timeout e strategie di fallback
- isolare i componenti lenti

È anche utile identificare:

- quale dipendenza contribuisce maggiormente al ritardo end-to-end
- se le chiamate siano seriali o parallele
- se i retry peggiorino il problema
- se i componenti lenti inneschino accodamento upstream

Questo trasforma un vago problema di "lentezza distribuita" in un comportamento di sistema diagnosticabile.

---

### Interpretazione pratica

La latenza di un sistema non è determinata solo dal "proprio codice".

Spesso è determinata dalla dipendenza più lenta nel percorso della richiesta.

Più dipendenze ha un sistema, più è probabile che la variabilità in un punto diventi visibile ovunque.

---

#### Idea chiave

**La latenza del sistema è spesso determinata dalla dipendenza più lenta**.

---

---

◀ 01-08-resource-level-performance | ▲ Index | 01-10-diagnostics-and-analysis ▶

## 1.10 – Diagnostica e analisi

Questo capitolo si interessa del come le problematiche di performance possano essere investigate, interpretate e validate.

Ci si concentra qui sui processi utilizzati per muovere dall'osservazione del sistema a una valutazione difendibile della performance dello stesso.

La diagnostica infatti non è solo una pratica di raccolta dei dati.

È la disciplina che si preoccupa di interpretare correttamente quei dati e di collegare i sintomi ai meccanismi di funzionamento sottostanti.

## Indice



---

## 1.10.1 Osservabilità e segnali

#### Definizione

La diagnostica parte evidentemente da segnali osservabili.

Questi segnali forniscono visibilità spesso indiretta sul comportamento interno del sistema sotto carico.

Non espongono direttamente i meccanismi di funzionamento, ma ne riflettono piuttosto gli effetti.

Per questo l'osservabilità è essenziale nella performance engineering: i problemi interni sono raramente visibili direttamente, ma lasciano spesso tracce misurabili rispetto a latenza, throughput, comportamento delle risorse e queueing.

---

#### Segnali fondamentali

I segnali primari sono:

- latenza (p50, p95, p99)
- throughput
- tasso di errore
- utilizzo delle risorse (CPU, memoria, I/O, rete)
- lunghezze delle code

(→ 1.2 Core metrics and formulas)
(→ 1.8 Resource-level performance)

Ogni segnale cattura una diversa dimensione del comportamento del sistema.

Solo un loro esame combinato fornisce una vista significativa del sistema.

- La latenza mostra l'impatto visibile per l'utente.
- Il throughput mostra il tasso di lavoro produttivo.
- Il tasso degli errori indica il comportamento in caso di guasto.
- Gli indici delle risorse mostrano dove la capacità viene consumata.
- Le code mostrano dove il lavoro si accumula.

---

### Caratteristiche dei segnali

I segnali devono essere:

- **accurati** → riflettere il comportamento reale
- **granulari** → esporre la distribuzione (es. percentili, non solo medie)
- **correlati nel tempo** → allineati attraverso tutti i componenti

Senza queste proprietà, l'interpretazione diventa inaffidabile, fuorviante o addirittura erronea.

Una metrica mal posta, non bene configurata o addirittura scollegata dall'intervallo di tempo pertinente può nascondere proprio quel meccanismo di funzionamento che intende invece rivelare.

---

### Qualità del segnale e interpretazione

La presenza di segnali non è tuttavia da sola sufficiente.

I segnali devono anche essere:

- pertinenti rispetto alle domande che ci si pone
- osservati rispetto al livello appropriato (sistema, servizio, risorsa, dipendenza)
- interpretati nel contesto

Per esempio:

- l'utilizzo CPU senza informazione sulla run queue può nascondere pressione di scheduling
- latenza media senza analisi dei percentili può nascondere instabilità in coda
- utilizzo della memoria senza comportamento della GC può nascondere pressione del runtime

Il valore diagnostico di una metrica dipende non solo dalla sua esistenza, ma da come viene correlata col resto delle evidenze.

---

### Implicazioni pratiche

Una diagnostica efficace necessità di:

- osservazione complessiva sei segnali
- correlare tali segnali nel tempo
- evitare il ragionamento basato su singole metriche

Osservare una metrica al di fuori di un contesto è spesso fuorviante rispetto alla comprensione della meccanica sottostante.

Questo è uno dei principali motivi per cui spiegazioni semplicistiche sono pericolose nell'analisi delle performance.

Un singolo numero può descrivere un sintomo, ma raramente spiega il comportamento complessivo del sistema in oggetto.

---

### Interpretazione pratica

L'osservabilità è la materia prima della diagnostica.

Senza segnali, non esiste analisi affidabile.
Con segnali di scarsa qualità, l'analisi sarà inaffidabile.
Con segnali ben strutturati, l'analisi diventa verificabile e ripetibile.

La diagnostica inizia dunque non con l'ottimizzazione, ma con l'analisi di quanto osservato.

---

### Idea chiave

La diagnostica dipende sia dalla disponibilità sia dalla corretta interpretazione dei segnali osservabili.

---

## 1.10.2 Sintomo vs causa

### Definizione

Un sintomo è un effetto osservabile.

Una causa è il meccanismo sottostante che produce quell'effetto.

Questa distinzione è fondamentale perché la maggior parte dei problemi di performance viene scoperto attraverso sintomi, non attraverso una manifestazione diretta della causa alla radice del problema.

---

### Distinzione

Sintomi tipici:

- elevata latenza
- importante utilizzo della CPU
- aumento del tasso di errore
- garbage collection frequente

Questi elementi descrivono *che cosa sta succedendo*, non *perché sta succedendo*.

Un sistema può mostrare lo stesso sintomo per ragioni molto diverse, e la stessa causa può produrre sintomi diversi a seconda del carico, del timing e dell'architettura.

---

### Esempio

- un elevato utilizzo della CPU può risultare da:
    - calcolo inefficiente
    - retry eccessivi
    - pressione di memoria
    - contesa
- un'elevata latenza può risultare da:
    - accumulo di code
    - ritardi di I/O
    - sincronizzazione

(→ 1.9 Common performance problems)

Per queste ragioni i sintomi devono essere trattati come punti d'accesso all'investigazione, non come spiegazioni.

---

### Implicazione diagnostica

Lo stesso sintomo può essere prodotto da cause diverse.

Senza identificare il meccanismo sottostante, le azioni correttive possono prendere di mira la parte sbagliata del sistema.

Per esempio:

- ridurre l'utilizzo della CPU può non ridurre la latenza se la causa radice è il queueing I/O
- fare tuning della GC può non aiutare se il tasso di allocazione d'oggetti rimane invariato

Una fix tecnicamente plausibile può quindi avere poco effetto se affronta una sola conseguenza visibile.

---

### Perché avviene la confusione

Sintomi e cause sono spesso confusi perché i sintomi sono relativamente facili da osservare.

Metriche, dashboard e sistemi di monitoraggio di solito mostrano:

- valori elevati
- che cosa è lento
- che cosa sta fallendo

Non spiegano automaticamente:

- perché i valori sono alti
- perché sono lenti
- perché stanno fallendo

Questo divario tra visibilità e spiegazione è esattamente ciò che la diagnostica deve colmare.

---

### Interpretazione pratica

Un buon processo diagnostico tratta ogni sintomo come un indizio, non come una conclusione.

L'obiettivo è passare da:

- "questa metrica è anomala"

a:

- "questo meccanismo sta producendo il comportamento anomalo"

Questo spostamento è ciò che distingue un ragionamento efficace sulle performance dal monitoraggio superficiale.

---

### Idea chiave

Il comportamento osservato non è la causa.

La diagnosi richiede di mappare i sintomi ai meccanismi sottostanti che li generano.

---

## 1.10.3 Correlazione e causalità

### Definizione

La correlazione è la variazione simultanea di due segnali.

La causalità è una relazione diretta in cui un fattore ne produce un altro.

Questa distinzione è essenziale nella diagnostica perché molte metriche si muovono insieme sotto carico, ma non tutte sono causalmente collegate nella stessa direzione.

### Errore comune

Due metriche cambiano insieme:

- la CPU aumenta
- la latenza aumenta

Questo non implica che la CPU sia la causa della latenza.

La correlazione può indicare:

- una causa sottostante comune
- una dipendenza indiretta
- una catena causale nella direzione opposta
- o semplice coincidenza nella stessa finestra temporale

---

### Esempio

Possibili interpretazioni:

- saturazione CPU → ritardi di scheduling → latenza
- ritardi di I/O → più thread concorrenti → maggiore utilizzo della CPU
- contesa → retry → sia CPU sia latenza aumentano

(→ 1.5 System behavior under load)
(→ 1.8 Resource-level performance)

In tutti e tre i casi, CPU e latenza si muovono insieme, ma il meccanismo sottostante è diverso.

---

### Implicazione diagnostica

La correlazione è un punto di partenza, non una conclusione.

Più meccanismi possono produrre gli stessi segnali correlati.

Solo un modello causale spiega come uno conduca all'altro.

Per questa ragione, il ragionamento diagnostico deve andare oltre "queste due metriche si sono mosse nello stesso momento".

Deve spiegare:

- quale è cambiata per prima
- quale meccanismo le collega
- perché la sequenza osservata è coerente con il comportamento del sistema

---

### Approccio pratico

Per stabilire la causalità:

- identificare la sequenza degli eventi
- verificare la coerenza con il comportamento noto del sistema
- validare attraverso osservazione o cambiamento controllato

Questo può includere:

- confrontare stati prima/dopo
- osservare se una metrica precede costantemente un'altra
- cambiare una condizione e verificare la risposta attesa

La causalità diventa più forte quando il sistema si comporta come il meccanismo proposto prevede.

---

### Limiti dell’analisi superficiale

Una dashboard può mostrare la correlazione molto chiaramente ma non può, da sola, provare la causalità.

Per questo la diagnostica richiede ragionamento e non soltanto “visualizzazione”.

Un performance engineer deve chiedersi:

- Questa metrica è il driver, la conseguenza o una ulteriore conseguenza dello stesso evento?
- La timeline supporta la spiegazione proposta?
- La spiegazione rimane coerente attraverso osservazioni ripetute?

Senza queste domande, la correlazione può facilmente portare a conclusioni scorrette.

---

### Interpretazione pratica

Una buona diagnostica tratta la correlazione come un generatore di ipotesi.

Aiuta a identificare dove guardare, ma non elimina la necessità di ragionare sui meccanismi sottostanti.

Questo è particolarmente importante nei sistemi complessi dove più colli di bottiglia interagiscono e i sintomi si propagano attraverso i componenti.

---

### Idea chiave

Non inferire causalità dalla correlazione.

La diagnosi richiede di identificare il meccanismo che collega i segnali.

---

## 1.10.4 Costruire un’ipotesi

### Definizione

Un’ipotesi è una spiegazione proposta che collega segnali osservati a un meccanismo del sistema.

Fornisce un modo strutturato per passare dall’osservazione alla spiegazione.

Senza un’ipotesi, l’analisi rimane descrittiva piuttosto che diagnostica.

---

### Processo

Un’ipotesi viene costruita:

1. osservando i segnali
2. identificando pattern coerenti
3. mappandoli su meccanismi noti

(→ 1.2 Core metrics and formulas)
(→ 1.5 System behavior under load)

Questo processo trasforma dati grezzi in una spiegazione testabile.

Collega:

- misurazioni
- comportamento del sistema
- ragionamento causale

---

### Esempio

Osservato:

- la latenza aumenta
- la lunghezza della coda aumenta
- la CPU si avvicina alla saturazione

Ipotesi:

- aumento del tasso di lavoro in arrivo → accumulo di coda → tempo di attesa più lungo → saturazione CPU

Questo collega segnali osservabili a un meccanismo di accodamento.

Fornisce anche una direzione all'investigazione: verificare se l'aumento della latenza sia causato principalmente dall'attesa piuttosto che da un tempo di servizio più lento.

---

### Requisiti

Un'ipotesi valida deve essere:

- coerente con i dati osservati
- fondata sul comportamento del sistema
- testabile attraverso misurazione o cambiamento

Un'ipotesi che non può essere testata può essere plausibile, ma non è ancora utile per la diagnostica.

Un'ipotesi che contraddice l'evidenza osservata dovrebbe essere rigettata anche se appare intuitiva.

---

### Implicazione diagnostica

Un'ipotesi guida l'investigazione.

Senza di essa, l'analisi diventa reattiva e non strutturata.

Invece di passare direttamente dal sintomo alla fix, il processo diagnostico dovrebbe passare da:

- sintomo
- ipotesi su meccanismo candidato
- validazione

Questa struttura riduce il guesswork e rende le conclusioni diagnostiche più robuste.

---

### Fonti delle ipotesi

Le ipotesi di solito emergono da:

- combinazioni di segnali osservati
- pattern di performance noti
- comportamento precedente del sistema
- conoscenza architetturale
- scenari di errore ripetuti

Per esempio:

- latenza crescente + code in crescita spesso suggerisce accodamento
- CPU moderata + thread bloccati può suggerire contesa o attesa I/O
- frequenza GC crescente + picchi di latenza può suggerire pressione di memoria

Queste associazioni non provano la spiegazione, ma forniscono un punto di partenza disciplinato.

---

### Interpretazione pratica

Una buona ipotesi è abbastanza specifica da essere testata e abbastanza generica da spiegare il comportamento osservato.

Non dovrebbe essere:

- vaga (“il sistema è lento”)
- circolare (“la latenza è alta perché le richieste sono lente”)
- puramente descrittiva

Dovrebbe esprimere un meccanismo.

Per esempio:

- “La saturazione del thread pool sta aumentando il tempo di coda, il che sta facendo salire la latenza p95.”

Questo tipo di affermazione può essere validato.

---

### Idea chiave

La diagnosi procede attraverso ipotesi esplicite e testabili, non attraverso assunzioni irrelate.

---

## 1.10.5 Restringere il collo di bottiglia

### Definizione

La diagnostica mira a identificare la risorsa o il meccanismo che limita la performance del sistema.

Questo fattore limitante determina il comportamento complessivo del sistema sotto carico.

Finché non viene identificato, gli sforzi di ottimizzazione rimangono incerti e spesso inefficaci.

---

### Approccio

L’analisi si concentra su:

- comportamento della CPU
- latenza I/O
- ritardi di rete
- pressione di memoria

(→ 1.8 Resource-level performance)
(→ 1.7 Runtime and memory model)

Queste dimensioni vengono esaminate perché la maggior parte dei limiti di performance, alla fine, si manifesta attraverso una o più di esse.

Tuttavia, il collo di bottiglia dominante, in un dato momento, è di solito dato da un solo vincolo primario piuttosto che da tutti i vincoli in ugual misura.

---

### Metodo

- isolare una dimensione alla volta
- confrontare segnali attraverso le risorse
- identificare il vincolo dominante

Questo riduce la complessità concentrandosi sul fattore più impattante.

L’obiettivo non è spiegare ogni metrica, ma trovare il meccanismo che in quel momento governa il comportamento del sistema.

---

### Esempio

Se:

- la CPU è bassa
- la latenza I/O è alta
- le code stanno crescendo

Allora:

- l'I/O è probabilmente il fattore limitante

Il sistema non è CPU-bound, anche se la CPU è attiva.

Questo tipo di restringimento è essenziale perché spesso sono coinvolte più risorse, ma solo una di esse è di solito dominante.

---

### Implicazione diagnostica

La performance è tipicamente limitata, in un dato momento, da un singolo collo di bottiglia dominante.

Ottimizzare risorse non limitanti produce poco o nessun miglioramento.

Questo è uno dei principi più importanti nella diagnostica:

- misurare in modo ampio
- concludere in modo specifico

Un insieme ampio di segnali è richiesto per evitare di perdere evidenze importanti.

Una conclusione specifica è richiesta per indirizzare l'azione sul vincolo reale.

---

### Perché i colli di bottiglia sono difficili da identificare

I colli di bottiglia sono spesso oscurati da effetti secondari.

Per esempio:

- I/O lento può aumentare il numero di thread
- l'aumento del numero di thread può aumentare l'overhead di scheduling della CPU
- l'aumento dell'attesa può gonfiare la retention di memoria
- i retry possono amplificare la domanda su più componenti contemporaneamente

Di conseguenza, l'effetto visibile può non apparire nel punto esatto del problema originario.

Per questo l'isolamento del collo di bottiglia richiede correlazione attraverso i layer piuttosto che interpretazione isolata di una singola metrica.

---

### Interpretazione pratica

Lo scopo della diagnosi non è solo dire che il sistema è sotto pressione.

È identificare:

- dove la pressione diventa limitante
- quale meccanismo produce il limite
- perché quel vincolo è attualmente dominante

Solo allora l'ottimizzazione diventa significativa.

---

### Idea chiave

Una diagnosi efficace riduce il sistema al suo fattore limitante.

## 1.10.6 Analisi iterativa e validazione

### Definizione

La diagnosi è un processo iterativo di test e raffinamento delle ipotesi.

Evolve attraverso osservazioni e validazioni successive.

Questo è necessario perché le spiegazioni iniziali sono spesso incomplete, parzialmente corrette o valide solo per un layer del sistema.

---

### Processo

1. osservare i segnali
2. costruire un'ipotesi
3. testare attraverso cambiamenti o misurazioni
4. validare o rigettare

Ogni passaggio produce un raffinamento nella comprensione del sistema.

Questo loop viene ripetuto finché la spiegazione proposta è coerente con il comportamento osservato e supportata dall'evidenza.

---

### Esempio

```
ExecutorService pool = Executors.newFixedThreadPool(10);

for (int i = 0; i < 1000; i++) {
    pool.submit(() -> {
        Thread.sleep(100);
        return null;
    });
}
```

Interpretazione:

- il thread pool fisso limita l'esecuzione parallela
- i task si accumulano
- l'accodamento aumenta la latenza

Questa ipotesi può essere testata:

- aumentando la dimensione del pool
- riducendo il tempo di blocking

Se la latenza diminuisce e l'accumulo di code si riduce, l'ipotesi guadagna evidenza.

Se il comportamento non cambia come previsto, la spiegazione deve essere rivista.

---

### Validazione

Un'ipotesi è validata se:

- i cambiamenti producono gli effetti attesi
- i segnali evolvono in modo coerente con il meccanismo proposto

In caso contrario, l'ipotesi deve essere rivista.

La validazione quindi dipende dalla coerenza tra:

- cambiamento osservato
- cambiamento atteso

- spiegazione causale proposta

Una fix che cambia una metrica senza migliorare il comportamento del sistema può indicare che è stato preso di mira il meccanismo sbagliato.

---

### Implicazioni pratiche

- evitare conclusioni in un solo passaggio
- iterare sistematicamente
- validare le assunzioni con dati osservabili

Una buona diagnostica raramente è istantanea.

Diventa affidabile attraverso confronto ripetuto tra:

- ciò che viene osservato
- ciò che ci si aspetta
- ciò che cambia realmente dopo l'intervento

Questa disciplina iterativa è ciò che trasforma il troubleshooting in engineering.

---

### Perché l'iterazione conta

I sistemi complessi raramente espongono una spiegazione complessiva in una singola osservazione.

È comune scoprire che:

- un collo di bottiglia iniziale era solo un effetto secondario
- rimuovere un vincolo ne espone un altro
- un miglioramento locale sposta altrove il fattore limitante
- il sistema si comporta diversamente sotto carichi di lavoro diversi

L'iterazione quindi non è un segno di incertezza.

È il metodo normale per arrivare a una spiegazione coerente.

---

### Interpretazione pratica

La diagnosi è un loop perché la comprensione del sistema viene costruita progressivamente.

L'obiettivo non è indovinare correttamente al primo tentativo.

L'obiettivo è passare dall'evidenza alla spiegazione attraverso ragionamento controllato e verifica.

Questo è ciò che rende l'analisi delle performance ripetibile e difendibile.

---

### Idea chiave

La diagnosi è un loop.

La comprensione emerge attraverso iterazione, verifica e raffinamento.

---

◀ 01-09-common-performance-problems | ▲ Index | 01-11-practical-checklists ▶

## 1.11 – Checklist pratiche

Questo capitolo fornisce checklist pratiche per preparare, eseguire e analizzare test di performance.

A differenza dei capitoli precedenti, che spiegano concetti e meccanismi, questo capitolo si concentra sulla disciplina operativa.

L’obiettivo è ridurre errori evitabili e assicurare che i test di performance producano risultati interpretabili, affidabili e utili.

## Indice



---

## 1.11.1 Prima di eseguire un test

### Obiettivi

Definire chiaramente che cosa il test intenda validare.

Obiettivi tipici includono:

- target di latenza
- obiettivi di throughput
- limiti di capacità

Un test senza un obiettivo chiaro può comunque generare dati, ma quei dati saranno difficili da valutare e interpretare.

La prima domanda dovrebbe sempre essere:

- che cosa questo test dovrebbe provare, validare o rivelare?

---

### Definizione del carico di lavoro

Definire il carico di lavoro con precisione:

- tasso di richieste o concorrenza
- mix di richieste
- durata

(→ 1.4 Types of performance tests)

Il carico di lavoro deve essere abbastanza specifico da essere riproducibile e abbastanza realistico da essere significativo.

Un carico di lavoro vago o artificiale può produrre risultati tecnicamente corretti ma operativamente irrilevanti.

---

### Coerenza dell’ambiente

Assicurarsi che:

- l’ambiente di test sia stabile
- la configurazione corrisponda alle assunzioni di produzione
- le dipendenze esterne siano controllate

Se l’ambiente cambia durante il testing, l’interpretazione si rivela impossibile.

I risultati di performance sono confrontabili solo se le condizioni di esecuzione rimangono sufficientemente coerenti.

Questo è particolarmente importante quando si valutano:

- cambiamenti di configurazione

- cambiamenti di codice
- cambiamenti infrastrutturali

---

### Setup delle metriche

Verificare che tutte le metriche richieste siano disponibili:

- percentili di latenza
- throughput
- utilizzo delle risorse
- tasso di errore

(→ 1.2 Core metrics and formulas)

È anche utile assicurarsi che segnali di supporto siano disponibili quando rilevanti, come:

- lunghezze delle code
- timing delle dipendenze
- attività GC
- stati dei thread o dei pool

Il test non dovrebbe iniziare prima che l'osservabilità sia in essere.

---

### Controlli di preparazione

Prima di eseguire il test, confermare che:

- il sistema target sia nello stato atteso
- il monitoring sia attivo
- il generatore di carico di lavoro sia configurato correttamente
- la durata del test sia appropriata per l'obiettivo scelto
- i criteri di successo e fallimento siano ben noti in anticipo

Questo evita un problema comune nel performance testing: eseguire un test tecnicamente valido che in seguito non può essere interpretato con autorevolezza.

---

### Interpretazione pratica

La preparazione è parte del test.

La maggior parte dei risultati inaffidabili non è causata da un comportamento complesso del sistema, ma da una scarsa preparazione del test:

- obiettivi poco chiari
- carico di lavoro non realistico
- ambiente incoerente
- metriche incomplete

Un test ben preparato rende la diagnostica successiva molto più agevole.

---

### Idea chiave

Un test è significativo solo se obiettivi, carico di lavoro e misurazioni sono chiaramente definiti.

---

## 1.11.2 Durante l’esecuzione del test

### Monitoring

Osservare il comportamento del sistema in tempo reale:

- evoluzione della latenza
- stabilità del throughput
- utilizzo delle risorse

Il monitoring durante l’esecuzione è importante perché alcuni problemi sono visibili solo mentre il test è in esecuzione, specialmente:

- saturazione improvvisa
- accodamento inatteso
- recupero instabile
- guasti delle dipendenze

Attendere solamente la fine del test può nascondere comportamenti essenziali che dipendono invece dal tempo.

---

### Controlli di coerenza

Assicurarsi che:

- il carico di lavoro sia applicato come previsto
- nessun disturbo esterno influenzi il test

Questo include verificare che:

- il tasso di richieste previsto sia effettivamente generato
- il mix di operazioni rimanga coerente
- nessuna attività non correlata stia distorcendo i risultati
- gli errori siano causati dalle condizioni di test piuttosto che da rumore esterno

Una discrepanza tra carico di lavoro previsto e carico di lavoro reale può invalidare l’intera interpretazione.

---

### Segnali precoci

Osservare:

- rapido aumento della latenza
- errori inattesi
- saturazione delle risorse

(→ 1.8 Resource-level performance)

Questi sono spesso i primi segnali che il sistema si sta avvicinando a un limite o che il carico di lavoro sta esponendo un collo di bottiglia non anticipato.

L’identificazione precoce è importante perché consente all’operatore del test di:

- catturare evidenze rilevanti
- preservare contesto utile
- evitare di perdere la parte più informativa dell’esecuzione

---

### Osservazioni a runtime

Durante l’esecuzione, è utile osservare non solo valori assoluti, ma anche il cambiamento nel tempo.

Esempi:

- latenza in aumento mentre il throughput rimane stabile

- lunghezze delle code in crescita prima della saturazione della CPU
- errori che appaiono solo dopo una soglia specifica
- degradazione di p95/p99 prima che la media cambi significativamente

Questi pattern spesso rivelano più di snapshot isolate.

Aiutano a distinguere tra:

- instabilità transiente
- sovraccarico stabile
- degrado lento
- collasso improvviso

---

### Disciplina di intervento

Durante un test, evitare di cambiare parametri a meno che il cambiamento non faccia parte del piano di test.

Un intervento non pianificato rende i risultati più difficili da interpretare perché mescola cause multiple nella stessa finestra di osservazione.

Se l'intervento diventa necessario, dovrebbe essere:

- documentato
- marcato temporalmente
- esplicitamente collegato al comportamento osservato

Questo preserva il valore diagnostico dell'esecuzione.

---

### Interpretazione pratica

L'esecuzione è la fase in cui la preparazione teorica incontra il comportamento reale del sistema.

Un test ben progettato può comunque diventare fuorviante se l'operatore non conferma che:

- il carico di lavoro sia corretto
- l'ambiente rimanga stabile
- il sistema si stia comportando come previsto o, cosa importante, in modo inatteso come il test intendeva rivelare

---

### Idea chiave

L'esecuzione non è passiva.

È richiesta osservazione continua per rilevare precocemente le anomalie.

---

## 1.11.3 Dopo l'analisi del test

### Revisione dei dati

Analizzare i dati raccolti:

- distribuzione della latenza
- trend di throughput
- utilizzo delle risorse

La revisione dei dati dovrebbe concentrarsi non solo sui valori medi, ma anche sulla forma del comportamento nel tempo.

Per esempio:

- quando il degradamento è iniziato
- se il throughput ha scalato come previsto
- se la latenza in coda si è ampliata prima che apparissero errori

Questo rende l'analisi più diagnostica e meno descrittiva.

---

### Correlazione

Mettere in relazione i segnali:

- latenza vs CPU
- latenza vs I/O
- errori vs carico

(→ 1.10 Diagnostics and analysis)

La correlazione aiuta a identificare quale risorsa o meccanismo sia più probabilmente associato al degradamento osservato.

Tuttavia, la correlazione dovrebbe essere trattata come un punto di partenza analitico, non come una conclusione finale.

---

### Interpretazione

Identificare:

- colli di bottiglia
- limiti di scalabilità
- pattern anomali

L'interpretazione dovrebbe rispondere a domande come:

- che cosa è cambiato per primo?
- che cosa è degradato dopo?
- quale vincolo è diventato dominante?
- il degrado è stato graduale, brusco o dipendente dal tempo?

Questo è il punto in cui misurazioni grezze diventano comprensione del sistema.

---

### Reporting

Riassumere:

- comportamento osservato
- problemi identificati
- raccomandazioni

Un report descrittivo è più efficace che un mero elenco di numeri.

Dovrebbe spiegare:

- che cosa il sistema era atteso fare
- che cosa ha effettivamente fatto
- dove si è discostato dalle aspettative
- quale evidenza supporta la conclusione

Questo rende i risultati utilizzabili per engineering, operations e test futuri.

---

### Orientamento ai passi successivi

Dopo l'analisi, definire che cosa dovrebbe accadere in seguito.

Questo può includere:

- rieseguire lo stesso test dopo modifiche
- raffinare il realismo del carico di lavoro
- raccogliere diagnostica più profonda
- isolare un sospetto collo di bottiglia
- espandere verso test di stress, soak o capacità

Senza una decisione sui passi successivi, l'analisi rimane informativa ma non operativamente utile.

---

### Interpretazione pratica

L'analisi post-test è il punto in cui la performance engineering diventa presa di decisione.

Lo scopo non è solo dichiarare che una metrica è cambiata, ma spiegare:

- perché il cambiamento è importante
- che cosa implica sul sistema
- che cosa dovrebbe essere fatto dopo

---

### Idea chiave

L'analisi trasforma dati grezzi in comprensione azionabile.

---

## 1.11.4 Errori comuni

### Interpretare male le medie

- le medie nascondono la latenza in coda
- i percentili forniscono una vista più chiara

(→ 1.2.7 Percentiles)

Un sistema può apparire sano in media pur producendo performance inaccettabili per una frazione significativa di richieste.

Questo è uno degli errori più comuni nell'interpretazione dei test.

---

### Ignorare il realismo del carico di lavoro

- carichi di lavoro non realistici producono risultati fuorvianti
- i pattern di produzione devono essere approssimati

Un carico di lavoro troppo sintetico può essere più facile da generare, ma se non riflette il reale mix di richieste, la concorrenza e il comportamento delle dipendenze, le conclusioni possono non trasferirsi alle condizioni di produzione.

Il realismo non richiede riproduzione perfetta, ma richiede un'approssimazione credibile.

---

### Confondere sintomo e causa

- alta CPU non è sempre il problema alla radice
- la latenza deve essere analizzata nel contesto

(→ 1.10 Diagnostics and analysis)

Questo errore spesso porta a ottimizzazione inefficace.

Il sintomo visibile può essere solo la conseguenza di un meccanismo più profondo come accodamento, blocking o rallentamento di una dipendenza.

### Trascurare i colli di bottiglia

- ottimizzare risorse non limitanti ha poco effetto
- il focus deve rimanere sul vincolo dominante

(→ 1.8 Resource-level performance)

Questa è una fonte frequente di sforzo mal posto.

Un sistema può contenere molte imperfezioni, ma solo alcune di esse contano nel punto operativo corrente.

---

### Eseguire test senza criteri di accettazione

Un test è difficile da interpretare se non esiste una definizione preventiva di comportamento accettabile.

Senza soglie esplicite, diventa poco chiaro se il risultato significhi:

- successo
- fallimento
- degrado
- rischio accettabile

I numeri di performance sono utili solo quando confrontati con aspettative definite.

---

### Trattare un solo test come definitivo

Una singola esecuzione di test raramente cattura il comportamento completo di un sistema.

Esecuzioni ripetute possono esporre:

- effetti di warm-up
- variabilità delle dipendenze
- drift di lungo termine
- comportamento di soglia sotto profili di carico diversi

Un'analisi di performance affidabile richiede di solito confronto, ripetizione e validazione.

---

### Ignorare la dimensione temporale

Alcuni problemi non appaiono immediatamente.

Un test breve può perdere:

- crescita lenta della memoria
- accumulo di code ritardato
- degrado graduale delle dipendenze
- instabilità del runtime nel tempo

Per questo la durata del test deve corrispondere al tipo di comportamento che si sta valutando.

---

### Interpretazione pratica

La maggior parte degli errori nel performance testing non è causata da strumenti inappropriati.

È causata da:

- assunzioni deboli
- visibilità incompleta
- cattiva interpretazione
- mancanza di disciplina metodologica

Evitare questi errori è spesso più prezioso che aggiungere maggiore dettaglio di misurazione.

---

### Idea chiave

Assunzioni scorrette portano a conclusioni scorrette.

Evitare errori comuni è essenziale per un'analisi di performance affidabile.

---

◀ 01-10-diagnostics-and-analysis | ▲ Index | ▶