Ars Digitale
Engineering Notes Series

PERFORMANCE ENGINEERING NOTES

Scalability · Latency · Throughput · Observability

Alessandro Fabri

2026 Edition
info@ars-digitale.com

Performance Engineering Guide

Alessandro Fabri

Ars Digitale

Performance Engineering Guide

1.3 – Work of a performance engineer

1.5 – System behavior under load

1.6 – Concurrency and parallelism

Performance Engineering Guide

A structured technical reference for application and system performance engineering

Alessandro Fabri

2026 Edition

Contact: info@ars-digitale.com

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

Copyright

Version: 1.0
Language: English

Contact: info@ars-digitale.com

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

2026 Edition

Preface

This guide is designed as a structured technical reference for performance engineering, with a focus on clarity, precision, and practical usefulness.

It covers application and system behavior under load, diagnostics, bottlenecks, queueing, concurrency, runtime behavior, and resource-level performance.

Each chapter is intended to function both as part of a coherent reading path and as a standalone reference for technical analysis.

About This Book

This book was designed as a practical and structured reference for performance engineering.

It aims to combine:

- technical precision
- conceptual clarity
- system-level reasoning
- operational usefulness
- long-term value as a reference

The EPUB edition is optimized for digital reading and chapter-based navigation.

Contact: info@ars-digitale.com

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

🌎 **Language:** English

Performance Engineering Guide

This index provides the complete **English (EN)** structure of the **Performance Engineering Guide**.

The guide can be read sequentially or used as a technical reference.

This published documentation currently covers the **core technical body of the guide**.

Guide structure

1.1 – Foundations

This section introduces the fundamental concepts required to reason about system performance.

It provides a conceptual model used throughout the guide.

It defines the core principles used in performance engineering to analyze system behavior under load.

Table of Contents

1.1.1 Throughput, latency, concurrency

Definition

These are the three primary dimensions used to describe system performance.

- **Throughput**: number of requests processed per unit of time (e.g. requests per second)
- **Latency**: time required to complete a request (response time)
- **Concurrency**: number of requests being processed at the same time

These concepts are fundamental to performance engineering and are used throughout the guide to describe system behavior.

Relationship

These quantities are not independent.

For a stable system:

- increasing throughput typically increases concurrency
- increasing concurrency tends to increase latency
- latency directly affects how many requests remain “in flight”

This relationship is central to understanding how systems behave under load.

Practical intuition

A system can be viewed as a processing pipeline:

- requests enter
- they are processed
- they exit

At any moment:

- some requests are being processed (concurrency)
- new requests arrive (throughput)
- each request takes time to complete (latency)

This mental model helps reason about flow, accumulation, and delays in real systems.

Example

If a system processes:

- 100 requests per second
- each request takes 200 ms (0.2 s)

then, on average:

- about 20 requests are in flight at any given time

This relationship is formalized by **Little's Law**:

→ 1.2.1 Little's Law

Practical interpretation

Throughput, latency, and concurrency form a closed system.

Changing one of them necessarily impacts the others.

For example:

- reducing latency reduces concurrency for the same throughput
- increasing throughput increases concurrency if latency remains constant
- high concurrency increases the probability of queueing and contention

Understanding this relationship is essential for diagnosing performance issues.

1.1.2 Service time vs response time

Definition

At a resource level, response time is composed of two parts:

- **service time (S)**: time spent doing actual work
- **waiting time (Wq)**: time spent waiting before being processed

This distinction is fundamental in performance analysis.

Relationship

Response time:

- includes both execution and waiting
- increases when queues form

Even if service time remains constant:

- response time can increase significantly due to waiting

This is one of the main reasons systems degrade under load.

Practical meaning

A slow system is often not slow because work is expensive, but because work is waiting.

As load increases:

- queues grow
- waiting dominates
- response time degrades

This decomposition is formalized as:

→ 1.2.3 Service time vs response time

Practical interpretation

Separating service time from response time allows:

- identifying whether the system is CPU-bound or queue-bound
- distinguishing between processing cost and contention
- understanding whether optimization should target execution or waiting

In many real systems, latency issues are caused primarily by queueing rather than computation.

1.1.3 Systems under load

Definition

A system under load processes a continuous stream of incoming requests.

Load is typically expressed as:

- requests per second
- concurrent users
- transactions per second

Load defines the operating conditions under which performance must be evaluated.

Behavior

As load increases:

- resource utilization increases
- queues start to form
- latency increases
- throughput eventually stabilizes or degrades

These effects are not linear and depend on system design and resource constraints.

Key observation

Systems do not degrade linearly.

At low load:

- performance is stable

Near saturation:

- small increases in load can cause large increases in latency

This non-linear behavior is a key characteristic of real-world systems.

Practical interpretation

Understanding system behavior under load is essential for:

- capacity planning
- performance testing
- diagnosing latency issues

It explains why systems may appear stable in testing but fail under slightly higher production load.

1.1.4 Saturation and bottlenecks

Saturation

A resource is saturated when it is busy most or all of the time.

Typical examples:

- CPU at or near 100%
- thread pool fully utilized
- connection pool exhausted

Saturation indicates that a resource cannot handle additional demand without degradation.

Bottleneck

A bottleneck is the resource that limits system throughput.

Characteristics:

- highest utilization
- longest queues
- dominant contribution to response time

The bottleneck determines the overall system capacity.

Practical meaning

Improving non-bottleneck resources has little or no effect.

Performance improvements require:

- identifying the bottleneck
- reducing its demand or increasing its capacity

This is a key principle in performance engineering.

Practical interpretation

In complex systems:

- multiple resources may appear busy
- but only one typically limits throughput at a given time

Correctly identifying the bottleneck is essential to avoid ineffective optimizations.

1.1.5 Why systems slow down

Common mechanisms

Performance degradation is usually driven by a small number of mechanisms:

- queueing due to saturation
- contention on shared resources
- inefficient use of resources
- external dependencies becoming slow

These mechanisms often interact and amplify each other.

Queueing effect

As utilization approaches its limit:

- waiting time increases rapidly
- response time becomes dominated by queueing

This behavior is closely related to utilization and queueing effects:

→ 1.2.2 Utilization Law

Amplification effects

Certain patterns amplify performance problems:

- retries increase load on already saturated systems
- timeouts lead to duplicated work
- cascading dependencies propagate delays

These effects can transform moderate load into severe degradation.

Practical interpretation

Performance degradation is rarely caused by a single factor.

Instead, it emerges from:

- interactions between components
- accumulation of waiting time
- feedback loops under load

Understanding these interactions is essential for effective diagnosis.

Practical conclusion

Most performance problems are not caused by a single slow operation, but by:

- interactions between components
- accumulation of waiting time
- overload conditions

Understanding these mechanisms is essential before applying formulas or running tests.

Key idea

System performance is determined by interactions between workload, resources, and concurrency.

Understanding these interactions is the foundation of performance engineering.

1.2 – Core metrics and formulas

A compact reference of the main formulas used in **application + system performance engineering**.

These formulas formalize the concepts introduced in:

→ 1.1 Foundations

They should be read as a complement to the conceptual model, not in isolation.

They provide the quantitative basis used to reason about system behavior, validate assumptions, and interpret performance test results.

Table of Contents

Notation (typical)

Symbol	Definition
`X` or `λ`	**throughput** / arrival rate (requests per second)
`R` or `W`	**response time** / time in system (seconds)
`S`	**service time** at a resource (seconds per request)
`U`	**utilization** of a resource (0–1)
`L`	**average concurrency** / in-flight requests (count)
`V`	**average number of visits** to a resource per request
`D`	**service demand** on a resource (seconds per request)

This notation is used consistently across the guide and allows formulas to be applied in a uniform way across different contexts.

1.2.1 Little's Law (system-level concurrency)

Definition

Relates average **concurrency** to **throughput** and **time in system**.

Formula

$L = \lambda \cdot W$

Where

- `L` = average number of requests in the system (in-flight / concurrency)
- `λ` = arrival rate / throughput (requests/s)
- `W` = average time in system (s) (often the average end-to-end response time)

Practical meaning

If you know throughput and average response time, you can estimate how many requests are concurrently "in flight".

This makes Little's Law one of the most useful tools for reasoning about system load and concurrency.

Example

If `λ = 200 req/s` and `W = 0.15 s`:

$L = 200 \cdot 0.15 = 30$

About **30** requests are in flight on average.

Practical interpretation

Little's Law connects three observable quantities:

- throughput
- latency
- concurrency

This allows:

- estimating concurrency from measurements
- validating system behavior
- detecting inconsistencies in metrics

It is widely used in performance engineering, capacity planning, and system diagnostics.

1.2.2 Utilization Law (resource-level busy time)

Definition

Utilization is the **fraction of time** a *single resource* is busy during a fixed interval (typically 1 second).
It is "busy time percentage".

Formula

$U = X \cdot S$

Where

- `U` = utilization (0–1)
- `X` = throughput observed by that resource (req/s)
- `S` = mean service time at that resource (s/req)

Resource

A **single service unit**, e.g. CPU core, thread/worker, DB connection, etc.

Example

A DB worker handles `50 req/s`, each query takes `10 ms = 0.01 s`:

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

Interpretation: the resource is busy **0.5 seconds per second**.

Practical interpretation

Utilization is a key indicator of resource saturation.

As utilization approaches 1:

- queueing increases
- latency grows non-linearly
- system stability decreases

This makes utilization one of the most important signals when diagnosing bottlenecks.

1.2.3 Service time vs response time (queueing)

Definition

Response time at a resource includes: - service time (actual work) - queue time (waiting)

Formula

$R = S + W_q$

Where

- `R` = response time at the resource
- `S` = service time
- `W_q` = waiting time in queue

Practical meaning

As utilization approaches saturation, queueing grows non-linearly and **dominates** response time, causing **long-tail latency**.

Practical interpretation

This formula explains why systems slow down under load even when computation cost does not change.

In many real systems:

- service time remains relatively stable
- waiting time increases rapidly

As a result:

- response time is dominated by queueing
- latency becomes unpredictable

Understanding this distinction is essential for diagnosing performance issues.

1.2.4 Service Demand (visits × service time)

Definition

Total service required on a resource per request, accounting for multiple visits.

Formula

$D = V \cdot S$

Where

- `D` = service demand on the resource (s)
- `V` = average visits to the resource per request
- `S` = service time per visit (s)

Example

A request performs `V = 3` DB queries, each takes `S = 5 ms = 0.005 s`:

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

Practical interpretation

Service demand represents the total work required from a resource per request.

It is particularly useful for:

- identifying heavily used resources
- estimating capacity limits
- understanding scaling behavior

Reducing service demand is often more effective than increasing raw capacity.

1.2.5 Throughput

Definition

Requests completed per unit of time.

Formula

Formula: `X = N / T`

Where

- `N` = number of completed requests
- `T` = observation window (seconds)

Practical interpretation

Throughput is one of the primary indicators of system performance.

It reflects the system's ability to process work.

However, throughput must always be interpreted together with:

- latency
- error rate
- resource utilization

High throughput alone does not guarantee acceptable system behavior.

1.2.6 Error rate

Definition

Fraction of requests that fail (timeouts, 5xx, etc.).

Formula

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

Practical interpretation

Error rate reflects system reliability under load.

An increase in error rate often indicates:

- overload conditions
- resource exhaustion
- instability

Error rate should always be monitored together with latency and throughput.

1.2.7 Percentiles (p50, p95, p99)

Definition

The `p`-th percentile is the value below which **p% of observations** fall.

- `p50` ≈ median (“typical request”)
- `p95` = threshold for the slowest 5%
- `p99` = threshold for the slowest 1%

Percentiles capture **distribution** and **tail behavior** better than averages.

Practical interpretation

Percentiles are essential for understanding real user experience.

In many systems:

- average latency appears acceptable
- tail latency (p95/p99) is significantly worse

This difference is critical for system evaluation and SLO definition.

1.2.7.1 How to compute a percentile (ordered sample)

Given `N` values sorted ascending:

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

Compute the theoretical position:

Formula: `P = (p / 100) × (N + 1)`

- If `P` is an integer → percentile = `v_P`
- If not, let `k = floor(P)` and `δ = P - k` (fractional part), then interpolate:

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

> Note: percentile definitions vary slightly across tools. This method is a clear and commonly used approach and is excellent for reasoning in interviews/assessments.

1.2.7.2 Interpretation vs average (why tails matter)

- If `p50` is much lower than the mean, the distribution is **right-skewed** (few slow requests inflate the mean).
- If `p95` or `p99` is far above the mean, you have **long-tail latency**.

A typical pattern: - mean looks “acceptable” - `p95/p99` are bad
→ user experience is degraded for a non-negligible fraction of users and SLOs are at risk.

Practical interpretation

Percentiles highlight behavior that averages hide.

They are essential for:

- defining service level objectives (SLOs)
- detecting tail latency issues
- understanding worst-case behavior

Ignoring percentiles often leads to incorrect conclusions about system performance.

1.2.8 Empirical CDF (threshold → percentage)

Definition

Given a threshold `t`, the empirical cumulative distribution function (CDF) tells the fraction of samples at or below `t`.

Formula

Formula: `F(t) = count(x_i ≤ t) / N`

Practical meaning

CDF answers: “If my SLO is 200 ms, what % of requests meet it?”

Percentiles answer the inverse: “What threshold corresponds to 95% of requests?”

Practical interpretation

CDF and percentiles are complementary views of the same data.

- CDF: given a threshold → what fraction meets it
- Percentile: given a fraction → what threshold corresponds

Both are useful for performance analysis and SLO validation.

1.2.9 Long-tail latency (what it is)

Definition

A small fraction of requests (e.g. 5% or 1%) is **much slower** than the majority.

Why the tail “dominates”

- SLOs are typically defined on `p95/p99`, so tails drive pass/fail.
- In distributed systems, the slowest dependency often determines end-to-end latency.
- Tail events are frequently driven by **contention/queueing**.

Common causes (high-level)

- thread pool / connection pool saturation (queueing)
- lock contention / synchronization hot spots
- slow DB queries, missing indexes, lock waits
- retries + timeouts amplifying tail latency
- hot keys in caches / uneven shard load

- GC pauses / memory pressure (stop-the-world)
- network jitter / packet loss / retransmissions
- disk I/O spikes, compactions, fsync/wal flush

Practical interpretation

Long-tail latency is one of the most critical aspects of system performance.

It explains why:

- average metrics can appear acceptable
- user experience is still degraded

Managing tail latency is often more important than improving average performance.

1.2.10 Quick checklist (what to measure in tests)

- Latency: `p50/p90/p95/p99`
- Throughput: `RPS/TPS`
- Error rate: `timeouts/5xx`
- Utilization: CPU, memory, DB, pools
- Queue lengths: thread pools, connection pools, message backlogs
- Dependency timings: DB/Redis/external APIs

Practical interpretation

These metrics form the minimal set required to understand system behavior during performance tests.

They allow:

- identifying bottlenecks
- detecting instability
- correlating workload with system behavior

Measuring only a subset of these metrics often leads to incomplete or misleading analysis.

Key idea

Formulas are not isolated abstractions.

They are tools used to explain observed behavior and validate system models.

Understanding how to apply them is essential for performance engineering.

1.3 – Work of a performance engineer

This section describes what performance engineering is in practice and how it is applied to real systems.

Table of Contents

1.3.1 What performance engineering is (in practice)

Definition

Performance engineering is the discipline of understanding, measuring, and controlling how a system behaves under load.

It is not limited to performance testing, nor to a specific tool or technology.

It is a way of reasoning about systems under load or when they are stressed.

It focuses on behavior as a whole, not on isolated metrics or individual components.

Performance and non-functional requirements

Performance engineering does not focus on a single property.

When a system is exercised under load, a subset of **non-functional requirements (NFRs)** becomes visible:

- latency and throughput (performance)
- scalability (vertical and horizontal)
- stability and resilience under stress
- resource usage and efficiency
- capacity limits

These properties are not independent.

They emerge together as the system is pushed.

Load acts as a forcing function that reveals how the system behaves.

A system that appears correct under low load may exhibit completely different behavior when stressed.

What performance engineering actually observes

Under load, a system reveals:

- how work flows through its components
- how resources are consumed
- where contention appears
- where queues form
- which limits are reached first

This requires:

- understanding the system model (→ 1.1 Foundations)
- measuring key metrics (→ 1.2 Core metrics and formulas)
- identifying limiting factors

The objective is not only to observe behavior, but also to explain it.

Not just testing

Performance engineering is often reduced to load testing.

In practice, testing is only one part of the work.

Tests are used to:

- expose system behavior
- validate assumptions
- reproduce problems

But performance engineering also includes:

- analyzing system design
- investigating production issues
- dimensioning resources (heaps, pools, threads, connections)
- explaining observed behavior

Testing without analysis produces data without understanding.

Practical perspective

In real scenarios, the work typically involves:

- preparing and calibrating test environments
- applying load or stress to reveal problems (often white-box)
- identifying and fixing bottlenecks
- validating behavior with controlled use cases
- dimensioning system components (CPUs, memory, pools, concurrency limits)
- tuning configuration and parameters
- running benchmarks to establish reference points
- executing long-duration (soak / endurance) tests to validate stability over time

These activities are not isolated.

They are part of a continuous process aimed at understanding system limits.

Key idea

Performance engineering is not about making a system faster in isolation.

It is about understanding how a system behaves when it is pushed, and ensuring that it remains:

- predictable
- stable
- scalable

Most issues are not caused by a single slow operation, but by:

- interactions between components
- accumulation of waiting time
- saturation of shared resources

Understanding these mechanisms is the core of performance engineering.

1.3.2 Typical workflow

Performance engineering is an iterative process where the system is progressively exercised, analyzed, stabilized, and understood under increasing levels of load.

The objective is not only to detect problems, but to build a reliable model of how the system behaves under realistic conditions.

1.3.2.1 Environment preparation and calibration

- verify and align test environment with production characteristics (as much as possible)
- verify configurations (CPU, memory, pools, connections)
- ensure observability (metrics, logs, traces)

Goal:

- establish a reliable baseline
- ensure repeatability of results

Without calibration, measurements are difficult to interpret and comparisons become unreliable.

1.3.2.2 Use case definition and workload modeling

Before applying load, the workload must be defined.

A system is not tested in isolation, but through the requests it processes.

This requires identifying:

- critical user and system paths
- typical operations (read, write, batch, background jobs)
- relative frequency of each operation
- concurrency patterns

A realistic workload includes:

- a mix of use cases
- weighted distribution (e.g. percentages of traffic)
- different request types and costs

Workload definition is one of the most critical steps.

Incorrect workload leads to misleading conclusions.

Non-functional requirements (NFRs)

In parallel with workload definition, **non-functional requirements** must be clarified.

These define what is considered an **acceptable system behavior**.

Typical examples:

- throughput targets (e.g. 30 req/s)
- concurrency levels (e.g. 500 concurrent users)
- latency objectives (e.g. p95 < 200 ms)
- error rate thresholds
- resource usage constraints

NFRs may be:

- explicitly defined by stakeholders
- partially defined
- missing or inconsistent

In all cases, they must be:

- reviewed
- validated
- made measurable

Practical implication

Workload and NFRs must be aligned.

For each use case:

- the expected load must be defined
- the acceptable behavior must be known

Otherwise:

- results cannot be evaluated
- tests cannot be considered successful or failed

Incorrect workload definition or missing NFRs leads to results that are technically correct but not actionable.

1.3.2.3 Initial load / stress testing (problem discovery)

The first phase under load aims to expose major issues.

Typical goals:

- identify obvious bottlenecks
- detect functional failures under load
- reveal instability (timeouts, crashes, saturation)

This phase is often:

- exploratory
- iterative
- partially white-box (using internal visibility)

The objective is discovery, not precision.

1.3.2.4 Analysis and bottleneck identification

Once issues appear, the system must be analyzed.

This involves:

- correlating metrics (latency, throughput, utilization)
- identifying where time is spent
- locating saturation points and queues

Typical questions:

- which resource is saturated?
- where does latency accumulate?
- what limits throughput?

This step relies on:

→ 1.1 Foundations
→ 1.2 Core metrics and formulas

1.3.2.5 Fixes and iterative validation

After identifying bottlenecks, fixes are applied.

These may include:

- code changes
- configuration updates
- resource adjustments

Each fix must be validated by re-running tests.

This creates an iterative loop:

- **Test → Analyze → Fix → Test** again

The goal is to progressively stabilize the system.

1.3.2.6 Intermediate validation (stable baseline)

Before moving to long-duration tests, the system must reach a stable baseline.

This means:

- no critical failures under expected load
- predictable behavior
- controlled latency and error rates

This phase ensures that:

- major issues are resolved
- results are reproducible

1.3.2.7 Long-duration validation (soak / endurance)

Once the system is stable, it must be observed over time.

This phase evaluates behavior under sustained load.

Typical goals:

- detect slow memory leaks
- observe resource accumulation (threads, connections, buffers)
- identify performance degradation over time
- validate long-term stability

This phase is essential because some issues:

- do not appear immediately
- emerge only after prolonged execution

The results of this phase directly impact:

- system dimensioning
- capacity planning
- runtime configuration

1.3.2.8 Dimensioning and capacity definition

Based on previous observations and also from unitary testing after the phase of baseline stabilization, system components are dimensioned.

This includes:

- heap and memory configuration
- thread pools and connection pools
- concurrency limits
- infrastructure sizing
- clustering

The goal is to define:

- how much load the system can handle
- under which conditions it remains stable
- what margins are required

Dimensioning must be based on observed behavior, not assumptions.

1.3.2.9 Tuning

Once dimensioning is defined, tuning refines system behavior.

Typical areas:

- garbage collection parameters
- thread scheduling and pool sizing
- database and connection settings
- caching strategies

Tuning aims to:

- reduce latency
- improve stability
- optimize resource usage

It is often iterative and context-dependent.

1.3.2.10 Verification and regression

After tuning, the system must be re-validated.

This includes:

- re-running key scenarios
- verifying that improvements are effective
- ensuring no regressions are introduced

This phase ensures consistency and reliability.

1.3.2.11 Benchmarking and reference points

Finally, benchmarks are established.

These provide:

- reference performance metrics
- comparison points across versions
- validation against expectations

Benchmarks are not goals by themselves.

They are used to:

- understand system behavior
- track evolution over time

Key idea

Performance engineering is an iterative loop:

- **define workload → test → analyze → fix → validate → optimize**

The objective is not only to improve performance, but to understand system limits and ensure predictable behavior under load.

1.3.3 Black-box vs white-box

Performance engineering can be approached from two complementary perspectives:

- black-box (external observation)
- white-box (internal observation)

Both are required to understand system behavior under load.

1.3.3.1 Black-box approach

In a black-box approach, the system is observed from the outside.

Only externally visible behavior is measured:

- response time
- throughput
- error rate

The internal implementation is not considered.

What it provides

Black-box observation allows:

- validating system behavior from a user perspective
- measuring end-to-end performance
- detecting visible failures under load

It answers questions such as:

- Is the system fast enough?
- Does it handle the expected load?
- Does it fail under stress?

Limitations

Black-box alone cannot explain:

- where time is spent
- which resource is saturated
- why performance degrades

It shows symptoms, not causes.

1.3.3.2 White-box approach

In a white-box approach, internal system behavior is observed.

This includes:

- resource utilization (CPU, memory, disk, network)
- thread and connection pools
- internal queues
- component-level timings

White-box observation provides a level of **introspection into the system execution**.

In many cases, this includes visibility close to the code level:

- method-level timings
- call paths and execution flows
- hotspots (slow or frequently executed methods)
- allocation patterns and memory behavior
- lock contention and synchronization points

What it provides

White-box observation allows:

- identifying bottlenecks
- understanding where time is spent
- detecting contention and queueing
- analyzing resource saturation

It answers questions such as:

- Which component is slow?
- Where is latency accumulated?
- What limits throughput?
- Which part of the execution is responsible for the slowdown?

Limitations

White-box alone does not guarantee:

- correct end-to-end behavior
- acceptable user experience

A system can appear efficient internally but still fail under real workload conditions.

1.3.3.3 Observability and tooling

Observability provides the data required for white-box analysis.

It typically includes:

- system and application metrics (e.g. CPU usage, latency, throughput)
- logs (events, errors, state changes)
- traces (request flow across components)
- application performance monitoring (APM)

These sources provide continuous visibility into system behavior.

Diagnostic artifacts

In addition to continuous observability, deeper analysis often relies on diagnostic artifacts.

These are typically collected on demand and provide a snapshot of the system state.

Common examples include:

- thread dumps (thread states, locks, contention)
- heap dumps (memory usage, object retention, leaks)
- profiling snapshots (CPU and allocation profiling)
- core dumps (process-level failure analysis)

These artifacts allow:

- inspection of internal execution state
- identification of blocking threads and deadlocks
- analysis of memory leaks and retention paths
- detailed investigation of performance anomalies

They are usually heavier and more intrusive than observability tools, and are used selectively during diagnostics.

1.3.3.4 Combining both approaches

Effective performance engineering requires combining both perspectives.

Typical workflow:

- use black-box to detect issues
- use white-box to explain them
- validate improvements again with black-box

This creates a feedback loop:

- **observe → analyze → fix → validate**

Key idea

Black-box observation reveals that a problem exists.

White-box observation explains why it exists.

Both are necessary to understand and control system behavior under load.

1.3.4 Load testing vs diagnostics

Load testing and diagnostics are often confused.

They serve different purposes and operate at different levels.

Both are required to understand system behavior under load.

1.3.4.1 Load testing

Load testing applies controlled workload to the system.

It is used to:

- observe behavior under specific conditions
- measure latency, throughput, and error rates
- validate assumptions about capacity and scalability

Load testing operates primarily at the **black-box level**:

- requests are generated externally
- responses are measured externally

What it provides

Load testing answers questions such as:

- Can the system handle the expected load?
- What happens when load increases?
- When does performance degrade?
- What is the maximum sustainable throughput?

Limitations

Load testing alone does not explain:

- why the system slows down
- which component is responsible
- how resources are used internally

It reveals behavior, but not causes.

1.3.4.2 Diagnostics

Diagnostics investigates the internal behavior of the system.

It is used to:

- identify bottlenecks
- understand execution paths
- analyze resource usage
- explain observed performance issues

Diagnostics operates at the **white-box level**:

- internal metrics are analyzed
- traces and execution paths are inspected
- diagnostic artifacts may be collected

What it provides

Diagnostics answers questions such as:

- Where is time spent?
- Which resource is saturated?
- Which component is responsible for latency?
- What causes performance degradation?

Tools and techniques

Diagnostics typically relies on:

- metrics, logs, and traces
- application performance monitoring (APM)
- thread dumps and heap dumps
- profiling and execution analysis

Limitations

Diagnostics without load testing may miss:

- real workload conditions
- interactions between components
- behavior under stress

It can explain a problem, but not necessarily reproduce it.

1.3.4.3 Relationship between load testing and diagnostics

Load testing and diagnostics must be combined.

Typical workflow:

- apply load to expose behavior
- use diagnostics to analyze internal state
- apply fixes
- validate again with load testing

This creates a loop:

- observe → explain → fix → validate

Key idea

Load testing reveals that a problem exists.

Diagnostics explains why it exists.

Neither is sufficient on its own.

Understanding system behavior requires both.

1.3.5 What actually matters (and what doesn’t)

Performance engineering involves many tools, metrics, and techniques.

However, not all of them are equally important.

Understanding what matters is essential to avoid wasting effort and drawing incorrect conclusions.

What matters

The most important aspects are:

- **understanding system behavior under load**
- **identifying bottlenecks and limiting factors**
- **using realistic workloads and validated NFRs**
- **reasoning about interactions between components**
- **measuring and interpreting results correctly**

Performance engineering is primarily about:

- building a mental model of the system
- validating that model with observations
- refining it through iteration

What does not matter (as much as it seems)

Some aspects are often overemphasized:

- tools and frameworks
- isolated metrics without context
- synthetic or unrealistic test scenarios
- micro-optimizations without system-level impact
- single test results taken in isolation

These elements can be useful, but they are not sufficient.

Common misconceptions

Several misconceptions frequently appear:

- “If I run a load test, I understand the system”
- “If CPU is low, the system is healthy”
- “If average latency is acceptable, the system is fine”
- “More hardware will solve the problem”

These assumptions often lead to incorrect conclusions.

System-level thinking

Performance emerges from interactions:

- between components
- between workload and resources
- between concurrency and queueing

Focusing on a single part of the system is rarely enough.

Understanding requires a global view.

Practical implication

Effective performance engineering requires:

- asking the right questions
- validating assumptions
- correlating multiple signals
- iterating based on evidence

Tools, tests, and metrics support this process, but do not replace it.

Key idea

Performance engineering is not about collecting data.

It is about understanding what the data means.

The goal is not to produce numbers, but to explain system behavior and make informed decisions.

◀ 1.2 – Core metrics and formulas | ▲ Index | 01-04-types-of-performance-tests ▶

1.4 – Types of performance tests

This chapter introduces the main categories of performance tests used in performance engineering.

Each type of performance test answers a different question about system behavior under load.

Together, they help evaluate performance, stability, scalability, recovery, and capacity in a controlled and measurable way.

Table of Contents

1.4.1 Purpose of performance testing

Definition

Performance testing evaluates how a system behaves under controlled workload conditions.

It provides measurable data about:

- latency
- throughput
- error rate
- resource usage

(→ 1.2 Core metrics and formulas)

Performance testing is therefore not only a measurement activity, but also a validation activity.

It is used to compare expected behavior with observed behavior under defined workload conditions.

Role in performance engineering

Performance testing is not only about measuring results.

It is used to:

- validate system behavior under expected conditions
- reveal bottlenecks and limitations
- support capacity planning
- validate architectural decisions

It also provides a controlled framework for comparing:

- versions of the same system
- different configurations
- infrastructure changes
- tuning choices

Without controlled testing, performance discussions often remain based on assumptions rather than evidence.

Workload as a model

A test workload represents a simplified model of real usage.

It defines:

- arrival rate (requests per second)
- concurrency (number of active users or requests)
- request patterns (distribution, mix of operations)

(→ 1.2.1 Little's Law (system-level concurrency))

A workload is not the real production environment itself.

It is a practical approximation of the most relevant usage patterns.

For this reason, the value of a performance test depends strongly on how realistic the workload model is.

Controlled conditions

A performance test is meaningful only if the execution conditions are understood.

This includes:

- the shape of the workload
- the duration of the test
- the environment in which it runs
- the metrics collected during execution

If these conditions are unclear, results may still produce numbers, but those numbers are difficult to interpret and compare.

Controlled conditions are what transform a test from a simple exercise into a useful engineering activity.

Relationship with the rest of the guide

Performance testing is the practical entry point for many of the concepts developed in the rest of the guide.

It exposes:

- queueing and saturation effects (→ 1.5 System behavior under load)
- concurrency limits (→ 1.6 Concurrency and parallelism)
- runtime and memory effects (→ 1.7 Runtime and memory model)
- resource saturation (→ 1.8 Resource-level performance)

For that reason, test design should always be connected to system reasoning.

Practical meaning

A good performance test does not only answer:

- "How fast is the system?"

It also helps answer:

- "Under which conditions does the system remain stable?"
- "What changes as load increases?"
- "Which limit is reached first?"
- "What kind of degradation appears?"

These questions are essential in performance engineering because they connect measurement to interpretation.

Key idea

Performance tests are controlled experiments.

They are designed to observe system behavior under specific workload conditions.

Their value lies not only in the measurements they produce, but also in the understanding they provide.

1.4.2 Load testing

Definition

Load testing evaluates system behavior under expected or typical workload.

It is the most common and most direct way to validate that a system behaves acceptably under normal operating conditions.

Objective

- verify that the system meets performance requirements
- validate latency and throughput targets
- observe resource usage under normal conditions

Load testing answers the question of whether the system behaves correctly in the operating range it is expected to support.

Characteristics

- workload is stable and controlled
- system operates within its expected range
- focus is on steady-state behavior

The purpose is not to break the system, but to establish whether the system performs correctly under the load it was designed for.

Example

A system designed for:

- 200 requests per second
- p95 latency < 300 ms

A load test verifies that these targets are met.

It may also verify that:

- error rate remains low
- throughput remains stable
- resource utilization remains within acceptable bounds

Diagnostic value

Load testing provides a baseline:

- normal latency distribution
- typical resource utilization
- expected throughput

This baseline is essential for comparison with other tests.

Without a reliable baseline, it is difficult to determine whether behavior observed in stress, spike, soak, or capacity tests is abnormal or simply normal for the system.

Limits of load testing

Load testing alone does not determine:

- the maximum system capacity
- the failure point of the system
- the long-term stability of the runtime
- the recovery behavior after abrupt changes in load

A system may pass a load test and still fail under overload, sustained execution, or rapid bursts of traffic.

For this reason, load testing is necessary but not sufficient.

Practical interpretation

Load testing is the reference point for the rest of performance analysis.

It defines the system's normal operating behavior and allows later tests to be interpreted in context.

If the system already behaves poorly under expected load, there is little value in moving immediately to more advanced test types.

Key idea

Load testing answers: *"Does the system behave correctly under expected load?"*

It establishes the baseline against which all other performance tests can be interpreted.

1.4.3 Stress testing

Definition

Stress testing evaluates system behavior beyond its expected capacity.

It is used to observe what happens when the system is pushed outside its intended operating range.

Objective

- identify system limits
- observe behavior under overload
- detect failure modes

Stress testing is primarily concerned with limit behavior and degradation under excessive demand.

Characteristics

- workload increases beyond normal levels
- system approaches or reaches saturation

(→ 1.8 Resource-level performance)

The overload may be applied progressively or maintained at a clearly excessive level.

In both cases, the goal is to expose how the system behaves when demand exceeds capacity.

Observable effects

- latency increases rapidly
- throughput plateaus or decreases

- error rate increases

(→ 1.5.3 Non-linear degradation)
(→ 1.5.4 Throughput collapse)

Additional effects may include:

- queue buildup
- timeout amplification
- pool exhaustion
- unstable resource usage
- retry-driven overload

Diagnostic value

Stress testing reveals:

- bottlenecks
- saturation points
- system stability under pressure

It is particularly useful for understanding whether degradation is gradual, abrupt, recoverable, or unstable.

Two systems with similar load-test results may behave very differently under stress.

Failure behavior

An important aspect of stress testing is not only whether the system fails, but how it fails.

Relevant questions include:

- Does latency increase before errors appear?
- Do errors appear gradually or suddenly?
- Does throughput flatten before it collapses?
- Does the system recover when load is reduced?

These questions matter operationally because overload is a realistic scenario in production systems.

Distinction from capacity testing

Stress testing and capacity testing are related, but they are not identical.

- **stress testing** focuses on overload behavior and failure modes
- **capacity testing** focuses on the maximum sustainable load that still meets requirements

Stress testing therefore continues beyond the acceptable operating range in order to examine degradation and failure.

Practical interpretation

Stress testing is useful when the engineering question is not only:

- “How much load can the system support?”

but also:

- “What happens after it can no longer support the load?”
- “Does it degrade gracefully?”
- “Can it recover cleanly?”

These are essential questions for resilience and operational robustness.

Key idea

Stress testing answers: *"What happens when the system is pushed beyond its limits?"*

It reveals how the system degrades, how it fails, and how much overload it can tolerate before becoming unstable.

1.4.4 Spike testing

Definition

Spike testing evaluates system behavior under sudden increases in load.

Unlike load testing or gradual stress testing, spike testing focuses on rapid transitions rather than stable operating conditions.

Objective

- observe reaction to abrupt workload changes
- evaluate elasticity and recovery
- detect transient instability

Spike testing is especially relevant for systems exposed to burst traffic, campaign peaks, event-driven demand, or short-lived surges in activity.

Characteristics

- workload increases rapidly in a short time
- system must adapt quickly

The defining characteristic is not only the volume of load, but the speed at which the load changes.

A system may handle a high load when it is reached gradually, but behave poorly when the same load arrives suddenly.

Observable effects

- temporary latency spikes
- queue buildup
- potential errors during transition

(→ 1.5 System behavior under load)

Additional effects may include:

- delayed scaling response
- transient connection exhaustion
- temporary timeout cascades
- slow recovery after the burst

Diagnostic value

Spike testing reveals:

- sensitivity to burst traffic
- queueing behavior under sudden load
- recovery capability after the spike

It is valuable because many systems are optimized for steady-state conditions but remain fragile during abrupt transitions.

Recovery behavior

The most important part of spike testing is often what happens after the spike.

Relevant questions include:

- Does the system return quickly to normal latency?
- Do queues drain in a controlled way?
- Are resources released correctly?
- Does the system remain degraded after the spike has passed?

A system that survives the spike but recovers slowly may still be operationally weak.

Practical interpretation

Spike testing is especially useful for systems that are:

- externally exposed to bursty traffic
- dependent on auto-scaling or elastic behavior
- sensitive to queue buildup
- subject to event-driven demand changes

In these cases, average load is often less important than short-term peaks and the system's reaction to them.

Key idea

Spike testing answers: *"How does the system react to sudden load changes?"*

It evaluates not only resistance to bursts, but also the ability to recover cleanly after them.

1.4.5 Soak testing

Definition

Soak testing evaluates system behavior over an extended period under sustained load.

It is sometimes also called endurance testing.

Its purpose is to expose problems that do not appear in short-duration tests.

Objective

- detect long-term issues
- observe stability over time
- identify gradual degradation

Soak testing is less concerned with peak performance and more concerned with consistency, accumulation, and drift.

Characteristics

- workload is constant or slowly varying
- test duration is long (hours or days)

The key dimension is time.

Some systems behave correctly for minutes but degrade after hours because of accumulation effects.

Observable effects

- memory growth
- resource leaks
- performance degradation over time

(→ 1.7 Runtime and memory model)

Additional long-duration symptoms may include:

- thread accumulation
- connection leakage
- slowly increasing queues
- GC overhead growth
- cache imbalance or uncontrolled retention

Diagnostic value

Soak testing reveals:

- memory leaks
- resource exhaustion
- long-term instability

It is often the only reliable way to validate whether the system remains healthy during prolonged activity.

This is essential for production systems expected to run continuously.

Time-dependent degradation

Soak testing is important because some failures are not threshold-based, but time-based.

Examples include:

- memory retained slowly over time
- pools not fully released
- background tasks accumulating drift
- retry patterns slowly increasing pressure
- caches growing without effective eviction

These issues may not appear in load or stress tests of short duration.

Operational value

A system that performs well for ten minutes but degrades after six hours is not stable.

Soak testing therefore contributes directly to:

- production readiness
- runtime confidence
- long-term reliability assessment
- infrastructure and runtime dimensioning

It also helps validate that monitoring remains meaningful over long periods of operation.

Practical interpretation

Soak testing is particularly important for systems with:

- long uptimes
- background processing
- memory-managed runtimes
- connection-heavy architectures
- resource pools that change slowly over time

In such systems, short-duration performance results are not sufficient to guarantee real stability.

Key idea

Soak testing answers: *"Does the system remain stable over time?"*

It validates long-duration behavior and reveals issues caused by accumulation, drift, and slow degradation.

1.4.6 Capacity testing

Definition

Capacity testing determines the maximum workload a system can handle while meeting performance requirements.

It is used to identify the practical operating limit of the system under acceptable conditions.

Objective

- identify maximum sustainable throughput
- determine safe operating limits
- support capacity planning

Capacity testing is therefore directly related to planning, sizing, forecasting, and operational decision-making.

Method

- gradually increase workload
- monitor latency, throughput, and errors
- identify the point where performance degrades

The increase in load should be controlled and measurable.

This allows the system limit to be located more precisely than in a purely exploratory stress test.

Interpretation

The capacity limit is reached when:

- latency exceeds acceptable thresholds
- error rate increases
- throughput no longer scales

(→ 1.2 Core metrics and formulas)
(→ 1.5 System behavior under load)

In practice, the limit is not always a single exact value.

It may be better understood as a range in which acceptable behavior begins to deteriorate.

What capacity testing reveals

Capacity testing reveals:

- the highest sustainable load under defined acceptance criteria
- the margin between expected load and maximum acceptable load
- the relationship between increasing demand and degrading behavior
- the point at which additional load no longer produces useful throughput

This information is essential for engineering and planning decisions.

Relationship with capacity planning

Capacity testing is one of the main inputs to capacity planning.

It helps answer questions such as:

- How much traffic can the current system support?
- How much headroom is available?
- When will scaling be required?
- Which component constrains capacity first?

This makes capacity testing especially useful for forecasting and operational preparation.

Distinction from stress testing

Capacity testing is not about forcing failure for its own sake.

It is about identifying the highest load that still satisfies defined requirements.

- **capacity testing** stops at or near the acceptable limit
- **stress testing** continues beyond that limit to examine overload behavior

The distinction matters because many business and engineering decisions depend on safe operation, not on total failure.

Practical meaning

Capacity is not only a number.

It depends on:

- workload mix
- concurrency level
- latency objectives
- acceptable error rate
- resource constraints

For this reason, any capacity number must always be interpreted in the context of the workload and acceptance criteria used during the test.

Practical interpretation

Capacity testing is most useful when the engineering objective is to answer:

- “What is the safe operating range?”
- “How much headroom do we have?”
- “When do we need to scale?”
- “What constrains future growth?”

It is therefore one of the most decision-oriented forms of performance testing.

Key idea

Capacity testing answers: *"How far can the system scale before it degrades?"*

It identifies the maximum sustainable operating range, not just the point of failure.

◀ 1.3 – Work of a performance engineer | ▲ Index | 1.5 – System behavior under load ▶

1.5 – System behavior under load

This chapter explains how systems behave as workload increases and capacity limits are approached.

It focuses on the main mechanisms that cause degradation under load, including **saturation**, **queueing**, **throughput loss**, and **tail latency amplification**.

These concepts are central to performance engineering because they explain why systems often appear stable at low load and become unstable near their limits.

Table of Contents

1.5.1 Load vs capacity

Definition

A system operates under load, but it has a finite capacity.

- **Load**: the amount of work applied to the system (e.g. requests per second, concurrent users)
- **Capacity**: the maximum amount of work the system can handle while remaining stable

Understanding the relationship between load and capacity is fundamental to performance engineering.

It defines the operating envelope of the system and determines when behavior is predictable and when degradation begins.

System behavior

At low load:

- resources are underutilized
- response time is stable
- throughput increases linearly with load

As load increases:

- resource utilization grows
- contention begins to appear
- response time increases

When load approaches capacity:

- queues form
- latency increases rapidly
- system behavior becomes less predictable

This transition is one of the most important aspects of performance analysis.

A system rarely moves directly from “healthy” to “failed.”
It usually passes through a region of increasing instability and reduced efficiency.

Capacity is not a fixed number

Capacity is often misunderstood as a single value.

In reality, it depends on:

- workload composition (use cases and distribution)
- resource configuration (CPU, memory, pools)
- system state (cold vs warm, cache effects)
- external dependencies (databases, services)

A system may handle:

- 100 req/s for simple requests
- but only 20 req/s for complex ones

Capacity is therefore always contextual.

It must be understood in relation to a specific workload, environment, and acceptance criteria.

Effective capacity

Capacity must be defined under constraints.

Typical criteria:

- latency within acceptable limits (e.g. p95)
- error rate below threshold
- stable resource usage

The maximum load that satisfies these conditions is the **effective capacity**.

This is the capacity that matters operationally.

A theoretical maximum that produces unacceptable latency or instability is not useful in practice.

Practical implication

Capacity cannot be assumed.

It must be:

- measured under realistic workload
- validated through testing
- monitored over time

Increasing load beyond effective capacity leads to:

- rapid degradation
- unstable behavior
- potential system failure

It may also reduce the system's ability to recover quickly after overload.

Link with previous concepts

The relationship between load, latency, and concurrency is formalized by:

→ 1.2.1 Little's Law

As load increases:

- concurrency increases
- waiting time grows
- response time degrades

This relationship is one of the foundations of understanding behavior under load.

Practical interpretation

Load and capacity should never be treated as abstract labels.

They determine:

- whether the system operates with headroom
- whether queueing is likely to appear
- how much margin exists before instability begins

In performance engineering, knowing that a system “works” is not enough.

What matters is knowing under which load conditions it remains stable and how close it is to its effective capacity.

Key idea

A system does not fail when it reaches capacity.

It starts to degrade before that point.

The goal of performance engineering is to identify:

- where capacity lies
- how the system behaves near it
- how much margin is required

1.5.2 Saturation and queueing

Definition

Saturation occurs when a resource is busy most or all of the time.

Queueing occurs when incoming work cannot be processed immediately and must wait.

These two phenomena are tightly connected.

They are among the most important mechanisms behind performance degradation in real systems.

Resource saturation

A resource becomes saturated when:

- its utilization approaches its limit
- it has little or no idle time

Typical examples:

- CPU close to 100%
- thread pool fully occupied
- connection pool exhausted

At this point:

- new requests cannot be processed immediately
- they must wait

Saturation does not necessarily mean total failure.

It means the system has lost processing headroom and is no longer able to absorb additional work without delay.

Queue formation

When requests arrive faster than they can be processed:

- a queue forms
- waiting time increases

This affects response time:

- service time remains the same
- waiting time grows

→ 1.2.3 Service time vs response time

Queueing is therefore the visible consequence of insufficient processing capacity at a given resource.

Non-linear effect

Queueing does not grow linearly.

As utilization increases:

- waiting time grows slowly at first
- then increases rapidly
- eventually dominates response time

Small increases in load can cause large increases in latency.

This is why systems often appear stable for a long time and then degrade suddenly near saturation.

Link with utilization

Utilization plays a central role:

→ 1.2.2 Utilization Law

As utilization approaches its limit:

- the probability of waiting increases
- queues grow
- latency becomes unstable

The important point is not only that a resource is “busy,” but that once it becomes continuously busy, incoming work begins to accumulate.

Practical implications

Queueing is often the main cause of performance degradation.

Symptoms include:

- sudden increase in response time
- long-tail latency (p95, p99)
- growing queues (threads, connections, requests)

Even if:

- CPU is not fully saturated
- average latency seems acceptable

Queueing may still be the dominant source of delay.

This is especially common in systems with shared pools, blocking operations, or dependency bottlenecks.

Example

A system handles requests with:

- service time = 10 ms

At low load:

- requests are processed immediately
- response time ≈ 10 ms

As load increases:

- requests start waiting
- response time becomes:
  10 ms (service) + waiting time

At high load:

- waiting time dominates
- response time increases rapidly

This example illustrates why latency growth under load is often caused more by waiting than by work itself.

Practical interpretation

Saturation is the condition.

Queueing is the consequence.

The system does not slow down primarily because each request requires more computation, but because more requests are competing for the same limited resources.

This distinction is essential:

- optimizing service time may help
- but reducing queueing is often even more important

Key idea

Saturation does not immediately break the system.

It introduces queueing.

Queueing increases waiting time.

Waiting time dominates response time.

This is the primary mechanism behind performance degradation under load.

1.5.3 Non-linear degradation

Definition

System performance does not degrade linearly as load increases.

Instead, degradation follows a non-linear pattern, especially near capacity limits.

This means that the relationship between load and response time is often smooth at first and then sharply unstable near saturation.

Linear vs non-linear behavior

At low to moderate load:

- throughput increases proportionally with load
- latency remains relatively stable

In this region, the system appears predictable.

As load approaches capacity:

- small increases in load produce large increases in latency
- variability increases
- behavior becomes unstable

This marks the transition to non-linear degradation.

The system no longer behaves proportionally to demand.

It begins to react disproportionately to additional work.

Root cause

Non-linear degradation is primarily caused by:

- queueing effects (→ 1.5.2 Saturation and queueing)
- high resource utilization
- contention between requests

As utilization increases:

- waiting time grows disproportionately
- response time becomes dominated by delays rather than service

This explains why degradation often accelerates suddenly rather than gradually.

Observable effects

Typical symptoms include:

- rapid increase in p95 and p99 latency
- widening gap between average and tail latency
- increased variance in response times
- intermittent errors or timeouts

These effects often appear suddenly.

The system may seem healthy just before entering a region of severe instability.

Misleading intuition

It is common to assume:

- “If the system handles 80 req/s, it should handle 100 req/s with slightly higher latency”

In reality:

- performance may remain stable up to a point
- then degrade sharply beyond that point

There is often no gradual transition.

This is one of the most common mistakes in capacity planning and performance expectations.

Example

A system behaves as follows:

- up to 70 req/s → stable latency (~100 ms)
- at 80 req/s → latency increases to 150 ms
- at 90 req/s → latency jumps to 400 ms
- at 100 req/s → system becomes unstable

The degradation is not proportional to load.

The last increments in load have a much larger effect than the earlier ones.

Practical implication

Capacity planning must consider non-linear behavior.

Operating a system near its limits leads to:

- unpredictable latency
- unstable performance
- poor user experience

Systems should operate with a margin below capacity.

That margin is not optional.

It is what allows the system to absorb normal variability without entering unstable behavior.

Link with previous concepts

Non-linear degradation is the visible effect of:

- increasing utilization (→ 1.2.2 Utilization Law)
- growing queueing (→ 1.5.2 Saturation and queueing)

It is therefore a system-level consequence of mechanisms already introduced in the previous sections.

Practical interpretation

Non-linear degradation explains why systems should not be operated too close to their theoretical maximum.

A small operational margin can be the difference between:

- stable performance
- unpredictable degradation

This is also why average resource usage alone is often misleading when assessing production safety.

Key idea

Performance degradation is not gradual.

It accelerates as the system approaches its limits.

Understanding this non-linearity is essential to avoid operating systems too close to capacity.

1.5.4 Throughput collapse

Definition

Throughput collapse occurs when increasing load no longer increases throughput, and may even reduce it.

Instead of scaling with demand, the system becomes less efficient as load grows.

This is one of the clearest signs that the system is operating beyond its effective capacity.

Expected behavior vs collapse

Under normal conditions:

- increasing load increases throughput
- until the system approaches capacity

However, beyond a certain point:

- throughput stops increasing
- may plateau or decrease
- latency increases significantly

This is throughput collapse.

More incoming work does not translate into more completed work.

Root causes

Throughput collapse is typically caused by:

- excessive queueing
- contention on shared resources
- resource thrashing (CPU, memory, I/O)
- retry amplification
- inefficient scheduling or locking

As the system becomes overloaded:

- more time is spent managing contention than doing useful work
- effective processing capacity decreases

This is the key reason why more demand can produce less output.

Queueing contribution

When queues grow:

- requests wait longer
- system resources remain occupied
- new requests add pressure without increasing completed work

Queueing can therefore:

- increase latency
- reduce effective throughput

This is especially visible when the system spends increasing time handling backlog rather than making forward progress.

Contention and thrashing

At high load:

- threads compete for shared resources
- locks become hotspots
- context switching increases
- cache locality degrades

In extreme cases:

- the system spends more time coordinating than processing

This leads to reduced throughput.

The system remains active, but its activity becomes increasingly unproductive.

Retry amplification

Failures under load often trigger retries.

This creates additional load:

- failed requests are retried
- more work is generated
- pressure increases further

This feedback loop can:

- accelerate collapse
- make recovery difficult

Retry behavior is therefore not only a symptom response, but also a frequent cause of worsening overload.

Observable effects

Typical symptoms include:

- throughput plateau or decrease despite increased load
- sharp increase in latency
- rising error rates (timeouts, 5xx)
- unstable or oscillating behavior

At this stage, the system may appear busy but is no longer scaling usefully.

Example

A system behaves as follows:

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

Increasing load reduces effective throughput.

This is a direct indicator that overload is harming useful work.

Practical implication

Throughput collapse indicates that the system is operating beyond its effective capacity.

At this point:

- adding more load worsens performance
- the system may become unstable

Mitigation requires:

- reducing load
- removing bottlenecks
- improving resource efficiency

In many cases, the first corrective action is not optimization but protection: rate limiting, admission control, or retry control.

Link with previous concepts

Throughput collapse is the result of:

- non-linear degradation (→ 3.5.3 Non-linear degradation)
- saturation and queueing (→ 3.5.2 Saturation and queueing)

It can therefore be understood as an advanced stage of overload behavior.

Practical interpretation

A system does not always process more work when more work is applied.

At some point, additional work becomes destructive rather than productive.

Recognizing this transition is essential in performance engineering, because it marks the difference between heavy load and overload.

Key idea

A system does not always process more work when more work is applied.

Beyond a certain point, additional load reduces the system's ability to process requests.

Understanding throughput collapse is essential to avoid overload conditions.

1.5.5 Tail latency amplification

Definition

Tail latency amplification refers to the disproportionate increase of high-percentile response times (e.g. p95, p99) under load.

While average latency may appear acceptable, a subset of requests becomes significantly slower.

This effect is one of the most important indicators of degraded user experience and hidden instability.

Percentiles vs average

Average latency hides variability.

Percentiles reveal distribution:

- p50 represents the typical request
- p95 and p99 represent the slowest requests

Under load:

- average latency may increase moderately

- tail latency can increase dramatically

→ 1.2.7 Percentiles

For this reason, averages alone are not sufficient to assess real performance quality.

Root causes

Tail latency amplification is primarily driven by:

- queueing delays
- contention on shared resources
- uneven workload distribution
- dependency variability (e.g. database, external services)

Even small delays in some components can:

- propagate through the system
- amplify end-to-end latency

Tail latency is therefore often an emergent effect, not just a local one.

Distributed systems effect

In systems with multiple components:

- a request often depends on several services
- overall latency depends on the slowest component

As the number of dependencies increases:

- the probability of a slow request increases
- tail latency becomes more pronounced

This is one of the reasons why tail latency is especially important in distributed architectures.

Under load

As load increases:

- queues grow
- contention increases
- variability expands

This leads to:

- a widening gap between average and p95/p99
- unpredictable response times for a subset of users

The system may therefore appear mostly stable while still producing unacceptable experience for a meaningful fraction of requests.

Observable effects

Typical symptoms include:

- stable average latency with degraded p95/p99
- intermittent slow responses
- timeouts affecting only a fraction of requests

This can be misleading:

- the system appears “mostly fine”

- but user experience is degraded

This is why tail metrics are essential in performance testing and production monitoring.

Example

A system shows:

- average latency = 120 ms
- p95 latency = 180 ms (acceptable)
- p99 latency = 1200 ms (problematic)

Most requests are fast, but a small percentage is very slow.

In many user-facing systems, this small percentage is enough to create visible dissatisfaction or SLO violations.

Practical implication

Performance evaluation must consider **tail latency**.

Relying on averages can:

- hide critical issues
- underestimate user impact

Systems should be designed and tested to:

- control tail behavior
- limit variability under load

This is particularly important for distributed systems, APIs, and interactive applications.

Link with previous concepts

Tail latency amplification is a consequence of:

- queueing (→ 1.5.2 Saturation and queueing)
- non-linear degradation (→ 1.5.3 Non-linear degradation)
- system interactions and dependencies

It is therefore one of the most visible manifestations of system stress under load.

Practical interpretation

Performance is not defined by the average request.

It is defined by the predictability of response times, especially for the slowest requests.

A system with acceptable average latency but poor p95/p99 behavior is not truly stable from a user or operational perspective.

Key idea

Performance is not defined by the average request.

It is defined by how the system behaves for the slowest requests.

Controlling tail latency is essential for predictable and reliable systems.

1.6 – Concurrency and parallelism

This chapter introduces concurrency and parallelism as core concepts in system performance engineering.

It explains how work is scheduled, how multiple tasks interact, and why coordination overhead, contention, and synchronization often become limiting factors under load.

Concurrency and parallelism are essential to scalability, but they also introduce complexity, overhead, and failure modes that directly affect latency, throughput, and system stability.

Table of Contents

1.6.1 Concurrency vs parallelism

Definition

Concurrency and **parallelism** are related but distinct concepts.

They are often confused, but they describe different aspects of system behavior.

Understanding the distinction is essential because a system may handle many tasks at once from a structural point of view without actually executing many tasks simultaneously at the hardware level.

Concurrency

Concurrency refers to the ability of a system to handle multiple tasks during the same time interval.

These tasks:

- may not run at the exact same time
- can be interleaved
- share system resources

Concurrency is about:

- structure
- coordination
- managing multiple in-flight operations

It is therefore primarily concerned with how work is organized and scheduled.

Parallelism

Parallelism refers to the execution of multiple tasks at the same time.

This requires:

- multiple processing units (e.g. CPU cores)
- true simultaneous execution

Parallelism is about:

- execution
- hardware utilization
- doing more work at the same instant

It is therefore primarily concerned with actual simultaneous progress.

Key difference

- **Concurrency** = dealing with many tasks
- **Parallelism** = executing many tasks simultaneously

A system can be:

- concurrent but not parallel (single core, interleaving tasks)
- parallel but not highly concurrent (few long-running tasks)

This distinction matters because the scalability properties of a system depend not only on how much work exists, but also on how that work is coordinated and scheduled.

Relationship with performance

Concurrency affects:

- how many requests can be in progress
- how resources are shared
- how contention arises

Parallelism affects:

- how fast work can be executed
- how well hardware is utilized

Both influence:

- throughput
- latency
- scalability

In practice, adding concurrency without sufficient parallelism may increase waiting and contention, while adding parallelism without good concurrency control may waste resources or expose coordination problems.

Practical intuition

A concurrent system:

- can accept many requests
- may still process them sequentially or with limited parallelism

A parallel system:

- can process multiple requests at the same time
- but may still suffer from contention or coordination overhead

For this reason, concurrency and parallelism should not be treated as automatically beneficial.

Their value depends on how they interact with workload, shared resources, and execution constraints.

Link with previous concepts

Concurrency increases:

- the number of in-flight requests (→ 1.2.1 Little's Law)

This leads to:

- resource sharing
- potential queueing (→ 1.5.2 Saturation and queueing)

This is one of the main reasons concurrency becomes a central topic in performance engineering rather than only a programming concern.

Practical interpretation

Concurrency is often required to support many simultaneous operations, especially in networked and I/O-driven systems.

However, concurrency also increases the probability of:

- shared state interactions
- queue buildup
- lock contention
- coordination overhead

Parallelism may increase throughput, but only if useful work is actually being performed rather than blocked or serialized.

Key idea

Concurrency determines how many tasks are active.

Parallelism determines how many tasks are executed at the same time.

Performance depends on both, and on how they interact with system resources.

1.6.2 Threads and execution model

Definition

The **execution model** defines how work is executed within a system.

In most systems, work is performed by **threads**, which run within a **process**.

The execution model determines how requests are mapped to execution units, how waiting is handled, and how system resources are consumed under load.

Processes and threads

A **process** is an isolated execution environment:

- it has its own memory space
- it contains resources (files, sockets, memory)

A **thread** is a unit of execution within a process:

- multiple threads share the same process memory
- threads execute tasks concurrently

In most applications:

- one process hosts multiple threads

- threads handle incoming requests

This shared-memory model makes threads efficient for communication, but also introduces shared-state complexity.

Threads

A thread:

- executes instructions
- consumes CPU time
- may block while waiting (e.g. I/O, locks)

Multiple threads allow a system to:

- handle multiple requests
- overlap computation and waiting
- increase concurrency

However, threads are not free.

Each additional thread adds memory overhead, scheduling overhead, and coordination complexity.

Thread lifecycle

A thread typically goes through several states:

- **running** (actively executing)
- **runnable** (ready to run, waiting for CPU)
- **waiting** / blocked (waiting for a resource or event)

Performance is affected by how threads move between these states.

A system with many runnable or blocked threads may appear active, but still make limited useful progress.

Understanding thread states is therefore essential when diagnosing concurrency issues.

Stack and memory

Each thread has its own **stack**:

- stores method calls and local variables
- grows and shrinks during execution

Implications:

- more threads → more memory usage (one stack per thread)
- deep call chains → larger stack usage
- stack exhaustion can lead to failures

This is particularly relevant in high-concurrency systems.

Thread count therefore affects not only scheduling, but also memory footprint and stability.

Execution models

Different systems use different **execution models**.

Common models include:

One thread per request

Each request is handled by a dedicated thread.

Characteristics:

- simple model
- easy to reason about
- blocking operations are straightforward

Limitations:

- high memory usage with many threads
- limited scalability under high concurrency

This model is conceptually simple, but it often performs poorly when concurrency becomes very large or when blocking is frequent.

Thread pool

A fixed number of threads handle incoming requests.

Requests are queued and assigned to available threads.

Characteristics:

- controlled concurrency
- reduced overhead compared to unbounded threads

Limitations:

- queueing when all threads are busy
- potential saturation of the pool

This model is widely used because it provides controlled resource usage, but it introduces an explicit queue and therefore a visible capacity limit.

Event-driven / asynchronous model

Work is handled using **non-blocking** operations and **event loops**.

Characteristics:

- few threads can handle many concurrent requests
- efficient for I/O-bound workloads

Limitations:

- more complex programming model
- requires careful handling of asynchronous flows

This model reduces the number of blocked threads, but it shifts complexity into coordination, callbacks, state handling, and non-blocking design.

Java perspective (example)

In Java, a common execution model uses thread pools.

For example:

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

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

Requests are:

- submitted to a queue
- executed by a limited number of threads

If all threads are busy:

- tasks wait in the queue
- latency increases

For a detailed explanation of threads in Java, see:

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

This example is simple, but it highlights a key idea: bounded execution resources naturally introduce queueing when demand exceeds immediate processing capacity.

Blocking vs non-blocking

Threads may:

- **block** (wait for I/O, locks, external resources)
- **remain active** (CPU-bound work)

Blocking reduces effective concurrency:

- threads are occupied but not progressing
- fewer threads are available for new work

Non-blocking approaches aim to:

- reduce idle waiting
- improve resource utilization

The distinction is important because high thread count does not necessarily mean high throughput.

If threads spend most of their time waiting, concurrency is present, but productive execution is limited.

Practical implications

The execution model determines:

- how concurrency is handled
- how resources are used
- how queueing appears

Typical effects include:

- thread pool saturation → request queueing
- blocking operations → reduced throughput
- too many threads → context switching overhead

The execution model also determines where bottlenecks become visible: in queues, in pools, in blocked threads, or in event loops.

Link with previous concepts

Thread behavior directly impacts:

- queueing (→ 1.5.2 Saturation and queueing)
- latency under load
- effective capacity of the system

It also influences how quickly a system moves from stable behavior to saturation when concurrency increases.

Practical interpretation

Choosing an execution model is not only a programming decision.

It is a performance decision.

The model affects:

- memory consumption
- scheduling overhead
- latency under waiting conditions
- scalability under real workload

A design that is easy to implement may not be the design that behaves best under sustained load.

Key idea

The execution model defines how work is scheduled and processed.

Threads are not free.

How they are used determines:

- how much work can be handled
- how efficiently resources are utilized
- how the system behaves under load

1.6.3 Contention and synchronization

Definition

Contention occurs when multiple threads compete for the same resource.

Synchronization is the mechanism used to coordinate access to shared resources.

These concepts are central to understanding performance degradation in concurrent systems.

They connect correctness and performance: the same mechanisms that protect shared state can also become the source of waiting and reduced scalability.

Shared resources

In concurrent systems, threads often share resources such as:

- memory structures (objects, caches)
- locks and monitors
- thread pools and queues
- database connections
- I/O channels

When access is not coordinated, data **corruption** may occur.

When access is coordinated, **contention** may appear.

This makes synchronization necessary, but not free.

Synchronization

Synchronization ensures that shared resources are accessed safely.

Common mechanisms include:

- locks (mutexes, monitors)
- synchronized sections
- semaphores
- atomic operations

Synchronization guarantees correctness, but introduces overhead.

That overhead may come from:

- waiting
- serialization of execution
- additional memory barriers
- coordination costs between threads

Contention

Contention arises when multiple threads attempt to access the same resource simultaneously.

When contention occurs:

- threads may block or wait
- execution is delayed
- throughput is reduced

The more threads compete:

- the higher the waiting time
- the lower the effective parallelism

A highly concurrent system can therefore behave like a partially serialized system if too much of its work depends on the same shared resources.

Lock contention

A common form of contention involves locks.

When a thread holds a lock:

- other threads must wait
- a queue of waiting threads may form

Effects include:

- increased latency
- reduced throughput
- potential bottlenecks

Lock contention is especially problematic when critical sections are long, frequently accessed, or placed on hot execution paths.

Contention vs utilization

High contention can occur even when CPU utilization is moderate.

For example:

- many threads are waiting on a lock
- CPU is partially idle
- system appears underutilized but is actually constrained

This is a common source of misleading diagnostics.

It explains why low or moderate CPU usage does not necessarily mean that the system has available capacity.

Fine-grained vs coarse-grained synchronization

Synchronization can be:

- **coarse-grained** (few locks, large critical sections)
- **fine-grained** (many locks, smaller critical sections)

Trade-offs:

- **coarse-grained** → simpler but higher contention
- **fine-grained** → more scalable but more complex

Choosing between them depends on workload characteristics, access patterns, and the cost of added design complexity.

Java perspective (example)

In Java, synchronization may be implemented using `synchronized` blocks:

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

Or explicit locks:

```
Lock lock = new ReentrantLock();

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

If many threads attempt to enter the same critical section:

- contention increases
- threads block
- performance degrades

This example highlights how a correctness mechanism can become a scalability constraint under load.

Symptoms of contention

Typical indicators include:

- increasing response time under load
- low CPU utilization with high latency
- threads in blocked or waiting states
- long queues on shared resources

These symptoms often appear before total saturation and may be mistaken for other resource problems if not analyzed carefully.

Practical implications

Contention limits scalability.

Even with:

- sufficient CPU
- adequate memory

A system may not scale if:

- threads spend time waiting instead of executing

Reducing contention often has a larger impact than optimizing individual operations.

This is especially true for systems where performance is constrained by shared access rather than by raw computation.

Link with previous concepts

Contention contributes to:

- queueing (→ 1.5.2 Saturation and queueing)
- non-linear degradation (→ 1.5.3 Non-linear degradation)
- throughput collapse (→ 1.5.4 Throughput collapse)

Contention is therefore both a local synchronization phenomenon and a system-level performance mechanism.

Practical interpretation

Concurrency increases opportunities for useful overlap, but it also increases competition for shared resources.

The practical challenge is not simply to add more threads, but to ensure that additional concurrency results in useful work rather than additional waiting.

Key idea

Concurrency introduces the need for synchronization.

Synchronization introduces contention.

Contention limits performance.

Understanding and controlling contention is essential for scalable systems.

1.6.4 Common concurrency issues

Concurrency introduces complexity.

When multiple threads interact, incorrect assumptions or poor coordination can lead to specific classes of problems.

These issues often appear under load and can severely impact performance and correctness.

Many of them are difficult to reproduce in light testing because they depend on timing, scheduling, or resource pressure.

1.6.4.1 Race conditions

Definition

A **race condition** occurs when multiple threads access shared data without proper synchronization, and the result depends on timing.

The outcome is therefore not deterministic and may vary from one execution to another.

Example

Two threads update a shared counter:

- Thread A reads value = 10
- Thread B reads value = 10
- Thread A writes 11
- Thread B writes 11

Expected result: 12
Actual result: 11

The final value depends on the order in which unsynchronized operations happen to execute.

Impact

- incorrect results
- inconsistent system state
- difficult-to-reproduce bugs

Race conditions may also corrupt internal assumptions in ways that only appear later under load.

Performance relevance

Race conditions may not always cause visible failures, but:

- they often require additional synchronization
- improper fixes can introduce contention

This is one reason correctness and performance cannot be treated as completely separate concerns in concurrent systems.

1.6.4.2 Deadlocks

Definition

A **deadlock** occurs when two or more threads wait indefinitely for each other.

Each thread holds a resource and waits for another resource held by another thread.

As a result, progress stops completely.

Example

- Thread A holds lock L1 and waits for L2
- Thread B holds lock L2 and waits for L1

Neither can proceed.

This circular waiting pattern is the defining characteristic of deadlock.

Impact

- system stalls
- requests never complete
- resources remain locked

Deadlocks are especially severe because they convert active resources into permanently blocked ones.

Detection

- threads remain blocked
- thread dumps show circular waiting

Deadlocks are often detected through thread analysis rather than through general performance metrics alone.

1.6.4.3 Livelocks

Definition

A **livelock** occurs when threads are not blocked but continuously change state in response to each other without making progress.

Unlike deadlock, activity continues, but useful work does not.

Example

Two threads repeatedly retry an operation:

- both detect conflict
- both retry at the same time
- conflict persists

The system remains active, but the conflicting behavior continues indefinitely.

Impact

- CPU is used
- no useful work is completed

Livelocks may therefore look like active processing even though progress is effectively zero.

1.6.4.4 Starvation

Definition

Starvation occurs when some threads are unable to obtain resources for a prolonged period.

Other threads continue to execute while some are effectively ignored.

This means the system is making progress, but not fairly or predictably for all work.

Causes

- unfair scheduling
- high-priority threads dominating execution

- resource monopolization

Starvation is especially problematic when a subset of requests experiences extreme latency while the rest of the system appears functional.

Impact

- some requests experience very high latency
- system appears partially functional
- tail latency increases

This makes starvation particularly relevant from both a performance and user-experience perspective.

1.6.4.5 Thread pool exhaustion

Definition

Thread pool exhaustion occurs when all threads in a pool are busy and incoming tasks must wait.

This is one of the most common concurrency-related bottlenecks in real systems.

Causes

- blocking operations within threads
- insufficient pool size
- long-running tasks

These causes may exist independently or reinforce each other under increasing load.

Effects

- request queue grows
- latency increases
- throughput may degrade

If saturation continues, thread pool exhaustion may also contribute to timeouts, retries, and instability in upstream components.

Link with previous concepts

Thread pool exhaustion is a direct example of:

- saturation (→ 1.5.2 Saturation and queueing)
- non-linear degradation (→ 1.5.3 Non-linear degradation)

It is therefore one of the clearest practical expressions of the system behaviors introduced in the previous chapter.

Key idea

Concurrency issues are not only correctness problems.

They are also performance problems.

Many performance degradations are caused by:

- contention
- blocking

- coordination failures

Understanding these issues is essential for diagnosing real-world systems.

◀ 1.5 – System behavior under load | ▲ Index | 01-07-runtime-and-memory-model ▶

1.7 – Runtime and memory model

This chapter explains how managed runtimes organize memory, allocate objects, reclaim unused memory, and behave under memory pressure.

It focuses on the runtime and memory mechanisms that directly affect latency, stability, and throughput under load.

Understanding these mechanisms is essential because many performance problems are not caused only by CPU or I/O limits, but by the way memory is allocated, retained, and reclaimed over time.

Table of Contents

1.7.1 Memory structure (heap, stack)

Memory management models

Different systems use different memory management strategies.

Two common approaches are:

- **manual memory management**
  Memory is explicitly allocated and freed by the programmer (e.g. C, C++)
- **managed memory**
  Memory is allocated automatically and reclaimed by the runtime (e.g. Java, .NET)

This guide focuses on **managed memory systems**, where:

- objects are allocated dynamically
- memory is reclaimed automatically (garbage collection)

This distinction matters because performance behavior changes significantly depending on whether memory lifecycle is controlled directly by the programmer or indirectly by the runtime.

Definition

Memory is organized into different regions with distinct roles.

The two most important areas for performance reasoning are:

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

These two regions support different aspects of program execution and have very different performance implications.

Heap

The heap is a shared memory area used for dynamic allocation.

In managed runtimes (such as Java):

- objects are allocated on the heap
- memory is managed by the runtime
- garbage collection reclaims unused objects

Implications:

- memory usage grows with allocation rate
- garbage collection impacts performance
- shared access may introduce contention

The heap is therefore not only a storage area, but a central part of runtime behavior under load.

Stack

Each thread has its own stack.

The stack stores:

- method calls (call frames)
- local variables
- intermediate values

Characteristics:

- private to each thread
- grows and shrinks during execution
- typically much smaller than the heap

Because the stack is private to the thread, access is simple and efficient, but the number of threads directly affects total stack memory usage.

Heap vs stack

Aspect	Heap	Stack
Scope	Shared across threads	Private per thread
Allocation	Dynamic (objects)	Automatic (method calls)
Lifetime	Managed by runtime	Tied to method execution
Performance	More complex	Very fast
Memory impact	Global	Per-thread

Interaction with threads

Each thread:

- has its own stack
- shares the heap

This creates a model where:

- execution is isolated per thread (stack)
- data is shared across threads (heap)

This interaction is a source of:

- contention (shared objects)
- coordination overhead

It also explains why concurrency and memory behavior are tightly linked in managed systems.

Performance implications

Heap:

- excessive allocation → increased GC activity
- large heap → longer garbage collection cycles
- shared access → potential contention

Stack:

- many threads → higher total memory usage (one stack per thread)
- deep call chains → increased stack usage
- stack overflow → failure in extreme cases

These implications become especially important when the system is under sustained or high-concurrency load.

Practical interpretation

Heap and stack are not just implementation details.

They affect:

- how data is shared
- how work is executed
- how memory grows under concurrency
- where runtime overhead appears

A system with many threads and frequent allocation will stress both regions differently: the stack through thread count and call depth, the heap through object creation and retention.

Key idea

The heap stores shared data.

The stack supports execution.

Performance depends on how these two interact under load.

Link with previous concepts

Memory behavior directly impacts:

- thread execution (→ 1.6.2 Threads and execution model)
- contention (→ 1.6.3 Contention and synchronization)
- latency under load (→ 1.5 System behavior under load)

This is why runtime and memory model cannot be analyzed separately from concurrency and system behavior.

1.7.2 Allocation and object lifecycle

Definition

In managed memory systems, objects are created dynamically and live for a certain period of time before being reclaimed.

The way objects are allocated and how long they live has a direct impact on performance.

Allocation behavior is therefore not only a memory concern, but also a latency and stability concern.

Allocation

Allocation is the process of creating new objects in memory.

In most managed runtimes:

- allocation happens on the heap
- it is designed to be fast and efficient
- it occurs very frequently in typical applications

Examples of allocation:

- creating request objects
- building data structures
- processing intermediate results

In high-throughput systems, allocation is often continuous and closely tied to workload intensity.

Allocation rate

The **allocation rate** is the amount of memory allocated per unit of time.

It is a key performance factor.

High allocation rate means:

- more objects created
- increased memory churn
- increased pressure on the runtime

Even if individual allocations are fast, large volumes impact the system.

This is one of the reasons why “fast allocation” does not automatically mean “low memory overhead.”

Object lifecycle

Objects do not all live for the same duration.

Typical categories include:

- **short-lived objects**
  created and discarded quickly (e.g. temporary request data)
- **medium-lived objects**
  survive for some time during processing
- **long-lived objects**
  remain in memory for extended periods (e.g. caches, shared state)

Understanding object lifetime is essential for reasoning about memory behavior.

It determines how much memory remains live over time and how the runtime must organize reclamation work.

Allocation patterns

Real systems tend to exhibit patterns such as:

- many short-lived objects per request
- occasional long-lived objects
- bursts of allocation under load

These patterns determine:

- memory usage
- garbage collection behavior
- performance stability

Allocation patterns are often more informative than isolated allocation events, because the runtime reacts to aggregate behavior over time.

Impact on performance

Allocation itself is usually fast.

The main impact comes from:

- increased memory usage
- pressure on garbage collection

High allocation rate can lead to:

- more frequent garbage collection cycles
- increased latency
- unpredictable pauses

The important point is that memory cost is often indirect: the system pays not only for creating objects, but for managing the consequences of creating many of them.

Under load

As load increases:

- more requests are processed
- more objects are created
- allocation rate increases

This amplifies:

- memory pressure
- garbage collection activity
- latency variability

A system that is stable at low load may therefore become memory-sensitive as request volume rises, even if the logic of each request remains unchanged.

Interaction with concurrency

Allocation is often performed by multiple threads.

This can lead to:

- contention on memory structures
- increased coordination overhead
- uneven memory usage patterns

In high-concurrency systems:

- allocation rate grows with concurrency
- memory becomes a shared bottleneck

This is one of the ways in which concurrency and memory behavior reinforce each other under load.

Practical implications

To reason about performance, it is important to consider:

- how many objects are created per request
- how long they live
- how allocation rate changes under load

Understanding allocation is essential to:

- explain latency behavior
- identify bottlenecks
- predict system limits

It also helps distinguish between problems caused by computation and problems caused by memory churn.

Practical interpretation

Allocation is often invisible at the code level because it is easy to write and usually inexpensive per operation.

However, at the system level, repeated allocation changes the runtime's workload.

A design that creates large numbers of temporary objects may work correctly, but still impose significant pressure on the memory subsystem.

Link with next concepts

Allocation and object lifetime directly influence:

- garbage collection behavior (→ next section)
- memory pressure
- latency under load

They therefore form the causal basis for the runtime effects described in the rest of this chapter.

Key idea

Performance depends on how much memory is allocated and how long it is retained.

Allocation patterns shape system behavior under load.

1.7.3 Garbage collection (conceptual)

Definition

Garbage collection (GC) is the process by which a managed runtime reclaims memory that is no longer in use.

Instead of requiring explicit deallocation, the runtime:

- identifies unused objects
- frees their memory
- makes space available for new allocations

Garbage collection is one of the defining mechanisms of managed runtimes and one of the main ways memory behavior becomes visible in performance analysis.

Basic principle

An object is eligible for collection when it is no longer reachable.

This means:

- no active reference points to it
- it cannot be accessed by the program

The runtime periodically:

- scans object references
- identifies unreachable objects
- reclaims their memory

This model allows memory to be managed automatically, but it also means that reclamation work must be performed during program execution.

Allocation and reclamation cycle

Memory usage follows a cycle:

1. objects are allocated
2. objects become unused
3. garbage collection reclaims memory

This cycle repeats continuously during execution.

The runtime therefore alternates between allocating new memory and reclaiming old memory, with overall behavior driven by allocation rate and retention patterns.

Java perspective (example)

In Java, object allocation is frequent and inexpensive.

For example:

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

This code creates a large number of short-lived objects.

In a managed runtime:

- these objects are allocated quickly on the heap
- they become unreachable shortly after creation
- garbage collection reclaims them

If such allocation patterns occur under load:

- GC activity increases
- memory pressure grows
- latency may become unstable

The impact depends not on a single allocation, but on the **allocation rate over time**.

This is why memory behavior should be analyzed as a pattern, not as an isolated operation.

Example: object retention

Objects that remain referenced are not collected.

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

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

In this case:

- objects are continuously allocated
- they are never released
- memory usage grows over time

This leads to:

- increased memory pressure
- more expensive garbage collection
- potential system instability

This example illustrates the difference between temporary allocation churn and persistent retention.

Cost of garbage collection

Garbage collection is not free.

It introduces overhead:

- CPU time to analyze memory
- pauses during collection (depending on strategy)

The cost depends on:

- allocation rate
- number of live objects
- memory size

In other words, GC cost depends not only on how much memory exists, but on how much memory is active, changing, and still reachable.

Stop-the-world effect

Some garbage collection phases may pause application execution.

During these pauses:

- threads are temporarily stopped
- no application work is performed

Even short pauses can:

- increase latency
- affect tail response times (p95, p99)

This is one of the reasons GC issues often appear first in percentile-based latency analysis rather than in averages alone.

Generational behavior (conceptual)

Most modern runtimes use a generational approach.

Based on observation:

- most objects are short-lived

- few objects live for a long time

Memory is organized so that:

- short-lived objects are collected frequently
- long-lived objects are collected less often

This improves efficiency because reclaiming many short-lived objects is usually cheaper than repeatedly scanning long-lived memory.

Under load

As load increases:

- allocation rate increases
- garbage collection runs more frequently

This can lead to:

- higher CPU usage
- more frequent pauses
- increased latency variability

Under heavy load, GC may therefore shift from being a background maintenance mechanism to being a visible part of the system's performance behavior.

Interaction with object lifecycle

Garbage collection behavior depends on:

- how many objects are created
- how long they live

Typical patterns:

- many short-lived objects → frequent collections
- many long-lived objects → heavier collections

This is why allocation and retention must be analyzed together: object count alone is not enough.

Observable effects

Garbage collection issues often appear as:

- latency spikes
- long-tail latency (p95/p99 degradation)
- periodic pauses
- increased CPU usage without clear cause

These symptoms are often intermittent, which makes GC-related problems difficult to diagnose without correlating memory and latency signals.

Practical implications

Performance analysis must consider:

- allocation rate
- object lifetime distribution
- frequency and cost of GC cycles

Optimization typically focuses on:

- understanding allocation patterns
- reducing unnecessary object creation
- controlling memory pressure

Tuning the collector may help, but it is usually more effective to first understand why the runtime is under pressure.

Practical interpretation

Garbage collection is not a bug or an anomaly.

It is a necessary runtime mechanism.

The performance question is not whether GC exists, but whether its cost remains compatible with the workload and latency objectives of the system.

Link with previous concepts

Garbage collection is directly linked to:

- allocation (→ 1.7.2 Allocation and object lifecycle)
- memory structure (→ 1.7.1 Memory structure)
- tail latency (→ 1.5.5 Tail latency amplification)

It is therefore both a runtime mechanism and a system-level contributor to performance variability.

Key idea

Garbage collection enables automatic memory management but introduces variability.

Performance depends on how efficiently memory is reclaimed.

1.7.4 Memory pressure and performance

Definition

Memory pressure refers to the stress placed on the memory system when allocation, retention, and reclamation interact under load.

It is not only about how much memory is used, but how memory behaves over time.

Memory pressure is therefore a dynamic condition, not simply a static measure of heap occupancy.

What creates memory pressure

Memory pressure is driven by a combination of factors:

- high allocation rate
- large number of live objects
- long object lifetimes
- inefficient memory reclamation

These factors reinforce each other and determine how much work the runtime must perform to keep memory usable.

Allocation vs retention

Two different patterns can create pressure:

- **high allocation rate**
  many objects are created and quickly discarded
- **high retention**
  objects remain in memory for long periods

These patterns create pressure in different ways.

High allocation increases churn and collection frequency.

High retention increases the amount of memory that remains live and must be scanned or preserved.

Example: high allocation rate

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

Characteristics:

- many short-lived objects
- frequent allocation
- frequent garbage collection

Effects:

- increased GC activity
- CPU overhead
- potential latency spikes

This example highlights pressure driven by churn rather than by long-term retention.

Example: memory retention

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

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

Characteristics:

- objects are retained
- memory usage continuously grows

Effects:

- increasing heap usage
- heavier garbage collection cycles
- eventual instability or failure

This example highlights pressure driven by retained memory rather than temporary allocation frequency alone.

Under load

As system load increases:

- more requests are processed
- more objects are created
- more objects are retained

This leads to:

- increased allocation rate
- increased memory usage
- increased GC activity

Memory pressure amplifies:

- latency variability
- tail latency

This is why memory-related degradation often becomes more visible as the system moves from moderate load to sustained high load.

Interaction with garbage collection

Garbage collection responds to memory pressure.

Under pressure:

- collections become more frequent
- pauses may increase
- CPU usage grows

In extreme cases:

- GC dominates execution
- useful work decreases

When this happens, the runtime is spending a significant share of its effort managing memory instead of processing application work.

Observable symptoms

Memory pressure often appears as:

- latency spikes without clear CPU bottleneck
- long-tail latency degradation (p95, p99)
- periodic pauses
- increased GC frequency
- growing memory usage over time

These symptoms are especially important because they can be mistaken for generic slowness unless memory behavior is examined directly.

Practical intuition

A system may appear:

- lightly loaded (moderate CPU)
- but still slow

This often indicates:

- memory pressure
- GC-related overhead

This is one of the main reasons why CPU alone is not sufficient to assess system health.

Simplified model

System behavior can be approximated as:

- allocation rate ↑ → GC activity ↑

- retention ↑ → memory usage ↑
- GC activity ↑ → latency variability ↑

These relationships are not linear.

They depend on runtime strategy, workload shape, object lifetimes, and the amount of live data.

Practical implications

To manage memory pressure:

- understand allocation patterns
- identify long-lived objects
- monitor GC behavior
- correlate memory metrics with latency

Optimization should focus on:

- reducing unnecessary allocation
- controlling object lifetime
- avoiding unbounded retention

In many cases, the most effective fix is not collector tuning, but reducing the memory work the runtime is forced to perform.

Link with previous concepts

Memory pressure contributes to:

- non-linear degradation (→ 1.5.3 Non-linear degradation)
- throughput collapse (→ 1.5.4 Throughput collapse)
- tail latency amplification (→ 1.5.5 Tail latency amplification)

It is therefore a direct bridge between runtime internals and visible system behavior under load.

Practical interpretation

Memory pressure explains why a system may degrade even when it is not obviously CPU-bound or externally blocked.

A runtime under memory stress can still appear active, but produce increasing latency, reduced throughput, and unstable behavior.

This makes memory pressure one of the most important hidden causes of performance degradation in managed runtimes.

Key idea

Memory pressure results from the interaction between allocation, retention, and garbage collection under load.

Understanding this interaction is essential to explain latency and stability issues in real systems.

◀ 1.6 – Concurrency and parallelism | ▲ Index | 01-08-resource-level-performance ▶

1.8 – Resource-level performance

This chapter explains how core system resources behave under load and how they constrain performance.

It focuses on CPU, I/O, network behavior, and the way bottlenecks emerge when one resource becomes saturated before the others.

Understanding resource-level performance is essential because system degradation is often the visible result of resource limits rather than application logic alone.

Table of Contents

1.8.1 CPU behavior

Definition

The **CPU** is responsible for executing instructions.

CPU performance is determined not only by how fast instructions are executed, but by how execution is scheduled across competing workloads.

This distinction is important because CPU-related degradation is often caused by scheduling pressure, queueing, and contention, not only by raw computational cost.

CPU utilization vs saturation

CPU utilization represents how much of the CPU capacity is being used.

High utilization does not necessarily mean a problem.

CPU saturation occurs when:

- there is more work than the CPU can execute
- threads are ready to run but cannot be scheduled immediately

Key distinction:

- **high utilization** → CPU is busy
- **saturation** → CPU is overloaded

A system may therefore show high CPU usage and still behave acceptably, as long as runnable work does not accumulate faster than the CPU can process it.

Scheduling and run queue

Threads do not execute continuously.

They are scheduled by the operating system.

At any moment:

- some threads are **running**
- some are **waiting** to run (run queue)

When the number of runnable threads exceeds available CPU cores:

- threads accumulate in the run queue
- scheduling delays increase

This directly impacts latency (→ 1.5 System behavior under load) and can be reasoned using concurrency relationships (→ 1.2.1 Little's Law (system-level concurrency)).

The run queue is therefore a critical signal of CPU pressure, because it shows not just that the CPU is busy, but that work is waiting to be executed.

Observable behavior (example)

A system under CPU pressure shows an increasing number of runnable threads.

```
$ 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
```

Interpretation:

- run queue (`r`) is high → threads waiting for CPU
- CPU idle (`id`) is close to zero → no available capacity
- CPU usage (`us + sy`) is near saturation

This indicates that threads are ready to execute but cannot be scheduled immediately (→ 1.6 Concurrency and parallelism).

The important point is that CPU saturation is not defined only by percentage values, but by the presence of runnable work that cannot make progress immediately.

Impact on performance

When CPU becomes saturated:

- scheduling delays increase
- response time increases
- throughput may plateau or decrease

This effect is non-linear (→ 1.5.3 Non-linear degradation).

As CPU saturation increases, the system may spend progressively more time waiting to be scheduled rather than performing useful work.

Interaction with concurrency

Concurrency increases the number of active threads.

As concurrency grows:

- more threads compete for CPU
- run queue length increases
- scheduling overhead increases

Beyond a certain point:

- adding threads reduces performance instead of improving it (→ 1.6 Concurrency and parallelism).

This is why adding more concurrent work does not always produce better throughput.

If CPU time becomes the limiting resource, concurrency turns into scheduling pressure.

Practical implications

To reason about CPU behavior:

- distinguish utilization from saturation
- observe runnable threads, not just %CPU

- correlate CPU metrics with latency (→ 1.2 Core metrics and formulas)

CPU issues are often not about raw usage, but about **contention for execution**.

It is therefore possible for a system to appear “fully busy” without being unstable, or to appear only moderately busy while already showing scheduling delays.

Practical interpretation

CPU analysis should focus on the ability of the system to keep up with runnable work.

A busy CPU is not automatically a problem.

A saturated CPU becomes a problem when runnable tasks accumulate, latency rises, and throughput no longer scales with incoming demand.

Key idea

CPU performance is limited by scheduling.

When threads cannot be scheduled immediately, latency increases even if the system appears fully utilized.

1.8.2 I/O and disk

Definition

I/O operations involve reading from or writing to storage devices.

Unlike CPU operations, I/O is typically slower and often blocking.

This means that many performance problems involving I/O are dominated by waiting time rather than by active computation.

Latency vs throughput

I/O performance has two key dimensions:

- **latency** → time to complete a single operation
- **throughput** → number of operations per unit of time

High throughput does not guarantee low latency.

A system may move a large amount of data overall while individual requests still experience significant wait times.

Blocking behavior

Many I/O operations are blocking:

- a thread initiates an operation
- it waits until completion

During this time:

- the thread is not executing useful work
- it may hold resources (locks, connections)

This is one of the main reasons why I/O bottlenecks often propagate into thread pool pressure, queueing, and reduced effective concurrency.

Queueing effects

When multiple requests perform I/O:

- operations queue at the device level
- waiting time increases

As queue length grows:

- latency increases
- variability increases (→ 1.5 System behavior under load)

This can be expressed as queueing delay (→ 1.2.3 Service time vs response time (queueing)).

The important point is that the cost of I/O is not limited to the duration of the operation itself.

It also includes the time spent waiting for previous operations to complete.

Observable behavior (example)

A system under I/O pressure shows increasing wait times.

```
$ iostat -x 1
Device            r/s     w/s   await   %util
sda              120      80     35.0    95.0
sda              130      90     42.0    98.0
```

Interpretation:

- high `await` → requests spend significant time waiting
- `%util` near 100% → device is saturated
- increasing latency indicates queue buildup

This reflects queueing effects (→ 1.2 Core metrics and formulas).

The increasing `await` value is especially important, because it often reveals that the device is not merely busy, but increasingly unable to absorb incoming work without additional delay.

Impact on performance

When I/O becomes a bottleneck:

- request latency increases
- throughput may degrade
- threads spend more time waiting than executing

This can reduce effective system capacity even when CPU usage remains moderate.

A system can therefore be I/O-bound without appearing CPU-bound.

Interaction with concurrency

More concurrent requests lead to:

- more I/O operations
- longer device queues
- increased latency

Increasing concurrency does not improve performance if the device is saturated (→ 1.6 Concurrency and parallelism).

Beyond a certain point, additional concurrency only increases waiting and worsens response time.

Practical implications

To reason about I/O behavior:

- focus on latency (`await`), not only throughput
- identify queue buildup
- correlate I/O wait with application latency (→ 1.5 System behavior under load)

I/O problems are often misunderstood because throughput may remain acceptable while latency degrades significantly.

Practical interpretation

I/O performance should be evaluated as a waiting system.

The core question is not only how many operations per second the device can support, but how long operations wait when the workload intensifies.

A storage subsystem that performs well at low concurrency may degrade sharply when requests begin to accumulate.

Key idea

I/O performance is dominated by waiting time.

As queues grow, latency increases and system responsiveness degrades.

1.8.3 Network behavior

Definition

Network performance is determined by the transfer of data between systems.

It depends on both latency and bandwidth.

In distributed systems, network behavior is often a major contributor to end-to-end response time, especially when requests traverse multiple services.

Latency and round trips

Network communication often requires multiple exchanges.

Each exchange introduces:

- transmission delay
- propagation delay
- processing delay

Multiple round trips amplify total latency (→ 1.5 System behavior under load).

This is especially important in request chains where each service call depends on the response of the previous one.

Even small delays can accumulate significantly across multiple network hops.

Bandwidth limitations

Bandwidth defines how much data can be transferred per unit of time.

When bandwidth is limited:

- large payloads take longer to transfer

- throughput becomes constrained

Bandwidth therefore matters most when the amount of transferred data becomes large enough to dominate communication time.

Latency, by contrast, matters even for small payloads when many round trips are required.

Amplification under load

As load increases:

- more requests are sent over the network
- contention increases
- queues may form in buffers

This leads to:

- increased latency
- packet delays or retransmissions (→ 1.5.5 Tail latency amplification)

Under load, network variability becomes especially important because occasional delays can affect only part of the traffic while still degrading overall user experience.

Observable behavior (example)

A system under network pressure shows connection and queue buildup.

```
$ 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
```

Interpretation:

- large number of established connections → high concurrency
- accumulation of connections may indicate slow processing or network delays

A growing number of open connections may indicate that requests are not completing quickly enough, either because downstream services are slow or because the system is unable to process network work efficiently.

Impact on performance

Network constraints lead to:

- increased response time
- higher variability
- cascading delays across services

In distributed architectures, these delays often propagate and amplify because one slow network interaction can delay many dependent operations.

Interaction with system design

Distributed systems amplify network effects:

- multiple services introduce multiple network hops
- latency accumulates across calls (→ 1.5 System behavior under load)

A system with many service boundaries may therefore suffer from network-induced latency even when each individual call appears relatively inexpensive.

Practical implications

To reason about network behavior:

- consider number of round trips
- observe connection patterns
- correlate network activity with latency

It is also important to distinguish between:

- bandwidth-limited behavior
- latency-limited behavior
- dependency-induced delay

These are related but not identical problems.

Practical interpretation

Network performance is not only about how fast bytes move.

It is also about how often systems communicate, how many dependencies are involved, and how delays in one component affect others.

In many service architectures, reducing unnecessary round trips can improve latency more effectively than simply increasing bandwidth.

Key idea

Network performance is driven by latency and communication patterns.

Under load, small delays accumulate and significantly impact response time.

1.8.4 Resource saturation and bottlenecks

Definition

A **bottleneck** is the resource that limits system performance.

Saturation occurs when that resource operates at or near its capacity.

This is the point where additional workload no longer translates into proportional useful throughput.

Identifying the limiting resource

At any moment, system performance is constrained by one dominant resource:

- CPU
- I/O
- network
- memory (indirectly through GC → 1.7 Runtime and memory model)

Identifying this resource is essential.

Without identifying the actual limiting resource, optimization efforts often target symptoms rather than causes.

Single bottleneck principle

Even in complex systems:

- performance is typically limited by one primary constraint

Improving non-limiting resources has little effect.

This principle is one of the reasons why performance engineering must remain system-oriented.

Many resources may appear active, but only one usually determines the current capacity limit.

Cascading effects

When a resource becomes saturated:

- queues build up
- latency increases
- upstream components slow down

This can propagate through the system (→ 1.5 System behavior under load).

A local bottleneck can therefore become a system-wide problem as delays spread to callers, workers, pools, and dependent services.

Interaction between resources

Resources are not independent:

- slow I/O increases thread wait time → affects CPU scheduling (→ 1.8.1 CPU behavior)
- network delays increase request lifetime → increases memory usage (→ 1.7 Runtime and memory model)
- CPU saturation delays processing → increases queue sizes (→ 1.2.1 Little's Law (system-level concurrency))

This interaction explains why bottlenecks often move or appear coupled under changing workload conditions.

The limiting factor may shift as one part of the system is improved or as workload composition changes.

Observable patterns

Common signs of bottlenecks:

- CPU near saturation with high run queue
- I/O latency increasing with high device utilization
- network delays with growing connection counts

These patterns are useful because they connect system-level symptoms with specific resource behaviors.

They help reduce diagnostic ambiguity.

Impact on system behavior

When a bottleneck is reached:

- throughput stops increasing
- latency grows rapidly
- system becomes unstable under further load

This corresponds to:

- non-linear degradation (→ 1.5.3 Non-linear degradation)

- throughput collapse (→ 1.5.4 Throughput collapse)

At this stage, additional demand often worsens the situation rather than increasing useful output.

Practical implications

To analyze performance:

- identify the saturated resource
- correlate resource metrics with latency
- focus optimization on the limiting factor

A correct diagnosis therefore depends on understanding not just which resources are busy, but which one is currently controlling system behavior.

Practical interpretation

Bottleneck analysis is the bridge between observation and action.

The purpose is not merely to collect CPU, I/O, or network metrics, but to determine which resource is constraining useful work at the current operating point.

Once that resource is identified, optimization becomes meaningful.

Key idea

System performance is limited by its bottleneck.

Understanding which resource is saturated is essential to explain and improve behavior under load.

◀ 01-07-runtime-and-memory-model | ▲ Index | 01-09-common-performance-problems ▶

1.9 – Common performance problems

This chapter describes common performance problems that appear in real systems under load.

These problems are not isolated categories. They often interact, reinforce each other, and become visible as latency growth, throughput loss, instability, or tail degradation.

The purpose of this chapter is to connect recurring symptoms to the underlying mechanisms already introduced in the previous chapters.

Table of Contents

1.9.1 CPU-bound inefficiency

Definition

A CPU-bound inefficiency occurs when the system spends excessive CPU time performing work that could be reduced, optimized, or avoided.

This does not necessarily mean that the system is CPU-saturated at all times.

It means that available CPU time is being consumed inefficiently, reducing the amount of useful work the system can perform before reaching saturation.

Typical causes

- inefficient algorithms (e.g. unnecessary complexity)
- repeated computations
- lack of caching for expensive operations
- excessive data transformations

These causes are common because CPU inefficiency often emerges from code that is functionally correct but structurally wasteful.

In performance engineering, inefficiency matters most when it occurs in hot paths or highly repeated operations.

Example

```java
public int countMatches(List<String> items, String target) {
    int count = 0;
    for (String s : items) {
        if (s.toLowerCase().equals(target.toLowerCase())) {
            count++;
        }
    }
    return count;
}
```

Interpretation:

- repeated `toLowerCase()` calls create unnecessary work
- CPU time increases with input size
- avoidable computation in hot paths

The problem is not only the cost of the loop itself, but the repeated transformation of values that could be normalized once instead of at every comparison.

Mechanism

CPU-bound inefficiency wastes execution capacity.

More CPU time is consumed than necessary to produce the same result.

As the workload grows:

- CPU utilization rises earlier
- runnable work accumulates sooner
- useful throughput reaches its limit earlier

This transforms inefficient code into a system-level bottleneck when request volume increases.

Impact under load

- increased CPU utilization
- reduced throughput
- earlier CPU saturation

This leads to scheduling delays (→ 1.8.1 CPU behavior) and non-linear latency growth (→ 1.5.3 Non-linear degradation).

In practical terms, the system reaches its CPU limit sooner than expected, leaving less headroom for bursts or concurrent traffic growth.

Observable symptoms

Typical symptoms include:

- high CPU usage under moderate load
- rising latency with increasing request volume
- throughput flattening earlier than expected
- significant CPU time spent in repeated or avoidable operations

These symptoms often appear before total CPU saturation and may initially look like a generic scaling problem.

Practical implications

- optimize hot paths
- avoid repeated work
- reduce algorithmic complexity

It is also important to identify which inefficiencies actually matter at system level.

An inefficient operation executed once may be irrelevant.

The same inefficiency executed millions of times becomes a bottleneck.

Practical interpretation

CPU inefficiency is one of the most common reasons a system fails to scale despite apparently sufficient hardware.

The issue is not lack of CPU in absolute terms, but poor use of the CPU that is available.

Optimization is therefore most valuable when it increases the amount of useful work performed per unit of CPU time.

Key idea

CPU inefficiency reduces the amount of useful work the system can perform before reaching saturation.

1.9.2 Excessive allocation and memory churn

Definition

Excessive allocation occurs when the system creates a large number of short-lived objects, increasing memory churn and pressure on the runtime.

This is a common problem in managed runtimes, where allocation is easy and often inexpensive per operation, but expensive in aggregate when performed continuously under load.

Example

```
for (Order o : orders) {
    result.add(new ReportRow(o.getId(), o.getAmount(), o.getStatus()));
}
```

Interpretation:

- many objects are created per iteration
- objects are short-lived
- allocation rate increases

If this pattern appears in frequently executed code, total allocation volume can become significant even when each individual object is small.

Mechanism

- high allocation rate increases memory churn
- garbage collection runs more frequently

(→ 1.7.2 Allocation and object lifecycle)
(→ 1.7.3 Garbage collection)

The system therefore pays not only for creating objects, but for reclaiming them, tracking them, and managing the runtime effects of frequent memory turnover.

Impact under load

- increased GC activity
- CPU overhead for memory management
- latency variability

This contributes to memory pressure (→ 1.7.4 Memory pressure and performance).

As load increases, allocation-related overhead often becomes more visible through pauses, jitter, and widening latency percentiles.

Observable symptoms

Typical symptoms include:

- increased garbage collection frequency
- periodic latency spikes
- growing gap between average and tail latency
- moderate CPU usage with unstable response times
- memory behavior that degrades as throughput increases

These symptoms are especially common in systems that allocate heavily in request-processing paths.

Practical implications

- reduce unnecessary object creation
- reuse objects when appropriate
- analyze allocation patterns

It is also important to distinguish between:

- necessary allocation
- avoidable allocation

- retained allocation that should have remained temporary

This distinction helps determine whether the issue is churn, retention, or both.

Practical interpretation

Excessive allocation is often invisible in code review because the code remains simple and correct.

Its effect becomes visible only at runtime, when repeated object creation changes GC behavior and memory pressure.

A system may therefore appear logically efficient while still behaving poorly because it creates too much transient memory traffic.

Key idea

Memory churn increases runtime overhead and introduces latency variability.

1.9.3 Contention and synchronization hot spots

Definition

Contention occurs when multiple threads compete for the same resource, forcing serialized access.

A synchronization hot spot is a part of the system where this competition becomes concentrated and repeatedly delays execution.

These hot spots are especially damaging because they reduce effective parallelism exactly where concurrency is expected to help.

Example

```
public class Counter {
    private int value = 0;

    public synchronized void increment() {
        value++;
    }
}
```

Interpretation:

- access is serialized through synchronization
- only one thread progresses at a time
- throughput is limited by the critical section

The issue is not that synchronization exists, but that a frequently accessed shared path can become the limiting point for the whole system.

Mechanism

- threads block while waiting for the lock
- contention increases with concurrency

(→ 1.6 Concurrency and parallelism)

As more threads compete for the same synchronized section:

- waiting time grows
- effective parallelism decreases

- more time is spent coordinating than progressing

This causes the system to behave as if its concurrency were lower than its thread count suggests.

Impact under load

- increased waiting time
- reduced throughput
- latency increases

This leads to queueing effects (→ 1.5 System behavior under load).

Under higher load, synchronization hot spots often become visible as latency growth without proportional CPU growth, because threads are waiting rather than computing.

Observable symptoms

Typical symptoms include:

- rising latency with moderate CPU usage
- many threads blocked or waiting
- reduced scalability as concurrency increases
- throughput limited by a small critical section
- lock-heavy code paths appearing on hot execution paths

These symptoms are often misleading because the system may appear only partially utilized while already constrained.

Practical implications

- minimize shared mutable state
- reduce critical section size
- use more scalable concurrency patterns

It is also important to identify whether the bottleneck is caused by:

- lock scope
- frequency of access
- long critical sections
- unnecessary synchronization

Different causes require different fixes.

Practical interpretation

Contention problems are often misunderstood as generic slowness.

In reality, the core issue is serialization: many threads are present, but only a few are making useful progress.

Performance engineering therefore focuses not only on adding concurrency, but on making sure concurrency does not collapse into waiting.

Key idea

Contention converts parallel work into serialized execution.

1.9.4 Blocking and waiting bottlenecks

Definition

Blocking occurs when a thread waits for an external operation to complete, preventing it from doing useful work.

This includes waiting for:

- I/O
- network responses
- locks
- external services
- other coordinated events

Blocking is often necessary, but it becomes a bottleneck when too many execution resources are occupied by waiting rather than progressing.

Example

```
public String fetchData() throws Exception {
    Thread.sleep(50); // simulate blocking call
    return "data";
}
```

Interpretation:

- thread is idle during wait
- resources remain allocated
- concurrency does not translate to throughput

The thread exists, but it is not advancing useful work during the blocked period.

Mechanism

- threads spend time waiting instead of executing
- thread pools may become saturated

(→ 1.6 Concurrency and parallelism)

As more threads become blocked:

- fewer threads remain available for new work
- queueing appears at the execution model level
- latency grows even if the CPU is not fully used

This is why blocking bottlenecks often coexist with moderate CPU usage.

Impact under load

- increased latency
- reduced throughput
- thread exhaustion

This amplifies queueing and saturation (→ 1.5 System behavior under load).

Under sustained load, blocking behavior often creates a feedback loop where queued requests wait for threads that are themselves waiting on slow operations.

Observable symptoms

Typical symptoms include:

- many threads in waiting or blocked states
- growing request queues
- moderate CPU with poor throughput
- rising latency during I/O-heavy or dependency-heavy operations
- thread pools that appear full without corresponding productive work

These symptoms are especially common in services that mix request concurrency with synchronous downstream calls.

Practical implications

- reduce blocking operations
- use asynchronous or non-blocking patterns when appropriate
- size thread pools carefully

It is also useful to distinguish between:

- unavoidable blocking
- avoidable blocking
- blocking placed in high-frequency execution paths

That distinction helps identify where redesign is necessary.

Practical interpretation

Blocking reduces effective concurrency.

A system may have many threads, but if a large share of them is waiting, the system behaves as if it had much less execution capacity than expected.

This is why blocking issues are often execution-model problems before they become raw resource problems.

Key idea

Blocking reduces effective concurrency and limits system throughput.

1.9.5 Queue buildup and saturation effects

Definition

Queue buildup occurs when incoming work exceeds processing capacity, causing requests to wait before being processed.

This is one of the most common and most important performance problems because queueing transforms moderate overload into rapidly increasing latency.

Mechanism

- arrival rate exceeds service capacity
- queues grow over time

This can be described using Little's Law (→ 1.2.1 Little's Law (system-level concurrency)).

As incoming demand continues while processing remains limited, waiting accumulates and response time begins to include increasingly large queue delay.

Impact under load

- waiting time increases
- response time increases
- latency becomes unstable

This leads to non-linear degradation (→ 1.5.3 Non-linear degradation) and throughput limits.

Once queueing becomes dominant, the system can deteriorate very quickly even if the original increase in load was relatively small.

Observable symptoms

- growing queue lengths
- increasing response times
- stable or decreasing throughput

Other symptoms may include:

- bursts of timeout errors
- widening p95/p99 latency
- delayed recovery after temporary overload

These effects often indicate that the system is operating near or beyond effective capacity.

Practical implications

- control concurrency
- increase capacity of the bottleneck resource
- reduce arrival rate if necessary

It is also important to determine where the queue is forming:

- thread pool
- connection pool
- device
- network buffer
- downstream service

The location of the queue often reveals the actual bottleneck.

Practical interpretation

Queue buildup is not just an operational detail.

It is often the direct mechanism through which overload becomes visible to users.

A system may still be functioning, but once work begins to wait systematically, latency growth becomes inevitable.

Key idea

Queues grow when demand exceeds capacity, driving latency.

1.9.6 Dependency amplification and cascading latency

Definition

Dependency amplification occurs when latency in one component propagates and increases latency across the system.

This problem is especially important in distributed systems, where a request often depends on multiple downstream calls before it can complete.

Mechanism

- requests depend on multiple downstream services
- delays accumulate across calls
- slow components affect the entire system

Even when each individual delay is small, the total effect can become significant once multiple dependencies, retries, or serial call chains are involved.

Example

```
public Response process() {
    Data a = serviceA.call();
    Data b = serviceB.call();
    return combine(a, b);
}
```

Interpretation:

- total latency depends on multiple dependencies
- slowest dependency dominates response time

In real systems, this effect becomes stronger when requests depend on many services, remote databases, or chained synchronous operations.

Impact under load

- latency amplification across services
- increased variability
- tail latency degradation

(→ 1.5.5 Tail latency amplification)

Under load, dependency amplification often becomes more severe because slow downstream systems retain upstream threads, requests, and queues for longer periods.

Observable symptoms

Typical symptoms include:

- sudden latency increases without local CPU saturation
- degraded p95/p99 behavior caused by downstream variability
- request chains that become slower as one dependency slows down
- instability spreading from one service to another
- retries and timeouts increasing pressure across the system

These symptoms are often difficult to interpret without correlating behavior across multiple components.

Practical implications

- minimize number of synchronous dependencies
- use timeouts and fallback strategies
- isolate slow components

It is also useful to identify:

- which dependency contributes most to end-to-end delay
- whether calls are serial or parallel
- whether retries worsen the problem
- whether slow components trigger upstream queueing

This turns a vague “distributed slowness” problem into a diagnosable system behavior.

Practical interpretation

A system’s latency is not determined only by its own code.

It is often determined by the slowest dependency in the request path.

The more dependencies a system has, the more likely it is that variability in one place will become visible everywhere else.

Key idea

System latency is often determined by the slowest dependency.

◀ 01-08-resource-level-performance | ▲ Index | 01-10-diagnostics-and-analysis ▶

1.10 – Diagnostics and analysis

This chapter explains how performance problems are investigated, interpreted, and validated.

It focuses on the reasoning process used to move from observed behavior to a defensible explanation of system performance under load.

Diagnostics is not only a matter of collecting data.
It is the discipline of interpreting that data correctly and connecting symptoms to mechanisms.

Table of Contents

1.10.1 Observability and signals

Definition

Diagnostics starts from observable signals.

These signals provide indirect visibility into the internal behavior of the system under load.
They do not expose mechanisms directly, but they reflect their effects.

This is why observability is essential in performance engineering: internal problems are rarely visible directly, but they usually leave measurable traces in latency, throughput, resource behavior, and queueing.

Core signals

The primary signals are:

- latency (p50, p95, p99)
- throughput
- error rate
- resource utilization (CPU, memory, I/O, network)
- queue lengths

(→ 1.2 Core metrics and formulas)
(→ 1.8 Resource-level performance)

Each signal captures a different dimension of system behavior.
Only their combination provides a meaningful view.

Latency shows user-visible impact.
Throughput shows productive work.
Error rate shows failure behavior.
Resource metrics show where capacity is being consumed.
Queues show where work is accumulating.

Signal characteristics

Signals must be:

- **accurate** → reflect real behavior
- **granular** → expose distribution (e.g. percentiles, not only averages)
- **correlated in time** → aligned across components

Without these properties, interpretation becomes unreliable or misleading.

A metric that is delayed, aggregated too heavily, or disconnected from the relevant time window may hide the very mechanism it is supposed to reveal.

Signal quality and interpretation

The presence of signals is not sufficient by itself.

Signals must also be:

- relevant to the question being asked
- observed at the correct level (system, service, resource, dependency)
- interpreted in context

For example:

- CPU usage without run queue information may hide scheduling pressure
- average latency without percentiles may hide tail instability
- memory usage without GC behavior may hide runtime pressure

The diagnostic value of a metric depends not only on its existence, but on how it is combined with other evidence.

Practical implications

Effective diagnostics requires:

- observing multiple signals together
- correlating them over time
- avoiding single-metric reasoning

Looking at one metric in isolation often hides the underlying mechanism.

This is one of the main reasons why simplistic explanations are dangerous in performance analysis.

A single number may describe a symptom, but it rarely explains system behavior.

Practical interpretation

Observability is the raw material of diagnostics.

Without signals, there is no reliable analysis.
With poor signals, there is unreliable analysis.
With well-structured signals, analysis becomes testable and repeatable.

Diagnostics therefore begins not with optimization, but with visibility.

Key idea

Diagnostics depends on both the availability and the correct interpretation of observable signals.

1.10.2 Symptom vs cause

Definition

A symptom is an observable effect.

A cause is the underlying mechanism that produces that effect.

This distinction is fundamental because most performance problems are first seen through symptoms, not through direct visibility into the root cause.

Distinction

Typical symptoms:

- high latency
- high CPU usage
- increased error rate
- frequent garbage collection

These describe *what is happening*, not *why it is happening*.

A system may show the same symptom for very different reasons, and the same cause may produce different symptoms depending on load, timing, and architecture.

Example

- high CPU usage may result from:
    - inefficient computation
    - excessive retries
    - memory pressure
    - contention
- high latency may result from:
    - queue buildup

    - I/O delays
    - synchronization

(→ 1.9 Common performance problems)

This is why symptoms must be treated as entry points for investigation, not as explanations.

Diagnostic implication

The same symptom can be produced by different causes.

Without identifying the underlying mechanism, corrective actions may target the wrong part of the system.

For example:

- reducing CPU usage may not reduce latency if the root cause is I/O queueing
- tuning GC may not help if allocation rate remains unchanged

A technically plausible fix may therefore have little effect if it addresses only a visible consequence.

Why confusion happens

Symptoms and causes are often confused because symptoms are easier to observe.

Metrics, dashboards, and monitoring systems usually show:

- what is high
- what is slow
- what is failing

They do not automatically explain:

- why it is high
- why it is slow
- why it is failing

This gap between visibility and explanation is exactly what diagnostics must bridge.

Practical interpretation

A good diagnostic process treats every symptom as a clue, not as a conclusion.

The goal is to move from:

- “this metric is abnormal”

to:

- “this mechanism is producing the abnormal behavior”

That shift is what distinguishes performance reasoning from superficial monitoring.

Key idea

Observed behavior is not the cause.

Diagnosis requires mapping symptoms to the mechanisms that generate them.

1.10.3 Correlation and causality

Definition

Correlation is the simultaneous variation of two signals.

Causality is a direct relationship where one factor produces another.

This distinction is essential in diagnostics because many metrics move together under load, but not all of them are causally related in the same direction.

Common mistake

Two metrics change together:

- CPU increases
- latency increases

This does not imply that CPU is the cause of latency.

Correlation may indicate:

- a common underlying cause
- an indirect dependency
- a causal chain in the opposite direction
- or simple coincidence in the same time window

Example

Possible interpretations:

- CPU saturation → scheduling delays → latency
- I/O delays → more concurrent threads → higher CPU usage
- contention → retries → both CPU and latency increase

(→ 1.5 System behavior under load)
(→ 1.8 Resource-level performance)

In all three cases, CPU and latency move together, but the underlying mechanism is different.

Diagnostic implication

Correlation is a starting point, not a conclusion.

Multiple mechanisms can produce the same correlated signals.
Only a causal model explains how one leads to the other.

For this reason, diagnostic reasoning must go beyond “these two metrics moved at the same time.”

It must explain:

- which changed first
- which mechanism links them
- why the observed sequence is consistent with system behavior

---

Practical approach

To establish causality:

- identify the sequence of events
- verify consistency with known system behavior
- validate through observation or controlled change

This may include:

- comparing before/after states
- observing whether one metric consistently leads another
- changing one condition and verifying the expected response

Causality becomes stronger when the system behaves as the proposed mechanism predicts.

Limits of superficial analysis

A dashboard can show correlation very clearly.

It cannot, by itself, prove causation.

This is why diagnostics requires reasoning, not only visualization.

A performance engineer must ask:

- Is this metric the driver, the consequence, or another consequence of the same event?
- Does the timeline support the proposed explanation?
- Does the explanation remain consistent across repeated observations?

Without these questions, correlation can easily lead to incorrect conclusions.

Practical interpretation

Good diagnostics treats correlation as a hypothesis generator.

It helps identify where to look, but it does not remove the need to reason about mechanisms.

This is especially important in complex systems where multiple bottlenecks interact and symptoms propagate across components.

Key idea

Do not infer causation from correlation.

Diagnosis requires identifying the mechanism that links signals.

1.10.4 Building a hypothesis

Definition

A hypothesis is a proposed explanation linking observed signals to a system mechanism.

It provides a structured way to move from observation to explanation.

Without a hypothesis, analysis remains descriptive rather than diagnostic.

Process

A hypothesis is built by:

1. observing signals
2. identifying consistent patterns
3. mapping them to known mechanisms

(→ 1.2 Core metrics and formulas)
(→ 1.5 System behavior under load)

This process transforms raw data into a testable explanation.

It connects:

- measurements
- system behavior
- causal reasoning

Example

Observed:

- latency increases
- queue length increases
- CPU approaches saturation

Hypothesis:

- increased arrival rate → queue buildup → longer waiting time → CPU saturation

This connects observable signals to a queueing mechanism.

It also gives the investigation a direction: verify whether the latency increase is caused primarily by waiting rather than by slower service time.

Requirements

A valid hypothesis must be:

- consistent with observed data
- grounded in system behavior
- testable through measurement or change

A hypothesis that cannot be tested may be plausible, but it is not yet useful for diagnostics.

A hypothesis that contradicts observed evidence should be rejected even if it appears intuitive.

Diagnostic implication

A hypothesis guides investigation.

Without it, analysis becomes reactive and unstructured.

Instead of moving directly from symptom to fix, the diagnostic process should move from:

- symptom
- to candidate mechanism
- to validation

This structure reduces guesswork and makes diagnostic conclusions more robust.

Sources of hypotheses

Hypotheses usually emerge from:

- observed signal combinations
- known performance patterns
- previous system behavior
- architectural knowledge
- repeated failure scenarios

For example:

- rising latency + growing queues often suggests queueing

- moderate CPU + blocked threads may suggest contention or I/O wait
- rising GC frequency + latency spikes may suggest memory pressure

These associations do not prove the explanation, but they provide a disciplined starting point.

Practical interpretation

A good hypothesis is specific enough to test and broad enough to explain the observed behavior.

It should not be:

- vague (“the system is slow”)
- circular (“latency is high because requests are slow”)
- purely descriptive

It should express a mechanism.

For example:

- “Thread pool saturation is increasing queue time, which is driving p95 latency up.”

This kind of statement can be validated.

Key idea

Diagnosis proceeds through explicit, testable hypotheses, not assumptions.

1.10.5 Narrowing down the bottleneck

Definition

Diagnostics aims to identify the resource or mechanism that limits system performance.

This limiting factor determines the overall system behavior under load.

Until it is identified, optimization efforts remain uncertain and often ineffective.

Approach

The analysis focuses on:

- CPU behavior
- I/O latency
- network delays
- memory pressure

(→ 1.8 Resource-level performance)
(→ 1.7 Runtime and memory model)

These dimensions are examined because most performance limits eventually manifest through one or more of them.

However, the dominant bottleneck at a given time is usually one primary constraint rather than all constraints equally.

Method

- isolate one dimension at a time
- compare signals across resources
- identify the dominant constraint

This reduces complexity by focusing on the most impactful factor.

The goal is not to explain every metric at once, but to find the mechanism currently governing system behavior.

Example

If:

- CPU is low
- I/O latency is high
- queues are growing

Then:

- I/O is likely the limiting factor

The system is not CPU-bound, even if CPU is active.

This kind of narrowing is essential because multiple resources are often involved, but only one is usually dominant.

Diagnostic implication

Performance is typically limited by a single dominant bottleneck at a given time.

Optimizing non-limiting resources produces little or no improvement.

This is one of the most important principles in diagnostics:

- measure broadly
- conclude narrowly

A broad set of signals is required to avoid missing important evidence.
A narrow conclusion is required to focus action on the actual constraint.

Why bottlenecks are difficult to identify

Bottlenecks are often obscured by secondary effects.

For example:

- slow I/O may increase thread count
- increased thread count may increase CPU scheduling overhead
- increased waiting may inflate memory retention
- retries may amplify demand on several components at once

As a result, the visible effect may not appear at the exact location of the original problem.

This is why narrowing down the bottleneck requires correlation across layers rather than isolated interpretation of one metric.

Practical interpretation

The purpose of diagnosis is not only to say that the system is under pressure.

It is to identify:

- where pressure becomes limiting
- which mechanism produces the limit
- why that constraint is currently dominant

Only then does optimization become meaningful.

Key idea

Effective diagnosis reduces the system to its limiting factor.

1.10.6 Iterative analysis and validation

Definition

Diagnosis is an iterative process of testing and refining hypotheses.

It evolves through successive observations and validations.

This is necessary because initial explanations are often incomplete, partially correct, or valid only for one layer of the system.

Process

1. observe signals
2. build hypothesis
3. test through changes or measurements
4. validate or reject

Each step refines the understanding of the system.

This loop is repeated until the proposed explanation is consistent with observed behavior and supported by evidence.

Example

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

for (int i = 0; i < 1000; i++) {
    pool.submit(() -> {
        Thread.sleep(100);
        return null;
    });
}
```

Interpretation:

- fixed thread pool limits parallel execution
- tasks accumulate
- queueing increases latency

This hypothesis can be tested by:

- increasing pool size
- reducing blocking time

If latency decreases and queue buildup is reduced, the hypothesis gains support.

If behavior does not change as expected, the explanation must be revised.

Validation

A hypothesis is validated if:

- changes produce expected effects
- signals evolve consistently with the proposed mechanism

If not, the hypothesis must be revised.

Validation therefore depends on consistency between:

- observed change
- expected change
- proposed causal explanation

A fix that changes one metric without improving system behavior may indicate that the wrong mechanism was targeted.

Practical implications

- avoid one-step conclusions
- iterate systematically
- validate assumptions with observable data

Good diagnostics is rarely instantaneous.

It becomes reliable through repeated comparison between:

- what is observed
- what is expected
- what actually changes after intervention

This iterative discipline is what turns troubleshooting into engineering.

Why iteration matters

Complex systems rarely expose a complete explanation in a single observation.

It is common to discover that:

- an initial bottleneck was only a secondary effect
- removing one constraint exposes another
- a local improvement shifts the limiting factor elsewhere
- the system behaves differently under different workloads

Iteration is therefore not a sign of uncertainty.
It is the normal method of reaching a stable explanation.

Practical interpretation

Diagnosis is a loop because system understanding is built progressively.

The objective is not to guess correctly on the first attempt.

The objective is to move from evidence to explanation through controlled reasoning and verification.

This is what makes performance analysis repeatable and defensible.

Key idea

Diagnosis is a loop.

Understanding emerges through iteration, verification, and refinement.

◀ 01-09-common-performance-problems | ▲ Index | 01-11-practical-checklists ▶

1.11 – Practical checklists

This chapter provides practical checklists for preparing, running, and analyzing performance tests.

Unlike the previous chapters, which explain concepts and mechanisms, this chapter focuses on operational discipline.

The goal is to reduce avoidable mistakes and ensure that performance tests produce results that are interpretable, reliable, and useful.

Table of Contents

1.11.1 Before running a test

Objectives

Clearly define what the test is intended to validate.

Typical objectives include:

- latency targets
- throughput goals
- capacity limits

A test without a clear objective may still generate data, but that data will be difficult to evaluate.

The first question should always be:

- what is this test supposed to prove, validate, or reveal?

Workload definition

Define the workload precisely:

- request rate or concurrency
- request mix
- duration

(→ 1.4 Types of performance tests)

The workload must be specific enough to be reproducible and realistic enough to be meaningful.

A vague or artificial workload can produce technically correct results that are operationally irrelevant.

Environment consistency

Ensure that:

- test environment is stable
- configuration matches production assumptions
- external dependencies are controlled

If the environment changes during testing, interpretation becomes uncertain.

Performance results are only comparable if the execution conditions remain sufficiently consistent.

This is especially important when evaluating:

- configuration changes

- code changes
- infrastructure changes

Metrics setup

Verify that all required metrics are available:

- latency percentiles
- throughput
- resource utilization
- error rate

(→ 1.2 Core metrics and formulas)

It is also useful to ensure that supporting signals are available when relevant, such as:

- queue lengths
- dependency timings
- GC activity
- thread or pool states

The test should not begin before visibility is in place.

Readiness checks

Before running the test, confirm that:

- the target system is in the expected state
- monitoring is active
- the workload generator is configured correctly
- the test duration is appropriate for the chosen objective
- success and failure criteria are known in advance

This avoids a common problem in performance testing: running a technically valid test that cannot later be interpreted with confidence.

Practical interpretation

Preparation is part of the test.

Most unreliable results are not caused by complex system behavior, but by poor test preparation:

- unclear objectives
- unrealistic workload
- inconsistent environment
- incomplete metrics

A well-prepared test makes later diagnostics far easier.

Key idea

A test is only meaningful if objectives, workload, and measurements are clearly defined.

1.11.2 During test execution

Monitoring

Observe system behavior in real time:

- latency evolution
- throughput stability
- resource usage

Monitoring during execution is important because some issues are visible only while the test is running, especially:

- sudden saturation
- unexpected queueing
- unstable recovery
- dependency failures

Waiting until the end of the test may hide important time-dependent behavior.

Consistency checks

Ensure that:

- workload is applied as expected
- no external disturbances affect the test

This includes verifying that:

- the intended request rate is actually being generated
- the mix of operations remains consistent
- no unrelated activity is distorting results
- failures are caused by the test conditions rather than by external noise

A mismatch between intended workload and actual workload can invalidate the entire interpretation.

Early signals

Watch for:

- rapid latency increase
- unexpected errors
- resource saturation

(→ 1.8 Resource-level performance)

These are often the first signs that the system is approaching a limit or that the workload is exposing an unanticipated bottleneck.

Early detection matters because it allows the test operator to:

- capture relevant evidence
- preserve useful context
- avoid losing the most informative part of the run

Runtime observations

During execution, it is useful to observe not only absolute values, but also change over time.

Examples:

- latency rising while throughput remains flat
- queue lengths growing before CPU saturation
- errors appearing only after a specific threshold
- p95/p99 degrading before the average changes significantly

These patterns often reveal more than isolated snapshots.

They help distinguish:

- transient instability
- steady overload
- slow degradation
- sudden collapse

Intervention discipline

During a test, avoid changing parameters unless the change is part of the test plan.

Unplanned intervention makes results harder to interpret because it mixes multiple causes into the same observation window.

If intervention becomes necessary, it should be:

- documented
- timestamped
- explicitly linked to the observed behavior

This preserves the diagnostic value of the run.

Practical interpretation

Execution is the phase where theoretical preparation meets real system behavior.

A well-designed test can still become misleading if the operator does not confirm that:

- the workload is correct
- the environment remains stable
- the system is behaving as expected or, importantly, as unexpectedly as the test was intended to reveal

---

Key idea

Execution is not passive.

Continuous observation is required to detect anomalies early.

1.11.3 After test analysis

Data review

Analyze collected data:

- latency distribution
- throughput trends
- resource utilization

Data review should focus not only on average values, but also on the shape of behavior over time.

For example:

- when degradation began
- whether throughput scaled as expected
- whether tail latency widened before failures appeared

This makes the analysis more diagnostic and less descriptive.

Correlation

Relate signals:

- latency vs CPU
- latency vs I/O
- errors vs load

(→ 1.10 Diagnostics and analysis)

Correlation helps identify which resource or mechanism is most likely associated with the observed degradation.

However, correlation should be treated as an analytical starting point, not a final conclusion.

Interpretation

Identify:

- bottlenecks
- scaling limits
- abnormal patterns

Interpretation should answer questions such as:

- what changed first?
- what degraded next?
- which constraint became dominant?
- was the degradation gradual, abrupt, or time-dependent?

This is the point where raw measurements become system understanding.

Reporting

Summarize:

- observed behavior
- identified issues
- recommendations

A useful report does more than list numbers.

It should explain:

- what the system was expected to do
- what it actually did
- where it diverged from expectations
- what evidence supports the conclusion

This makes the results actionable for engineering, operations, and future testing.

Next-step orientation

After analysis, define what should happen next.

This may include:

- re-running the same test after changes
- refining workload realism
- collecting deeper diagnostics
- isolating a suspected bottleneck
- expanding to stress, soak, or capacity testing

Without a next-step decision, analysis remains informative but not operationally useful.

Practical interpretation

Post-test analysis is where performance engineering becomes decision-making.

The purpose is not only to state that a metric changed, but to explain:

- why the change matters
- what it implies about the system
- what should be done next

Key idea

Analysis transforms raw data into actionable understanding.

1.11.4 Common pitfalls

Misinterpreting averages

- averages hide tail latency
- percentiles provide a clearer view

(→ 1.2.7 Percentiles)

A system can appear healthy on average while still producing unacceptable performance for a meaningful fraction of requests.

This is one of the most common mistakes in test interpretation.

Ignoring workload realism

- unrealistic workloads produce misleading results
- production patterns must be approximated

A synthetic workload may be easier to generate, but if it does not reflect real request mix, concurrency, and dependency behavior, conclusions may not transfer to production conditions.

Realism does not require perfect reproduction, but it does require credible approximation.

Confusing symptom and cause

- high CPU is not always the root problem
- latency must be analyzed in context

(→ 1.10 Diagnostics and analysis)

This pitfall often leads to ineffective optimization.

The visible symptom may be only the consequence of a deeper mechanism such as queueing, blocking, or dependency slowdown.

Overlooking bottlenecks

- optimizing non-limiting resources has little effect
- focus must remain on the dominant constraint

(→ 1.8 Resource-level performance)

This is a frequent source of wasted effort.

A system may contain many imperfections, but only some of them matter at the current operating point.

Running tests without acceptance criteria

A test is difficult to interpret if there is no prior definition of acceptable behavior.

Without explicit thresholds, it becomes unclear whether the result means:

- success
- failure
- degradation
- acceptable risk

Performance numbers are useful only when compared to defined expectations.

Treating one test as definitive

A single test run rarely captures the full behavior of a system.

Different runs may expose:

- warm-up effects
- dependency variability
- long-term drift
- threshold behavior under different load profiles

Reliable performance analysis usually requires comparison, repetition, and validation.

Ignoring time dimension

Some problems do not appear immediately.

A short test may miss:

- slow memory growth
- delayed queue buildup
- gradual dependency degradation
- runtime instability over time

This is why test duration must match the type of behavior being evaluated.

Practical interpretation

Most mistakes in performance testing are not caused by bad tools.

They are caused by:

- weak assumptions
- incomplete visibility
- poor interpretation
- lack of methodological discipline

Avoiding these pitfalls is often more valuable than adding more measurement detail.

Key idea

Incorrect assumptions lead to incorrect conclusions.

Avoiding common pitfalls is essential for reliable performance analysis.