cloudsweep

cost aware engineering

WHITE PAPER

Unlocking true efficiency

How to enable Elastic, Predictable, Cost-Efficient Cloud Applications

Executive Summary

In traditional on-premises environments, infrastructure is sized for **peak load**—the worst burst of demand the system might ever face. Servers are fixed assets, scaling is slow, and insufficient provisioning leads to outages. As a result, organizations historically over-provision hardware to remain safe.

Public cloud was meant to change this model:

"Size for typical load. Scale elastically for peak. Pay only for what you need."

But in practice, most companies still run their cloud as if it were an on-premises data center:

- large, static, overprovisioned baselines
- autoscaling fleets that rarely change capacity
- clusters running <30% CPU utilization
- infrastructure sized for comfort and low risk, not for observed workloads
- skyrocketing cloud bills with little visibility into root causes

Why does the cloud so often behave like on-prem?

Because cloud elasticity assumes **stable, predictable, well-behaved software**—and most modern codebases do not meet this requirement, even with software that was specifically designed to run on the cloud. Three software behaviors break the cloud elasticity model:

- **Unpredictable resource usage** — high variance per request, superlinear algorithms, "heavy user" paths
- **Performance degradation over time** — memory leaks, GC thrashing, cache bloat
- **Micro-spikes that outrun autoscaling** — traffic spikes faster than scale-up windows

When these occur, infrastructure teams are forced to **size the cloud like on-prem**: for peak, not typical load.

On top of that, most organizations are structured so that:

- **Production** owns the cloud bill
- **Engineering** writes the code that drives the bill

… but neither team is incentivized to fix the misalignment

Cloudsweep solves this by analyzing code, runtime behavior, and cloud economics together, generating actionable code improvements and safe rightsizing recommendations that transform workloads into predictable, elastic-friendly services.

This white paper explains how cloud sizing *should* work, why it usually fails, and why fixing code—not just tweaking infrastructure—is the only reliable way to achieve true cloud efficiency.

On-Premises vs. Cloud: Two Different Sizing Models

On-Premises → Hardware Sized for Peak Demand

In the traditional data center, hardware procurement cycles are slow (weeks or months). Scaling is manual, expensive, and operationally risky. Capacity must be provisioned in advance. Under-provisioning leads to outages, and outages are **unacceptable**. Therefore:

On-prem sizing = provision for the highest predicted peak.

This made sense since it would not be possible to procure and install the required hardware on-time for a peak. You plan for months and hope for the best. The price to pay is highly underutilized infrastructure. This is considered the "cost of doing business".

Cloud → Elastic Infrastructure Sized for typical Demand

The cloud introduces an elastic computing model that provides "infinite" capacity – as long as you can afford it. In this model, you size for your typical load (+percentile margin) and you use the cloud's scaling capability to fill in the gaps when peak arrives. Thus, the cloud changes the equation:

- Compute can scale in minutes.
- Capacity is (practically) infinite.
- Workloads can scale horizontally.

Therefore:

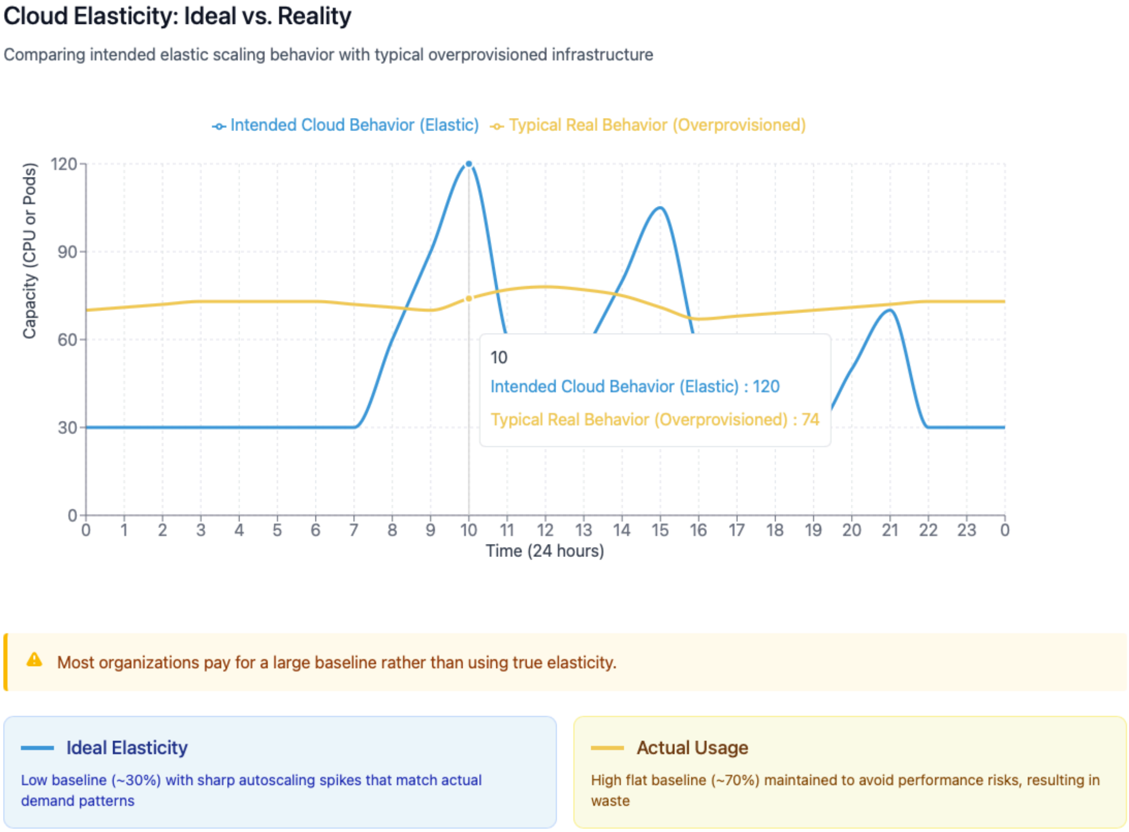
Cloud sizing = provision for the typical load, not peak.

And you use autoscaling elasticity to handle bursts. BUT there is some small print in this contract.

Elasticity's hidden assumptions

However, Cloud elasticity **works only** if:

- Workload behavior is **predictable**
- Scaling actions are **timely and effective**
- There is no **hidden performance decay**
- Bursts **align with scale-up latency** (time to serve the peak is lower than peak duration).



Cloud Elasticity: Ideal vs. Reality

Comparing intended elastic scaling behavior with typical overprovisioned infrastructure

⚠ Most organizations pay for a large baseline rather than using true elasticity.

Ideal Elasticity

Low baseline (~30%) with sharp autoscaling spikes that match actual demand patterns

Actual Usage

High flat baseline (~70%) maintained to avoid performance risks, resulting in waste

If these assumptions fail, the cloud behaves like on-premises —but in the worst, most expensive way possible.

Why Cloud Rightsizing Fails?

Most cloud optimization tools (including CSP “recommendations”) operate under the naive assumption:

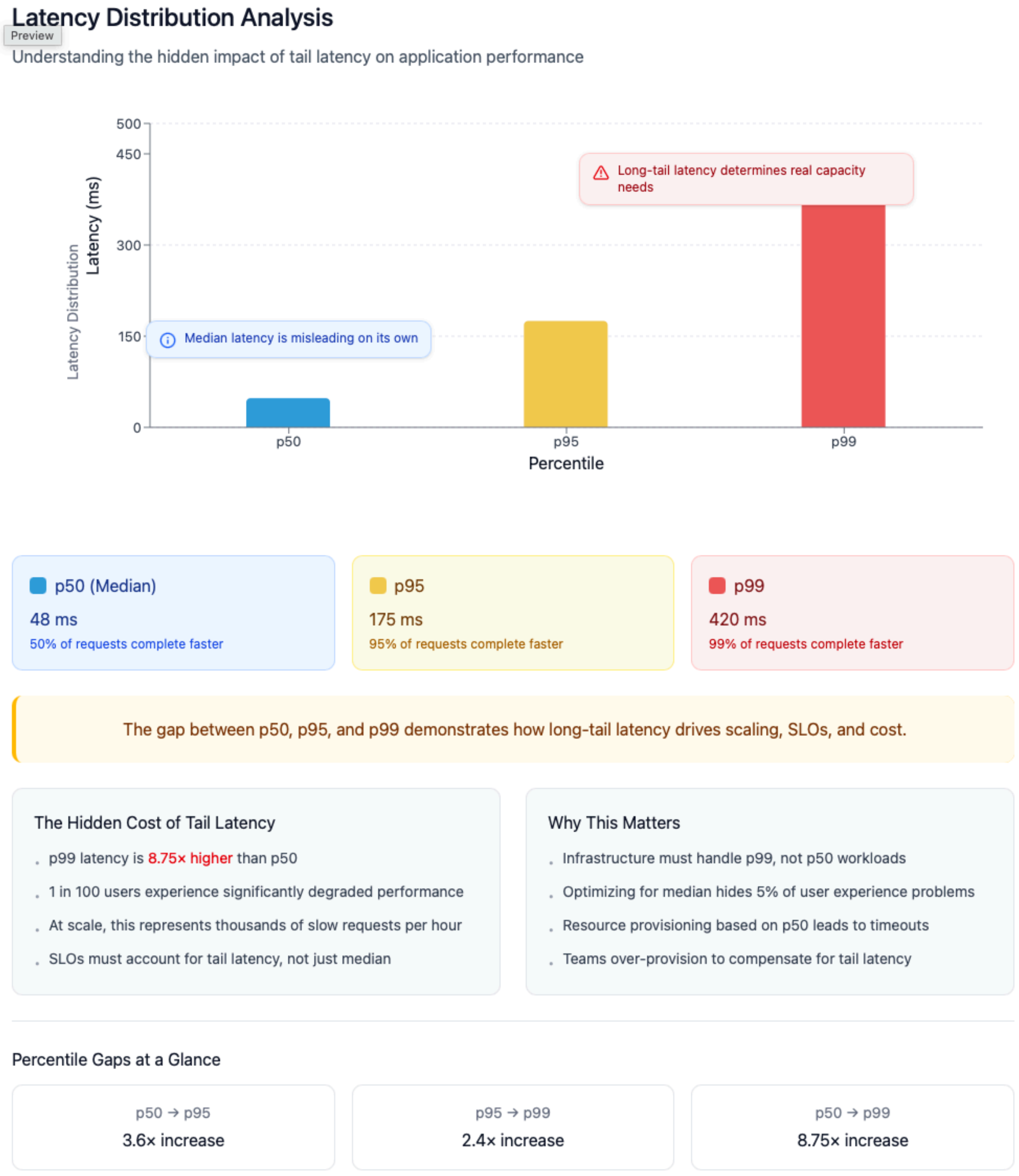
Historical average usage → future required capacity.

But this assumption only holds if workloads behave consistently. The problem is that **software rarely does**. Three failure modes undermine the entire elasticity model.

Failure Mode #1 — Unpredictable Resource Usage

Cloud elasticity requires **low variance**: each new request or event must consume roughly the same CPU/memory.

Example: N+1 Query Variant of a “Simple” Endpoint



Preview
500
450
300
150
0
Latency Distribution
Latency (ms)
Long-tail latency determines real capacity needs
Median latency is misleading on its own
p50
p95
p99
Percentile

Latency Distribution Analysis

Understanding the hidden impact of tail latency on application performance

🟦 p50 (Median)
48 ms
50% of requests complete faster

🟨 p95
175 ms
95% of requests complete faster

🟥 p99
420 ms
99% of requests complete faster

The gap between p50, p95, and p99 demonstrates how long-tail latency drives scaling, SLOs, and cost.

The Hidden Cost of Tail Latency

- p99 latency is 8.75× higher than p50
- 1 in 100 users experience significantly degraded performance
- At scale, this represents thousands of slow requests per hour
- SLOs must account for tail latency, not just median

Why This Matters

- Infrastructure must handle p99, not p50 workloads
- Optimizing for median hides 5% of user experience problems
- Resource provisioning based on p50 leads to timeouts
- Teams over-provision to compensate for tail latency

Percentile Gaps at a Glance

p50 → p95	p95 → p99	p50 → p99
3.6× increase	2.4× increase	8.75× increase

Consider an “Order History” endpoint:

- 90% of users have 10–20 orders → *CPU cost ~5ms*
- 10% of users have thousands of orders → *CPU cost ~200–400ms*

To autoscaling and rightsizing algorithms, this appears as:

- p50 CPU per request: **tiny**
- p95–p99: **enormous**

This generates unpredictable spikes when “heavy users” randomly arrive. The consequences are:

- The infrastructure baseline must be sized for heavy users, not average users
- Autoscaling triggers oscillate due to noise
- Cost-per-request becomes impossible to estimate

This is a simple example, but in a real application you would have dozens of endpoints working like this, and hundreds of thousands, if not millions of lines of code, making cloud sizing for typical load a risky exercise. Since the production guys like to sleep at night, there is only one option for them: **overprovision**.

Cloudsweep’s Approach

Cloudsweep identifies the code causing variance:

- N+1 patterns
- super-linear loops
- inefficient serialization
- expensive fan-out
- cache misses
- excessive object allocation / GC churn

… and others

It quantifies their cost (e.g. “extra $48k/year due to long-tail CPU variance”) and recommends/fixes to each one of the problems making your system behave in a predictable, linear way. Cloudsweep can do this over millions of lines of code (your technical debt) and orchestrate an implementation roadmap together with your engineering and production teams. Cloudsweep also avoids introducing these problems on day-to-day activities, since it monitors your Pull Requests and automatically provides a recommendation / fix when it finds cost anti-patterns.

Failure Mode #2 — Performance Degradation Over Time

Even predictable workloads break elasticity if they degrade while they run:

- memory leaks

- unbounded caches
- connection pool exhaustion
- GC pauses growing over time
- write amplification
- concurrency starvation

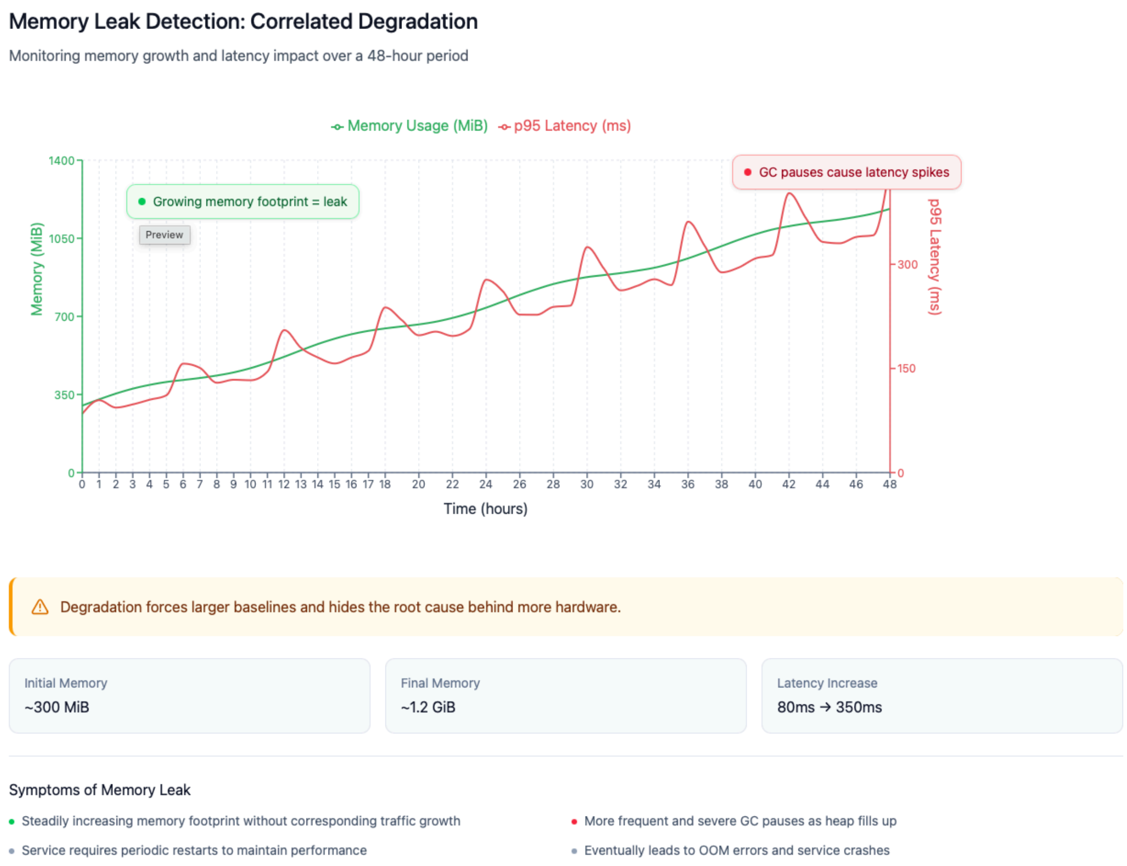
Example: Java Microservice with a Small Memory Leak

Imagine a service that works like this:

- Starts at 300MB RAM → healthy
- After 8 hours at steady load → 1.2GB
- GC overhead grows → latency spikes
- CPU jumps → autoscaling fires
- More pods = more leaks = more cost

*GC: garbage collection

Autoscaling has compensated for a **code defect**, not demand.



Memory Leak Detection: Correlated Degradation

Monitoring memory growth and latency impact over a 48-hour period

⚠ Degradation forces larger baselines and hides the root cause behind more hardware.

Initial Memory	Final Memory	Latency Increase
~300 MiB	~1.2 GiB	80ms → 350ms

Symptoms of Memory Leak

- Steadily increasing memory footprint without corresponding traffic growth
- More frequent and severe GC pauses as heap fills up
- Service requires periodic restarts to maintain performance
- Eventually leads to OOM errors and service crashes

Cloudsweep's Approach

Cloudsweep analyzes:

- Code and intended behavior
- Memory profiles
- Allocation hotspots
- Long-running pod behavior

It determines the leak pattern and quantifies cost inflation (e.g. "2 extra nodes = $22k/year"). Cloudsweep will also present you the reason for degradation and recommendation /fix for it. Allowing you to avoid the cost in the first place.

Failure Mode #3 — Spike Patterns Incompatible with Autoscaling

Autoscaling works by observing specific metrics (CPU usage, memory, traffic, etc.) and triggering a new pod running the software when the threshold is met. However, autoscaling needs:

- 20–60s to observe metrics – detect the scaling pattern
- 20–60s to evaluate rules – decide what to do
- 30–120s for new pod/instance to start – start the infrastructure
- warmup time (depends on your application) – start your code

Total: **40–180+ seconds**.

Most traffic spikes in real systems occur in **under a minute**. Therefore, autoscaling only works for predictable, well-behaved, spikes not the ones that are heavy on demand and arrive unpredictability.

Example: Synchronous Fan-Out Under Campaign Load

One on-line checkout request for an e-commerce website fans-out:

→ Pricing request

→ Inventory check

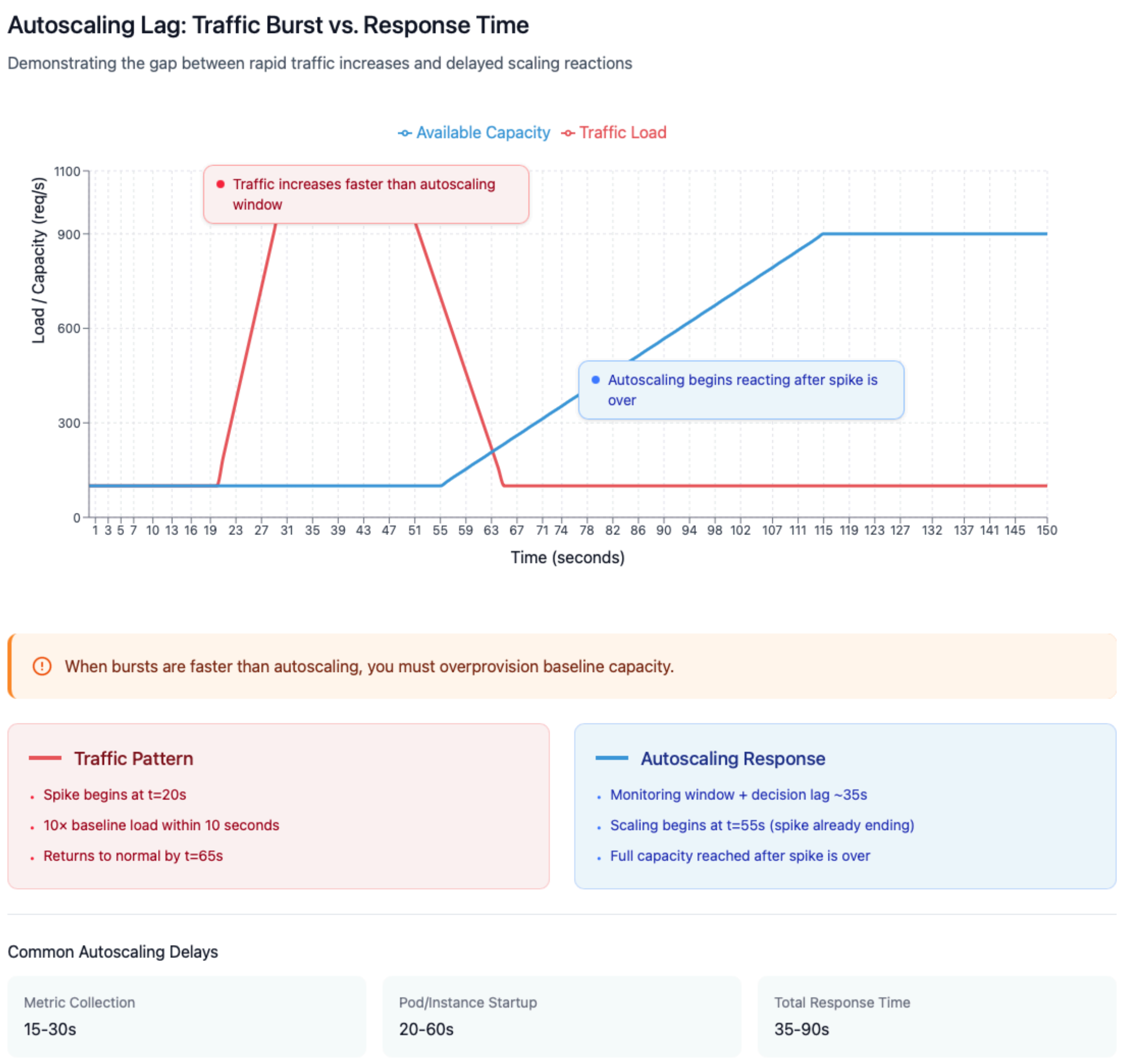
→ Fraud detection

→ Recommendations options

During a 30-second marketing spike:

- traffic jumps 10X
- internal calls multiply → 40X backend load
- CPU spikes instantly

autoscaling begins scaling after the spike is already over. The user suffers delays or service interruptions, **and** you are still paying for the additional cloud capacity.



Autoscaling Lag: Traffic Burst vs. Response Time

Demonstrating the gap between rapid traffic increases and delayed scaling reactions

● Traffic increases faster than autoscaling window

● Autoscaling begins reacting after spike is over

When bursts are faster than autoscaling, you must overprovision baseline capacity.

Traffic Pattern

- Spike begins at t=20s
- 10x baseline load within 10 seconds
- Returns to normal by t=65s

Autoscaling Response

- Monitoring window + decision lag ~35s
- Scaling begins at t=55s (spike already ending)
- Full capacity reached after spike is over

Common Autoscaling Delays

Metric Collection	Pod/Instance Startup	Total Response Time
15-30s	20-60s	35-90s

Consequence: Ops must overprovision typical capacity “just in case”.

Cloudsweep’s Approach

Cloudsweep analyses your code and architecture, together with your telemetry to find:

- synchronous bottlenecks
- concurrency explosions
- fan-out chains
- thundering herd problems
- queue starvation

And suggests:

- batching
- queuing
- concurrency guards
- circuit breakers
- local caches
- architectural shifts that smooth spikes
- …

Cloudsweep also addresses other types of problems like slow instance starting times and allows you to improve the overall autoscaling behavior of cloud + code.

This smooths the code's behavior allowing for a more reasonable cloud provisioning strategy and an "Elastic friendly" behavior of your code. In turn, it also provides your production team an explanation for the system's behavior allowing them to feel more at ease with lowering the typical capacity sizing.

Why can't I do this myself?

A fair question that many CTO's and Engineering leads ask. You have a top-notch engineering team; they have been delivering outstanding software for years. Why can't they just fix the software? The answer is: you can, but it will be slow and expensive. The reasons are:

- **No one knows the whole codebase**. If you have been operating for a few years, have 500K > lines of code and the team has regularly rotated. Who knows every line of code?
- **Focus**: doing this manually will require your best engineers focused on a cost problem. They cannot create new features while they do this.
- **Using an off-the-shelf LLM** and making it review your full codebase will not work. The full-size of your code base is way bigger than the LLM's context window and once that happens you are taking a huge risk.

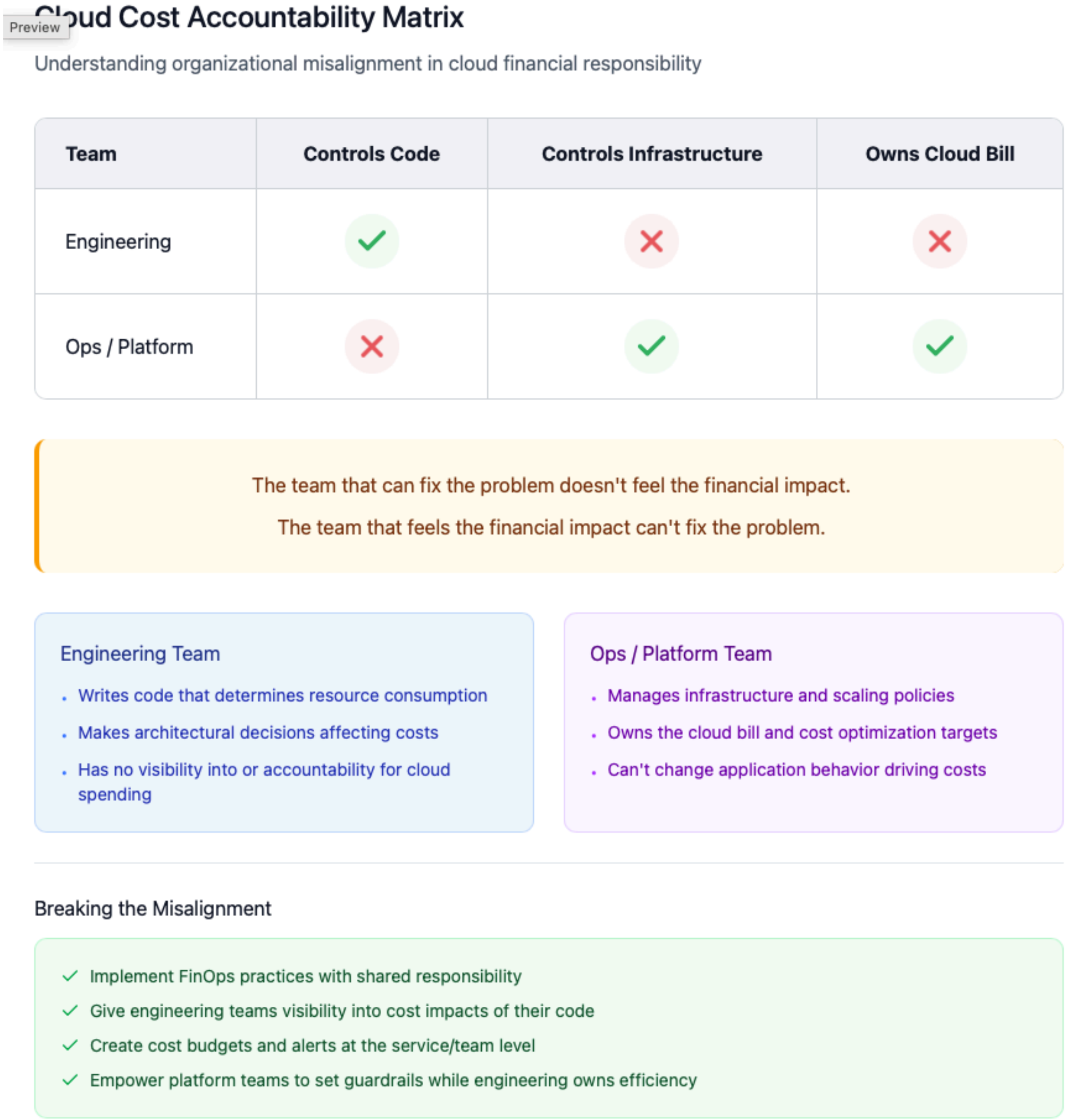
- **You cannot do it the whole time:** this is not a solved once, solved forever, type problem. As your codebase grows, the problem compounds, you need to do it as a daily activity. One wrong Pull Request is all it takes to incur huge costs.
- **Pride is a poor advisor.** People naturally overlook the negative consequences of their own work, which is why having an objective third party to identify and fix inefficient code is not just helpful—it's essential.

Cloudsweep's Approach

Cloudsweep acts as a specialized 24/7 expert team member that reviews and provides fixes for code and infrastructure problems. It takes care of your backlog and your pull requests and acts as a code reviewer/fixer in near real-time. This melts the cost backlog, allows you to focus where it matters (your business), proactively avoiding problems. It also helps the production team to rightsize the infrastructure by providing root cause analysis on how the system behaves.

The Hidden Organizational Failure: Wrong Team Owns the Problem

Even when technical issues are understood, most organizations **cannot fix cloud cost** because of structural misalignment:



Cloud Cost Accountability Matrix

Understanding organizational misalignment in cloud financial responsibility

Team	Controls Code	Controls Infrastructure	Owns Cloud Bill
Engineering	✓	✗	✗
Ops / Platform	✗	✓	✓

The team that can fix the problem doesn't feel the financial impact.

The team that feels the financial impact can't fix the problem.

Engineering Team

- Writes code that determines resource consumption
- Makes architectural decisions affecting costs
- Has no visibility into or accountability for cloud spending

Ops / Platform Team

- Manages infrastructure and scaling policies
- Owns the cloud bill and cost optimization targets
- Can't change application behavior driving costs

Breaking the Misalignment

- ✓ Implement FinOps practices with shared responsibility
- ✓ Give engineering teams visibility into cost impacts of their code
- ✓ Create cost budgets and alerts at the service/team level
- ✓ Empower platform teams to set guardrails while engineering owns efficiency

This creates the conditions for systemic failure:

- Production sees a cost problem but cannot fix the code.
- Engineering can fix the code but has no incentive, nor availability..
- Management does not understand the problem and pushes everyone.

The result: overprovisioning becomes the safe default for production and engineering can say it is not our problem.

In most organizations everyone knows the infrastructure is grossly underutilized, but this is considered “**Cost of doing business**”. Where have we seen this before?

The result is bloated cloud costs that are out of control. FinOps was created to resolve this problem, however by failing to recognize that the root cause is mainly poorly designed code and architecture, **FinOps addresses the symptoms** and not the root-cause of the problem.

Cloudsweep's Approach

By providing root cause analysis of the code and architecture problems and relating them to the cloud rightsizing recommendations, Cloudsweep becomes the shared truth:

- Explains **which code** forces **which infra decisions**
- Quantifies each problem in **dollars, latency, and SLO risk**
- Generates PRs → Engineering allowing engineering to **take action**
- Gives Ops confidence to downsize safely and understand what Engineering is doing
- Provides Finance with ROI-driven clarity

This realigns incentives and closes the loop.

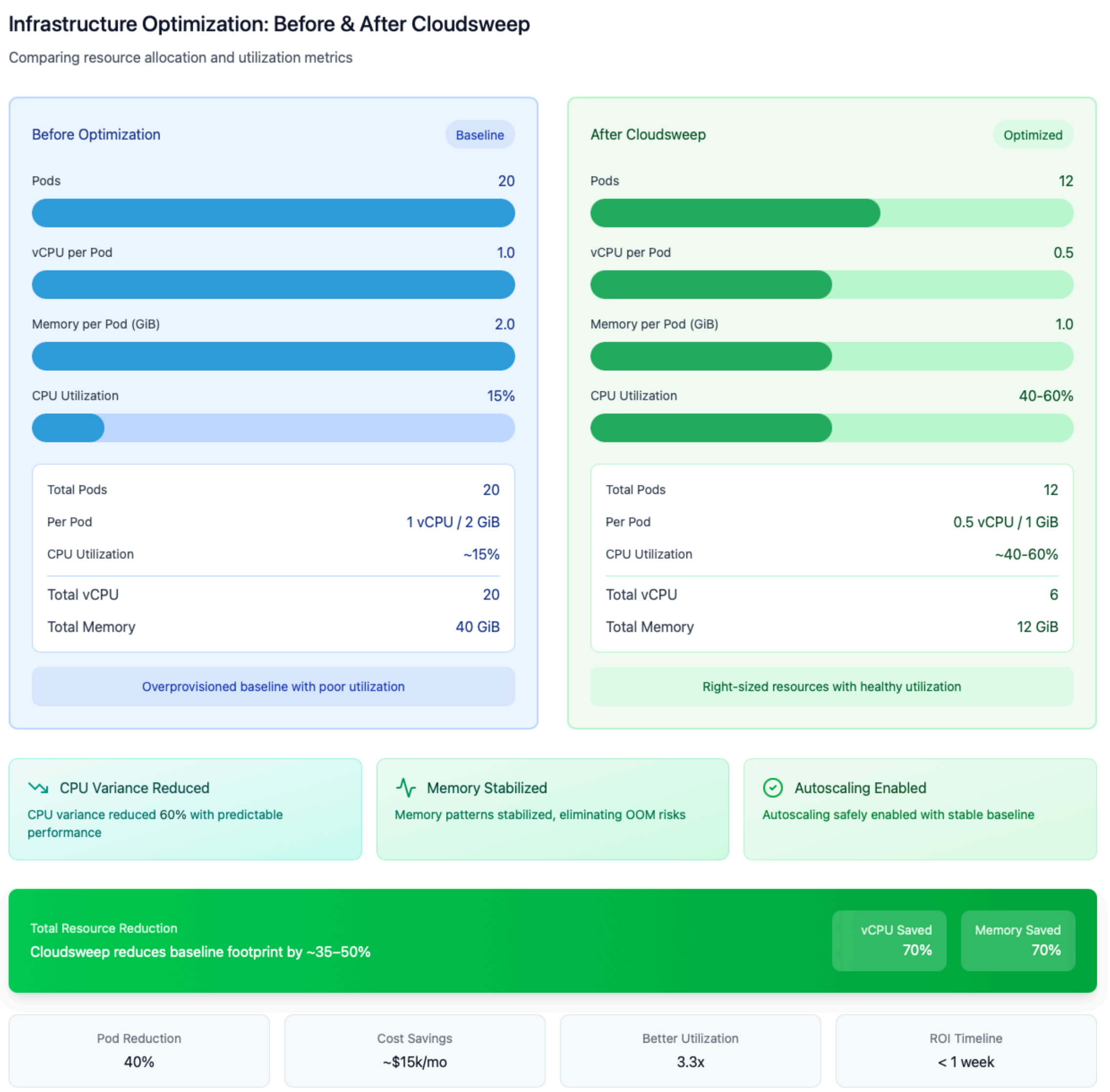
10. Conclusion

The promise for cloud adoption was that you would be paying only for what you need:

- pay-for-usage
- on-demand scaling
- efficiency through elasticity

However, these benefits only emerge when software behaves predictably and avoid the elasticity blocking bottlenecks.

- Most workloads don't.
- Most teams don't know why.
- And most organizations are structured so that **the wrong people own the problem**.



Infrastructure Optimization: Before & After Cloudsweep

Comparing resource allocation and utilization metrics

Before Optimization — Baseline

Pods	20
vCPU per Pod	1.0
Memory per Pod (GiB)	2.0
CPU Utilization	15%

Total Pods	20
Per Pod	1 vCPU / 2 GiB
CPU Utilization	~15%
Total vCPU	20
Total Memory	40 GiB

Overprovisioned baseline with poor utilization

After Cloudsweep — Optimized

Pods	12
vCPU per Pod	0.5
Memory per Pod (GiB)	1.0
CPU Utilization	40-60%

Total Pods	12
Per Pod	0.5 vCPU / 1 GiB
CPU Utilization	~40-60%
Total vCPU	6
Total Memory	12 GiB

Right-sized resources with healthy utilization

CPU Variance Reduced
CPU variance reduced 60% with predictable performance

Memory Stabilized
Memory patterns stabilized, eliminating OOM risks

Autoscaling Enabled
Autoscaling safely enabled with stable baseline

Total Resource Reduction
Cloudsweep reduces baseline footprint by ~35–50%

vCPU Saved 70%

Memory Saved 70%

Pod Reduction	Cost Savings	Better Utilization	ROI Timeline
40%	~$15k/mo	3.3x	< 1 week

Cloudsweep fixes all three dimensions:

- Code behavior
- Infrastructure sizing
- Organizational alignment

By revealing the deep connection between code and cloud costs—and then fixing the underlying problems—Cloudsweep enables companies to run cloud infrastructure the way it was meant to be run.