FinOps anti-patterns: What (not) to do to unlock more budget for Innovation

22 October 2025 - 2 min. read

Here we are with yet another portmanteau of something and Operations.

This time it's Finance's turn.

Generally, the first encounter with the term FinOps is approached with a certain superficiality: it's taken for granted that if Operations concerns infrastructure and Finance means "$$", then the union of the two must necessarily mean "reducing waste in the cloud", right?

Well, not exactly.
Or rather, not only.

Reducing costs is certainly part of the game, but stopping there would be like thinking that DevOps only means "automating builds".

FinOps is much more; it's a cultural and operational approach that unites technical, financial, and business teams to maximize the value of the Cloud, not simply to save a few dollars.

To clear up any doubts, spending a paragraph explaining some FinOps concepts in broad strokes is worth it. If you already know them, feel free to skip it :)

What does FinOps mean?

First, let's clarify a fundamental point: FinOps doesn't introduce revolutionary concepts. Instead, it's a structured way of bringing attention to efficiency back into focus. The framework was created to maximize the business value of the Cloud,

enabling rapid, data-driven decisions and creating shared financial responsibility among engineering, finance, and business.

In this sense, it represents a real paradigm shift: cost management is no longer a post-mortem activity, but becomes an integral part of daily decision-making processes.

The FinOps journey unfolds in three iterative phases that feed into each other: **Inform**, **Optimize**, and **Operate**.

In the **Inform** phase, attention ifocuses on collecting and valuing data related to cloud costs, usage, and efficiency. This data is transformed into analysis and reporting tools that offer accurate visibility into spending dynamics. This leads to budgeting and forecasting capabilities and the ability to measure performance through KPIs and internal or external benchmarks. The elastic and on-demand nature of the Cloud, together with discount mechanisms, makes continuous monitoring essential: only with timely information is it possible to anticipate spending, prevent surprises, and maintain a return on investment consistent with business objectives.

The **Optimize** phase, on the other hand, focuses on efficiency. This is when the collected data is used to identify concrete optimization opportunities, for example, by resizing underutilized resources, adopting more modern architectures, automating workload management, and reducing waste. Rates also become a field for optimization, thanks to discount models such as Reserved Instances and Savings Plans. This phase is not limited to cutting costs but aims for continuous dialogue between

the different teams to ensure that cloud performance always remains aligned with the organization's strategic objectives.

Finally, in the **Operate** phase, changes are consolidated. This is where governance policies are defined, compliance is monitored, people are held accountable, and a culture of shared responsibility is spread. Technical, financial, and commercial teams collaborate constantly, making iterative and incremental decisions based on the evidence collected in the previous phases. The approach is cyclical: each action leads to new information, which fuels further optimizations and makes processes more mature and aware.

The value of FinOps lies precisely in this continuous cycle. Informing, optimizing, and operating are **not isolated moments** but phases that intertwine and repeat, leading the organization to an increasingly evolved cloud management capable of generating concrete value.

Three Common Anti-patterns

Now that we've clarified what FinOps is and how it works, let's look at where things can go wrong.

As consultants, we're certainly no strangers to desperate situations. Sometimes, however, the most insidious errors don't manifest immediately: they remain silent during the design and implementation phases, only to reveal themselves after going live in production. And it's precisely then, when the infrastructure begins to scale and costs increase, that it's too late to intervene and the bill (literally) arrives.

These errors result from designs that have considered only technical requirements, ignoring (or superficially treating) the economic impact of architectural choices. An approach that contrasts with the "shift-left" of costs: bringing financial awareness from the earliest stages of design, when decisions can still make a difference.

But how cool is serverless?

Let's face it: the concept of serverless is fascinating.

The idea of not having to manage infrastructure (nor sysadmins!) is every developer's dream.

And so, thanks to ease of use and automatic scalability, a serverless architecture is often chosen even when it's not the most suitable choice.

To make informed decisions, it's necessary to have clear use cases.

Serverless is perfect for:

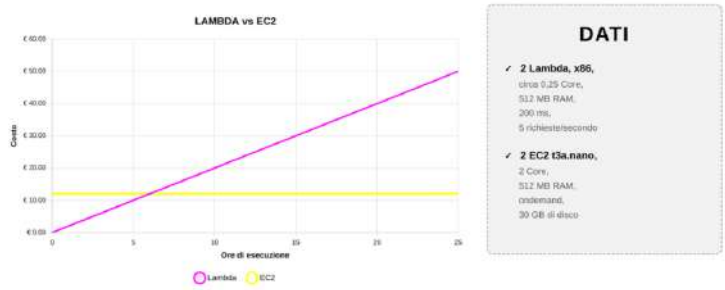
- **Applications with variable load:** In these situations, the scaling capabilities of serverless provide maximum benefit.
- **Microservices or event-driven applications:** Each microservice can become a managed container or a serverless function launched by a trigger.
- **Economic efficiency for sporadic use:** Many serverless services can scale to zero, making costs null in case of non-use.

It's worth asking a few more questions if you have:

- **Applications with constant load**: A virtual machine will always cost less and perform better for constant use.
- **Applications requiring low latency:** If latency is critical, serverless might not be the best choice due to "cold starts."
- **Fear of vendor lock-in:** The serverless approach leads to greater integration with the cloud provider, which not all requirements allow.

Let's take a suitably simplified example of comparing the running costs of a Lambda Function and an EC2 machine. Lambda costs grow more or less linearly with usage time, while the costs of a certain size EC2 machine, always on, are fixed. Furthermore,

it's important to consider that the example EC2 has greater computational power and a disk.



Clearly, this reasoning falls apart if the workload varies greatly over time. An EC2 machine that is always on has infinitely higher costs than a Lambda executed sporadically.

More generally, *Modeling* is a cornerstone principle of FinOps that requires modeling the load profile and cost trends over time before making architectural decisions. This approach integrates economic analysis as a primary non-functional requirement (NFR), ensuring that the initial design is sustainable and aligned with the goal of maximizing long-term business value.

As Werner Vogels also reminds us in his project The Frugal Architect:

“a well-designed architecture is one that balances performance, resilience, and cost over time”.

In other words, being “frugal” does not mean spending less, but spending better by building systems that scale sustainably without compromLogging like no tomorrowising quality.

Logging like no tomorrow

Those who do our job know that logs and metrics are fundamental to understanding the health of an infrastructure. Two of the most common solutions on AWS for centralized log management are **Amazon CloudWatch Logs** and **Amazon OpenSearch Service**. Both offer robust functionalities, but the choice between them largely depends on the specific requirements of the use case, data volume, query complexity, and available budget.

Native, well-integrated, and simple to use, CloudWatch is almost always the default choice. During a project kickoff, merely hearing about the costs and complexity of OpenSearch then removes any doubt. Unfortunately, however, even this time the simplest solution is not necessarily the best; as always, the best solution is: *it depends.*

The crucial difference lies in the economic model. CloudWatch's consumption-based approach can quickly escalate costs with an increase in the number of high-cardinality metrics, sampling frequency, and ingested and stored logs. A managed OpenSearch cluster, on the other hand, introduces a more "tiered" cost: nodes are sized, and volume is absorbed, with a cost that scales more predictably. Beyond a certain volume threshold, moving high-volume logs and custom metrics to OpenSearch (while keeping essential heartbeats and alarms in CloudWatch) proves significantly more efficient.

In the following example, a large-scale application was considered, based on EC2 in autoscaling and with an average of ten thousand daily users making an average of one hundred calls each. The single log record is quite large: 4kB. Finally, it is important to consider that the analysis took into account oversized OpenSearch instances, so as to avoid problems with scarce resources and allow for data analysis operations.

From the graph, it can be seen that for a limited number of virtual machines, Cloudwatch remains economically advantageous, but when the instances are in a medium/high number, the cost of OpenSearch reaches a plateau, while that of Cloudwatch continues to rise linearly.

A non-functional requirement specifies the criteria that can be used to judge a system's operation (accessibility, availability, scalability, ...), but what is often overlooked is the cost. Projects can fail because costs are not considered at every stage of the business: from design to development to operation.

From a Frugal Architect perspective, this means introducing economic awareness already in the design phase: every metric collected, every retention policy, and every log stream must be intentional.

"Measure everything but pay attention to what you keep", Vogels would say.

To Tenancy or Not to Tenancy

Your SaaS is gaining traction: increasing customers, demands for data isolation, and pressure on operational costs. The temptation might be to dedicate a complete installation (Single tenancy) for each client, replicating load balancers, EC2 instances, RDS databases, messaging queues, and so on. While this might seem like a tidy approach initially, this model presents critical issues: costs grow **linearly** with each new contract, management complexity explodes (upgrades and patches multiply), and margins quickly evaporate.

A well-designed multi-tenancy allows for secure sharing of resources and infrastructure, maintaining logical isolation between clients.

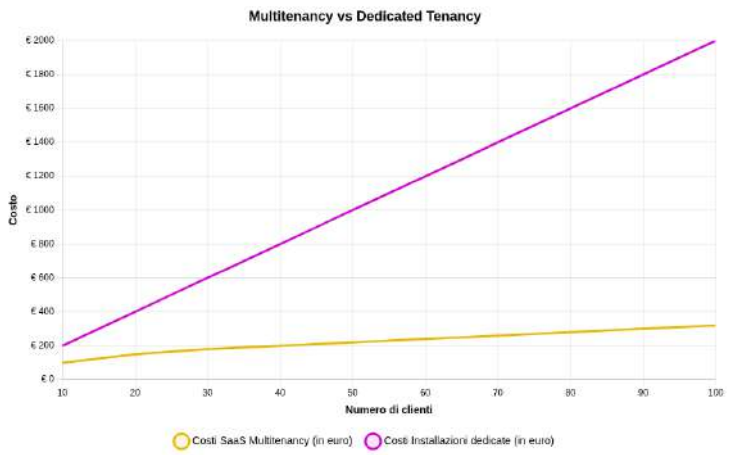
The main options are:

1. **Total Sharing:** sharing of the application layer and the database (with logical separation by schema or through row-level controls).

2. **Hybrid Approach:** shared application with a dedicated database per tenant, reserving this option only for premium clients or those with particularly stringent isolation requirements.

Security remains an absolute priority, but it is achieved through logical isolation (data separation via code and configuration), rigorous policies, per-tenant encryption, and advanced observability to prevent "noisy neighbors" (clients that negatively affect the performance of others), rather than by duplicating the entire infrastructure.

The example diagram this time is very simple. As mentioned previously, costs with single-tenant installations increase linearly with the number of installations. It is more difficult to outline the costs of the multi-tenant approach; for this analysis, we considered an initial increase of about 50%, decreasing as the number of clients increases.



Sustainable systems align costs with revenue curves. In a multi-tenant architecture, cost growth is much flatter; each additional customer brings positive margins rather than the need to maintain a new dedicated "mini-platform." Avoiding infrastructure cloning is not just a matter of architectural elegance, but it's what allows the business to scale sustainably, without blowing up the income statement.

Summing up

All of the **anti-patterns** we have examined have one thing in common.: they **arise from decisions made by considering costs as an "afterthought" problem**, something **to be addressed when the application is already in production**. But by then it's too late: the architecture is defined and modifying it becomes costly and risky.

The real lesson of FinOps is to **bring awareness to the very first phases of the software life cycle**. This "shift-left" of costs means treating economic impact as a requirement

on par with performance and security.

As The Frugal Architect reminds us: "Cost is a non-functional requirement". And like any self-respecting non-functional requirement, it should be considered from day one.

About Proud2beCloud

Proud2beCloud is a blog by beSharp, an Italian APN Premier Consulting Partner expert in designing, implementing, and managing complex Cloud infrastructures and advanced services on AWS. Before being writers, we are Cloud Experts working daily with AWS services since 2007. We are hungry readers, innovative builders, and gem-seekers. On Proud2beCloud, we regularly share our best AWS pro tips, configuration insights, in-depth news, tips&tricks, how-tos, and many other resources. Take part in the discussion!

Andrea Pusineri

DevOps Engineer @ beSharp. I love solving problems and I'm back belt of finding them. Linux enthusiast and security guy wannabe, I like to play CTFs, but in my spare time I'm an avid comic/manga/book reader. btw I use Arch

---

Nicola Ferrari

Cloud Infrastructure Line Manager @ beSharp and AWS authorized instructor champion.I live my life one level at a time getting superpowers by collecting caffeine hidden here and there in my daily map. I'm a hardened internet surfer (yes, I surfed the whole internet... twice!) and tech-addicted with a passion for computers and networking. Building great IT things all nice and tidy contribute to achieving my main goal: the pursuit of perfection!
