Usage Before Rate

Sequencing optimization work on a grown cloud estate, and why commitments usually come last

This paper describes how to sequence the work of the Optimize phase on a cloud estate that has grown for years without anyone looking at cost, and why Rate Optimization usually belongs at the end of that sequence rather than the beginning. It draws on twelve months of work on the production and staging accounts of one organization, where monthly spend fell from around 15,000 to around 2,500 euros with the same applications, the same users and no reduction in production availability. After reading it you should be able to build a cost baseline you can trust from raw billing data, decide in which order to work through a list of savings, argue to Finance and Leadership why commitments wait, and set up the handful of controls that stop an estate from growing back.

Who Should Read this Paper

- Engineers and architects who have been handed a bill nobody can explain and asked to do something about it
- FinOps practitioners at Crawl or early Walk maturity whose Optimize work so far consists mostly of buying commitments
- Engineering managers and product owners who have to decide how much team time cost work may take away from feature work
- Finance partners who want to understand why a cost project that starts with deleting data produces more durable savings than one that starts with a discount

It is written for estates in the range of a few thousand to a few tens of thousands of euros or dollars a month, run by one team or a handful of teams. The method transfers to larger estates. The proportions do not, see Exceptions and Considerations.

Prerequisites

- Familiarity with the FinOps Framework phases (Inform, Optimize, Operate) and with the Domains Understand Usage & Cost and Optimize Usage & Cost
- Read access to the detailed billing export of at least one cloud provider (the AWS Cost and Usage Report, an Azure cost export, a Google Cloud billing export) and the ability to query it with SQL
- A basic understanding of how managed services are metered: per hour, per gigabyte stored, per terabyte scanned, per request, per metric
- Infrastructure as code is not strictly required, but almost every measure below is a one-line change where it exists and a ticket where it does not

Paper Body

Which part of the Framework this paper is about

The paper sits in the Optimize phase and touches three Capabilities in the Domain Optimize Usage & Cost: Usage Optimization, Rate Optimization and Architecting & Workload Placement. It depends on two Capabilities from Understand Usage & Cost, Data Ingestion and Reporting & Analytics, because nothing in it works without a baseline the team believes. The closing section reaches into Manage the FinOps Practice, namely Governance, Policy & Risk, Anomaly Management and Unit Economics, because the least comfortable lesson of the twelve months was that everything grows back.

The aspect that bears explaining is the order. The Framework lists Usage Optimization and Rate Optimization side by side. It does not say which to do first, and in practice most teams start with rate because it is the convenient option: no code change, no operational risk, a few clicks, a visible discount on next month's invoice. On a grown estate the convenient first step is the expensive one. The rest of this document is about why, and about what to do instead.

Scope and context: the estate behind the figures

Two accounts, production and staging, running the same set of applications for a mid-sized business. Together they cost around 15,000 euros net per month when I took them over. The estate was four years old and had been built by a small team that always had something more important to do than look at cost. That is not a criticism. For most of those four years, growth was the right thing to optimize for.

Two properties of this estate matter for how far the findings transfer.

There was no single mistake. No forgotten cluster, no crypto miner, no database running in triplicate. The bill was high for the reasons most bills are high: new metrics stayed forever, data was never deleted, capacity was sized against the worst day anyone remembered, and staging ran around the clock. Bills rarely explode. They sediment. That determines the approach, because looking for the one culprit means looking past the problem. It also has a pleasant consequence: cost work does not have to expose anyone. It clears away what accumulates by nature.

And there was hardly any classic virtual machine fleet. The bill consisted almost entirely of managed services: RDS and DynamoDB for data, ECS on Fargate and Lambda for compute, S3 for storage, CloudWatch for monitoring and Athena for analytics. If you believe that serverless and managed services make the cost question go away, this is the counterexample. All that changes is where the money leaks. Not into oversized instances, but into data volumes, query patterns and observability.

The team's expectation at the start was a saving of ten to fifteen per cent. Nobody, myself included, would have signed off on more than eighty.

All figures below are rounded net amounts, converted from the invoiced US dollar amounts at the exchange rate of the respective month, with commitments shown amortized over their term. What that means, and why it matters, is the subject of the section on measurement.

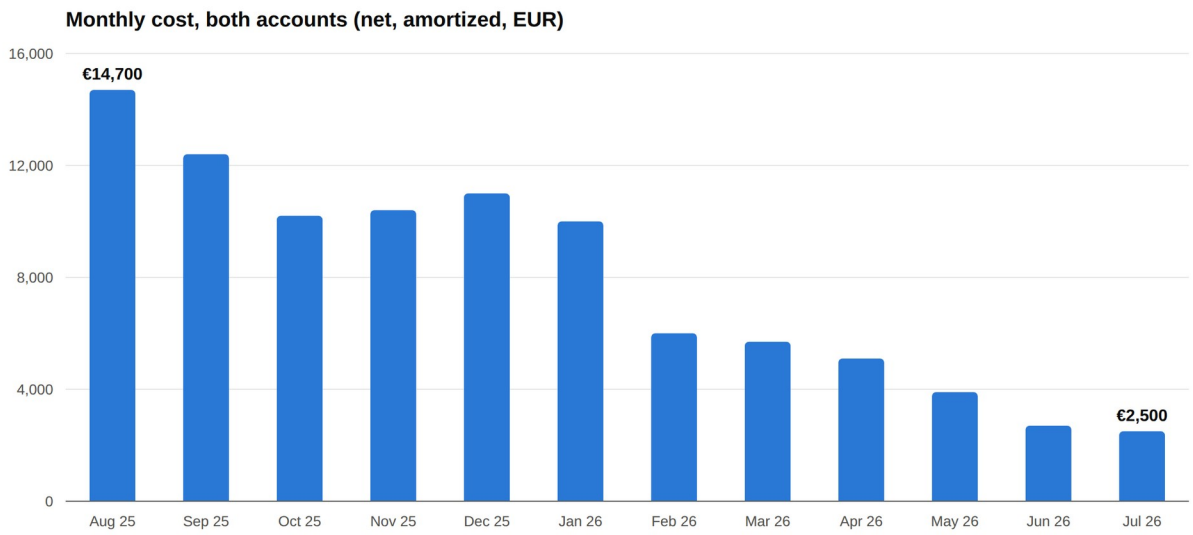

Figure 1: Monthly net cost of both accounts, amortized, August 2025 to July 2026. The two large steps are the observability and analytics clean-up around the turn of the year, then rightsizing, commitments and the storage clean-up in spring.

The common understanding, and what it leaves out

The usual picture of cost optimization on a grown estate looks like this: open the cost console, sort by service, buy commitments for the largest steady items, rightsize whatever the recommendation engine flags, and remove orphaned resources when someone gets round to it. Each of those is a legitimate measure. Done in that order, they fix the price of the problem instead of the problem.

The more complete picture differs in a few places.

The unit of analysis is the usage type and the resource, not the service. "CloudWatch: 2,600 euros" is not information anyone can act on. Behind it sit metrics, log ingestion, dashboards and alarms, with completely different causes and remedies. Until you can see the usage type and the resource behind a number, you are guessing.

A commitment locks in a spend or capacity baseline for its term, and that baseline is exactly what Usage Optimization changes. A commitment on a bloated estate cements the waste at a discount and takes the economic point out of much of the later optimization. The downsized database may save little or nothing against the committed baseline, because the old size has already been paid for.

On a managed-services estate, observability and data are cost centres in their own right. Here, monitoring and analytics together cost more than the databases they were supposed to watch. Two items that appear in no architecture diagram as a cost centre, because they just run alongside everything else.

Measure before touching anything

The first week produced no saving and was the most valuable week of the year.

Enable the detailed billing export with resource identifiers and make it queryable with SQL. On AWS that is Data Exports with the Cost and Usage Report 2.0, delivered to S3 and queried through Athena; the other large providers offer equivalent exports into a queryable store. Include resource IDs when you create the export. That is the part people forget, and it is what later lets you resolve a usage type down to the individual log group, table or bucket. Setup takes an hour. From then on, cost questions are answered with a query rather than an opinion.

Two queries carry almost the whole project. The first groups a month of spend by product and usage type and lists the top fifty. The second takes one usage type that stands out and resolves it by resource. The column names below are those of the AWS export; the shape of the query is the same everywhere. Both queries use unblended cost because they describe the baseline before any commitment existed. Once commitments are in place, use the amortized cost fields of the export for workload reporting, for the reason given a few paragraphs further down.

```
-- Query 1: where does the money go, by usage type rather than by service
SELECT
  line_item_product_code,
  line_item_usage_type,
  ROUND(SUM(line_item_unblended_cost), 2) AS cost
FROM cur
WHERE line_item_usage_start_date >= DATE '2025-08-01'
  AND line_item_usage_start_date <  DATE '2025-09-01'
GROUP BY 1, 2
ORDER BY cost DESC
LIMIT 50;

-- Query 2: one usage type that stands out, resolved to the resource
SELECT
  line_item_resource_id,
  ROUND(SUM(line_item_unblended_cost), 2) AS cost
FROM cur
WHERE line_item_product_code = 'AmazonCloudWatch'
  AND line_item_usage_type = 'EUC1-CW:MetricMonitorUsage'
  AND line_item_usage_start_date >= DATE '2025-08-01'
  AND line_item_usage_start_date <  DATE '2025-09-01'
GROUP BY 1
ORDER BY cost DESC
LIMIT 50;
```

The same analysis pattern also works against a FOCUS export, the FinOps Foundation's provider-neutral billing schema, which AWS Data Exports can deliver alongside the CUR. Unblended cost corresponds to `BilledCost`, the amortized view to `EffectiveCost`, the resource identifier to `ResourceId` and the product code to `ServiceName`. The usage type has no one-to-one column; `ChargeDescription` and `SkuId` carry that detail. Written against the corresponding FOCUS fields, the analysis becomes provider-neutral.

In this estate the first query overturned the team's mental model in one afternoon. The surprise was not that databases cost money. The surprise was that observability and analytics had caught up with them.

The output of the first week was a list of roughly thirty items, each with three figures: expected saving per month, effort in days, and risk to operations. That list is the working document for the rest of the project. Everything after it is working through the list in the right order.

Before the measures, a word on the numbers themselves, because there are traps in them and each has cost me time in earlier projects.

There is no such thing as "the" cost. The export carries several cost columns and they answer different questions. Unblended cost answers what was charged in a given month. As soon as commitments exist, only the amortized view shows the running cost of a workload, because it spreads upfront payments and recurring reservation fees across their term. This estate provided its own cautionary example: after the database reservations were bought, the RDS line in the unblended view dropped to a fraction. Reading that as "the databases are nearly free now" is measuring wrong. Strip one-off effects out as well. A data export for a migration is an event, not a trend.

Credits, discounts, tax and exchange rates move the invoice without moving usage. A saving read off the invoice total in a month where credits expired is noise. The provider bills in US dollars, so a euro comparison between two months moves even when nothing about consumption changed. So measure usage, meaning hours, gigabytes stored, terabytes scanned and requests, next to money. Usage does not lie.

Month-on-month comparison needs context. February is ten per cent shorter than January. A month with a marketing campaign is no benchmark for one without. I compare weekly averages against weekly averages, always alongside a business unit of measure.

This sounds pedantic. It decides whether the project builds trust or loses it. The first figure that turns out to be wrong costs more credibility than three correct ones can rebuild, and Finance in particular will remember it.

The sequence

The list of thirty items was worked through in this order:

1. Remove what nobody needs and cap retention
2. Review observability: logs, metrics, trails, the running cost of operating the system
3. Get data under control: storage classes, data volumes, query patterns
4. Size capacity against measured load
5. Change architecture where it burns money
6. Only then, commitments

Steps 1 to 4 are Usage Optimization in the Framework's terms. Step 5 is Architecting & Workload Placement. Step 6 is Rate Optimization. The order runs from lowest risk and fastest result to highest risk and slowest result, and it ends with the one measure that cannot be undone.

Step	What changed	Capability	Risk to operations	Saving per month
1	Retention on log groups, snapshots and backups, leftover project data	Usage Optimization	Practically none	~1,200 €
2	High-cardinality metric dimensions, metrics nobody read	Usage Optimization	Low, once alarm references are checked	~1,800 €
3	Partitioning, deleting data, lifecycle rules, backup retention	Usage Optimization	Low to medium, retention decisions needed	~4,000 €
4	Task sizes, function memory, interruptible work on spare capacity	Usage Optimization	Medium, one step at a time	~2,000 €
5	Scheduler for staging, caching, events instead of polling	Architecting & Workload Placement	Medium, touches running systems	~800 €
6	Reservations and a savings plan on the remaining load	Rate Optimization	Financial rather than operational	~2,700 €
	Total			~12,500 €

Table 1: The six steps, what they touched, and what each produced. Savings are approximate monthly net amounts, amortized, each measured in the weeks after its step completed and rounded to the nearest hundred. Their sum is therefore not identical to the difference between two individual months in Figure 1, which also carry exchange rate movements and usage differences; both readings come out at about 83 per cent.

Step 1: remove and cap retention

The unglamorous part, and little in the twelve months had a better ratio of effort to saving. What had accumulated: log groups without retention that had kept every byte for years, snapshots and backups far beyond any recovery requirement, and data left over from finished projects whose only job had been to cost money. A retention policy of 30 to 90 days is one line of infrastructure code.

The most important rule in this step is organizational, not technical. Never delete anything irreversibly straight away. Stop it, archive it or take a snapshot, wait two weeks to see whether anyone notices, then delete. In twelve months exactly one person noticed, and restoring took twenty minutes.

That rule is also the answer to the objection this step always produces: "how do you know nobody needs that?" I do not know. I do not need to know. I only need a way to correct a mistake in twenty minutes rather than in a crisis meeting. Stopping is reversible. Debating whether something might theoretically still be needed only costs weeks in which the resource keeps burning money.

Step 2: observability that costs more than what it observes

The most revealing finding of the project sat in CloudWatch, and it will look familiar to anyone running custom metrics with dimensions on any provider. Every unique combination of metric name and dimension values is a separate metric, and each is billed separately. That sounds harmless until high-cardinality values go into dimensions. A "processing time" metric is one metric. The same metric with a customer dimension is as many metrics as there are customers.

Add a container ID or a version and it multiplies. A handful of well-meant measurement points had become tens of thousands of billed metrics, and nobody had ever made a decision that felt wrong. Every dimension was a good idea on the day it was added.

For a while monitoring was the second-largest item on the entire bill, peaking in December, more expensive than the databases it watched. Observability that costs more than the thing observed has stopped being observability. It is a leak.

The approach: count metrics rather than estimate them, and find out which of them any alarm or dashboard has read in the last 90 days. Most are written and never read. Move high-cardinality values out of dimensions and into structured logs, where you query them when a question comes up instead of paying for every distinct value permanently. Dimensions answer "how is the system doing"; logs answer "what happened with customer X". Aggregate what belongs together; a percentile across a group is almost always enough where measurement was per instance.

Check the alarm references before cutting anything. A deleted metric with a production alarm attached is exactly the outage this project cannot afford.

Step 3: data. Storing, querying, throwing away

The largest block, and it had two stories with the same root: too much data, kept too long, queried too broadly.

Athena bills per terabyte scanned, as do the scan-metered analytics services of the other providers. That is a fair model as long as queries read only what they need. Here the data was unpartitioned, so every query, however small its question, scanned the whole dataset, and the dataset grew every day. Around 2,500 euros a month for answers that sat in a fraction of the data. The fix had two parts. Partition by date, so a question about last week reads last week rather than four years. And, the less comfortable part, ask why the dataset was so large at all. A substantial share was no longer needed for a single report. Less data means less scanning, less storage, less of everything. A columnar format amplifies the effect. But the order is: throw away first, then partition, then worry about formats.

The second story was the staging environment. It is not superfluous; every test needs it. But its data was treated like production data, kept as though something depended on it. Test data is consumable. Today aggressive lifecycle rules apply there and data is deleted after a short time. The effect only became fully visible in early summer, because the retention questions had to be settled first. What does the business need to keep, what does a legal retention duty require, and what is merely habit? Settling that takes longer than the line of infrastructure code afterwards, and it is the real core of the measure. It is also where the Product and Finance personas come in, because an engineer cannot answer it alone.

On top came the usual moves. Two weeks of access logging to see read patterns before writing rules. Lifecycle rules for everything with a clear pattern. Deleting aborted multipart uploads after seven days, an invisible item almost everyone has, because the fragments show up in no listing and cost storage like whole objects. Automatic tiering only for data with unknown or changing access patterns, after weighing object sizes against the per-object monitoring fee. And database backup retention set back to a level that matches an actual recovery requirement.

The saving here is permanent in a way a discount never is. It rests on work that no longer happens.

Step 4: capacity, sized against fear

Only now compute. Not earlier, because after steps 1 to 3 it was clear what load actually remains. Rightsizing a workload whose cause is being optimized away is wasted diligence.

The pattern with the container services was the one I know from virtual machine fleets: sized against fear, not against load. CPU and memory reservations used at a fraction on a weekly average. Capacity for a load case that occurs twice a year, held around the clock. I understand that fear. Size a task too small and cause an outage, and you have a conversation ahead of you. Make it twice as large as needed and you have none, because the cost does not stand out to anyone. All the incentives point towards oversizing, which is exactly why rightsizing only works with data that replaces the fear.

Concretely: task sizes from fourteen-day utilization metrics, usually one or two steps smaller than configured, applied one step at a time with a week of observation between steps. Jumping to a quarter in one step produces the outage that discredits the whole project. Function memory measured rather than guessed, because on a serverless function platform the memory setting also buys CPU and billing is in gigabyte-seconds; several functions became so much faster with more memory that total cost fell despite the higher allocation, while for others the existing setting was already right. Interruptible batch work moved to spare capacity at the provider's discount for it, provided the jobs checkpoint and can resume. Forcing spare capacity onto systems that cannot tolerate interruption is not cost optimization. It is a deferred outage.

Around 2,000 euros a month, and the most time-consuming step, because every change wants observing.

Step 5: architecture, where the bill is a symptom

Beyond a certain point, high cost is an architecture problem rather than a configuration problem. Some cases from this estate.

Staging ran around the clock and was used on weekdays between nine and six. That is 45 of 168 weekly hours, so 73 per cent of its runtime was unused. A scheduler that scales the services to zero and stops the databases in the evening, and reverses that before the working day starts, was an afternoon of work. The objections to switching things off disappeared after the first week. Nobody has asked about it since.

The same expensive query ran hundreds of times a minute with a result that rarely changed. A cache in front of the database took exactly that load away. Caching here is not a performance measure but a cost measure: the cheapest query is the one that never reaches the database.

Processes polled every minute to ask whether there was anything to do, creating baseline load around the clock. Event-driven triggering means compute only when something actually happens. For loads that arrive in waves, that is the difference between continuous operation and genuine usage-based billing.

None of these would have shown up as a finding in a cost tool. A tool sees a busy database and recommends a reservation for it. That the load itself is unnecessary is visible only to someone

who understands the system. This is the point where cost work turns from configuration maintenance into engineering, and the reason it cannot be fully automated. The smallest saving on the list, and the one with the biggest operational side effect: a system with less background noise.

Step 6: commitments, only on the cleaned-up baseline

After about six months the load was cleaned up and stable: analytics partitioned, monitoring slimmed down, containers on realistic sizes. Only then came commitments, and deliberately only on the load that will certainly stay. Not on the peaks, not on anything still being rebuilt. The storage clean-up even continued afterwards, so the bill kept falling after the commitments were bought, which is precisely what confirms the order.

Reserved Instances for the databases, Reserved Capacity for DynamoDB, a Compute Savings Plan for the Fargate load, each with a one-year term. The mechanism is the same everywhere and on every provider: a lower rate in exchange for a committed spend or capacity baseline over a fixed term. How literally that baseline fixes individual resources differs between instruments, and flexible plans soften it, but none of them shrink with the workload. Three years is an eternity in this industry, and the extra discount of the longer term is a bet that nothing substantial about the architecture will change. After six months of rebuilding I did not want to take that bet.

My rule of thumb: commit at most 70 to 80 per cent of demonstrated baseline load, never 100. The rest stays flexible. An oversized commitment is the one cost optimization that cannot be undone.

Around 2,700 euros a month on the remaining load.

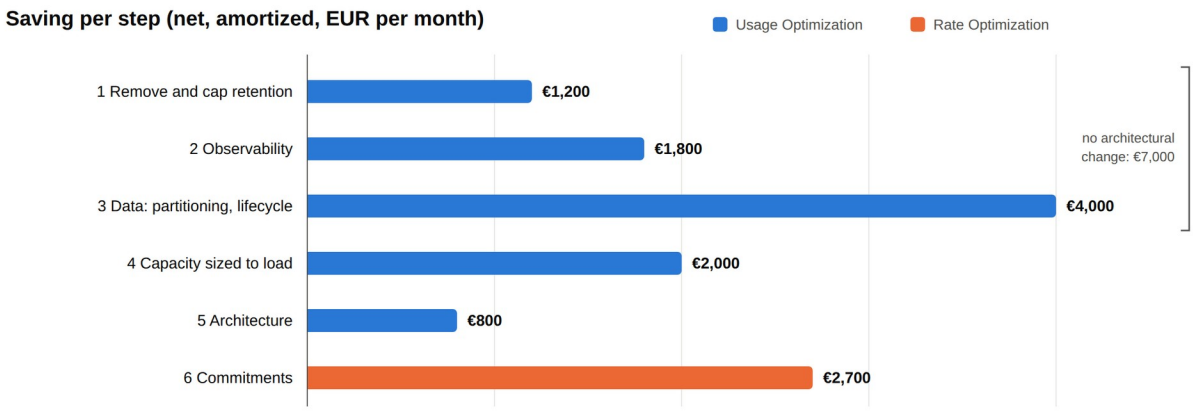

Figure 2: Saving per step, approximate monthly net amounts. Steps 1 to 3 required no architectural decision and account for 7,000 euros, more than half the total. Rate Optimization came last and was not the largest item.

Why the order matters

One reason is economic and has been stated already: a commitment locks in a spend or capacity baseline for one to three years. In this estate a commitment on the initial state would have locked in the database and compute baseline of that time, a multiple of what was actually

needed after the clean-up. Put differently, the most convenient first step would have ended the project before it began.

The other reason is organizational, less obvious and at least as important. Steps 1 to 3 consist of measures with immediately visible results and practically no risk, and they required not a single architectural decision. That is how you earn the trust you will need for the steps that touch running systems. Start with the riskiest rebuild and stumble, and you get no backing for the rest.

Figure 2 shows the distribution, and the distribution is the real message here. The three steps without any architectural change account for 7,000 of the 12,500 euros. The part people picture when they hear "cost optimization", reservations and discounts, came last and was not the largest item.

How the work fits alongside feature work

Twelve months sounds long. It was not a full-time project but a rhythm: one measure per week, each with the same routine. Implement, observe for a week, note the result, move on.

The rhythm had effects I had not expected. It makes the project interruptible. There were weeks when nothing happened because something more important came up, and that cost nothing but time, with no half-finished rebuilds and no state anyone had to understand again first. Cost work always competes with feature work, and it rightly loses that conflict when it behaves like a major project. As a weekly rhythm it never gets big enough to be in the way.

And it produces a track record. After three months a document said: measure, expected saving, actual saving. That list ended more discussions than any argument. Nobody argues with "several thousand a month saved with seven measures, here are the next five".

Keeping it from growing back

The least comfortable finding of the twelve months: without countermeasures, everything grows back. The same forces that pushed the bill to 15,000 euros are still at work. Every new service brings new metrics, every new pipeline new data, every new developer sizes against fear again. So the last month was not an optimization month but a set-up month. It is also where the story crosses from Optimize into Operate.

- Tagging with enforcement (Allocation). Four mandatory tags, team, environment, application and cost centre, enforced through tag policies and checks in the infrastructure pipeline. A deployment without them fails. A voluntary tag is missing from a third of resources after six months, and then every report is worthless.
- Budgets with alerts per account and environment (Budgeting), read as deviation from the expected trajectory rather than as an absolute amount. A budget that only raises the alarm at month end is an obituary.
- Anomaly detection (Anomaly Management) for the cases no budget anticipates. It does not replace analysis, but it shortens the time between "something got expensive" and "someone knows about it". The metric explosion in step 2 would have surfaced months earlier with it.

- Retention as the default (Governance, Policy & Risk). Every new log group and every new bucket gets a retention or lifecycle rule as a mandatory parameter of the infrastructure module. Anyone who wants to keep data indefinitely has to write that down and be able to justify it.
- Cost in everyday development (FinOps Education & Enablement). One cost figure per week in the team channel, broken down by application. Not as control, as visibility. Developers make cost-aware decisions as soon as they can see the numbers at all. Before that they have no chance to.
- Cost per business unit rather than the total (Unit Economics). Cost per thousand requests, per customer, per import. A rising total with usage rising faster is success, not a crisis. Without a denominator you cannot tell the difference, and that distinction is what turns cost discussions from panic into steering.

Personas involved

Engineering did almost all of the work and carried the operational risk. Everything in steps 1 to 5 is an engineering decision, which is why this work cannot be handed to a central team that does not know the systems.

Product and Finance owned the retention decisions in step 3 and the expectation of what the project would yield. "What does the business need to keep" is not an engineering question, and an engineer who answers it alone is guessing with someone else's risk.

Leadership had one job: to accept that steps 1 to 3 would take weeks before anyone touched the items that looked big, and that commitments would wait half a year. The weekly track record made that acceptance cheap.

Finance benefited most from the measurement discipline. Amortized figures, usage next to money and weekly averages meant that the saving reported was the saving that showed up in the invoice.

The FinOps Practitioner role did not exist in this organization. In an estate of this size it was one engineer with a mandate. The Framework's Capabilities still applied; the team model did not.

Maturity characteristics

In Framework terms the estate started before Crawl: no allocation, no reporting beyond the console, no accountability for cost, and Rate Optimization not yet begun. By the end it sat at Walk for the Capabilities it needed, and deliberately no further.

- Crawl, reached after the first two months: billing export queryable, a prioritized list with saving, effort and risk per item, the first clean-up done, cost reported monthly.
- Walk, reached after twelve: enforced tags, budgets per environment, anomaly detection on, retention as a default, commitments covering a defined share of baseline load, a weekly cost figure per application, and unit cost for the two or three business units that matter.
- Run would mean rightsizing recommendations acted on by policy, commitment coverage managed continuously, and unit economics driving product decisions. None of that was

worth its cost for two accounts. The Framework's maturity model says the same: mature the Capabilities that pay for it.

What wasted time

The list of dead ends is shorter than the list of measures, but it exists.

Chasing small amounts. Somewhere in month three I spent half a day understanding a 40-euro item. The amount has been worth 40 euros every month since; the analysis was not. Since then a hard threshold applies: anything below a defined monthly amount is not investigated while something larger is still open. The ten largest usage types in this estate explained over 80 per cent of the bill.

Evaluating tools instead of writing queries. I looked at two cost optimization tools seriously. Both would have found what the first week of queries found, at a recurring price of several per cent of the saving. For an organization with hundreds of accounts the maths may work out differently. For two accounts, the billing export plus an afternoon of SQL was the better deal. Finding was never the problem. Changing was, and no dashboard does that for you.

Discussing architecture rebuilds too early. In week two there was a long debate about rebuilding part of the system. In hindsight that was procrastination at a high level. Steps 1 to 3 required no architectural decision and delivered the bulk of the saving. Months later the same discussion was short, because by then there was data instead of opinions.

Conclusion

Costs did not fall because a tool found a discount. They fell because unused work was removed, the remaining load was sized to what had been measured, and only then committed. The steps that needed no architectural decision produced more than half the saving. Rate Optimization produced less than a quarter, and it produced that much only because it came last.

A few footnotes that tend to be missing from success stories. The 83 per cent is not a benchmark. These accounts had four years of sediment, and an estate someone looks at regularly might yield 20 to 30. It took twelve months because the work ran alongside feature work and every change was observed before the next; full time, this is three months, but not a sprint and not a button. And the feature set stayed the same, but the system did not. It got simpler. Most cost optimizations are simplifications in disguise: fewer unused resources, fewer special cases, fewer things that can break at night. The bill turned out to be a surprisingly good indicator of architectural quality. It is just the only one measured in currency.

Indicators of Success

- Share of spend resolvable to a resource: at least 95 per cent of resource-attributable workload spend can be tied to a named resource from the billing export. Account-level charges such as support plans, tax, credits and line items that carry no resource identifier are excluded from the denominator. Validate by summing resolved cost against the workload total.

- Amortized cost per business unit flat or falling while usage rises. Measure on weekly averages, not calendar months, with the denominator agreed with Product.
- Ratio of Usage Optimization saving to Rate Optimization saving. In this estate it ended near 4:1. On a previously unmanaged estate, a ratio below 1:1 is a signal to examine whether commitments were bought before the usage baseline had been optimized far enough. On an estate that was already well kept, more rate than usage potential is entirely legitimate.
- Commitment coverage between 70 and 80 per cent of demonstrated baseline load, with utilization of the commitments above 95 per cent. Both come from the provider's coverage and utilization reports.
- Zero unplanned production incidents attributable to a cost measure. Every change has a rollback path and an observation period; count incidents in the change log against the list of measures.
- Time from a cost anomaly to someone acknowledging it below one week, from anomaly detection alerts and the weekly team figure.
- Tag compliance at 100 per cent for the mandatory set, checked in the deployment pipeline rather than after the fact.

Exceptions and Considerations

- Regularly maintained estates. The 83 per cent came from four years of sediment. An estate with an owner who looks at cost every quarter may find 20 to 30 per cent, and steps 1 to 3 may be nearly empty. The sequence still holds; the proportions do not.
- Commitments that already exist. If commitments were bought before the clean-up, the economics of steps 3 to 5 change: reducing a workload below its committed baseline saves little or nothing until the term ends, depending on how flexible the instrument is. Plan the usage work to finish shortly before renewal, and do not renew at the old quantity.
- Early-stage products. Before product-market fit, or when the engineer hour costs more than the line item it would save, cost optimization is the waste. Whether an analysis is worth it is a question for the start, not the end.
- Systems about to be replaced. Optimizing a workload that will be rebuilt next quarter optimizes something that is about to disappear. Step 1 still applies; steps 4 and 5 do not.
- Legal retention duties. Where one exists, step 3 is a classification exercise before it is a deletion exercise. Involve whoever owns the duty. An engineer's guess is not a retention decision.
- Resilience against cost. Zone-aware routing and single-zone placement save data transfer and are in tension with availability targets. What has to be multi-zone stays multi-zone; that saving is not worth an outage.
- Large, multi-account organizations. With hundreds of accounts, the central tooling I set aside for two accounts may pay for itself, and a central FinOps team becomes the natural owner of steps 1 and 6. Steps 2 to 5 remain engineering work in the teams that own the systems.

- Other providers. The mechanisms transfer: billing exports with resource identifiers exist everywhere, monitoring services meter custom metrics, analytics services meter scanned data, and commitments lock in a baseline. The specific findings, which services dominated the bill, do not transfer. Run the first query and let the estate tell you.

Related Papers

- How to Optimize Cloud Usage (FinOps Foundation): the catalogue of Usage Optimization measures. The sequence above adds the order in which to apply them on a grown estate, and the evidence for putting rate last.
- Commitment Based Discounts Playbook (FinOps Foundation): how to buy and manage commitments. The argument above is about when: after usage work has stabilized the baseline, and at 70 to 80 per cent coverage rather than full coverage.
- Adopting FinOps – Getting Started (FinOps Foundation): the organizational entry point. The twelve months described here are one example of what the first year of the Optimize phase can look like once the billing export is in place.

Related FinOps Resources and Framework Capabilities

- Usage Optimization (https://www.finops.org/framework/capabilities/usage-optimization/): steps 1 to 4 of the sequence.
- Rate Optimization (https://www.finops.org/framework/capabilities/rate-optimization/): step 6, and the argument for its position.
- Architecting & Workload Placement (https://www.finops.org/framework/capabilities/architecting-workload-placement/): step 5.
- Data Ingestion and Reporting & Analytics (https://www.finops.org/framework/capabilities/): the baseline and the traps in the numbers.
- Allocation, Anomaly Management, Budgeting, Unit Economics, and Governance, Policy & Risk (https://www.finops.org/framework/capabilities/): the section on keeping the estate from growing back.
- FinOps Maturity Model (https://www.finops.org/framework/maturity-model/): the Crawl, Walk and Run characteristics described above.
- FOCUS, the FinOps Open Cost and Usage Specification (https://focus.finops.org/): the provider-neutral column names referenced next to the sample queries.
- Source material for the figures in this paper: the author's field report "Cutting AWS costs: from 15,000 to 2,500 euros a month" (https://timrutte.de/en/blog/aws-cost-reduction-case-report/), which also contains the CLI commands and additional queries used.

Attribution

Written by Tim Rutte, Cloud & Software Architect, Germany. More than twenty years in software development, AWS Certified Solutions Architect – Professional. The work described here was

carried out for a client between August 2025 and July 2026. The client is not named, and all figures are the author's own measurements from the billing export of that estate. No third-party tooling was used, and none is recommended. LinkedIn: https://www.linkedin.com/in/tim-rutte. Web: https://timrutte.de/en/