

Token Economics: A Capacity Planning Framework for Enterprise GenAI Cost Governance

Shaik Irfan

Independent Researcher, India. Email: shaikh.urfan@gmail.com

Abstract: Enterprise generative AI cost is hard to predict because token consumption does not behave like ordinary infrastructure demand. A token is the unit a model provider bills on, a chunk of text roughly three quarters of a word long. Token use rises with how complicated a prompt is, which model was picked, and how people end up using the feature. It does not rise neatly with headcount or transaction volume. So finance ends up explaining invoices after they arrive instead of forecasting spend before it happens. This paper treats tokens as a capacity resource rather than a line-item expense. It borrows three practices that are decades old in compute and storage capacity planning and almost entirely missing from token cost management: modeling demand by type of work rather than by team, measuring what normal consumption looks like for each type and raising an alert when it moves, and treating the number of steps a model takes as something to forecast rather than something to accept. The framework sets out a six-step cycle that runs from sorting work into classes through to reconciling the forecast against the invoice, an instrumentation layer defining what every call record has to carry, and a control model that separates three kinds of limit organizations routinely confuse: a safety cap that blocks, a soft threshold that alerts, and a spending ceiling that escalates. No threshold numbers are published, because every threshold has to be set against a baseline measured inside the organization deploying it. The framework has not been deployed, instrumented or measured. This is a design contribution, not an empirical one.

Keywords: Token economics, capacity planning, generative AI, cost governance, FinOps, demand forecasting, workload classification, LLM inference cost, agentic systems, drift detection, unit economics

1. INTRODUCTION

An enterprise that ships a generative AI feature takes on a cost line unlike anything else in its infrastructure budget. Nobody provisions it in advance. Nobody reserves it. It does not scale with any countable unit that somebody forecasts. It is created one call at a time, by a model deciding at the moment of the request how much work the task needs.

What follows inside the organization is predictable and close to universal. Finance receives an invoice. Engineering explains it. Nobody predicted it.

The one-line diagnosis is that token cost is a capacity problem being managed with expense-report tooling.

This is worse than a reporting inconvenience, because several things go wrong at once and each one is invisible on its own.

Pilots multiply. Teams get their own keys and no central list exists of who is calling what. The shape of usage drifts. A feature launches with short prompts. Then it picks up a longer set of standing instructions, then some worked examples, then a step that looks documents up and pastes them in. No code review catches it, because nobody reviewing a wording change thinks of it as a cost change. Agentic workflows multiply calls. In an agentic workflow the model is allowed to take several steps on its own to finish a task, so cost stops tracking traffic and starts tracking



how hard the model decides the job is. Retries pile up quietly. With no limit on retrying, one failure downstream can multiply spend before anyone even notices the failure. And nobody owns unit economics, meaning what one unit of the thing costs to deliver. Token cost is not tied to a per-customer number the way compute is, so the question of whether a feature costs a sensible amount for what it produces has no owner.

Finance notices first, and by then it is late. Engineering teams watch latency and error rates. Cost is rarely instrumented as an operational metric at all. The first real signal is the monthly invoice, which arrives weeks after the behavior that caused it. By then several more changes have shipped on top, and tracing a single line item back to one prompt change is genuinely hard. The underlying principle is old and well evidenced in software. The later a problem is found, the more it costs to fix [1].

The organizational damage outlasts the money. Finance stops trusting engineering's numbers and starts gating budgets defensively, which slows down work that was never the problem. Teams get pushed to cut the cost of features already running in production, which is harder and riskier than designing with limits from the start. And leadership loses the ability to decide whether to scale something or pause it, because the data arrives too late and too lumped together to act on.

This paper specifies a framework for forecasting and bounding that cost before it is spent. Section II explains why existing tooling does not carry over. Section III sets out what actually drives the cost and how each driver is measured. Section IV specifies what has to be recorded and where. Section V gives the six-step cycle. Section VI gives the control model. Section VII deals with the question that decides whether any of this survives contact with a real engineering organization. Section VIII states where it breaks down, including two limits that cap what the approach can ever achieve.

2. Why existing tooling does not transfer

A. What organizations do today

There are three common postures, and usually all three are running at once in different teams. The first is invoice-reactive. The provider console or the bill is the main signal, and it gets explained after the month closes. The second is tool-repurposed. An existing cloud cost platform gets pointed at model spend because it is already installed, and it reports that spend against accounts and tags. The numbers are accurate and they answer no useful question. The third is guardrail-improvised. A token ceiling appears on one path, a rate limit on another, a cheaper model on something high-volume. Those are real controls. They are individual choices rather than an organizational position.

What is missing almost everywhere is a forecast. Very few organizations can say ahead of time, with reasoning they would be willing to defend, what a feature will consume once a given number of people are using it.

B. The five assumptions token workloads break

Figure 1 sets them out.

Why cloud cost tooling does not transfer

Five assumptions underneath cloud cost management. Token workloads break every one of them.

cloud cost tooling assumes	what token workloads actually do
You provision before you consume	tokens are never provisioned; consumption is generated at inference time, so there is no reservation event to attach governance to
Cost scales linearly with demand	agentic fan-out breaks this structurally; cost scales with how hard the model decides the task is, not with traffic
The billable unit maps to an infrastructure object	what needs governing is a prompt template, an agent, a user journey, none leaves an infrastructure footprint to tag
Configuration changes trigger cost changes	the expensive changes are content changes that ship through normal code review and trip no alarm
History predicts the future	features, models, prompt patterns and user behavior are all moving at once
The composite. Cloud tools report what was spent on things provisioned in advance. Token workloads need a statement of what will be consumed by something generated at run time by model behavior. That is a modeling gap, not a reporting gap, which is why pointing an existing cost platform at LLM spend produces something accurate and useless.	

FIG. 1. Why cloud cost tooling does not transfer.

Cloud cost management grew up around a particular cost shape, the one that appeared when compute became something you rent by the hour [2]. Five assumptions sit underneath it and hold it up. You provision before you consume, and reserving capacity is a decision someone reviews, which gives governance something to attach to. Cost rises in step with demand. The thing you are billed for maps to an infrastructure object you can tag. Changing a configuration is what changes cost. And the past predicts the future.

Token workloads break all five. Consumption is created at the moment of the request, so there is no reservation to review. Agentic work fans out into many calls, which breaks the straight-line relationship between demand and cost. What actually needs governing is a prompt template, an agent or a user journey, and none of those leaves an infrastructure object behind to tag. The changes that cost the most are wording changes, and they ship through ordinary code review. And features, models, prompt patterns and user behavior are all moving at the same time, so history is a poor guide.

Put together: cloud tools report what was spent on things provisioned in advance. Token workloads need a statement of what will be consumed by something the model generates at run time. That is a modeling gap, not a reporting gap.

C. What is actually new here

Being precise here matters, because applying FinOps to AI is already a circulating idea and is not original. FinOps is the practice of treating cloud spend as something engineering manages day to day rather than something finance reconciles at the end of the month.

The proposal here is narrower. Treat token consumption as a capacity resource with demand curves that can be forecast, and bring across three practices that are standard in infrastructure capacity planning [3], [4], [5] and missing from token cost management.

Model demand by type of work rather than by team. Break consumption into repeatable units defined by how complicated the prompt is, how much retrieved text gets pasted in, and how many steps the model takes. Forecasts then get built up from behavior instead of extrapolated from a history that keeps changing shape. Measure normal consumption for each type and alert when it moves. This is the same idea as alerting on rising CPU before an outage [6], [7], and it is the control that catches prompts quietly growing longer. Treat the number of steps as a forecasting variable in its own right, modeled as a range with a top end rather than left buried inside a single API cost line.

None of those practices is new. They are decades old for compute and storage [3], [8]. What is new is recognizing that token consumption is a resource they legitimately apply to, working out the breakdown that makes token demand modelable at all, and treating the depth of agentic work as something forecastable rather than open-ended. The distinction that matters is between tracking AI spend, which is now common, and forecasting and bounding it before it is spent, which is not.

3. What drives the cost

Figure 2 gives the drivers and the instrumentation that follows from them.

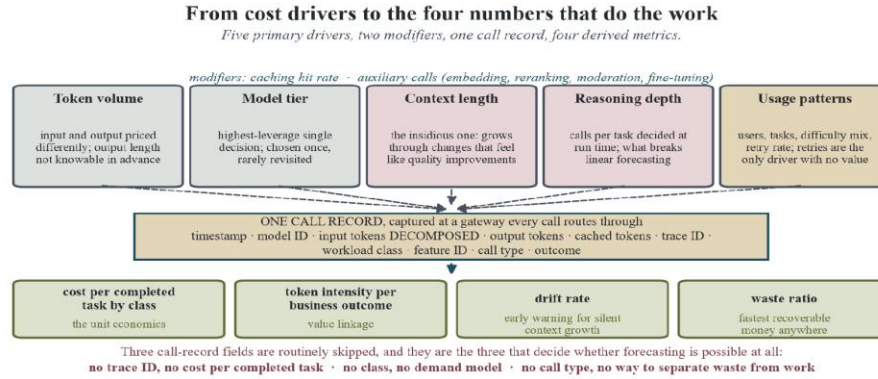


FIG. 2. From cost drivers to the four numbers that do the work.

TABLE I. Five primary drivers, two modifiers, and how each is measured.

Driver	Why it matters	How it is measured
Token volume	Text going in and text coming out are priced differently, and output is more expensive at essentially every provider. So two workloads with the same total volume can have very different bills depending on the ratio, and how long the output will be is not knowable before the call. How much a model actually consumes to answer is an active research question in its own right [9]	The counts of prompt and completion tokens come back on every call. This is the most reliable data available and needs no estimating. The usual failure is simply not logging the response envelope
Model tier	Often the single highest-leverage decision, because the price gap between the cheapest and most expensive model in a provider's range is wide. The governance problem is that the choice gets made once, early, by someone optimizing for output quality, and then never revisited as cheaper models get better	Log which model was used on every call, and track the mix weighted by cost rather than by number of calls. What share of the money went to the premium model, not what share of the calls. Those two numbers often tell opposite stories
Context length	The most insidious driver, because it grows through changes that feel like quality improvements. Standing instructions accumulate as edge cases get patched. Worked examples get added for consistency and then get paid for on every call forever. The number of documents pulled in by retrieval gets tuned to improve answers, with cost as a side effect nobody looked at.	Split the input tokens four ways and log each separately: standing instructions, worked examples, retrieved documents, conversation history. Total input tokens on its own is a useless number. This is the highest-value instrumentation in the

	And in a chat interface the conversation history keeps growing	whole stack, because it turns one opaque figure into four that can be governed independently. No provider gives you this split
Reasoning depth	This is what breaks straight-line forecasting. The number of calls it takes to finish one task is decided at run time by the model's read of how hard the task is [10], [11]	One shared identifier stamped on every call belonging to the same job, usually called a trace ID. Without it, agentic systems show up as an undifferentiated flood of calls and the cost of finishing one task cannot be worked out. With it, you can see the spread of step counts, and particularly the worst cases, which is where the damage happens. This has to be designed in. Adding it later is substantial work
Usage patterns	The demand side: how many people are active, how many tasks each runs, what mix of easy and hard work they bring, and how often things fail and get retried. That last one matters because it is the only driver that can spike while producing nothing of value	Application telemetry joined to the cost data. Retries have to be marked as retries at the point the call is made, or there is no way to tell them apart from real work
Caching (modifier)	Caching means reusing part of a previous request instead of paying for it again, which makes the hit rate a genuine cost variable rather than only a speed one	The count of cached tokens and the hit rate, taken from the API response and from the local cache layer
Auxiliary models (modifier)	The supporting models: ones that turn text into numbers for search, ones that reorder search results, ones that screen content, plus any fine-tuning fees. Each is small, they run at high volume, and they get left out of the accounting entirely because they do not feel like the AI feature	The same fields, applied to every one of those call sites too

A. Measurement traps

There are five, and each one quietly breaks a comparison without telling you. Tokenizers are not the same across model families, so normalize to cost rather than tokens when you compare providers. Treat output length as a distribution, because a single average will mislead you. Record the time the stream finishes, not the time the request starts. Give cached tokens their own field, or they will drag your cost-per-token average off. And store raw token counts with prices applied at query time, because anything built on fixed dollar figures goes stale the moment a rate card changes, and nothing warns you.

4. Where measurement lives

There are three options, and only one holds up at scale.

Provider consoles report totals, they lag, and the most they can tell you is which API key spent the money. That is enough for accounting and no use for governance.

Tagging each call site, meaning each individual place in the code that calls the model, gives you rich context. But it also decays quietly. Every new call someone adds without the tagging is spend you stop seeing.

A gateway that every call routes through is the right answer. A gateway is a single internal service that sits between the application and the model provider. It captures everything because of how it is built rather than because

everyone remembered, it enforces one record format, and it gives you a natural place to put controls later. Its one weakness is that it does not know why a call was made, and that is solved by requiring the caller to pass that context along with the request.

TABLE II. Required fields on every call record.

Field	What you lose without it
Timestamp	Any analysis over time
Model ID	The cost-weighted view of which models the money went to
Input tokens, split into standing instructions, worked examples, retrieved documents, history	Prompts growing longer becomes invisible, and four separately fixable levers collapse into one opaque number
Output tokens	The ratio of text in to text out
Cached tokens	Cost-per-token averages get distorted
Trace ID	The cost of finishing one task
Workload class	Any demand model at all
Feature ID	Any way to trace spend back to a business owner
Call type: primary, retry, auxiliary	Any way to separate waste from work
Outcome	The ability to work out how much spend produced nothing

Three of these get skipped routinely, and they are the three that decide whether forecasting is possible at all. They are marked in bold above.

Four derived numbers do most of the work. The first is cost per completed task, by class. That is the unit economics, and it is the only number that connects spend to something of value. The second is how many tokens it takes to produce one business outcome. The third is drift rate, meaning how much the average tokens per call inside one class has changed since last period, which is the early warning that prompts are quietly getting longer. The fourth is waste ratio, meaning retries, failures and rejected outputs as a share of spend. That last one is usually the fastest money anyone can recover.

5. The six-step cycle

Figure 3 gives the cycle.

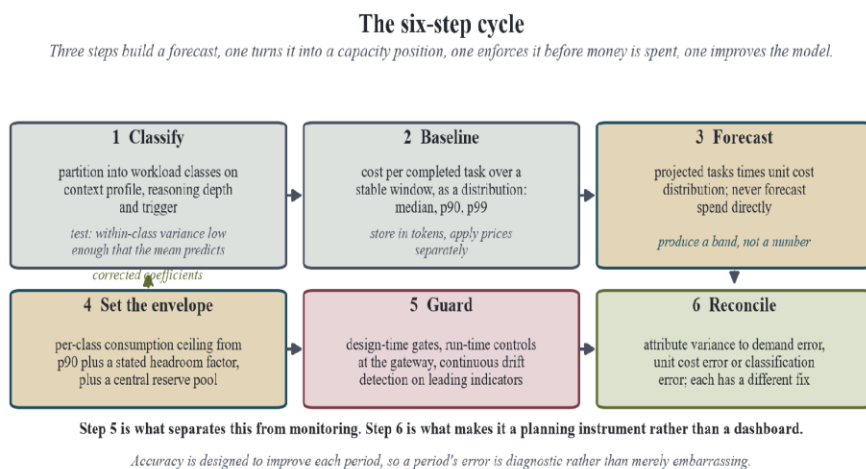


FIG. 3. The six-step cycle.

Step one, classify. A workload class is the unit everything else rests on. It is a set of calls that behave similarly on cost, and it is defined along three axes. How the context behaves, meaning fixed, topped up by retrieval, or accumulating as a conversation goes on. How deep the reasoning goes, meaning one shot, a fixed chain of steps, or a variable number of steps the model decides for itself. And what triggers it, meaning a user, a scheduled batch, or a background process.

Crossing those three gives a small, workable set of classes: one-shot work on a fixed prompt such as classification; question answering with documents pulled in; multi-turn conversation where the history builds up; pipelines with a known number of steps; agentic work of variable depth; and background batch work such as generating embeddings.

The test that governs all of it is that cost variation inside a class has to be small enough for the average to mean something. If the spread is wide, it is not one class, and it should be split. That test is what keeps the taxonomy honest, and ignoring it is the most common way implementations go wrong.

Step two, baseline. For each class, measure the cost of completing a task rather than the cost of a call, over a stable period, and record the spread rather than just the average. Record the middle value, the level nine out of ten tasks stay under, and the level ninety-nine out of a hundred stay under. For agentic classes, record the spread of step counts too, because that is what drives the worst cases. A workload that has not launched yet gets baselined from a pilot that was instrumented on purpose, rather than from an estimate, because a small measured run beats any amount of reasoning about what consumption ought to be. Baselines are recorded in tokens with prices applied separately, so a price change re-prices the model instead of invalidating it.

Step three, forecast. Forecast the drivers separately and put them together. Never forecast spend directly. For each class, multiply projected tasks by the spread of unit costs, where projected tasks breaks down into how many people are active, how many tasks each runs, and what mix of classes they use.

Two disciplines apply. Produce a band, not a number. Run low, expected and high adoption against both the middle and the nine-in-ten unit cost, and you get a defensible range with the assumptions written down. And for variable-depth work, forecast the expected value using the average step count but size the ceiling against the nine-in-ten case, because the worst agentic runs are where budget damage actually happens.

Step four, set the envelope. Turn the forecast into an approved consumption ceiling per class per period, rather than one number for the whole organization. That ceiling is the capacity position, and finance and engineering own it jointly. It has three parts. Per-class ceilings set from the nine-in-ten case rather than the middle, which is the same headroom logic capacity planners have always used on compute [3], [4]. An explicit allowance for adoption being uncertain, written down as an actual number rather than left inside somebody's optimism. And a reserve pool held centrally, so a team approaching its ceiling has somewhere to go. Without that reserve the only options are stopping work or quietly overspending, and organizations reliably pick the second.

Step five, guard. This is Section VI.

Step six, reconcile. When the period closes, work out which of three things caused the variance, because each has a different fix. Demand error means volume was not what was projected, so the adoption model needs work. Unit cost error means the cost of a task drifted inside a class, so treat it as drift and tighten the checks that run before code ships. Classification error means the variance inside the class is too wide to predict at all, so the class needs splitting. Feed the corrected numbers into the next cycle.

That last step is what makes this a planning instrument rather than a dashboard. Accuracy is meant to improve, and each period's error tells you something rather than just being embarrassing.

A. On expected accuracy

It will not be high early on, and saying so is better than overclaiming. What the framework gives you is a bounded range with the sources of error named, in place of an unbounded unknown. That is the same standard traditional capacity planning meets on workloads it has not seen before [3], [5]. The software cost estimation literature reached the same conclusion about its own field. Estimates for unfamiliar work are poor in absolute terms and still good enough to support the decisions they exist to inform [12]. That is the bar here too: whether to scale a feature, what to set aside, and where to intervene first.

6. Guardrails

Figure 4 gives the control model.

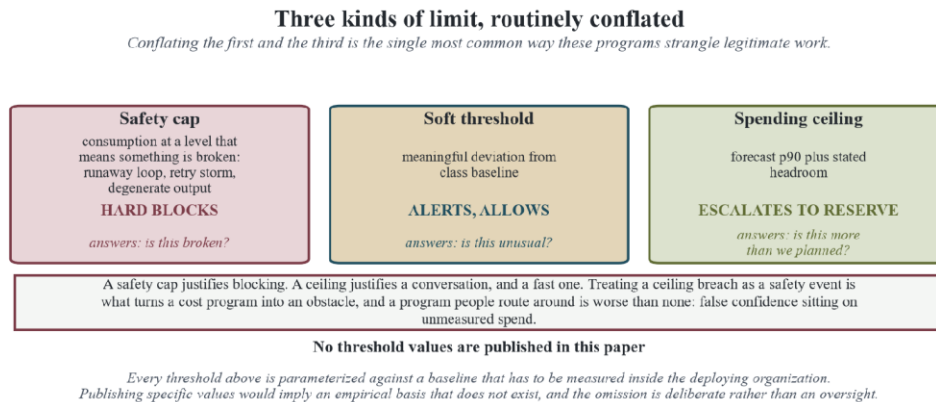


FIG. 4. Three kinds of limit, routinely conflated.

A. Three limit types

A safety cap is set at the level where consumption means something is broken: a loop that will not stop, a storm of retries, output that has degenerated. It blocks the call outright.

A soft threshold is a meaningful move away from the baseline. It raises an alert and lets the work through.

A spending ceiling is the forecast plus headroom. It escalates to the reserve and does not block anything.

Confusing the first with the third is the single most common way these programs end up strangling legitimate work. A safety cap answers the question of whether something is broken. A ceiling answers the question of whether this is more than was planned for. The first justifies blocking. The second justifies a conversation, and a quick one.

B. Controls at three points in time

Before code ships. Any new class, or any material change to a prompt, gets a cost estimate reviewed the way a performance regression would be, which puts the check in the delivery pipeline where automated quality gates already live [13]. Each class declares a maximum context budget, treated as a contract. Agentic work declares a maximum number of steps and how deep it may recurse. Choosing a model tier comes with a written justification that gets revisited on a schedule.

At run time, at the gateway rather than left to individual call sites. Hard caps on how long the output can be, how much context can be sent, and how many steps one task may take. A retry budget per task with a circuit breaker, meaning something that stops retrying once a set number of attempts have failed. That is the single highest-value control, because retry storms are pure waste and they multiply faster than any other failure mode. Rate limits per class. And automatic routing of high-volume simple traffic down to a cheaper model [14].

Continuously, on early signals. Average tokens per call moving outside a band around the baseline. The cost-weighted mix of models shifting. Waste ratio crossing a threshold. And burn-rate pacing, meaning a running projection of where consumption will land by the end of the period compared with the ceiling. That last one is what makes the framework predictive rather than backward-looking. It fires mid-period based on the trajectory, weeks before the invoice that would otherwise be the first anyone hears of it.

C. On threshold values

None are given in this paper, deliberately. Every threshold described above is set relative to a baseline that has to be measured inside the organization deploying it. Publishing specific numbers would imply an evidence base that does not exist, and saying that plainly is better than leaving it looking like an oversight.

7. Controlling cost without killing legitimate usage

This is what decides whether the framework survives. A framework that slows down valuable work gets routed around, and one people evade is worse than having none at all, because it leaves false confidence sitting on top of spend nobody is measuring. Three questions decide it.

A. How spend is attributed

Attribution has to happen at the application layer, because infrastructure tags cannot see the units that matter here. The chain runs from a call, to a task, to a workload class, to a feature, to a business owner, and each link is a field on the call record rather than something reconstructed afterwards.

Two choices decide how that lands politically. Do showback before chargeback. Showback means telling a team what they spent. Chargeback means moving the money onto their budget. Introducing chargeback before people trust the data produces immediate gaming: teams argue about methodology instead of managing consumption, and at worst they route around the gateway specifically to avoid being attributed at all. Run showback until the numbers survive being challenged, then move to chargeback only if the money genuinely needs to move. Many organizations never need that step.

Accept some imprecision on shared costs. Shared search indexes, embedding pipelines and the gateway itself resist being allocated cleanly. Split them by usage share, leave the genuinely shared part central, and resist the pull

toward more precision. Attribution needs to be good enough to drive decisions, not good enough to win an argument. Over-engineered allocation burns more in disputes than it recovers in savings.

B. Where limits are set

Where the cost is shaped, not where it is spent. Context budgets belong where the prompt is assembled. Step caps belong where the steps are orchestrated. Rate limits belong at the feature boundary. Ceilings belong at the class level. A limit placed downstream of the decision that created the cost can only reject work that was already designed to fail.

C. Six commitments that keep it from becoming an obstacle

Alert first, block last. The default posture is watching. Blocking is reserved for the safety tier.

Make the exception path fast. The reserve only works if drawing on it takes hours. Speed matters far more than size. A large fund with a two-week approval gets routed around. A modest one with same-day turnaround gets used the way it was meant to be.

Judge spend against value, not against its absolute size. An expensive workload that clears its value bar should pass without friction. A cheap one producing nothing should not get a pass just for being cheap.

Give every team a small exploration allowance that sits outside the ceiling and needs no approval. It costs little, it tells you a lot, and it produces exactly the pilot data needed to baseline a new class.

Block waste, escalate volume. Retry storms and runaway loops get blocked. Legitimate high volume gets escalated to a human. The difference is whether the spend is producing anything, and that is almost always visible from the call type and the outcome field.

Fail open, with an alarm. If the governance layer is down or unsure, let the call through and raise an alert. A cost control that causes a production outage loses its mandate permanently, and no amount of savings buys that back.

D. Sequencing

Whether this gets adopted is decided by the order you do it in. Lead with the controls nobody will defend: circuit breakers on retry storms, obvious cases of an expensive model doing simple high-volume work, and abandoned pilots still spending money after the team moved on. Those cut waste without constraining a single deliberate decision, and they establish the program as something that recovers money rather than something that rations it.

The pre-ship checks and ceiling enforcement do create real friction. They land completely differently once there is a track record of finding waste. Trying them first inverts the politics, and that is how these programs die.

8. Where it breaks down

Figure 5 gives the limits.

Where the framework breaks down

Stated as limits rather than as future work, because two of them bound what the approach can ever do.

limit	why it bites	what it costs the framework
Open-ended agentic work	cost is driven by the specific input, not the class, so the heavy tail lives inside a single request type and splitting the class does not fix it	degrades from forecasting to bounding: cap steps, hold inside a ceiling. A weaker result, stated as such
Cold start	a baseline requires observing real traffic; a workload that has never run has nothing to observe	unsolved. Cross-organization benchmarks would solve it, and that is a job for an industry body
Baseline expiry	every baseline is anchored to a specific model. Swap the model and the baseline is void	THE REAL LIMIT: convergence rate must beat model churn rate. Whether real churn permits convergence is an open question
Measures cost, not value	governance design assumes you can quantify what a task is worth; most organizations cannot	value calibration collapses back to raw spend thresholds, the thing it was meant to avoid
The adoption paradox	needs a gateway, tracing, and finance and engineering jointly owning the numbers	an organization with all three probably manages cost tolerably already. Those who need it most are least equipped to run it

Inside a period, a bad forecast and a genuine anomaly are indistinguishable in the burn-rate signal. That is a hole, not a caveat.

FIG. 5. Where the framework breaks down.

Open-ended agentic work. The whole approach rests on sorting work into classes whose members cost roughly the same. Open-ended tasks such as researching a question or resolving a support ticket refuse to cooperate. What drives the cost is the specific request rather than the class it belongs to, which produces a long tail that splitting the class does not fix, because the variation lives inside a single request type. For that kind of work the approach drops from forecasting down to bounding. You can still cap the steps and hold the work inside a ceiling. That is a weaker result and it is stated as one.

Cold start. Setting a baseline means watching real traffic, and something that has never run has no traffic to watch. This is unsolved.

No objective test for how many classes there should be. Too few and the averages mean nothing. Too many and each one holds too little data while costing real effort to maintain. Right now that is judgment, not method.

Model economics change in three ways, and the framework only handles one of them. Storing raw token counts and applying prices separately handles price changes cleanly. Capability change does not announce itself. When a cheaper model becomes good enough for work currently running on an expensive one, the original choice is now wrong, and catching it depends on a review cadence that people let slip. Capability arriving at ever lower price points is the predictable result of how model performance scales with training investment [15], which makes this a recurring condition rather than a one-off correction. Architectural change breaks the measurement itself rather than the arithmetic. Reasoning models that spend hidden tokens thinking before they answer break the assumption that what you can see going in and coming out is the whole cost. Caching behaves differently from provider to provider. And some providers bill by request or by character rather than by token at all.

Baseline expiry is the real limit of the entire approach. Every baseline is tied to one specific model. Change the model and the baseline is void, and the observation period starts again. So the rate at which the model converges has to beat the rate at which the organization swaps models. An organization changing models every quarter, where convergence takes two or three cycles, will never hold a mature forecast. Whether real-world churn leaves enough room to converge is an open question, and it is the first thing worth measuring.

A. What happens when a forecast is badly wrong

The two directions fail very differently.

Too low, and ceilings get breached early, the reserve drains, and escalations stack up. The damage is not financial. It is that the escalation path saturates, teams start experiencing cost governance as an obstacle, and they

route around it. That is governance collapse rather than budget collapse, and it is much harder to recover from. So a defined invalidation procedure is required. Past a certain size of breach, stop escalating, suspend the ceiling entirely, fall back to pure safety limits, and re-baseline. Continuing to treat a broken forecast as authoritative is worse than admitting it is broken.

Too high is more insidious. Money gets reserved that was never needed, and finance quietly decides the model is alarmist. Nobody escalates an over-forecast, so it goes uncorrected far longer, and it erodes credibility in exactly the direction that gets a program defunded.

Underneath both of those: while a period is still running, a bad forecast and a genuine anomaly look identical in the burn-rate signal. You only find out which it was at reconciliation, after the fact. That is a real hole, not a caveat.

B. Scope limits

It measures cost, not value. Much of the governance design assumes you can put a number on what a task is worth, so spend can be judged against it. Most organizations cannot do that, so judging spend against value quietly collapses back into judging it against raw thresholds, which is the thing it was supposed to replace. This is the widest gap between how the framework is designed and what organizations can actually execute.

There is an adoption paradox. The framework needs a gateway every call routes through, tracing across multi-step work, and finance and engineering jointly owning the numbers. Any organization that already has all three probably manages its cost tolerably well already. The organizations that need this most are the least equipped to run it.

It is a cost framework, not an architecture framework. The biggest savings usually come from architectural decisions: cache aggressively, use a cheap classifier instead of a model, or do not make the call at all. This tells an organization where the money is going. It does not tell them whether the AI was the right tool for the job.

The scope is inference bought through an API. Training, the capital cost of fine-tuning, and running inference on your own hardware are provisioned-capacity problems, and ordinary capacity planning already handles those better.

9. What the framework would produce, and what would prove it

Six measurable outputs become available once somebody actually runs it. Forecast error per class. Drift detection lead time, meaning the number of days between a drift alert and the invoice that would otherwise have been the first signal. Waste ratio before and after, which is measurable on day one, before any forecasting maturity exists at all. How often ceilings get breached per period. How stable the classification stays. And how long the exception path takes, from a reserve request to a resolution.

Waste ratio being available immediately is also why it should lead politically. It is the number that earns the mandate for everything else.

What would prove it. Instrument a real multi-team estate, run the cycle for three or more periods, and report forecast error by class. The falsifiable claim is that error inside a class shrinks with each cycle, while a comparison estate with no instrumentation shows no such improvement. Anyone with a gateway and three months could run that study.

A. What would be built next

Five things, ordered by how much each would unlock.

Automated workload classification, deriving cost classes by clustering the telemetry rather than declaring them by hand, which removes the largest manual judgment call in the whole method [16].

Model-migration simulation, replaying stored traffic against a candidate model's measured behavior and pricing, so a migration can be priced before anyone commits to it. This attacks baseline expiry directly.

A way to tell forecast error apart from a genuine anomaly while the period is still running, rather than waiting for reconciliation.

Value instrumentation, connecting cost data to outcome data so that cost per outcome is a real number rather than an aspiration. This is the hardest of the five and the highest payoff.

Anonymized cross-organization cost benchmarks, so a new implementer starts from a reference point rather than from zero. That solves cold start, and it is a job for an industry body rather than any individual practitioner.

10. Limitations

Nothing has been deployed, instrumented or measured. No pilot, no reference implementation, no dataset, no code, no schema built. That governs every claim in this paper.

No threshold values are offered, and the omission is deliberate rather than an oversight, because publishing them would imply an evidence base that does not exist.

The design rests on one practitioner's exposure to enterprise generative AI workloads. That exposure shaped the problem statement and the design. It validates neither.

Two limits cap what the approach can achieve no matter how well it is executed: baselines expiring against model churn, and the inability to tell forecast error apart from an anomaly inside a period. Both are stated in Section VIII rather than pushed into future work, because anyone deciding whether to adopt this needs them before they decide.

The economics described here are current at the time of writing. Provider pricing models, billing units and architectural conventions are all moving, and Section VIII-A sets out which kinds of change the framework absorbs and which ones break its measurement basis.

11. Conclusion

Enterprise generative AI spend is currently governed with tooling built for a cost shape it does not have. Cloud cost platforms report what was spent on resources provisioned in advance. Token consumption is created at run time by a model deciding how much work a task needs, and there is no reservation event for governance to attach itself to.

This paper treats tokens as a capacity resource and brings across three practices that are decades old for compute and storage and missing here: modeling demand by type of work rather than by team, measuring what normal looks like for each type and alerting when it moves, and treating the number of steps a model takes as a bounded range to forecast rather than an unknown to absorb. Around those it specifies a six-step cycle, an instrumentation contract naming the three call-record fields whose absence makes forecasting impossible, and a control model separating a safety cap that blocks from a spending ceiling that escalates, on the argument that confusing those two is how these programs end up strangling the work they were built to protect.

None of it has been run. The contribution is a design specified precisely enough to be built and to be proved wrong, with its two hardest limits stated rather than deferred: that every baseline expires when the model changes, so convergence has to outpace model churn, and that inside a period a bad forecast is indistinguishable from a real anomaly. The study that would settle whether the approach works needs a gateway, three months, and a willingness to report the error honestly.

Note on scope

This is a design contribution. The framework has not been deployed, instrumented or measured. There is no pilot, no reference implementation and no dataset, so no forecast accuracy figures, savings percentages or overrun-avoidance numbers appear anywhere in this paper. The author worked on enterprise generative AI workloads during 2024 through 2026. That experience shaped the problem statement and the design. It is not offered as evidence that the framework works, which would be a different claim entirely. Three things are claimed here. The framework is

specified precisely enough that someone could build it and prove it wrong. Its parts are established practice moved across to a new kind of resource, so what is original is the transfer and the breakdown rather than the techniques themselves. And the underlying problem is visible in public industry practice rather than resting on the author's assertion. No employer, client, product or vendor is named.

REFERENCES

1. B. Boehm and V. R. Basili, "Software Defect Reduction Top 10 List," *Computer*, vol. 34, no. 1, pp. 135-137, 2001. doi: 10.1109/2.962984
2. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50-58, 2010. doi: 10.1145/1721654.1721672
3. D. A. Menasce and V. A. F. Almeida, "Business process choreography for B2B collaboration," *IEEE Internet Computing*, vol. 8, no. 1, pp. 75-83, 2004. doi: 10.1109/MIC.2004.1260702
4. M. Calzarossa, L. Massari, and D. Tessera, "Workload Characterization: A Survey Revisited," *ACM Computing Surveys*, vol. 48, no. 3, pp. 1-43, 2016. doi: 10.1145/2856127
5. S. Islam, J. Keung, K. Lee, and A. Liu, "Empirical prediction models for adaptive resource provisioning in the cloud," *Future Generation Computer Systems*, vol. 28, no. 1, pp. 155-162, 2012. doi: 10.1016/j.future.2011.05.027
6. T. Llorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559-592, 2014. doi: 10.1007/s10723-014-9314-7
7. A. Gandhi, M. Harchol-Balter, R. Raghunathan, and M. A. Kozuch, "AutoScale: Dynamic, Robust Capacity Management for Multi-Tier Data Centers," *ACM Transactions on Computer Systems*, vol. 30, no. 4, pp. 1-26, 2012. doi: 10.1145/2382553.2382556
8. N. Roy, A. Dubey, and A. Gokhale, "Efficient Autoscaling in the Cloud Using Predictive Models for Workload Forecasting," in *Proc. IEEE International Conference on Cloud Computing (CLOUD)*, 2011, pp. 500-507. doi: 10.1109/CLOUD.2011.42
9. S. Samsi, D. Zhao, J. McDonald, B. Li, A. Michaleas, M. Jones, W. Bergeron, J. Kepner, D. Tiwari, and V. Gadepally, "From Words to Watts: Benchmarking the Energy Costs of Large Language Model Inference," in *Proc. IEEE High Performance Extreme Computing Conference (HPEC)*, 2023, pp. 1-9. doi: 10.1109/HPEC58863.2023.10363447
10. L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, pp. 186345, 2024. doi: 10.1007/s11704-024-40231-1
11. W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, "Efficient Memory Management for Large Language Model Serving with PagedAttention," in *Proc. ACM Symposium on Operating Systems Principles (SOSP)*, 2023, pp. 611-626. doi: 10.1145/3600006.3613165
12. M. Jorgensen and M. Shepperd, "A Systematic Review of Software Development Cost Estimation Studies," *IEEE Transactions on Software Engineering*, vol. 33, no. 1, pp. 33-53, 2007. doi: 10.1109/TSE.2007.256943
13. M. Shahin, M. A. Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. 5, pp. 3909-3943, 2017. doi: 10.1109/ACCESS.2017.2685629
14. L. Chen, M. Zaharia, and J. Zou, "FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance," *arXiv preprint*, 2023. doi: 10.48550/arXiv.2305.05176
15. J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, "Scaling Laws for Neural Language Models," *arXiv preprint*, 2020. doi: 10.48550/arXiv.2001.08361
16. Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, D. Y. Fu, Z. Xie, B. Chen, C. Barrett, J. E. Gonzalez, P. Liang, C. Re, I. Stoica, and C. Zhang, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," *arXiv preprint*, 2023. doi: 10.48550/arXiv.2303.06865