WHITE PAPER

The Cost of an Answer

Token cost is decided by how a system is built, long before anyone asks a question

Martin Jack
Founder and CTO, BessWave
September 2026

Executive summary

Every answer a language model produces carries an inference cost, and tokens are the part of it that scales with every question: the fragments of text the model reads on the way in and writes on the way out. A product that answers questions over a company's operational systems pays that cost on every question, from every person, every working day. Across a team and across a year the number stops being incidental. It decides whether the product can be sold at a flat price per person or whether every question has to be metered.

This paper makes one argument. Token cost is not primarily a pricing decision or a choice of model. It is an outcome of architecture, and most of it is settled before a question is ever asked. A system that answers by placing as much data as it can in front of the model will spend heavily on every question and will keep spending on the same question tomorrow. A system that works out what a question actually needs, computes what can be computed, and retrieves only the records that bear on the answer spends a fraction of that, on the same question, using the same model.

BessWave is built the second way. The material placed in front of the model is sized to what a question requires rather than to how much data happens to exist. Relationships between records across systems are modeled in advance, so the records that bear on an answer can be gathered precisely rather than approximately. And the preparation that makes this possible happens ahead of the question rather than being repeated each time one is asked.

The same choices improve the answer. A smaller, better-chosen context is not merely cheaper than a large one; it is more likely to be right. Cost and quality pull in the same direction here, which is what makes this worth building around rather than optimizing later.

> *Token cost is not a line item to negotiate. It is a consequence of how the answer is assembled.*

What you are actually paying for

Start with the mechanics, briefly, because the rest of the paper depends on them. A token is roughly a fragment of a word. A language model is charged for the tokens it reads and the tokens it writes. In a product like this one, the input is where the volume sits. The question a person types is short. The answer they read back is short. The operational data placed in front of the model so that the answer is grounded in something real is not short at all. Output tokens are often priced higher, but there are far fewer of them.

That asymmetry is the whole point. The largest lever on the cost of an answer is how much data the system decides to show the model. Model choice matters too, and can matter a great deal. But architecture decides how large the grounding burden is before model pricing enters the picture at all.

It follows that architecture largely determines what each answer costs, and usage determines how often that cost is incurred. Two systems can answer the same question, for the same person, with the same model, and differ by a wide margin in what that question cost, entirely because one of them was more disciplined about what it put in front of the model.

> *The question is short. The answer is short. The grounding is not. That is where the money goes.*

Why the obvious approach gets expensive

The obvious approach is obvious for good reasons. Context windows have grown large, connecting a system is easy, and putting more data in front of the model is the fastest way to get a demonstration working. It is a reasonable place to start. It is an expensive place to stay. Two patterns account for most of the waste, and they are worth taking one at a time.

Put the data in the context window. When the window is large enough to hold a quarter of task records, the temptation is to hold a quarter of task records. The cost then scales with the size of the data rather than the difficulty of the question, and it is paid again on the next question and the one after that. There is a second problem beyond the bill. Models do not use a long context evenly. Material placed in the middle of a long input is recalled less reliably than material at either end, and performance degrades as the input grows.[1] The expensive path is often the less accurate one.

Retrieve broadly and hope. The standard correction is to retrieve rather than load everything: find the passages most similar to the question and pass those along. This is cheaper, and for finding a document or a single fact it works. The difficulty is that similarity is an imprecise instrument, and the usual response to imprecision is to ask for more of it. Fifty passages instead of ten, in the hope that the right one is somewhere in the set. That is a large volume of near-miss material paid for on every question. On connected questions it is worse than imprecise, because the records that matter may not resemble the question at all. A contract does not look like a question about slipping deliverables, even when the contract is the reason the answer matters.

The common thread is that each pattern pays for material that did not change the answer. That is the definition of an avoidable cost, and it is avoidable through design rather than through discipline on the part of the person asking.

> *Every token that did not change the answer was paid for anyway.*

Most of the cost is decided before the question is asked

This is the idea the rest of the paper rests on. Work done once, offline, is amortized across every question that follows. Work deferred to the moment of the question is paid for again each time

[1] Nelson F. Liu et al., “Lost in the Middle: How Language Models Use Long Contexts,” Transactions of the Association for Computational Linguistics, 2024; arXiv:2307.03172. The study finds that model performance is highest when relevant information sits at the beginning or end of the input context and degrades when it sits in the middle, and that performance falls as the input context grows longer. The finding is cited here to indicate that a larger context is not automatically a better one, not as a measurement of any specific system.

someone asks. Where a system chooses to do its work is therefore a cost decision as much as an engineering one.

BessWave does the heavy work when data is brought in rather than when a question arrives. Connected sources are prepared and indexed for retrieval, and the operational graph of how their records relate is built. That preparation does not have to be repeated when someone types a question, or for the next person who asks something similar.

A system that skips this step is not saving anything. It is deferring the same work to the most expensive possible moment and then paying for it repeatedly.

How BessWave brings the cost down

What follows describes what the platform does, not the logic by which it decides. That distinction is deliberate. The principles are worth explaining, and the specific rules are not ours to publish.

The input is sized to the answer, not to the data. This is the commitment the rest of it serves. What reaches the model is governed by what the question requires, so the volume behind an answer does not grow simply because the underlying data has grown. That property is the difference between a cost that rises with a customer's success and one that does not.

Relationships between records are modeled in advance. Similarity search over-fetches because it has no way of knowing which records are connected to which. BessWave models how records relate across systems, so the records that bear on an answer can be gathered precisely rather than approximately. Precision is what keeps the context small, and it is also what makes the answer better.

Graph-based retrieval has shown advantages over similarity search alone where an answer has to be assembled from across a body of material.[2] BessWave applies the underlying principle to a different problem: a graph of the explicit relationships between operational records, followed rather than guessed at.

It is worth being precise about where this helps most. A straightforward lookup often does not need relationships followed at all. The value becomes pronounced on connected questions, where the answer depends on following links across records and systems, and those are the questions where broad retrieval over-fetches worst. The place it saves the most is the place it is most needed.

The model is one component, and it is replaceable. BessWave can move between providers and between model sizes. Heavier models are used where the work warrants them rather than everywhere by default, and falling model prices are captured as they arrive instead of being locked out by a design that assumed one fixed model.

Cost per answer is measured. It is tracked as an engineering metric rather than discovered on an invoice a month later. Nothing stays under control as a product grows unless someone is watching the number.

> *The model writes the answer. The architecture decides what the answer costs.*

[2]Darren Edge et al., “From Local to Global: A Graph RAG Approach to Query-Focused Summarization,” Microsoft Research, April 2024; arXiv:2404.16130. The paper reports that graph-based retrieval improves the comprehensiveness and diversity of answers over naive vector retrieval on broad, corpus-wide questions, at a lower token cost than passing the full source text. Its setting is summarization over text collections rather than traversal of structured operational records, so it is cited here for the general principle and not as a measurement of BessWave’s architecture.

The same question, two paths

An illustration makes the difference concrete. The example below uses one kind of operational data; a different operation would ask a different question across different systems. The structure of the comparison is what carries over.

Take the question: how many tasks were logged against each project last quarter, and which of those projects have milestones already past their due date?

The first path. Retrieve the quarter's task records and the milestone schedule, place both in front of the model, and ask it to group the tasks by project, count them, match milestones to projects, and compare dates. The input grows with the number of tasks, so a busy quarter costs more to ask about than a quiet one, which is a strange property for a question that has not changed. The join between milestones and projects is left to the model to work out in prose. And next quarter the whole thing starts again from nothing.

The second path. What reaches the model is a compact set of results rather than the underlying records: a short table of task counts by project, and the milestones already past their due date with the projects they belong to attached. The model puts that into plain language and names the systems it drew from. The size of the input is a function of how many projects are in the answer, not how many tasks were logged.

The second path costs less because the model is asked to do less, not because a cheaper model was used. The same substitution would reduce the cost of the first path too, and the first path would still be the expensive one.

> *In the first path, cost scales with how busy the quarter was. In the second, it scales with how many projects are in the answer.*

What this means for a buyer

Most of the above is invisible to the person asking a question, and it should be. What it buys them is straightforward.

A predictable price. BessWave is sold at a flat price per person per month, published on the website. There is no per-question charge and no consumption meter. That is only a sustainable way to sell a product like this if the unit cost of an answer is controlled, which is why the architecture is a commercial matter and not only a technical one.

No reason to ration questions. Metered pricing quietly teaches a team to ask less, which is the opposite of what an operations intelligence tool is for. The value of BessWave comes from people asking the connected questions they used to leave unasked. A pricing model that penalizes that would undo the product.

A price that does not move with usage. Because the price is set per person rather than per question, a team that asks more this month does not see a larger invoice. Keeping that true is an engineering matter, and it is the reason the work described in this paper gets the attention it does.

Falling model prices become headroom. Because the model is swappable, a cheaper or better model can be adopted without redesigning the system around it. That shows up as margin and as capability rather than as a price change for the customer.

Where this matters less, and what we are not claiming

It is worth being clear about the limits of the argument.

If a team asks a handful of questions a month against one small system, token cost is not the problem worth solving, and no architecture will make that fact interesting. The case in this paper applies where questions are frequent, data is substantial, and more than one system is involved. That is also the case where BessWave is worth having at all, so the two conditions travel together.

We are not publishing a cost-per-answer figure or a percentage saving. Any such number would depend on the customer's data volume, their mix of questions, and the model prices in force on the day it was measured, and it would be obsolete quickly. The honest claim is structural: a system that retrieves less to answer the same question spends less on it, and that remains true whatever a token happens to cost this quarter. If we publish a number later, it will be from our own benchmark, with the dataset and the question set described.

Nor is cheapest the objective. Some questions deserve a heavier model and a wider retrieval, and deciding the path in advance exists to spend where spending is warranted rather than to spend as little as possible everywhere. The aim is that no question costs more than the answer required.

How the two approaches compare

The distinction reads most easily side by side. The comparison is between the common approach of putting large volumes of data in front of the model, whether by loading it directly or by retrieving broadly, and the approach described here.

Dimension	Context stuffing and broad retrieval	BessWave
What sets the size of the input	The size of the data	The shape of the answer
Counting and trend questions	Rows are retrieved and the model adds them up	The model is given results to put into words, not records to work through
Connected questions	More material is retrieved to compensate for imprecision	Relationships are traversed in the operational graph
Preparation work	Repeated at the moment of every question	Done once, ahead of the question
Model choice	Fixed by what the design assumes	Swappable, and matched to the work
Cost visibility	Discovered on the invoice	Tracked per answer as an engineering metric

The shift

The conversation about language models has moved on from what they can do to what they cost to run every day, at scale, for every person in a company. That is the question a buyer is really asking when

they ask about pricing, and it is the question a product has to answer in its architecture rather than in its rate card.

The systems that last will be the ones where the unit cost of an answer was engineered rather than absorbed. Placing more data in front of a larger model is the fastest route to something that looks impressive and the slowest route to something that can be sold at a flat price. The alternative is not a cheaper model. It is a system designed so that the material behind an answer is set by the answer itself, and so that the work making that possible is done ahead of the question rather than repeated with each one.

The clearest way to evaluate this is on your own data: connect the systems you already run, ask the cross-system questions your team has been answering by hand, and look at what comes back and how quickly. The economics described here are the reason those answers can keep arriving without the price changing.

Martin Jack
Founder and CTO, BessWave
martin.jack@besswave.com | besswave.com