Speedshop Tune: Fin

@nateberkopec

Prepared with care for Intercom

@nateberkopec

Contents

- Outcome: Reduce Median Fin Latency to Less Than 10 Seconds
  - Recommendation: Create an internal benchmark for vendor/model selection. (1-2 seconds)
  - Recommendation: Constrain output tokens in certain scenarios (2-4 seconds)
  - Recommendation: Constrain/eliminate modifications to model output, prefer only input constraints (2 seconds)
  - Recommendation (key): Constrain Ruby blocking time pre-answerbot to 1 second (2-5 seconds)
  - Recommendation: Implement CQRS wherever possible pre-answerbot (1 second)
  - Recommendation: Communicate the latency price of workflows or other "turned on features"/config to customers (not the base case) (0-1 seconds)
  - Recommendation: Direct "simple/meaningless" queries (often used in demos) to simpler models (2-4 seconds)
- Outcome: Improve Perceived Performance
  - Recommendation: Laboratory test various latency experiences (blind, recall based testing, surveys).
  - Recommendation: Respond with intermediate or precomputed answers
  - Recommendation: Respond with something to read/look at in the meantime to keep user attention in the window (support articles, etc) to increase user engagement
  - Recommendation: Create a determinate progress bar based on past performance
- Summary of Recommendations

Hello Intercom!

Thanks for having me take a look under the hood (again!). I've identified several areas for investment in performance on Fin. This was an interesting project, which is a bit different from my usual "tell me what our Rails app is doing wrong!". The constraints imposed by the Intercom product and the nature of an AI-powered support assistant are intriguing!

There's a pathway towards halving median latency from about 20 seconds to about 10 seconds.

However, there are dangers on the road ahead. A lot of people are excited to work on performance, and they're getting a lot of work done. That's great. But, I'm seeing:

1. **No common goal**. There's currently not a common ruler that every one agrees on for "this is how we measure how slow Fin is."
2. **Microbenchmarking**. Focusing on a 50% latency improvement to something sounds really good, until you realize that it takes 1% of the total end-to-end latency. It's easy to fall into micro-level work that doesn't

make a difference to the big picture.

3. **Too much tribal knowledge**. I have the luxury of having access to everyone, but most people don't. If *I* can barely figure out answers to particular questions through interviewing specific people, what hope does a fresh junior have on a random Fin-focused product team?

These three conditions will lead to what feels like a lot of busy people getting a lot of things done, but in 6 months the product will feel just as slow as it does today. Software performance is a hard engineering discipline, which requires a systematic approach that is accessible to every engineer from the freshest to the most senior.

While I've got several dozen individual recommendations in this report, it boils down to a few themes:

1. **Using hackathon/AI-generated tools to fill gaps in understanding**. While I want to use Honeycomb and `deep-traces` as the foundation for future work, it's got a lot of gaps. We can fill these in with short "hackathon" projects, probably heavily AI-codegen-assisted, that provide internal tools which help us to understand Fin latency.
2. **Create common measurements and language**. I frequently found myself in Slack just wondering what people were talking about, as we used different words to refer to the same feature. Without an agreement on how we'll measure impact, we can't do work that will matter.
3. **Creating an engineering workflow**. Performance is engineering. Engineering is the science of meeting requirements while respecting constraints. We can use data to help us do both: meet requirements, and set constraints. We don't have to guess. We have data.

These themes also led to three key recommendations. If you're skimming, I recommend focusing on reading these ones:

1. Stop ad-hoc microbenchmarking, agree on an org-wide macrobenchmark and measure every effort against this.
2. Agree on an E2E latency metric and track daily p50/p95, synthetic and real.
3. Constrain Ruby blocking time pre-answerbot to 1 second.

This document is organized at the top level by our desired Outcomes, which are my goals for your performance improvements over the next year. Underneath that are specific Recommendations to achieve those outcomes. Each Recommendation has an associated cost and benefit, rated subjectively on a 5 point scale.Ratings are designed mostly to be relative to each other, i.e. a 5 is always harder or more valuable than a 4, etc. Even cost/benefit roughly means I think it's a toss-up whether or not you should do it, while a cost higher than the benefit rating means I think it's not worth doing in the near term future.

I hope you enjoy this document and find it a useful conversation starter for where Fin can go in the future.

Nate Berkopec, owner of The Speedshop

Outcome: Improve Observability of Fin Latency

Two things really stood out to me as I investigated the latency of Fin:

1. Latency is extraordinarily high and has an incredibly low standard deviation. Average experiences are in the 15-20 second range, and p95's are not much higher than that at all. That is to say, **Fin is slow, and it's equally slow for pretty much everyone**.
2. We have very little insight into the "end to end" experience, and most work does not seem oriented towards that experience.

To optimize something, we have to understand it. I see observability as a prerequisite for any performance improvement effort, as it tells us both where the time is going, and if we're making things better or worse. Unclear answers to questions about metrics are symptoms of an observability experience which doesn't provide everything it needs to, and sloppy metrics lead to sloppy thinking and wasted work which doesn't actually impact users.

This is particularly tough at large organizations. Large orgs require "home-built" solutions to observability problems, because the off-the-shelf options are absolutely exorbitant in cost. The size and scope of the product naturally lead to teams focusing on their own specific fiefdoms, without being able to place it in a larger context.

You have a number of ideas already for improving latency for Fin, which all are great, but what I'm seeing is a lack of a framework for understanding if these changes actually impacted anyone.

The recommendations in this section, which comprises the majority of this report, are therefore focused on observing Fin latency, mostly from the perspective of a customer.

Recommendation (key): Stop ad-hoc microbenchmarking, agree on an org-wide macrobenchmark and measure everything against this.

A lot of the conversation around Fin looks like this:

> **Owner of Fin component**: We're going to work on improvement X. We think this will make *part Y* of *our component* of Fin faster! *Time passes.* **Owner of Fin component**: We made improvement X, and it made our sub-part Y of our component of Fin 80% faster! Hurrah! **Slack**: Many reacts, much wow.

The problem here is that no one asked at any point how important sub-part Y was to the entire E2E experience of Fin. The conversation instead focuses around a goal that was chosen not for it's relevance to the overall experience but for it's proximity to the component owner's ownership and area of expertise.

This kind of conversation is *extremely common* and happens *at almost every organization I work with.* Usually, it plays out at a smaller scale: an individual IC on a team sees something in the code they don't like, benchmarks it, makes it 100x faster, then posts the PR and gets lots of kudos. Then, it turns out that this 100x faster method was only 1 millisecond of latency to begin with, and is called once during a 60 second long background job.

This can happen not just at the level of an IC on a team, but also on teams in a cross-team product like Fin. It is the inevitable result of a lack of cohesive buy-in on a concrete engineering requirement. Without clarifying the high-level requirement, people make up sub-requirements on the spot and ad-hoc.

Latency requirements always exist, they simply are unspoken and cultural. Everyone agrees that there is *some* latency of Fin that would be unacceptable. If Fin took 2 minutes to respond, that would be unacceptable and would kill the product outright. So, the requirement is less than 2 minutes. What's the requirement though? Probably no one at Intercom could tell you, today, how to express that as a number.

My recommendation is to establish a common framework and **single source of truth** for talking about and measuring latency at the highest possible level (end to end), and requiring that all benchmarks reference that benchmark/understanding and relate themselves to "which part of it we're going to fix".

Once this is in place, **all performance improvements must be discussed in relation to this number**.

As far as I can tell, the closest we have to to this today is the `messenger-to-fin-with-first-token` span in `deep-traces` in Honeycomb. The duration of this span is the high-level macrobenchmark that everyone should be trying to optimize around. However, it has some deficiencies:

1. **Honeycomb is probably the wrong place for this to live** (but it may be fine to measure it here). My experience at the workshop in December was that a lot of engineers are a bit intimidated by Honeycomb. The discoverability *in Honeycomb* of this is also extremely poor. To know that this exists, basically you have to be told by a senior/staff engineer and shown how to get to it. This introduces a culture where performance is only understood by top staff engineers and the rest of us just sort of wing it. It's perfectly fine to measure this in honeycomb, but the 1 to 4 top-line latency numbers that everyone needs to focus probably need to be exported to a different place.
2. **What should the length of this span be if streaming is done?**. This is bleeding into "perceived performance", which will come later in this report, but should everyone optimize around getting the first token into the browser (nb: the first token streamed TO THE CLIENT, not to an intermediate Intercom internal service for further processing). To me, the answer to this currently is unclear (and the normative answer may also be different than the current reality).
3. **Is there anything this does not capture in the E2E experience?** This gets into the "span glossary" I'll discuss later, but it's not clear exactly *when* the span is started and when it's finished. That is critically important. If there is any error/delay in starting/finishing (for example: let's say the span only starts once the Thinking… bit pops up, which we already know is heavily delayed vs user input), we won't be measuring what the user is experiencing. The answer *should be* "it starts when the user hits enter/clicks send, ends when the first/last token paints in the browser", but is it?

Cost: 2, Benefit: 5: Aligning all teams working on this project around a single methodology is critically important, and makes this one of the few **key recommendations** I have.

Recommendation (key): Agree on an E2E latency metric and track daily p50/p95, synthetic and real.

My suggestion for the 4 numbers that matter is:

1. E2E p50 latency of a Fin response, real-world measured.
2. E2E p50 latency of a Fin response, synthetically measured in production against a "blank" customer.
3. E2E p95 latency of a Fin response, real-world measured.
4. E2E p95 latency of a Fin response, synthetically measured in production against a "blank" customer.

These four numbers capture a wide variety of things:

1. **Every step is represented**. We have to measure the entire experience from the user's perspective. This will necessarily include the latency of every impacted team.

2. **The variance is represented**. By measuring p95 and p50, we can get some idea on if the "median" case was affected by changes we make, or only "outliers".
3. **We have low-noise and high-noise scenarios**. Production data is incredibly invaluable because it includes real human beings doing real things which we cannot predict. Synthetic test scenarios are limited by our own imaginations as far as what behavior they can activate/codepaths they will run. However, the synthetic data is also useful in that it will be far less noisy than the production data, giving us the ability to detect smaller changes.
4. **Demo latency is important**, maybe even just as important as production latency. It's clear that a lot of the pain around Fin latency is in the *demo experience*, which more closely tracks the synthetic scenario rather than the production usecase. Since this is important and leads to sales, it can be tracked with the same level of importance as real-world conversations.

With these numbers, I recommend the following actions:

- **Track it**. There should be a line graph somewhere with four lines on it. Everyone should know how to get to this. It should not be something that you need to set up yourself in Honeycomb every time you want to find out what it is.
- **Post it**. Put these 4 numbers somewhere *outside* of Honeycomb in a place that can be easily looked up.
- **Slack it**. People live in Slack all day. Post this number to the relevant channel(s) on a regular basis (at least the speeding up fin channel for now).

Cost: 2, Benefit: 5. This is a **key recommendation** because agreeing on a set of 4 or fewer numbers ensures we all have the same understanding of what work is important and what is not.

Recommendation: Remove outliers and don't focus on p95+ if client latency is involved in any span or child spans.

Whenever we discuss a measurement which potentially covers latency to a browser or other 3rd-party client, we should ignore outliers and try not to focus on measurements greater than p95, such as p99 or pmax.

In my experience, a lot of people apply an SRE-based mentality of "which percentiles to monitor" to the client/browser domain. I'm only interested in a particular percentile *if the traces around that percentile have interesting information in them*. In client/browser scenarios, the *reason* a trace is at the p99 is usually an entirely *uninteresting* reason like "they live in rural India" or "they had really shitty conference wifi". We can't do anything about these reasons, they don't give us any actionable steps to take. In particular, client-based tim-

ing can just have absolutely banana things happen re: system clocks that causes some people to have 3 hour load times, etc.

Whenever you're working with data that comes from a user's browser:

1. Ignore outliers. I typically just exclude all traces/inputs that are more than 4-5x the p95.
2. Don't look at p99. It's perfectly fine to monitor p99+ when the E2E of the span is completely within your own infra.

Cost: 0, Benefit: 1. I'm telling you to ignore something, so that has no cost. It's not a big deal, but it's a common footgun I see re: teams with little RUM experience.

Recommendation: Create and maintain a "blocking"/critical path diagram.

One of the first things I did when I started to try to understand the E2E Fin experience was to turn the `deep-trace` span graph from 2 dimensions into 1 to create a "critical path" diagram.

The critical path is the longest series of steps you need to complete one after the other in an operation. This path shows which tasks must be finished as quickly as possible, because any delay in these steps will slow down the whole operation. In other words, if you have a series of tasks where some tasks depend on others being completed first, the critical path is the chain that takes the most time to finish.

A critical path is one-dimensional, unlike the 2D span graph.

The `deep-trace` span graph can (and should!) measure parallel operations. However, of any set of parallel operations, only one of them can be "on the critical path".

As far as I can reckon, the current critical path of Fin looks like:

1. Unknown amount of client latency to reach Intercom servers.
2. 5-6 seconds of time spent in internal (Ruby?) services.
3. 10-12 seconds blocking on the LLM.
4. Potentially a couple more seconds blocking on an internal Ruby service.
5. Unknown amount of client latency back to the Messenger.

We should *automatically* turn the `deep-trace` span graph into a critical path diagram. It would require some information about the dependencies between spans. You could probably implement this with tags that said what span blocks this one, and require that all spans must have this tag except the very top level/first span. At the least, you could use a simple heuristic that says "this span ended and just after that a new one started, they're probably linked".

I am describing another hackathon/AI project. If this is automatically maintained based on a regular Honeycomb export, it could even send an alert to the

Slack channel if certain changes are made to the critical path.

Without an understanding of the critical path, it is impossible to determine "what can we speed up here to make the entire E2E operation faster" just by looking at the `deep-trace` span graph, because you are lacking that information about span dependency. I basically had to ask a lot of questions and interview people to figure that out.

If not automated, this could also be included in the trace glossary (discussed next).

Cost: 2, Benefit: 3. This "critical path visualizer" is one of many tools I describe in this report that fit "good thing to ship in a few days with Cursor". Internal tools which only need to work on a temporary basis, only have an internal customer, and could potentially be one-shotted by AI make great fodder for that kind of work. We used to call them hackathon projects, now we let AI have all the fun.

Recommendation: Create a trace glossary.

`deep-traces`, while useful, does not "explain itself". Anyone truly looking to understand and optimize a given trace has to understand, for every span:

1. What does this span represent? What operations, which are not traced, happen inside of this span?
2. Where, exactly, does this span start? When does the "stopwatch" start for this span? Potentially, could we even point to a line of code where the hook is?
3. When exactly does this span end?

Currently, deciphering the meaning, start and end points of each span requires extensive investigation and tribal knowledge. A trace glossary would provide this missing context at scale.

We cannot optimize what we do not understand and when it comes to tracing spans, details really matter. A ~20 second E2E operation is inevitably extremely complex, and one cannot simply read the `subject_name` of 10 spans and understand the entire process. That's like 1kb of data, but our internal context window is going to need 10x that amount to understand what's going on here.

We cannot optimize something this complicated alone, which means that tens and potentially over a hundred people will need to understand and use the toolset here. That scale of knowledge cannot be disseminated through one-on-one chats and meetings.

If `deep-traces` is going to be the primary tool around which this effort is built, we need to answer those three questions about every span in the trace and be able to explain those answers at scale.

This could be a google doc, this might be something more dynamic which pulls a sample of Honeycomb traces down and creates the "keys" in this glossary dictionary based on the span names, and then human beings go and fill in the "values" for each key. This could also alert you when you've added a new "key" without a "value".

For every span, it should include:

1. **Span description:** A clear explanation of the operations represented by the span, including any untraced operations that occur within it.
2. **Start point:** Precise definition of when the span's timer begins, ideally with a link to the relevant code or instrumentation hook.
3. **End point:** Precise definition of when the span's timer ends.

Cost: 2, Benefit: 4. This recommendation basically just describes a Google doc or quick hackathon tool that should exist but does not.

Recommendation: Create a junior-engineer compatible "wrapper" around Honeycomb.

Honeycomb is perfectly fine as a tool, and I understand the constraints created by Intercom's scale and how that impacts your choice of observability tooling. I'm thankful that Intercom didn't have to go *completely* bespoke and still gets to pull off-the-shelf stuff, even it's more specialized than, e.g., just running Datadog APM everywhere.

Yet, it's still not very intuitive to use. **Since any engineer can introduce a performance issue, every engineer needs access to the tooling to observe and fix those issues**. You wouldn't accept a workflow where only staff+ engineers knew how to monitor exceptions. Errors in program correctness are more common but no less serious than a perf bug that adds several seconds onto an important workflow.

Honeycomb has a number of "sharp edges" which can trip up even senior engineers. When looking at a query result, if you click "traces", you get a list of the *slowest* traces in the time range. But these traces, particularly if you're looking at the whole fin E2E spans, are basically useless! They're the weirdest, most unusual spans which *by definition* only occurred once out of a million times! They contain extreme client latency, edge cases and outliers. It's *far more useful* to click "explore data" and look at *recent* traces which are "around" the average experience. But this kind of "pro tip" is so difficult to instill at scale and can't just be unwritten knowledge that *might* get passed around in a Slack DM.

Part of this "wrapper" is the trace glossary defined in the previous recommendation. Here are the other features it can include:

1. The p50/p95 "benchmark" numbers and results
2. Links to "example" traces from the last 24 hours at various latency thresholds: here's 10 traces from the ~p25, ~p50, ~p75 etc respectively.

3. Links to previous "experiments" and views of those experiments in honeycomb, e.g. the used_eager_request tag and various queries used to evaluate the performance of that experiment
4. Links to tools *outside* of Honeycomb you're using to evaluate Fin perf

This is more than a Google Doc but less than something you could ship in a couple days. I think it is another "hackathon/AI" project.

Cost: 2, Benefit: 4. Honeycomb "training wheels" plus a bit extra isn't hard to ship, but provides an onramp for getting juniors/intermediates onto a "standard" workflow. "People like us do things like this."

Recommendation: Enforce a workflow: if you have an idea for a latency improvement, identify or introduce a span you think will decrease in size.

If you package the previous six recommendations together, you can summarize them as "orient your workflow around Honeycomb `deep-traces`, and make it usable by everyone".

Once you've done that, every performance improvement can be clearly judged based on it's impact to production `deep-traces`.

You have a hypothesis for an improvement to Fin E2E latency? Great!

1. Which span(s) will your improvement affect?
2. How much do you think you will remove from those spans?
3. If you remove time from that span, what happens? Is it on the critical path (see previous discussion on critical paths)?
4. When you do put this into production, can you tag/flag your change and show with the data that you improved what you said you would?

This is the final piece that prevents the ad-hoc microbenchmarking discussed in the previous recommendation. When all performance improvements are viewed in the same common framework, their impact can be clearly judged, both pre- and post- deployment.

Cost: 1, Benefit: 3 This is easy once the previous recommendations have been implemented.

Recommendation: Include browser TTFB/controller breakdown in trace.

The first and last spans of `deep-traces` contain an uncertain amount of client RTT network latency. This makes those particular spans very difficult to analyze (how much of the latency comes from time spent in network versus time blocking on our service?), particularly at the p95+.

Split `controller_ms` into `controller_ms` and `client_ms`, where:

- The *start time* of `client_ms` is the input event kicking off the Fin interaction, like a keypress or click.
- The *end time* of `client_ms` is the start of `controller_ms`, which should be the time that the request is received by Intercom load balancers.

Cost: 2, Benefit: 3. Combining network time and backend server time into a single span makes it too difficult to interpret.

Recommendation: Include browser geography in trace.

You already have browser version, so I know you have this browser data. However, it would be useful to include user geography. Geography is the single biggest correlate with latency, which will help to interpret the `client_ms` spans you'll get when you implement the previous recommendation.

Mixing everyone's geography together means that you're comparing apples to oranges, because everyone's client network latencies are going to be significantly different. If you can look at "all Fin E2E traces from the EU" that's a very different picture than "all Fin E2E traces from India".

Client network latency is *mostly* out of our control, so we want to be very clear about where it is and where it's coming from. If we're not, we'll end up basing work on the wrong conclusion. "I don't need to improve this particular thing because the transaction is dominated by client latency" - ok, that might be true in the aggregate, but what about US users only?

Cost: 1, Benefit: 1 Probably not the biggest fish you need to fry, but also not difficult since a lot of the stuff to capture this data is probably already built.

Recommendation: Capture actual timing of generating token 1 versus final token.

Currently, the `answerbot_ms` span can be a `streamed_response` or not. However, the span length is the same in both cases. This means that we're not capturing the timing of *when* the first token is generated versus the last. Even if the response *isn't* streamed, it would be useful to have a span showing "this is when the first token came back, and here's where we finished", to evaluate model performance. Currently the `answerbot_ms` span captures this + everything else, which isn't enough detail to determine if `answerbot` is the problem or if the model itself is.

With streamed responses, we may decide that "when we started streaming" versus "when we completed" is more important. That data is not currently available in the span.

Cost: 1, Benefit: 2: This gets more important when responses are streamed, which I would like to do more of.

Recommendation: Include input/output token count and model as a field on answerbot_ms.

Output token count is probably the most sensitive parameter to LLM latency. Generating 100% of an answerbot response basically follows the formula of:

```
time = tokens/sec*input_coefficient*tokens
```

…where tokens/sec is a constant determined by the perf of the model, input_coefficient is a factor that increases as we feed the model more input/context, and tokens is the number of tokens of output.

The funny thing about LLM models is that that the `tokens` count scales 1:1 with time. With the input, it scales (I think linearly) at a much lower factor - adding 10x the context might slow down the response by increasing the input_coefficient by only 10%.

So, both of these parameters are important for evaluating *why* any given LLM response took as long as it did.

Cost: 1, Benefit: 2. We almost certainly have both of these numbers easily/readily available. They just need to shipped as fields on `answerbot_ms`.

Recommendation: Close instrumentation gap in between ai_agent_request and answerbot_ms spans.

This was mentioned in Slack but it hasn't yet been resolved.

Currently, there's a gap of 1-200ms between the `ai_agent_request_waiting` span and `answerbot_ms` span. It is the only such gap in the `deep-trace` instrumentation. There was some speculation as to what this gap might represent.

We can't measure empty space, we need a span there to be able to understand, monitor, and optimize it.

Cost: 1, Benefit: 2. Not much to add - measure the thing!

Recommendation: Institute synthetic benchmarking and profiling with a tag on deep-traces.

Normally, I'm quite against synthetic benchmarking and performance testing. It's too difficult to set up, and usually doesn't cover any useful scenarios because a useful scenario would require way too much data (which is difficult to generate/copy from prod, etc etc).

However, Fin has two things going for it which make it a great candidate for synthetic benchmarking:

1. p50 is very close to p95. The experience is slow for the base case in the same way that's slow for the extreme case (at least at this stage).

2. You have a lot of resources and org weight pulling for this product, which makes investment in a big project like this easier.

The best way to measure this would be to just keep it all oriented around `deep_traces`. Create synthetic traffic that generates the data you want inside `deep-traces`, and then let people extract/export it to go where it needs to go. This lets us keep oriented around `deep-traces` as "the final measurement" of all latency, keeping methodological consistency while giving us the ability to profile our synthetic traffic on top "for free".

Cost: 2, Benefit: 4 What we're really talking about is adding a tag/field to `deep_traces`, setting up a bot to run the scenario every hour, and then extracting that into a benchmark number on a regular basis. Not that complicated.

Recommendation: Separate blocking CPU/Ruby/service time from any queueing, waiting as spans.

We've already talked about the need to separate client network latency from other work. However, it's also important to separate queueing and waiting from blocking on a service response.

The reason is that the answer to "what we should do to fix it" is completely different.

- **Waiting and queueing is fixed by scaling horizontally**. If you're waiting on SQS, the answer to making that span shorter is to make more stuff pull from the queue.
- **Blocking on a service is fixed by increasing efficiency of the service**. If we spend 1 second blocking on a Ruby controller action response, the way you fix that is by optimizing that response.

This is extremely helpful because **waiting and queueing can be solved extremely quickly by throwing money at the problem** while service efficiency is a long-term, high-effort project.

We should separate waiting time into its own span and service time into its own span wherever possible.

Cost: 2, Benefit: 3: This is gonna require some staff+ level scoping for "what queues have a p99 of greater than 50ms?" to even determine which queues are worth instrumenting, but my impression was that these queues do exist in the current trace.

Recommendation: Measure latency impact on NPS/survey result instead of just resolution.

You may already have this data.

When we discussed the impact of latency on this project, we discussed it primarily in the context of impact on resolution rate. This is definitely extremely

important and interesting.

However, I wonder if it doesn't capture everything. Could a faster experience leave people more satisfied, even if the problem was not resolved (you didn't fix my problem, but at least the experience was satisfying) versus a painful experience (this thing is slow as shit!) *also* leading to no resolution?

I'm not a Product Guy and there's without a doubt 15 years of history of this kind of data collection/thinking about it at Intercom, so I don't need to re-invent the wheel and interject my own opinion here.

This recommendation is just to wonder: is resolution rate really the best or only lens to view the impact of reduced latency? How should we weight resolution rate versus a more subjective/holistic experience rating?

Cost: 2, Benefit: 2: This may become more important to evaluate "perceived" performance improvements versus real latency drops.

Recommendation: Put a latency price to every product feature pre-answerbot in `deep-traces`.

As of today, your problem is that *everyone's* experience is slow. You will fix this, and eventually the problem will become *some people's* experience is slow. It's already pretty clear that "some people" is going to be "people with a lot of product features enabled that can change the answerbot response".

The easiest way to do this would probably be to add a field to every relevant span that captures feature state: this thing is on/off, we ran X number of workflows, etc. More difficult (cost and implementation wise) but maybe more useful would be for all of these sub-steps to become spans. That's a tradeoff you'll have to evaluate.

Cost: 3, Benefit: 3: It probably doesn't make sense to implement this within the next 3-6 months, but it will become a problem eventually.

Outcome: Reduce Median Fin Latency to Less Than 10 Seconds

Currently, Fin latency as measured by `deep-traces` is:

- `p50`: 19 seconds
- `p95`: 30.5 seconds

There is a pathway towards halving these numbers.

In the study of human-computer-interaction, there's a concept of a "just noticeable difference" between two different latencies. That difference is usually held to be about 20 percent. A 20 percent faster experience is "just noticeable" for a user, anything less than that and they'll subjectively evaluate it to be the same.

That means there are roughly three "just noticeable" steps or thresholds on the way to the improvement which is possible:

- **16 second p50**.
- **13 second p50**.
- **10 second p50**.

These can become intermediate goals.

Overall, the 19 seconds breaks down to something like:

1. ~1 second of client network latency (guessing).
2. 5-6 seconds of blocking (Ruby) internal service time.
3. 12-13 seconds of time blocking on LLMs.

We have a "menu" of options in the following recommendations which can be chopped/changed and combined to "achieve" these various intermediate targets. I've included an estimate of how much p50 latency is "on the table" for each.

Recommendation: Create an internal benchmark for vendor/model selection. (1-2 seconds)

As far as latency goes, models have a number of important characteristics:

1. Time to first token.
2. Tokens per second.
3. Impact of input/context on #1 and #2.
4. (Optional): how many tokens does the model tend to output? If a model is more verbose than another, it may be less suitable for Fin.

I looked around and there isn't a good public dataset which compares these 3 characteristics with any recency (since this stuff is changing literally weekly) and with the particular vendors you have or might evaluate.

Unfortunately that means you're stuck building this yourself. Maybe it's not that difficult, I haven't worked with LLMs enough to really know if this kind of timing is difficult or not.

Without this kind of information, I don't think you can implement some of the other recommendations to come in a data-driven way. Will using gpt-3-turbo be a better fit for some answers? Depends on how *much* faster it is: 10x is a big difference versus 1x. Maybe it has fast tokens/sec but takes twice as long to generate token number 1. As far as I know, no one has this data.

You might *not* implement this recommendation if you feel that switching models in/out of production is low risk and easy. In that case, you might just implement my other recommendations to add fields to `answerbot_ms` tracking these 4 numbers and just "test it in prod".

Cost: 3, Benefit: 4 Maybe it's also fodder for the engineering blog - everybody loves AI benchmarks!

Recommendation: Constrain output tokens in certain scenarios (2-4 seconds)

As we already discussed, one of the most important aspects of LLM latency is just how many tokens you have to output. It's a straight up linear relationship.

LLMs, and in particular certain models, tend to be overly verbose. Far more than a human. This *may* be a strength in terms of resolution (you'll have to run your own data on that), or it may *not* increase resolution rates and may simply be a weakness in adding unnecessary latency.

It's also possible that you could fork certain types of queries into a "low-output-token" path.

Since time-to-last-token is an incredibly important part of the experience and time between first and last token is so affected by token count, controlling output length could easily halve the time of certain responses.

Cost: 2, Benefit: 4. The difficulty is in figuring out when you can do this without affecting resolution, not in the implementation.

Recommendation: Constrain/eliminate modifications to model output, prefer only input constraints (2 seconds)

Currently you have this `process_fin_response_ms` span on most (but not all) responses. My understanding is that you take the complete response and make some modifications to it before sending it on to the client.

This has two problems:

1. It impacts streaming the response back to the client. I'm not sure if this completely blocks streaming or not.
2. It's very slow, with about a 2 second p50 duration.

In theory, we should be able to *explain* to the model the output we want without modifying it's output after the fact. I understand this isn't 100% possible (that's why DeepSeek had such funny behavior regarding Tianamen Square!)

I'm not sure there's a lot of value in making this step *faster* so much as there is in *removing* it, or perhaps only applying it *in the browser*. If it impacts streaming, it's quite costly on its own, but whatever we're doing in this step is also extremely laggy.

Cost: 4, Benefit: 5 2 seconds is nothing to sneeze at. However, I recognize that getting this done without impacting product quality will be very, very difficult.

Recommendation (key): Constrain Ruby blocking time pre-answerbot to 1 second (2-5 seconds)

Starting the answerbot span is probably one of the most important parts of the response, as the work on eager requests has shown. What surprises me is how much of the time pre-answerbot can be spent blocking on internal services, like `workflows` and `controller_ms`.

It's my professional experience that **it is almost always possible to do what we need to do with 1 second or less** of response time. I have not yet met a problem that can't be solved by a 1 second response. Maybe you're the first. I'm not sure. But the p50 for internal blocking time is something north of 2 seconds right now, which means it's an area that could be improved.

This recommendation is merely to *set the standard* that internal blocking time pre-answerbot should be 1 second or less. We may need a new span (starting at the beginning of the E2E and ending at answerbot_ms) to reflect and measure this time.

Cost: 2, Benefit: 2 I'm only asking for 1 additional span.

Recommendation: Implement CQRS wherever possible pre-answerbot (1 second)

Command Query Responsibility Segregation is a design pattern, where you separate the actions that change data (commands) from actions that read that data (queries). We use commands to modify the system, and queries to fetch info about the system.

I find the "frame" of CQRS is helpful for evaluating what work needs to be done and what work does not need to be done over the course of a whole operation and instead can be backgrounded. If we "command" some change in the data but then do not "query" it later or use it to produce our output, that work can be moved out of the critical path and done in some background flow.

Backgroundable work no doubt exists somewhere between the start of `controller_ms` and the start of `answerbot_ms`. If you consider this step as a black-box function, you would say that:

1. The input is what the user typed in the chat window.
2. The output is what we feed to the model.

Everything else along the way which does not *query data* or *command data* that is later *queried* to create the output can and should be done off the critical path.

This recommendation is to audit this particular section of the E2E flow with this in mind. This particular part of the flow is also probably the most complex, the most historical, and carries the most baggage and complexity, which is why I didn't get to it yet. This could be an interesting project for future work.

Cost: 3, Benefit: 4 Heavy lift, moderate payoff compared to some of our other options.

Recommendation: Communicate the latency price of workflows or other "turned on features"/config to customers (not the base case) (0-1 seconds)

In VSCode, we have this neat feature that shows how long each extension contributes to startup time delay.

Since workflows will inevitably add latency to conversations, you might simply display to this a user.

If a workflow, on the median, adds more than 1 second to conversation latency, show that latency in the UI:

I don't think this will be that common (probably only happens to a few complex customers), but simply surfacing the cost of a user's action to them might be helpful.

Cost: 3, Benefit: 3: Low impact only on particular customers.

Recommendation: Direct "simple/meaningless" queries (often used in demos) to simpler models (2-4 seconds)

In demos, and when users are testing out Fin for themselves, they might ask a simple, meaningless thing:

"Hey Fin, how are you?" "Hi" "Yo"

You might direct these to models with better performance characteristics. Unfortunately you can't run them through *another* model to classify the query as "easy" or "simple".

This recommendation also interacts with my other recommendation re: removing post-processing steps. If the user can only tweak model responses via inputs instead of modifying outputs, we can provide that same input to this different, lighter-weight model, rather than rely on our post-processing which was optimized/tuned for a different output from a different model.

I've been told this option has been considered but as yet hasn't been implemented. The thing is that answerbot_ms spans are *so* long and *such* a huge proportion of the response, that even if you can only do this 10% of the time, it still knocks 1-2 seconds off the median response.

Also, it strikes me that these kinds of queries will probably be over-represented in demo and sales scenarios, which is also a key goal of this Fin latency project.

Cost: 3, Benefit: 4 Probably worth it more for the impact on demo scenarios than its impact in the real world.

Outcome: Improve Perceived Performance

The rest of this report talks about things which don't decrease the "time until the last token is in the user's browser". Performance is subjective, and so there are things we can do that play on that subjective perception.

The difficulty in recommending these things is that it's hard to measure their effectiveness. This is particularly true if the only output variable we're monitoring is resolution rate, rather than "NPS" or satisfaction or performance perception surveys (discussed in a previous recommendation).

However, they can't be ignored. And, even in a perfect world, latency of this experience will never be lower than ~5 seconds, so we really have to start from the premise of "this will always be slow and we have to use psychology to minimize that frustration".

Recommendation: Laboratory test various latency experiences (blind, recall based testing, surveys).

We can't really do too much experimentation on Intercom's customers. They're not our users. But, we can do experiments of our own, on our own people!

I always think about performance in an *absolute* sense. How bad does a 20 second experience feel? How about a 10 second experience? 5?

The cool thing is we don't have to guess at what that feels like, we can just try it for ourselves.

I recommend setting up a fake testbed experience. It should look and feel exactly like messenger, except the responses are always 100% canned and not backed by AI, and instead just outputs lorem ipsum or something. However, every step of the experience can be controlled:

1. Speed of the tokens being streamed per second
2. Time of first token appearing in the browser
3. Size of the response
4. Trying other "experiments" I'm about to suggest in upcoming recommendations

Currently, each of these things has a p50/p95 and distribution of latency. We can set up the tool to have that same distribution, and then also different distributions of what we think we could achieve in the future. Then, just try it out: do you notice a difference? How does it feel? Better? Worse? In what way? Which kinds of latency, which experiments had the biggest subjective impact on you?

Then you can just run this on the captive audience you have of Intercom employees and ask people to fill out a survey on what they thought.

In performance UX testing, we often use recall based testing. People tend to

recall the "peak intensity" of wait frustration and the "end" of the wait. How people *remember* their experience of latency is just as important as how long as it actually was. In this study design, you ask people how long they *perceived* the wait was. If your experiment works, they might say "it felt like two seconds" when actually it was four seconds.

Try some recall-based testing using this "skeleton" experience on an internal audience and see what you can come up with.

Cost: 2, Benefit: 3. Could be fun. Feels like a week-long hackathon project.

Recommendation: Respond with intermediate or precomputed answers

This is another one that's been discussed often as "interesting but difficult to implement from a product perspective", because many settings and customer context can change Fin's response, so it's difficult to "precompute" answers.

What I'm talking about re: "intermediate or precomputed" answers is a response before the final response which can be done quickly and perhaps without actually calling an LLM in-line.

If this improvement actually does improve perceived performance (see previous recc), the memory/compute tradeoff could be worth it. If you imagine every possible user setting/workflow/input that could possibly modify an LLM response as part of our cache key, you might just increment a "version" number every time one of those things changes, and then precompute a few hundred "stalling" answers every time the version number changes. It's possible that this is *not* an option if things change change the model "stalling response" *without user input* or setting changes, i.e the time of day/week changing, maybe user content or something not in the "workflow/admin settings" area changing.

Humans do this to reduce the perceived wait of their customers, so why not emulate that ourselves?

Cost: 3, Benefit: 3: Tough to think through from a product perspective.

Recommendation: Respond with something to read/look at in the meantime to keep user attention in the window (support articles, etc) to increase user engagement

People will subjectively evaluate waits as being more tolerable if they are **engaged** during the wait.

Disney famously used various techniques to make waits at their theme parks "feel" less onerous. At the Haunted Mansion theres this whole prelude where you go into a dimly lit foyer with spooky portraits, and then the "end" of the wait experience (remember the peak/end principle from the previous example) is this big "wow" moment where the room stretches upward.

Uber is pretty well known for having an engaging "wait" experience. Next time you order an Uber, pay attention to all the animations and visual distraction flying by. Getting an Uber is inevitably going to be a 10 second plus interaction, probably even a minute in cases, so it's quite similar to Fin in that respect.

Currently, the wait experience on Fin is basically a glorified spinner. The times where you switch out the "Thinking…" text with other stuff is already quite effective, but I wonder if there's more room to improve here. Displaying chain of thought is extremely engaging but probably not always possible, but maybe there's some kind of more extensive animation or experience that could be added here.

You could also find ways to dump more to read in front of the user while the response is being generated. Reading is engaging, and we only need them to read for 10-20 seconds. Is there some kind of relevant content that could be inserted into this wait?

Cost: 2, Benefit: 4: I feel strongly that this could improve perceived performance.

Recommendation: Create a determinate progress bar based on past performance

In my experience, engineers are really reluctant to use a determinate progress bar (0-100%) versus an indefinite indicator, like a spinner. To me, this is often just an "engineer mind" getting in the way of a good product decision. The engineer thinks, I have no idea how long this is going to take and I don't have some definite steps between 0 and 100%.

The problem with this mindset is that definite progress bars are a huge UX improvement over indefinite ones:

- **Sets expectations**. "Thinking…" provides no feedback as to how long this is going to take. However, we already know that 95% of the time it will take between 10 and 30 seconds.
- **Improves engagement**. Determinate progress indicators are constantly changing, which, as boring as it sounds, gives the user something to occupy their attention. Remember how fun it was to watch the Windows 95 disk defragger?
- **Creates trust**. It makes the system appear more reliable. The wait is not "between 0 and infinity", it's "between about 10-30 seconds from now".

This means we should try to implement definite progress bars wherever possible. In the case of E2E Fin latency, we really do know most of the time how long this is going to take. We should communicate that.

Cost: 2, Benefit: 4: I'd love to see how this performs on an in-house laboratory test.

Summary of Recommendations

- Outcome: Improve Observability of Fin Latency
  - Recommendation (key): Stop ad-hoc microbenchmarking, agree on an org-wide macrobenchmark and measure everything against this. *Cost: 2 Benefit: 5*
  - Recommendation (key): Agree on an E2E latency metric and track daily p50/p95, synthetic and real. *Cost: 2 Benefit: 5*
  - Recommendation: Institute synthetic benchmarking and profiling with a tag on deep-traces. *Cost: 2 Benefit: 4*
  - Recommendation: Enforce a workflow: if you have an idea for a latency improvement, identify or introduce a span you think will decrease in size. *Cost: 1 Benefit: 3*
  - Recommendation: Create a junior-engineer compatible "wrapper" around Honeycomb. *Cost: 2 Benefit: 4*
  - Recommendation: Create a trace glossary. *Cost: 2 Benefit: 4*
  - Recommendation: Separate blocking CPU/Ruby/service time from any queueing, waiting as spans. *Cost: 2 Benefit: 3*
  - Recommendation: Include input/output token count and model as a field on answerbot_ms. *Cost: 1 Benefit: 2*
  - Recommendation: Capture actual timing of generating token 1 versus final token. *Cost: 1 Benefit: 2*
  - Recommendation: Include browser TTFB/controller breakdown in trace. *Cost: 2 Benefit: 3*
  - Recommendation: Create and maintain a "blocking"/critical path diagram. *Cost: 2 Benefit: 3*
  - Recommendation: Remove outliers and don't focus on p95+ if client latency is involved in any span or child spans. *Cost: 0 Benefit: 1*
  - Recommendation: Put a latency price to every product feature pre-answerbot in `deep-traces`. *Cost: 3 Benefit: 3*
  - Recommendation: Measure latency impact on NPS/survey result instead of just resolution. *Cost: 2 Benefit: 2*
  - Recommendation: Include browser geography in trace. *Cost: 1 Benefit: 1*
  - Recommendation: Close instrumentation gap in between ai_agent_request and answerbot_ms spans. *Cost: 1 Benefit: 2*
- Outcome: Reduce Median Fin Latency to Less Than 10 Seconds
  - Recommendation (key): Constrain Ruby blocking time pre-answerbot to 1 second (2-5 seconds) *Cost: 2 Benefit: 5*
  - Recommendation: Constrain output tokens in certain scenarios (2-4 seconds) *Cost: 2 Benefit: 4*
  - Recommendation: Direct "simple/meaningless" queries (often used in demos) to simpler models (2-4 seconds) *Cost: 3 Benefit: 4*
  - Recommendation: Implement CQRS wherever possible pre-answerbot (1 second) *Cost: 3 Benefit: 4*

  - Recommendation: Constrain/eliminate modifications to model output, prefer only input constraints (2 seconds) *Cost: 4 Benefit: 5*
  - Recommendation: Create an internal benchmark for vendor/model selection. (1-2 seconds) *Cost: 3 Benefit: 4*
  - Recommendation: Communicate the latency price of workflows or other "turned on features"/config to customers (not the base case) (0-1 seconds) *Cost: 3 Benefit: 3*
- Outcome: Improve Perceived Performance
  - Recommendation: Create a determinate progress bar based on past performance *Cost: 2 Benefit: 4*
  - Recommendation: Respond with something to read/look at in the meantime to keep user attention in the window (support articles, etc) to increase user engagement *Cost: 2 Benefit: 4*
  - Recommendation: Laboratory test various latency experiences (blind, recall based testing, surveys). *Cost: 2 Benefit: 3*
  - Recommendation: Respond with intermediate or precomputed answers *Cost: 3 Benefit: 3*