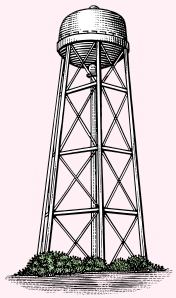
Rate Limit Types for Stream Programs

work-in-progress

Lucas Du and Caleb Stanford
UC Davis



Large-scale stream-processing programs

are widely used in industry for high-volume, real-time data ingestion and analysis.

kafka

Hydro

Some programming challenges

...ordering guarantees, causality, automatic distribution, efficient parallellism, low latency, high throughput...

load-management

- *Manage load* **by limiting rate**
- e.g. at most 5 events every 10 seconds, or at most 10000 requests every hour

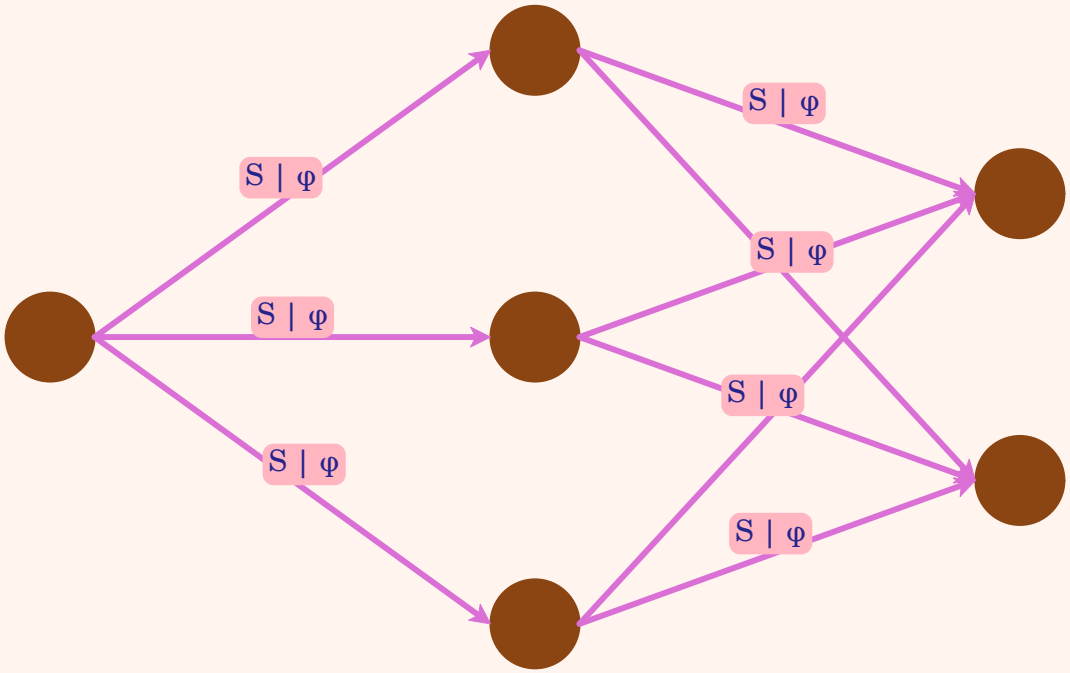
- *Manage load* **by limiting rate**
- e.g. at most 5 events every 10 seconds, or at most 10000 requests every hour
- Commonly used by public-facing APIs, i.e. Github's API, Shopify's API, or LLM APIs
    - Github's API currently allows authenticated users 5000 requests per hour

Can we get more

programming language and correctness

support?

A vision for verified stream programming



S | φ Where S a stream and φ is some property. In other words, a *refinement type* or *(property type)*.

(For us, the φs we care about are **stream rates**.)

Contributions (WIP)

- A **refinement type system for stream rates.**
- A **semantics** based on *timed regular languages.*
- An efficient type checking algorithm.
- An implementation and corresponding evaluation of the type system.

Contributions (WIP)

- A **refinement type system for stream rates.**
- A **semantics** based on *timed regular languages.*
- An efficient type checking algorithm.
- An implementation and corresponding evaluation of the type system.

A brief aside

For implementation and evaluation, we would like to use some existing, well-engineered systems infrastructure for distributed stream processing.

Hydro seems like a great choice!

(And we would love to collaborate!)

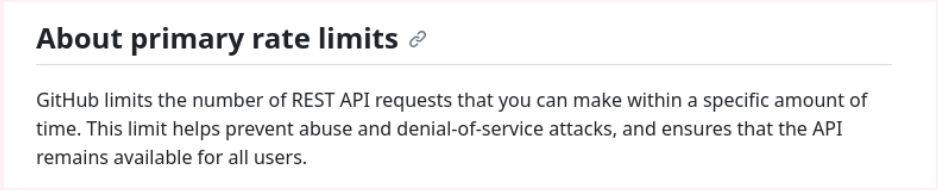
What is a stream rate?

- for our purposes, 2 parts: an event count (n) and a window size (t)
- (at most) n events per each window of size t

What is a stream rate?

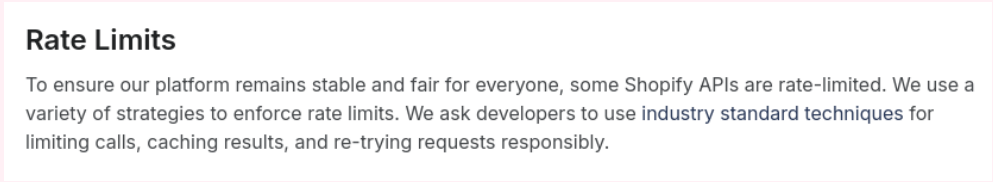
- for our purposes, 2 parts: an event count (n) and a window size (t)
- (at most) n events per each window of size t
- a sound static abstraction to reason about dynamic behavior of stream rates in the wild

What is a stream rate?



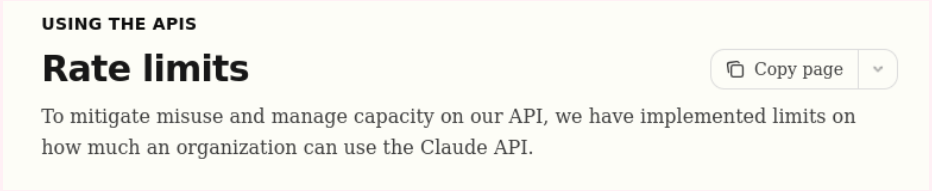
About primary rate limits

GitHub limits the number of REST API requests that you can make within a specific amount of time. This limit helps prevent abuse and denial-of-service attacks, and ensures that the API remains available for all users.



Rate Limits

To ensure our platform remains stable and fair for everyone, some Shopify APIs are rate-limited. We use a variety of strategies to enforce rate limits. We ask developers to use industry standard techniques for limiting calls, caching results, and re-trying requests responsibly.



USING THE APIS

Rate limits

Copy page

To mitigate misuse and manage capacity on our API, we have implemented limits on how much an organization can use the Claude API.

What is a stream rate?

- **latency spikes:** as requests get backed up and added to a buffer, requests start taking longer to return
- **dropped events:** some events/requests may just be dropped and not processed
- **service outage:** maybe machines crash because they are overloaded

What is a stream rate?

When we compose program A with program B, and A's output rate exceeds B's input rate limit, we may run into these kinds of problems.

What is a stream rate?

What is a type structure that can model stream rates?

$@n/t$

$@n/t$

- n events in every window of size t units
- $@10/5$ means 10 events in every 5 unit window

$S_1 \cdot S_2$

$S_1 \cdot S_2$

- S_1 and S_2 are stream rate types (so our definition is recursive)
- a stream of rate S_1 **followed by** a stream of rate S_2
- $@10/5 \cdot @12/6$

$$S_1 + S_2$$

$S_1 + S_2$

- a stream of rate S_1 **or** rate S_2
- $@10/5 + @12/6$

$$S_1 \parallel S_2$$

$S_1 \parallel S_2$

- a stream of rate S_1 **in parallel with** a stream of rate S_2
- $@10/5 \parallel @12/6$

$$S_1 \wedge S_2$$

$S_1 \wedge S_2$

- a stream of rate S_1 **and** rate S_2
- $@10/5 \wedge @12/6$

Stream rate semantics

Two main kinds of rate distributions

1. segmented (or tumbling) rates

2. uniform (or sliding) rates

(For example, Github enforces a tumbling rate and Shopify enforces a sliding rate. Claude enforces a sliding rate.)

Tumbling rates

Github: tumbling rate of 5000 requests per 1 hour, with fixed 1 hour windows starting from time of first request

Sliding rates

Shopify: sliding rate of 1000 "points" per 60 seconds, for any arbitrary 60 second window

From now on, we will focus solely on **sliding rates**, for the sake of simplicity. Other interesting things happen when we consider tumbling rates and combinations of sliding and tumbling rates! But we won't talk about those things here.

Reasoning about rate limits is still largely a manual process.

This can get complicated and unwieldy when stream programs get larger and more complex.

Even relatively simple cases can be tricky to reason about.

Even relatively simple cases can be tricky to reason about.

For example, Shopify allows a sliding rate of 1000 points per 60 seconds.

Even relatively simple cases can be tricky to reason about.

For example, Shopify allows a sliding rate of 1000 points per 60 seconds.

Say we have a program that sends a stream of requests to our Shopify API endpoint at a rate of 2000 points per 120 seconds. Are we guaranteed to remain under the rate limit?

Can we get more

programming language and correctness

support?

Can we design a **refinement type system** to help us reason about rate limits?

Types offer a lightweight, interactive way to automate lots of tricky reasoning!

Can we design a **refinement type system** to help us reason about rate limits?

Types offer a lightweight, interactive way to automate lots of tricky reasoning!

More concretely: Can problems like *safe rate composability* (i.e. the previous Shopify API example) be reduced to a **property of type safety**?

A key piece any refinement type system is the question of

subtyping

A key piece any refinement type system is the question of

subtyping

Practically speaking, can one stream fit safely into another?

A key piece any refinement type system is the question of

subtyping

Practically speaking, can one stream fit safely into another?

More concretely, remember our Shopify example: Does a stream with sliding rate of 2000 points per 120 seconds fit safely into a stream with sliding rate of 1000 points per 60 seconds?

A key piece any refinement type system is the question of

subtyping

Practically speaking, can one stream fit safely into another?

More concretely, remember our Shopify example: Does a stream with sliding rate of 2000 points per 120 seconds fit safely into a stream with sliding rate of 1000 points per 60 seconds?

From a general type system perspective, subtyping allows flexibility: i.e. can we safely cast an int to a float (or, similarly, use an int in place of a float)?

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$\frac{@n_1/t_1 <: @n_2/t_2}{@10/5 <:? @12/5}$$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$\frac{@n_1/t_1 <: @n_2/t_2}{}$$

$$@10/5 <:? @12/5 \checkmark$$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

$$@10/5 <:? @12/5 \checkmark$$

$$@10/5 <:? @12/6$$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$\frac{}{@n_1/t_1 <: @n_2/t_2}$$

$$@10/5 <:? @12/5 \checkmark$$

$$@10/5 <:? @12/6 \times$$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

$@10/5 <:? @12/5$ ✓

$@10/5 <:? @12/6$ ✗

$@10/5 <:? @22/6$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

$@10/5 <:? @12/5$ ✓

$@10/5 <:? @12/6$ ✗

$@10/5 <:? @22/6$ ✓

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

$$@10/5 <:? @12/5 \checkmark$$

$$@10/5 <:? @12/6 \times$$

$$@10/5 <:? @22/6 \checkmark$$

$$@40/6 <:? @40/5$$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

$@10/5 <:? @12/5$ ✓

$@10/5 <:? @12/6$ ✗

$@10/5 <:? @22/6$ ✓

$@40/6 <:? @40/5$ ✓

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

$@10/5 <:? @12/5$ ✓

$@10/5 <:? @12/6$ ✗

$@10/5 <:? @22/6$ ✓

$@40/6 <:? @40/5$ ✓

$@41/6 <:? @40/5$

Subtyping for "raw" rate types

we will use the $<:$ symbol to represent a subtyping relationship

$$@n_1/t_1 <: @n_2/t_2$$

$@10/5 <:? @12/5$ ✓

$@10/5 <:? @12/6$ ✗

$@10/5 <:? @22/6$ ✓

$@40/6 <:? @40/5$ ✓

$@41/6 <:? @40/5$ ✗

RAW-RATE-SUB

$$\frac{(t_2 < t_1 \wedge n_1 \leq n_2) \vee (t_1 \leq t_2 \wedge n_1 \leq n_2/(\lceil t_2/t_1 \rceil))}{@n_1/t_1 \leq @n_2/t_2}$$

What happens with more complex types?

i.e. ones that use more of our grammar.

For certain type structures, we have simple solutions. For example:

$$\text{CONCAT-BOTH-SUB}\quad \frac{S_1 <: S_3 \wedge S_2 <: S_4}{(S_1 \cdot S_2) <: (S_3 \cdot S_4)}$$

But others, particularly ones that involve parallelism and concatenation, are problematic.

But others, particularly ones that involve parallelism and concatenation, are problematic.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

But others, particularly ones that involve parallelism and concatenation, are problematic.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

$$@7/5 \cdot @16/13 <:? @10/6$$

Let's focus on the case with parallelism.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

What can we do?

Let's focus on the case with parallelism.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

What can we do?

Maybe we can take apart each parallel rate and check the individual parts separately?

$$@10/3 <:? @40/4$$

$$@12/5 <:? @10/5$$

Let's focus on the case with parallelism.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

What can we do?

Maybe we can take apart each parallel rate and check the individual parts separately?

$$@10/3 <:? @40/4$$

$$@12/5 <:? @10/5$$

Why might this not work?

We can attempt to solve this problem by

abstracting sound upper and lower bounds

for problematic constructors, and using those bounds in our subtype checks.

We can attempt to solve this problem by

abstracting sound upper and lower bounds

for problematic constructors, and using those bounds in our subtype checks.

$$@a/b <: @n_1/t_1 \parallel @n_2/t_2$$

$$@n_1/t_1 \cdot @n_2/t_2 <: @a/b$$

And indeed, there are some plausible ways we can do this. Again, let's focus on the parallelism case.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

And indeed, there are some plausible ways we can do this. Again, let's focus on the parallelism case.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

$$@a_1/b_1 <: @40/4 \parallel @10/5$$

And indeed, there are some plausible ways we can do this. Again, let's focus on the parallelism case.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

$$@a_1/b_1 <: @40/4 \parallel @10/5$$

$$@(40 + 10)/5 <: @40/4 \parallel @10/5$$

And indeed, there are some plausible ways we can do this. Again, let's focus on the parallelism case.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

$$@a_1/b_1 <: @40/4 \parallel @10/5$$

$$@(40 + 10)/5 <: @40/4 \parallel @10/5$$

$$@10/3 \parallel @12/5 <: @a_2/b_2$$

And indeed, there are some plausible ways we can do this. Again, let's focus on the parallelism case.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

$$@a_1/b_1 <: @40/4 \parallel @10/5$$

$$@(40 + 10)/5 <: @40/4 \parallel @10/5$$

$$@10/3 \parallel @12/5 <: @a_2/b_2$$

$$@10/3 \parallel @12/5 <: @(20 + 12)/5$$

And indeed, there are some plausible ways we can do this. Again, let's focus on the parallelism case.

$$@10/3 \parallel @12/5 <:? @40/4 \parallel @10/5$$

$$@a_1/b_1 <: @40/4 \parallel @10/5$$

$$@(40 + 10)/5 <: @40/4 \parallel @10/5$$

$$@10/3 \parallel @12/5 <: @a_2/b_2$$

$$@10/3 \parallel @12/5 <: @(20 + 12)/5$$

$$@(20 + 12)/5 <: @(40 + 10)/5 \checkmark$$

But this abstraction is neither

general nor composable

But this abstraction is neither

general nor composable

Not only does this abstraction not work over general stream types, but this abstraction just over the set of "raw" rates does not form a lattice, i.e. there is no notion of least upper bound/greatest lower bound for pairs of "raw" rates.

But this abstraction is neither

general nor composable

Not only does this abstraction not work over general stream types, but this abstraction just over the set of "raw" rates does not form a lattice, i.e. there is no notion of least upper bound/greatest lower bound for pairs of "raw" rates.

For example, let's try to compare some upper bounds on:

$$@10/3 \parallel @12/5$$

When might this be a problem? Here's a slightly more complicated example:

$$@10/3 \parallel @12/5 \parallel @20/10 <: @500/1000 \parallel @36/40 \parallel @75/60$$

When might this be a problem? Here's a slightly more complicated example:

$$@10/3 \parallel @12/5 \parallel @20/10 <: @500/1000 \parallel @36/40 \parallel @75/60$$

If there's no least upper bound and greatest lower bound for parallel pairs of "raw" rates, how do we choose what window size to use?

(Concurrent) Kleene algebra?

- The observation here is the Kleene algebra already looks very similar to our stream rate types.

(Concurrent) Kleene algebra?

- The observation here is the Kleene algebra already looks very similar to our stream rate types.

Regular languages?

(Concurrent) Kleene algebra?

- The observation here is the Kleene algebra already looks very similar to our stream rate types.

Regular languages?

Finite state and/or timed automata?

(Concurrent) Kleene algebra?

- The observation here is the Kleene algebra already looks very similar to our stream rate types.

Regular languages?

Finite state and/or timed automata?

Maybe, but it seems like directly checking inclusion on some kind of rate automata is very expensive.

possibly EXPTIME on window size?

Interesting directions

- Develop a model for rates with a regular (timed) automata.
- Prove a Kleene theorem that allows us to algebraically/equationally manipulate and simplify a regular language for rates.
- Introduce sound abstractions over the underlying automata model (some abstract interpretation) that further reduce the regular language to a boolean-like algebra of rates.
- Decide implication/entailment between two rates in the boolean-like algebra, aka subtyping.

Summary

- Rate limits, why they matter, and how types might help.
- Some ideas for how to solve the subtyping problem for one formulation of a rate type system.
- Pointers towards directions that may help solve some outstanding issues.
- Hydro as a compilation target for implementation and evaluation! Possible opportunities for collaboration? :)