How 500 Real Users Are Worse Than 3000 Bot Users

Adam Thornton

October 29, 2025

How 500 Real Users Are Worse Than 3000 Bot Users

NSF-DOE Vera C. Rubin Observatory

I work for NSF-DOE Vera C. Rubin Observatory (henceforth, ”Rubin Observatory” or just ”Rubin”).

Observatory Seen With Night Sky

It's an optical-band telescope with an 8.4m mirror, located on Cerro Pachon in Chile. It contains both the largest single piece of glass and the largest digital camera ever built (that we know of).

Night Sky Seen With Observatory

We will begin survey operations around the end of the year. The camera is 3.2 GPx; it will take about 1000 30-second exposures a night every night for 10 years.

Night Sky Seen With Observatory

The telescope field of view is 3.5 degrees, or seven full moons side by side.

During the ten-year survey we will collect 15-20TB per night of astronomical data. The primary catalog will be about 15PB, and total data holdings about 500PB.

Rubin Science Platform

Much of my work is on the Rubin Science Platform (henceforth, "RSP"). It provides hosted ad-hoc analytics to the Rubin science community:

- Jupyter notebooks with a kernel containing our data analysis pipelines.
- A search-and-visualization tool for Rubin astronomical data.
- API access to Rubin astronomical data and compute.
- Various other services.

My work often revolves around the Jupyter notebook service, and scaling that is what this talk is about.

Our JupyterHub/JupyterLab environment

- Based on Zero to JupyterHub.
- Individual user namespaces (and individual user domains).
- Prepulled, very large, Lab container images.
- Custom spawner.
  - Split session management and Kubernetes actions.
  - Because of individual namespaces, the K8s actor must be highly privileged.
- Custom Lab extensions.
- Deployed with Phalanx.

Data Preview 1

On June 30, 2025, we had our first public release of observational catalog data.

The primary site for astronomical community access is this deployment hosted at Google Cloud.

There are other Rubin Science Platform deployments on-premises; this talk is only concerned with our Google Cloud Platform-hosted production instance and its integration counterpart where we did the actual scaletesting.

Site setup

We run our JupyterLab based service on Google Kubernetes Engine, with autoscaling enabled.

Each node has 32 cores and 128GB of memory (n2-standard-32, investigating Autopilot) supporting a typical user pod of 4 cores / 16 GB RAM.

The main advantage of this environment is the ability to run analyses on our compute without having to download the data, which quickly becomes prohibitive at Rubin scale.

Name ↑	Status	Version	Number of nodes	Machine type	Image type	Autoscaling
core-pool-2d7b	Ok	1.32.7-gke.1079000	6	n2-standard-32	Container-Optimized OS with containerd (cos_containerd)	5 - 100 nodes per zone
user-lab-pool-9c8d	Ok	1.32.7-gke.1079000	5	n2-standard-32	Container-Optimized OS with containerd (cos_containerd)	1 - 100 nodes per zone

Scale Testing

We estimate our eventual user base to be about 10,000 people. They won't all be on the same RSP instance.

For Data Preview 1 we set a scaletesting goal of 3,000 concurrent users; this was a a deliberate over-estimate of the expected users in order to surface scaling issues.

3 months after Data Preview 1 we have 1,500 registered users, peaking under 500 concurrent active labs.

Testing methodology

We used our service, called `mobu`, that is able to run various payloads (primarily Jupyter notebooks) within the RSP.

It is mostly used for automated regression testing and for exercising new features as the analysis pipelines have evolved.

By design, a `mobu`-driven bot user is indistinguishable (from JupyterHub's point of view) from an astronomer logging in and doing work. Mobu uses the Hub API to establish a JupyterLab session and then can run Python code within JupyterLab kernels, either as entire notebooks or as individual statements.

Overall goal: get to 3000

Our victory condition was to get to 3000 simultaneous users each running a trivial Python workload. We did not expect to succeed immediately.

We began in late January 2025, and finished our JupyterHub/Lab testing in late April, doing one three-hour scaletesting session a week on our integration cluster.

Incidentally, scale-testing is a fun Friday afternoon team activity; recommended.

Initial Concurrency Results

Our very first test was 1000 users who logged in, did not do anything (not even start a pod), and logged out; success.

3000 users only failed because of our own lack of foresight: we'd designed `mobu` with the assumption that 1000 concurrent tasks would be more than enough. Hub user lifecycle management is nowhere near a bottleneck.

Then we actually started spawning Lab pods.

100 simultaneous users "running" a codeless notebook (no Python execution, just text) worked fine, and GKE autoscaling was performing as advertised.

1000 users failed: at 300 users we started to get spawn timeouts as the K8s control plane failed to keep up with the requests.

Remediation

Scaletesting in February and March was devoted to chasing down timeouts and internal Hub and controller errors.

- We found race conditions in our controller code that would have been difficult to find in a reasonably-loaded system.
- We had to use a less aggressive polling cadence to reconcile the controller's view of the world with reality.
- We realized that our practice of cloning tutorial repositories into user labs at startup was hitting GitHub rate limits at scale.

More memory and CPU for mobu and the Hub helped, but we were still getting timeouts from Lab-to-Hub communications.

The JupyterHub database

Eventually we realized that JupyterHub uses a single database connection, and all database operations are synchronous and block the rest of the process.

The only remediation we could immediately take was to drastically reduce the frequency of lab activity reports for culler polling.

This made it possible to get to our goal without significant reduction in functionality. Polling each user for activity every five minutes is gratuitous if our culling threshold is on the order of a week.

Desired JupyterHub enhancements

The single-threading on the database is becoming problematic. We can only reduce poll frequency so much.

As the Hub database page explains, work is underway to move to a database-session-per-request model.

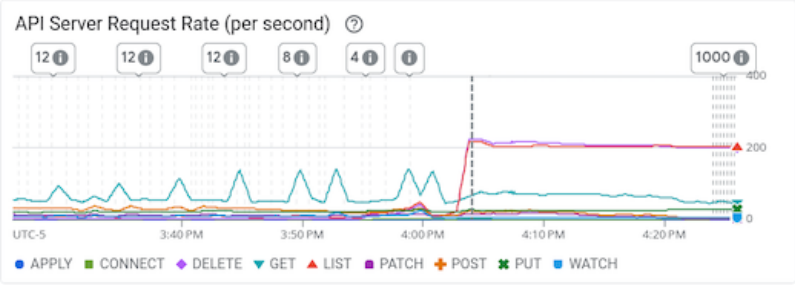
This will allow scaling the Hub horizontally, and we intend to be early and enthusiastic adopters when that becomes possible.

Interesting scaling items we found

IBM's jupyter-tools has some very useful tuning advice specifically for stress-testing JupyterHub. This is where, for instance, we got our initial recommendations for culling and activity polling.

GKE imposes a 200-requests-per-second limit on the K8s control plane. We smeared this out by dispatching pod startups in batches rather than all at once (more realistic anyway). However, this ultimately constrains the scale of a single cluster at GKE.

Ghcr.io imposes a high but finite rate limit for pulling container images. We worked around this by hosting the both the init and Lab containers in Google Artifact Registry, which did not exhibit this behavior.



Early April: meeting testing criteria

After we'd made the above changes we got 3000 simultaneous start-then-execute-a-print-statement-then-quit Labs.

At this point, with the Data Preview 1 deadline approaching, we declared victory and moved on to other services.

Data Preview 1 Reality

We got close to 500 users attempting to spawn Labs when Data Preview 1 went live. That was within our expectations, and maybe even a little disappointing (even if it's still about two percent of all the professional astronomers in the world).

This went less smoothly than we had hoped: spawn failures started to occur at a far lower user count (about 300) than we had achieved in scaletesting.

The problem was in the proxy, not the Hub or the controller. It wasn't the memory exhaustion we'd already seen and fixed.

How Are 500 Real Users Worse Than 3000 Bot Users?

The very simple answer: **bots log out**.

Configurable Hub Proxy and Websockets

Abandoned open websockets wreck CHP v4.

Human users, despite the fact that we give them a perfectly good menu item to save their work and shut down their pod, don't use it. *At best* they close their browser tab, and most of them don't even do that.

CHP v5 (the new default in z2jh) addresses this problem adequately. After adopting v5, that concurrency problem vanished and we haven't seen it again.

We have since been coping well with 350-ish simultaneous users doing science work.

Post-Data Preview 1 lessons

We are also validating assumptions about data access. This involves notebooks that make large queries that require a lot of memory.

We found we needed to make our overcommital ratio more tunable. A normal real-user workload allows a high overcommital ratio.

If your workload is 50 bot users all simultaneously doing very memory-intensive work, when the Labs all ask for their whole memory limit at once (even though each process stays just under its limit), node memory runs out.

Most of our remaining bottlenecks are neither in Hub nor Lab but in the services notebooks consume.

Your Platform Probably Isn't Just A Notebook Service

At the very least, you probably have some sort of A&A sytem, a Notebook service, and a data source. You may have services that sit in between your notebooks and your data store. We certainly do.

If so, you will likely need to (internally) rate limit access to other services, especially if they perform significant computation on the user's behalf.

We have Gafaelfawr for this (thus it's built into our A&A system). You're going to want to use something similar.

Challenging Usage

- A few people tried to mine crypto. With four cores and no GPU, I'm pretty sure they didn't make much money. Google is great at detecting this, so don't try this at home, kids.
- There were some real outlier users of our APIs; e.g., those harvesting data to build training sets. We had to scale-up some of our back-end workers to allow them to proceed at a reasonable pace without crowding out other users.
- API rate limits can have some perverse effects; e.g., penalising people who do short, well-thought-out catalog queries - which are what you want! Concurrent queries in flight are a fairer measure.
- You absolutely need disk quotas if you provide per-user persistent storage. Before we imposed quotas, one user used more disk space than all the thousand others combined.

Summary of Scaling Lessons

In rough order of importance:

- Use CHPv5 (now default, hurray!).
- Reduce polling frequency where you can get away with it.
- Quota internal resource usage.
  - Persistent storage
  - CPU and memory
  - Resource-intensive internal services
- Stay alert for signs of abuse or clueless enthusiasm and know how you will deal with misbehaving users.

Sometimes you have to downgrade a few users' experience to keep the overall experience tolerable for everyone.

Links

- Rubin Key Numbers
- Nublado (docs)
- Phalanx (docs)
- Gafaelfawr (docs)
- General Rubin Observatory talk (source)
- This talk (pdf) (source)