NEON

Serverless Postgres:

the journey from ~1s startup time to 10s and back again

Alexey Kondratov, Berlin, Feb'25
PostgreSQL Meetup Group

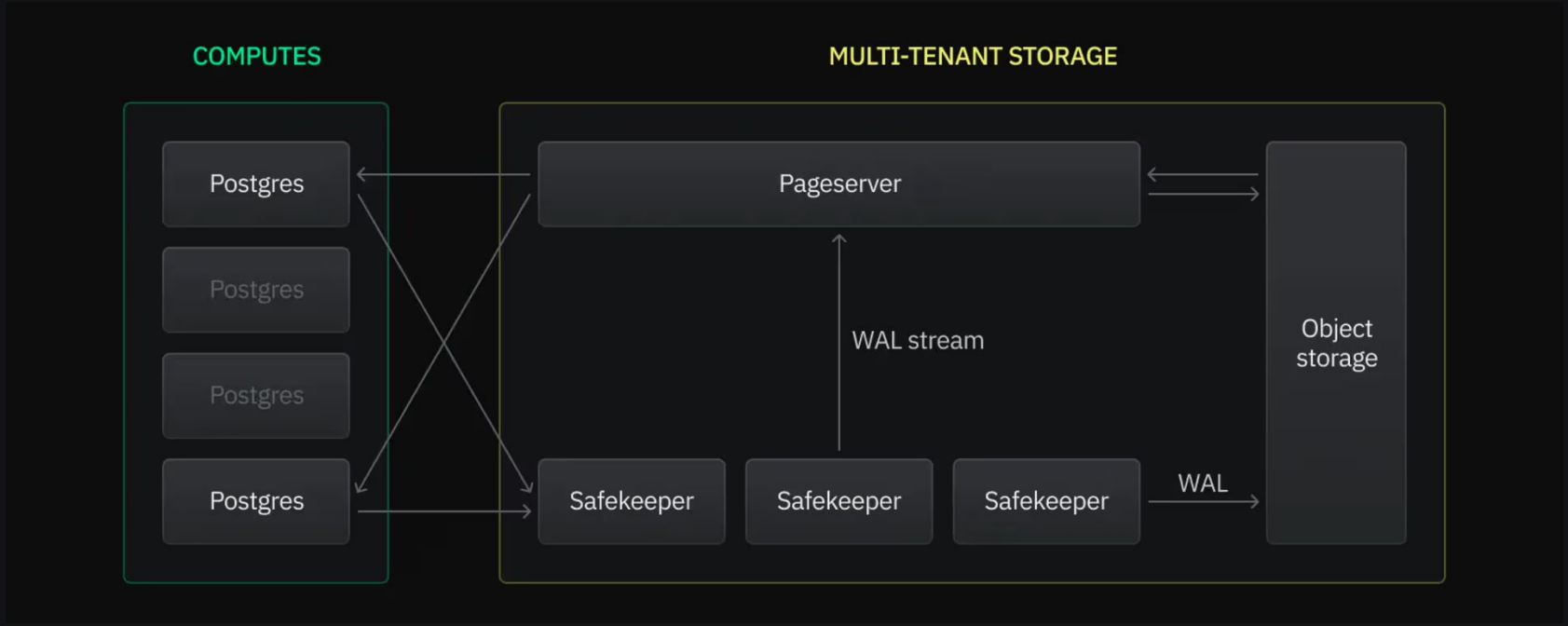
About me

Lead Software Engineer @ neon.tech

- GSoC'17 Postgres participant
- Postgres contributor in ~2018-2021
- At Neon, worked on provisioning, configuring and monitoring Compute (Postgres) instances
- github.com/ololobus
- linkedin.com/in/alexeyko

What is Neon?

Brief architecture overview



I. Compute is **stateless**
II. Data persistency is guaranteed by **distributed storage**

Why Serverless?

Sure, there are still ***servers*** somewhere

1. Stateless Compute

Compute is running as a **stateless virtual machine** (VM) in Kubernetes (k8s). No persistent volumes, Stateful Sets and complicated k8s operators. No WAL redo at start.

2. Distributed storage

Storage scales independently from the Compute. **Replica Computes are cheap** — they do not requires new copy of data. **Branches are cheap** — they are copy-on-write.

3. Autoscaling

You set limits — VM resources are scaled automatically based on working set size, load average and other metrics/signals.

4. Scale-to-zero

When you do not use your Compute it is auto-suspended. Storage and unused branches are archived to the block storage after a longer period of inactivity. When you need it back, you just connect to the same connection string — we seamlessly bring it back under the hood.

Compute startup time

Latency between the moment client initiated connection and Neon **Compute is ready to accept connections**

or roughly time-to-first-query

Why does it matter?

- **Onboarding experience** → ready-to-use Postgres in a few **seconds** instead of ***minutes***.
- **Scale-to-zero** → get your idle instance back quickly.
- **Branching** and **dev environments**.

Cold starts

The beginning

EoY 2021

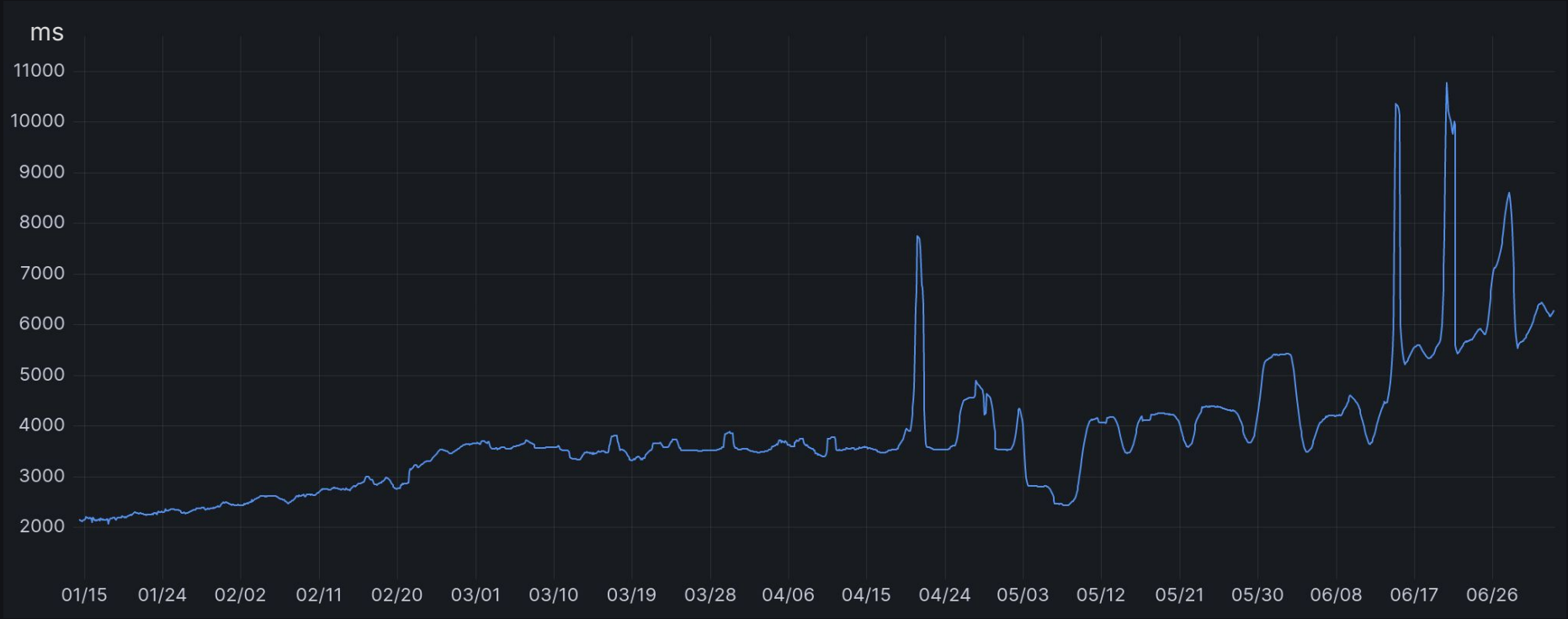
Naive implementation of the ‘Serverless’ Compute with **scale-to-zero** in **k8s**. Zero optimizations, but **2-3s startup time**. **Good enough.**

EoY 2022

Dropped the invite gate. The first organic flow of new users. **Still good.**

Cold starts

Scaling up



It’s getting worse over the course of H1’2023 as we onboard more clients →

Cold starts

K8s is slow. Why?

- Typical breakdown for Service + Deployment creation in k8s was: ~1s (to get a running container) + 4s (networking) + ~500ms (Neon-specific overhead).
- Default k8s CNI in AWS EKS is **Amazon VPC CNI**, which is based on `iptables` rules, so it scales poorly with the number of addresses.
- Control plane in a managed k8s cluster is a black box. When it's slow you have bad time guessing why. And it's getting slower with a number of running deployments.
- The same seems to apply to Azure AKS.

Cold starts

Pre-provisioned computes

EoY 2021

Naive implementation of the ‘Serverless’ Compute with **scale-to-zero** in **k8s**. Zero optimizations, but **2-3s startup time**. **Good enough.**

EoY 2022

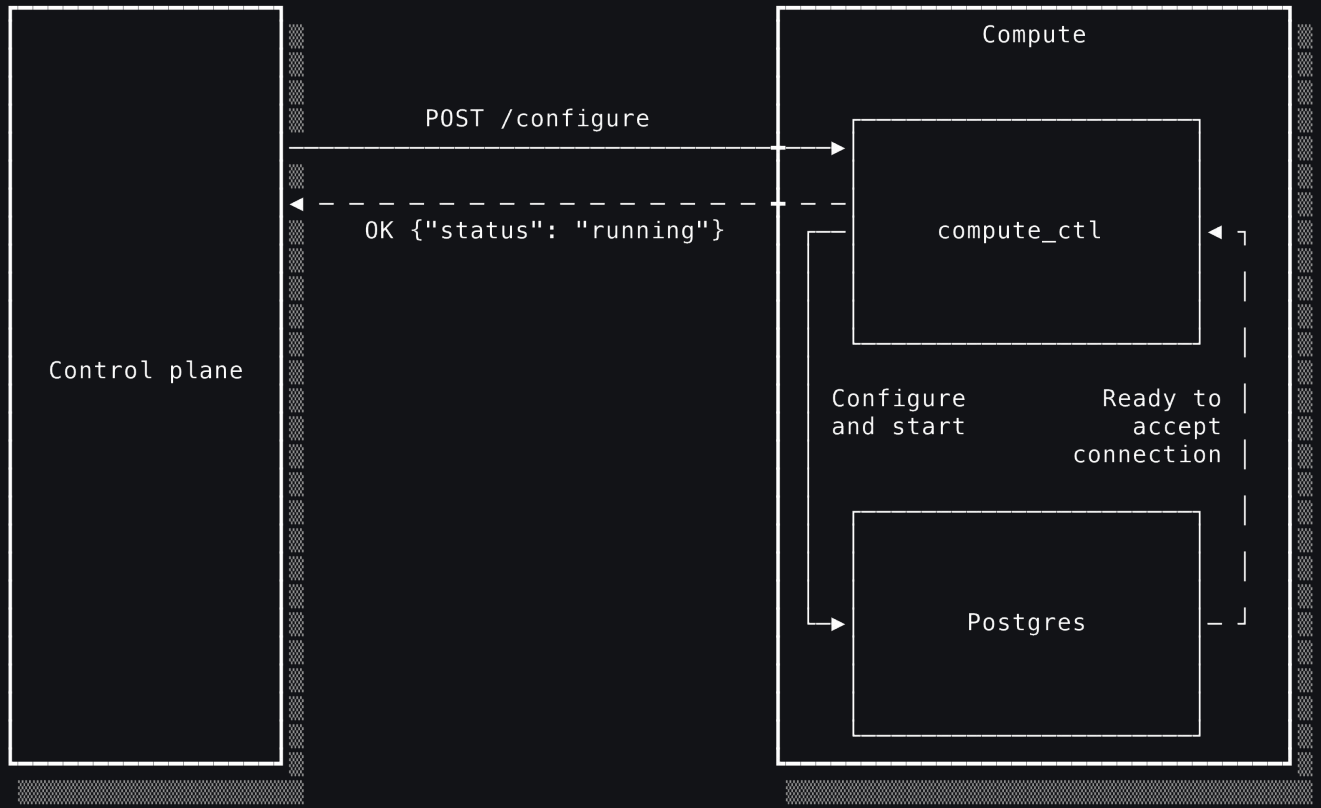
Dropped the invite gate. The first organic flow of new users. **Still good.**

Mid-2023

Jumps up to ~10s. We implement and roll out a **pool of pre-provisioned computes.**

Pre-provisioned computes

Configuration



Pre-provisioned computes

Startup latency: before/after

Pre-provisioned computes

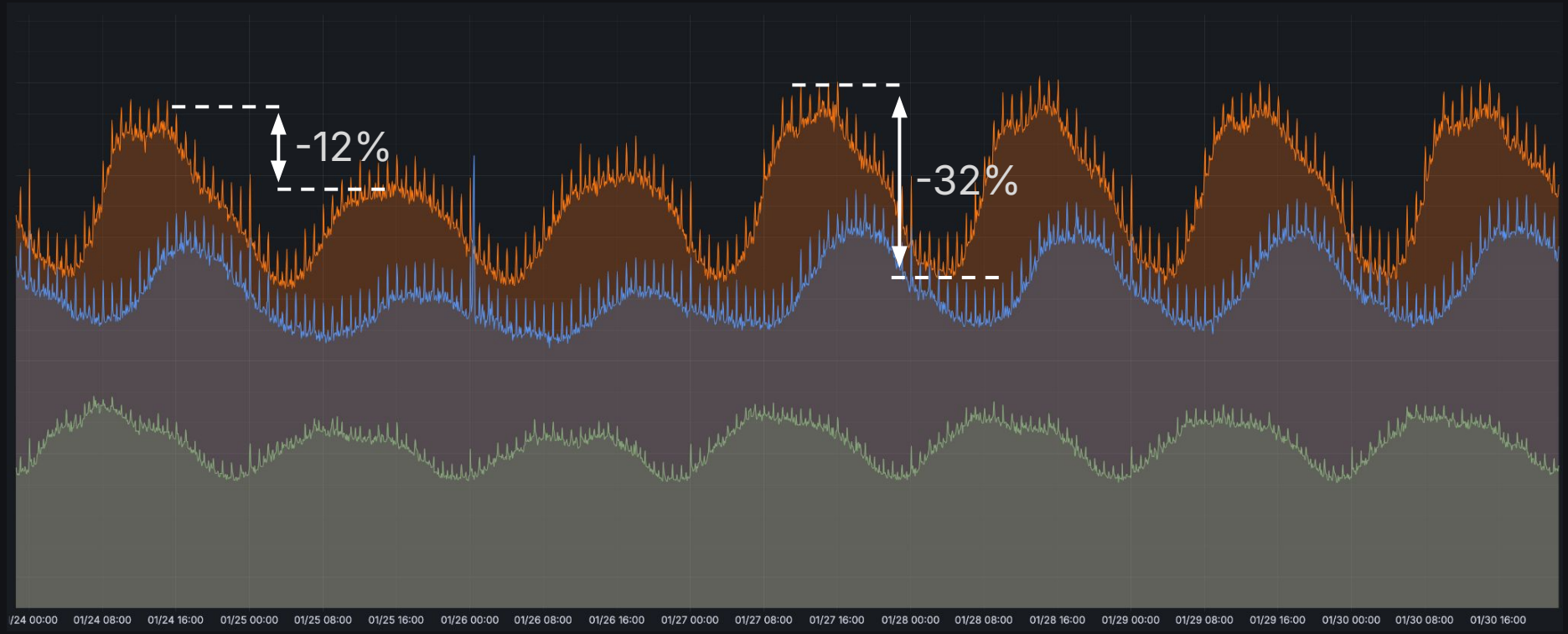
Problems

- **Seasonality**: hourly and daily. Demand for computes fluctuates significantly day-to-day, hour-to-hour.
- **Steady trend**: every 1-2 months we naturally have more compute starts.
- **Unexpected trends**: some partner may start onboarding new or migrating old clients to Neon, so we can get 100, 1.000, or 10.000 new instances within a relatively short timeframe.

Regular **manual tuning** and dramatic **over-provisioning**

Pre-provisioned computes

Problems: seasonality



Pre-provisioned computes

Pool size prediction

EoY 2021

Naive implementation of the ‘Serverless’ Compute with **scale-to-zero** in **k8s**. Zero optimizations, but **2-3s startup time**. **Good enough.**

EoY 2022

Dropped the invite gate. The first organic flow of new users. **Still good.**

Mid-2023

Jumps up to ~10s. We implement and roll out a **pool of pre-provisioned computes.**

Early 2024

Fully automated pool size management with load prediction and **hit rate >99%.**

Load prediction

Forecasting algorithm

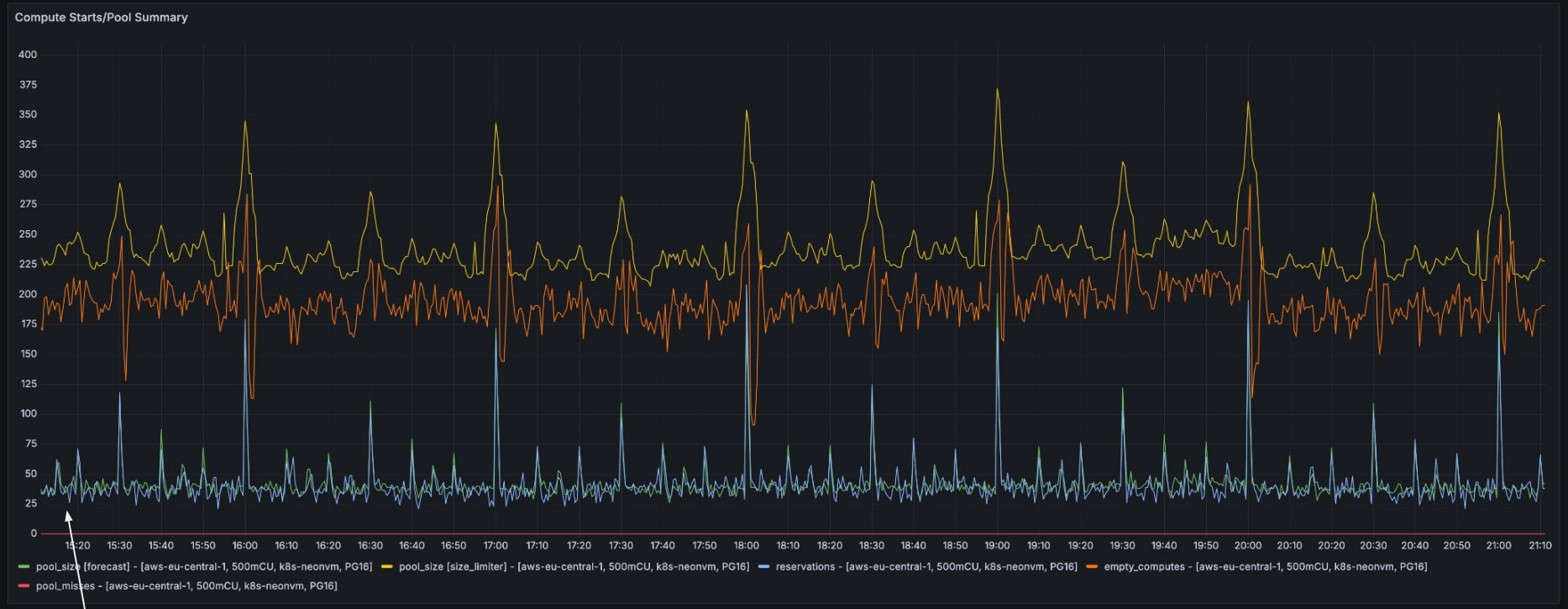
We train the data for every hour and produce the forecasts for the next hour. Current model is as following:

- Forecast the next hour's start compute count using Unobserved Components model with daily and weekly seasonalities. Also handles the overall trend.
- Decompose the sub hourly time series into trend, seasonal and noise components with Seasonal-Trend decomposition using LOESS (STL).
- Get rid of the noise, normalize the sub hourly series per hour, take the median of each bucket for the last 3 hours and normalize it again.

Credits to Muhammet Yazici for implementing that github.com/mtyazici

Load prediction

Pool size: results



Forecasting is **very** accurate, but we still overshoot pool sizes. Why?

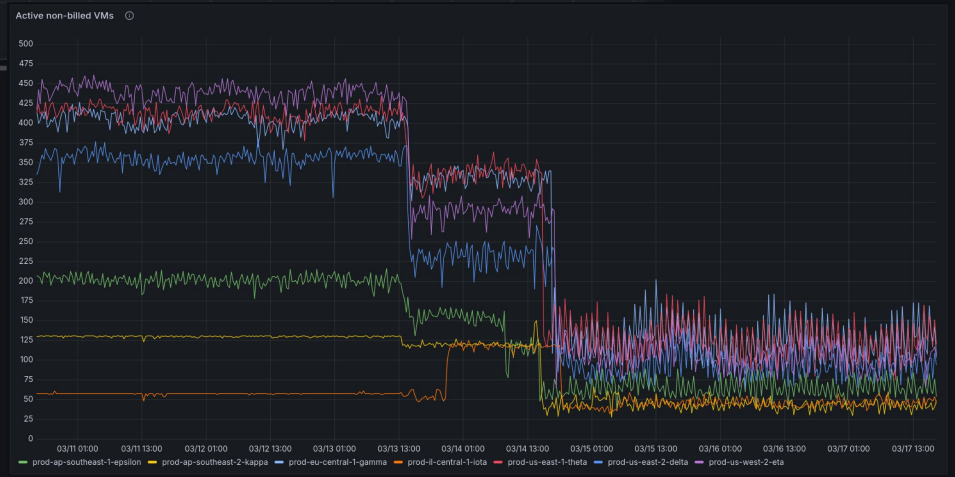
Load prediction

Data pipelining

- Compute VM provisioning can be very slow and take up to **minutes**, but we need computes to be created ahead of time.
- We had to implement pipelining and trend adjuster on top of the raw forecasting data:
  - Smoothen forecasting data (to reduce spikes)
  - Adjust by trend (to account for unexpected trend changes)
  - Bump by fraction
  - Bump by constant
  - Limit size (to avoid extreme values if our prediction is off)

Load prediction

Number of pre-provisioned computes: before/after



Three times reduction of the number of pre-provisioned instances, while still maintaining a >99% hit rate.

What’s next?

Caches

Compute is not completely stateless

- Many performance-critical Postgres GUCs cannot be changed without restart, e.g., `shared_buffers` or `max_connections`.
- We allocate some of them (like `max_connections`) as per the max autoscaling limit; and use swap to deal with steep memory allocations by Postgres, until autoscaling kicks in.
- We replace shared buffers scaling with local file cache (LFC), which could be scaled at runtime.
- Unfortunately, this means that restarting Postgres in 400ms satisfies HA needs, but performance and query latency might be degraded due to dropped caches → we need to implement LFC prewarming and ‘traditional’ failover to the ‘warm’ replica.

Further optimizing startup times

Both VM starts and acquiring computes from pool

- K8s can be fast*er*
  - Some non-default CNIs scale much better. For example, Cilium, which is **eBPF-based** and presumably doesn't have the same scalability issues as Amazon VPC CNI.
  - Self-host k8s or **deploy a separate k8s control-plane** within managed cluster.
- We currently download SLRUs (e.g., clog or commit log) as a part of basebackup at start. Yet, they might be huge in some cases, so we plan to switch to the on-demand SLRU download.
- Consider unifying all Postgres versions under one VM image.

COGS

Cost of goods sold

- We do not fully benefit from precisely forecasting the load
  - Yes, we are much less over-provisioned.
  - But when we suspend random ~30% of the fleet overnight, we end up with the same number of k8s worker nodes, just sparsely loaded.
- Using live-migrations is likely the answer.
- Another option is to use a dedicated node group for a highly seasonal load, so it could be scaled down when the demand is low.

Thank you!

Questions?