+ tableau

Tableau at the speed of EC2

Getting the most out of your Tableau Server in AWS, version 1.0

Chris Bullock, Miranda Osterheld,
Scott Smith, and Zach Ahrens

Table of Contents

The mission: Tableau on AWS EC2

You've purchased Tableau Server, and decided to run it in an Amazon Web Services (AWS) Elastic Compute Cloud (EC2) environment. Now, all you need to do is choose a good fit to support your anticipated workload of Tableau content and user community. As you explore the options, you realize that the array of choices is both stimulating and perhaps a little intimidating. Many of the choices (EC2 instance types) could work well, but how do you get a good sense of what is likely to work optimally for your needs?

This is where Tableau can be your guide as well as the goal of your deployment. Why? Everything in the cloud can be measured, creating data. Tableau helps people see and understand their data. Therefore, Tableau is your guide to maturity in AWS, starting with the performance of Tableau Server itself. By using a free load testing solution called Tabjolt, you can test EC2 instance types, collect the data, and analyze it, equipping yourself with key information for selecting your EC2 instance.

In this experiment, we set out to do just that. We load tested Tableau Server (version 2018.1) on a set of AWS EC2 instance types using Tabjolt. By testing the Server with varying amounts of concurrent users, we were able to collect and compare transactions per second (TPS), error rates and load times for each instance type. These results provide a useful guide for EC2 evaluation and a template for running similar tests with your organization's workloads.

The method: Tabjolt as a guide to EC2 selection

Tabjolt is a load testing solution provided by Tableau to measure Tableau's unique VizQL processing at scale. It is a point and run load generator built on top of JMeter, and is available as a free download as-is from GitHub. Tabjolt is commonly used by both our customers and Tableau itself as a way to test upgrade validation and for growth planning.

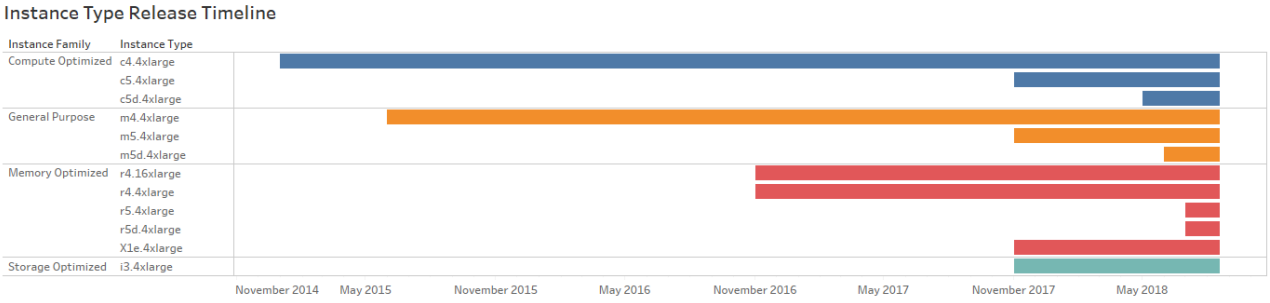
A version of this experiment was done by Russell Christopher using Tableau Server version 9.0. He has documented his research in his blog, and we recommend it as a good starting point to understand Tabjolt elements we will build on here. We especially recommend the summary of lessons learned here. The key measures we will use Tabjolt to focus on are throughput (TPS), error rate, and response time. Details are documented in The Journey section.

Although Tabjolt is a simple load testing solution, there are many things to keep in mind when producing and interpreting results. In the end, we wanted to provide a testing approach that is usable, simple, and focused on providing a relative benchmark which can be observed across EC2 instance

types. We also wanted a test set spanning the General Purpose, Compute Optimized (better processors), Memory Optimized (more RAM), and Storage Optimized instance families in AWS.

Our EC2 test set

EC2 launch options are grouped by specific instance types into instance families. We selected a set of instance types to cover these families. Our choices were a combination of well utilized selections such as the m4.4xlarge and its newer version, the m5.4xlarge, and emerging instance types we wanted to test such as the x1e.4xlarge. The innovation is amazing as new instance types were released during our testing. Detailed information about EC2 instance types can be referenced here from AWS.



A single Tableau Server production deployment requires a minimum 8 cores of processing, so we focused on the equivalent 16 virtual core (4xlarge) options, with one exception using the r4.16xlarge. We stocked each instance with 500 GB SSD storage. Following are some additional vitals.

General Compute

- m4.4xlarge — 2.3 GHz Intel Xeon E5-2686 v4 (Broadwell) processors or 2.4 GHz Intel Xeon E5-2676 v3 (Haswell) processors, 16 vCPU (equivalent to bare hardware 8 core), 500 GB SSD Elastic Block Storage (EBS)
- m5.4xlarge — 2.5 GHz Intel Xeon Platinum 8175 with new Intel Advanced Vector Extension (AXV-512) instruction set, enhanced networking throughput
- m5d.4xlarge — The m5.4xlarge with NVMe-based SSD block level instance storage physically connected to the host server

Compute Optimized

- c4.4xlarge — High frequency Intel Xeon E5-2666 v3 (Haswell) processors optimized specifically for EC2

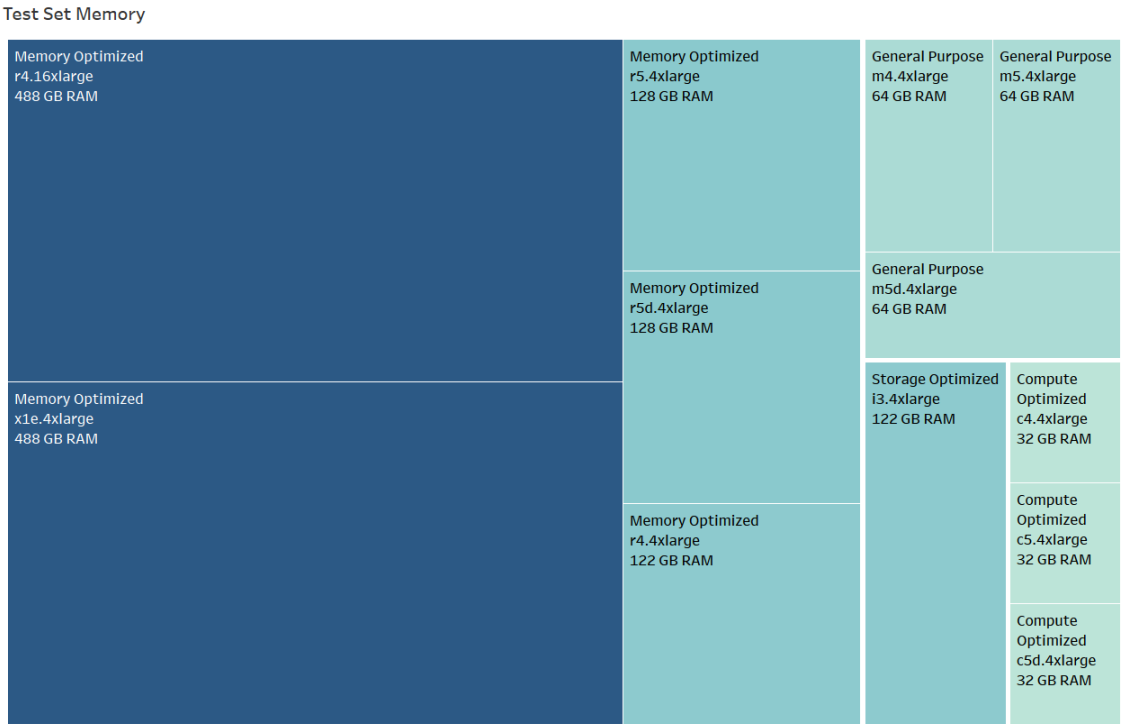
- c5.4xlarge — 3.0 GHz Intel Xeon Platinum processors with new Intel Advanced Vector Extension 512 (AVX-512) instruction set
- c5d.4xlarge — The c5.4xlarge with NVMe-based SSD block level instance storage physically connected to the host server

Memory Optimized

- r4.4xlarge, r4.16xlarge — High Frequency Intel Xeon E5-2686 v4 (Broadwell) processors
- r5.4xlarge — 3.0 GHz Intel Xeon Platinum processors with new Intel Advanced Vector Extension 512 (AVX-512) instruction set
- r5d.4xlarge — The r5.4xlarge with NVMe-based SSD block level instance storage physically connected to the host server
- x1e.4xlarge — High frequency Intel Xeon E7-8880 v3 (Haswell) processors

Storage Optimized

- i3.4xlarge — High Frequency Intel Xeon E5-2686 v4 (Broadwell) processors (same as the m4 instance type) and Non-Volatile Memory Express (NVMe) SSD-backed instance storage optimized for low latency, very high random I/O performance, high sequential read throughput and provide high IOPS



Our test goals

The purpose of this test was to provide a simple yet meaningful baseline for single Tableau server instance tests on various EC2 instance types. We did not tweak the installations or Tabjolt much as we wanted the approach to be as simple and repeatable as possible.

What we wanted

- Single server instance types with 16 vCPUs
- Default Tableau server installation options
- Focus on three simple measures: transactions per second (TPS), response time, error rate
- Solid test durations of 30 minutes
- Broad load scaling—small to very large numbers of users; we wanted to find the breaking points of the virtual machines
- Simple networking configurations with a focus on test results first
- We used Windows 2016 base servers for this experiment although Tableau Server also runs on Linux servers

What we didn't want

- Tableau cluster testing
- Custom configuration different from default installation
- Tableau extract creation benchmarks
- How specific calculations work in different environments
- Latency answers, such as across regions

The results: seeing and understanding our data

Let's begin this section with the usual disclaimer: this was a fixed set of tests we ran with our results. It does not represent your workload or the value of specific EC2 selections for their intended purposes. With EC2 pricing so attractive and the time investment to load test more minimal than ever, we strongly encourage you to review our approach and results, and do similar testing on your specific workloads. We

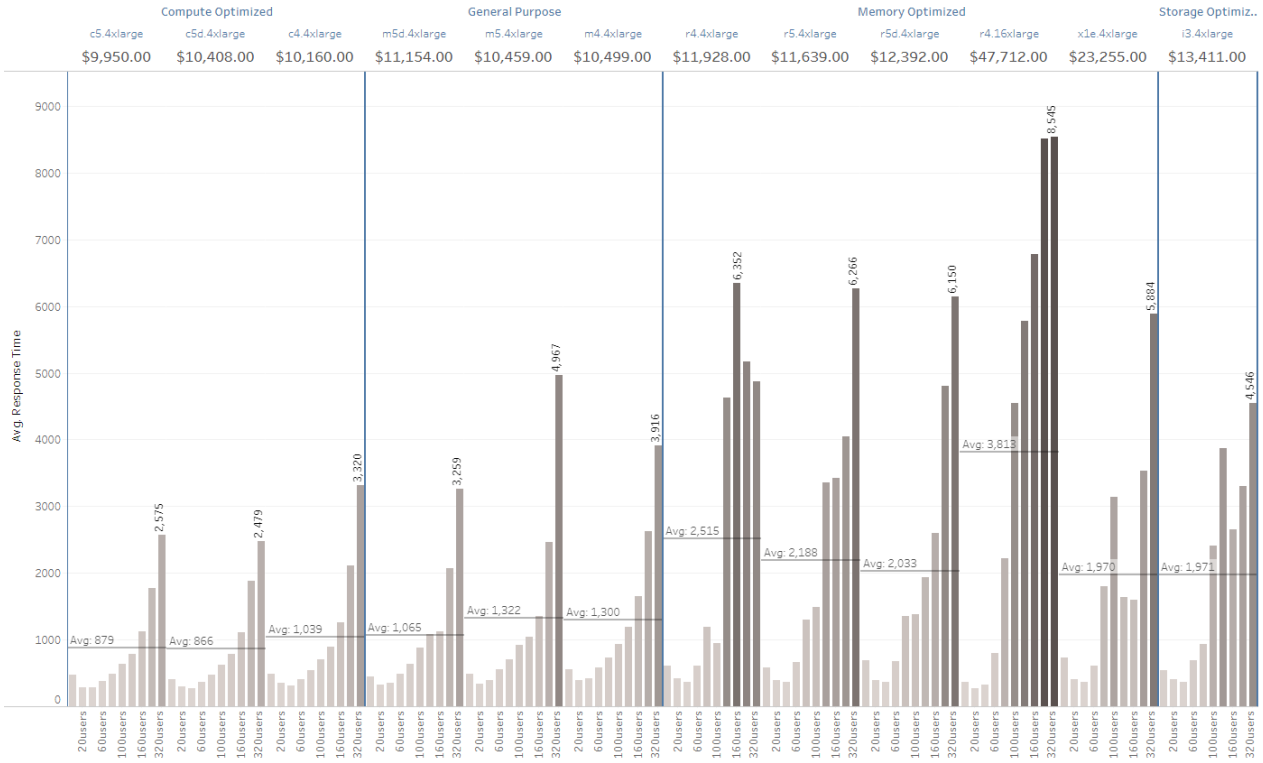
trust that through our work you'll find this is very doable; it is the only way to know your own workload performance.

That said, here are some key highlights overall:

- Our observations overall were that Tableau server 2018.1 and the Hyper data engine performed with excellence. The load times were extremely impressive even after attempting to push the servers with very high c values—the proxy for extremely high concurrent, active users. For example, 320 active users (16 threads) on an 8-core.

- Clean workbooks (that can be cached) let you put a lot more users on your server. RLS and workbooks that update often are, by nature, more work for the server. So the more complex/user-specific your workbook design, or the less performant your data sources, the fewer total users your server will support. We legitimately tried to push our servers more by including things like complex calculations and very large crosstabs within the dashboards, and we still didn't get "bad" load times or error rates. The actual values of our metrics are specific to the server deployment we tested, but the general trends across the instance types are favorable.

Okay Tableau enthusiasts... Let's analyze the data! Key metrics are summarized below, and we strongly urge you to dig into the data yourself here.

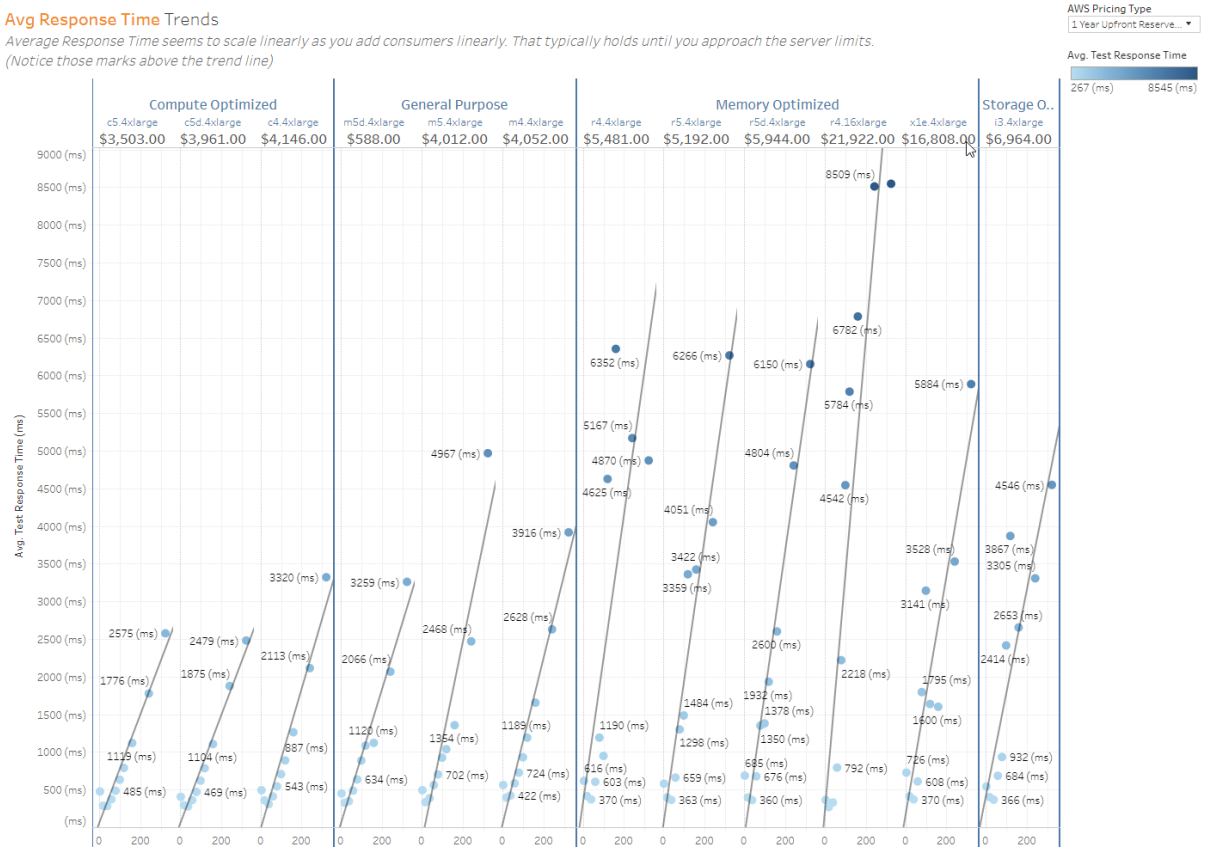


Response times

The Compute Optimized (C Series) EC2s provided the best average response times (how long it took to load a viz) followed very closely by the General Purpose (M Series). The Memory Optimized (R Series) lagged notably behind. This appears to support the idea that appropriate resources, strong processing, and caching outperforms excessive memory. The i3 and xle instances had results similar to the R series, but they had a lot more variance within each of the concurrent user loads.

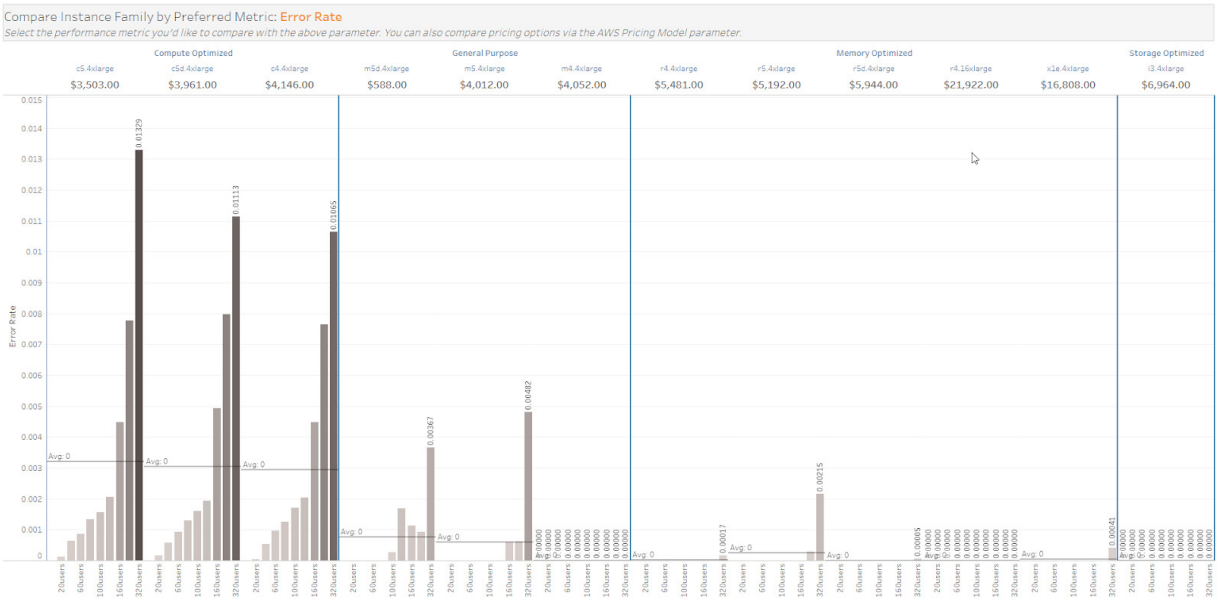
Note: There is actually an improved response time moving from the five users tests to ten users. This comes a result of the server "warming up" during the beginning of the test (all concurrent user "steps" were performed in sequence). We also observed that once moving past the "warm up" period, response time continues increasing with added user load as anticipated.

We also saw that our server performance scaled linearly as we added concurrent users. Note the R^2 and P values both indicate a good match for the linear trends depicted below. Only at their highest ranges of concurrent users, did we see performance begin to slow at a faster rate. If you find similar results in your deployment, this can be a valuable estimation for planning Tableau user growth.



Error rates

Although the Compute Optimized instances were the most performant overall, they also produced the most errors. This was anticipated as they have the lowest memory allotment and these error rates are consistent with Tableau Server hitting memory limits. By contrast, Memory Optimized instances didn't see errors until extremely high thread counts. The General Purpose M Series can be looked at as a happy medium. Astonishingly, the m4.4xlarge had no discernible errors across our thread counts!

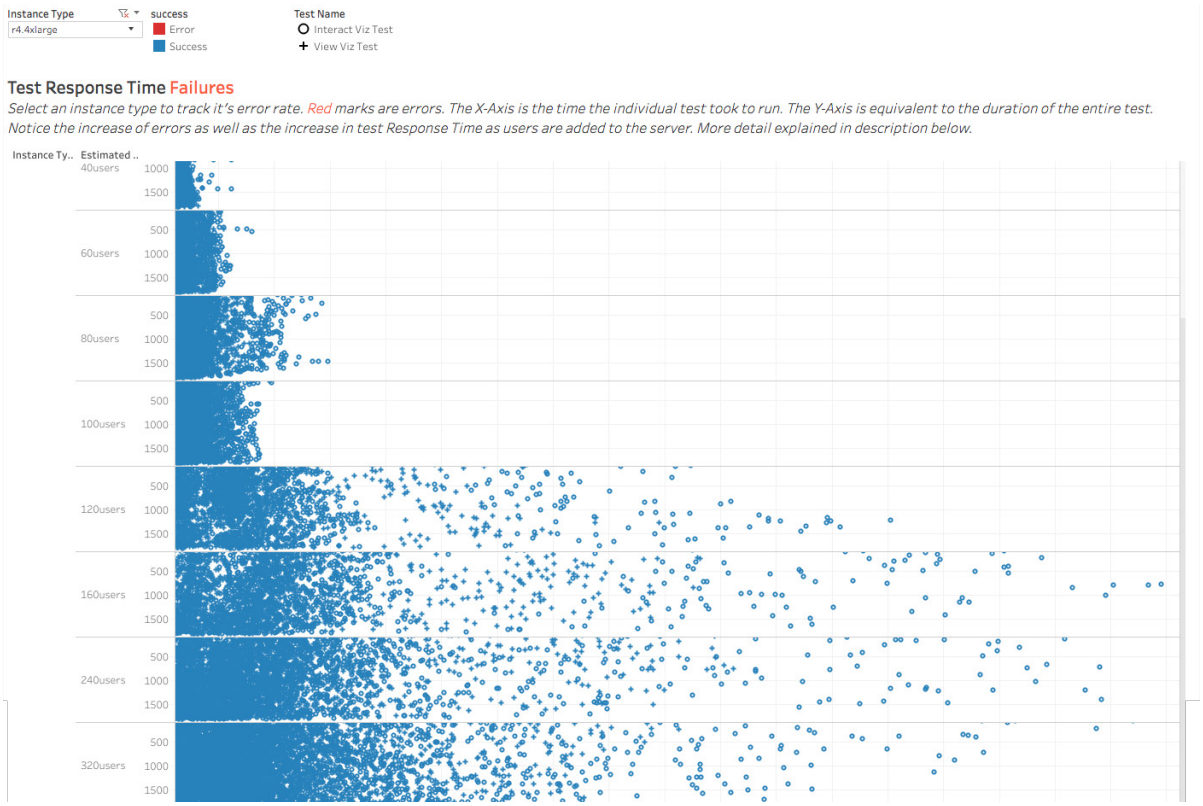


Test response time and error rate

On this next view the vertical axis indicates how long a test had been running and the horizontal axis indicates how long an individual request took to be fulfilled (how long it took to load the viz). This viz allows you to check for two things. First, it shows if the VizQL Process got so overloaded that it had to restart. And secondly, it shows how increased user concurrency—especially when prolonged—impacts the instances.

In Tableau Server, if the VizQL Process gets completely overwhelmed by bandwidth, it automatically restarts and picks up where it left off. When the process shuts down, it halts any test currently running, throwing an error result. This leaves a horizontal red line of errors. To see for yourself, check out a few examples that occurred on the c4.4xlarge instance. Impressively, the VizQL Process immediately restarts and is able to complete the following tests within seconds.

Secondly, you can see when response times start to spread out for each instance, whether or not it's correlated with increased error rates. For example, the r4.4xlarge starts to see increased spread in response time around 120 users. All previous concurrency rates were fairly clustered in terms of response time.

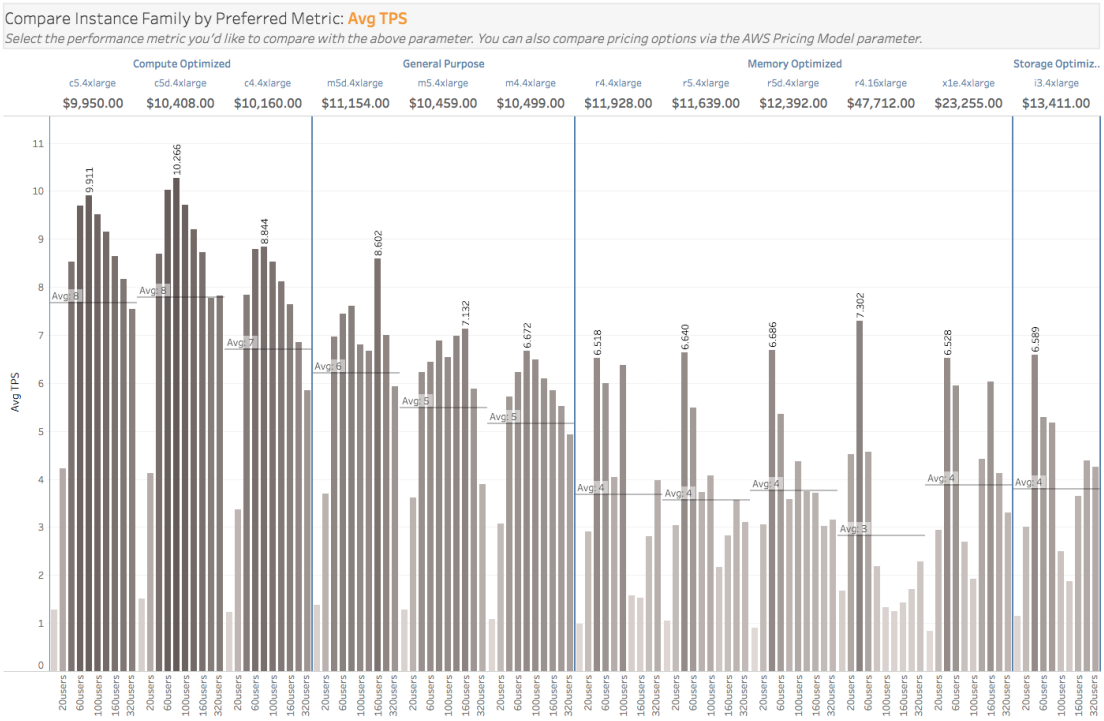


Alternatively, the m4d.4xlarge started to see increased response time at 100 users after being under load for a long period of time. You can see the line of longer response times towards the bottom of the 100-user pane, which is toward the end of the 30-minute test.

This visualization is especially useful to see patterns in performance variation across an instance type and within a user load. Understanding how a particular server reacts on a request-by-request basis during a prolonged period of load (in our case, 30 minutes) is crucial for predicting variation in user experience.

Throughput thresholds

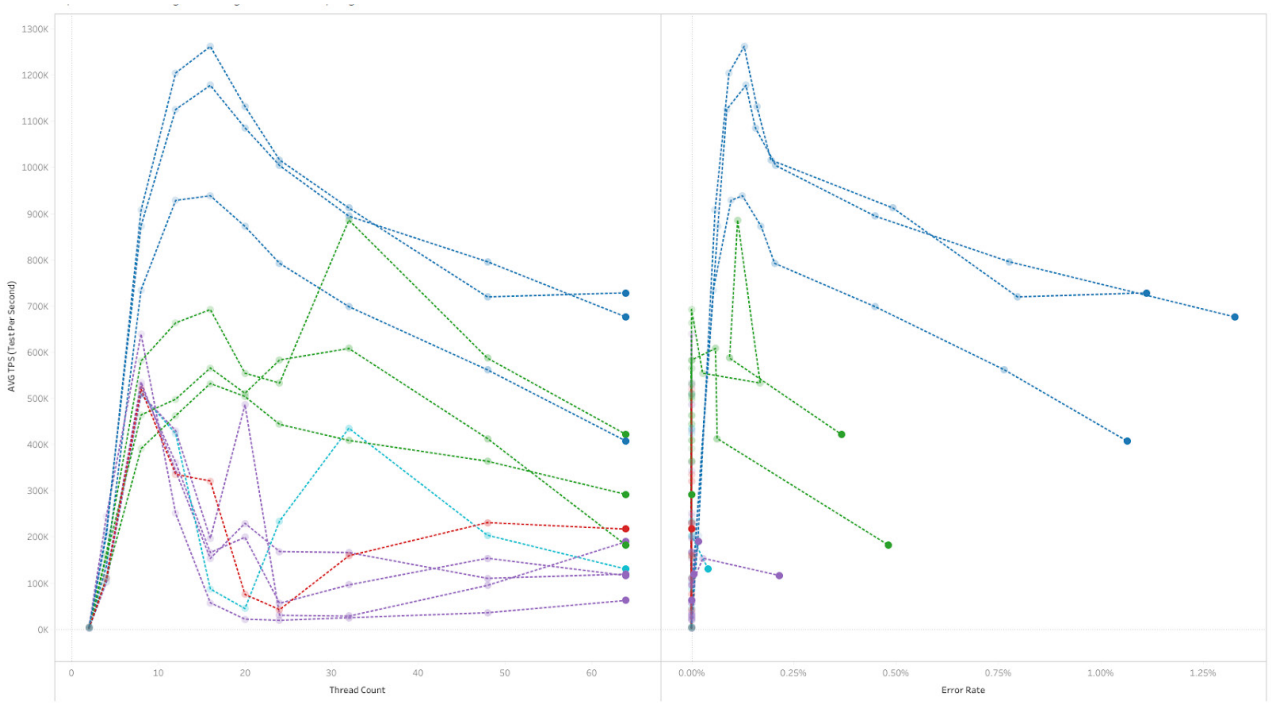
Our Compute Optimized instances had the highest average throughput, and both the Compute and General instances follow a similar bell curve of Avg. TPS. The Memory Optimized Series and the ie and x1e have more variance across thread counts.



This next view is especially useful for tracking throughput as opposed to response time. It identifies the throughput (average Test Per Second) of each server as concurrent users grow. You'll notice that as users are added to the server, performance dips off at different rates for each instance type.

They reach a point (thread count) where their throughput maxes out, and then the throughput decreases as more threads are added. Tableau Server's VizQL cores take additional overhead for the repository and logging, but it's clear that at some point we reach the optimum number of concurrent users when all available memory is saturated. For the Compute Optimized instances, this happens consistently around 80 concurrent users. The Memory Optimized (R and x1e Series) and Storage Optimized (i3) all have much more variable results across the thread counts.

It also highlights the fact that average performance is significantly impacted by error rates. When it begins running into an error, a server churns through the response, trying it's best to get the result. Ultimately it errors out, but adds additional time in the process. Not to mention that if the VizQL Process needs to restart, TPS will decrease. That's another reason why the Compute and General Optimized instances are impacted most dramatically as users are added.



Keep in mind

Before reading the summary, it's important to remember one last thing. Average response time, error rates, and throughput aren't the only important metrics to examine. You can also test for things like extract success rate and duration with the Server Admin views built into Tableau Server. And there are a host of tools available to track network latency. Remember that all of these values still produce measurable data, even in the cloud. And since they can be tracked, they can be planned for. Therefore Tableau can help you plan for and optimize your Tableau Server, providing a data-driven approach to optimizing the experience of your own Tableau Community.

Providing additional resources to your server has never been easier than it is now. As of April 2018, our new Subscription offering is completely user-based. This leaves the cost of hardware as the only financial barrier to architecture improvements. And that's never been more affordable than with public cloud offerings. Furthermore, with our updates to the Tableau Server Monitor command line (2018.2 for Windows, 2018.1 for Linux), you can even configure your server with Hot Topology to adjust your architecture dynamically.

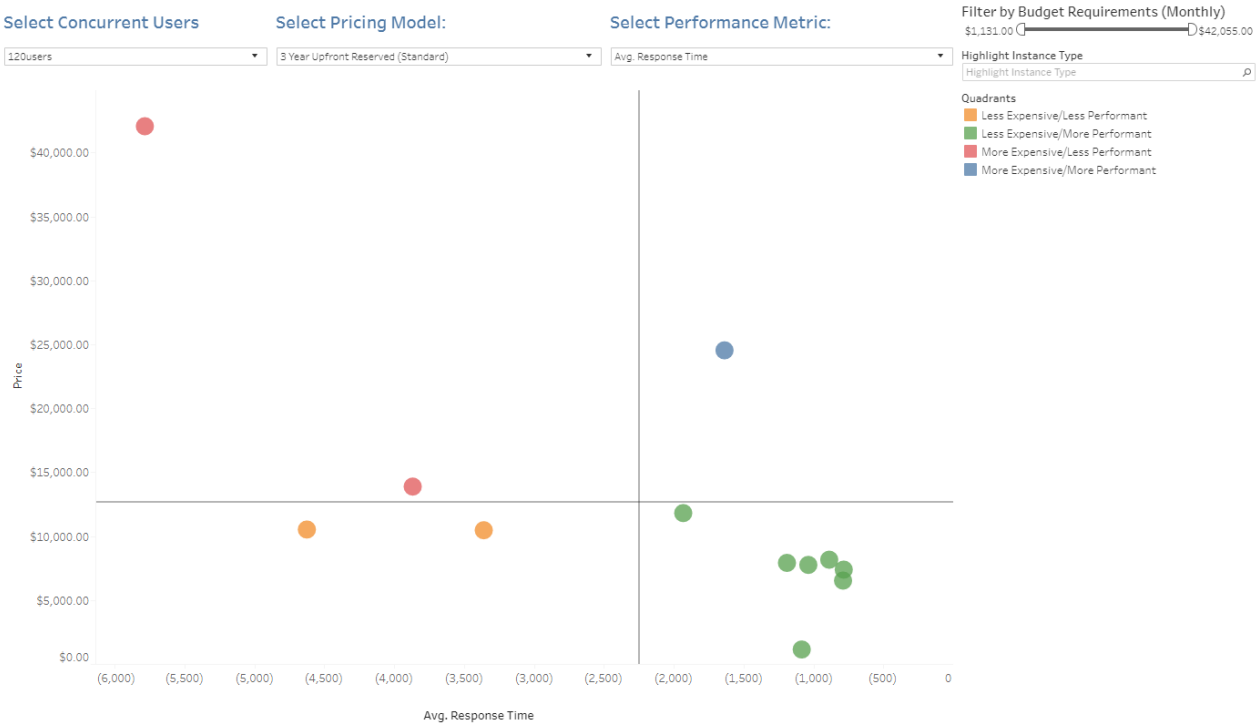
Summary

To recap, we saw a lot of what we expected. Compute Optimized and General Purpose instance types were able to crank through queries fast with their higher compute power. Average test time tended to

scale linearly as concurrent users were added, regardless of instance selected. But errors tended to have a higher impact on average performance tests, although exact variance depended on instance selected. So make sure to keep increased capacity in mind when preparing your own tests. We didn't truly push the memory limits of our server during our testing. But in your tests, you may see the R Series begin to shine in more complex deployments because of their higher memory capacity.

And so, we leave you with one final dashboard. If you are interested in finding the ideal AWS EC2 instance for your deployment, this dashboard can help you determine which instances to start with as contenders for your deployment. Selecting your anticipated user amount, preferred pricing model, and budget will leave you with a handful of instances to evaluate your unique server deployment and configuration.

Feel free to use this tool with your own data when you run tests on your own server as you evaluate your move to the cloud! *Please note: Pricing is as of 9/4/18 for US East (N. Virginia) on Windows from the AWS website. Note that it does not include Tableau Server licensing costs.*



The journey: testing, testing...

How did we do this testing? First, we had to learn how to use Tabjolt and how to produce useful tests. This section provides deeper notes from our learning path.

Developing a Tabjolt test plan for EC2

Our journey started simple, but as others have also learned, the more we tested, the more we wanted to test. A variety of changes drove new thinking in our plans. Tableau released a major new server version. AWS released new instance types while we executed our tests. We wanted to test a) more instances b) the most recent version of tableau server and c) have more robust results, so we had to rethink the design of our first experiment. These factors came together to influence our testing architecture, which is similar to how we test Tableau servers here at Tableau.

We initially tested several tableau servers in sequence by

1. Spinning up the tableau server machine with the base image
2. RDPing into the tajolt server machine (we used an m4.2xlarge) and editing the batch script with the correct environment being tested, then editing the server config with the Tableau Server IP address before running the batch script.
3. Waiting for the test to run (each test run was about 15 minutes, so a total of about an hour and 15 minutes). We tested threads (c) of 4, 6, 8, 10, 12 for each of the instances.
4. RDPing in, and repeating for each individual tableau server instance.

This had several drawbacks starting with the notable time commitment involved. The more servers we wanted to test, the more time commitment the process demanded. We decided to move the Tabjolt postgres database into RDS (AWS's Relational Database Service), deploy five Tabjolt servers to run concurrent testing, and run longer tests.

More on Tabjolt

Here is a summary of our notes about Tabjolt:

- Tabjolt collects built-in metrics from Windows PerfMon, Java Health, Java Mbean (JMX) counters. JMX Counters are not needed in most scenarios; they can collect more granular details on services running on each worker. We didn't use them for our tests.
- Tabjolt should be used as an A-B tester. To be able to accurately collect, analyze, and interpret your results, you should start with a baseline or a control group, and then change one variable at a time. Generally, in the field, your "A" group is modeled after your existing server in terms of workbooks, users, server concurrency, think time, and test types.
- VIZQL Cores and Concurrency: When a user sees a spinner as a viz loads, you can safely assume that a single core of your server is running at 100%. This will start with VizPortal,

then VizQL, then data engine, then go back to VizQL to render the dashboard. We can make the approximation that a concurrent user is a person watching a spinner. So, if eight people watching spinners will consume 100% of eight VIZQL cores (with a little additional overhead for the repository and logging), any additional users will have to wait for someone to finish (queueing). When a user is waiting in a queue their dashboard will take longer than expected to load. **But eight people on an 8-core server sounds a little ridiculous, right?** What we actually see in the field is eight people waiting for spinners, with ~24 additional users actually looking at a view and not watching a spinner. So 32 total active users before there is any decrease in load times—about a 1:4 or 1:5 ratio for users waiting for something on a server to users looking at a view. To be clear, a thread (the "c" variable in Tabjolt) is not a single user, but this approximation helps us to loosely equate the number of active users observed on a server to the number of open threads we would expect.

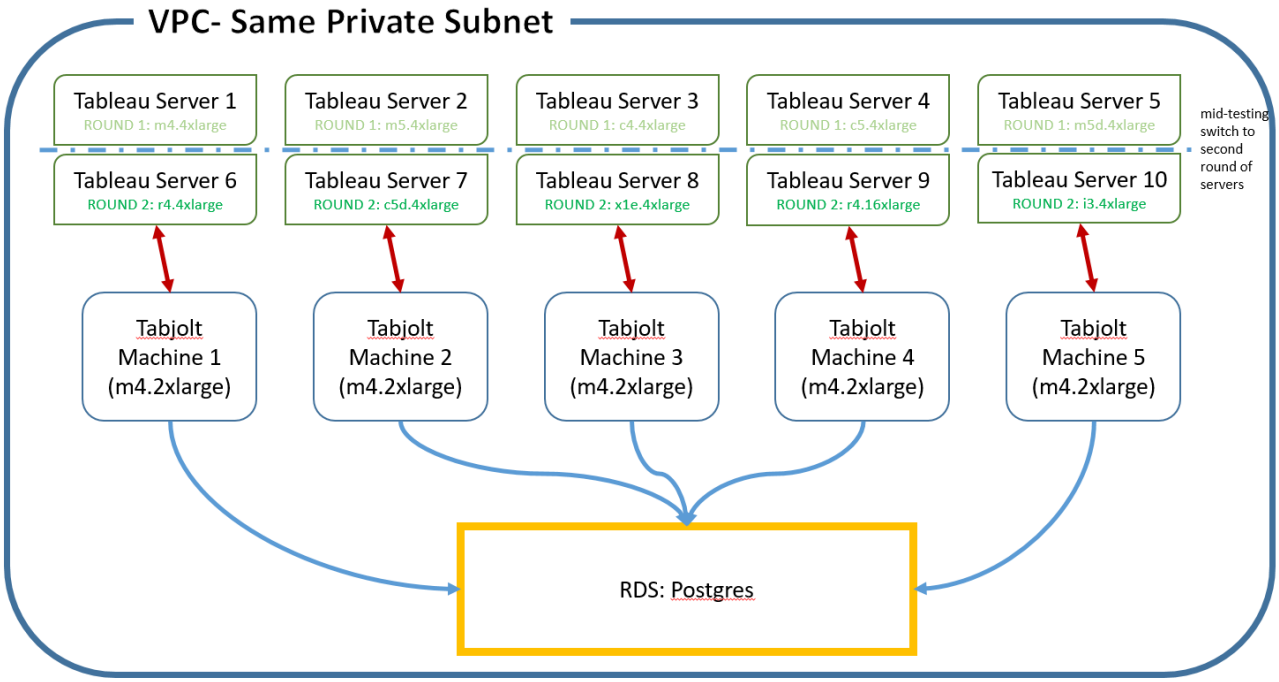
For example:
5 threads = 20 to 25 active users
10 threads = 40 to 50 active users
15 threads = 60 to 75 active users
20 threads = 80 to 100 active users

If we reverse-engineer that math, and think of a server with 1000 users on it, 20 open threads would be pretty rare to see, because that would represent ~100 active users and a concurrency rate of 100/1000, or 10%, which is high. We generally would expect to see 10 threads, for a 5% active users rate (50 active users).

- VizQL processes have built in resource monitoring thresholds. As you run a load test, when resources become scarce, VizQL will restart itself.

Networking simplicity is your friend

We wanted to focus on our test results first so we chose to keep networking as simple as possible. We used a VPC with all of the action happening in a single subnet with a single security group. When setting up the RDS (refer to the Do it yourself! section) we use this VPC when we create the subnet group. We also used private IP addressing between the Tabjolt servers and Tableau servers and between Tabjolt and RDS. This provided a simple, secure private network with minimized latency.



Our resulting architecture had five separate Tabjolt injectors testing five different tableau servers simultaneously, all feeding results (and receiving test ids) from an independent postgres hosted on RDS. This allowed us to run longer tests (more robust results!) and test more instances in one day of testing, with very little effort on our part—we simply RDP'd into the Tabjolt machines after the first round of testing was done, and put in the IPs for the next five tableau servers.

We set up our base Tableau Server (Tableau Server 1) exactly how we wanted it, and then saved an AMI that we applied to all subsequent EC2 instances. This process and others are documented in the Do it yourself! section.

Our base Tableau Server configuration

We set up Tableau Server 2018.1 with 65 total users and a variety of workbooks, some with ginormous extracts, other with smaller extracts, some with embedded data sources, some with data sources published separately, all using .hyper. We did our best to replicate the variety of workbook types one might see in a typical production database.

- Why 65 users? Because with Tabjolt, it's less about the number of total users in your userpool and more about how many open threads (c) you have running. If Tabjolt gets to the end of the userpool, it will start over and will sub in users from the top of the pool again. Within the limited time of a Tabjolt test (30 mins), it's unlikely that a large variety of users will hit the

server—unless we had weights and have longer tests (in the timespan of hours). We're not testing row level security, we're simulating open sessions. It's creating memory pressure, but we expect in a normal production server for people, some sessions will get thrown away due to interactivity while others are starting up, so 65 users seemed like a good base number for both low thread counts and high thread counts.

- One global admin account.
- Users 01-20 are Creators, Users 21-35 Explorers (can publish), Users 36-50 Explorers, Users 51-65 Viewers. Note: The role of these users was essentially meaningless because for the purpose of the Tabjolt test, all these users were assigned "ViewerOrInteractor" (one of the two options for users in your userpool, the other being ServerAdmin). We list the site role here simply to allow for reproducibility.
- 11 total distinct "views" used for the vizpool, all using extracts so that we don't introduce any latency caused by a live connection to a database.

What we learned

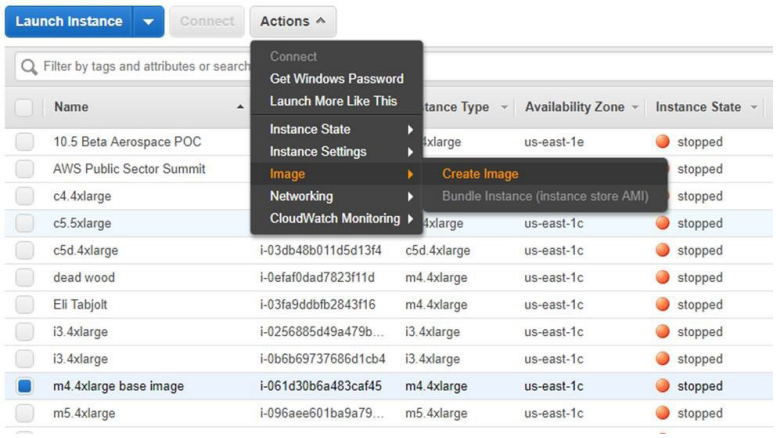
- Lesson learned: The first time we tried this setup, some of the workbooks we published up were stories instead of distinct dashboards. Turns out, Tabjolt can't interact with stories besides (we think) maybe loading it.
- Lesson learned, again: We ended up having to add username() or now() to each of the sheets in here because our first run had impossibly low load times, indicating that we were always using the cache. We wanted to ensure the datasource would be re-queried a good amount of the time to accurately reflect what happens in most production servers.

Do it yourself!

Creating the first server and base AMI

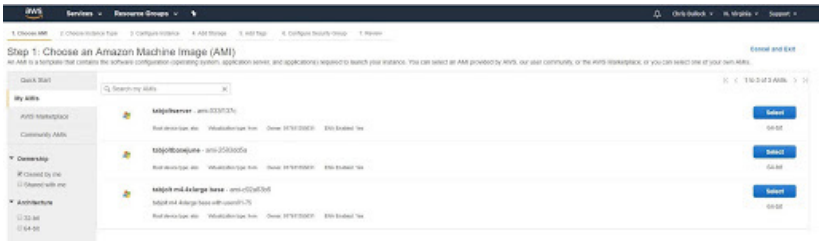
1. In EC2, create your Tableau Server base image.
2. Install Tableau Server, define the base content desired, and define the user accounts you'll want to test with.
3. Document the following : the set of viz URLs for testing, the Tableau server credentials, and Windows credentials; these will save with your AMI and be the same for each instance you create from it (you will tweak the server URLs for the viz set).

4. Once satisfied, stop the instance.
5. Create an image and give it a name you'll remember like Tabjolt-Tableau-server-base.
6. (Optional) Create and save an AMI of the Tabjolt server instance as well if you plan to do parallel testing.



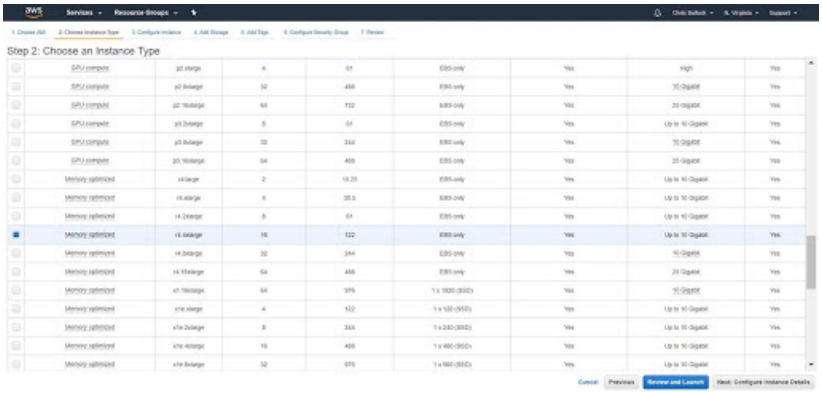
Creating instances from this new image

1. When you Launch Instance from the EC2 menu, you'll notice the default Quick Start options along the left side menu. Click instead on My AMIs.

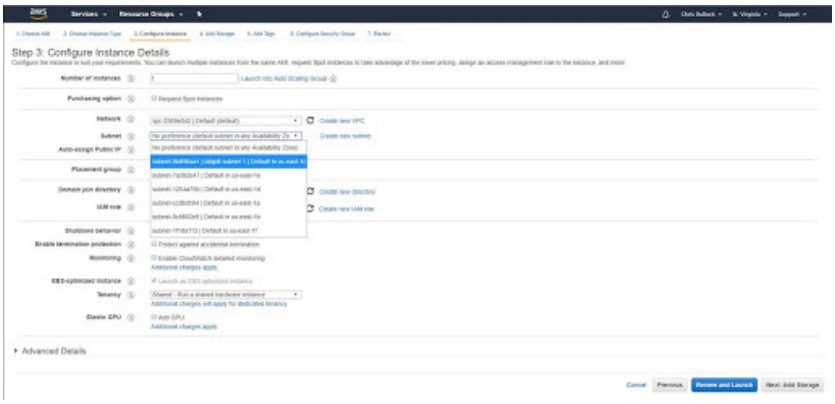


2. You'll find your new image here. It will keep the image content and disk size.
3. You can now create the different instance type replicas from this AMI. I'll use an r4.4xlarge as an example.

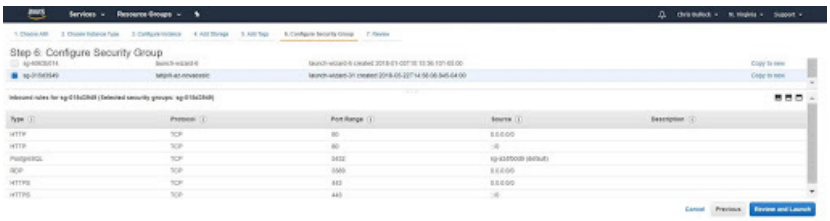
4. Select the instance type, then Next: Configure Instance Details in the lower right.



5. For simplicity, we put everything Tabjolt related in the same VPC and subnet. Click Next: Add Storage.



Note that the drive space size is inherited in these new instances too so no need to change. Click Next: Add Tags and apply any you wish, then Next: Configure Security Group.

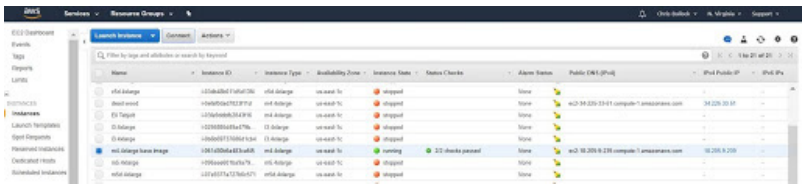


6. We configured our security group to be ready to go with HTTP, HTTPS, RDP, and the PostgreSQL RDS ports open. We kept the source URLS open as this was a test but generally we'd advise limiting these to only what is needed for security best practices.

7. Click Review and Launch.

Connecting to your instances

1. Connect to your Tabjolt and Tableau servers via RDP. Using the m4.4xlarge base as an example, select the running instance and click Connect button along the top. Then click Download Remote Desktop file.



2. All Tableau server instances created from this image will share the same login credentials as will the Tabjolt servers if doing parallel testing.

3. The new Tableau server instances will likely need their Tableau licenses refreshed. RDP into each new running instance and update the server license as needed.

Note: Some of these, like the r4.16 and i3, may take longer to initialize than others. If you RDP in and it's a black screen even though the console says 2/2 checks passed, just give it more time and eventually it should be ready.

Setting up Tabjolt PostgreSQL in RDS

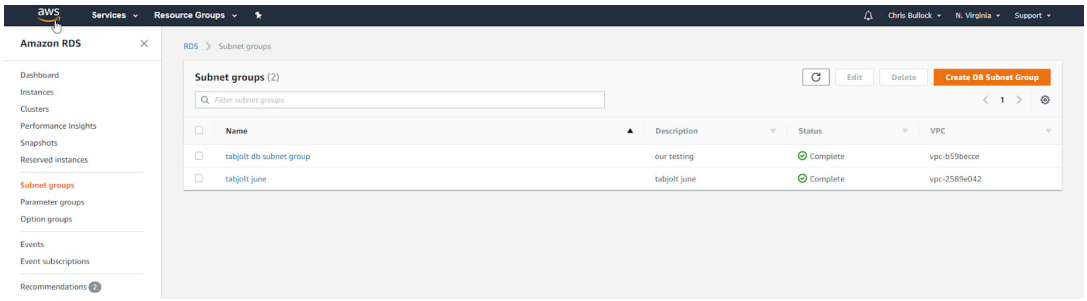
Here is how we set up the Tabjolt PostgreSQL DB in RDS. If you plan to do testing on many instances we recommend this approach. If you only need a few tests we recommend the default Tabjolt configuration with PostgreSQL local to the Tabjolt server and running sequential tests.

Steps to keep things as simple as possible

Have the Networking simplicity is your friend section in view as you review these steps. We recommend reading and understanding the process at a high level end to end before taking action as you may elect to tweak a few things along the way.

1. Decide which region and subnet you want to work in. Create or identify the VPC you want to use. Refer to the guidance from AWS but know that we are simplifying those steps here. Use the same subnet / security group you are using for the Tableau servers.

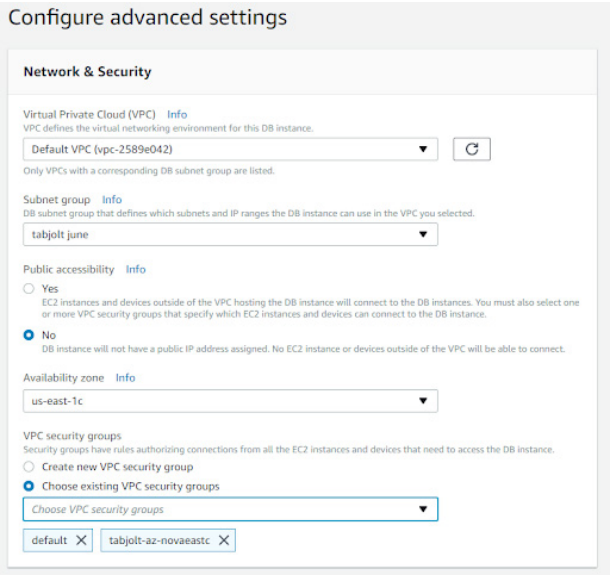
2. Navigate to the Amazon RDS menu in the region you are using. Before creating the RDS, along the left menu, click on Subnet groups. Create the DB Subnet Group for your PostgreSQL RDS by adding the VPC and click Add all the subnets related to this VPC.



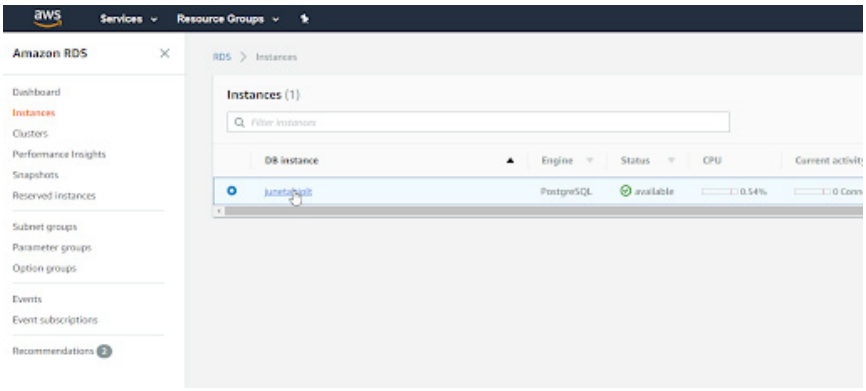
3. Now return to the RDS instance menu and create a PostgreSQL RDS. We followed the guidance from AWS here. Here are the settings we used:

   - Use Case: Dev/Test
   - Engine: PostgreSQL 9.6.6
   - DB Instance Class: db.m4.xlarge
   - Storage: 500 GB
   - Provisioned IOPS: 2000
   - Multi AZ: No
   - Publicly accessible: No
   - Auto Backups: Every seven days

4. In the Configure advanced settings dialog we tie the networking together. Select your target VPC and subnet group. Select the Availability zone where your subnet is (in our case us-east-1c), then Choose existing VPC security groups and select the security group you have used to build your servers in (in our case Tabjolt-az-novaeastc).



Note: Once running and available, you are provided the DNS address (endpoint) of your server but never the IP address. AWS manages that. This is found by clicking in the RDS Instances menu, then selecting the RDS you've created. Scroll down to the Connect section where you'll find the endpoint and port information.



5. Connect the Tabjolt instances to the Postgres RDS by importing the schema from the postgres database that ships with Tabjolt into the new Postgres RDS. Ensure you preserve the same table names and user credentials (postgres/testresults) as given in the Tabjolt user guide to reduce the odds that any wires get accidentally crossed. Update the Tabjolt server image's PerfTestConfig file, pointed to the new postgres server's IP address, so that every subsequent Tabjolt machine will be pointed to it instead of the default install.

How to get started

You can self-deploy Tableau Server on an Amazon EC2 instance that you provision, deploy Tableau Server using the AWS CloudFormation templates in the Tableau Server on AWS Quick Start, or deploy Tableau Server on AWS using an AWS Marketplace Amazon Machine Image (AMI).

Tableau Server	AWS Self-Deployment	AWS Quick Start	AWS Marketplace AMI
Production ready	✓	✓	✗
Upgradable	✓	✓	✗
Install on Linux	✓	✓	✗
Install on Windows	✓	✓	✓
Scale-up	✓	✓	✓
Scale-out (add additional nodes)	✓	✓	✗
Active Directory support	✓	✗	✗
14-day trial license	✓	✓	✓
BYOL license	✓	✓	✓
Subscription license	✓	✓	✓

The most flexible and scalable way to run Tableau on AWS is to spin up an Amazon EC2 instance, download Tableau Server, and install it on that instance. This will give your organization the most granular control and freedom to scale as business demands. For the best performance, try to stick to 16 vCPU instances, like the m4.4xlarge.

The AWS Marketplace offers one-click deployment of Tableau Server instances. The BYOL (Bring-Your-Own-License) instance types come with a free trial of Tableau Server, and if you choose to purchase you can obtain a license from Tableau. This allows for deployment of the Tableau Server AMI in an organization's AWS environment very quickly.

Tableau is also offered in 10, 25, 50, or 100 licenses per hour, and the price of those licenses is rolled directly into the AWS bill. This is the least flexible and scalable way to run Tableau on AWS and is most often used for a proof of concept. If license needs change, you must cancel and order again. In addition, clustering is unavailable with this option, which may not be ideal for especially complex or demanding workloads. However, this is by far the fastest and easiest way to get started running Tableau on AWS. There are also no long-term commitments, and everything is billed through AWS, which is very convenient.

About Tableau

Tableau helps people transform data into actionable insights. Explore with limitless visual analytics. Build dashboards and perform ad hoc analyses in just a few clicks. Share your work with anyone and make an impact on your business. From the individual analyst looking at specific sales performance to the sales executives looking at overall performance in the pipeline and ability to hit targets that support company goals, people everywhere use Tableau to see and understand their data.

Additional resources

Tableau on AWS EC2 Comparison Tool

Tableau Server on AWS Technical Deployment Guide

Tableau Server on AWS Quick Start

Tableau in the AWS Marketplace

Learn more about Tableau and Amazon Web Services

Learn more about Tableau products

Whitepaper: Optimize Tableau and Redshift Performance