On-Premises to Cloud: Ephemeral Scale Testing

Sam Gomena, Andrew Graham, Anthony Spurgeon

Tripwire | sgomena@tripwire.com | agraham@tripwire.com | aspurgeon@tripwire.com

Abstract

The proliferation of cloud technologies has caused software development, testing, tools, and applications to face a new set of challenges. One such challenge, is determining if a non-native cloud application can be converted to work within and take advantage of a cloud. Such a migration brings about the potential for high reward. It also increases scalability demands for development and testing in this new domain.

The viability of applications developed before the rapid expansion of cloud computing must be evaluated before migrating them into the cloud. Cloud environments offer pathways for increased product reach. This expanded reach comes with the expectation that non-native cloud applications will scale according to the potential for greater demand.

This paper demonstrates an approach for testing non-native cloud applications at cloud scale. Specifically, we will cover test design considerations, tooling, lessons learned, and key takeaways from working to overcome these challenges.

Biography

Sam Gomena is a Software Engineer at Tripwire on the performance testing team. He is a senior in Portland State University's Computer Science program.

Andrew Graham is a Software Development Engineer in Test for Tripwire. He is the technical lead for the performance testing team and is actively working on enabling Software as a Service efforts. Andrew graduated from Portland State University in 2014 with a Bachelor of Science in Computer Science.

Anthony Spurgeon is a Software Engineer at Tripwire on the performance testing team. He is a senior in Arizona State University's Software Engineering program.

Copyright Tripwire Inc. 15 June 2019

1 Introduction

Proliferation of cloud technologies has quickly changed expectations and opportunities for software applications. Applications deploy to the cloud while serving users on demand, all at scale. Non-native cloud applications were designed either before the advent of the cloud or without initial consideration for deployment. This rapidly evolving space is providing a second wind for non-native cloud applications.

Non-native cloud applications are at a design disadvantage, as they were not engineered to be served at the scale that a cloud has the potential to provide. To add to the complexity, users can connect and disconnect from cloud applications rapidly. Applications must be prepared for the following circumstances:

1. High load
2. High connection flux

"High load" is a subjective term depending on the application and the conditions around its use. In the scope of this paper, the definition of high load will be arrived at in two parts. First, it is the peak quantity of connections to the application under test over some period. Second, that peak quantity will be applied to the application under test for an extended period. In combination, these two parts create the conditions that we will accept as high load.

High connection flux, in the scope of this paper, describes the rate at which new connections are established and existing ones expire. It is important to note that we consider both connections and disconnects. Each operation comes with some computational cost for the application under test. These operations will be referred to as onboarding and offboarding, respectively.

These requirements, high load and high connection flux, lead to the development of the ephemeral scale testing outlined in this paper. Ephemeral scale testing intends to create the conditions necessary to simulate both criteria in an efficient, controlled, and reproducible manner. The goal of the ephemeral scale test is to provide a development team with confidence of an application's ability to perform at scale in a cloud environment.

Ephemeral scale testing will be described in three sections: Test design, test tooling, and a case study.

1.1 Problem Statement

How do we design a test that allow us to determine, with confidence, that an application can perform in a cloud-based environment? How do we design and implement a tool capable of simulating high load and high connection flux?

2 Test Design

Ephemeral scale testing, put simply, is a scale test that adds an element of onboarding and offboarding to the application under test. These two capabilities meet the criteria of high load and high connection flux.

2.1 Application Under Test

The application under test, for the purposes of this paper, is comprised of two basic components:

1. Agents installed on assets
2. Console

2.1.1 Components

Agents are responsible for reporting information to the console and they are installed on assets. Assets are systems, physical or virtual, that a user wants to monitor. The number of agents can range from dozens to thousands at any given point in time. Agents can connect and disconnect at will.

For the sake of simplifying the test we will say the following about agents:

1. An agent is installed on some asset
2. An asset needs to:
    a. Be provisioned
    b. Do some work
    c. Destroy itself when its work completes
3. An agent on such an asset will:
    a. Attempt to connect to the console
    b. Do its own work
    c. Attempt to disconnect once its work completes and is reported

The console, which is a component of the application under test, is responsible for aggregating the results of all the agents. There is only a single console for the purpose of our test. Users consume the reports from the console.

The console must:

1. Perform some onboarding procedures when a new agent wants to connect
2. Waits for the agent to do its work
3. Receives the agent's report
4. Perform off-boarding procedures once the agent disconnects

2.1.2 Behavior

If the console is over-saturated, then it will begin to queue onboarding agents. As mentioned earlier, the assets hosting agents have their own work to accomplish. If they finish work before the agent connects to the console and does its own work, then the opportunity is lost. The asset will destroy itself and the agent will never have reported.

To simplify the relationship between an asset and the agent that is installed on it we will simply refer to this pair as the agent. The agents cannot exist without an underlying asset, so we consolidate them into a single entity from here forward. The agents are a component of the console. The application under test in this setup is the console itself.

2.2 Test Tool Requirements

To reiterate, the ephemeral scale test must meet two requirements:

1. **High load**: quantified by the peak maximum number of agents that exist at one time
2. **High connection flux**: quantified by the rate at which new agents are instantiated and existing agents expire.

2.3 Test Tool Design Considerations

For the ephemeral scale test to achieve the above requirements it must be able to:

- Provision a large quantity of test agents (mock or real)
- Control or simulate the lifetime of the agents

Generating a large quantity of agents at a given point in time is only part of the problem for the ephemeral scale test. It must be able to provision many agents at a varied or constant rate over some period. The test can specify the desired peak agents in order to satisfy the high load requirement.

During the test the current quantity of agents will not be constant; however, it may reach a peak quantity multiple times over the course of the test.

High connection flux is met by defining batches of agents with lifetimes. A batch is a term used to define some positive, non-zero quantity of agents. Batches are defined in a list before the execution of the test. The agents in a batch share a common configuration and lifetime. Lifetime is defined as the duration, in minutes, that an agent exists.

To re-iterate, an agent on an asset will:

1. Be provisioned
2. Connect to a console
3. Do work
4. Disconnect
5. Destroy itself.

That cycle represents the lifetime of an agent. For simplicity, the agent's lifetime and the asset's lifetime are treated as synonymous.

When the test begins, it starts a clock that ticks at a user defined interval. The test will attempt to acquire a random unused batch from the list of batches. The batch will perform their lifetime tasks, and then the batch is freed back into the list.

The test checks for an unused batch to deploy every tick. If no batches are available, it simply waits for another tick to check again. Since each batch has its own individual lifetime there will be a varying number of agents transitioning through the various stages of their lifetime cycle.

2.4 Value Proposition

The test will determine at what rate and scale the application under test can handle onboarding and offboarding of agents.

During a successful test run, an application under test will continue to handle onboarding and offboarding procedures. Ideally it would not develop an onboarding queue. No agent connections will be missed. This behavior proves that the application under test can continue to work optimally under conditions that a cloud environment would provide.

A failure will result in the application under test being unable to properly handle new connections for onboarding procedures. The application under test will begin to grow a queue of connection requests. If a queue of connection requests from agents begins to grow, then the risk of those requests becoming invalid increases.

The agents are only meant to live if they have work to do. If the console does not onboard the agents and receive their work, then it is a missed opportunity. Depending on the scenario, it can be considered poor performance or outright failure on the part of the console.

2.5 Limitations and Considerations

The number of peak agents that can be generated across all batches is strictly limited by the hardware on which the test is executing. As a concrete example, we typical use the following physical attributes for asset virtual machines running the agents:

- 1 virtual CPU

- 2 gigabytes memory
- 40 gigabytes disk; thin provisioned

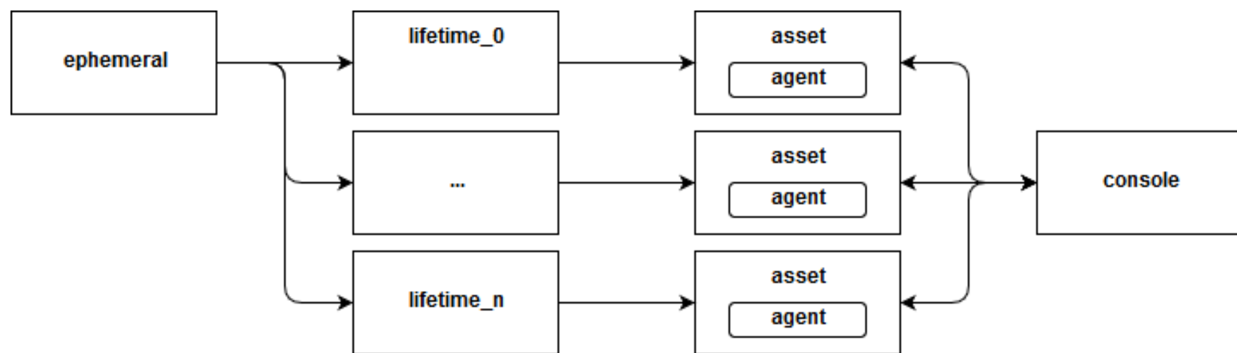
We typically run this test in a vSphere cluster with the following attributes:

- 1 THz compute / 6 TB memory
- 1 PB datastore
- 10 Gb network link

Using the above physical hardware, we have run this experiment with a peak of 10,000 agents.

3 Test Tooling

The ephemeral scale test tool is a Go application. It was designed to satisfy the previously stated requirements of high load and high connection flux. It deploys predefined batches of assets at various rates. Those assets are subsequently provisioned with agents, do work, and are destroyed after a given lifetime.



3.1 Language Consideration

We decided to create the ephemeral scale test in Go. There were several reasons that influenced this decision, as follows:

- **Goroutines:** Goroutines are lightweight alternatives to traditional threads. Goroutines allow the ephemeral scale test to deploy and provision many assets concurrently.
- **Portability:** Our teams operate in a variety of development environments. We require that our ephemeral scale test be able to do so as well.
- **Familiarity:** Our team already had experience working with Go.
- **Existing Tooling:** The ephemeral scale test leverages pre-existing tools and libraries that were already written in Go.

We considered Python and Ruby as alternatives for the ephemeral scale test. Both languages are represented in much of our current tooling. As a team, we are trying to move away from Ruby, and have decided not to develop any new tools in the language. We continue to use Python elsewhere, but we determined that it did not allow for the same level of efficient concurrency as Go.

3.2 Design Implementation

The ephemeral function is the backbone of the test. A ticker is instantiated here. The ticker is a clock that provides a time interval and allows the test to check for and interact with free resources at regular periods.

```
// ephemeral manages the main test loop logic
func ephemeral() {
    var waitgroup WaitGroup
    var ticker = NewTicker()


    // Start the ticker
    for tick := range ticker {
        // Try to acquire an available batch
        var batch = getAvailableBatch()

        if batch != null {
            batchChannel := makeChannel()
            defer close(batchChannel)

            waitgroup.add(1)

            // Manage the lifetime of the asset(s)
            go lifetime(batch, batchChannel, waitgroup)

            // Consume results of the batch channel
            report(batchChannel)
        }
    }
}
```

Each batch is a collection of assets to be deployed, configured, and destroyed. A batch also encompasses some relevant metadata, as shown:

```
// Batch contains basic information for assets
type Batch struct {
    ID      int         // Control index
    Assets  Asset       // Some deployable asset
    Status  int         // Status code
    Created time.Time   // Time created
}
```

The lifetime of each asset is represented below. During its lifetime, the asset is provisioned, an agent is installed and configured, and finally the asset is destroyed when its time is up. Status for the batch's

Copies may not be made or distributed for commercial use

lifetime is returned through a channel to the parent thread. Channels, in Go, allow for bi-directional communication between threads.

```
// lifetime manages the life of assets
func lifetime(data Batch, batchData chan Batch, waitgroup WaitGroup) {
    defer waitgroup.Done()


    provision(data)

    configure(data)

    cleanup(data)

    done <- data
}
```

The ephemeral scale test itself is simple. Every tick, the test checks for any available resources to be deployed. If this check finds any available resources, a new lifetime function is started for the resource.

Within this goroutine, the available batch generates its assets which exist for the duration of their lifetime. Assets are provisioned and agents are configured on those assets. The assets and agents do their work. Once an asset's lifetime expires, the asset is destroyed. Once all assets in a given batch are destroyed, then the batch is freed up. This process continues for the duration of the ephemeral scale test.

Much of the heavy lifting itself (the deployment and configuration of the assets and agents) is handled by existing internal tooling and libraries. These tools were also written in Go. Interaction between the ephemeral scale test and these tools is through an interface that the ephemeral scale test provides.

The result is that, given a proper configuration, once the ephemeral scale test is begun, it will operate for its given duration without further input. During the test, it will deploy batches of assets running various operating systems for given lifetimes, destroy them, and repeat indefinitely.

4 Practical Application

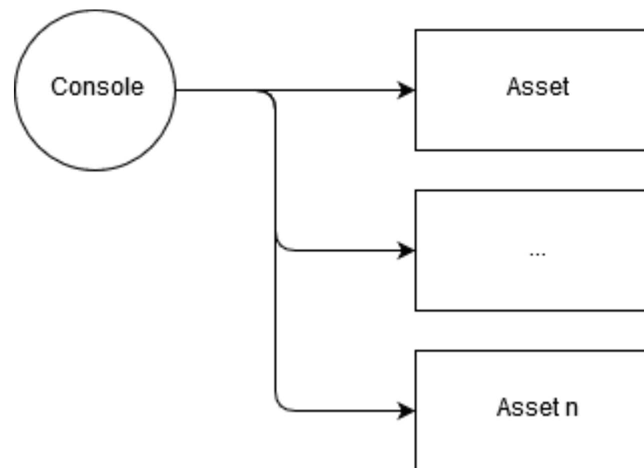
A Tripwire customer who was in the process of transitioning from on-premise to cloud operations tasked Tripwire with demonstrating the viability of monitoring assets in a cloud environment. The customer was transitioning from on-premises operations to cloud operations. They wanted to ensure that their assets could continue to be monitored in the new environment. An extra challenge was added in that assets would no longer be static and there was no guaranteed lifetime – meaning they could come and go so long as they had utility.

Our team had a hypothesis that our application would function appropriately in a cloud environment. We needed to quantify performance and behavior in order to assure the customer that our hypothesis was correct. Enter ephemeral scale testing: We needed to understand if the application under test could onboard and offboard assets fast enough and at scale, and to know if the application could keep up with the flux of connections.

4.1 Application Under Test

The application under test is a combination of two applications that work jointly to monitor assets. As aforementioned, assets are systems, physical or virtual, that a user intends to monitor. For the sake of focusing on the ephemeral scale test itself we'll use the term monitor in the most general sense. Agents are installed on assets and have the responsibility of monitoring their behavior and files based on rules received from the console. The console aggregates data from the agents and presents a consolidated report to the user.

The application under test, while not the primary focus of this paper, will be generally described as follows:



4.2 Test Setup

Test setup initially consisted of installing a console on a Linux machine running in Amazon Web Services. The console was static for the duration of the test. The ephemeral scale test was setup on a remote machine and configured to deploy agents according to the specifications of the customer. An overview of the specifications we were provided is as follows:

- Asset peak volume: 5,000
- Asset lifetime range: 10 minutes to 60 minutes

We then defined limits for the test:

- Test duration of 72 hours
- Asset peak volume of 10,000
- Mixture of Windows and Linux virtual machines
- Batch ranges of 25 to 250 agents
- Batch lifetime ranges between 10 minutes to 60 minutes

4.3 Considerations and Limitations

Once started, the test requires no interaction. We relied on logging built into the product itself.

Initial runs were hindered by high resource consumption on the machine running the ephemeral scale test. This was easily overcome by quadrupling the test machine's memory.

Copies may not be made or distributed for commercial use

The agents were deployed exclusively in an on-premises vSphere cluster. The console was deployed in an Amazon Web Services datacenter on the West coast. There was a latency overhead communicating between on-premises assets and the public cloud-based console. That latency did not have remarkable effect on our test. The customer's stated use case was to begin with a hybrid on-premises and public cloud solution. Their mature state would have both the console and agents running in a public cloud. It was not specified whether they would use separate datacenters available in the public cloud.

The agents used for the test did not do any substantial work. That is, once created, the only program running on the virtual machine was the agent software and any native programs used by the operating system. This likely allowed an agent to provision quickly. In comparison, an agent running alongside a process intensive application may have a harder time allocating resources for monitoring, or sending data to the console.

We used both Windows and Linux virtual machines for agents. The Linux based virtual machines were headless and thus lighter weight than their Windows counterparts.

An important factor in using an on-premises cloud for the assets was the cost prohibitive nature of a public cloud. For instance, a t2.micro instance in Amazon Web Services costs $0.0116/hr at the time of our test. At 72 hours with a peak of 10,000 agents, we predicted that the average number of assets existing at any point in time during the test would be approximately 5,000. Our testing would have conservatively cost $4,000 ($0.0116/hr * 72 hrs * 5,000 mean assets = $4,176) per test run.

Physical attributes of assets were not necessarily specific to the customer use case. It was not considered an important attribute; we cared more about asset lifetime. A machine may have done work 'faster' or 'slower' depending on its physical attributes. Lifetime of the asset was the sole requirement provided by the customer.

4.4 Results

We picked a peak agent volume twice as large as the customer's requirements. We wanted to ensure that we exceeded the customer's expectations. We did not empirically find an upper limit, nor did we calculate a theoretical upper limit. The tests results supported our hypothesis that the application under test could handle the high load and high connection flux of a cloud environment.

The console did not experience any meaningful onboarding queue and all assets were able to communicate with the console within their designated lifetimes.

As a pass-fail test for this particular test scenario, the application under test passed and the tool performed as designed.

5 Closing Remarks

Testing non-native cloud applications at scale is a daunting endeavor. As with any testing, before beginning it is important to have a well-defined understanding of:

- Why you are testing
- A baseline of quantifiable results that will determine success
- Whether the test itself will measure something meaningful

With respect to the ephemeral scale test, we needed to provide a customer with confidence that the application under test could help them transition into a cloud environment.

The baseline of results that determined that success was that no assets would miss connection with the console. The test run that we performed had double the assets that the customer expected, and the

console behaved beyond expectations. During testing it witnessed 100% onboarding and offboarding success with no assets missing a connection during their lifetime.

5.1 Additional Use Cases

The ephemeral scale test provided high confidence that the application under test, a non-native cloud application, could operate in a cloud environment. The test has since been used for other applications as outlined here:

- **Stress testing:** We found that the tool works quite well for fine-tuning stress tests to pinpoint areas of fragility. For instance, we ran an auxiliary test that required a constant influx of data.
- **Functional scale testing:** While somewhat obvious, we were easily able to extend the tool to run an arbitrary test suite against the application under test. This allowed us to provide additional benefit and metrics to a functional test cycle.

There is additional utility to be found in software and environment testing. For example, situations that require the generation of a high volume of ephemeral assets.

Copies may not be made or distributed for commercial use

References

"Documentation." Documentation - The Go Programming Language. Accessed June 12, 2019. https://golang.org/doc/.

“Documentation.” Documentation - AWS Elastic Compute Documentation. Accessed June 14, 2019. https://docs.aws.amazon.com/ec2/?id=docs_gateway

“Documentation.” Documentation - Go by Example: Goroutines. Accessed June 15, 2019. https://gobyexample.com/goroutines