Cloud Computing and Big Data: A Fault Tolerant Elastic Cloud Processing Infrastructure

Jordan Taylor

University of Bristol, Bristol BS8 1UB, UK

Abstract. This paper concerns the workings and testing of a bespoke fault tolerant elastic cloud processing infrastructure. The first motivation of the project was to provide a basis to run parallelizable tasks on streamed data, utilizing the most efficient use of cloud technologies. The second motivation was the system's reliability; both data and nodes should be fault tolerant. In the real world, this translates to reduced cloud processing cost overheads and fewer lost requests.

In this paper's solution, processing requests can be sent to a buffered entrypoint queue, handled by one of multiple scaling worker instances, then output to an endpoint with their result.

The scalability of the infrastructure is two-fold. Worker instances can be created and terminated on demand via creator instances. Furthermore, each worker instance hosts an adaptable Docker pod manager program, written specifically for this project, referred to as a *'Borg Kube'*.

The infrastructure has been built on Amazon Web Services, with EC2 machines acting as worker instances. Borg Kube utilizes Docker to containerize locally running microservices which make up the Kube. Diagnostic data from the infrastructure and subcomponents was gathered in real time for various input stream rates, node failures, and workers. This was used to ascertain the robustness of data processing, fault tolerance, and output rate.

This paper evaluates the success of the infrastructure's two-fold scalability. Increasing the amount of worker instances proportionally increases the rate of data processing. The Borg Kube's effectiveness of scaling based on docker CPU usage utilization has been shown, yielding increased request processing throughput. Throughout the system, data integrity is preserved using callbacks and acknowledgements with queues. The system has been demonstrated to be fault tolerant regarding its structure; continuing to function if creator or worker nodes are taken offline.

It is hoped that this paper describes how EC2 instances may be used for scaling, as well as demonstrating a model of how Kubernettes Pod style managers may be built using Python.

It should be noted that in this paper, existing technologies are referred to in **bold,** whereas new terminology shall be written in *italics.* Terms written in italics can be looked up in the appendix glossary.

1 Architecture

1.1 Solution Overview

Setup.

The system is started by invoking the setup and status monitor. This can be run locally or on an **EC2** instance. This takes a set of AWS credentials to create a session client using Python's **Boto3** [1].

The first act it takes is to declare four FIFO queues using Amazon **SQS (Simple Queue Service) [2].** These queues handle the transfer and buffering of requests from the producer, creator integrity checks, creator instance command codes, and the output diagnostic queue.

Secondly, the setup script declares two creator EC2 instances. These instances are responsible for creating and destroying workers. The creators work in tandem to poll the *creator command queue,* containing commands sent from the status monitor. Commands are relatively simple, specifying positive or negative n, where n is the number of workers to add or remove from the worker pool. Workers are instantiated with a user data script as a parameter; used to pass setup instructions to provision the instance with environment variables, AWS credentials, **Docker [3],** and *Borg Kube.*

Additionally, creators poll the integrity queue. This contains pings sent periodically from the status monitor. If a ping times out, a creator is presumed dead, and a new creator instance is spun up. This redundancy allows for the worker pool to scale even on a creator node failure. Since this is a non resource-intensive task on our end, t2 micros were selected as the platform.

Next, a producer instance is created directly from setup. The producer is used to generate input requests to the system at specified rates to facilitate testing simulations.

The last step of the setup process is to instantiate an **S3 bucket [4]** to hold the *Borg Kube* program files. S3 was chosen as a platform because of its boto3 interface, hosting reliability, static URL, and facilitation of public object download requests. Workers download *Borg Kube* files upon startup, as defined in a setup bash script.

Runtime

Once setup is complete, the application enters runtime. The producer(s) will begin to generate messages and send them to the *input queue.* The status monitor runs in a loop, polling the mean change in the number of requests stored in the input queue over time since the last poll at 45 second intervals. An increase/decrease sends the appropriate signal to the *creator command queue* to create or destroy worker instances. If the queue length remains at approximately zero, a worker will be destroyed. As a result, number of worker instances scales with demand. Since there are no workers to process the queue immediately after setup, a worker is immediately created. Worker instances now utilize *Borg Kube* to parallelise tasks between Docker containers in a local environment. The status monitor reads the length of the input queue, scaling the worker pool as needed.

1.2 Borg Kube

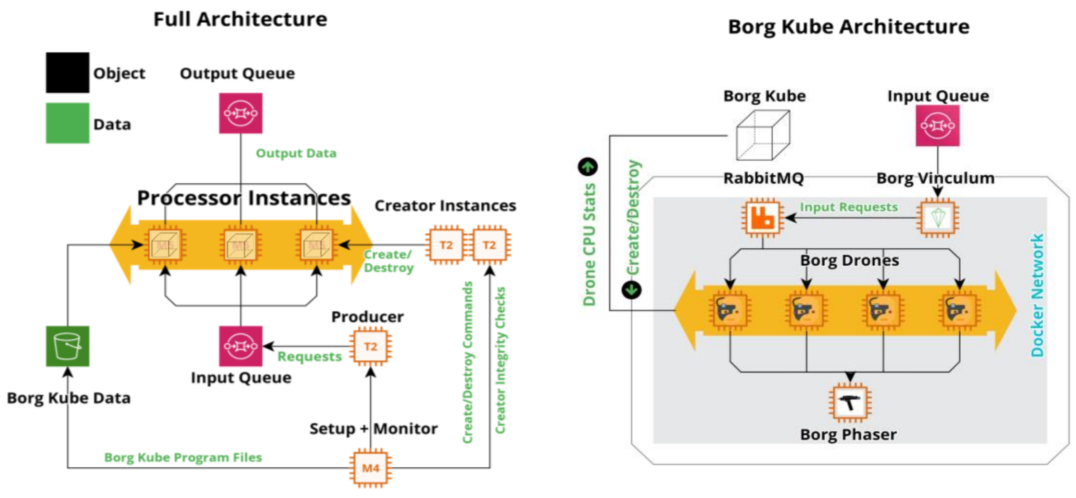


Fig. 1. Architecture diagram showing an overview of the cloud infrastructure layout and a Borg Kube. Arrows indicate commands/transfer of data. Objects are instantiated at runtime.

Borg Kube is a program written for this project. Its architecture is based on scalable **Pods [5]** – a sub application of **Kubernettes –** and its smallest deployable unit of computing . *Borg Kube* scales the number of drones as a function of virtual CPU usage. When the mean drone CPU usage hits a threshold, a new drone is spawned. If the mean usage drops below a certain threshold, a drone is killed. Positive scaling may continue until the host machine's maximum CPU utilization is reached. Each drone has their virtual CPU polled multiple times over a duration of 10-45 seconds, calculating the mean to mitigate extreme values and excessive wait time between tasks. This is similar to Kubernetes, which attempts to poll the CPU every 15 seconds [6] [7].

The *Borg Vinculum* acts as the master node. It is responsible for polling the SQS input request queue and distributing the tasks to available drones. It defines a work queue in **RabbitMQ [7],** running over a local Docker bridge network. Each drone requests tasks from RabbitMQ. The *Vinculum* can dedicate all of its time and resources to polling the *input queue.* This allows the *Vinculum* to request messages in batches and store requests in a local buffer – requiring only a single acknowledgement. The local buffer is used to populate the earlier defined RabbitMQ work queue. This is advantageous in comparison to each drone polling the input queue, since the local work queue has a far lower latency than the cloud based SQS queue, thus minimizing the downtime each drone has between tasks.

A *Borg Drone* is a Docker container with an exposed entry point to the local RabbitMQ work queue. It is able to run any required program within its Dockerfile specification. Borg Drones are most effective when their programs are written using coroutines, especially when they must utilize web requests, to mitigate excess waiting.

Borg Phaser is the final container in the Kube. It defines a *work result* queue in RabbitMQ which is used by drones send their processed output data. The Phaser collates drone output data and enables a data delivery system to another location on the internet with the correct AWS credentials. Phaser utilized batch sending to reduce the wait time for acknowledgements.

2 Simulation and Test Results

2.1 Simulation Premise

Motivation

In order to appropriately assess the efficacy and work distribution capabilities of the cloud architecture, a simulation should demonstrate that a sample application is able to run on the architecture and achieve these goals. This was carried out by running the architecture with varies diagnostic plugs, sending tagged data to python script via an aggregate queue. Diagnostics were using a specialized variable-read-rate parser and graphs plotted using the **matlibplot** module**.** Time was accurately tracked using changes in timestamps.

Scenario:

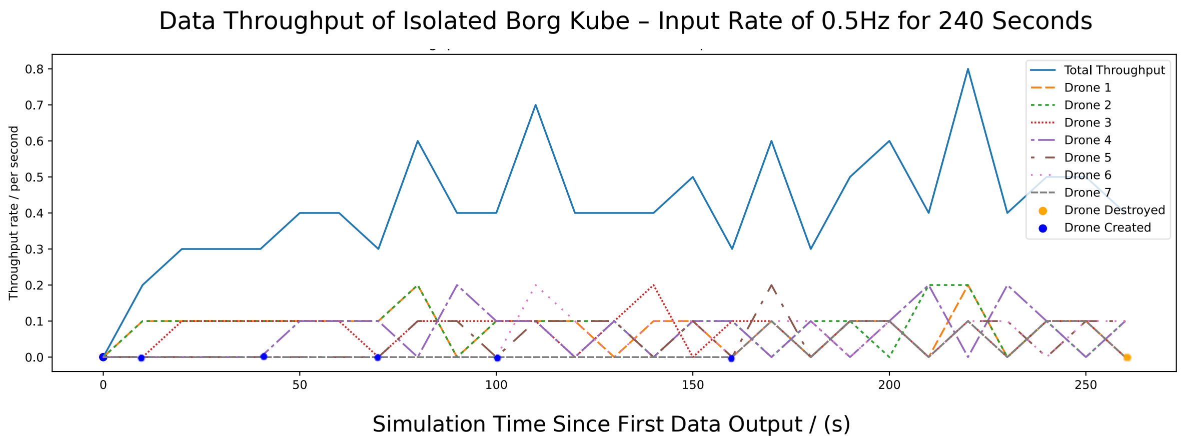
A Borg Cube has become entangled in some sort of dimensional anomaly and transported to another Universe! Ever programmed to carry out their function, the Borg must determine whether the inhabitants of this universe are suitable for assimilation. A Borg probe sends new species designations to the cube's processing hubs. From here, work is distributed between drones. Each drone must ascertain the name of a species from its designation by consulting the cube's **DynamoDB,** as well as retrieving whether the species is suitable for assimilation. Drones must maximize use of their hardware and concurrently perform CPU intensive tasks while awaiting data retrieval.

In reality, this involves each drone hosting a python script that sends get requests to DynamoDB using a coroutine, while simultaneously brute force cracking the four digit species designation as if it were a password to simulate CPU usage.

2.2 Simulation Results

Borg Kube Performance

In addition to being a part of the wider system architect, a Borg Kube instance was tested in isolation. Borg Kube isolation simulations were run on a laptop using a stream of inputs generated directly inside a modified vinculum script.



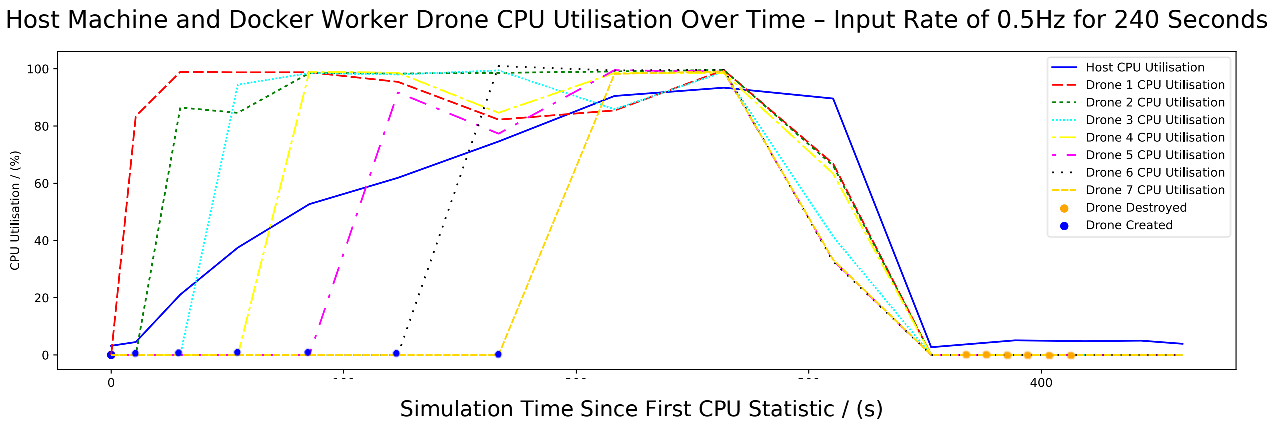


Fig. 2. Graphs of reported throughput and CPU usage of host machine and Docker drones. Diagnostics sent upon each successful sent item and collective CPU poll respectively.

The graphs in figure 2 describe two aspects of a single simulation. Inputs were sent at a constant rate of once per two seconds (0.5Hz) for a fixed duration of 240 seconds, after which transmission ceased.

From figure 2, it can be deduced that that maximum throughput a drone can handle in this scenario is approximately 0.1 requests per second. When a drone is processing, drone throughputs appear to scale linearly. Despite this, some inefficiencies can be seen. Total throughput seems to fluctuate between a medium and high rate when there are more drones. This is most likely due to too many drones accessing the work buffer at once, all waiting for their next job. To improve on this, heuristics could be deployed to estimate the required number of drones for a job, rather than scaling on current drone CPU usage. Alternatively, multiple work local queues could be created to distribute work to a subset of drones each.

Figure 2 shows CPU usage of the host machine and drones over time. This graph exhibits expected behaviour. Drones are spawned until the host CPU usage is close to maximum. The last message was processed at approximately 260 seconds into runtime. The worker is still utilizing a large amount of CPU resources at this point. This indicates that there has been a processing bottleneck, as there are still requests to be processed after 240 seconds. After requests have been processed, the drones use less resources and begin to terminate.

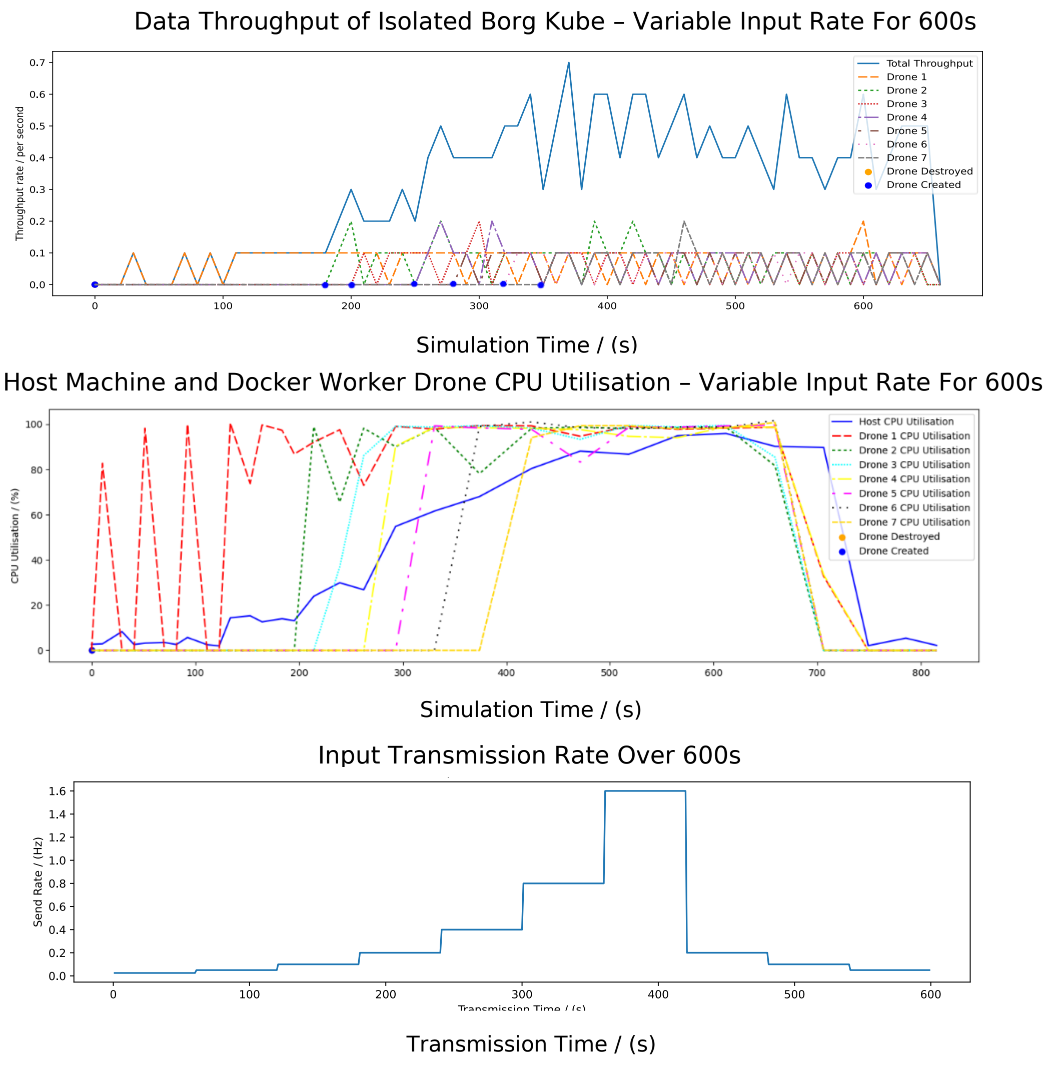


Fig. 3. Graphs of reported throughput and CPU usage of host machine and Docker drones during a variable rate input simulation.

The graphs in Figure 3 demonstrate how a Borg Kube is able to scale adaptively with a changing rate of input. As the input transmission rate doubles over time, until reaching a peak at 360 seconds, the total throughput and host's CPU usage increases. This is until the transmission rate is too large and input requests begin to back up - being stored in the work buffer. The CPU utilization graph starts with several spikes of Drone 1, this is due to it using a large amount of CPU time on infrequent tasks. Consequently, the average CPU utilization is still low, and more drones are not spawned.

Overall Architecture Performance

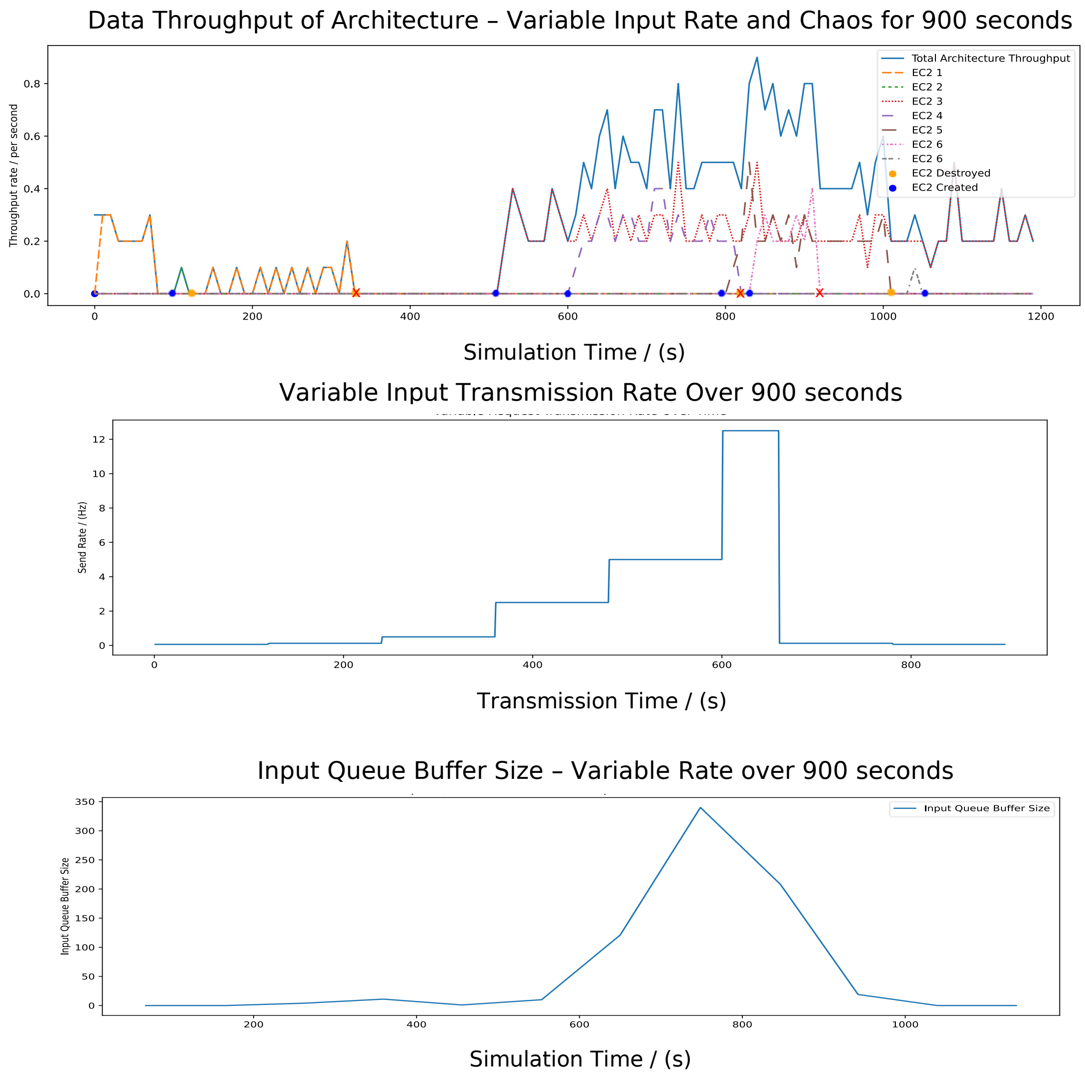


Fig. 4. Graphs showing the throughput of EC2 workers under variable input and chaos.

The simulation for the overall architecture was run for the duration of a variable input rate. This simulation lasted for 1200 seconds, and transmission lasted for 900 seconds. The difference accounts for a buildup in the Input Queue, as can be seen in the third graph. This forced the cluster work off a backlog of tasks.

The simulation was run under chaos conditions. That is, at any time, the simulation has a chance to kill off any EC2 instance. Killed EC2 instance events are denoted as a bright orange marker with an ‘x’, while instantiated EC2s are a darker blue. The first killed EC2 instance leads to an extended period of downtime where no processing occurs. This could be due to the long time between polling the queue average coupled with the low send rate at the time – the Input Queue would look largely empty.

The first downed EC2 instance was taken down by a command from the status monitor. This is because the monitor determined that two EC2 instances were using too many resources for the low input rate.

The apparent overall throughput of the architecture per worker is lower than that of the isolated Borg Kube simulations. This is due to the greater processing power that was available while testing on a local machine compared to the m5 large instance. In a future simulation, the Borg Kube should be tested outside of isolation.

Overall, it can be seen that the architecture is resilient to having nodes taken down at any point as proven in the chaos test. EC2 instances are created when needed to adapt to failure, providing fault tolerance due to fifo queues.

The Borg Kube effectively mimics a local pod cluster; scaling in real time based on CPU use and providing message tolerance through RabbitMQ acknowledgements.

References

1. Boto3 Documentation, https://boto3.amazonaws.com/v1/documentation/api/latest/guide/quickstart.html, last accessed 2021/12/09
2. SQS Amazon AWS Page, https://aws.amazon.com/sqs/ , last accessed 2021/12/08
3. Docker Homepage, https://www.docker.com/, last accessed 2021/12/01
4. S3 Bucket Documentation, https://aws.amazon.com/s3/, last accessed 2021/12/06
5. Kubernetes Pods Documentation, https://kubernetes.io/docs/concepts/workloads/pods/, last accessed 2021/12/09
6. Kubernetes Horizontal Scaling Documentation, https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, last accessed 2021/12/09
7. D. Balla, C. Simon and M. Maliosz, "Adaptive scaling of Kubernetes pods," NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium, 2020, pp. 1-5
8. RabbitMq Homepage, https://www.rabbitmq.com/, last accessed 2021/12/07

Appendix:

Glossary:

Borg Kube:

The Borg Kube is a technology written specifically for this paper. It is analogous to a Kubernetes pod manager has multiple similar characteristics. The Borg Kube is hosted on an EC2 instance and scales a pool of worker *Borg Drones* by CPU usage.

Borg Drone

A Borg Drone is a container managed by a Borg Kube. It can host one or more programs simultaneously, though with diminishing efficiency. Drone lifecycles end when they are terminated by a Borg Kube or exit all their running programs. Drones receive input data from their local *Borg Vinculum* via RabbitMQ. Drones have access to the wider internet via port forwarding. When a task is complete, a Drone may send the result to the *Borg Phaser* via the local RabbitMQ *Work Result Queue.* This significantly reduces wait time as the Drone does not have to handle its own posting to SQS.

Borg Vinculum

The Borg Vinculum behaves similarly to a master node in a cluster. Its job is to dedicate its time and resources to polling the *Input Queue* and pulling data in batches. The Vinculum has a local storage buffer in which it may keep a specified number of work requests for quick dispatch to Drones. Work requests are dispatched to Drones via a local instance of RabbitMQ utilizing a bridge network.

Borg Phaser

Borg Phaser is the output point of a Borg Kube. It defines the *Work Result Queue* used by Drones to quickly send off their results, minimizing downtime until the next task. The Borg Phaser collates the results of the work queue and stores it in a local buffer. It uses batch posting to send the results to an output queue as efficiently as possible.

Work Result Queue

The Work Result Queue is defined by a local instance of Rabbit MQ within a Borg Kube. It is the channel for Drones to quickly send their results to the Borg Phaser.

Creator Command Queue

The Creator Command Queue is used to instantiate or destroy EC2 instances by sending command codes a pool of *Creator Instances*. By using a pool of creators, only a single creator needs to respond to the command to carry out the action. This provides creator redundancy, allowing scaling even if a creator instance is taken down.

Input Queue

The Input Queue is the name for the queue of requests sent by producers in the architecture. A worker may access it using its Borg Kube's Vinculum..

Github Repository Link:

https://github.com/CynicalCheddar/CCBD-Coursework