分散式系統與雲端應用開發實務 期末報告

大吉大利 - 動物養成遊戲

R10725015 張若瑜　　R10725029 莊海因　　R10725011 陳佑甄　　R10725058 王佩晨

一、系統介紹

此專案為一個動物養成遊戲，使用者可於網頁介面上對隨機的動物進行養成，藉由點擊來與動物互動，例如餵食、洗澡、玩樂。根據使用者的互動，動物會蛻變成不同階段的樣貌，並在最終階段得到動物的結局，並且可以下載結局圖片。

二、服務元件

1. frontend：

   前端部分以 HTML/CSS, JavaScript 編寫，呈現動物圖片、點擊效果等畫面。

2. backend：

   後端以 python flask 編寫，主要分為三個 API。

   (1)get_info(animal, history, isEnd)

   藉由動物種類 animal (column family)、歷史紀錄 history (row key)，查詢 BigTable 並回傳對應的動物資料 name、pic，若 isEnd 為 true 表示已進行到結局，則也要回傳結局文字 doc；若歷史紀錄 history 為 “0”，表示是動物初始階段，則回傳 round 表示動物到結局總共會有幾輪。

   (2)download(animal, history)

   藉由動武種類 animal (column family)、歷史紀錄 history (row key)，查詢 BigTable 並回傳結局圖片 endpic 網址供使用者下載。

   (3)get_animals

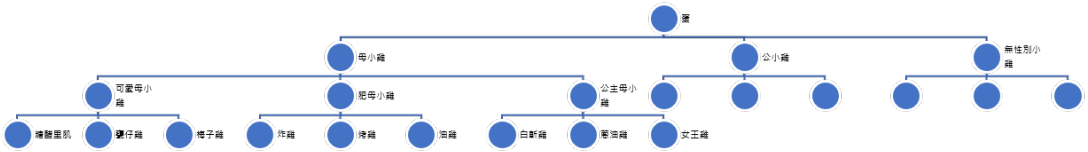
   回傳目前 BigTable 中的所有動物基本資料 genre, name, picture, round。

   在部署時，Docker image 使用 uwsgi-nginx-flask:python3.6，其使用 uWSGI 作為 flask 與 nginx 之間的介接。

三、資料庫 - BigTable

1. 架構：

   在此專案的 Bigtable 中共有兩種動物，分別是雞和企鵝。下圖以雞為例，每個節點為一張圖片，視為一種狀態（state）。每條路徑為一種行為走法，會由最多的行為選項決定，目前定義四種行為（behavior）分別為 origin(O)、play(P)、eat(E) 與 bath(B)。此例中共有四個 level，不包含 level 1 的節點，每個節點狀態為前一個狀態與行為所產生的疊加狀態，故整體走法會如下圖樹的架構呈現。



table：animal

row key：行為歷程

column family：chicken, penguin

column qualifier：name, picture, doc, endpic, round

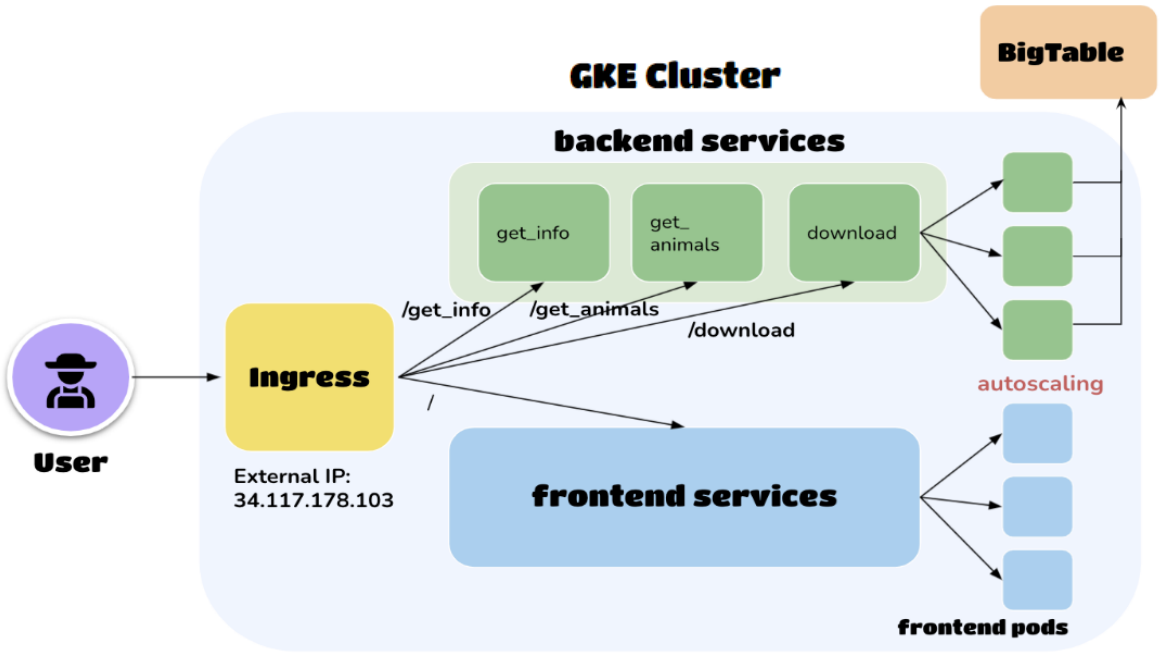
2. 資料：

將資料建立符合上述格式的 csv 後，透過程式自動建立 BigTable。

- Data: Bigtable_data_penguin BigTable_Data_chicken
- Code: read_table.py

```
0
  chicken:name                                  @ 2022/06/03-11:17:00.031000
    "egg"
  chicken:picture                               @ 2022/06/03-11:17:00.901000
    "https://i.imgur.com/3Voe1qY.png"
  chicken:round                                 @ 2022/06/04-10:38:56.404000
    "4"
  penguin:name                                  @ 2022/06/07-12:08:49.850000
    "egg"
  penguin:picture                               @ 2022/06/07-12:08:50.768000
    "https://i.imgur.com/gn8LCx4.png"
  penguin:round                                 @ 2022/06/07-12:10:02.866000
    "3"
```

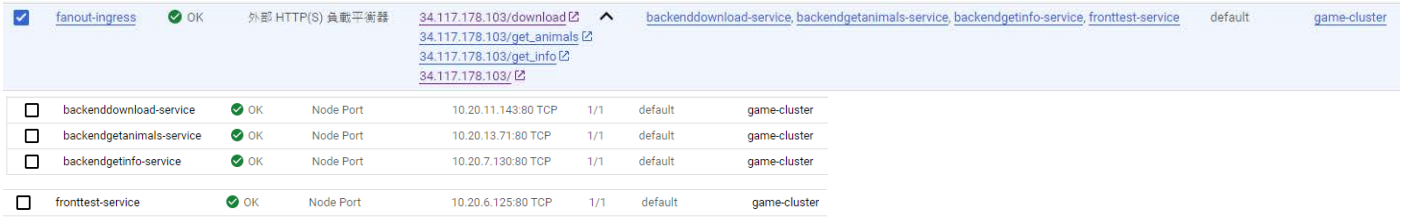
四、系統架構



上圖為系統佈署上 GKE 的架構。

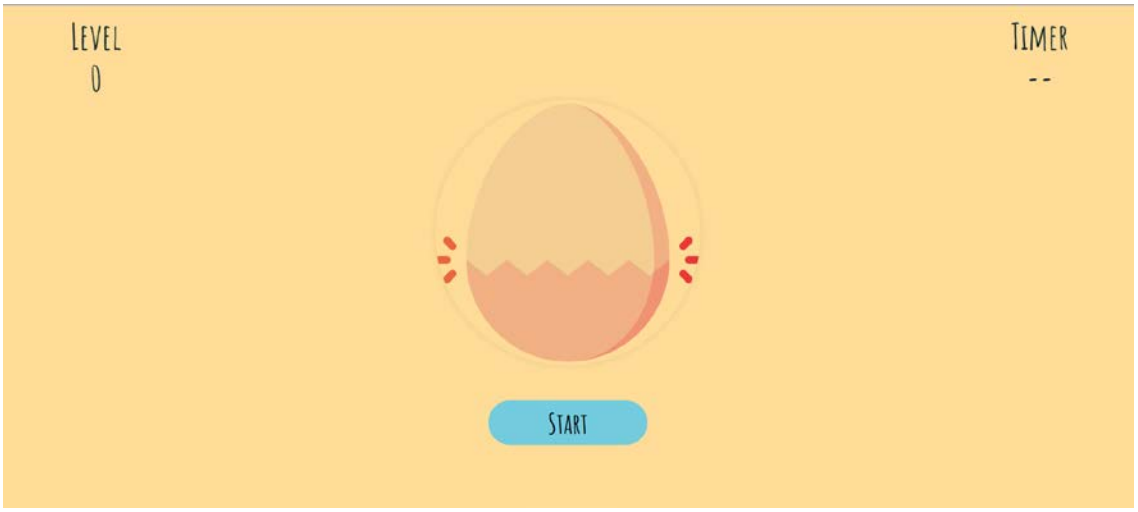
以微服務的角度來看，架構設計盡量要 loosely coupled，讓每個 pod 中的 container 服務越單純越好，因此我們的架構中每個 pod 只含有一個功能的 container，每個 pod 會分別對應服務元件中的四個功能 frontend, get_info, get_animals, download，並 autoscaling deployment 使得 pod 數量最小為 1，最大為 20，當 CPU 使用 > 50% 時會自動新增。後端服務藉由 service account 進行驗證，並使用 Cloud BigTable API 向 BigTable 讀取資料。

在所有 services 之前，使用 Ingress 做負載平衡、routing，以及 expose 外部 IP。網址將根據 IP 之後不同的網址進行導流，讀取前端服務或呼叫後端 API。下圖為在 GCP 上部署好的 ingress，可見其中包含四個分別從不同 deployment expose 出的 services，型態皆為 Node Port 並皆屬於同一 cluster。

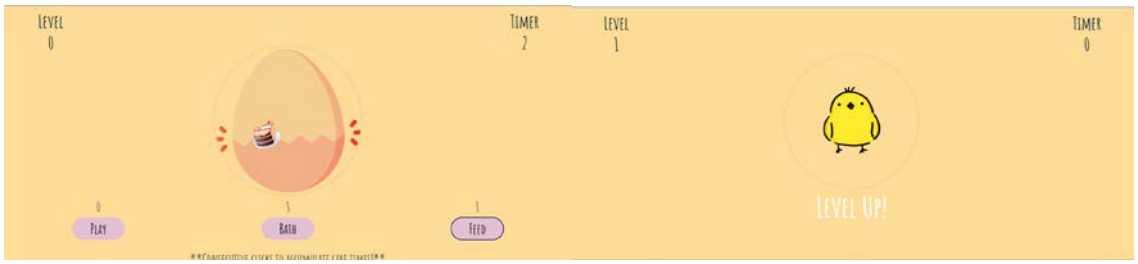


五、系統功能運作

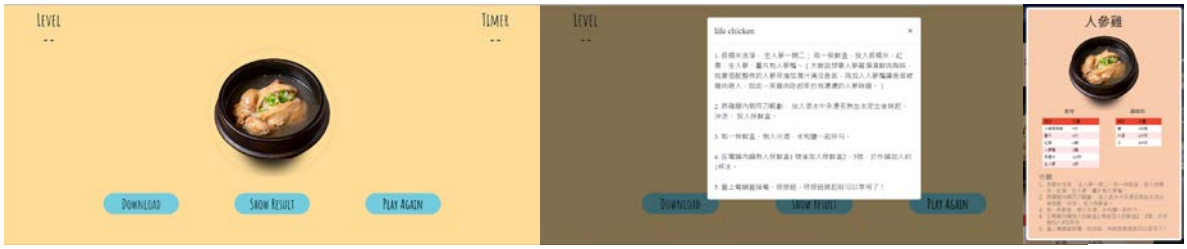
在初始載入遊戲畫面時，前端會呼叫 get_animals API，與後端取得系統現有的所有動物基本資訊，包含每種動物第 0 階段的圖片（以範例雞而言是一顆蛋），而後隨機選擇一種動物開啟遊戲。使用者在過程中可重新整理網頁，以體驗不同動物的成長歷程。



在動物的每一個成長階段，使用者都有 5 秒鐘的時間可以與其互動，互動方式包含玩樂（Play）、洗澡（Bath）與餵食（Feed），透過連續點擊可以累計互動次數。在倒計時結束後，系統會根據使用者最常點擊的互動行為，導往不同的生長歷程，並透過 get_info API 取得下一階段的相關資訊。



當遊戲進入最後階段，系統除了會展示動物成熟後的模樣，也會透過 Show Result 視窗展示相對應的療癒文字，現階段我們預計使雞的結局走向一道美味的佳餚，而企鵝則會獲得一則療養身心的有趣文字。此外，若點選下載按鈕，系統便會透過 download API 取得最終結果的展示圖片，並回傳給使用者以供下載。



六、壓力測試

壓力測試我們在 GCP 上建立一台 VM 並利用 k6 進行模擬攻擊，並利用 wercker/stern 工具與 GCP UI 介面查看該服務的 pods log 與 autoscaler 運作情況。

1. Load Testing：

   此部分的壓力測試為了模擬出平時使用者流量，並想要找出系統在多少使用者的情況下會出現 overload 與系統瓶頸在哪裡。後端測試部分由於我們的系統架構設計，三個後端 services 的架構與 autoscaler 設定相同，故以測試 get_animals 為例。以下為我們進行的實驗。

   - **針對後端服務 get_animals 進行實驗**

Autoscaler 設定：

Cluster node autoscaler：min=3, max=10

Deployment (service) pod autoscaler：min=5, max=20

對 get_animals 發 POST request：

http.post('http://34.117.178.103/get_animals')

由於 k6 的限制，POST 無法超過 50 個 virtual user，故這邊進行 vu=10, 30, 50 三種實驗，並針對 vu=50 連續進行四次實驗來取平均，實驗結果如下表：

get_animal	vu	duration	http_req_duration(avg)	http_req_failed	Iterations
10_1	10	30s	296.4ms	2.55%	235 (7.58 /s)
30_1	30	30s	221.81ms	0.00%	753 (24.15 /s)
50_1	50	30s	826.79ms	41.23%	7458 (29.83 /s)
50_2	50	30s	501.17ms	59.29%	12250 (48.98 /s)
50_3	50	30s	771.83ms	43.23%	7841 (31.36 /s)
50_4	50	30s	506.49ms	58.93%	12044 (48.15 /s)
50_avg	-	-	651.57ms	50.67%	9898 (39.58 /s)

可見在少量 user 時後端服務皆能在少量時間內回覆 request，並有較低的 failure rate，然而當 user 增長到 50 時就開始出現較嚴重的延遲與較高的 failure rate，且相較於前端的數據（下個實驗）後端由於會需要向 Big table 讀取資料故平均 request 時間較長。

- **針對前端服務進行實驗**

Autoscaler 設定：

Cluster node autoscaler：min=3, max=10

Deployment (service) pod autoscaler：min=1, max=20

對 frontend 發 GET request：

http.get('http://34.117.178.103/')

對前端發的 request 為 GET，故以下進行不同 virtual user 情況的測試，並針對 vu=10000 連續進行四次實驗來取平均，實驗結果如下表。

front	vu	duration	http_req_duration (avg)	http_req_failed	Iterations
50_1	50	30s	3.24ms	0.00%	1500 (49.73 /s)
100_1	100	30s	3.27ms	0.00%	3000 (99.53 /s)
500_1	500	30s	9.71ms	0.00%	15000 (490.23 /s)
1000_1	1000	30s	10.27ms	0.00%	30000 (974.70 /s)
10000_1	10000	30s	65.55ms	57.96%	65450 (1704.27 /s)
10000_2	10000	30s	12.01ms	73.33%	106869 (3308.72 /s)
10000_3	10000	30s	18.52ms	76.88%	119310 (3721.80 /s)
10000_4	10000	30s	9.11ms	78.94%	137852 (4308.18 /s)
10000_avg	-	-	26.30ms	71.78%	107370 (3260.74 /s)

由上述結果可見在少量 user 時，前端不管在 request time 或是 failure rate 上的表現都很好，當使用者來到 10000 時才出現 overload 的跡象，雖然我們有設定 autoscaler 但由於 配置新的 pod 與 node 皆需要時間，而我們的測試區間只有 30s，故 failure rate 較高。

- **模擬一天流量**

針對後端服務 get_animals 進行模擬實驗，設定模擬一天當中使用者逐漸增加再逐漸減少的情況，並同步觀察 autoscaler 運作情形。

Autoscaler 設定：

Cluster node autoscaler：min=3, max=10

Deployment (service) pod autoscaler：min=1, max=20

對 get_animals 發 POST request：

http.post('http://34.117.178.103/get_animals')

K6 Script 設定：

Stages:

vus: 20 duration: '30s'
vus: 20 duration: '60s'
vus: 50 duration: '30s'
vus: 50 duration: '20s'
vus: 20 duration: '30s'
vus: 20 duration: '60s'
vus: 0 duration: '20s'

結果：

http_req_duration: avg=913.72ms; p(90)=2.24ms; p(95)=3.35ms

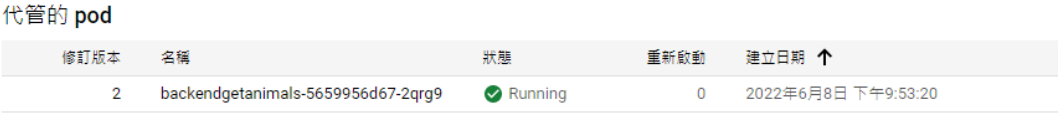
http_req_failed: 35.79%

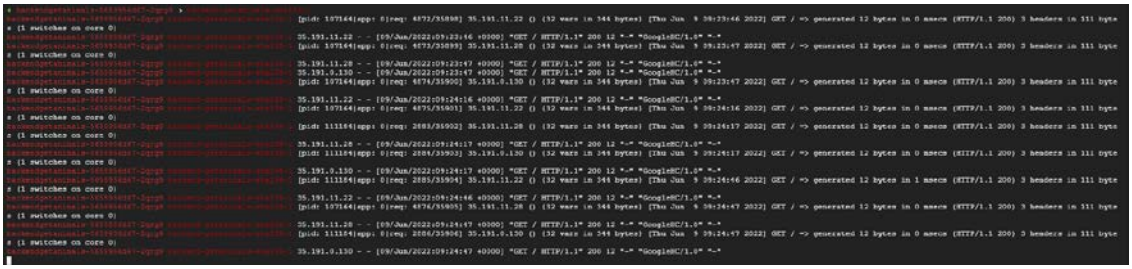
Iterations: 6626 (26.497 /s)

Cluster 中原本只有三個 nodes：

開始執行後會自動增加 node：

Deployment 中原本只有一個 pod，且所有流量都導流至該 pod：

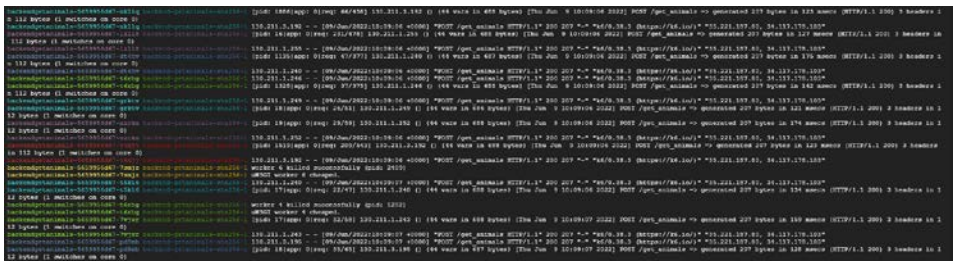




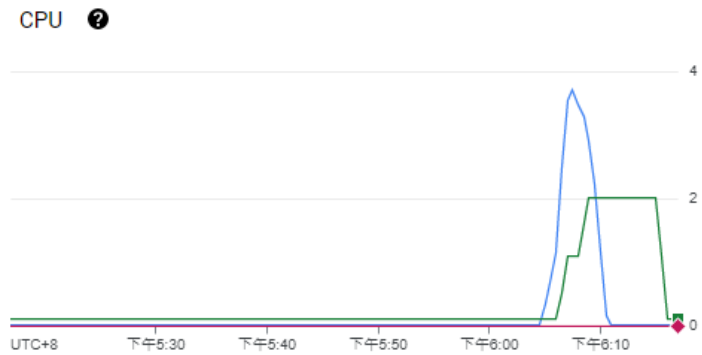
開始執行後會自動增加 pod，並可見流量分布至不同 pods：

代管的 pod

修訂版本	名稱	狀態	重新啟動	建立日期 ↑
2	backendgetanimals-5659956d67-2qrg9	Running	0	2022年6月8日 下午9:53:20
2	backendgetanimals-5659956d67-t4mjj	Running	0	2022年6月9日 下午6:05:53
2	backendgetanimals-5659956d67-hd8hz	Running	0	2022年6月9日 下午6:05:53
2	backendgetanimals-5659956d67-7smjz	Running	0	2022年6月9日 下午6:05:53
2	backendgetanimals-5659956d67-6bngx	Running	0	2022年6月9日 下午6:06:08
2	backendgetanimals-5659956d67-9pvw8	Running	0	2022年6月9日 下午6:06:08
2	backendgetanimals-5659956d67-lzll8	Running	0	2022年6月9日 下午6:06:08
2	backendgetanimals-5659956d67-nkllq	Running	0	2022年6月9日 下午6:06:08
2	backendgetanimals-5659956d67-t6rbg	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-dt4hv	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-245vs	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-t5kl6	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-7vjvr	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-b9f2n	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-8gjlt	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-tv6fb	Running	0	2022年6月9日 下午6:06:23
2	backendgetanimals-5659956d67-pd9mb	Running	0	2022年6月9日 下午6:06:38
2	backendgetanimals-5659956d67-nzckn	Running	0	2022年6月9日 下午6:06:38
2	backendgetanimals-5659956d67-wsmwf	Running	0	2022年6月9日 下午6:06:38
2	backendgetanimals-5659956d67-grktv	Running	0	2022年6月9日 下午6:06:38



以下為 backendgetanimals deployment 的 CPU 用量圖：



在執行完成後隨著 request 量降低，會自動減少 pod 數量至一個：

代管的 pod

修訂版本	名稱	狀態	重新啟動	建立日期 ↑
2	backendgetanimals-5659956d67-2qrg9	Running	0	2022年6月8日 下午9:53:20

此部分影片檔附於繳交資料夾內：壓力測試_後端.mp4。

2. Spike Testing：

此部分的壓力測試為了模擬短時間內突然大量流量的情況，由於上述提及的 k6 的限制，我們只能針對前端服務進行 GET request 的方式來進行實驗。

Autoscaler 設定：

Cluster node autoscaler: min=3, max=10

Deployment (service) pod autoscaler: min=1, max=20

對 frontend 發 GET request：

http.get('http://34.117.178.103/')

K6 Script 設定：

vitrual user：15000

duration：300s

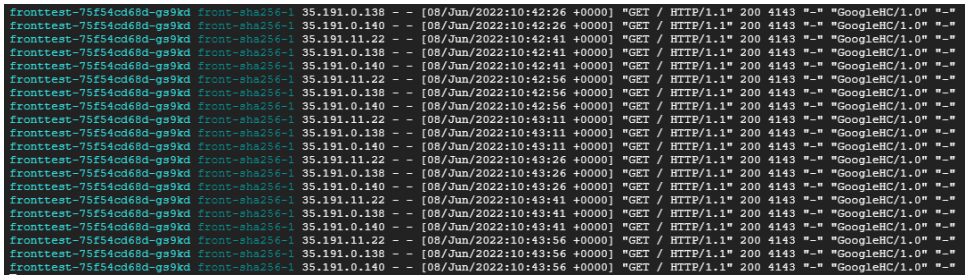
結果：

http_req_duration: avg=6.2ms; p(90)=3.83ms; p(95)=5.16ms

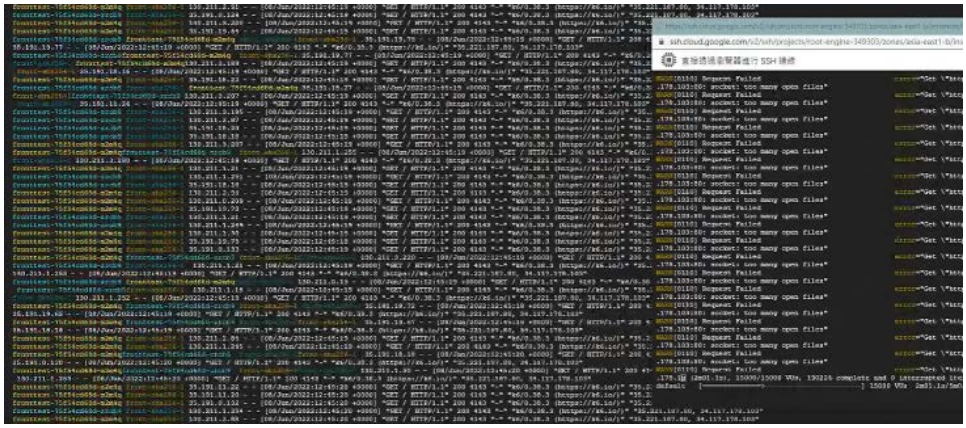
http_req_failed: 39.24%

Iterations: 4964403 (1540.653769 /s)

Deployment 中本來只有一個 pod：



開始執行後會自動增加 pod：



在執行結束後隨著 request 降低 node 減少至一個：

此部分影片檔附於繳交資料夾內：壓力測試_前端.mp4。