Intern Project

Author : Amirkasra Ahmadi
Summer 2024

List

1. Project overview

2. Objectives

- **Develop a Load Testing Framework**

- **Deploy and Evaluate Knative's Autoscaling**

- **Conduct Parameterized Load Testing**

- **Collect and Analyze Performance Data**

- **Visualize and Interpret Results**

Project Overview

This project involves the development of a load testing framework that utilizes a gRPC-based architecture to evaluate the performance and scalability of distributed systems. The framework allows for parameterized testing of system response under varying conditions, providing insights into how different configurations impact performance metrics such as latency and success rate.

Objectives :

1. **Develop a Load Testing Framework**

2. **Deploy and Evaluate Knative's Autoscaling**

3. **Conduct Parameterized Load Testing**

4. **Collect and Analyze Performance Data**

5. **Visualize and Interpret Results**

Develop a Load Testing Framework :

First of all we need to implement a **Load Generator** in Go that can simulate client requests at varying rates to stress-test the Worker function. Here is the first version of Load generator which only used one rps:

```go
package main

import (
    "fmt"
    "net/http"
    "time"
)

func main() {
    url := "http://localhost:8080/worker"
    rps := 10
    interval := time.Second / time.Duration(rps)

    ticker := time.NewTicker(interval)
    defer ticker.Stop()

    for {
        select {
        case <-ticker.C:
            go sendRequest(url)
        }
    }
}

func sendRequest(url string) {
    resp, err := http.Get(url)
    if err != nil {
        fmt.Printf("Failed to send request: %v\n", err)
        return
    }
    defer resp.Body.Close()

    fmt.Printf("Request sent. Response status: %s\n", resp.Status)
}
```

As you can see it is a simple load generator which has a rps that is used to specify the number of request which should be sent to the worker. As you can see in this version it only used the 8080 worker and as the result only one worker can communicate to it.

Implement a **Worker Function** in Go that processes requests by performing CPU-intensive operations for specified durations, thereby mimicking real-world workloads. Here is the first version of the worker :

```go
import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "time"
)

type Metrics struct {
    Ack          string  `json:"ack"`
    E2ELatencyMs float64 `json:"e2e_latency_ms"`
}

func cpuSpin(duration time.Duration) {
    end := time.Now().Add(duration)
    for time.Now().Before(end) {
    }
}

func workerHandler(w http.ResponseWriter, r *http.Request) {
    startTime := time.Now()

    duration := 100 * time.Millisecond
    if d, err := time.ParseDuration(r.URL.Query().Get("duration")); err == nil {
        duration = d
    }

    cpuSpin(duration)

    e2eLatency := time.Since(startTime).Seconds() * 1000

    metrics := Metrics{
        Ack:          "Request processed",
        E2ELatencyMs: e2eLatency,
    }

    w.Header().Set("Content-Type", "application/json")
    if err := json.NewEncoder(w).Encode(metrics); err != nil {
        log.Printf("Failed to encode response: %v", err)
        http.Error(w, "Internal Server Error", http.StatusInternalServerError)
        return
    }
}

func main() {
    http.HandleFunc("/worker", workerHandler)
    port := ":8080"
    fmt.Printf("Worker function is running on port %s\n", port)
    if err := http.ListenAndServe(port, nil); err != nil {
        log.Fatalf("Failed to start server: %v", err)
    }
}
```

So after implementation of both of them I had to implement gRPC for their communication.

Use **gRPC** for efficient communication between the Load Generator and Worker functions, ensuring low-latency interactions and accurate performance measurements. Here is the proto file that was used for communication :

```
1    syntax = "proto3";
2
3    package myproject;
4
5    option go_package = "intern-project/proto;proto";
6
7    // The service definition.
8    service MyService {
9      rpc ProcessRequest (MyRequest) returns (MyResponse);
10   }
11
12   // The request message.
13   message MyRequest {
14     string name = 1;
15     int32 duration_seconds = 2; // Duration of CPU-bound task
16   }
17
18   // The response message.
19   message MyResponse {
20     string message = 1;
21     int64 end_to_end_latency_ms = 2;
22     string worker_id = 3; // Add Worker ID to the response
23   }
24
```

You can see the updated worker and load generator codes which used gRPC in the next page.

```go
3    import (
4        "context"
5        "flag"
6        "google.golang.org/grpc"
7        "google.golang.org/grpc/reflection"
8        "log"
9        "net"
10       "time"
11       pb "worker/proto"   // Adjust to match your module path
12   )
13
14   type server struct {
15       pb.UnimplementedMyServiceServer
16       id string  // Unique identifier for the Worker instance
17   }
18
19   func (s *server) ProcessRequest(ctx context.Context, req *pb.MyRequest) (*pb.MyResponse, error) {
20       startTime := time.Now()
21
22       // Simulate CPU-bound work
23       duration := time.Duration(req.GetDurationSeconds()) * time.Second
24       endTime := startTime.Add(duration)
25       for time.Now().Before(endTime) {
26           // Busy-wait to simulate CPU-bound task
27       }
28
29       latency := time.Since(startTime).Milliseconds()
30
31       response := &pb.MyResponse{
32           Message:                "Processed " + req.GetName(),
33           EndToEndLatencyMs:      latency,
34           WorkerId:               s.id,  // Add the Worker ID to the response
35       }
36       return response, nil
37   }
38
39   func main() {
40       port := flag.String("port", "50052", "The server port")  // Default to 50052 or any available port
41       id := flag.String("id", "worker-1", "Unique identifier for this worker")
42       flag.Parse()
43
44       lis, err := net.Listen("tcp", ":"+*port)
45       if err != nil {
46           log.Fatalf("failed to listen: %v", err)
47       }
48       s := grpc.NewServer()
49       pb.RegisterMyServiceServer(s, &server{id: *id})
50       reflection.Register(s)
51       log.Printf("Starting gRPC server on port %s with ID %s...", *port, *id)
52       if err := s.Serve(lis); err != nil {
53           log.Fatalf("failed to serve: %v", err)
54       }
55   }
```

In the updated worker we used the gRPC communication and get the port which it should use as the input in command we run this, and the default port is 50052.

```go
1   package main
2
3   import (
4       "context"
5       "google.golang.org/grpc"
6       "log"
7       "math/rand"
8       "time"
9       pb "load-generator/proto"  // Adjust to match your module path
10  )
11
12  const (
13      requestsPerSec = 5
14      durationSec    = 5
15  )
16
17  var workerAddresses = []string{
18      "localhost:50052",
19      "localhost:50053",  // Add additional worker addresses as needed
20  }
21
22  func sendRequest(client pb.MyServiceClient) {
23      ctx, cancel := context.WithTimeout(context.Background(), time.Second*30)
24      defer cancel()
25
26      req := &pb.MyRequest{
27          Name:            "LoadTest",
28          DurationSeconds: durationSec,
29      }
30      resp, err := client.ProcessRequest(ctx, req)
31      if err != nil {
32          log.Printf("could not greet: %v", err)
33      } else {
34          log.Printf("Response: %s, Latency: %d ms, Worker ID: %s", resp.GetMessage(), resp.GetEndToEndLatencyMs(), resp.GetWorkerId())
35      }
36  }
37
38  func main() {
39      var clients []pb.MyServiceClient
40      for _, address := range workerAddresses {
41          conn, err := grpc.Dial(address, grpc.WithInsecure())
42          if err != nil {
43              log.Fatalf("did not connect to %s: %v", address, err)
44          }
45          clients = append(clients, pb.NewMyServiceClient(conn))
46      }
47
48      ticker := time.NewTicker(time.Second / time.Duration(requestsPerSec))
49      defer ticker.Stop()
50
51      for {
52          select {
53          case <-ticker.C:
54              client := clients[rand.Intn(len(clients))]  // Randomly select a worker
55              go sendRequest(client)
56          }
57      }
58  }
```

Also added the new port connection which another worker can connect to it too.

Deploy and Evaluate Knative's Autoscaling:

So in this section I needed to create a single node cluster with stock-only deployment in vhive on cloudlab so I follow up this link but it didn't go too well and I had too much struggle to deploy it. So after creating the cluster I tried to deploy my worker and load generator, so I needed to dockerize them. Here you can see the docker file of the Load generator :

```
1    # Build Stage
2    FROM golang:1.22 AS builder
3
4    # Set the Current Working Directory inside the container
5    WORKDIR /app
6
7    # Copy go.mod and go.sum files to the workspace
8    COPY go.mod go.sum ./
9
10   # Download all dependencies. Dependencies will be cached if the go.mod and go.sum files are not changed
11   RUN go mod download
12
13   # Copy the source code into the container
14   COPY . .
15
16   # Build the Go app as a statically linked binary
17   RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o loadgen main.go
18
19   # Final Stage
20   FROM alpine:latest
21
22   # Install certificates for secure communication
23   RUN apk --no-cache add ca-certificates
24
25   # Set the Current Working Directory inside the container
26   WORKDIR /app
27
28   # Copy the binary from the builder stage
29   COPY --from=builder /app/loadgen .
30
31   # Expose any ports the app is expected to run on (if applicable, but not needed for loadgen as it's a client)
32   # EXPOSE 50052
33
34   # Command to run the executable
35   CMD ["./loadgen"]
36
```

I also build the image which you can find in this address.

So for deploying it I also needed to make a yaml file for the load generator which you can see it in the next page. kubectl apply --filename service.yaml

Loadgen-deployment.yaml :

```
kasrahmi@node-000:~/vhive$ cat loadgen-deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: loadgen
spec:
  replicas: 1
  selector:
    matchLabels:
      app: loadgen
  template:
    metadata:
      labels:
        app: loadgen
    spec:
      containers:
        - name: loadgen
          image: kasrahmi/loadgen-intern
          ports:
            - containerPort: 8080
```

Loadgen-service.yaml :

```
kasrahmi@node-000:~/vhive$ cat loadgen-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: loadgen
spec:
  ports:
    - port: 8080
  selector:
    app: loadgen
```

So I deployed the load generator by using kubectl apply –filename file.yaml for both of these files.

And here is the dockerfile for worker :

```
1    # Build Stage
2    FROM golang:1.22 AS builder
3
4    WORKDIR /app
5
6    COPY go.mod go.sum ./
7
8    RUN go mod download
9
10   COPY . .
11
12   # Build a statically linked binary
13   RUN CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -o worker main.go
14
15   # Final Stage
16   FROM alpine:latest
17
18   RUN apk --no-cache add ca-certificates
19
20   WORKDIR /app
21
22   COPY --from=builder /app/worker .
23
24   # Expose the default port
25   EXPOSE 50052
26
27   # Use an environment variable for the port
28   ENV PORT=50052
29
30   CMD ["./worker"]
31
```

And you also can find its image in this link. So I should deploy the worker as a knative function so I used :

```
kn service create worker-service \
  --image=docker.io/kasrahmi/worker-intern:latest \
  --port 50052 \
  --env PORT=50052
```

So after deploying both of them I tried to check the loadgen logs to see are they communicating or not and the logs shows that I got rpc errors for greeting between worker and load generator. So I worked on this bug for 3 days and I couldn't debug it. I made a test-pod and deploy it on the server and tried to communicate with it using curl, wget, nc, and grpcurl but I couldn't connect to this worker function.

Conduct Parameterized Load Testing:

In this part we needed to perform a **parameter sweep** by varying the Requests Per Second (RPS), invocation distribution (uniform vs. Poisson), and CPU-spin duration to test a range of scenarios. So I have updated the load generator to this :

```go
package main

import (
    "context"
    "fmt"
    "google.golang.org/grpc"
    "log"
    "math/rand"
    "os"
    "time"

    pb "load-generator/proto" // Adjust to match your module path
)

var workerAddresses = []string{
    "localhost:50052",
    "localhost:50053",
}

// sendRequest sends a request to a worker and logs the results.
func sendRequest(client pb.MyServiceClient, logFile *os.File, rps int, distribution string, duration int) {
    ctx, cancel := context.WithTimeout(context.Background(), time.Second*30)
    defer cancel()

    req := &pb.MyRequest{
        Name:            "LoadTest",
        DurationSeconds: int32(duration),
    }
    startTime := time.Now()
    resp, err := client.ProcessRequest(ctx, req)
    endTime := time.Now()

    if err != nil {
        log.Printf("could not process: %v", err)
        logFile.WriteString(fmt.Sprintf("Time: %v, RPS: %d, Distribution: %s, Duration: %d ms, Request: %v, Error: %v\n",
            endTime.Format(time.RFC3339),
            rps,
            distribution,
            duration,
            req.GetName(),
            err))
    } else {
        latency := endTime.Sub(startTime).Milliseconds()
        logFile.WriteString(fmt.Sprintf("Time: %v, RPS: %d, Distribution: %s, Duration: %d ms, Request: %v, Response: %v, Latency: %d ms, Worker ID: %s\n",
            endTime.Format(time.RFC3339),
            rps,
            distribution,
            duration,
            req.GetName(),
            resp.GetMessage(),
            latency,
            resp.GetWorkerId()))
    }
}
```

```go
57   func generatePoissonInterval(lambda float64) time.Duration {
58       // Use exponential distribution for Poisson process
59       interval := rand.ExpFloat64() / lambda
60       return time.Duration(interval * float64(time.Second))
61   }
62
63   func main() {
64       var clients []pb.MyServiceClient
65       for _, address := range workerAddresses {
66           conn, err := grpc.Dial(address, grpc.WithInsecure())
67           if err != nil {
68               log.Fatalf("did not connect to %s: %v", address, err)
69           }
70           clients = append(clients, pb.NewMyServiceClient(conn))
71       }
72
73       // Create a log file
74       logFile, err := os.Create("loadgen_log.txt")
75       if err != nil {
76           log.Fatalf("failed to create log file: %v", err)
77       }
78       defer logFile.Close()
79
80       // Test with different RPS and CPU-spin duration settings
81       for rps := 5; rps <= 50; rps += 5 {
82           for duration := 100; duration <= 1000; duration += 100 {
83               for _, distribution := range []string{"Uniform", "Poisson"} {
84                   startTime := time.Now()
85                   logFile.WriteString(fmt.Sprintf("Starting test with %d RPS, %s distribution, %d ms duration\n", rps, distribution, duration))
86                   totalDuration := 10 * time.Second
87                   endTime := startTime.Add(totalDuration)
88
89                   ticker := time.NewTicker(time.Second / time.Duration(rps))
90                   defer ticker.Stop()
91
92                   for time.Now().Before(endTime) {
93                       select {
94                       case <-ticker.C:
95                           client := clients[rand.Intn(len(clients))] // Randomly select a worker
96                           go sendRequest(client, logFile, rps, distribution, duration)
97                       default:
98                           if distribution == "Poisson" {
99                               time.Sleep(generatePoissonInterval(float64(rps)))
100                              client := clients[rand.Intn(len(clients))] // Randomly select a worker
101                              go sendRequest(client, logFile, rps, distribution, duration)
102                          }
103                      }
104                  }
105
106                  log.Printf("Test with %d RPS, %s distribution, %d ms duration completed\n", rps, distribution, duration)
107              }
108          }
109      }
```

So in this updated code it iterate over 2 different distribution, 10 different rps, and 10 different CPU-spin duration. And the logs were written to a txt file so I can analyze using that. For your awareness I had run the load generator only by using go run main.go and workers using go run main.go –port=50052 & go run main.go –port=50053.

Collect and Analyze Performance Data & Visualize and Interpret Results:

Here are the tasks I had done in these two parts :

Collect performance metrics over a 12-minute testing period, excluding the initial 4 minutes to account for system warm-up.

Calculate and analyze metrics such as end-to-end (E2E) latency, success rates, and system overhead (E2E slowdown) to understand system behavior.

Create visualizations to present collected data, highlighting trends and performance differences across configurations.

All of this are done in this python file :

```python
import pandas as pd
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Load log data
log_data = []

# Path to the log file
log_file_path = '/Users/kasrahmi/Desktop/intern-project/new/intern-project/loadgen/loadgen_log.txt'

# Read the log file
with open(log_file_path, 'r') as log_file:
    for line in log_file:
        if "Starting test" in line:
            continue  # Skip the start test lines
        if "Response" in line or "Error" in line:
            parts = line.split(", ")
            timestamp = parts[0].split(": ", 1)[1]
            rps = int(parts[1].split(": ")[1])
            distribution = parts[2].split(": ")[1]
            duration = int(parts[3].split(": ")[1].split(" ")[0])
            request_name = parts[4].split(": ")[1]
            response = "Success" if "Response" in line else "Error"
            latency = int(parts[6].split(": ")[1].split(" ")[0]) if "Response" in line else None
            worker_id = parts[7].split(": ")[1].strip() if "Response" in line else None

            log_data.append([timestamp, rps, distribution, duration, request_name, response, latency, worker_id])

# Create a DataFrame
df = pd.DataFrame(log_data, columns=["Timestamp", "RPS", "Distribution", "Duration", "Request", "Response", "Latency", "WorkerID"])

# Convert Timestamp to datetime
df['Timestamp'] = pd.to_datetime(df['Timestamp'])

# Filter out the warm-up period (first 4 minutes)
start_time = df['Timestamp'].min()
warmup_filtered_df = df[df['Timestamp'] >= start_time + pd.Timedelta(minutes=4)]

# Verify that data exists after filtering
if warmup_filtered_df.empty:
    raise ValueError("No data available after filtering out the warm-up period. Please check the log data.")

# Calculate success rate and average latency per configuration
grouped = warmup_filtered_df.groupby(['RPS', 'Distribution', 'Duration'])
success_rate = grouped['Response'].apply(lambda x: (x == "Success").mean())
average_latency = grouped['Latency'].mean()

# Calculate E2E slowdown (overhead)
if not average_latency.empty:
    min_latency = average_latency.min()
    e2e_slowdown = average_latency / min_latency
else:
    e2e_slowdown = pd.Series(dtype='float64')
```
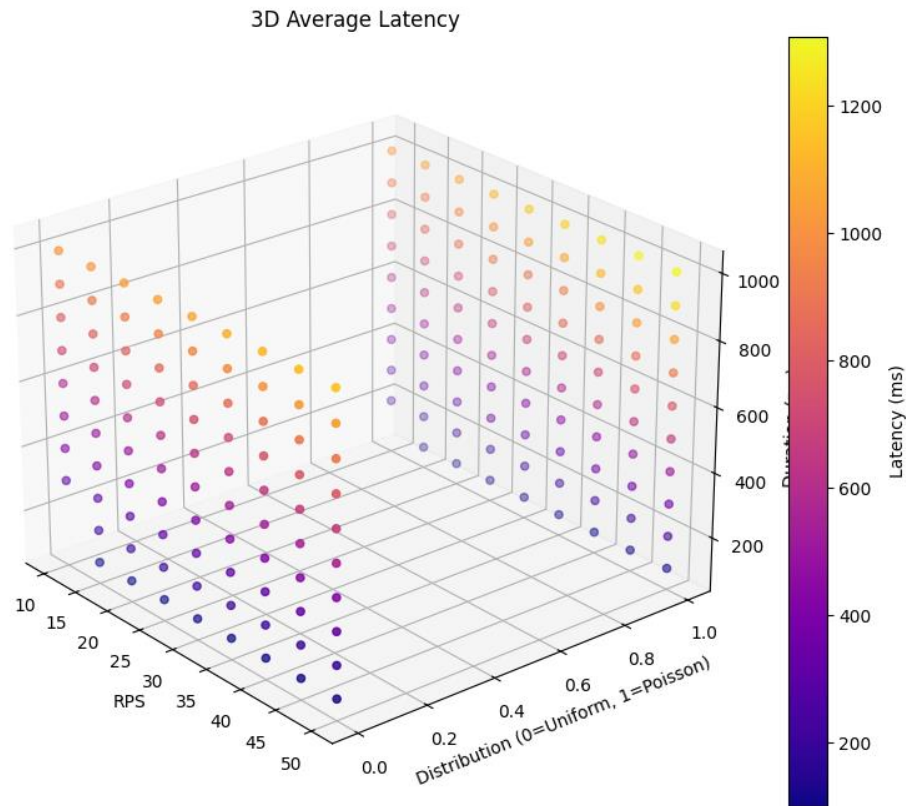
```
54
55    # Map distribution to a numerical value for plotting
56    distribution_map = {'Uniform': 0, 'Poisson': 1}
57
58    # Prepare data for 3D plotting
59    def prepare_plot_data(metric_series):
60        plot_data = []
61        for (rps, dist, dur), value in metric_series.items():
62            plot_data.append((rps, distribution_map[dist], dur, value))
63        return pd.DataFrame(plot_data, columns=['RPS', 'Distribution', 'Duration', 'Metric'])
64
65    # Plot 3D metrics
66    def plot_3d_metric(metric_series, title, zlabel, color):
67        data = prepare_plot_data(metric_series)
68        fig = plt.figure(figsize=(10, 8))
69        ax = fig.add_subplot(111, projection='3d')
70
71        scatter = ax.scatter(data['RPS'], data['Distribution'], data['Duration'], c=data['Metric'], cmap=color, marker='o')
72        ax.set_xlabel('RPS')
73        ax.set_ylabel('Distribution (0=Uniform, 1=Poisson)')
74        ax.set_zlabel('Duration (ms)')
75        ax.set_title(title)
76
77        # Add color bar to indicate metric value
78        plt.colorbar(scatter, ax=ax, label=zlabel)
79
80        plt.show()
81
82    # Plot 3D Success Rate
83    plot_3d_metric(success_rate, '3D Success Rate', 'Success Rate', 'viridis')
84
85    # Plot 3D Average Latency
86    plot_3d_metric(average_latency, '3D Average Latency', 'Latency (ms)', 'plasma')
87
88    # Plot 3D E2E Slowdown
89    plot_3d_metric(e2e_slowdown, '3D E2E Slowdown', 'Slowdown (relative)', 'inferno')
90
91    # Discuss behaviors across different RPS, invocation distribution, and CPU spin duration
92    for rps, dist, dur in grouped.groups:
93        num_requests = grouped.get_group((rps, dist, dur)).shape[0]
94        num_errors = grouped.get_group((rps, dist, dur))['Response'].value_counts().get('Error', 0)
95        error_rate = num_errors / num_requests
96        if error_rate > 0.1:
97            print(f"Configuration with RPS={rps}, Distribution={dist}, Duration={dur} ms is unable to run due to high error rate ({error_rate:.2%}).")
98
```

You can see the plots in the next page.

Average latency :

Analysis of Average Latency

Overview

The analysis of average latency provides insights into the responsiveness of the system under varying load conditions. The results indicate differences in latency behavior when using different invocation distributions (Uniform vs. Poisson) and highlight how these differences evolve with changes in Requests Per Second (RPS) and CPU-spin duration.

Observations

1. **Impact of Invocation Distribution**:

   o The **Poisson distribution** tends to exhibit higher average latency compared to the Uniform distribution. This can be attributed to the inherent variability and burstiness of Poisson-distributed requests, which can lead to periods of higher contention and resource saturation.

2. **Effect of Increasing RPS**:

   o As the **RPS increases**, the average latency generally rises, indicating that the system experiences greater stress under higher load conditions.

   o However, the impact of RPS on latency is more pronounced under the Poisson distribution than the Uniform distribution, likely due to the aforementioned variability causing more frequent congestion at peak loads.
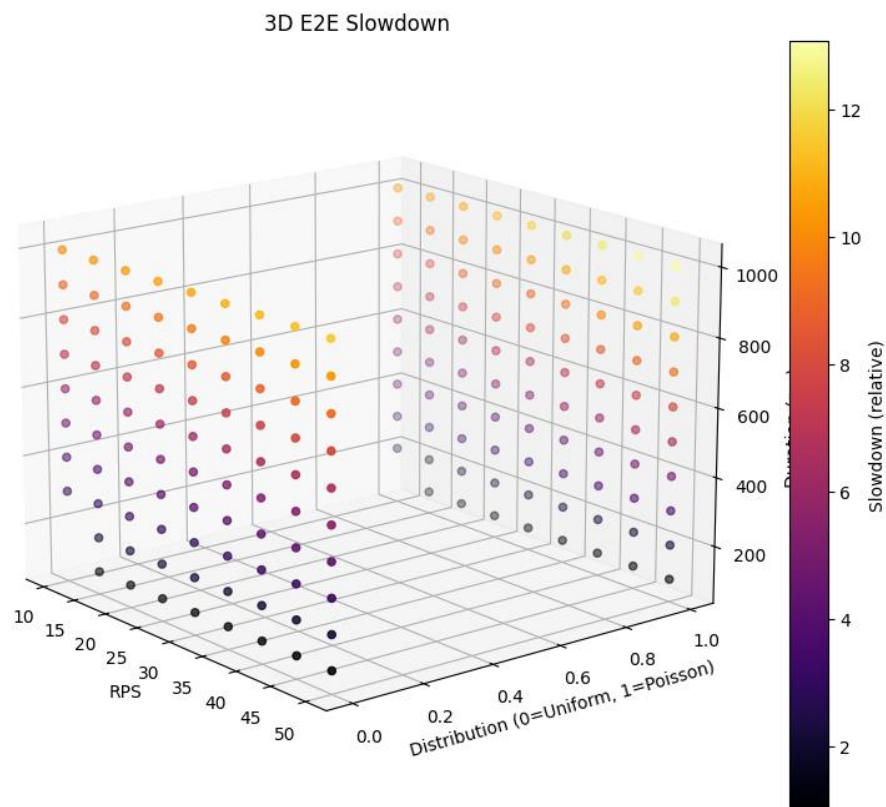
3. **Influence of CPU-spin Duration**:

   o With longer **CPU-spin durations**, the increase in latency is more evident. This is expected, as longer processing times per request naturally contribute to higher overall latency.

   o The effect of CPU-spin duration on latency is especially significant in the Poisson distribution, as the combination of bursty traffic and lengthy processing times exacerbates the delay.

Conclusion

The analysis reveals that the Poisson distribution imposes greater challenges in maintaining low latency compared to the Uniform distribution, particularly under high RPS and prolonged processing times. Understanding these latency trends is crucial for optimizing system performance and ensuring a satisfactory user experience under diverse workload conditions.

E2E Slowdown:

3D E2E Slowdown

Analysis of E2E Slowdown

Overview

The end-to-end (E2E) slowdown metric measures the overall performance degradation in the system under varying conditions. It provides a comprehensive view of how different parameters affect the efficiency and responsiveness of the system. The observations reveal significant differences in slowdown behavior between invocation distributions and highlight how these differences are accentuated with changes in Requests Per Second (RPS) and CPU-spin duration.

Observations

1. **Impact of Invocation Distribution**:

   o The **Poisson distribution** shows a markedly higher E2E slowdown compared to the Uniform distribution. This difference is more pronounced than in the average latency analysis, suggesting that the bursty nature of Poisson traffic leads to greater resource contention and queuing delays throughout the system.

2. **Effect of Increasing RPS**:

   o With an increase in **RPS**, the E2E slowdown increases significantly, indicating the system's inability to handle high load efficiently.

   o The impact of RPS on slowdown is particularly strong in the Poisson distribution, where the unpredictable arrival patterns of requests result in greater performance degradation as the system struggles to manage peak loads.

3. **Influence of CPU-spin Duration**:

- As the **CPU-spin duration** increases, the E2E slowdown becomes more pronounced. Longer processing times per request exacerbate the system's performance challenges, leading to greater overall slowdown.

- The combination of longer CPU-spin durations and Poisson-distributed requests creates a scenario where the E2E slowdown is especially significant, as the system contends with both prolonged processing and erratic request bursts.

Conclusion

The analysis of E2E slowdown underscores the challenges posed by Poisson-distributed requests, especially when combined with high RPS and extended processing times. These factors lead to substantial performance degradation, highlighting the need for effective resource management and optimization strategies to maintain system efficiency under varying load conditions.