Consensys Assignment

Francesco Piva 10/10/2024

Question 1: Describe a complex Kubernetes networking issue you have encountered in a production environment. How did you diagnose the problem, and what steps did you take to resolve it.

Setup description

In all the 20+ AWS EKS clusters we hosted a set of what we have called "Ground Services". The "Ground Services are the **required applications** for the developer teams to be able to deploy their applications and make sure that everything worked. A non exhaustive list of the services and apps running

- CoreDNS
- External DNS
- AWS Load Balancer
- Kyverno
- -> Gloo Edge Proxy - API Gateway
- . . .

The issue discussed here was found with the **API Gateway Gloo Edge Proxy**.

Problem encountered

This crucial service needed to have an 99.99% uptime SLA. We have notice that during peak loads, rescheduling or upgrade of the service there were connection drops and increased latency.

We wanted to test the effect of rolling restarts or deployments of Gloo Proxy pods. As this happened during upgrades or configuration changes, we wanted it to go as smoothly as possible. This helped us in tuning and aligning the different timeouts.

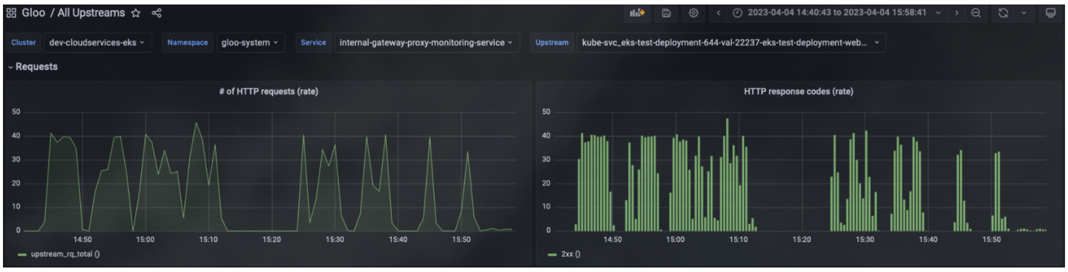
Details on resilience testing during rolling deployments A load test can be performed using `hey` like this:

Listing 1: Fail health check

```
hey -disable-keepalive \
  -c 4 -q 10 --cpus 1 -z 30s -m GET -t 3 \
  -T 'application/json' \
  https://eks-test-deployment-cs.app.domain.com/status
```

This will run a load test with 4 clients, each requesting 10 calls per second, during 30s. The timeout is aggressively set to 1s to detect as much as possible. The content type header may have to be set to application/json to avoid 415 return codes.

In Grafana, the Gloo / All Upstreams dashboards will show the number of requests per second and the return code. `hey` will also output a summary of the result codes at the end.


Gloo / All Upstreams

2023-04-04 14:40:43 to 2023-04-04 15:58:41

Cluster: dev-cloudservices-eks
Namespace: gloo-system
Service: internal-gateway-proxy-monitoring-service
Upstream: kube-svc_eks-test-deployment-644-val-22237-eks-test-deployment-web...

Increasing the request timeout (above is set to 1s), the likelihood of getting client timeouts will reduce. A value of 30s will reflect more real world scenarios.

Diagnosis of the problem

Envoy admin console on internal-gateway-proxy - health checks (fail/ok)

Listing 2: Fail health check

```
wget --post-data "" -O /dev/null 127.0.0.1:19000/healthcheck/fail ; date
```

Listing 3: Restore health check

```
wget --post-data "" -O /dev/null 127.0.0.1:19000/healthcheck/ok ; date
```

Run `curl` for generating requests

Listing 4: Testing the AWS Network Load Balancer

```
while true; do echo -n "$(date) "; curl -qs -k -m 3 -I -X GET https://eks-test-deployment.app.domain.com/status | egrep 'HTTP|x-envoy' ; s
```

Listing 5: Test via Kubernetes service

```
while true; do
    echo -n "$(date) ";
    curl -qs -k -m 3 -I -X GET -H 'Host: eks-test-deployment.app.domain.com' https://internal-gateway-proxy.gloo-system.svc.cluster.local/
    sleep 1 ;
done
```

With the curl commands we can see that the we have failing requests.

We can then check the status of Kubernetes service and NLB targets:

Listing 6: Check healthcheck on healthCheckNodePort

```
watch -n1 "echo 172.16.40.92; curl -qs -X GET -I http://172.16.40.92:31149/healthcheck | grep HTTP;
           echo 172.16.40.11; curl -qs -X GET -I http://172.16.40.11:31149/healthcheck | grep HTTP;
           echo 172.16.41.10; curl -qs -X GET -I http://172.16.41.10:31149/healthcheck | grep HTTP"
```

Listing 7: Check healthcheck on healthCheckNodePort

```

watch -n1 -d -- "aws elbv2 describe-target-health --target-group-arn arn:aws:elasticloadbalancing:eu-west-1:123456789012:targetgroup/k8s-g
```

Solution to solve it

Gloo proxies configuration

The NLB configuration for the Gloo proxies is based on configuring a NLB

We found out that `cross_zone` enabled causes issues during rolling deployment of **Envoy.**

- Gloo Intermittent Connection Failure
- Gloo Zero Downtime Gateway Rollout also disables it with when cross-zone load-balancer is enabled, client requests time out intermittently for 45s whenever the readiness check fails

Listing 8: AWS Load Balancer Controller parameters

```
service.beta.kubernetes.io/aws-load-balancer-attributes: "load_balancing.cross_zone.enabled=false"
service.beta.kubernetes.io/aws-load-balancer-healthcheck-healthy-threshold: "2"
service.beta.kubernetes.io/aws-load-balancer-healthcheck-unhealthy-threshold: "2"

# 10s or 30s
service.beta.kubernetes.io/aws-load-balancer-healthcheck-interval: "10"

# 2s is the minimum and is recommended to detect failure quickly
service.beta.kubernetes.io/aws-load-balancer-healthcheck-timeout: "2"
```

So with a healthcheck interval of 10s minimum, and an unhealthy threshold of 2, at least 20s will pass before the NLB will stop forwarding.

Listing 9: Gloo proxy readiness and graceful termination settings

```
# graceful shutdown:
# Envoy will fail health checks but only stop after 7 seconds
terminationGracePeriodSeconds: 37
gracefulShutdown:
  enabled: true
  sleepTimeSeconds: 35

...

topologySpreadConstraints:
  - maxSkew: 1
    topologyKey: topology.kubernetes.io/zone
    whenUnsatisfiable: DoNotSchedule
    labelSelector:
      matchLabels:
        gateway-proxy-id: external-gateway-proxy

...
```

We had to remove the hostname spread constraint, as it was causing issues with rolling deployments and the NLB (pods were scheduled on a new host, and the NLB target becomes unhealthy:

Listing 10: Gloo Pod topology constraints

```
    # Prefer to not schedule on the same node if possible
  - maxSkew: 1
    topologyKey: kubernetes.io/hostname
    whenUnsatisfiable: ScheduleAnyway
    labelSelector:
      matchLabels:
        gateway-proxy-id: external-gateway-proxy
```

With cross zone loadbalancing enabled on the NLB, we see intermittent request timeouts during rolling deployments, and even by just failing the healthcheck in Envoy without disrupting the service itself. Hence we disable it through the annotations:

Listing 11: Disable Load-Balancing Cross-Zone

```
service.beta.kubernetes.io/aws-load-balancer-attributes: "load_balancing.cross_zone.enabled=false"
```

We also updated the deploy strategy:

Listing 12: Deploy strategy

```
gatewayProxies:
    gatewayProxy:
      kubeResourceOverride:
        spec:
          strategy:
            rollingUpdate:
              # Equal to the number of availability zones
              # so that new scheduled pods get evenly distributed among AZ but on the same node
              # DO NOT SET to 1
              maxSurge: 2
              maxUnavailable: 0
            type: RollingUpdate
```

This will add 2 pods with the new deployment, and since they are scheduled simultaneously, they are placed as 1 pod per Availability Zone. Adding only 1 pod at a time could result in 2 pods in the same AZ, depending on the order of scheduling and termination.

Preserve Client IP setting on NLB target group still causes some issues when a node health check becomes unhealthy (e.g.: a single gateway-proxy pod on a node is scaling down). Disabling this preserve client ip setting fixes the issue, but then the real client ip is lost in the Gloo access logs - hence we had to leave it enabled.

Question 2: Explain how you would implement monitoring for Crossplane itself and the resources it manages.

Quick unordered summary:

- Crossplane Health:
  - Prometheus and Grafana for Crossplane metrics.
  - Centralized logging for Crossplane and controller logs.
- Managed Resources Health:
  - Use native cloud monitoring tools (AWS CloudWatch, GCP Stackdriver, Azure Monitor) to track the status of infrastructure.
  - Monitor custom resource statuses in Kubernetes for errors or failed reconciliation.
- API and Drift Monitoring:
  - Monitor API calls with AWS CloudTrail or(GCP and Azure equivalent).
  - Watch for frequent drift reconciliation to detect manual changes outside of GitOps.
- Alerting:
  - Set up alerts for reconciliation errors, resource failures, rate limits, and security anomalies using tools like Prometheus Alertmanager.

Prometheus, Grafana Dashboards and Kubernetes Health Checks

Prometheus: Crossplane and its related components expose metrics that can be scraped by Prometheus. **Grafana Dashboards**: Grafana dashboards are helpful to visualize the Crossplane metrics captured by Prometheus. As Crossplane manages resources declaratively as a Kubernetes Custom Resources, the 'automated way' to monitor them would be to leverage the **Kube Prometheus Stack** to scrape the resource statuses and create alerts for failures or issues in reconciliation. In a similar way, since Crossplane operates through **Kubernetes events**, you can monitor these events for issues related to "Events Management" by shipping the metrics to **Prometheus** and logs to **Loki**.

These are a couple things that you can set up Prometheus to monitor:

- The health and availability of Crossplane itself.
- The status of Crossplane controllers (e.g., whether they are reconciling resources properly).
- The performance of Crossplane's reconciliation loops.
- Error rates in creating, updating, or deleting managed resources.

You can visualise all metrics captured by Prometheus in Grafana Dashboards, as well as:

- The history of reconciliation success/failure rates.
- The number of external resources under management.
- Time and latency in resource provisioning.
- Any time-related metric that could giving us an overview of Crossplane health.

Here are a few example `PromQL` queries for Crossplane:

- Tracks the error rate in resource reconciliation over a 5 minute window: `rate(crossplane_reconcile_errors_total[5m])`.
- Measures the duration of reconciliation processes: `crossplane_reconcile_duration_seconds_bucket`.

Using Kubernetes Health Checks give us information on the pods' health

Prometheus and Grafana integrate very well with cloud native tools: such as `Kube-State Metrics` or `Node Exporters` to view the health of the Crossplane pods. As a matter of fact, these tools can also all ship their metrics to **Prometheus**, which, makes them easy to use with **Grafana Dashboards** to visualize better where some issues could arise.

Crossplane Logs and Monitoring Security

Shipping the pod Logs to Loki (or equivalent such as `E.L.K Stack` or `FluentD`). Crossplane needs also to be monitored for security-related events. This involves tracking for unauthorized access attempts or any network anomaly. Here is a non-exhaustive list on some logging tools and topics logs can help identify:

- **Reconciliation errors**: Logs that show issues provisioning or updating resources.
- **Crossplane controller logs**: To detect issues with controllers responsible for managing infrastructure providers.
- **AWS CloudTrail** can help detect which IAM Role or IAM User is trying to access Crossplane. ecosystem.
- **VPC Flowlogs** can help detect as well if there's any weird traffic going on.

Cloud Provider Resource Monitoring:

If we run Crossplane in AWS then we can use a few of the AWS resources to monitor Crossplane 'AWS Resources':

- **AWS Cloudwatch**: can be used to monitor for EC2, S3, RDS etc ...

- **AWS CloudTrail**: can be used to track API requests made by Crossplane, allowing to monitor failed API calls, unusual activity, or excessive API throttling.
- **VPC Flowlogs** can help detect traffic anomalies in the VPC where Crossplane is hosted.

Prometheus can track such errors by scraping logs or metrics related to rate limiting. Setting up some **monitoring for API rate limits on AWS** can become handy to prevent Crossplane hitting cloud provider limits - this is common when handling a large infrastructure at scale.

Configuration Drifts:

Crossplane continuously reconciles the desired state defined in the GitOps repository with the actual state of the managed infrastructure. If someone makes manual changes directly to the cloud resources, Crossplane will detect this "drift" and attempt to bring the infrastructure back in sync.

Monitoring tools can be configured to alert you if drift reconciliation is triggered frequently, indicating someone is bypassing GitOps processes or unexpected changes are occurring.

Alerting

I have mainly used **Alertmanager** and **OpsGenie** combined with **Slack** notifications/messages on specific "alerting" channels. We can use **Alertmanager** and hook it up to some `PromQL` queries that will trigger alerts and send them to the OpsGenie (pager) to notify the Platform Team.

Question 3: In the context of an IaC GitOps Repository, explain the workflow(s) you would put in place to safely handle teamwork and change management.

Summary Workflow

- Develop Feature: Developer checks out a new feature branch, implements changes, and creates a PR.
- Automated Tests: CI/CD runs static analysis, security checks, and tests, then outputs `terraform plan` or similar.
- Code Review: Peers review the code, check the plan, and test results. If everything passes, the PR is merged.
- Deploy to Staging: The merge triggers automatic deployment to a staging environment.
- Test in Staging: Teams test in staging to verify that the changes behave as expected.
- Approval for Production: Once staging tests pass, the changes are approved for production.
- Deploy to Production: The code is automatically applied to the production environment.
- Monitor and Audit: After deployment, monitor logs, alerts, and audit trails for any issues.

In an Infrastructure as Code (IaC) GitOps repository, teamwork and change management are critical to ensuring stability, collaboration, and security. The goal is to make infrastructure changes in a way that is traceable, peer-reviewed, and automated, minimizing the risk of errors while fostering efficient team workflows.

This workflow ensures smooth teamwork, safe change management, and continuous improvement within an IaC GitOps repository. I would not be surprised to handle **ALL** Infrastructure needs in the IaC GitOps repository, such as:

- Code to provision the cloud provider: such as **Terraform**, **Terragrunt** or **Pulumi**
- Code to provide "Ground Services" for Kubernetes: such as **Helm** or **Pulumi** for Kubernetes.

Modules and libraries would be in individual repositories with semantic versioning enabled. This would allow the main IaC GitOps Repository to be versioned as well.

Here's a detailed breakdown of the key workflows:

Branching Strategy

I personally have implemented and successfully used with my previous teams, both a `Git-Flow` and a `Git-Trunk` approach, to manage and collaborate on a large IaC repository. Both had their pros and cons. A solid branching strategy ensures that changes are developed and tested in isolation before being merged into production. A simple and typical strategy might look like this:

- **Main Branch**: Represents the production state of the infrastructure.
- **Develop Branch**: Represents the integration branch, where features are merged after testing but before release.
- **Feature Branches**: For new infrastructure features, updates, or bug fixes. Each team member creates a branch from the `develop` branch for their changes.
  Example: `feature/load-balancer-improvement`
- **Hotfix Branches**: For urgent changes or fixes that need to be applied quickly to production.
  Example: `hotfix/fix-dns-issue`

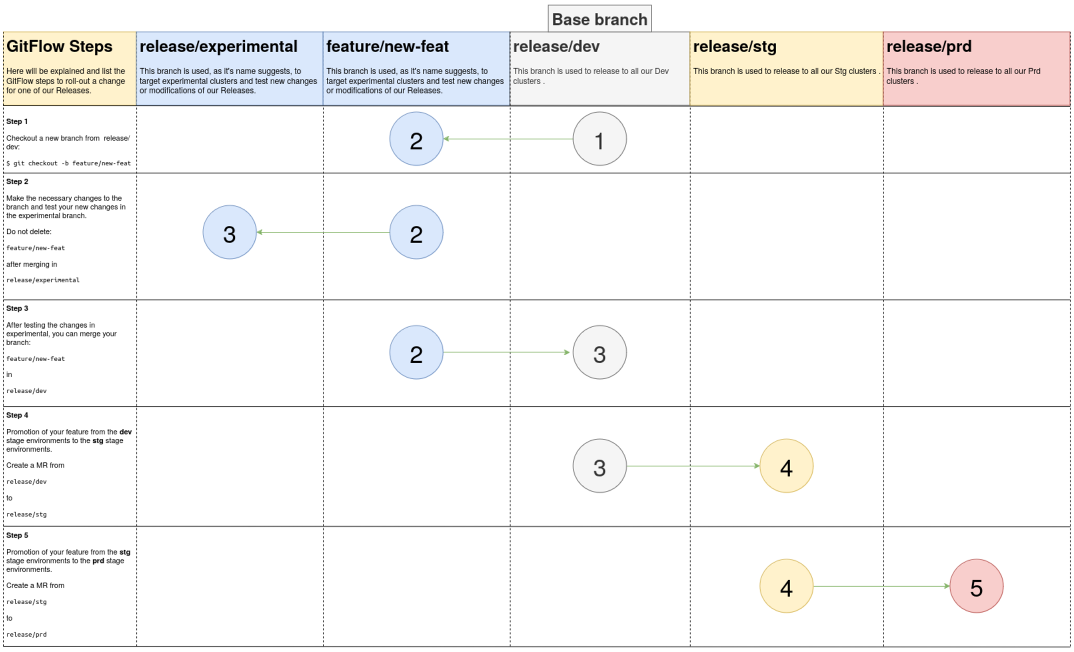
Here's a more complex git branching setup; the following branches must exist:

- `release/experimental`: branch designed for testing and breaking - which targets Ephemeral / Experimental Environments.
- `release/dev`: default branch - which targets all Dev Environments.
- `release/stg`: staging branch - which targets all Staging Environments.
- `release/prod`: prod branch - which targets all Production Environments.

The merge flow would be as follows:

1. Engineer checks out the `release/dev` branch into a feature branch: `git checkout -b feature/something-new-and-cool`.
2. Engineer pushes the modifications on his `feature/something-new-and-cool` branch, and merges it - WITHOUT deleting the branch - in `release/experimental`.
3. CICD checks happen on the `release/experimental` branch - and once all is green and sucessfull
4. Engineer can merge his branch `feature/something-new-and-cool` into `release/dev`.
5. Engineer can then promote his changes with changes of his other peers: merging `release/dev` into `release/stg` then again `release/stg` into `release/prod`.

By having ArgoCD setup to monitor specific branches and to target specific environments, we are sure that we are rolling out only changes based on the environment.



GitFlow Steps Here will be explained and list the GitFlow steps to roll-out a change for one of our Releases.	**release/experimental** This branch is used, as it's name suggests, to target experimental clusters and test new changes or modifications of our Releases.	**feature/new-feat** This branch is used, as it's name suggests, to target experimental clusters and test new changes or modifications of our Releases.	Base branch **release/dev** This branch is used to release to all our Dev clusters .	**release/stg** This branch is used to release to all our Stg clusters .	**release/prd** This branch is used to release to all our Prd clusters .
Step 1 Checkout a new branch from release/dev: `$ git checkout -b feature/new-feat`		2	1		

Step 2
Make the necessary changes to the branch and test your new changes in the experimental branch.
Do not delete:
`feature/new-feat`
after merging in
`release/experimental`
	3	2			

Step 3
After testing the changes in experimental, you can merge your branch:
`feature/new-feat`
in
`release/dev`
		2	3		

Step 4
Promotion of your feature from the **dev** stage environments to the **stg** stage environments.
Create a MR from
`release/dev`
to
`release/stg`
			3	4	

Step 5
Promotion of your feature from the **stg** stage environments to the **prd** stage environments.
Create a MR from
`release/stg`
to
`release/prd`
				4	5

Pull Request (PR) Workflow

The use of pull requests ensures that infrastructure changes are reviewed before they are merged into the main branch. This allows for code quality checks, feedback, and collaborative problem-solving.

- **PR Creation**: When a team member completes changes on a feature branch, they create a PR against the `develop` or `main` branch, depending on the deployment cycle.
- **Code Reviews**: A minimum number of peer reviews (e.g., 2) must be completed before the PR is approved and merged. The review focuses on:
  - Correctness of changes
  - Security implications
  - Compatibility with existing infrastructure
  - Adherence to best practices
- **Automated Checks**: PRs trigger automated tests (e.g.: unit tests for Terraform) and linters that verify the syntax, security scans, and infrastructure compliance.
- **Merge and Deploy**: Once approved, the PR can be merged into `develop` for further testing or into `main` for deployment to production. GitOps automations can ensure that this deployment is automatically triggered.

Continuous Integration / Continuous Deployment (CI/CD) Pipelines

The CI/CD pipeline is key to automating the process of testing, validating, and deploying infrastructure changes. Commonly, this is achieved using GitOps tools like Argo-CD, and works wonderfully in pair with Argo-Rollouts and Argo-Workflows.

Testing Stage:

- **Static Code Analysis**: Tools such as `terraform validate` or `tflint` are run to ensure the code meets standards.
- **Unit Testing**: Terraform modules or other scripts can be tested using frameworks like `Terratest`.
- **Plan Execution**: The CI/CD pipeline should execute `terraform plan` (or ArgoCD diff-view) to show the exact infrastructure changes without applying them. The output should be visible and reviewed as part of the PR. For `terraform` and `terragrunt` a tool such `Atlantis` makes the changes clear and visible in the PR.
- **Ephemeral Environments Testing**: Argo Workflows can help us spawn ephemeral Cluster in Cluster (with tools such as `vCluster`) environments to test what the changes in a simulated "environment".
- **Canary or Blue-Green Deployments**: Argo Rollouts can help understanding how the changes are impacting the cluster.

Deployment Stage:

- **Staging Environment**: Before production, changes are automatically applied to a staging environment where further manual or automated validation occurs. This reduces the risk of introducing changes that break production.
- **Production Deployment**: Upon final approval, changes are automatically applied to the production environment.

Change Approval Process

In a team environment, some changes, particularly those impacting sensitive infrastructure, should go through a formal change control process.

Change Requests: Major infrastructure changes should require a formal change request in your version control system (e.g., GitHub Issues or Jira), including details on: Justification for the change, Rollback plans, Impact analysis.

Approvals: Require that senior engineers or architects review, approve and understand these changes before they are merged.

Version Control and Tagging

For change tracking and rollback, version control and tagging are essential.

- **Versioned Releases**: When merging to the main branch, each release should be tagged with a version number, especially when managing multiple environments (e.g., dev, staging, prod).
- **State Locking**: For Terraform, enabling state locking (with backends like S3 and DynamoDB) is crucial to prevent multiple people from applying changes simultaneously.

Automated Drift Detection

Infrastructure drift occurs when changes are made to live infrastructure that aren't reflected in the code repository. Automated drift detection mechanisms should be in place:

- **Continuous Reconciliation**: Tools like Flux or ArgoCD can continuously reconcile the state of the infrastructure with the declared Git state, applying changes if drift is detected or flagging issues.
- **Manual Notifications**: Send alerts or notifications to team channels (like Slack or email) if drift is detected, so it can be quickly corrected.

Audit and Logging

Maintaining an audit trail of changes is key to ensuring security and compliance.

- **Commit History**: Git provides a detailed log of every change made to infrastructure code, including the who, what, and when of changes.
- **Audit Logs**: Enable logging at the IaC level (e.g., AWS CloudTrail, Azure Activity Logs) to track infrastructure actions at a resource level.
- **Notifications and Alerts**: Set up automatic alerts for any changes to sensitive infrastructure, helping teams react quickly to potential issues.

Secrets Management

Handling secrets in an IaC repository requires careful attention to prevent leakage.

- **Secret Vaulting**: Use tools like HashiCorp Vault, AWS Secrets Manager, or Azure Key Vault to manage sensitive information. Secrets should never be hard-coded into the infrastructure code or stored in the Git repository.
- **Secure Access**: Ensure that only authorized users and pipelines have access to secrets, and audit access regularly.

Audit and Logging

Maintaining an audit trail of changes is key to ensuring security and compliance.

- **Commit History**: Git provides a detailed log of every change made to infrastructure code, including the who, what, and when of changes.
- **Audit Logs**: Enable logging at the IaC level (e.g., AWS CloudTrail, Azure Activity Logs) to track infrastructure actions at a resource level.
- **Notifications and Alerts**: Set up automatic alerts for any changes to sensitive infrastructure, helping teams react quickly to potential issues.

Question 4: Create an automated procedure for deploying and updating a Backstage instance while gracefully handling failures. It would be nice to provide a way to test your procedure locally (for example, with containers) and a document with instructions on how to use your method.

No particular technology is required you can choose to use whatever you like Your procedure should be resilient to errors and leave the installation in a working state even if something wrong happens during an update

You can find the project details in my Github Repository:

https://github.com/chess-seventh/assignment-fpiva-consensys

All details are in the README, feel free to reach out so we can discuss this further.