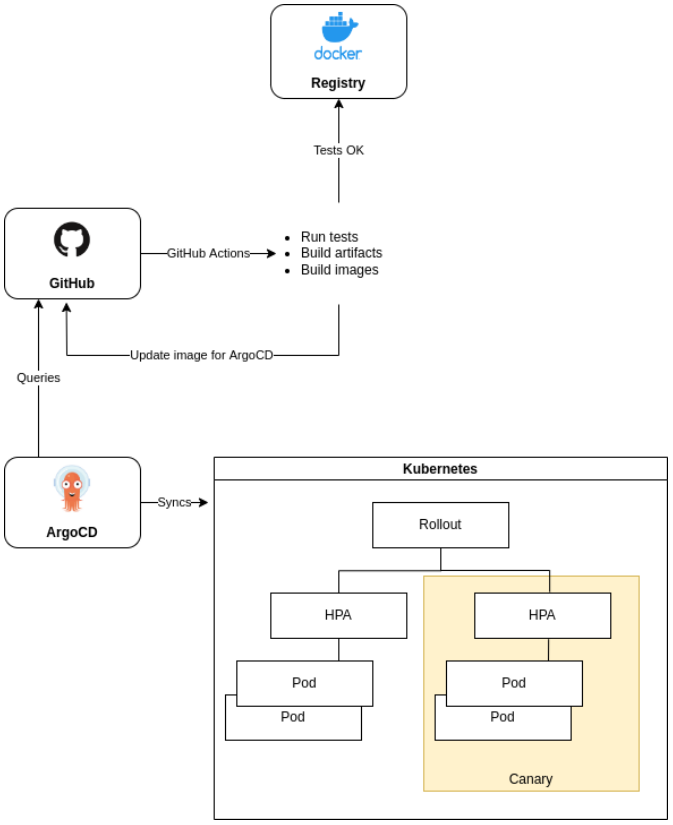
Product service project proposal

CI/CD

This proposal for a CD pipeline serves two purposes:

- Continuous delivery of the application
- Control and configuration of the Kubernetes objects related to its deployment

The basic design for the CD pipeline is as follows:



The flow proposed is:

- Commit
- Maven tests are run
- Build artifact
- Build docker image (tagged with commit sha)
- Deploy docker image to registry
- Update the rollout descriptor with the new image

With the tests added in the deployment phase we can ensure a proper canary deployment that is gradual and safe.

This proposal uses GitHub Actions so we don't add additional requirements (ex: Jenkins) in our deployment process.

Possible improvements:

- Include some kind of load testing as a test
- Trigger Argo with a webhook instead of relying on time sync
- Separate workflows and have them call each other when appropriate
- Add pre-production environment
- Limit CD to tags/certain branches
- Every time actions run, the repository updates so no local repo is ever up to date
- Security and dependency testing

Possible alternatives:

- Jenkins/JenkinsX, but only for the build phase, we are leveraging Istio/Argo integration for deployment rollouts.

Observability

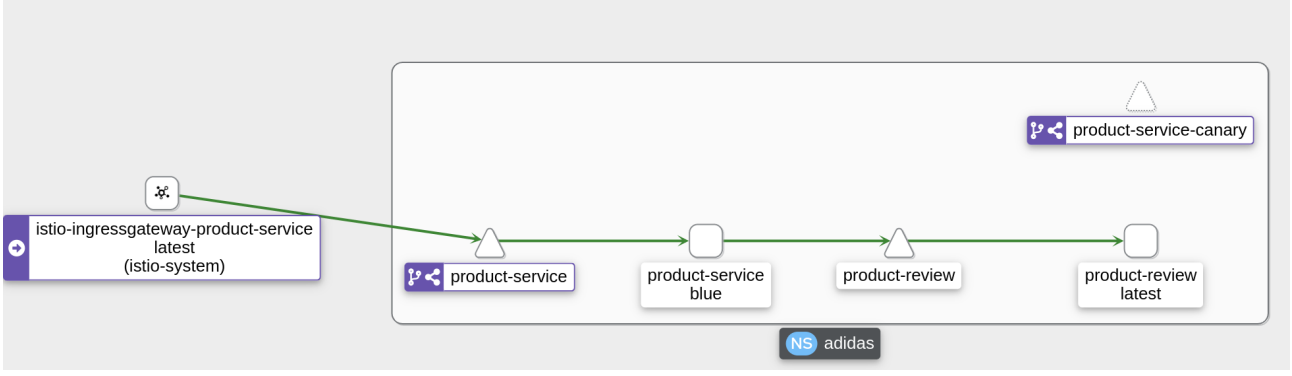
For observability the proposal contemplates two domains with related SLOs:

- Kubernetes objects / infra domain
  - Round-trip latency 99.5 pct is 200ms
  - Response is 200OK for 99.5% of requests
- Application domain
  - Time spent in product-service 99.5 pct is 120ms

To accomplish this, we will deploy the following:

- Istio (+Kiali)
  - Service mesh that provides traffic shaping, tls, and observability
- Jaeger
  - Tracing
  - Depending on traffic volume, we may have to deploy Kafka as buffer
- Prometheus (+Alertmanager)
- Grafana
- Loki
  - Log aggregation at the kube-node level
  - Integration w/ Grafana

Here we can see the traffic for a normal state of the application:

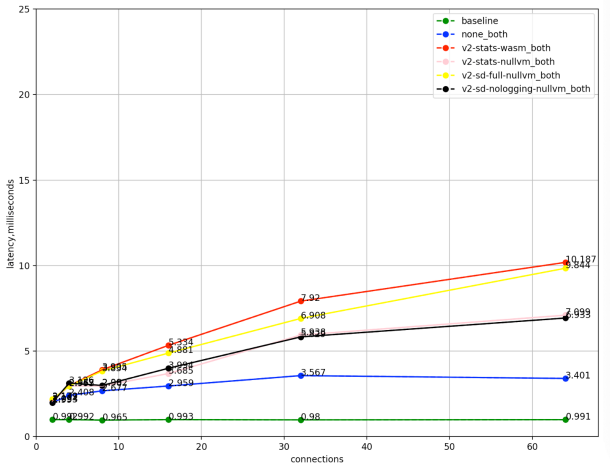


We identify several key metrics in the Kubernetes domain:

- Traffic volume ingested by Istio ingress gateway
  - Alarm when traffic volume varies by X% in Y timeframe
- Success rate for Istio ingress gateway
  - Alarm when SLO is getting breached
  - Alarm when X% of requests suffer Y% variance in Z timeframe
- Latency results for Istio ingress gateway
  - Alarm when SLO is getting breached
  - Alarm when X% of requests suffer Y% variance in Z timeframe
- Per pod volume and success rate
- Product review volume and success rate
  - Alarm when success rate drops to X%

The first three are fundamental in order to meet our SLOs and the rest may provide useful information. We don't consider alerting based on logs useful and defer them to debugging/monitoring purposes.

The overhead of Istio is pretty minimal, with ~5.9 ms in our current configuration.



Since we depend on products-review, we propose a blackbox monitoring approach to test for availability besides the implicit monitoring we do in app.

For the application domain, we propose:

- Latency for round-trip in product service
- Time spent in product-review API call
- Status code of both
- Integrate tracing with a 1/1000 sample rate

We assume that the cluster already has basic container monitoring (ex: cAdvisor) and we can monitor CPU/memory usage of the app pods.

Possible improvements:

- Fine tune alarms so we don't go too much below error budget
- Alerts at the JVM level
- Minor info alerts
- Alerts for regressions
- "The best data is more data" – not necessarily for alerting
- Decouple Prometheus from the TSDB – sizing problems

Possible contender: Linkerd vs Istio

Although Linkerd is faster and is less resource intensive, Istio provides more general use cases (which is paid by increasing complexity). Both products support a similar set of features and for this proposal both can be used.

SLO

As mentioned earlier, we will leverage Istio ingress gateway monitoring to ensure our SLOs are met. We have chosen the ingress gateway as it provides round-trip metrics for product-service. To ensure we meet our SLO we will be scaling the HPA using req/s.

In order to properly manage our error budget, we will implements alerts based on rate increases possibly impacting our SLO (ex: Prometheus predict_linear) so we don't burn our entire budget while allowing some room for failures.

Possible improvements:

- Maybe external exporters w/ Prometheus Gateway that can query the service and provide external metrics besides the ones we get from Istio.

Deployment

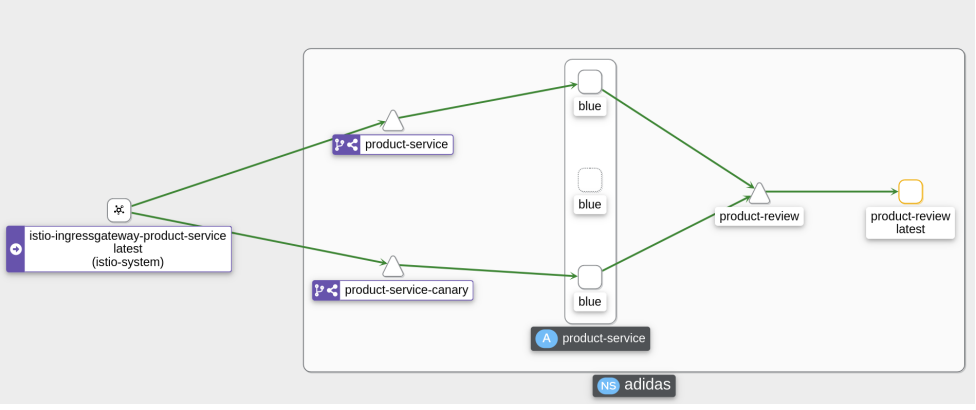
In order to met the CD capabilities and HA requirements we have decided to use a canary rollout of 1 HPA with a minimum 2 replica and maximum 6, scaling based on req/s sourced from Istio.

The deployment process is:

1. Argo picks up new image after scanning GitHub
2. Creates additional HPA with the new image
3. Redirects traffic to new HPA
    1. 20% of inbound traffic for 1 minute
    2. Runs analysis (checks for the last minute % of 5XX error, if < 0.5%, analysis passes, several runs with pause inbetween)
    3. 50% of inbound traffic for 2 minutes
    4. Runs analysis
4. Promotes the new image as production image

This process allows a good balance between hands-off approach and cautious testing while ensuring minimal performance degradation in a new deployment.

Here we can see the traffic shaping in action in a new deployment:



Since we are deploying an HPA, if there is a combination of high volume and a new deployment we run the risk of maxing both HPAs for the duration of the testing, doubling our resource consumption.

The HPA will be triggered by req/s sourced from Istio. After conducting tests we determined that 1 pod can handle ~250req/s before triggering the autoscaler, so we will assign 200req/s to each pod to ensure proper margins.

Possible improvements:

- Roll back procedure in worst-case would be manual
- Capacity planning
- Improve granularity of the load testing

Resiliency

Since we are fully dependent on product-review, we propose caching the calls after being made using Memcached. This will allow us to withstand a failure of product-review (although degraded) while the service recovers.

If we add Memcached, we need to add basic monitoring at the pod level (CPU/memory), the cache stats (hit/miss, drop from cache…) and an SLO for stale content (99.5% of the content we serve has to be <5m old, for example)

Depending on traffic volume and distribution (peaks, etc) we may need to do a proper analysis and capacity planning in case a region goes out.

For proper analysis we recommend:

- Identify failure modes of the application
- Identify failure modes and domains of the underlying infrastructure
- Conduct Fault Tree Analysis to better understand the relationship between subsystems and product-service

In addition to that, we propose the creation of runbooks and recommended response steps. Product-service is simple enough that doesn't need developing specific self-healing, and since it is stateless it should be able to recover easily. We could improve recovery times and/or elasticity by reducing the memory and CPU footprint.

Possible improvements:

- Model requests/cache ttl relation
- Having the cache after API calls, makes it only useful in a disaster scenario, having it before, a lower TTL reduces stale content, but reduces time margin for disaster resiliency
- Document/automate/procedure/add rollback capabilities and circuit breakers to pipeline. Manual override is possible, but improvements can be made