reflections on CKI architecture

OSCI version

Michael Hofmann

Iñaki Malerba
(retired from CKI)

team background

- CKI: Continuous Kernel Integration - CI as a service
- prevent bugs from being merged into kernel trees
- managing the CI infrastructure for Red Hat's kernel development
- in a nutshell:
  - GitLab pipeline per kernel revision, testing in Beaker
  - platforms: OpenShift, OpenStack, Beaker, AWS EC2
  - RabbitMQ AMQP messaging cluster hosted on AWS
- home page and documentation: https://cki-project.org
- code: https://gitlab.com/cki-project
  - one GitLab CI pipeline and ~ 70 microservices/cron jobs
  - ~20 changes/day merged and automatically deployed to production

prelude

actually, what is CKI?

- home page: "finding kernel bugs before they hit your distribution"
- what we thought:
  - develop Python code to build and test kernel patches
  - what is important for development: DevOps!11!!
- what we actually do:
  - run Kernel Testing as a Service
  - what is important for running a service: ???

    (later we found out it it is called SRE 🤦)

Red Hat

early experiences

- there was only one person to fix a bug ("domain expertise")
- no one knew what was deployed where ("development-focused")
- unclear internal dependencies ("machine under some desk")
- brittle external dependencies ("cluster is down")
- silent failures ("certificates expired")
- manual recovery ("tech lead is clicking buttons to restart jobs")

early conclusions

- team-maintenance (but differently structured code everywhere)
- GitOps for deployment (but how)
- document/manage dependencies (but how to find all of them)
- reduce (influence of) external dependencies (but what to keep)
- monitor failures (but how to find and surface them)
- automated recovery (but what does that even mean)

so what should CKI be

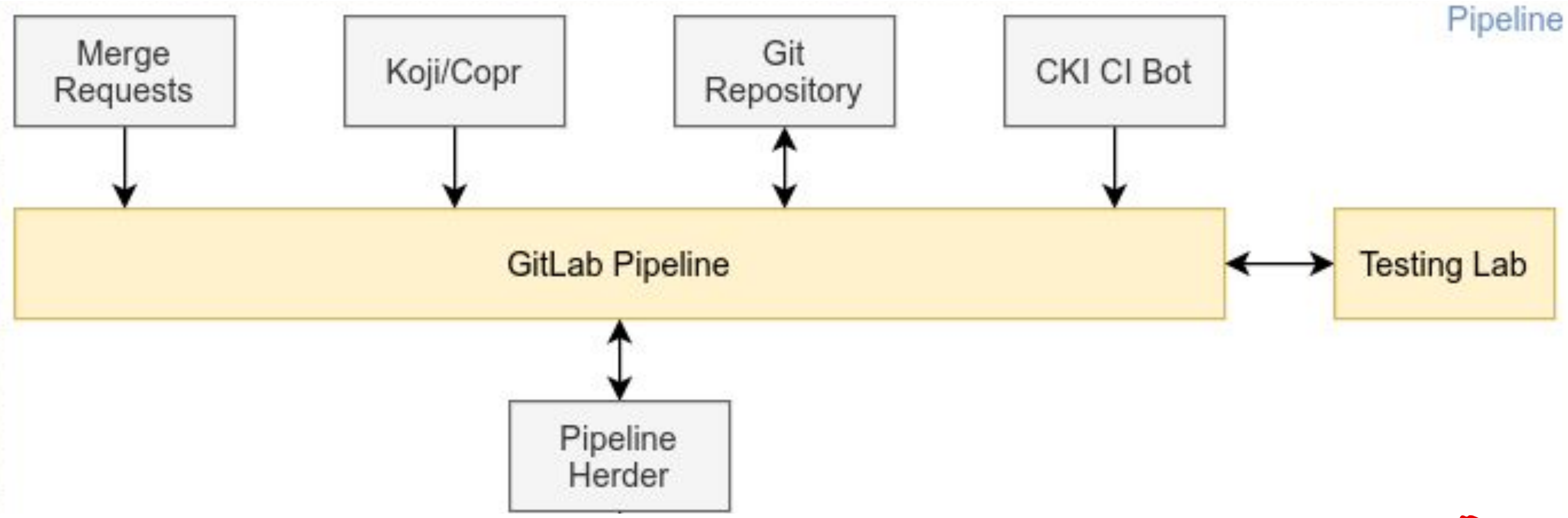
- common repo setup and code structure
- automated central service deployment
- managed infrastructure, no manual intervention allowed
- strategic thinking about required external services
- monitoring and alerting setup
- automated recovery at all layers

our product owner (PO) mentioned that we were also supposed to have

- a Gitlab CI pipeline with Beaker testing for kernel development 🤦

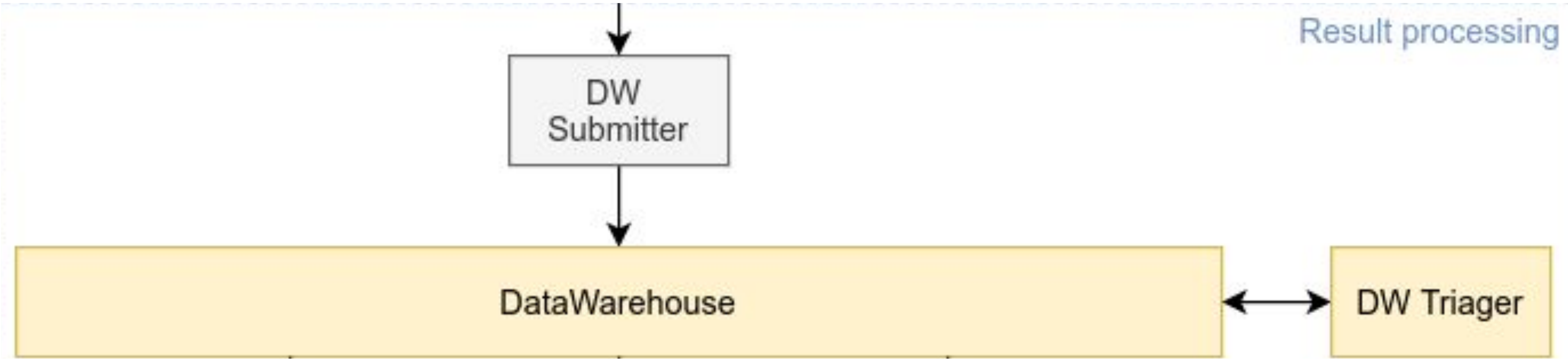
right, back to CKI architecture

PO says 1/3: Gitlab CI pipeline

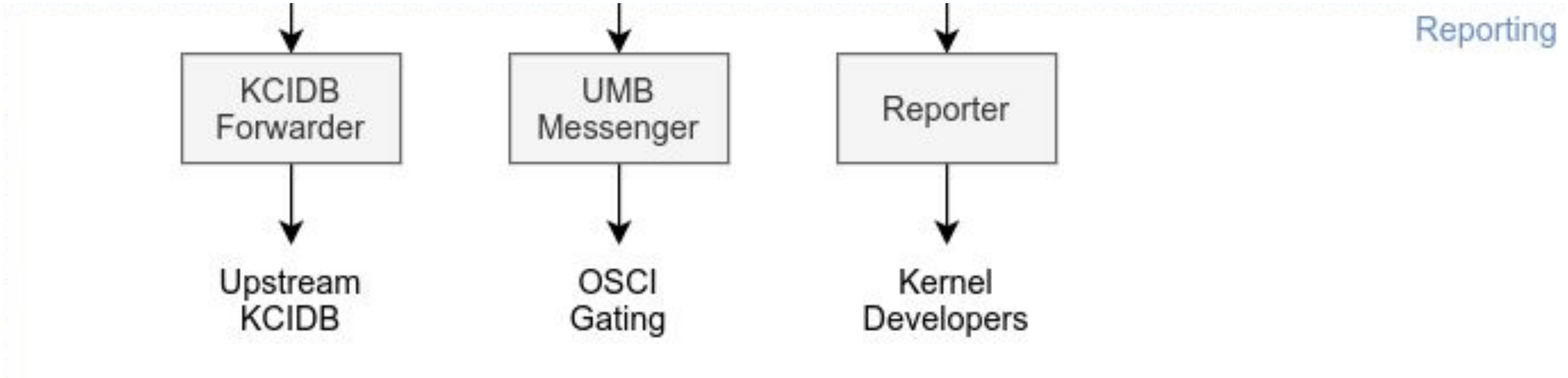


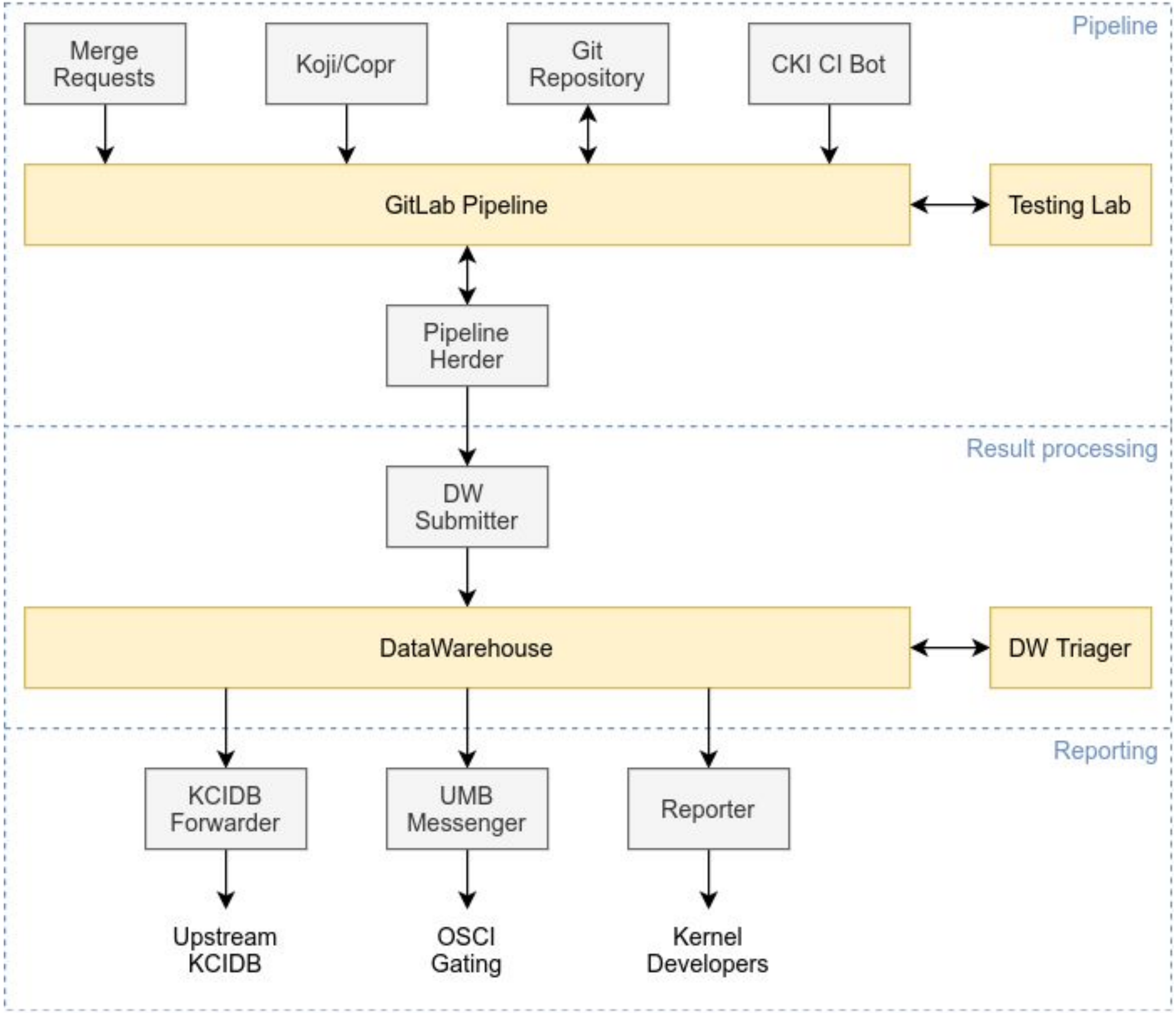
Red Hat

PO says 2/3: result storage, analysis and known issue detection



PO says 3/3: reporting and gating




Pipeline
Merge Requests
Koji/Copr
Git Repository
CKI CI Bot
GitLab Pipeline
Testing Lab
Pipeline Herder
Result processing
DW Submitter
DataWarehouse
DW Triager
Reporting
KCIDB Forwarder
UMB Messenger
Reporter
Upstream KCIDB
OSCI Gating
Kernel Developers

what your PO forgot to tell you

we found out that users are only happy if these are taken care of as well:

- continuous integration (CI):
  - common repo setup, code structure and testing
- continuous delivery/deployment (CD):
  - automated central service deployment
  - managed infrastructure, no manual intervention allowed
- site reliability engineering (SRE):
  - monitoring and alerting setup
  - strategic thinking about required external services
  - automated recovery at all layers

(and this is what the following slides are about)

continuous integration

standardized CI setup

- container images for delivery
  - tag images with :p-1234, :mr-1234, :latest
- share across projects:
  - job templates: building, tagging, testing, deployment, …
  - Containerfile snippets: entrypoint, Python service, cleanup, …
  - CI pipeline and container image builder images
  - common Python library: logging, Sentry, Prometheus metrics, …
  - linting/tox helper: pylint, trigger dependent project pipelines, …
  - coverage check: fail on regression, GitLab UI visualization
- GitLab project configuration as code: approval rules, …

Containerfile fragments

```
# setup

FROM registry.access.redhat.com/ubi8/ubi
RUN echo "install_weak_deps=false" >> /etc/dnf/dnf.conf

# python

/* Python stack */
RUN dnf install -y git-core python39 python39-devel \
    python39-pip python39-setuptools python39-wheel

RUN mkdir -p /code
COPY . /code
WORKDIR /code
RUN python3 -m pip install -e . && dce_pip_install.sh
CMD ["dce_entrypoint.sh"]

# cleanup

/* Update everything possible */
RUN dnf --skip-broken -y update

/* Remove build artifacts */
RUN dnf clean all && rm -rfv /root/.cache /var/cache /var/log
```

- Containerfiles preprocessed by cpp
  - comments with /* foo */ instead of # bar
- Encapsulate common steps:
  - setup and cleanup
  - Python application, …
- Example Containerfile for a Python application:

```
#include "setup"

EXPOSE 5000/tcp
ENV DCE_START_FLASK="dce_app_1.greeting"

#include "python"
#include "cleanup"
```


Red Hat

common library

```python
from cki_lib import messagequeue
from cki_lib import metrics
from dce_common import logging
from dce_common import misc

# ENV DCE_LOGGING_LEVEL
LOGGER = logging.get_logger('dce.my-app')


if __name__ == '__main__':
    # ENV METRICS_{HOST,PORT}
    metrics.prometheus_init()
    # ENV SENTRY_DSN, DCE_ENVIRONMENT
    misc.init_sentry()

    # ENV DCE_ENVIRONMENT
    if not misc.is_production():
        LOGGER.warning('Not running in production mode')

    # ENV RABBITMQ_{HOST,PORT,USER,PASSWORD,QUEUE,EXCHANGE}
    messagequeue.MessageQueue().consume(callback)
```

- Consolidated code as a highway to unification
  - CKI_DEPLOYMENT_ENVIRONMENT
  - logger configuration for sane debugging
  - Prometheus metrics, message queues
  - https://gitlab.com/cki-project/cki-lib

code coverage

```
# .gitlab-ci.yml
lint_and_test:
  coverage: '/^TOTAL.*\s+(\d+\%)$/'
  artifacts:
    reports:
      cobertura: coverage/coverage.xml

# dce_lint.sh
coverage run
coverage report -m
coverage xml -o coverage/coverage.xml
```

- Specify coverage regex to parse job output
- Export raw coverage data for visualization:

```
@@ -11,5 +11,7 @@ def hello_world(name: str) -&gt; str:

 def goodbye(name: str) -> str:
     """Output a customized farewell."""
+    if os.environ.get('DCE_FAREWELL') == 'no':
+        return f'<p>I cannot let you go, {name}!</p>'
     farewell = os.environ.get('DCE_FAREWELL', 'goodbye')
     return f'<p>{farewell.capitalize()}, {name}!</p>'
```

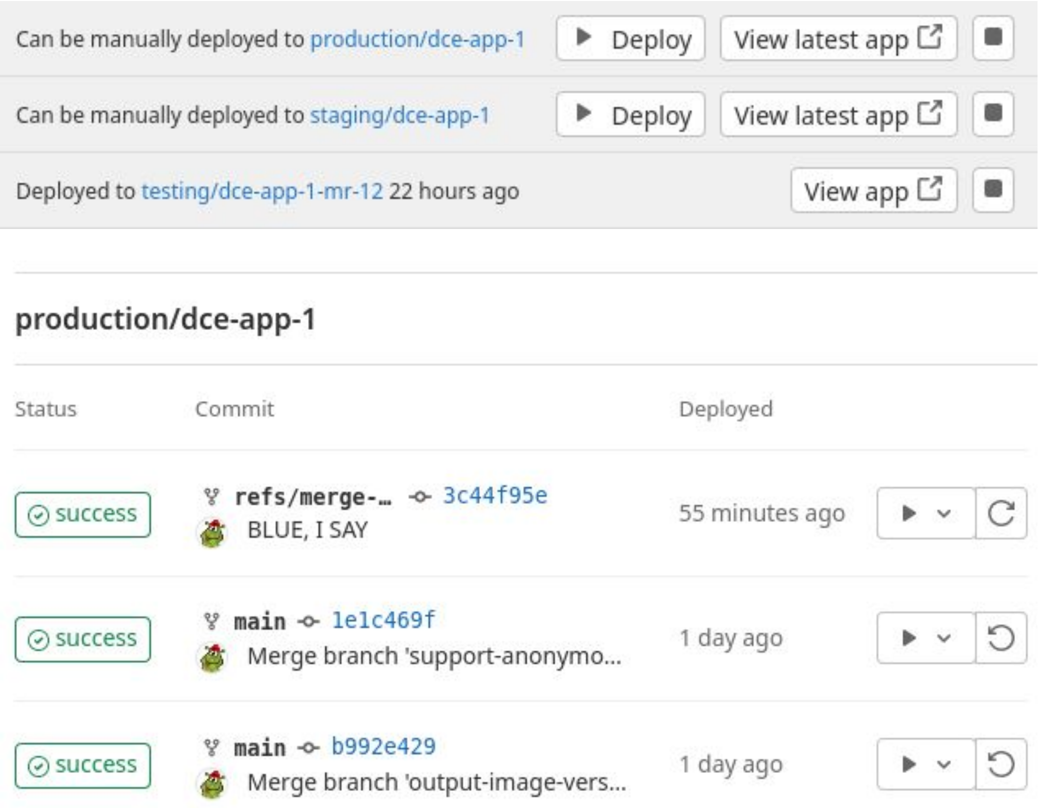

Red Hat

continuous delivery/deployment

automated central service deployment

- one deployment repository to rule them all
  - centralization of all infrastructure knowledge and processes
  - simple to support multiple Kubernetes clusters, environments
  - easy onboarding of yet another microservice
  - common parts heavily templated
- merge to main redeploys everything: ~200 jobs
  - keeps everybody honest
  - jobs can also be run manually from deployment repository MRs

specific deployments



- source repo changes: redeploy one service
  - automatically on merge to main
  - manually from merge requests
- deployed via
  - tag image with :staging/:production
  - trigger limited deployment pipeline
- rollbacks:
  - GitLab environments have history
  - rollback/rollout buttons via old jobs

Red Hat

templating Kubernetes resources

- ~ansible-lite: custom Jinja2 template processor
- global variables from external YAML files/dictionaries
- Jinja2 macros/templates to do the heavy lifting
- why not Helm: know exactly what is deployed
  - really, there is nothing inherently ✨✨✨ about k8s deployments
- DRY Kubernetes service specs
  - do what I mean
  - one central place to configure all deployments

Red Hat

Kubernetes microservice example

```
# micro-service/project-vars.yml.j2
secret:
 - SENTRY_DSN: $MICRO_SERVICE_SENTRY_DSN
deployment:
  image: quay.io/cki/micro-service:production
  variables:
    - RABBITMQ_HOST: {{ cki_variable('HOST') | tojson }}
    - RABBITMQ_USER: cki.consumer
    - RABBITMQ_PORT
  volumes:
    - {pvc: $PROJECT_NAME, mountPath: /data}
storage: 100Gi
services:
  - port: 8000
    route: {cki_project_subdomain: micro.internal}
    metrics: true
serviceaccount:
  - apiGroups: ['']
    resources: [services, endpoints, pods]
    verbs: [get, list, watch]
```

- projects-vars.yml for common configuration
- Jinja2 templates/macros hide all the magic
  - sensible defaults, easily customizable
- variables/secrets (Hashicorp Vault) helper
  - available in Bash, Ansible, Python, Jinja2
- create/reference PVCs, proper storage class
- expose a service via a route/DNS
  - metrics collection, liveness probe
- custom service accounts
- init containers, cron jobs, config maps, …

managed infrastructure

- serverless > containers (Kubernetes) > disposable VMs > pet VMs
- AWS Lambda/EC2, OpenStack, Beaker, PSI/ROSA/MP+ K8s
- exclusively configured via Ansible/Jinja2 templating
- ~infosec-ready machines
  - automated updates/reboots once per week
  - Qualys monitoring hooked up to Grafana/alertmanager

site reliability engineering

detection: monitoring and alerting setup

- detecting issues in build/test pipelines, u-services, cron jobs, FaaS
- logging (Loki)
- monitoring (Prometheus)
- visualization (Grafana)
- exceptions (Sentry)
- alerting (Alertmanager)

Loki and Prometheus

- logging via Loki:
  - standardized Python logger names, levels
  - Loki for processing: *‘Like Prometheus, but for logs!’*
  - log-based alerting
- expose metrics via Prometheus endpoints:
  - Expose internal status of services
  - Python’s prometheus-client: simple onboarding/deployment
  - K8s autodiscover: automatic monitoring of all services


Datawarehouse / Micro Services
Last 6 hours
1m
Triager
Message Processing Time
Production Queues
Pods Count
2
KCIDB Forwarder
Message Processing Time
Production Queues
Pods Count
1
KCIDB Submitter
Message Processing Time
Production Queues
Pods Count
2
Pipeline Herder
Message Processing Time
Production Queues
Pods Count
1

Sentry and Alertmanager

- collect exceptions via Sentry:
  - know before your users and track errors in real time
  - allows to fix the long tail of unlikely errors
- surface alerts via Alertmanager:
  - mail, dedicated Slack channel
  - emergency alerts via text messages

reliable service with unreliable dependencies

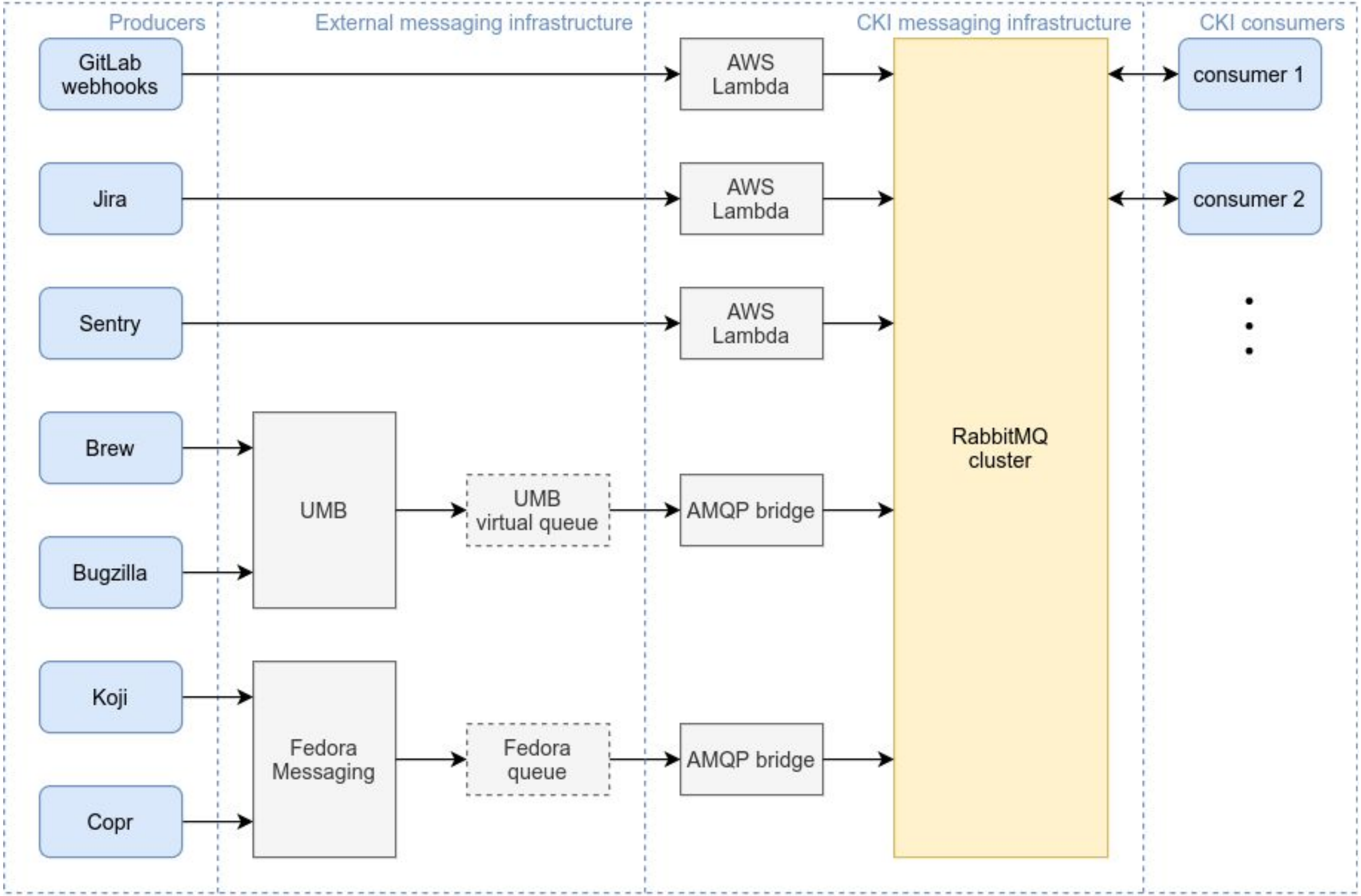
- lemma: any dependency that can fail will fail
  - ... some will fail more than others
- categories:
  - essential (needed for service to run): gitlab.com, AWS, Beaker
  - necessary (should work): microservices (K8s), result database
  - optional (nice to have): observability stack
- reduce dependencies and increase portability:
  - NFS/volumes/local storage/... -> move to S3
  - unreliable sources -> mirror on S3/quay.io
  - buildroot -> freeze via container images

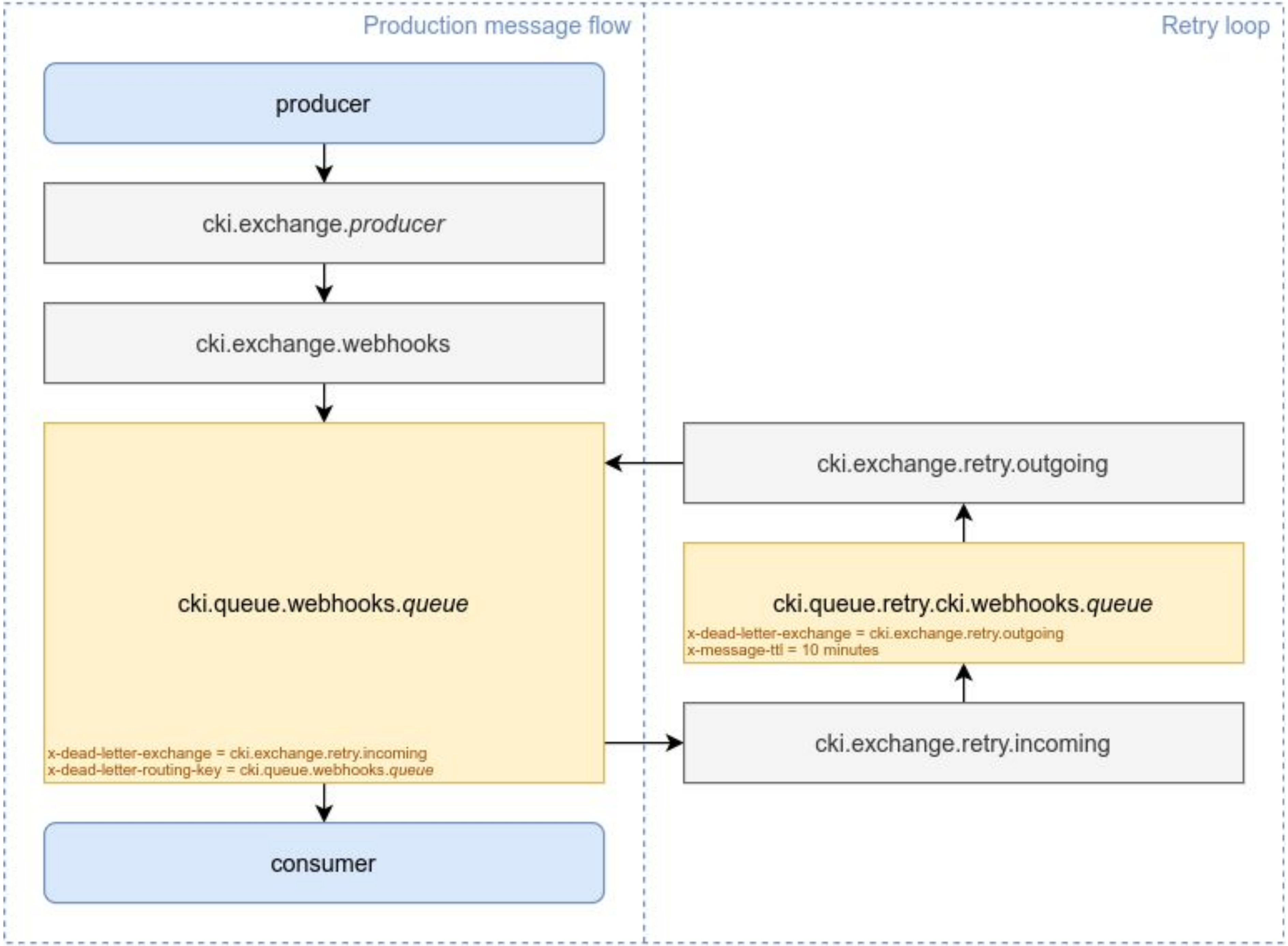
automated recovery: HTTP

- Python:
  - common Python requests session setup
  - takes care of certificate setup and logging as well
- Curl:
  - configure for exitcode!=0 and retries
  - common curl configuration embedded in container images
- random shell code:
  - looping helper to do the right thing
  - keep set errexit enabled in shell functions

automated recovery: longer-lived dependency failures

- goal: retry nearly all failing Python code endlessly
- minimize REST API use and move to AMQP message queues
- automatically reject messages on exception
  - after cooldown, messages will be requeued again
  - can circulate until message successfully handled
- deals with a variety of issues:
  - external dependencies down -> waits until it is up again
  - edge case not considered -> waits until fixed code is deployed


Producers
External messaging infrastructure
CKI messaging infrastructure
CKI consumers
GitLab webhooks
Jira
Sentry
Brew
Bugzilla
Koji
Copr
UMB
Fedora Messaging
UMB virtual queue
Fedora queue
AWS Lambda
AWS Lambda
AWS Lambda
AMQP bridge
AMQP bridge
RabbitMQ cluster
consumer 1
consumer 2


Production message flow
Retry loop
producer
cki.exchange.*producer*
cki.exchange.webhooks
cki.queue.webhooks.*queue*
x-dead-letter-exchange = cki.exchange.retry.incoming
x-dead-letter-routing-key = cki.queue.webhooks.*queue*
consumer
cki.exchange.retry.outgoing
cki.queue.retry.cki.webhooks.*queue*
x-dead-letter-exchange = cki.exchange.retry.outgoing
x-message-ttl = 10 minutes
cki.exchange.retry.incoming

automated/manual recovery: GitLab jobs

- pipeline herder microservice:
  - keeps track of failed GitLab jobs
  - detects common transient errors
  - retries jobs with increasing interval of time
- if GitLab CI runners fail:
  - containerized jobs can run ~anywhere
  - scripted provision helper for Beaker-based fallback runners
  - e.g. site-specific lab outages, mainframes offline, …


🤗 Question time 🤗