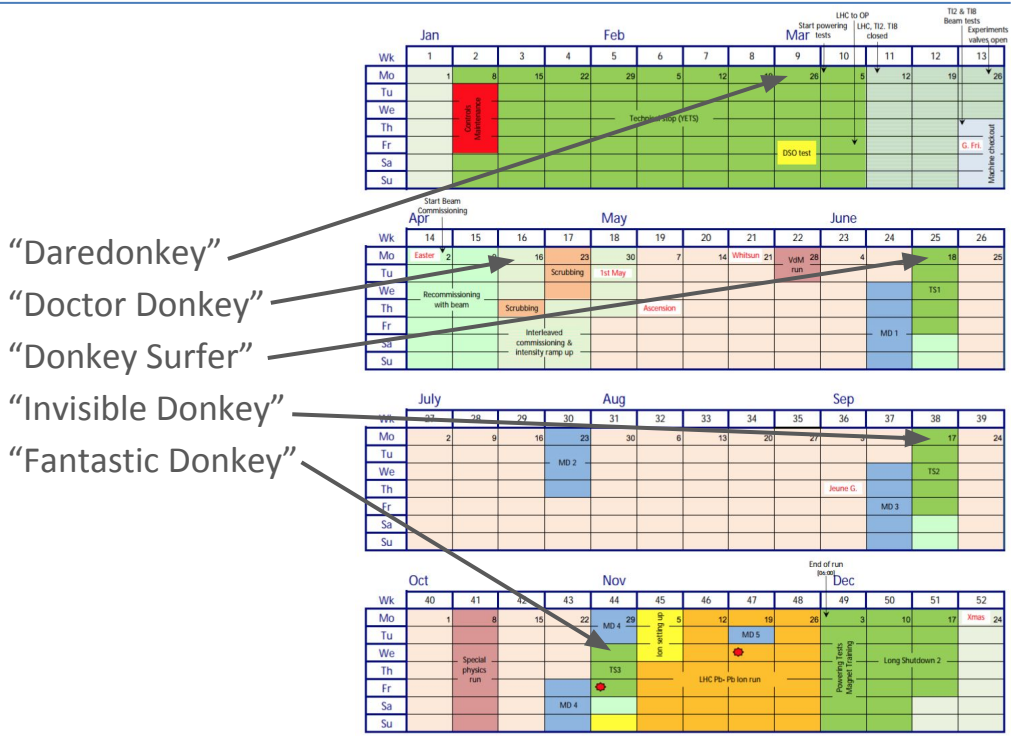
Advanced architecture & development process

Release cycle

- Types of releases
  - Feature releases 1.XX.00: Major features, database schema changes
  - Patch releases 1.15.XX: Bugfixes and small features
  - Hotfix releases 1.15.03.post1: To quickly address a specific issue with a release
  - (Pre releases 1.15.0.pre1: Pre releases for feature releases)
- Three different packages: rucio (core), clients, webui
- New patch release for all 3 packages every 2 weeks
- Feature releases 4-5 times per year (Usually coincide with LHC technical stops)
- Additional clients and webui releases if needed

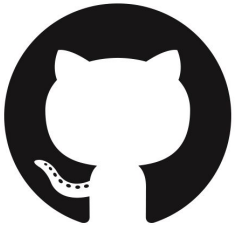
Feature releases 2018

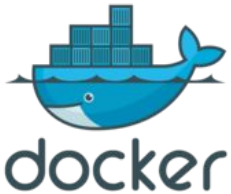
- Feature releases for 2018
  - February 2018 1.15.0
  - April 2018 1.16.0
  - June 2018 - TS1 1.17.0
  - September 2018 - TS2 1.18.0
  - November 2018 - TS3 1.19.0
- 2017
  - 5 feature releases
  - 25 patch releases



How to contribute 1/3

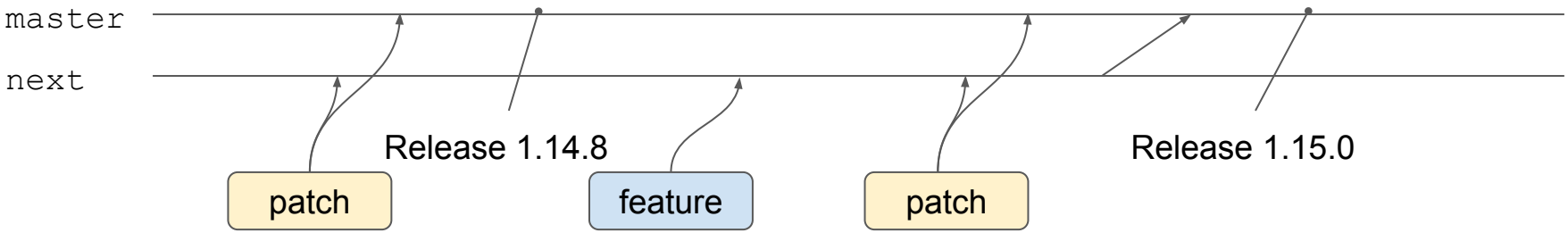
- Join our slack channel to interact with the other developers
- Fork the repository on GitHub
  - Read / Write issues
  - Review / Comment Pull Requests
  - Submit Pull Requests
- Read the documentation on readthedocs
- Download the packages from PyPi
- Download docker images on docker hub
  - Server, daemon, UI, clients, dev
  - Pushed automatically on new release





How to contribute 2/3

- Described in detail in the CONTRIBUTING guide
- If you have identified a contribution, open an issue and describe it
- Two development branches
  - `master` → Holds the development for the next patch release
  - `next` → Holds the development for the next feature release;
- Depending if the contribution is a feature or patch:
  - Feature → Pull request is opened against the `next` branch
  - Patch → Since this is a bugfix it needs to be opened against both the `next` and `master` branch

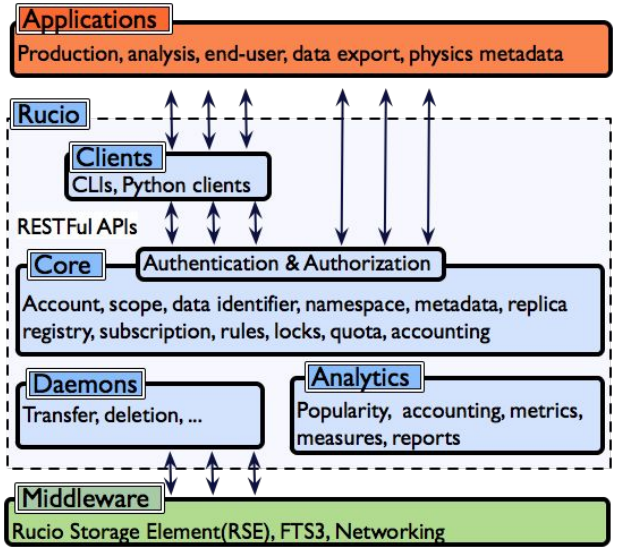


How to contribute 3/3

- Create unit tests for your code
- Document your code
- Run flake8 and pylint to sanitize your code
- Automatic Pull Request testing on Travis
  - Every Pull Request tested against Oracle, MySQL and PostgreSQL
  - Every Pull Request requires a positive test result from travis (400+ unit tests)
- Human review of all Pull Requests
  - Everyone is welcome to make comments
  - Approve / Request changes by the core development team
  - Merging done centrally

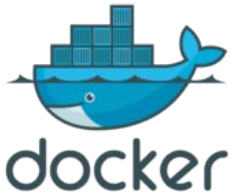
Architecture

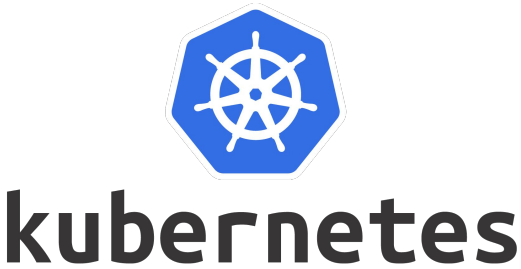
- Clients (Command Line, Python)
- Server
  - RESTful APIs based on WSGI containers, HTTPs
  - Token-based authentication
- Daemons
  - Asynchronous orchestration of all system tasks
  - Transfers, Rules, Deletion, Dataset deletion, Messages Rebalancing, Pre-Placement, Consistency, Recovery, …
  - Lightweight, thread-safe, horizontally scalable
- Database
  - Object Relational Mapper (ORM) ⟶ Oracle, PostgreSQL, MySQL, MariaDB, sqlite
- Middleware
  - Storage protocols, FTS3



How to deploy Rucio

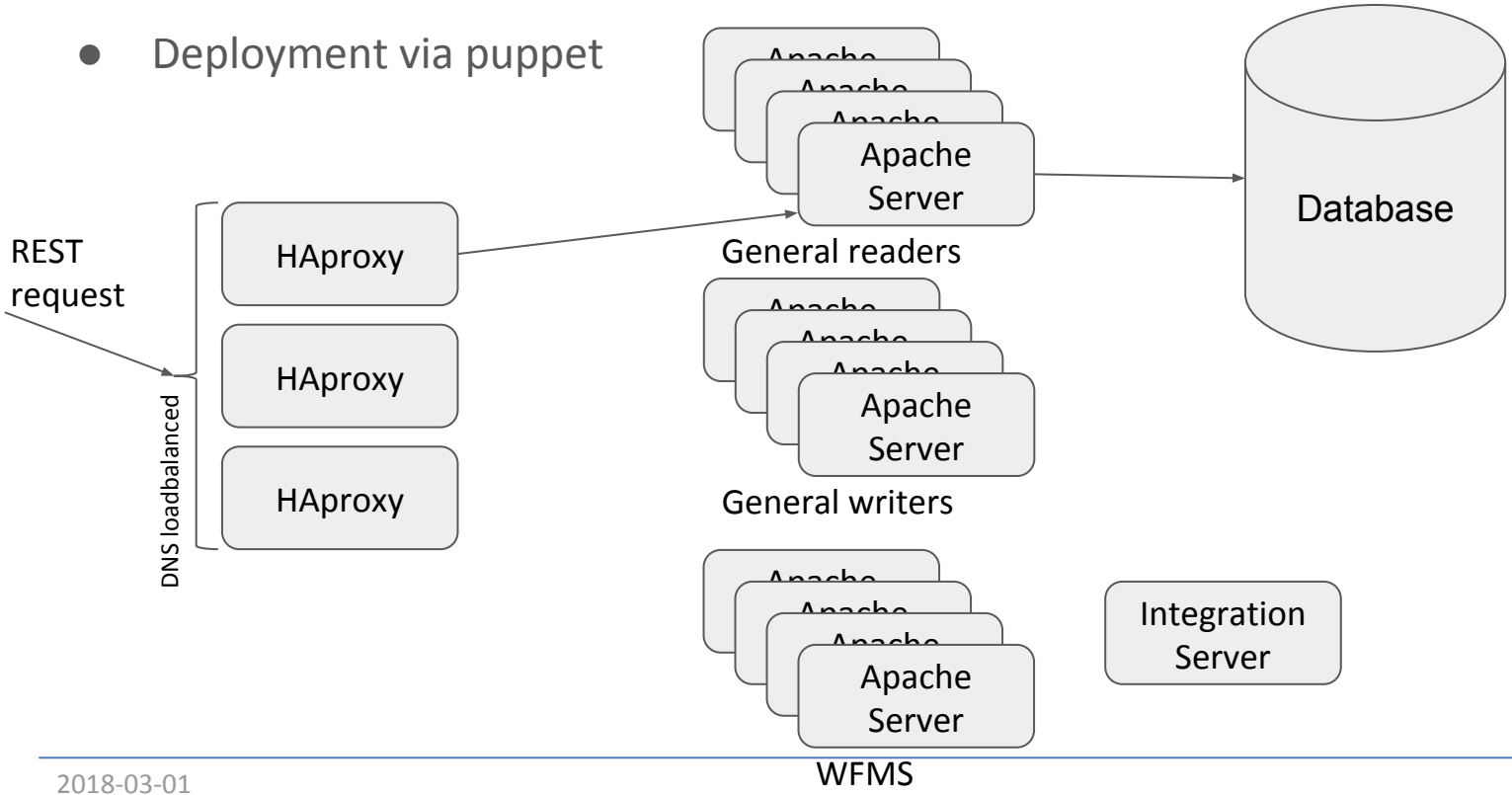
- Installation via PyPi
- Puppet
  - Templates for servers / daemons
- Docker
  - Containers for server / daemons
- Kubernetes
  - In preparation





ATLAS deployment

- Deployment via puppet



Server

- Two types of servers
  - Auth: Provide an authentication token to users based on their credentials (certificate, kerberos, …)
  - Server: Answer REST calls (authentication token required)
- Apache server runs multiple WSGI container which integrate the Rucio server code
- Each RESTFul request goes through different layers in the server
  - The auth token of the REST request is checked first
  - On success the request is handled in the `rest` layer of Rucio
    - Decoding of parameters, jsons, etc. preparation and execution of the actual API call
  - The `api` layer does all the permission checks with the loaded permission module
    - Minor input data transformations might be done
  - The `core` layer holds the actual business logic of Rucio, it processes the input data from the api layer; No permission checks are done in this layer; The daemons also use the core methods directly!
- REST request always follows the path: `rest` → `api` → `core`

Daemons 1/2

- Rucio consists of a (long) list of (partly optional) daemons
- All daemons fetch some kind of requests from the database, process them and perform state changes on the database
  - They do this mostly with the same set of `core` functions like the servers

- `auditor`
  - Consistency checks
- `bb8`
  - Automatic data rebalancing
- `c3po`
  - Pre-placement of popular data
- `cache`
  - Retrieves cache information to synchronize catalog

- `conveyor`
  - Processes rucio internal transfer requests and invokes transfers
  - `submitter`
    - Submits transfers to FTS3
  - `poller / receiver`
    - Consumes FTS3 messages
  - `finisher`
    - Updates replicas and rules
  - `throttler`
    - Throttles and releases transfers

Daemons 2/2

- `hermes`
  - Messaging daemon (STOMP & eMail)
- `kronos`
  - Processes traces and updates dids
- `judge`
  - `cleaner`
    - Removes expired rules
  - `evaluator`
    - Evaluates rules of changed dids
  - `repairer`
    - Repairs stuck rules
  - `injector`
    - Asynchronously injects rules
- `necromancer`
  - Recover lost replicas
- `reaper`
  - Deletes eligible replicas
- `transmogrifier`
  - Evaluates new dids and creates rules based on subscriptions
- `undertaker`
  - Deletes expired dids

Partitioning & Heartbeats

- In order to cope with the load in larger systems, we use a partition principle, based on heartbeats, in all daemons
- All daemons threads regularly send heartbeats to the server
  - This reports their health status and also informs them about their worker# and the amount of total workers
- Based on the worker number and the number of total workers it selects a distinct partition of the workload to process. (Based on hashing)
  - Daemons adapt automatically when more daemons (of their type) are started or removed
  - If no heartbeat is sent for some time (crash) the other daemons adapt their partitions
- Advantage:
  - Largely minimizes row-lock contention on the database
  - Workload elastically manageable; automatic failover