Lab SAD2022
Queue Service on a PaaS

Motivation

- Cloud services should be designed combining microservices
- Interaction among microservices should be reactive
  - Events as messages sent among microservices
- Queues are used to store messages by microservices
  - To be consumed later on by other microservices
- A well understood pattern
  - Kafka
  - NATS
  - Artemis
  - …
- Example application: FaaS

Queues

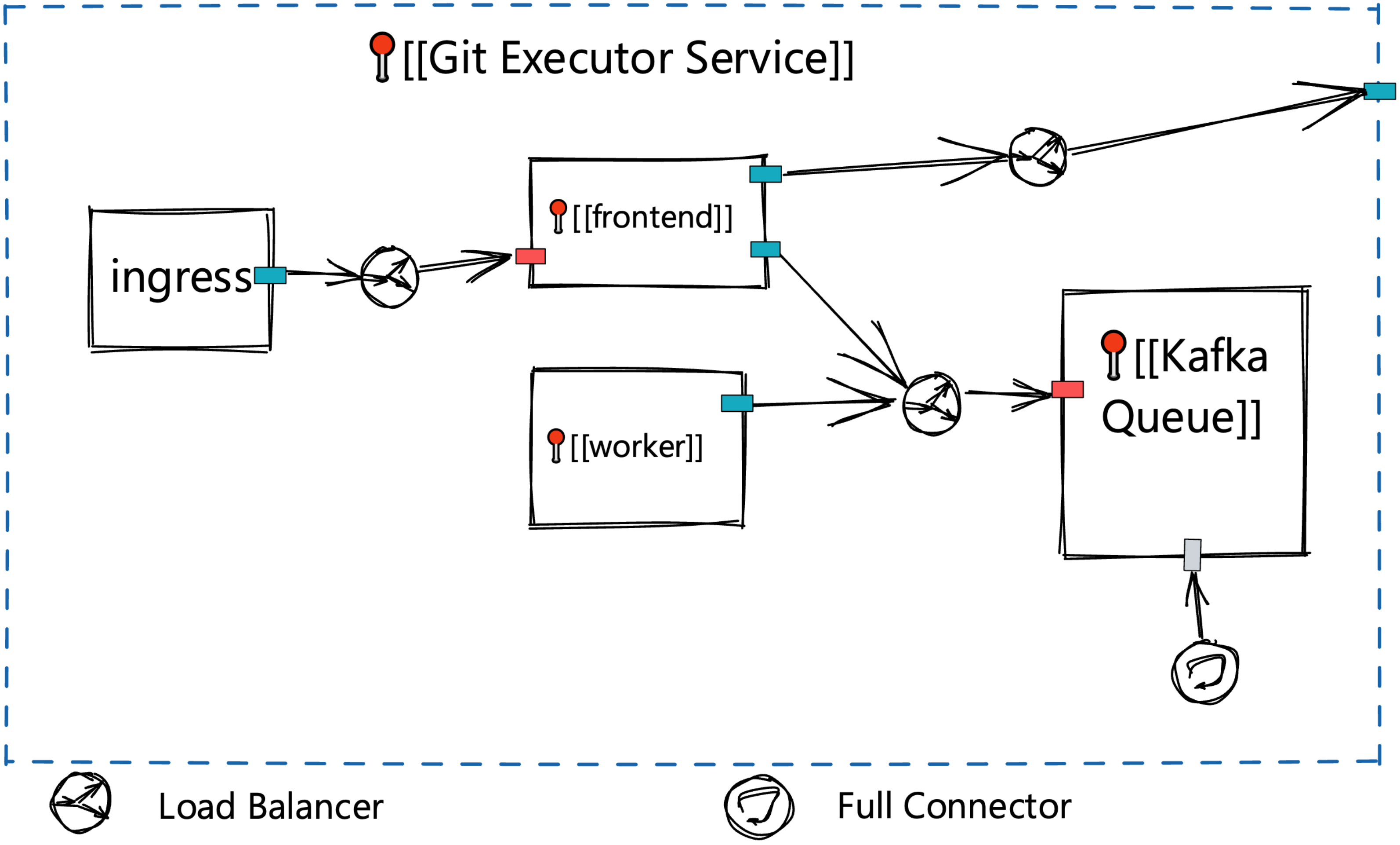
- Generic PUB/SUB pattern
  - Events/Messages are published to a queue by PRODUCERS
    - Messages have a topic
    - CONSUMERS express interest on one or various topics
  - Variations on how many times a message is delivered to consumers
    1. At most once (no repeat, demonstrably best effort when no failures occur)
    2. At least once
    3. Exactly once
- To increase reliability the queue process ought to be replicated
  - Also for scalability

FaaS

- Also called serverless
- In contraposition to SaaS: no state is maintained after a computation is requested
  - Fire and forget:
    - Request execution of a function
    - Get the results
- Implemented as a combination
  - Frontend/adapter for the FaaS API (typically REST)
  - Queue: persist the job orders until the function executor is ready to run it
  - Job Workers, potentially specialized, capable of running the job orders and produce results

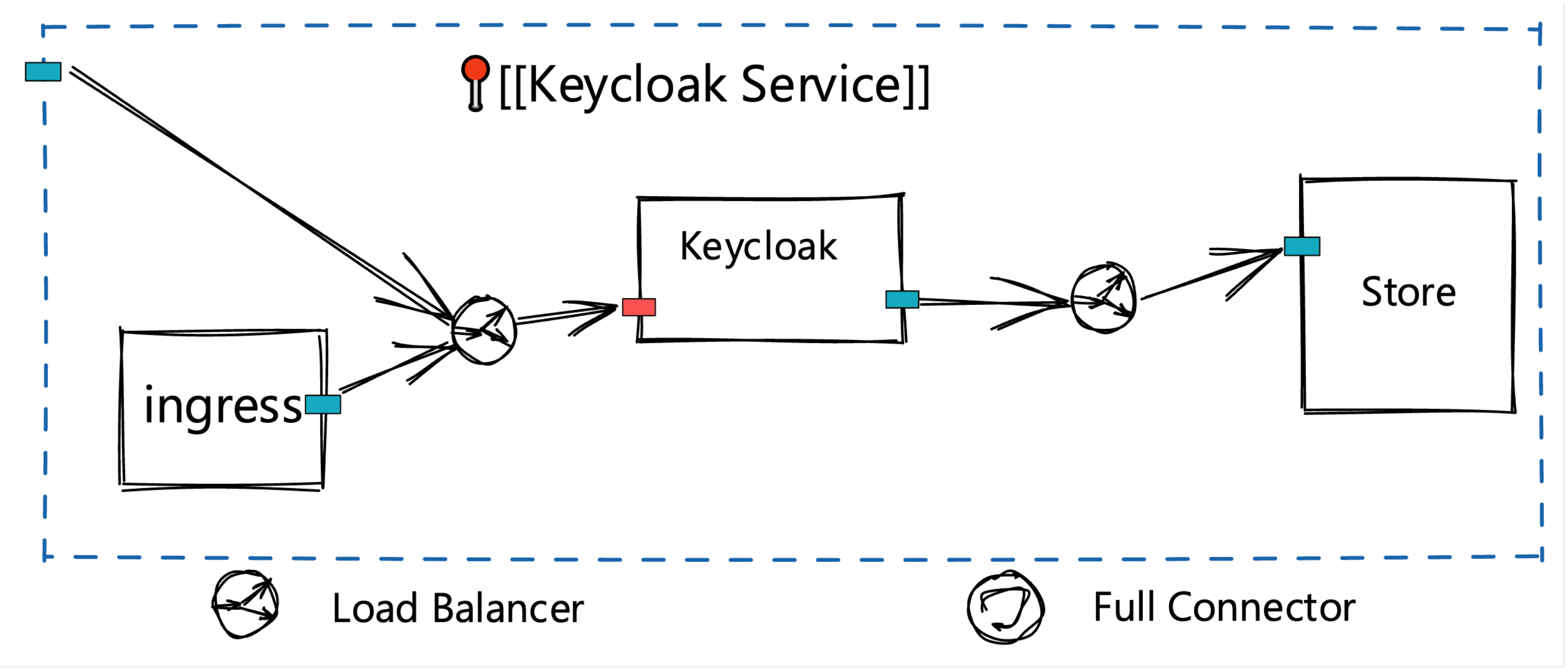
FaaS: Git Executor workers

- The complete service will have at least three microservices
  - Frontend
    - Receives requests from clients
    - Requests are REST, with json payload
    - Prepares them to be placed on the queue
  - Worker
    - Picks up work orders from a queue, carries out the job, and places the result back to another queue
  - Queue
    - Implements the queue functionality
    - Can coordinate with other copies of itself


[[Git Executor Service]]
[[frontend]]
ingress
[[worker]]
[[Kafka Queue]]
Load Balancer
Full Connector

FaaS: Letting Jobs in

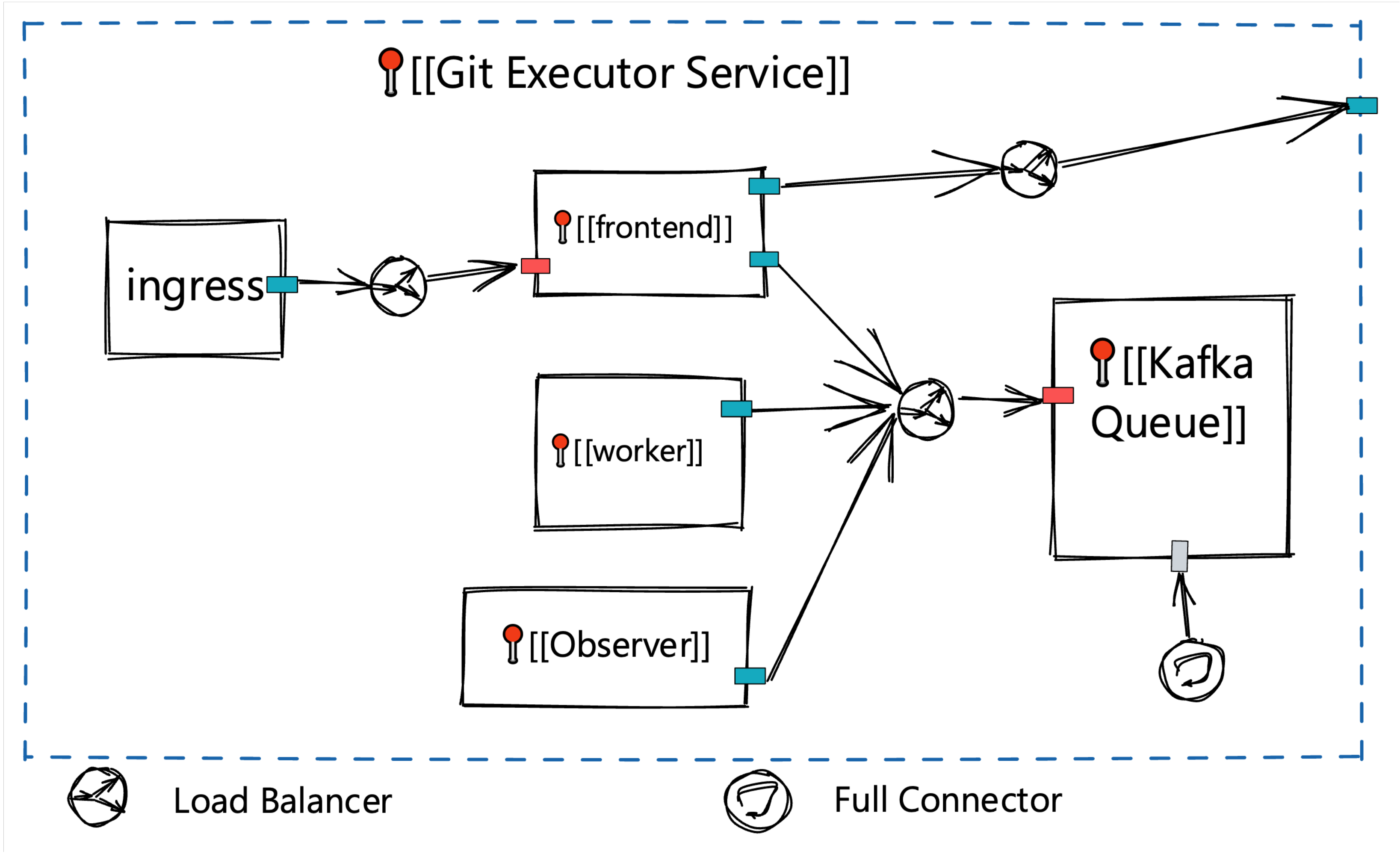
- The FrontEnd must protect the service
- Should only let in those requests that are authenticated
  - Needs to trust an authenticator service
  - Implemented with Keycloak



FaaS: Observer

- Microservice that inspects the queues
  - Job queue
  - Responses queue
- Propose an algorithm for elasticity
  - Based on queue length/response times
    - Configured max rate/response time
  - Determines when a new worker should be added
  - Determines when a worker must be removed
  - Sends its decisions to another queue in the queue server
    - Observed rate of arrival
    - Observed rate of completion
    - Response times

Basic Architecture with Observer



Components/Microservices

- Three components (at least)
  - Frontend, with two “channels”
    - entrypoint
      - □ Suggestion: Implement with express or similar
        - □ To easily handle REST requests
        - □ To prepare to serve SPA client app
    - Queueclient
      - □ Protocol depends on Queue technology
  - Executor
    - Queueclient
  - Kafka Queue
    - discovery, coordination
  - Observer (Optional)

Implementation: Frontend

- Rest server (http)
  - Protected with Bearer Token
    - With claims about identity
- API
  - Send job/return job ID
    - Frontend injects identity of request (from bearer token)
  - Question status of job ID
  - Get result of JOB id
    - With elapsed time
      - □ Erases Job from queue (WARNING: observer)
  - Get list of own jobs submitted
  - Set parameters for observer (optional)
  - Subscribe to changes in jobs via websockets (optional)

Implementation: Executor

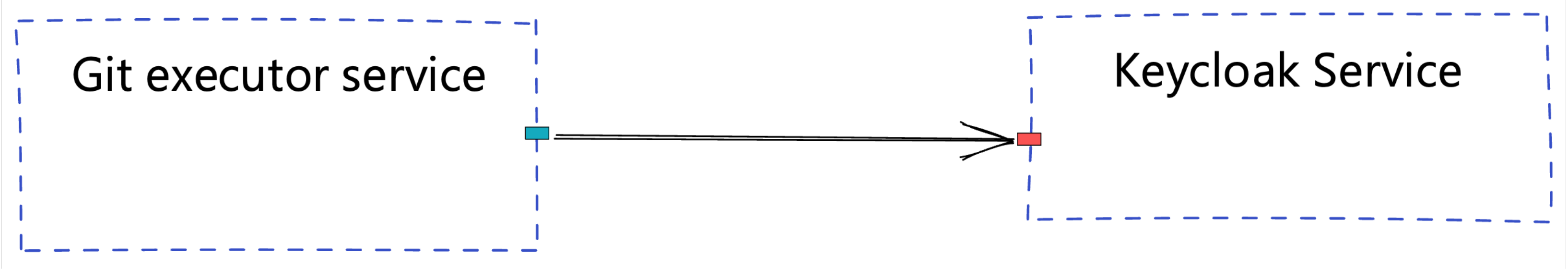
- Picks up Jobs from job queue
- Job Description:
  - GIT REPO
  - Data source links
  - Data
  - Data sink links
  - Credentials needed
    - Should be sent encrypted
    - Encryption/decription key passed in environment
  - Proceeds by cloning repo and then
    - Executing a "main" function passing data/credential parameters
    - Sending "results" to a results queue
    - Tagging result with elapsed time

Implementation: Observer

- Reads the Job Queue
- Reads the results queue
- Execs its algorithm
  - Determines info
    - Average rate of arrival
    - Average rate of service
    - Average response time
    - Add/remove/nothing worker decission
- Reports back on the observer queue

Deployment

- Each microservice's code will be encapsulated within its own docker image
- Two services linked



Deployment

- START with two docker-compose
  - One for Faas, another for keycloak
- THEN, model it within a specific PaaS:
  - https://docs.kumori.systems
  - Two services: FaaS/Keycloak
    - Choose any store for keycloak
    - May be in the same container
    - The two services modeled as service applications
    - The earlier images provide the architecture of the service apps.

Deliverables (yours)

- Paper, describing the architecture and functioning of the service
  - Extras you have undertaken must be included
  - Extras you think may be worth exploring
- Code
  - Folder per microservice
    - Including the Dockerfile, if needed
    - Including docker-compose files
    - Including KPaaS model manifests
  - Folder with service app manifest per service
  - Test suite used to verify the properties of your service (as described in the paper)
- In git.upv.com
  - Group: SAD2122, under SAD. You will be assigned a user per group
- DEADLINE: Jan 31

Grading

- Base:
  - Spec for job/response structure
  - Keycloak configuration
  - Implementations frontend/worker
  - Docker-compose files
  - Model manifests
  - Well-written paper (Spanish/english) explaining the work
- Test code with good coverage
- EXTRA:
  - Observer

Resources

- Virtual Machine in portal-ng.dsic.upv.es
  - To use docker
  - To carry out initial experiments with docker-compose
  - To use development environment of KPaaS
- Access to PaaS setup to experiment
  - Will be provided later on
- KPaaS documentation: https://docs.kumori.systems
- Use one of these queue systems:
  - Kafka: https://kafka.apache.org/
- NOTE: as much as possible, try to use the official/available docker images

Platform

- URL: admission-forge.vera.kumori.cloud
- User: your student username preceded with SAD_
  - Password…TBD

Brief introduction to Kumori PaaS

- Built on top of Kubernetes
  - Does not expose Kubernetes concepts
- Manages deployment of service applications
  - Built according to the Kumori Service Model
  - Using the Platform tools to deploy
    - Which, in turn, exercise the platform's API

Kumori’s Service Model

- Set of concepts and a language to express them
- Main concepts:
  - Component
  - Service Application
- Component == the “program” of a microservice.
- Service Application == the “program” of a complete complex service
- In both cases
  - Specification of a service

Kumori's Service Model

- Specification of a service
  - Common to components and service apps
  - Communication channels
    - Server
    - Client
    - Duplex
  - Configuration
    - Parameters
      - □ By value
    - Resources
      - □ By reference

Component-specific elements

- Vertical size
  - CPU, Memory, Bandwidth
- Code
  - Docker image
    - Or images, as more than one container can be specified
  - Mappings
    - Relates configuration parameters to elements within a container
      - □ Files
      - □ Directories
      - □ Environment variables
  - Starting command
  - Service discovery convention
    - DNS-based
    - Resolving client channel names

Service-app specific elements

- Role
  - Implemented by a Component
  - Equivalent of a microservice’s deployment
  - Multiple instances of a Role can exist in the deployment of a service.
  - Specifies how the configuration of the application transforms into the configuration of the underlying component
  - Specifies ”horizontal size” properties
    - Instances
    - Resilience
  - Inherits the channels of its component

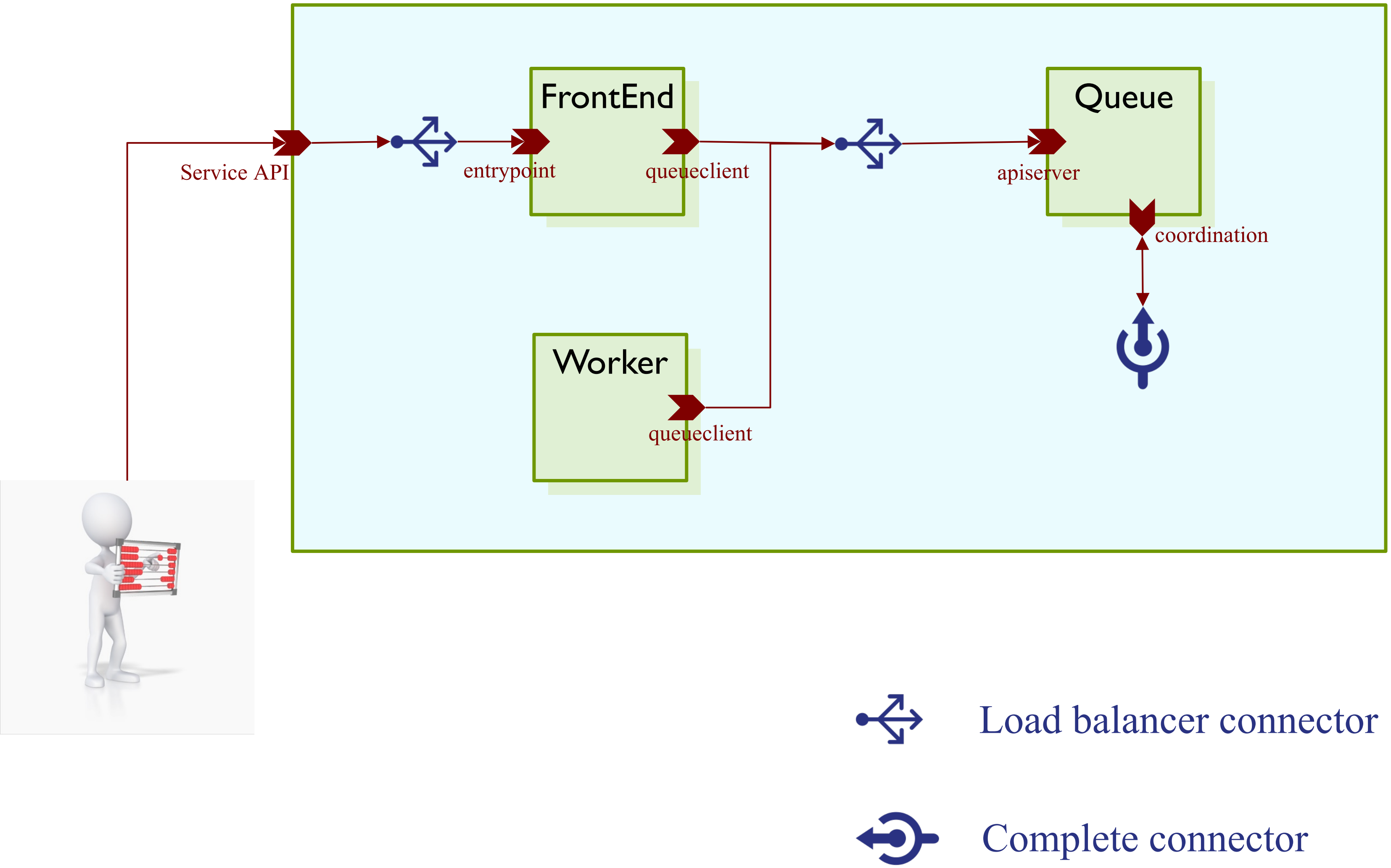
Service-app specific elements

- Connector
  - Intermediates communications between roles
    - Connects client channels to server channels
    - Also interconnects duplex channels
  - Two kinds:
    - LB (load balancer)
      - □ From client to server
      - □ Chooses the instance of the server that receives the connection
      - □ PURPOSE: Client-server patterns
    - Complete/Full
      - □ To duplex
      - □ From client to server
      - □ Exposes to client (or duplex) all instances of server (duplex) channels.
      - □ PURPOSE: Peer to Peer pattern

Service-app specific elements

- Links
  - From client channels in roles to connectors
  - Or
  - From connectors to server (or duplex) channels in roles
  - Only complete connectors can be linked to duplex channels
- The topology of the service is the graph resulting from
  - Roles
  - Connectors
  - Links
- Graphical representation
  - Server channels on the left
  - Client channels on the right
  - Duplex channels at the bottom

Example service application specification



FrontEnd

Queue

Worker

Service API

entrypoint

queueclient

apiserver

coordination

queueclient

Load balancer connector

Complete connector

Formalism

- Components and Service apps speced in CUE
  - https://cuelang.org
  - Gaining adoption within the devops ecosystem
- Syntactically a superset of JSON
  - All JSON is syntactically CUE
- Logical language with two main operations
  - Disjunction
  - Unification
- Useful built-in operators

Tooling

- An instance of the platform can be
  - A cluster in a private CPD
    - Physical
    - Virtual
    - Hybrid
  - A set of VMs in an IaaS provider
  - A mixed environment
- kam
  - CLI tool used in development and deployment
    - Interacting with instances of the platform
    - Needs Nodejs installed.
    - Installed via npm: npm install –g @kumori/kam
    - Expect to update frequently
  - General format
    - kam <subcommand> <options and arguments>

Tooling: kam

- Authoring kpaas modules
  - kam mod init <module name>
  - kam mod dependency <dependent module name>
- Deploying services
  - kam ctl deploy <deployment manifest path>
- Setting up a context cluster
  - kam ctl config –a <admission url for cluster>
  - kam ctl login <username>
- Obtaining list of registered resources
  - kam ctl get resources
  - kam ctl get resource <resource name>
  - kam ctl register resource <resource name> -d <data>
  - …