Strategy and Tools to Test Software in the SKA Project: The CSP.LMC Case

Gianluca Marotta, Elisabetta Giani, Ivana Novak, Martino Colciago, Giorgio Brajnik and Carlo Baffa

What do we want to achieve?

How to make our software component 100% reliable?

What do we want to achieve?

How to make our software component ~~100% reliable~~?

Maybe it's impossible…

Better questions are:

What do we want to achieve?

How to make our software component ~~100% reliable~~?

Maybe it's impossible…

Better questions are:

- How to make our software component as reliable as possible?

What do we want to achieve?

How to make our software component ~~100% reliable~~?

Maybe it's impossible…

Better questions are:

- How to make our software component as reliable as possible?

- How to quantify the reliability of our software component?

Outline

- *What* are we testing ?
  - The SKA telescope
  - The CSP-Local Monitoring and Control
  - The CSP.LMC and its environment
- *How* are we testing it?
  - Testing SKA Software
  - Unit / Component / Integration
  - Code structure
  - Fault Conditions Analysis
- *When* and *where* are we testing it?
  - CI/CD pipeline
- Improve and quantify “reliability”
  - Data mining on test results

The SKA telescope

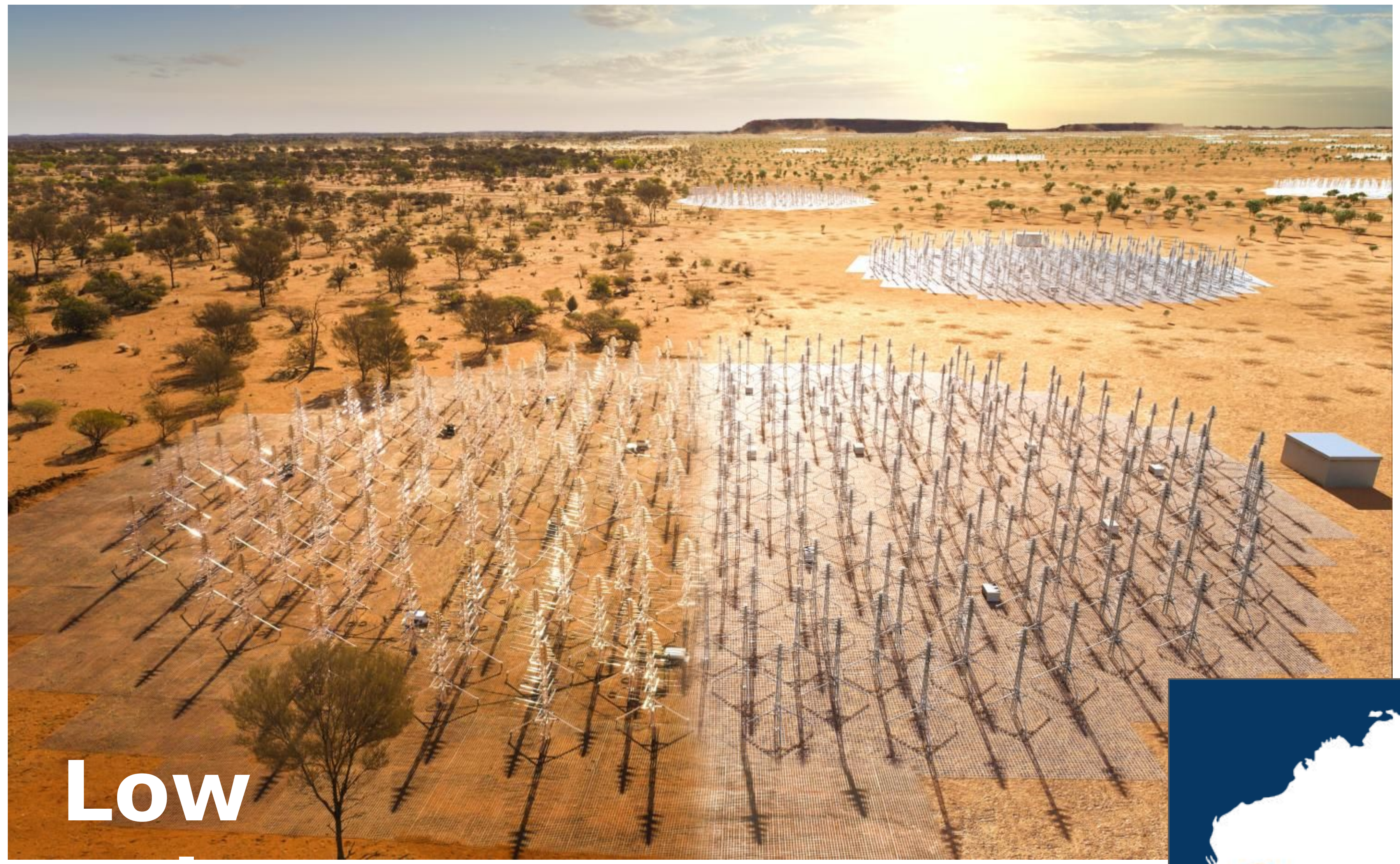
What are we testing?

The *Square Kilometer Array* (SKA) is an international effort to construct the *world's biggest radio telescope.*

Location: South Africa

350 Mhz to 15.3 GHz

197 dishes - max baseline: 150km[1]



Location: Australia

50 MHz to 350 GHz

131000 antennas- max baseline: ~65km[1]

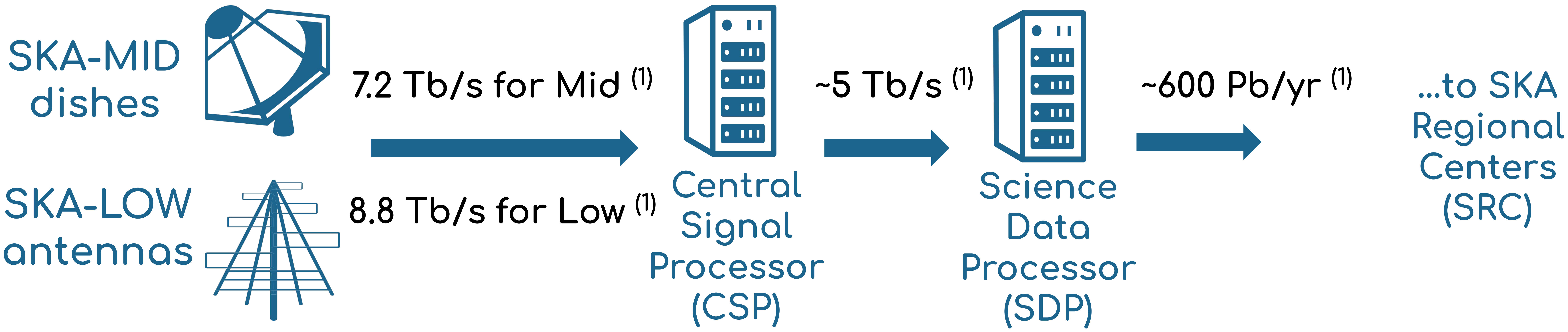
[1] Data for SKA1 implementation

Ref: skao.int

The SKA telescope

What **are we testing?**

SKA will produce a *huge amount of data*



- The purpose of *CSP* is to correlate, filter and make a preliminary analysis
- *SDP* makes further data reduction
- *SRC* stores data and made them available for scientific analysis

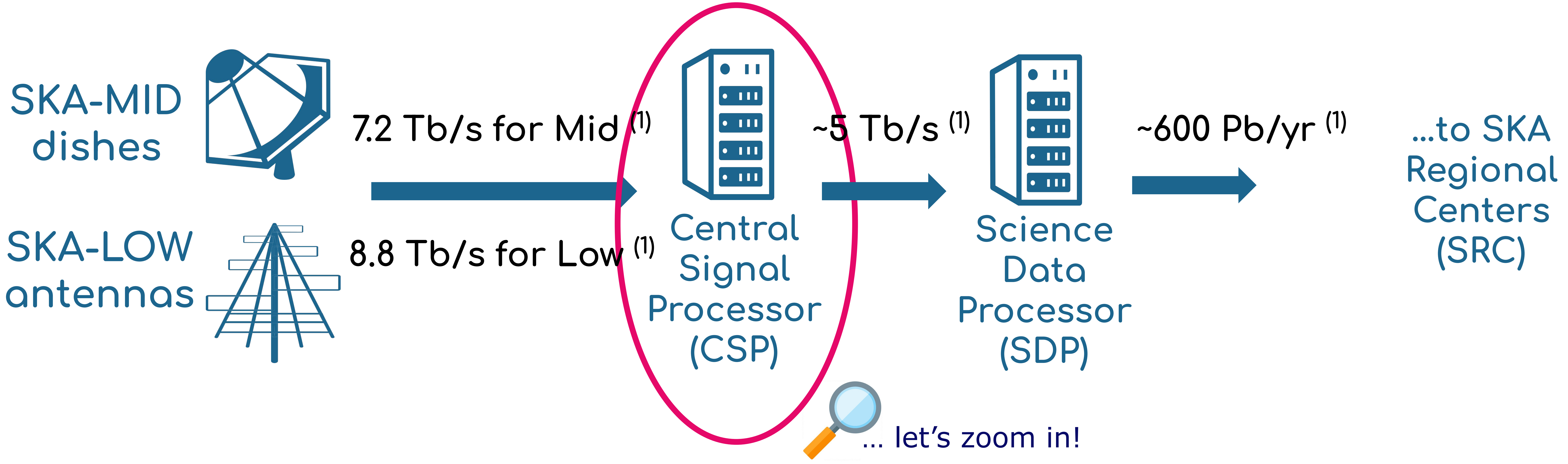
(1) Data for SKA1 implementation

Ref: skao.int

The SKA telescope

***What* are we testing?**

SKA will produce a *huge amount of data*

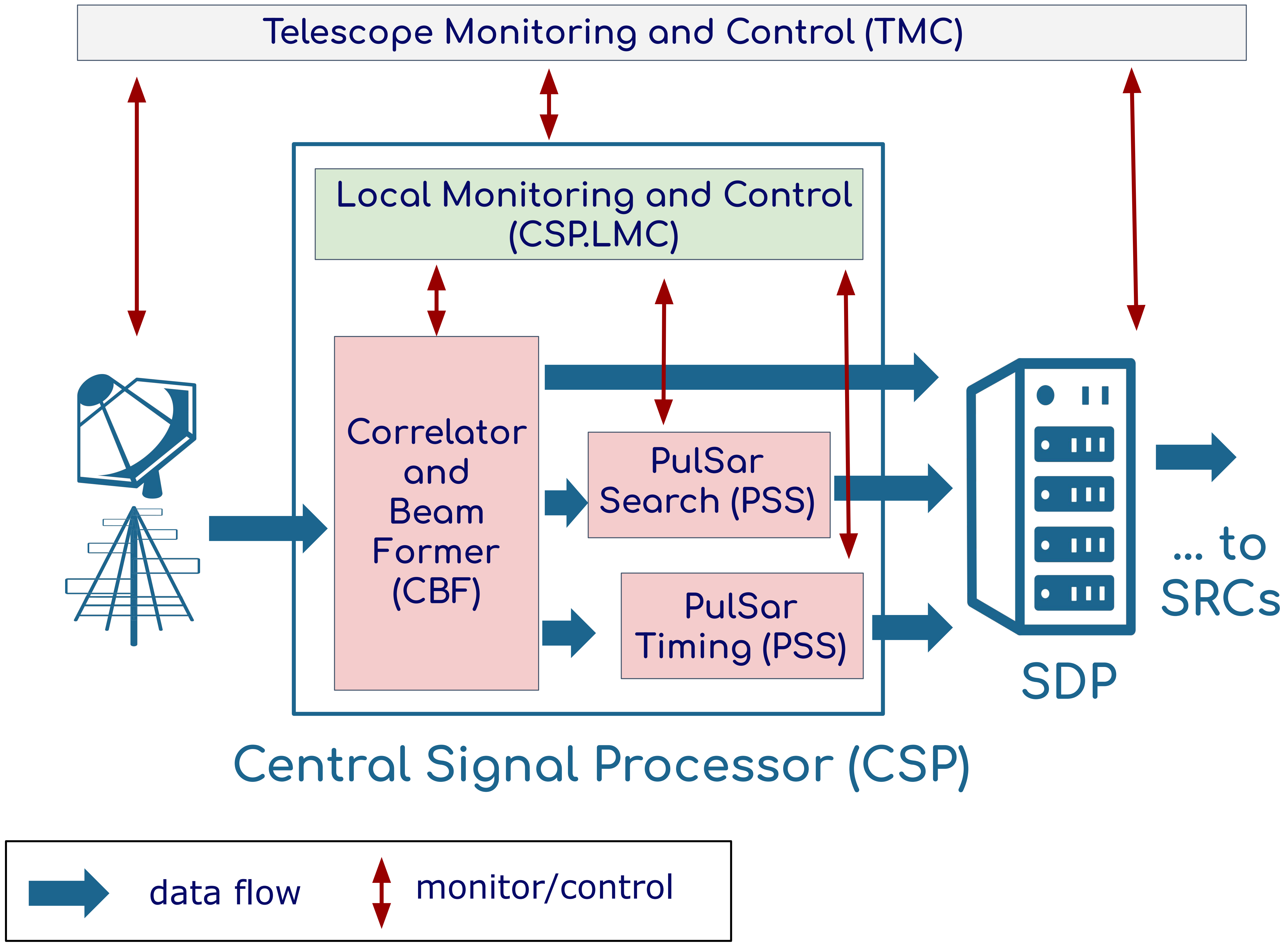


- The purpose of *CSP* is to correlate, filter and make a preliminary analysis
- *SDP* makes further data reduction
- *SRC* stores data and made them available for scientific analysis

(1) Data for SKA1 implementation

Ref: skao.int

The CSP.Local Monitoring and Control

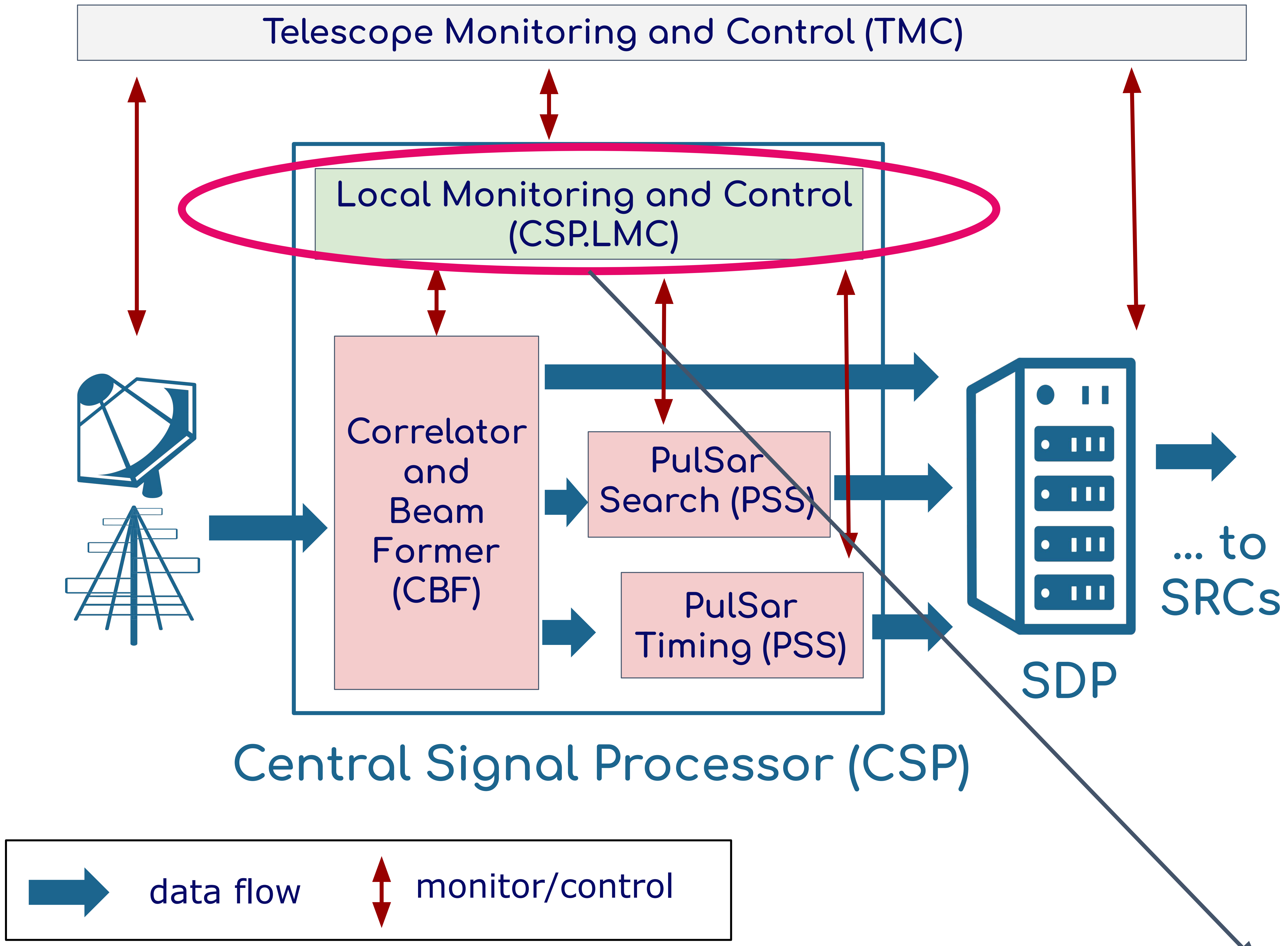


CSP is composed of 4 main subsystems:

- 3 for data reduction (CBF, PSS, PST);
- 1 for monitoring/control (CSP.LMC)

CSP.LMC provides the *interface* to TMC *without exposing CSP internal complexity.*

The CSP.Local Monitoring and Control



CSP is composed of 4 main subsystems:

- 3 for data reduction (CBF, PSS, PST);
- 1 for monitoring/control (CSP.LMC)

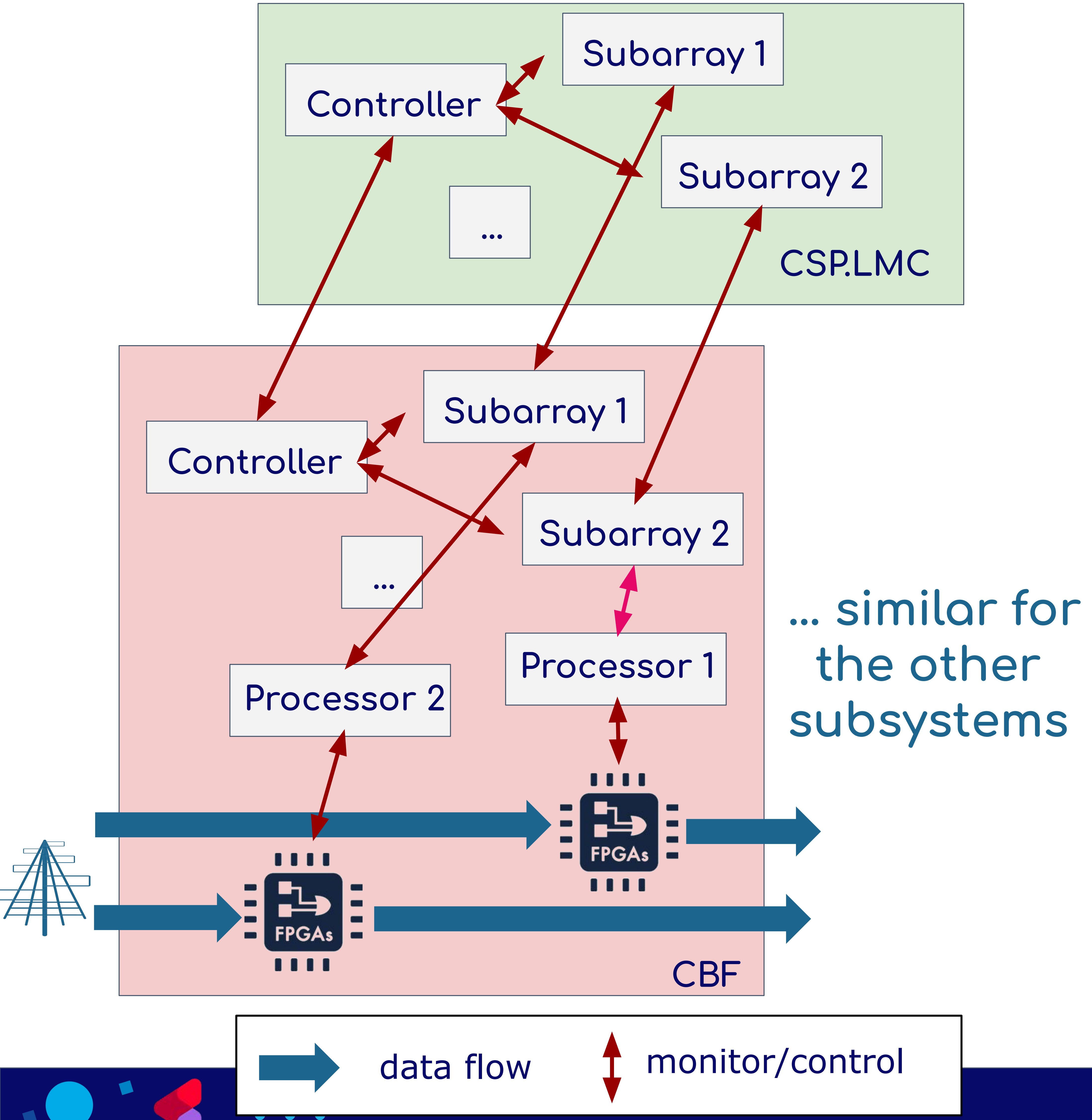
CSP.LMC provides the *interface* to TMC *without exposing CSP internal complexity.*

This is our *System Under Test*!

… let's zoom in! (again)

The CSP.LMC and its environment

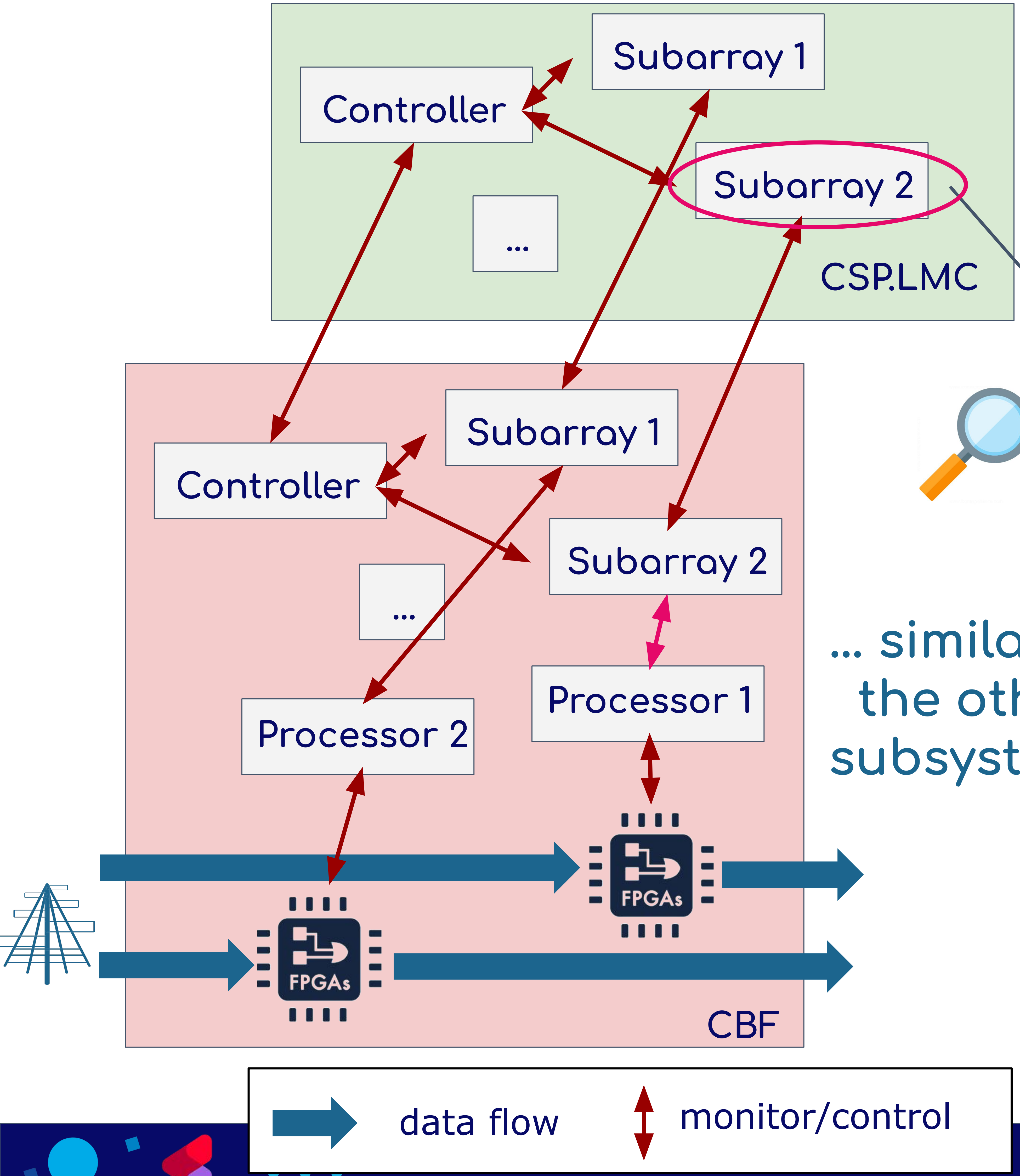
A *very simplified* view of the internal structure…



- A *software component* is a *TANGO Device* written in *Python.*
- Each TANGO Device is containerized and orchestrated with *Kubernetes (k8S)*

The CSP.LMC and its environment

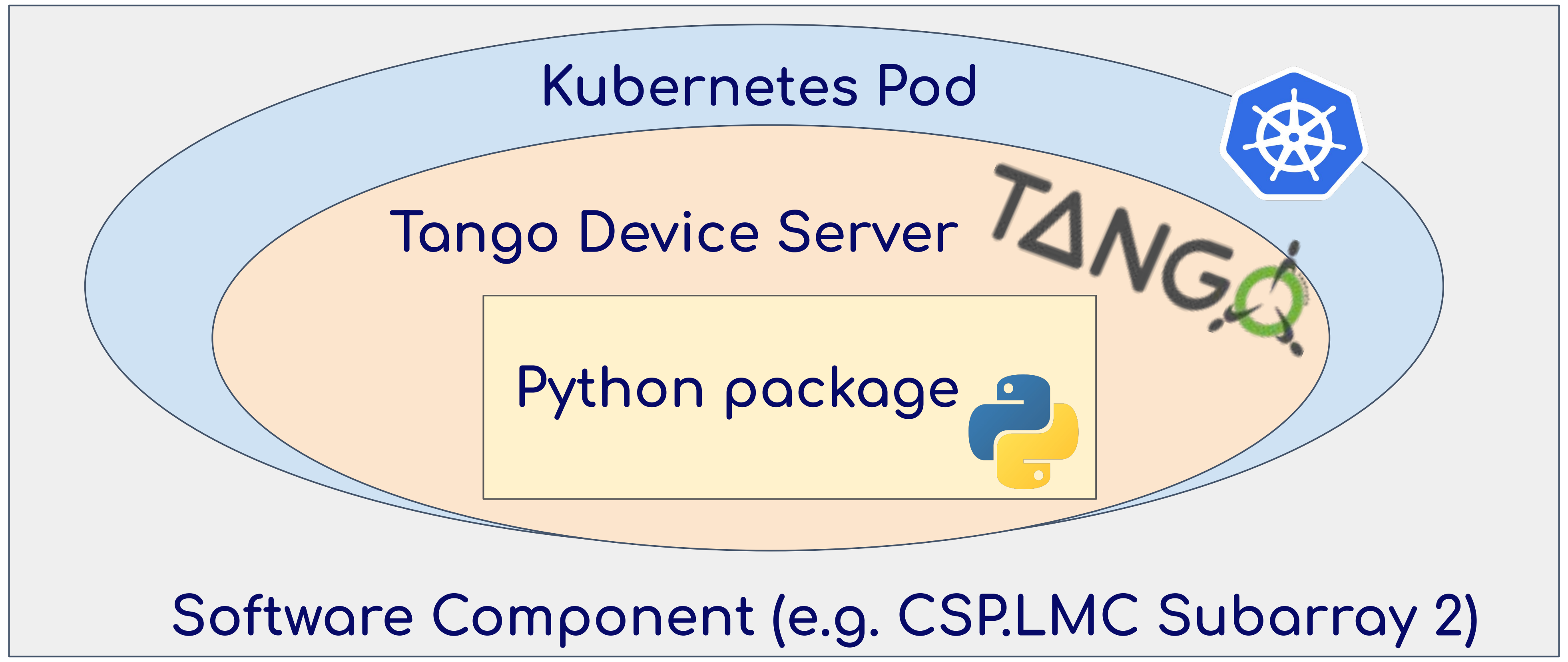
A *very simplified* view of the internal structure…



… similar for the other subsystems

- A *software component* is a *TANGO Device* written in *Python.*
- Each TANGO Device is containerized and orchestrated with *Kubernetes (k8S)*

… let's zoom in! (last time)



Bugs and failures can happen anywhere!

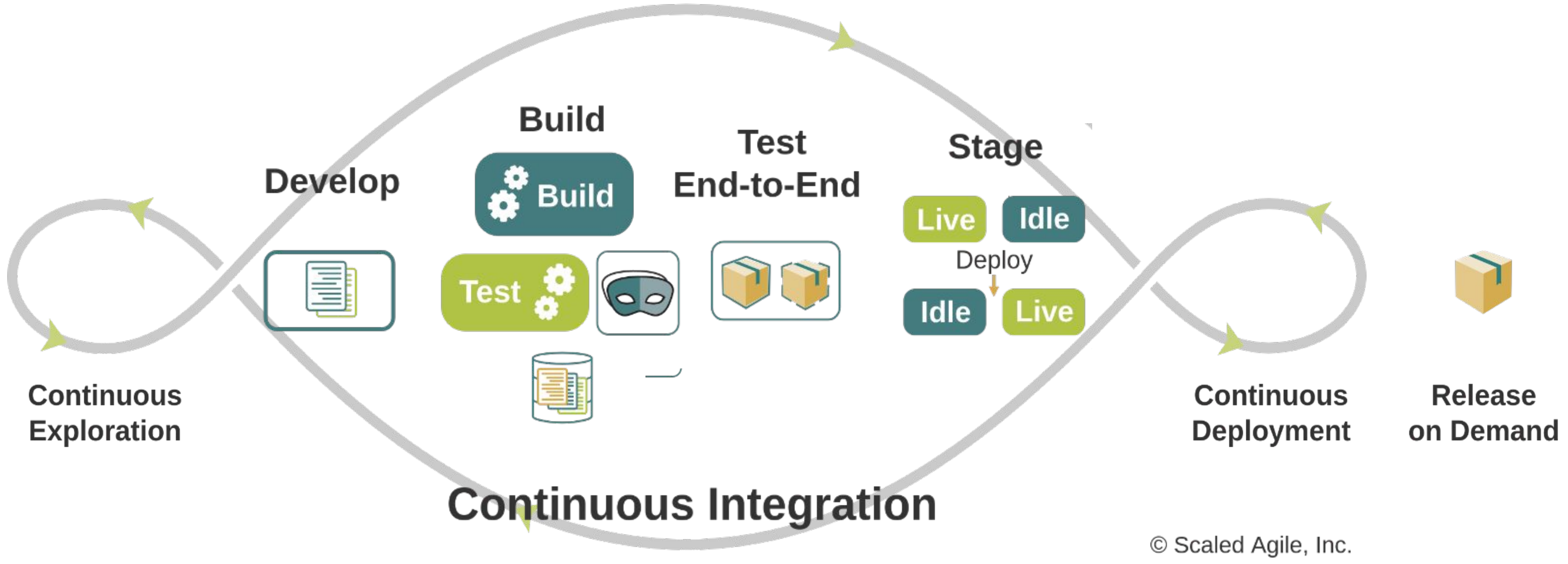
Testing SKA Software

The Software Engineering Group at SKAO is made by more than 100 developers organized into different *Agile Teams*

PROVIDED BY SCALED AGILE

- Individual teams are *responsible* for *a specific software subsystem*, for *its quality and its testing strategy*
- Verification Tests based on requirements are done by AIV[1] teams
- A *Testing Community of Practice* gather developers from different teams to share knowledge and practices

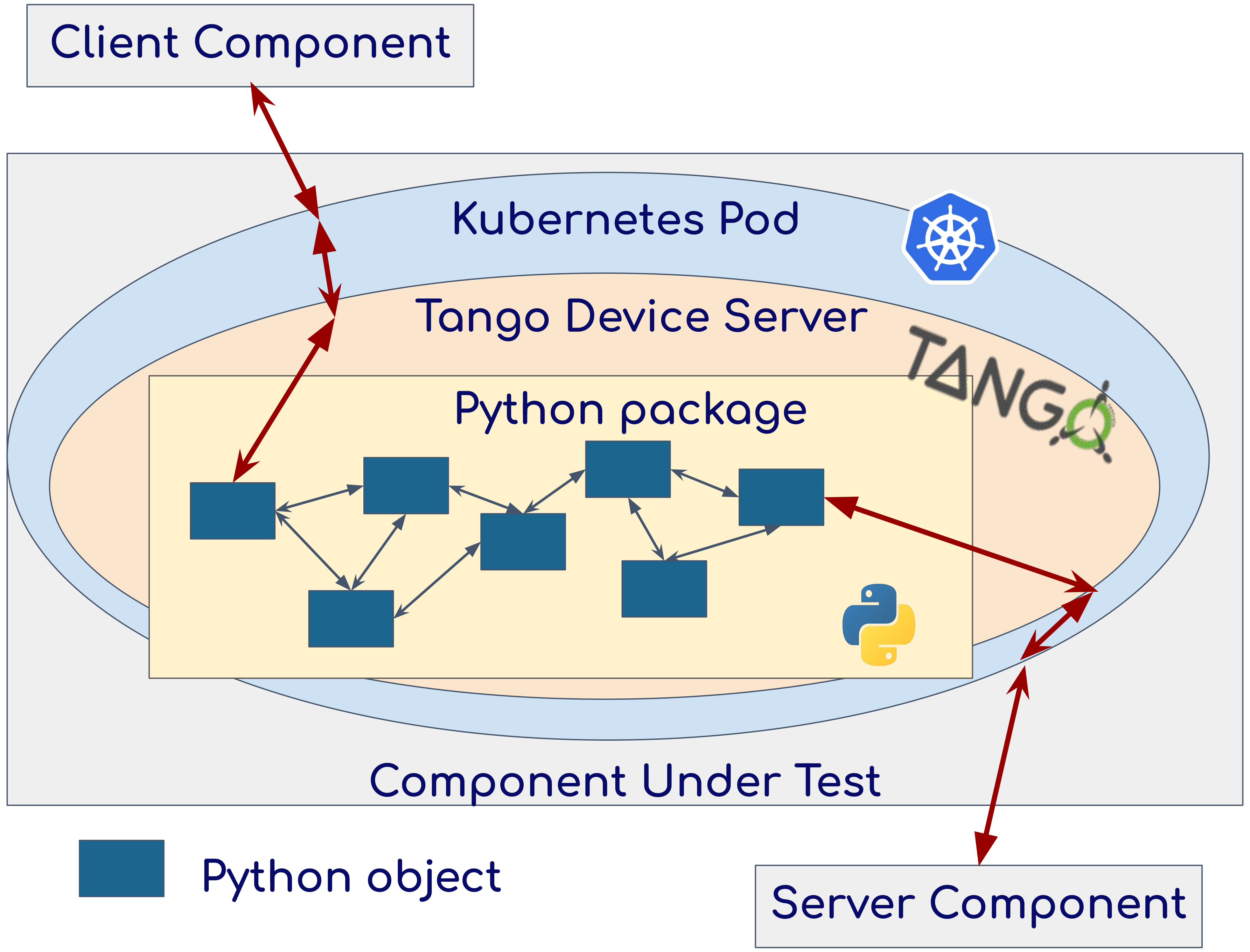
Tests are essential to demonstrate and validate functionalities in the framework of Continuous Integration.



[1] Assembly, Integration and Verification

Unit Tests

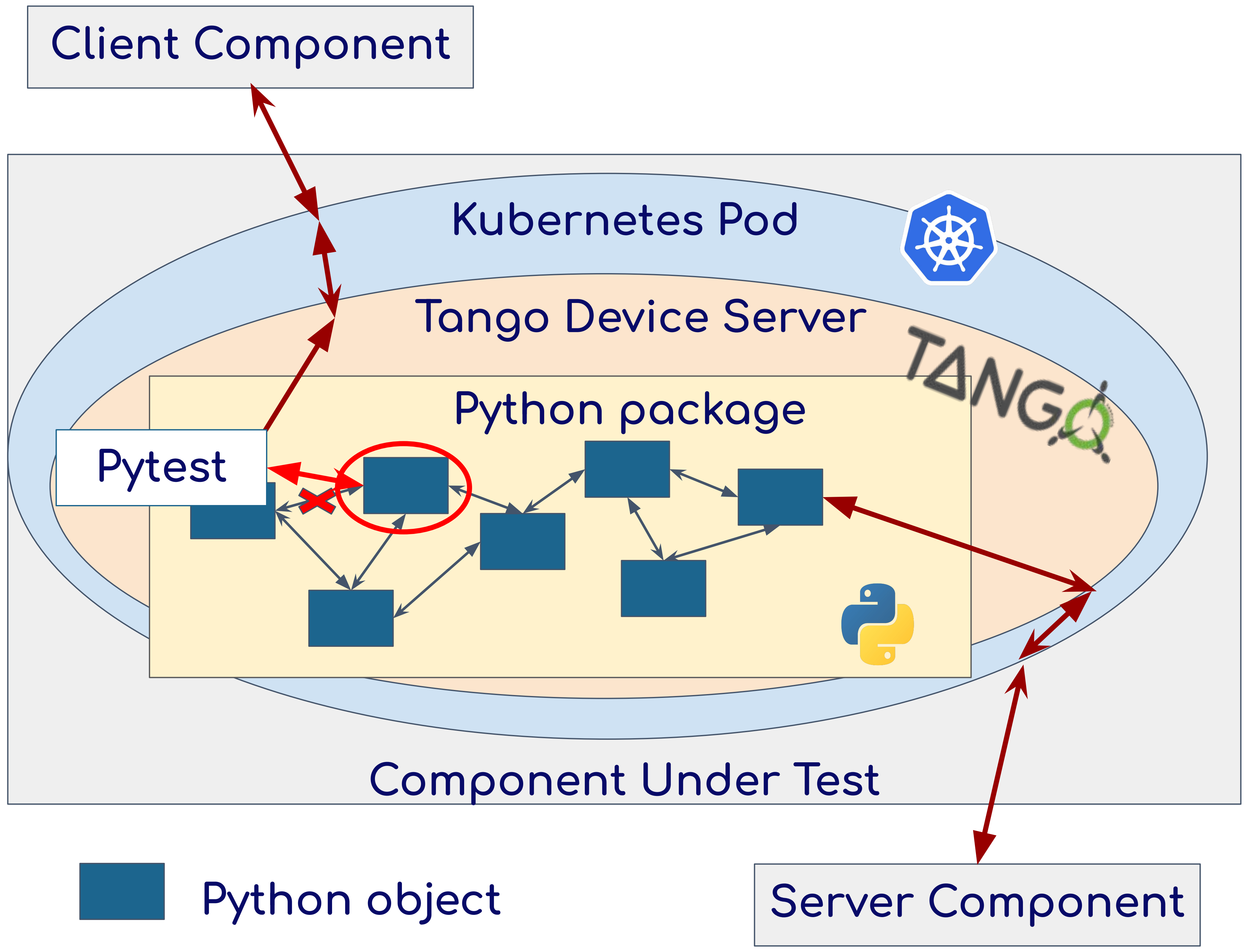
- "The testing of *individual software units* [...] that can be tested in isolation."[1]



[1] From SKAO "Software Testing Policy and Strategy":

Unit Tests

- "The testing of *individual software units* [...] that can be tested in isolation."[(1)]



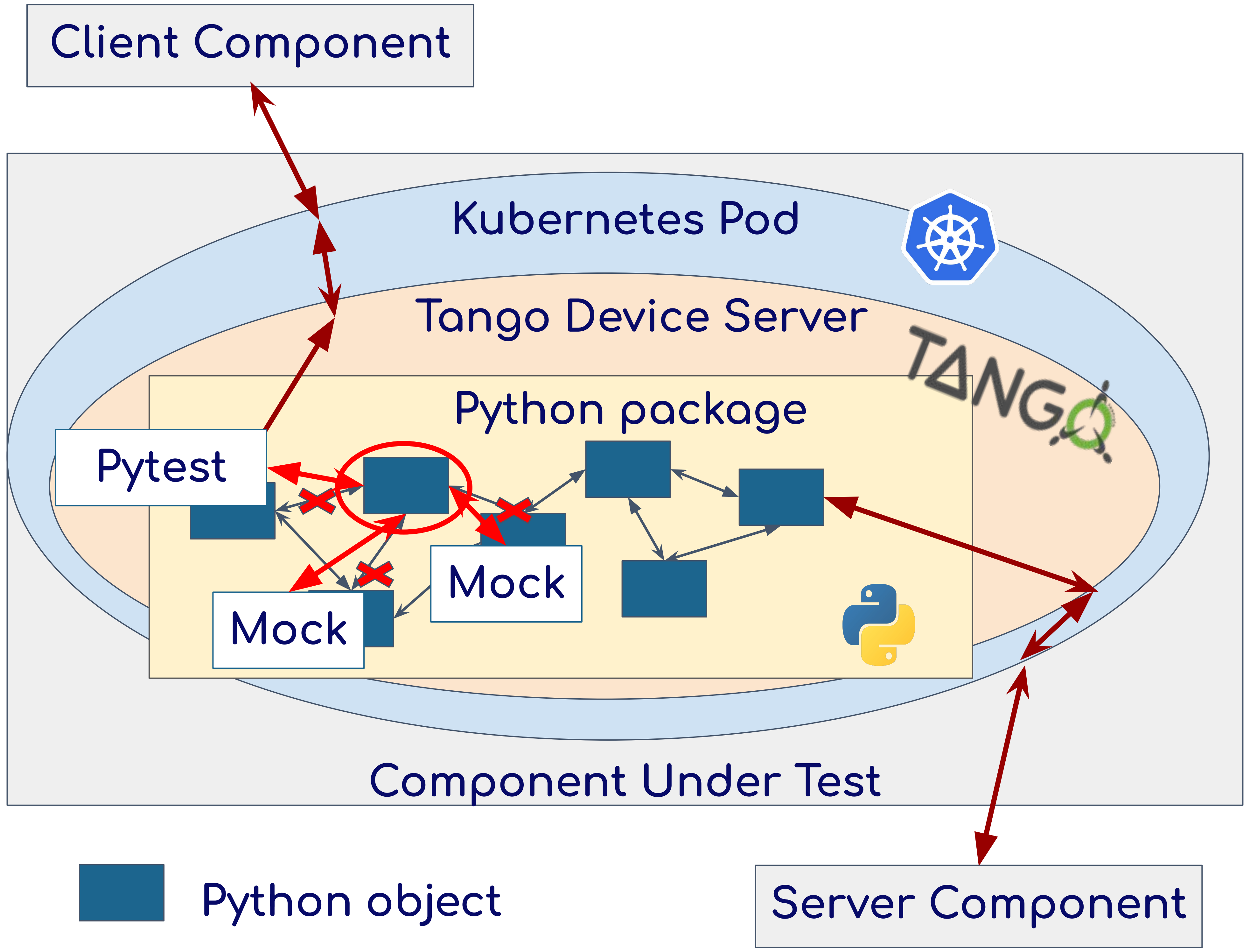
A "software unit" is a *Python object*:

- Test client is a python software (pytest)

(1) From SKAO "Software Testing Policy and Strategy":

Unit Tests

- "The testing of *individual software units* [...] that can be tested in isolation."(1)



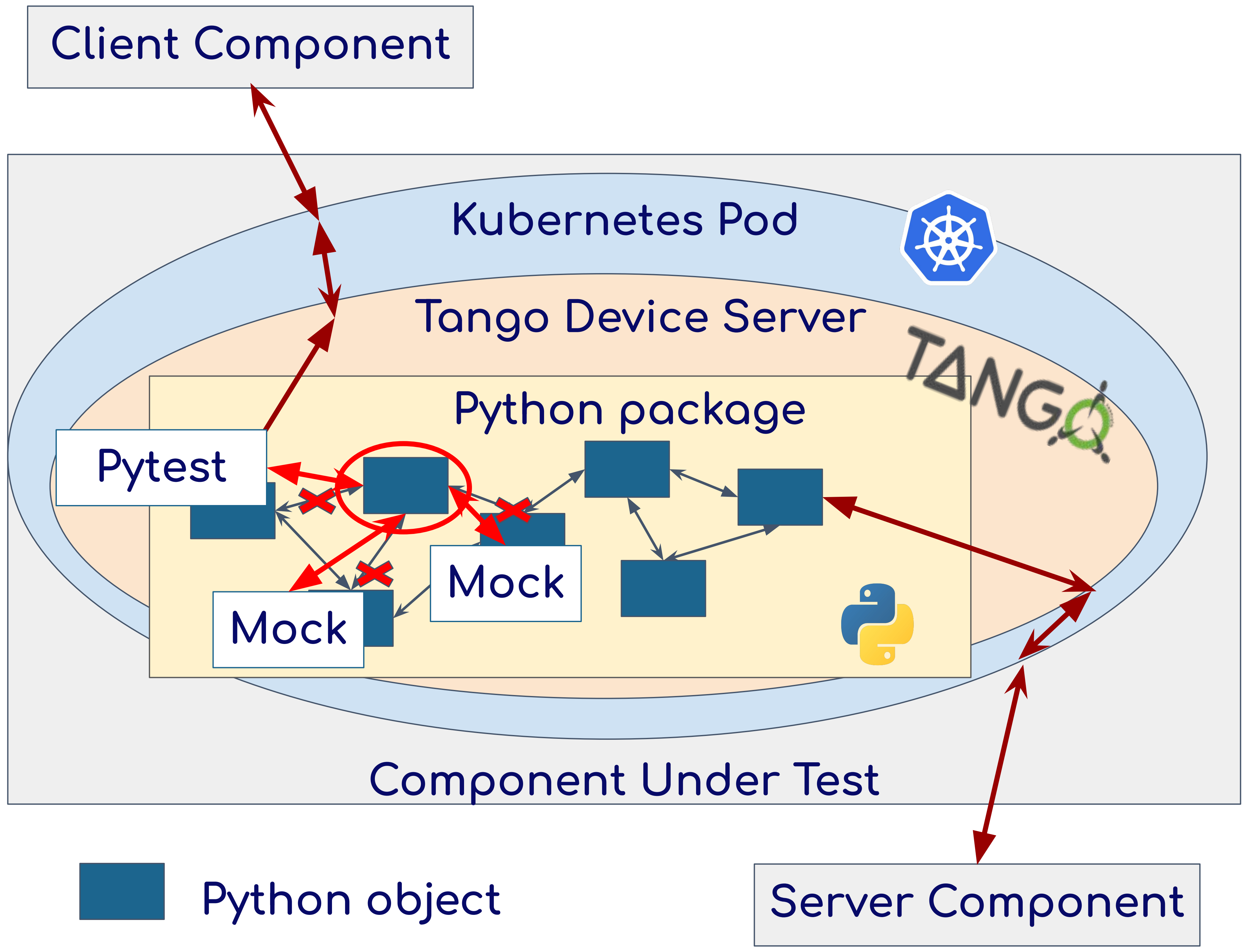
A "software unit" is a *Python object*:

- Test client is a python software (pytest)
- The isolation is obtained by using python *mocks*

(1) From SKAO "Software Testing Policy and Strategy":

Unit Tests

- "The testing of *individual software units* [...] that can be tested in isolation."(1)



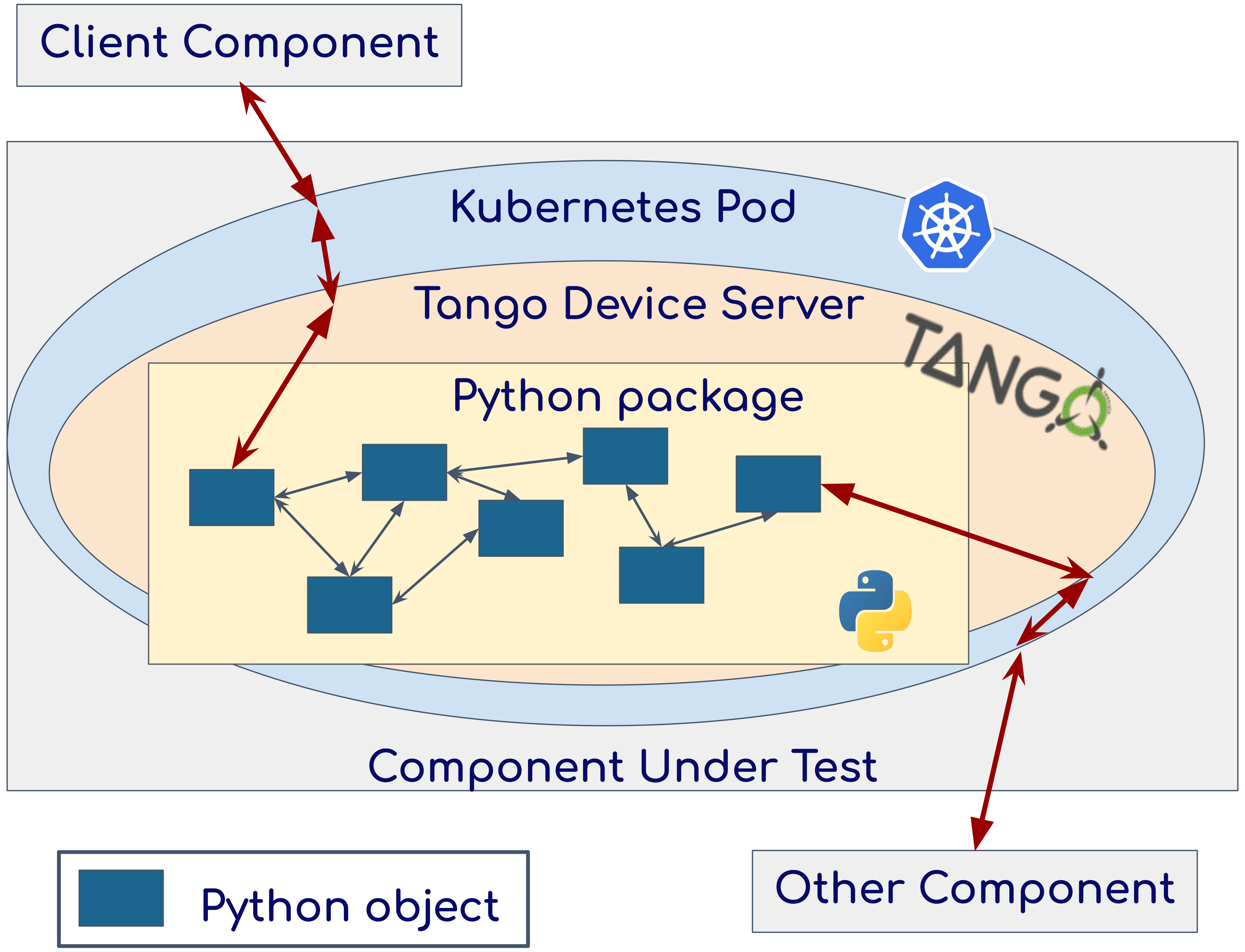
A "software unit" is a *Python object*:

- Test client is a python software (pytest)
- The isolation is obtained by using python *mocks*
- written with a *Test Driven Development (TDD)* approach

(1) From SKAO "Software Testing Policy and Strategy":

Component Tests

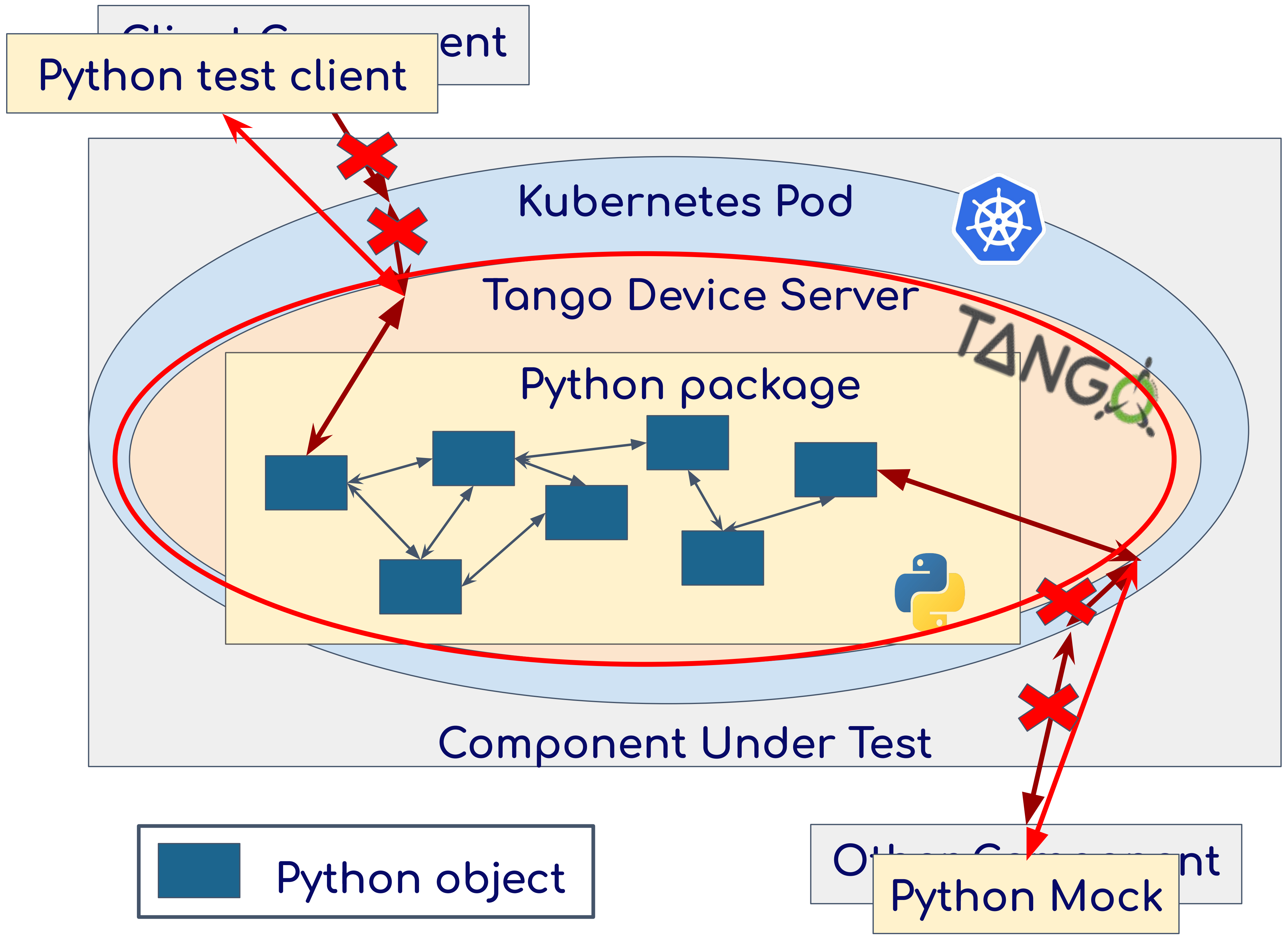
- "Component testing aims at exposing *defects of a particular component*"[1]



[1] From SKAO "Software Testing Policy and Strategy":

Component Tests

- "Component testing aims at exposing *defects of a particular component*"(1)



The "component" is *the Tango Device*

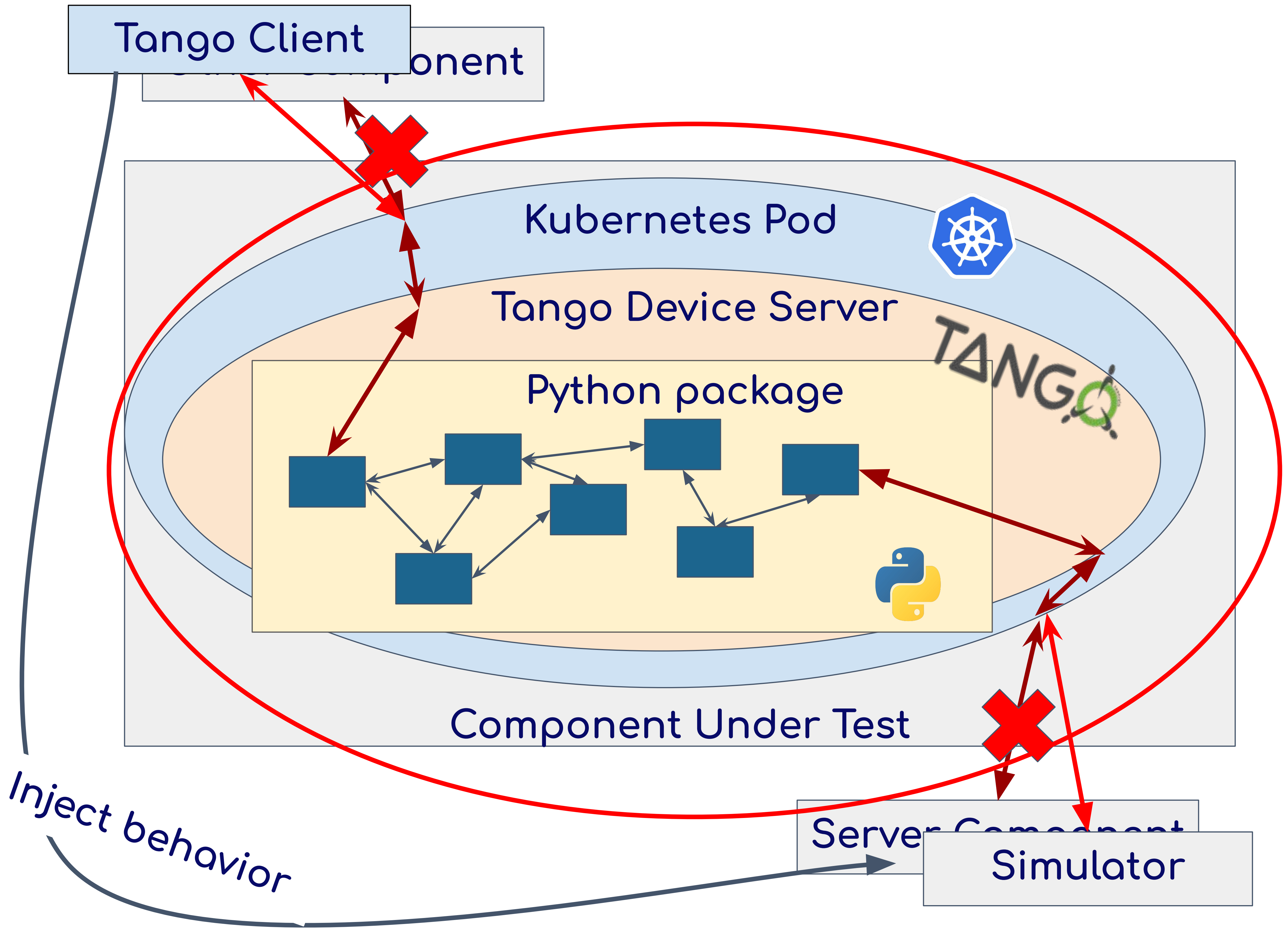
Python-Component Tests

- Other components are substituted with *python Mock*
- Test client is pytest

(1) From SKAO "Software Testing Policy and Strategy":

Component Tests

- “Component testing aims at exposing *defects of a particular component*”[1]



The “component” is the kubernetes (k8s) pod

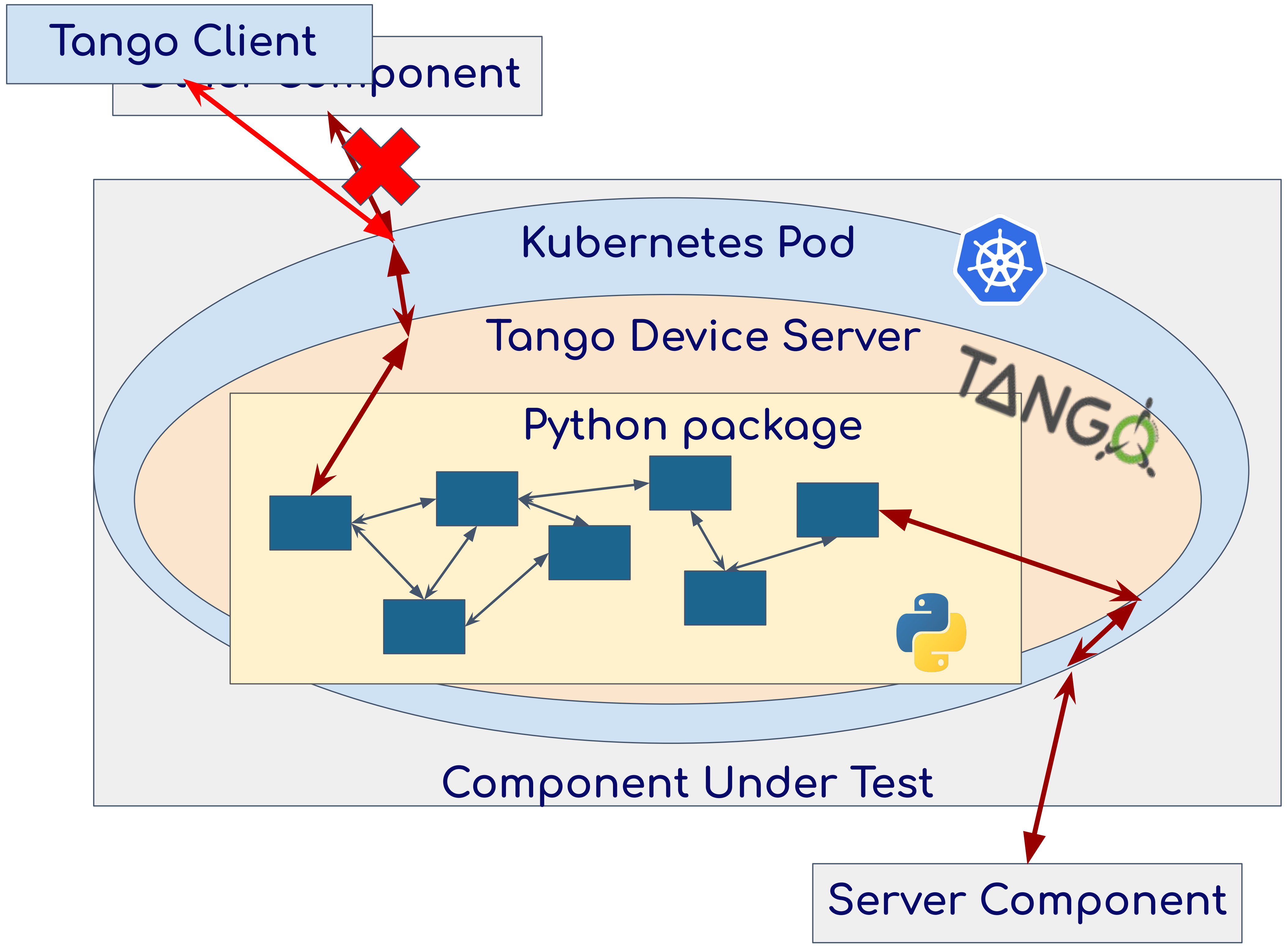
k8s-Component Tests

- Server components are *simulators* (custom Tango devices in k8s)
- Test client is a tango client running on a kubernetes pod.
- Test client can also *inject Simulator’s behavior* (e.g. fault conditions)

[1] From SKAO “Software Testing Policy and Strategy”:

Integration Tests

- "Testing performed to *expose defects* in the interfaces and *in the interaction between components* [...][1]"



- "Integration testing may also include *hardware-software* tests[1]"

(1) https://developer.skao.int/en/latest/policies/ska-testing-policy-and-strategy.html

Fault Conditions Analysis

Fault Conditions tested in CSP.LMC:

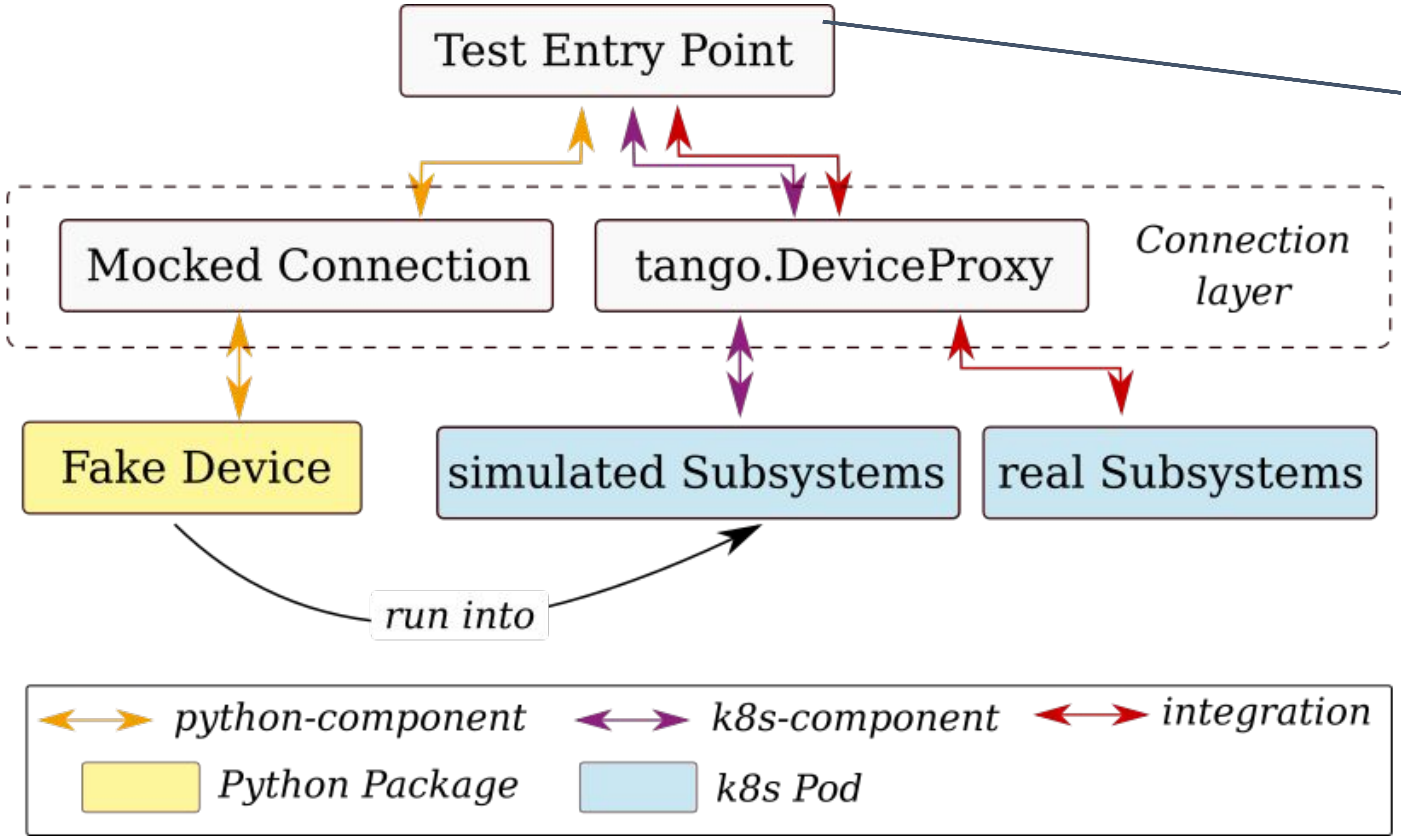
Category[1]	I	II	III	IV	V	VI
Networking	TangoDB connection	Lost connection with a still running device	Lost connection with a stopped device	Event subscription	Disconnection during a command execution	Connection timeouts*
Configuration	Invalid configuration	Unavailable resources*	Unresponsive subsystems*			
Command execution	Wrong inputs	Command not allowed	LMC device failure*	Subsystem device failure*	Slow execution*	
Monitoring	Device failures	Conflicting events	Race conditions			
Infrastructure	Failing/restarti ng pods	Tango DB configuration errors	Tango DB unavailability			

*errors that can be tested only with component tests

[1]From CSP.LMC Fault Condition analysis

Testing Infrastructure

Component/Integration tests can be triggered by the same *Gherkin syntax*



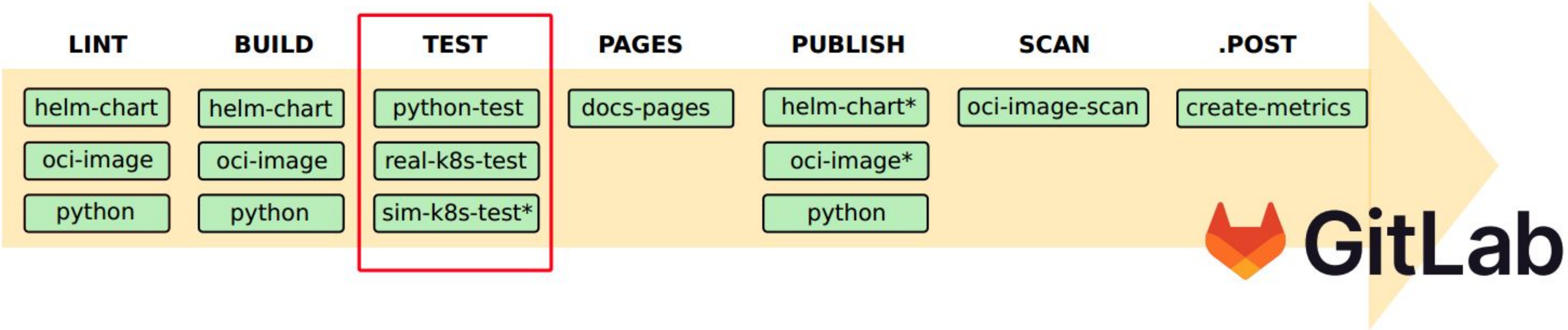

```
@python_component @k8s_component @integration
Scenario: Turn on CSP
    Given CSP Controller setup as off
    And all CSP Subarrays are setup as off
    When CSP Controller's OnCommand is triggered with default
    Then CSP Controller is on
    And all CSP Subarrays are on
```

- Decorators select the context where to run the test

Running the same test on different context (python/k8s/integration) helps to find the root of the failure

CI/CD pipeline

Tests are performed by a *Continuous Integration & Delivery and/or Deployment (CI/CD)* Pipeline[1]:



- at every change of the code (automated regression tests);
- on demand;
- with scheduled periodic jobs

Integration tests can be triggered on *different facilities, with and without hardware.*

[1] M.Di Carlo et al. *"CI-CD Practices at SKA"* Proc SPIE 12189 (2022)

Data mining on test results

Collecting information on test execution, will give us the possibility :

- to explore correlation between failures;
- to quantify the rate of success of a specific functionality.

Test name	Result	Version	Date and Time	Test Type	Exec. time (ms)	Facility	Hardware	Cause of failure (log)	Category
Turn On CSP	PASSED	0.16.2	29/09/2023 19:22	python-compone nt	300	STFC	//	//	Happy paths
Turn On CSP	FAILED	0.16.2	29/09/2023 19:22	k8s-component	10000	STFC	//	See attachment	Happy paths
Turn On CSP	FAILED	0.16.2	29/09/2023 19:22	integration	10000	STFC	NO	See attachment	Happy paths
Turn On CSP	PASSED	0.16.2	29/09/2023 21:00	integration	20	PSI	NO	//	Happy paths
Turn On CSP	PASSED	0.16.2	29/09/2023 21:03	integration	450	PSI	YES	//	Happy paths
…	…	…	…		…	…	…	…	

Conclusions

- **a multi-level approach** (unit/component/integration) is employed to evaluate our software within **distinct contexts**
- **testing infrastructure** has been consolidated to eliminate redundancy with **shared testing scripts**
- A systematic approach has been devised for to the **categorization and testing of fault conditions**
- **data mining techniques** can be used to collect and analyze the results.

ffff

For further informations: gianluca.marotta@inaf.it

Thank you for your attention!

ICALEPCS CAPE TOWN SOUTH AFRICA 2023

Any questions?

INAF ISTITUTO NAZIONALE DI ASTROFISICA

COSYLAB Advancing humanity. Engineering remarkable.

IDS Design Coaching for Excellence