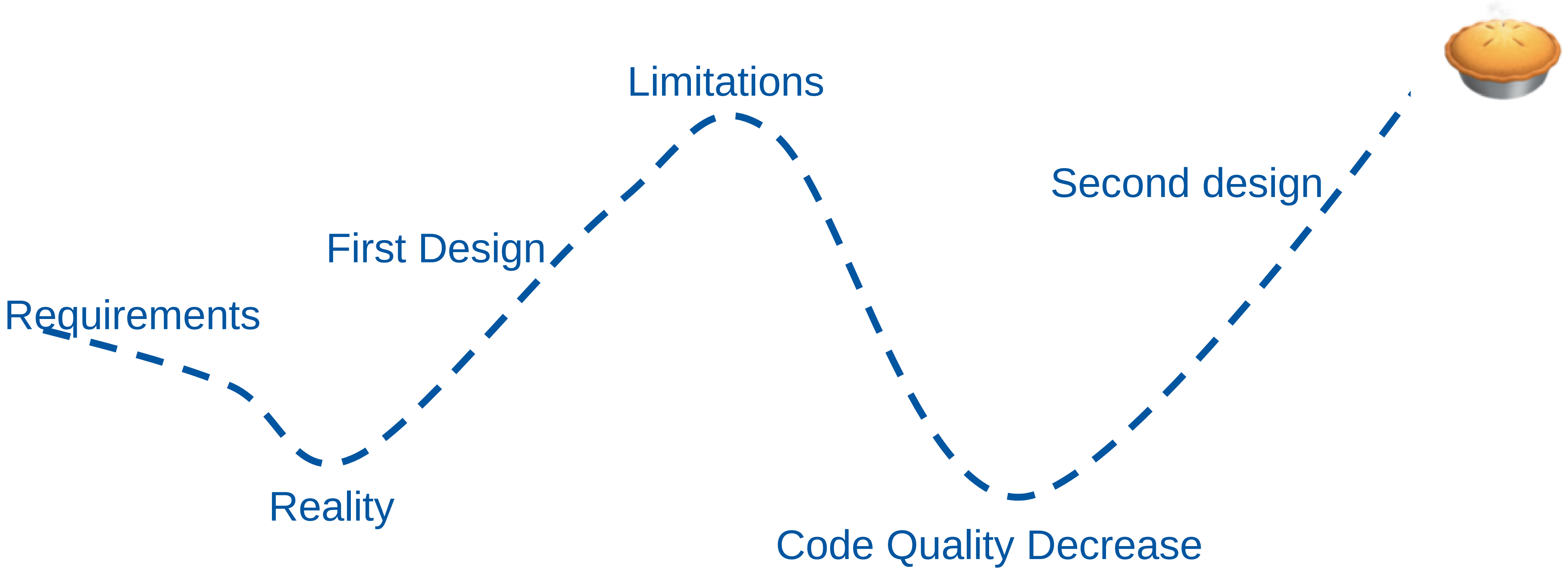
LHC Signal Monitoring Project

Development of an Embedded Domain Specific Language for Signal Query and Analysis

Kasper Andersen, Zinur Charifoulline, Per Hagen, Michał Maciejewski, Christoph Obermair, Arjan Verweij, Sivert Sagmo

TE-MPE-MS: Thibaud Buffet, Jean-Christophe Garnier, Tiago Martins Ribeiro, Markus Zerlauth
IT-DB-SAS: Piotr Mrówczynski, Prasanth Kothuri
IT-ST-FDO: Diogo Castro

Developer’s Story



Requirements

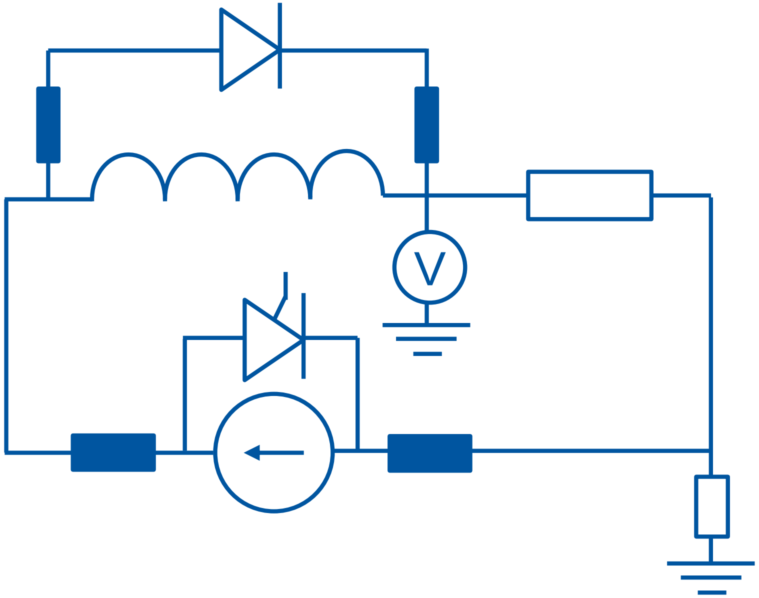
Input

Data concerning converters, busbars, current leads, magnets, QPS, cryogenics, etc, as stored in:

1. CALS
2. PM files

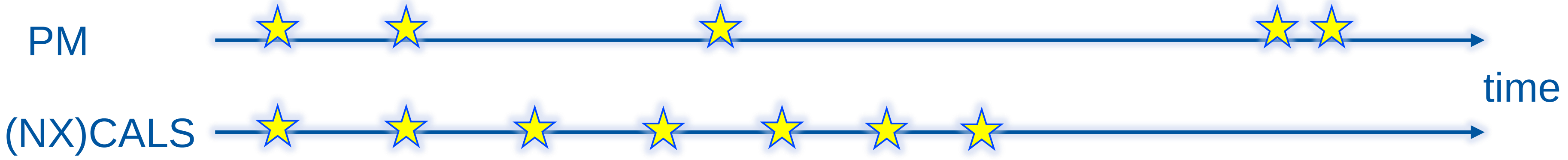
CALS and PM will soon be merged into NXCALS.

Often we are interested in parameters that are derived from one or more existing signals, e.g. R=V_MEAS/I_MEAS, current decay time constant during a FPA, etc.



→ Heterogeneous data sources
→ Various signal processing algorithms

Logging Databases - *Overview*



	PM	CALS	NXCALS
time definition	event	period of time	period of time
signal definition	system, source, className, signal	signal	system, (device, property), signal
return type	json	dictionary of arrays	spark DataFrame
time unit	ns	us	ns
API	REST	pytimber	Apache spark

Logging Databases - *API*

PM – REST API

http://pm-api-pro/v2/pmdata/signal?system=FGC&className=51_self_pmd&source=RPTE.UA47.RB.A45&timestampInNanos=1426220469520000000&signal=STATUS.I_MEAS

CALS - pytimber

```python
import pytimber
ldb = pytimber.LoggingDB()
ldb.get('RPTE.UA47.RB.A45:I_MEAS', '2015-03-13 05:20:59.491000200', '2015-03-13 05:21:19.491000')
```

NXCALS - spark

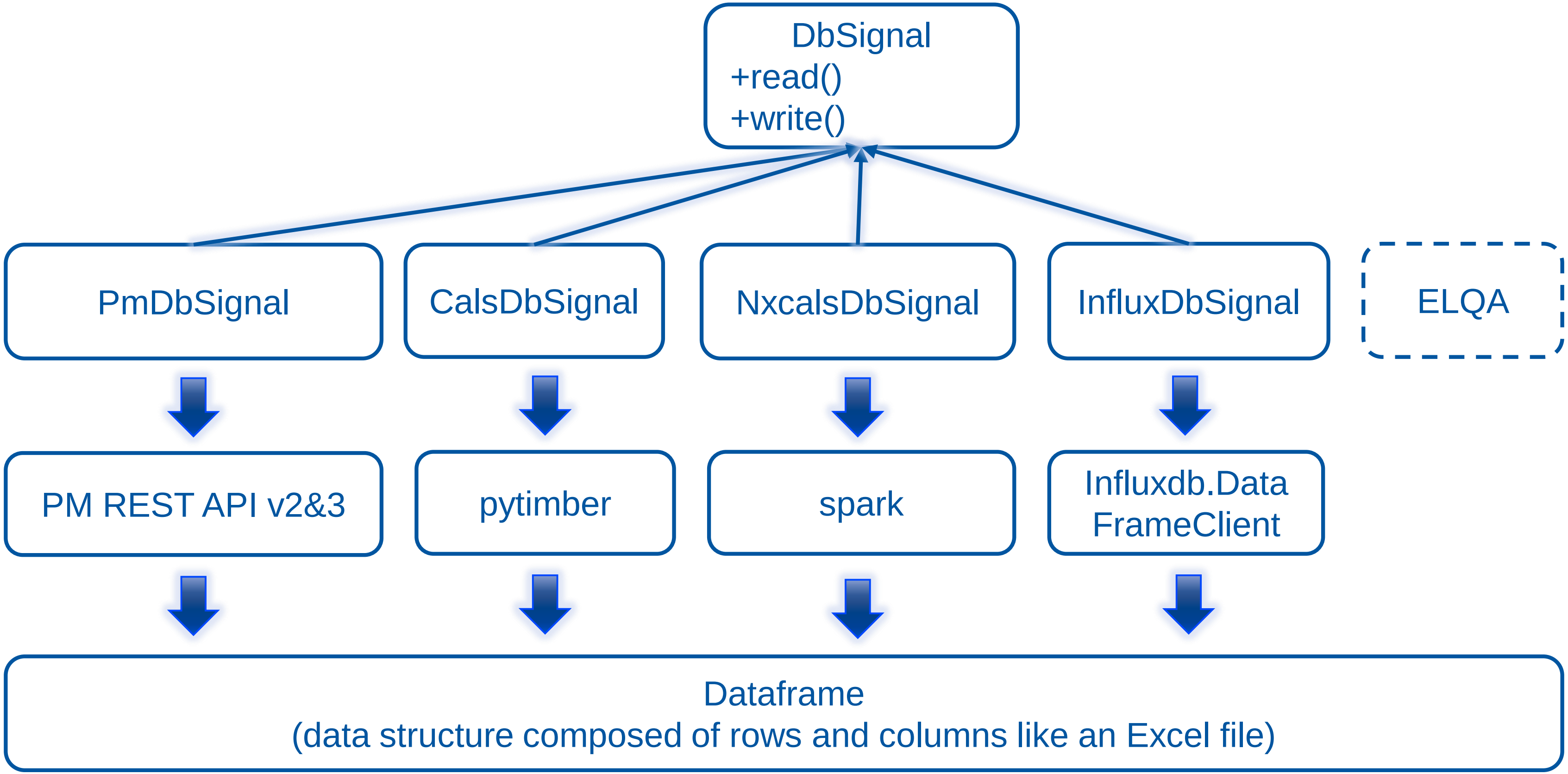
```python
from cern.nxcals.pyquery.builders import *
import pandas as pd
signal_df = DevicePropertyQuery.builder(spark).system('CMW') \
    .startTime(pd.Timestamp('2015-03-13T04:20:59.491000000').to_datetime64())\
    .endTime(pd.Timestamp('2015-03-13T04:22:49.491000000')) \
    .entity().device('RPTE.UA47.RB.A45').property('SUB') \
    .buildDataset().select('acqStamp', 'I_MEAS').dropna().sort("acqStamp").toPandas()
```

Logging Databases – *Analytics*

	PM	PM	CALS	CALS	NXCALS	NXCALS
	event query	signal query	signal query	feature query	signal query	feature query
timing	fast	fast	can be slow	rather fast	slow	fast
execution	serial	serial	serial	?	serial	parallel
use	simple	simple	simple	simple	simple	hard

→ Need to extend analysis capabilities natively provided by the databases

Unified Database Access



DbSignal Classes

PM

```
i_meas_df = Signal().read('pm', signal='STATUS.I_MEAS', system='FGC',
                          source='RPTE.UA47.RB.A45', className='51_self_pmd',
                          eventTime=1426220469520000000)
```

CALS

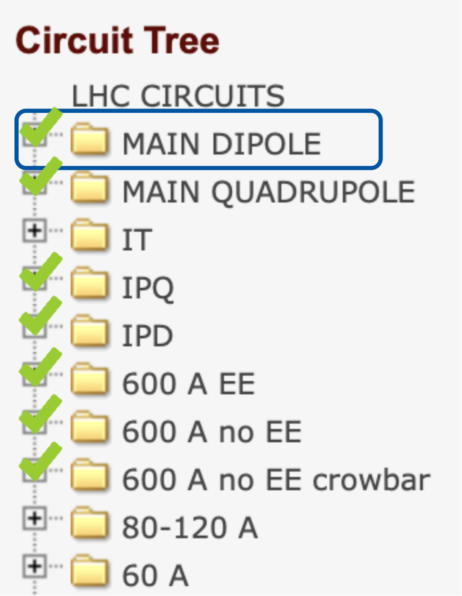
```
import pytimber
ldb = pytimber.LoggingDB()
i_meas_df = Signal().read('cals', signal="RPTE.UA47.RB.A45:I_MEAS",
                          t_start='2015-03-13 05:20:59.4910002', duration=[(10, 's'), (100, 's')], ldb=ldb)
```

NXCALS

```
i_meas_df = Signal().read('nxcals', signal='I_MEAS', nxcals_system='CMW',
                          nxcals_device='RPTE.UA47.RB.A45', nxcals_property='SUB',
                          t_start=1426220469491000000, duration=[(10, 's'), (100, 's')], spark=spark)
```

Metadata

The **Metadata** module contains methods to access various signal and circuit names.



System

- CIP
- CRYO
- PIC
- PC
- QDS
- QH
- BUSBAR
- DIODE
- VF
- LEADS_EVEN
- LEADS_ODD
- EE

Signal Name

- I_A
- I_B
- I_EARTH
- I_EARTH_PCNT
- I_MEAS
- I_REF
- V_MEAS
- V_REF

Circuit Name

- RB.A12
- RB.A23
- RB.A34
- RB.A45
- RB.A56
- RB.A67
- RB.A78
- RB.A81

Wildcard

- Cell
- Magnet
- Crate
- VF
- Busbar

Mapping circuit components → Circuit topology in one place

Structure for each circuit is the same → single analysis for many circuits

Signal names change over time → we need to keep track of the changes

Limitation

```
circuit_type = 'RB'
circuit_name = 'RB.A12'
t_start = '2015-01-13 16:59:11+01:00'
t_end = '2015-01-13 17:15:46+01:00'
db = 'NXCALS'
system = 'PC'

metadata_pc = SignalMetadata.get_circuit_signal_database_metadata(circuit_type, circuit_name, system, db)
I_MEAS = SignalMetadata.get_signal_name(circuit_type, circuit_name, system, db, 'I_MEAS')

i_meas_nxcals_df = Signal().read(db, signal=I_MEAS, t_start=t_start, t_end=t_end,
                                 nxcals_device=metadata_pc['device'], nxcals_property=metadata_pc['property'], nxcals_system=metadata_pc['system'],
                                 spark=spark)

i_meas_nxcals_df = SignalUtilities.synchronize_df(i_meas_nxcals_df)
i_meas_nxcals_df = SignalUtilities.convert_indices_to_sec(i_meas_nxcals_df)
```

Several design flaws leading to inconsistency and code duplications:

- use of multiple methods, multiple arguments (duplicated across methods)
- multiple local variables (naming consistency across analysis modules)
- order of methods and arguments (with duck typing) not fixed

What if we want to get current for each circuit?
What if we want to get several current signals?

Domain Specific Language

Natural languages have certain structure [1]

English: {Subject}.{Verb}.{Object}: John ate cake

Japanese: {Subject}.{Order}.{Verb}: John-ga keiki-o tabeta
John cake ate

One can enforce syntactical order in code:

- Domain Specific Language – new language, requires parser
- Embedded Domain Specific Language – extends existing language

[1] K. Gulordava, Word order variation and dependency length minimisation: a cross-linguistic computational approach, PhD thesis, UniGe

pyeDSL

We propose a python embedded Domain Specific Language (pyeDSL):

```
{DB}.{CIRCUIT_TYPE}.{DURATION}.{METADATA}.{QUERY}
```

+ each parameter defined once (validation of input at each stage)
+ single local variable
+ order of operation is fixed
+ support for vector inputs
+ time-dependent metadata

e.g.

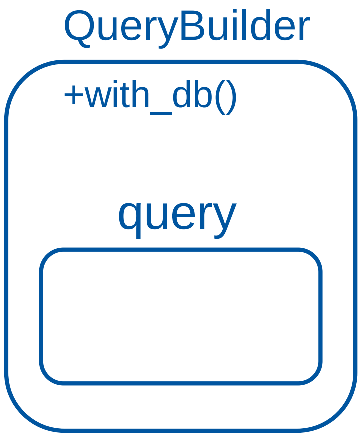
```
df = QueryBuilder().with_db().with_circuit_type().with_duration().with_metadata()\
        .signal_query().dfs[0]
```

Demo 1

Signal query

How does it work?

At each stage only a few methods are available, which update hidden container.

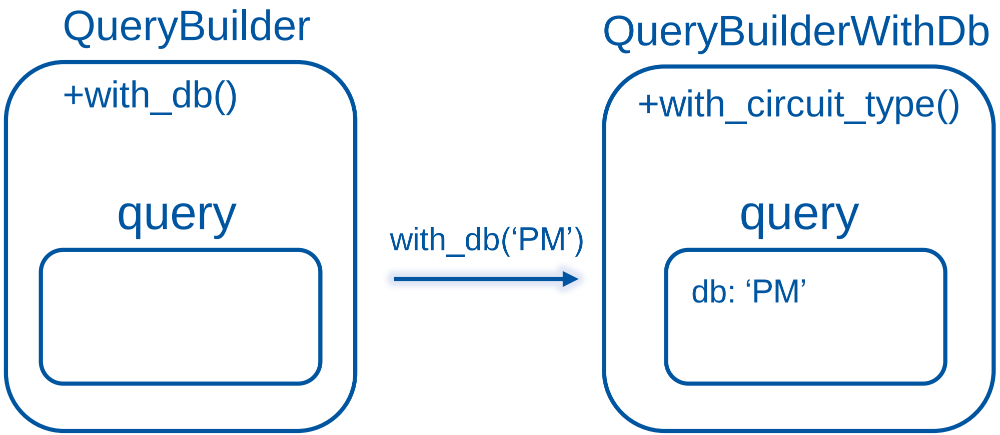



```
dfs = QueryBuilder()
```

→ Nested Builder Design Pattern
→ Abstract Factory Design Pattern

How does it work?

At each stage only a few methods are available, which update hidden container.

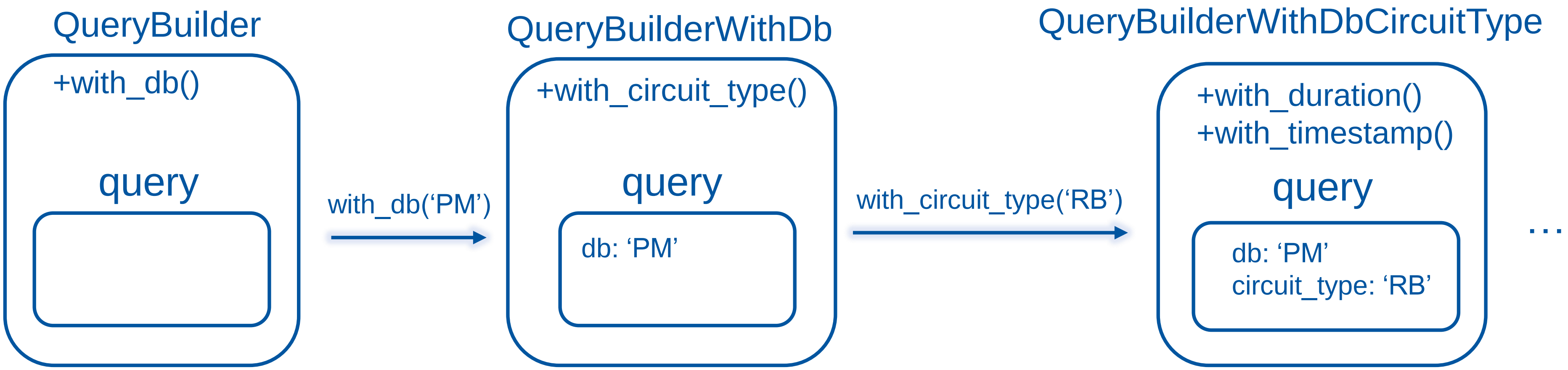



```
dfs = QueryBuilder().with_db('PM')
```

→ Nested Builder Design Pattern
→ Abstract Factory Design Pattern

How does it work?

At each stage only a few methods are available, which update hidden container.




```
dfs = QueryBuilder().with_db('PM').with_circuit_type('RB')
```

→ Nested Builder Design Pattern
→ Abstract Factory Design Pattern

pyeDSL – Examples (1/2)

PM – event query

```
QueryBuilder().with_db('PM').with_circuit_type('RB')\
    .with_duration(t_start='2015-03-13 05:20:59.4910002', duration=[(100, 's'), (100, 's')]) \
    .with_metadata(circuit_name='RB.A45', system='PC').event_query().df
```

PM – signal query

```
QueryBuilder().with_db('PM').with_circuit_type('RB')\
    .with_timestamp(1426220469520000000) \
    .with_metadata(circuit_name='RB.A45', system='PC', signal='I_MEAS').signal_query().dfs[0]
```

CALS – signal query

```
QueryBuilder().with_db('CALS').with_circuit_type('RB')\
    .with_duration(t_start='2015-03-13 05:20:59.4910002', duration=[(100, 's'), (100, 's')]) \
    .with_metadata(circuit_name='RB.A45', system='PC', signal='I_MEAS').signal_query(dbconnector=ldb).dfs[0]
```

→ One can quickly change a database and circuit type, name
→ A sentence created with the language corresponds to database type

pyeDSL – Examples (2/2)

NXCALS – signal query

```
QueryBuilder().with_db('NXCALS').with_circuit_type('RB')\
    .with_duration(t_start='2015-03-13 05:20:59.4910002', duration=[(100, 's'), (100, 's')]) \
    .with_metadata(circuit_name='RB.A45', system='PC', signal='I_MEAS').signal_query(dbconnector=spark).dfs[0]
```

NXCALS – feature query*

```
QueryBuilder().with_db('NXCALS').with_circuit_type('RB')\
    .with_duration(t_start='2015-03-13 05:20:59.4910002', duration=[(100, 's'), (100, 's')]) \
    .with_metadata(circuit_name='RB.A45', system='PC', signal='I_MEAS') \
    .feature_query(features=['min', 'max', 'std', 'mean'], dbconnector=spark).dfs[0]
```

*work in progress

pyeDSL – Polymorphism

Multiple circuit names

```
QueryBuilder().with_db('NXCALS').with_circuit_type('RB')\
    .with_duration(t_start='2015-03-13 05:20:59.4910002', duration=[(100, 's'), (100, 's')]) \
    .with_metadata(circuit_name=['RB.A12', 'RB.A45'], system='PC', signal='I_MEAS')\
    .signal_query(dbconnector=spark).dfs[0]
```

Multiple system names

```
QueryBuilder().with_db('PM').with_circuit_type('RB').with_timestamp(1544622149598000000) \
        .with_metadata(circuit_name='RB.A12', system=['LEADS_EVEN', 'LEADS_ODD'], signal='U_HTS') \
        .signal_query().dfs
```

Multiple signal names

```
QueryBuilder().with_db('PM').with_circuit_type('RQ').with_timestamp(1544622149598000000) \
        .with_metadata(circuit_name='RQD.A12', system='QDS', signal=['U_1_EXT', 'U_2_EXT'],
                       source='16L2', wildcard={'CELL': '16L2'})\
        .signal_query().dfs
```

Wildcard

```
QueryBuilder().with_db('CALS').with_circuit_type('RQ')\
        .with_duration(t_start=int(1544622149613000000), duration=[(50, 's'), (150, 's')]) \
        .with_metadata(circuit_name='RQD.A12', system='DIODE_RQD', signal='U_DIODE_RQD', wildcard={'MAGNET': '*'})\
        .signal_query(dbconnector=ldb)
```

→ Internal handling of for loops – reduced amount of code in analysis

Adding Adjectives

Once a signal is queried, we can perform some operations on each of them.
In this case, the order of operations does not matter (but can be checked)

```
{DB}.{CIRCUIT_TYPE}.{DURATION}.{METADATA}.{QUERY}.{PRE-PROCESSING}
                                                  .synchronize_time()
                                                  .convert_index_to_sec()
                                                  .filter()
                                                  .remove_initial_offset()
```

e.g.

```
df = QueryBuilder().with_db().with_circuit_type().with_duration().with_metadata()\
         .signal_query().synchronize_time().convert_index_to_sec().dfs[0]
```

Demo 2

Signal query and processing

Signal Assertions

Hardware Commissioning procedures check ranges of certain signals

Responsible	Type of analysis	Criterion
	Automatic analysis on earth current and error current	I_EARTH_PLI3_A5 < I_EARTH_MAX I_ERR_PLI3_A5 < I_ERR_MAX
MP3	Splice signals	From board A and board B separately R_bus_max <3 nOhm Individual R_splice_max<0.5nOhm R_mag<50 nOhm
	Current lead	46 < TT891A < 54K Abs(U_RES)< 40mV and no drift Abs(U_HTS) < 0.5mV
	Calorimetric (if done)	dT/dt (TT821)< 5 mK/hr

AssertionBuilder class performs signal assertions.

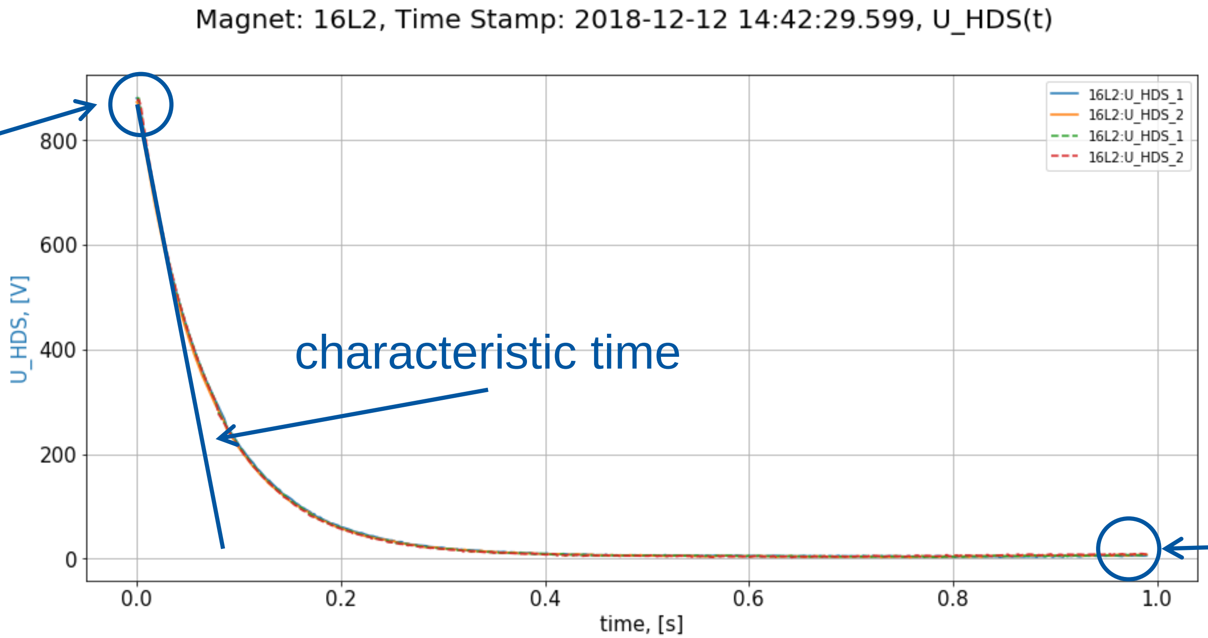
Demo 3

Signal assertion

Feature Engineering

Signal analysis (e.g., quench heater discharges) requires extraction of certain characteristic features

initial voltage



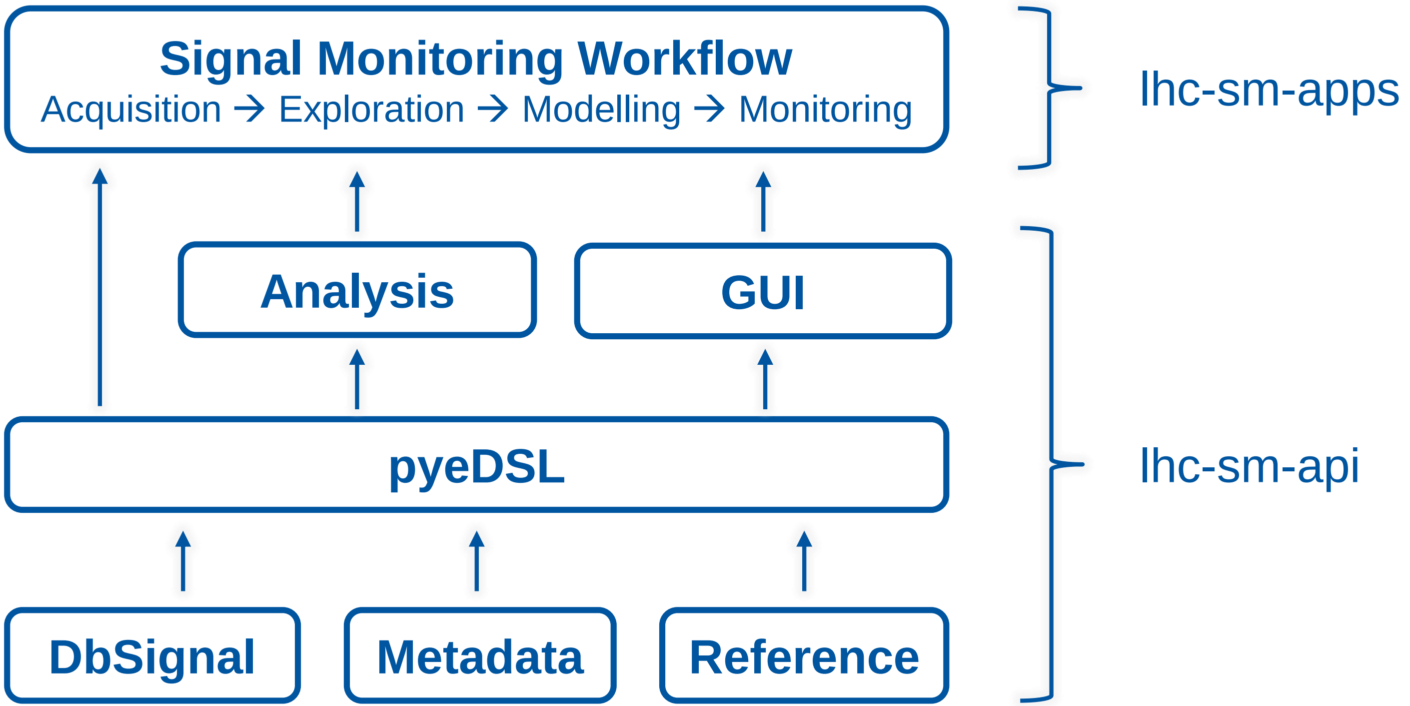
final voltage

FeatureBuilder performs feature engineering in a generic way.

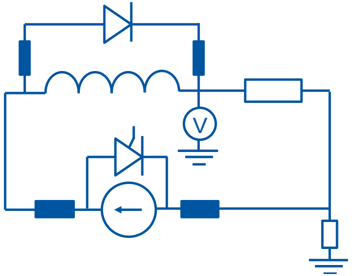
Demo 4

Signal feature extraction

Architecture



→ With a solid signal query and processing API we can advance faster with developing HWC and monitoring notebooks and extend to other circuits.



Analysis - Summary

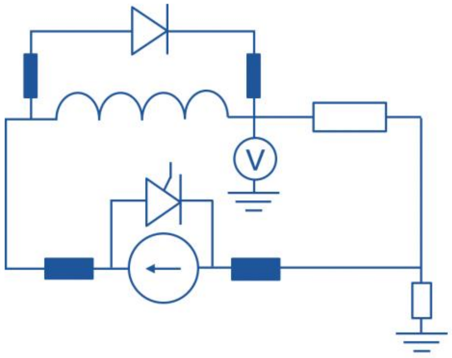
Signal Monitoring Workflow

Acquisition → Exploration → Modelling → Monitoring

QH	RB	RQ	RQX	IPQ/D	
COLDBB	RB	RQ			
PC	RB	RQ			600A
DIODE	RB	RQ			
GND	RB	RQ			
EE	RB	RQ			600A
MAGNET	RB	RQ			
DFB	RB	RQ			

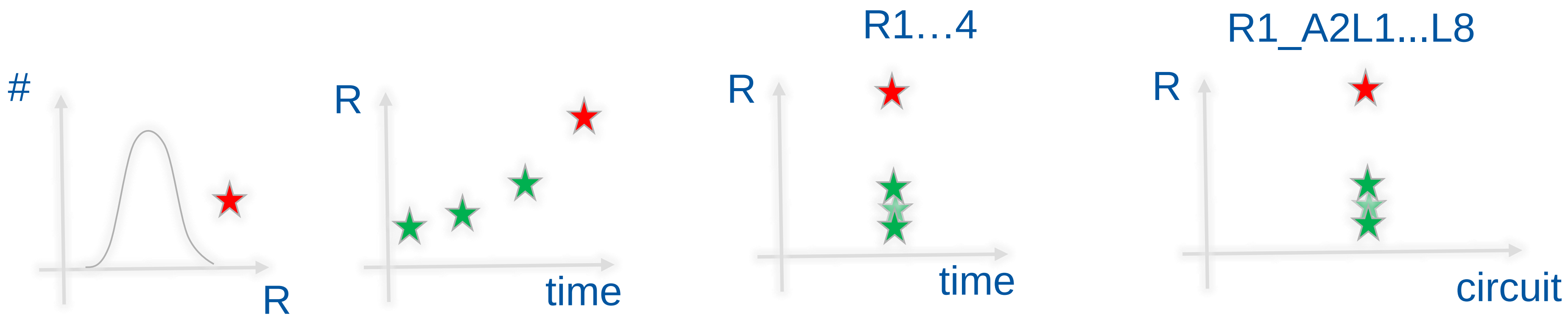
Run 3 - Monitoring

HWC and Quench Analysis



Analysis - Modelling

1. With historical data we derive expected behavior and trends.
2. With on-line data we compare behavior with others.



digital-twin | trends | intra-component | cross-population

Modelling Methods

Physical Models

+ Provide access to non-measurable states
+ High-numerical precision
- As accurate as available parameters, measurements

Data-Driven Models

Threshold-Based

+ Embed expert knowledge on parameters
+ Give a clear answer
- Require adjustments

Probabilistic

+ Distribution of parameters (features)
- Do not give a clear answer
- As good as data

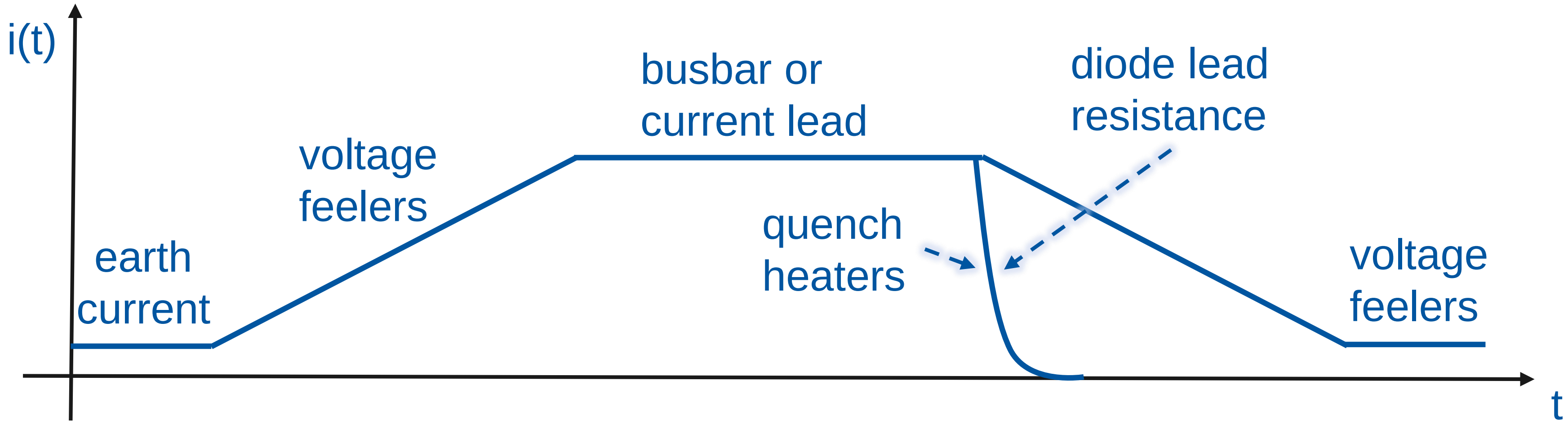
Machine Learning

+ Encode non-obvious relations
+ Find patterns, correlations
+ Encode expert knowledge
- Do not give a clear answer
- As good as input data

Hybrid Methods

+ Mixture of deterministic (limited) and non-deterministic (flexible)
- More complicated analysis

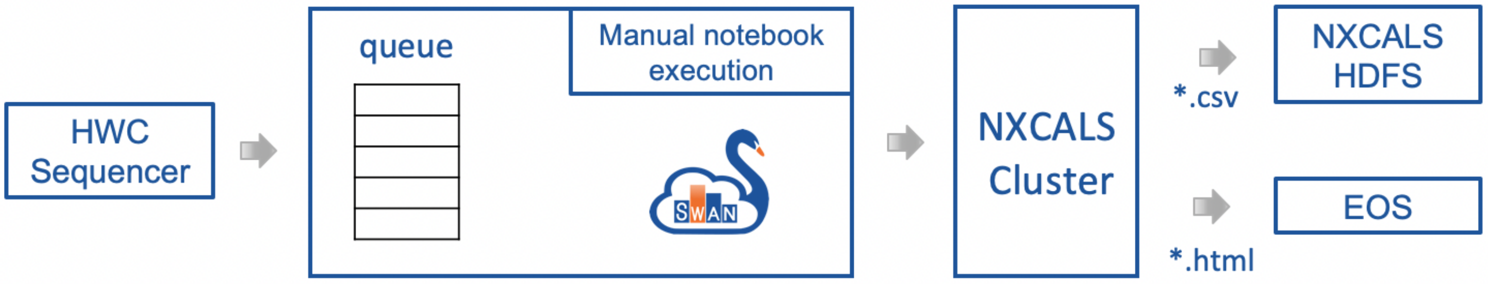
Acquisition → Exploration → Modelling → Monitoring



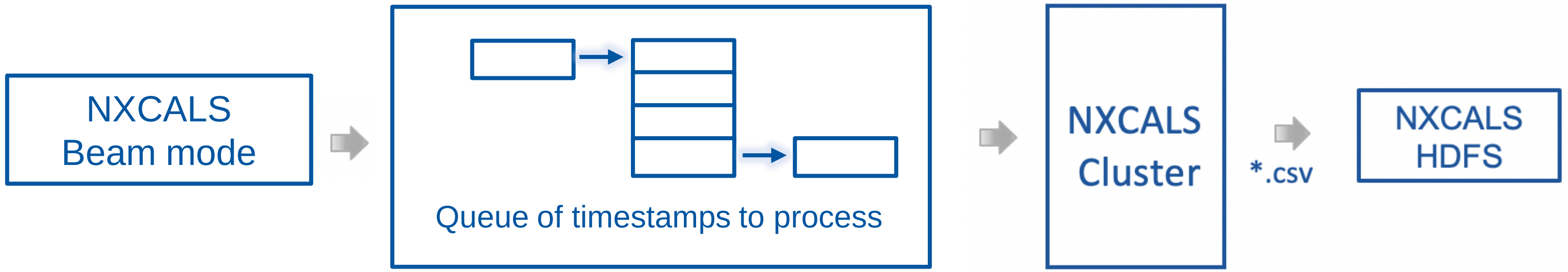
Automatic execution of monitoring application depends on the operation state:
- triggered by PM events (PC, QH, MAGNET)
- triggered by change in the beam mode (GND, COLDBB)
- in regular intervals, e.g. every hour (DFB)

Automatic Execution

Manual execution of HWC notebooks



Automatic execution of long-running historical analyses*



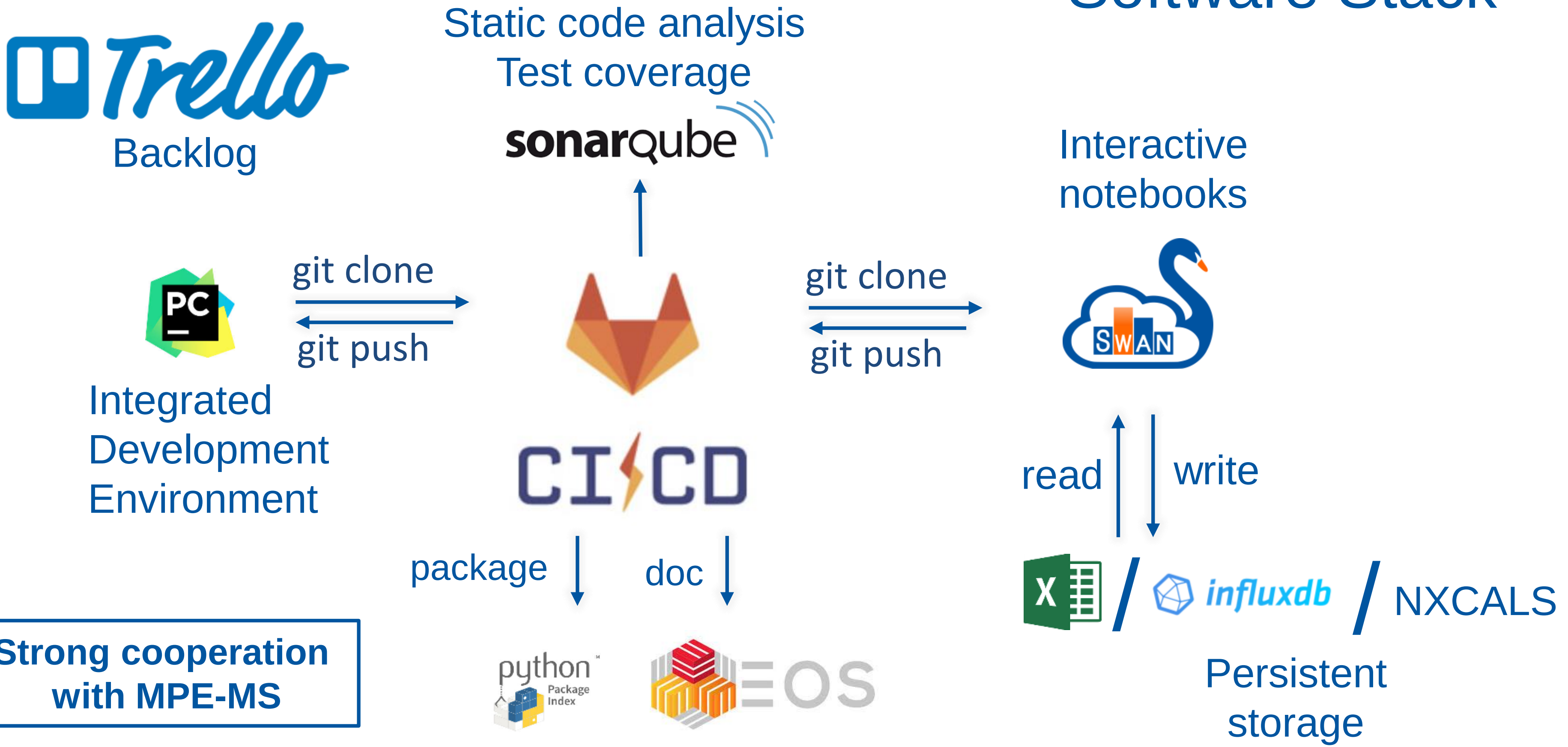
→ Need for analysis trigger and analysis supervision (Apache AirFlow)

Courtesy: P. Mrówczyński
https://gitlab.cern.ch/db/swan-spark-notebooks
https://gitlab.cern.ch/LHCData/lhc-sm-apps/merge_requests/1

Summary

1. Introduction of pyeDSL unifies database query and simplifies code
2. The signal processing and feature engineering is provided but limited
3. The pyeDSL introduces clear structure by enforcing order of operations
4. The directions for extension are clearly identified
5. The development time and maintenance effort havereduced considerably

Software Stack



Strong cooperation with MPE-MS

We rely on industry-standard tools for the development automation
Majority (except for PyCharm IDE and Python Package Index) services are supported by CERN IT

Resources

API

In order to use the project the API has to be installed in SWAN

```
pip install --user lhcsmapi
```

Check the latest version at https://pypi.org/project/lhcsmapi/
The documentation for the API is stored at http://cern.ch/lhc-sm-api.
The repository of the API is available at a GitLab http://gitlab.cern.ch/lhcdata/lhc-sm-api

Applications

The released use cases are available at the SWAN gallery
The beta versions of the use cases are stored at http://gitlab.cern.ch/lhcdata/lhc-sm-apps

Open in SWAN

Project website: https://twiki.cern.ch/twiki/bin/view/TEMPEPE/Signal_Monitoring