DCWiz Data Modules

Zhaomeng Zhu

Agenda

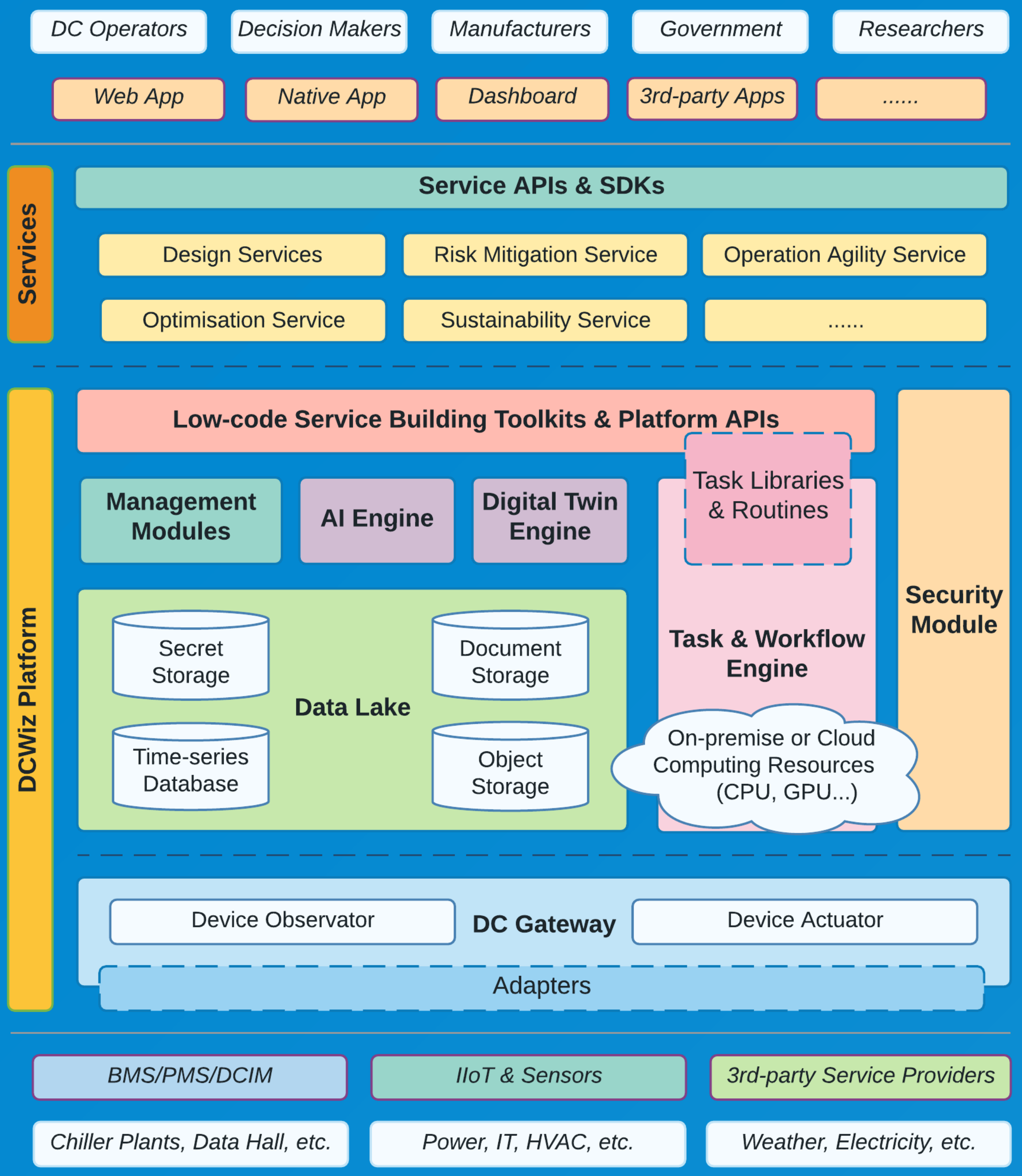
- DCWiz Platform
- Data-related Problems
- Karez: the Data Collection Framework
- Utinni: Bridging the Data Lake and Applications
- Summary & Roadmap

DCWiz Platform

Main Features

- Collecting DC data (from DCIM, BMS, PMS, etc.);
- Storing data in data lake;
- Providing the data and models to AI/DT engines and applications;
- Providing toolset to help building AI and DT models;
- Scheduling AI & DT tasks (on-demand, on-event or periodically);
- Managing assets and users;
- And more...

DCWiz Platform Architecture (Simplified)



Data-relared Problems

Given the heterogeneity of

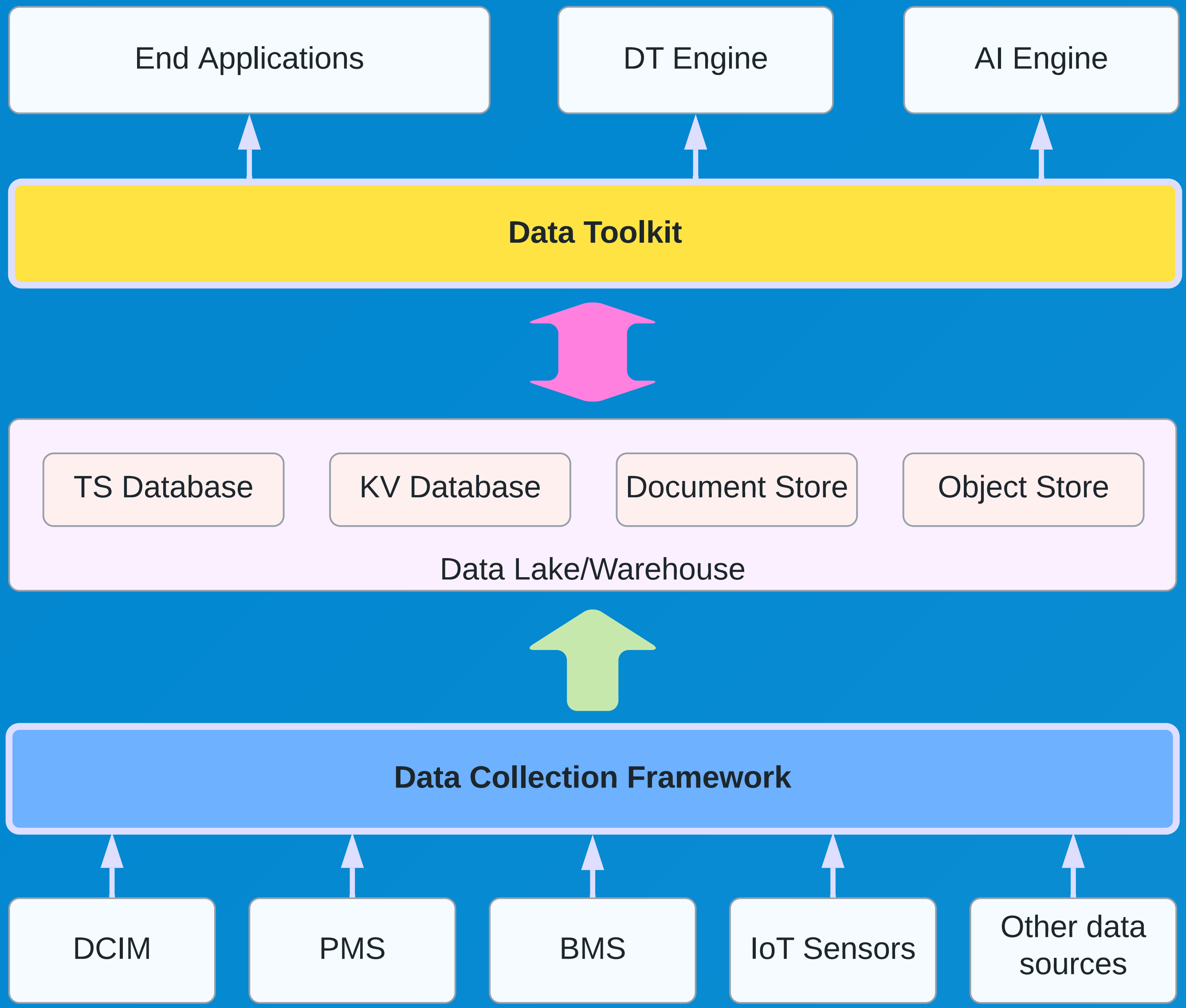
- data sources
- data formats
- storage types
- use scenarios

How to

1. **effeciently collect data from data sources?**
2. **easily make use of the data in various application ?**

DCWiz Data Modules

- Collection of data
- Use of data



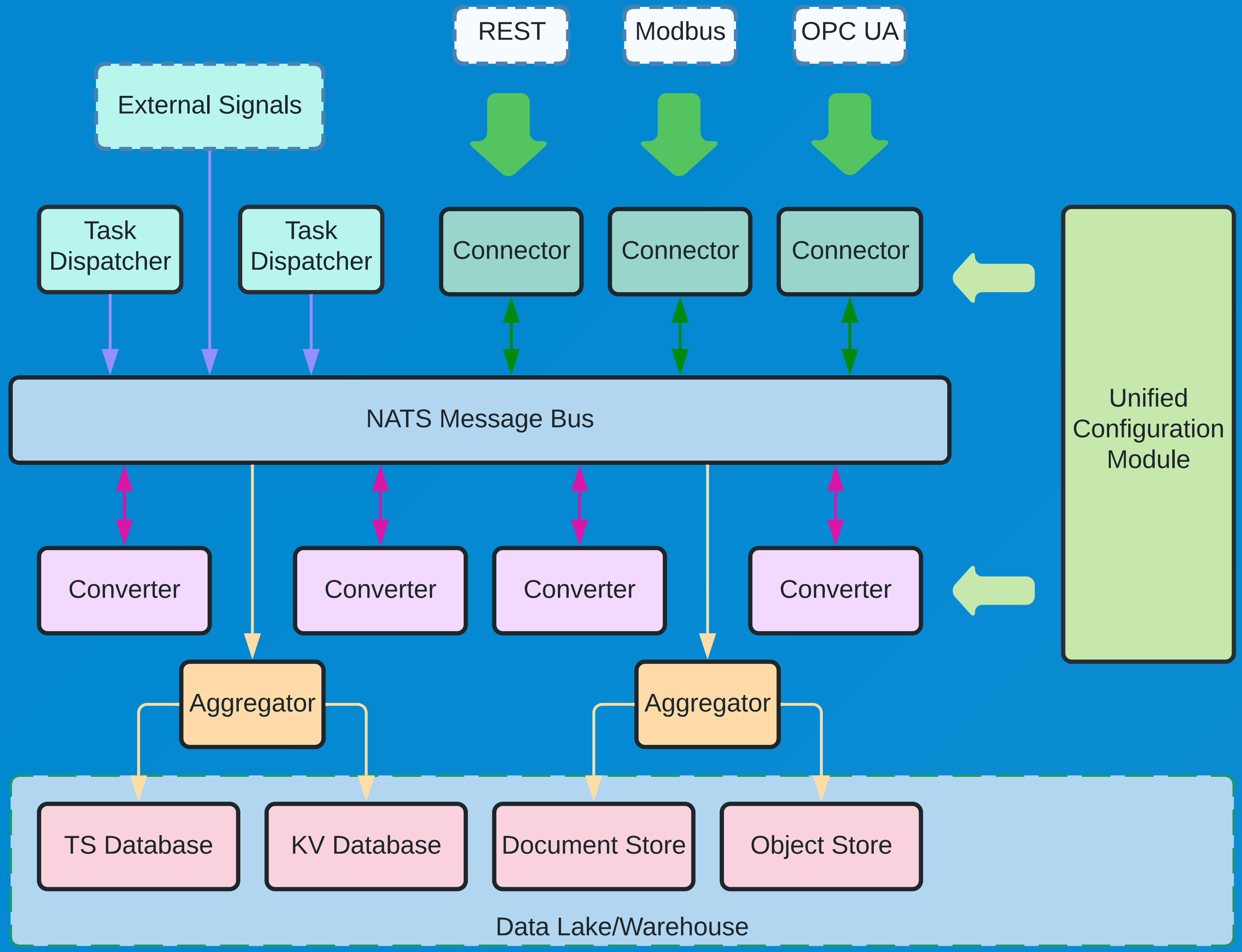
Karez: the Data Collection Framework

Motivations & Objectives

- Modularity
  - Different parts can be developed by different parties
- Pluggable
  - Can be stripped and customised easily
- Language / Platform-agnostic
  - Adapts to various industry environments
- Performance & Availability
  - Millions of data points per second
- Easy to use

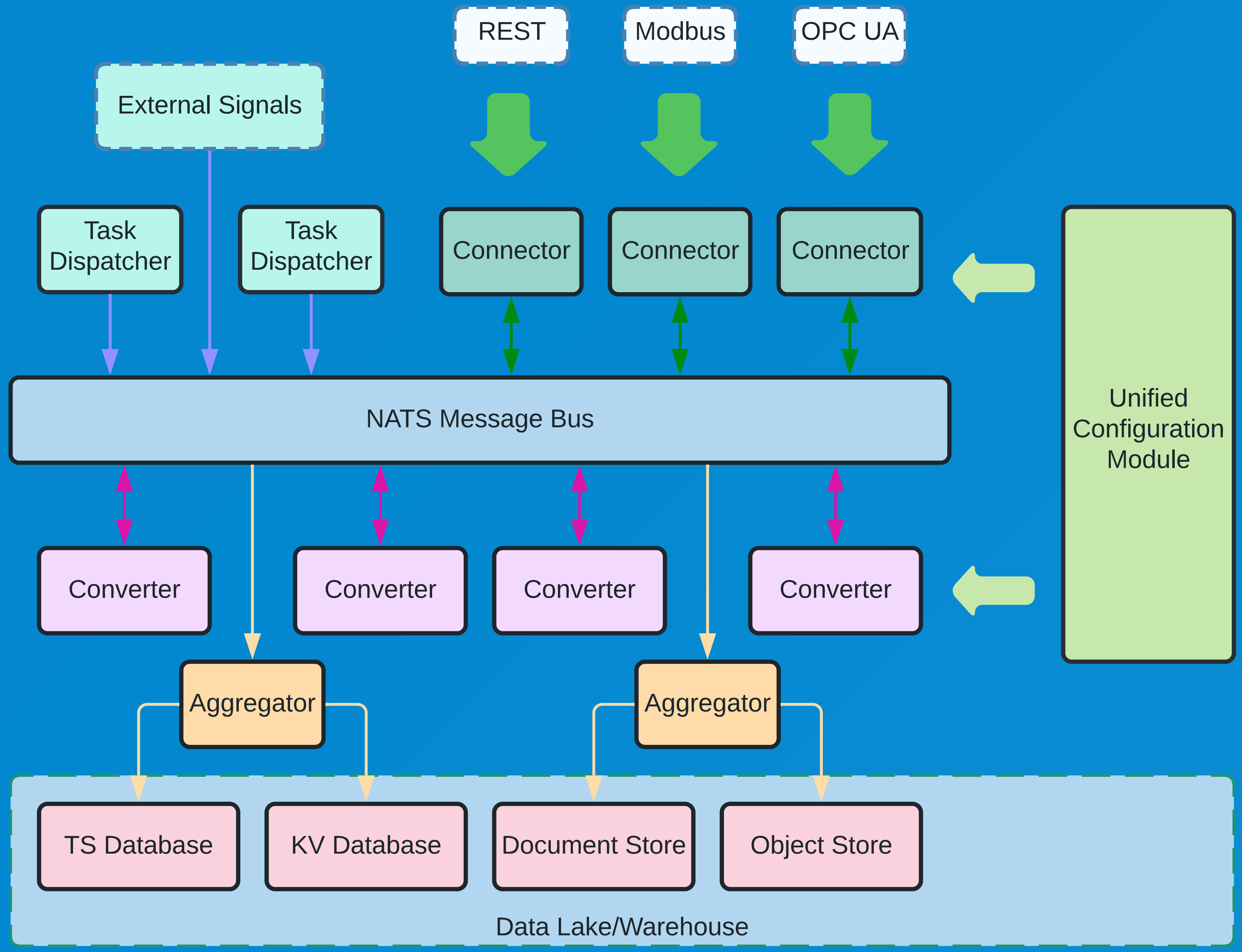
Architecture

1. Message Bus
2. Pluggable roles:
    1. Dispatcher
    2. Connector
    3. Converter
3. Aggregator
4. Configuration



Architecture

1. **Message Bus**
2. Pluggable roles:
    1. Dispatcher
    2. Connector
    3. Converter
3. Aggregator
4. Configuration



Components: Message Bus

Why using a Message Bus?

- Pluggable
- Language / Platform-agnostic
- Scalability

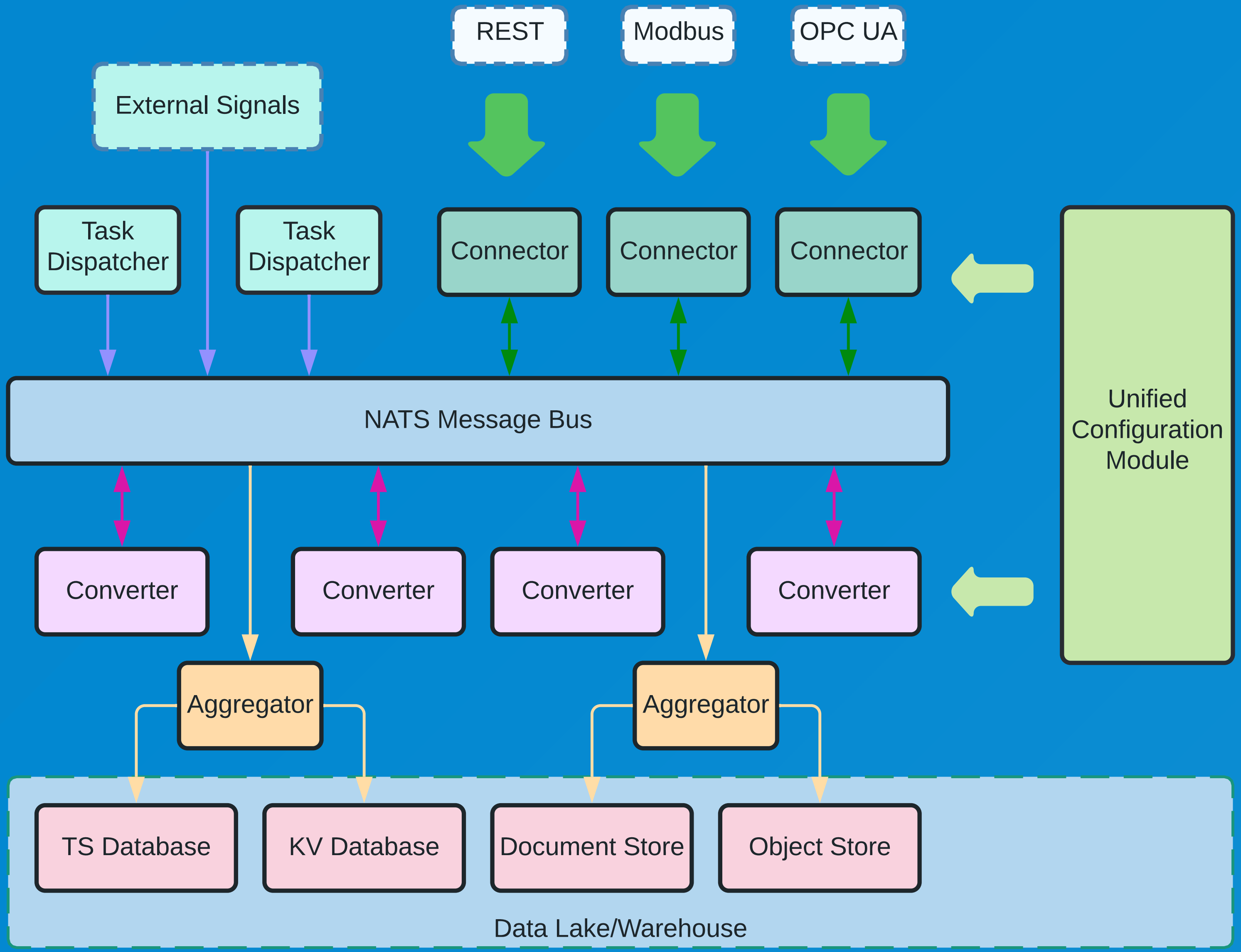
Components: Message Bus

*Why using **NATS**?*

- Small footage => *may need to be deployed on edge devices*
- High performance
- High availablity
- Security, AuthN & AuthZ
- Built-in multi-tenancy support

Architecture

1. Message Bus
2. **Pluggable roles**:
    1. Dispatcher
    2. Connector
    3. Converter
3. Aggregator
4. Configuration



Components: Pluggable roles

Roles:

- **Dispatcher**: Partitioning and dispatching tasks
- **Connector**: Connecting to data sources using certain protocols
- **Converter**: Transforming data formats & post processing

Notes:

- The roles can be implemented by extending Python base class, or using any other languages.
- Also, they can be distributedly deployed.

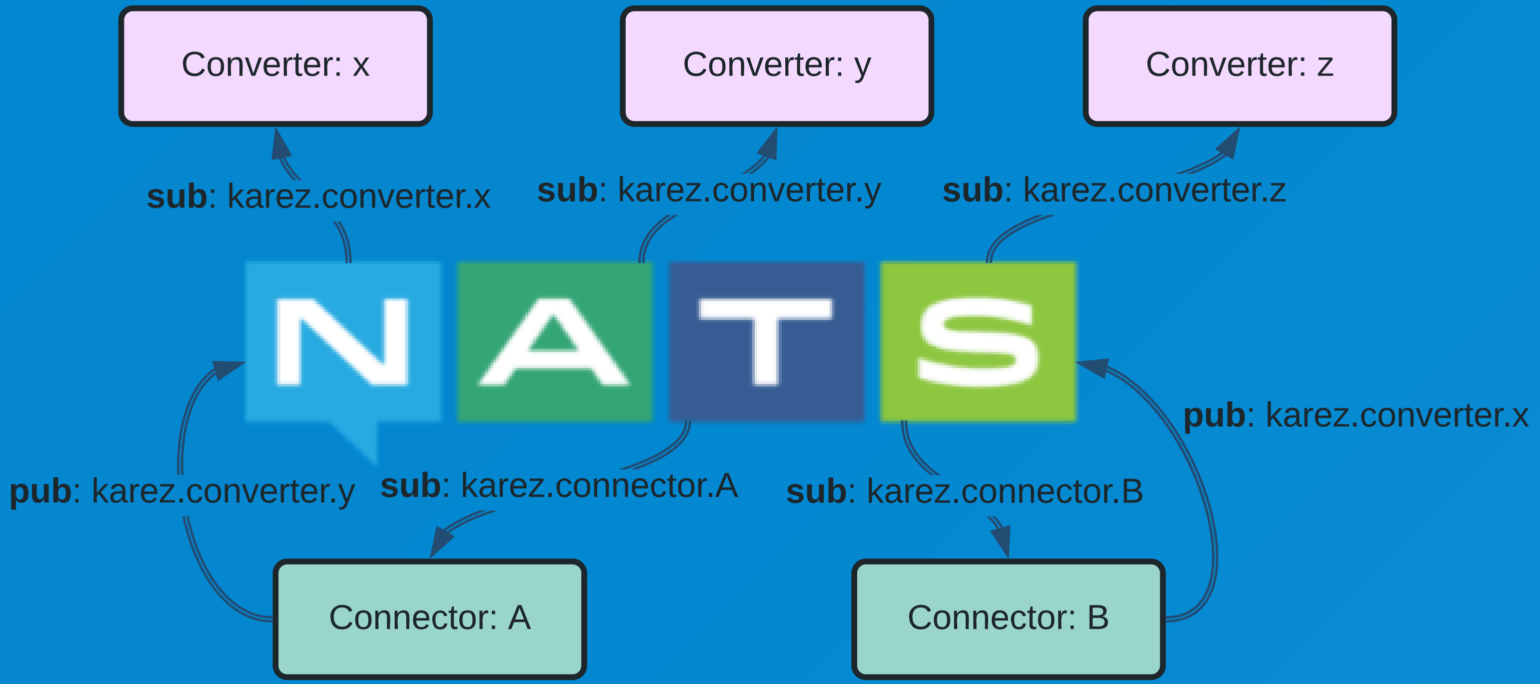
Components: Pluggable roles

Mechanism:

- Every role (except aggregators) listens on topic `karez.{role type}.{role name}`.
- If multiple roles have the same name, a message will only be send to one of them.
    - So it is scalable when necessary.
- Having completed its job, a role sends the result to another downstreaming role.

Components: Pluggable roles

Mechanism:



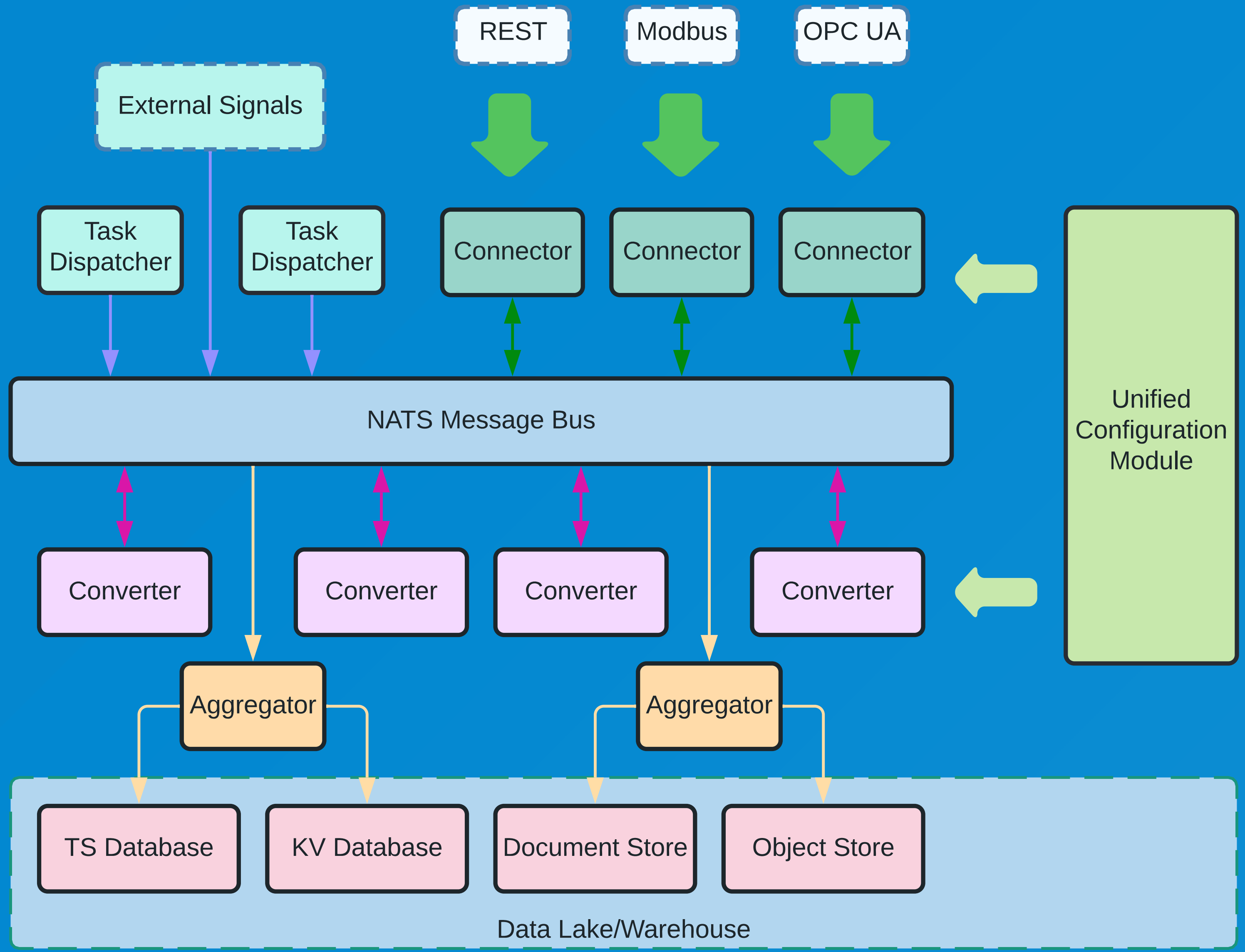
Components: Pluggable roles

Why *connectors and converters?*

- Improve the performance of connectors
  - IO intensive VS. CPU intensive
- Resue converters
- Finer controls of scalable resorces
  - M connectors : N converters

Architecture

1. Message Bus
2. Pluggable roles:
    1. Dispatcher
    2. Connector
    3. Converter
3. **Aggregator**
4. Configuration

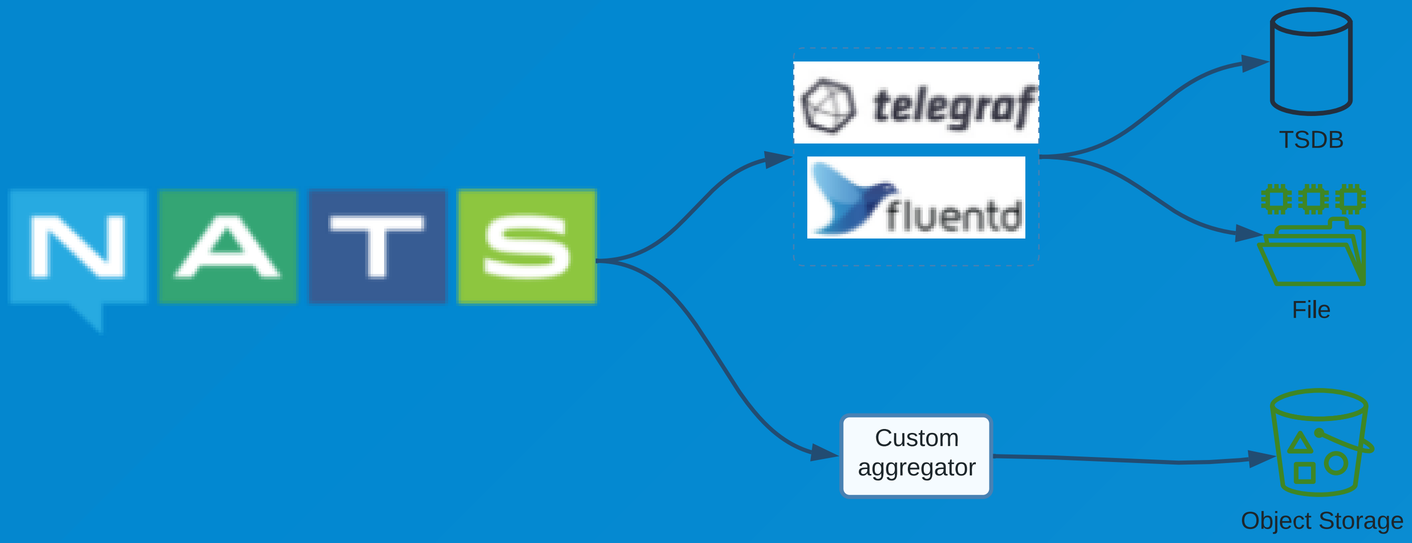


Components: Aggregator

Collecting data from the msg bus & send them to particular storages

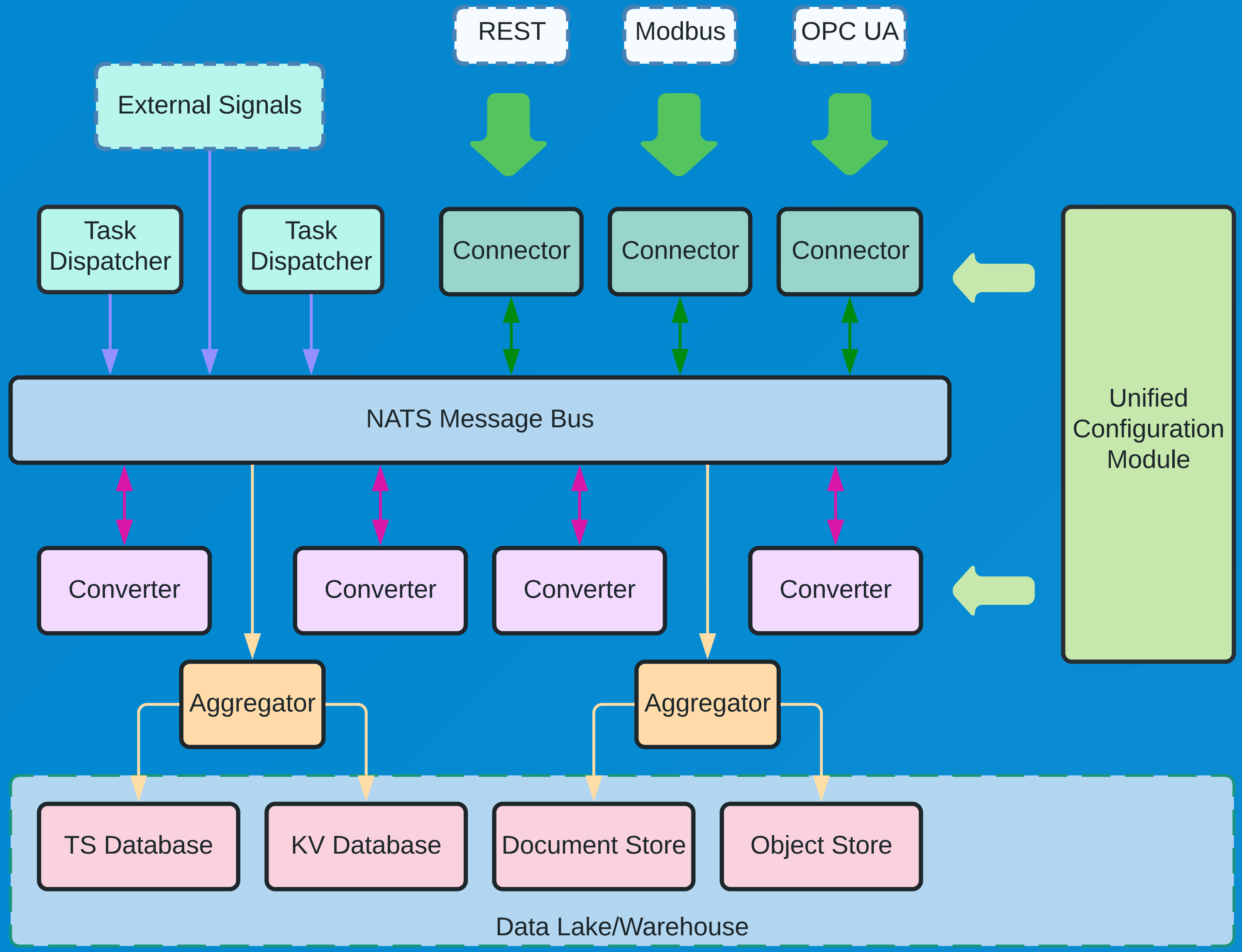
Usually ransparent to users

Can using existing solutions like telegraf or fluentd



Architecture

1. Message Bus
2. Pluggable roles:
    1. Dispatcher
    2. Connector
    3. Converter
3. Aggregator
4. **Configuration**



Configuration Framework

```
class Converter(ConverterBase):
    @classmethod
    def role_description(cls):
        return "Converter to format time-series points."

    @classmethod
    def config_entities(cls):
        yield from super(Converter, cls).config_entities()
        yield ConfigEntity(name="measurement",
                           description="Key name to be used as measurement name in TSDB.")
        yield OptionalConfigEntity(name="field_name",
                                   default=None,
                                   description="Key name to be used as field name in TSDB.")
        yield OptionalConfigEntity(name="field_value",
                                   default="value",
                                   description="Key name to be used as value in TSDB.")
```

Configuration Framework

A CLI tool is provided to query config options.

```
$ karez config converter fmt_ts_point -p ../plugins/
[converter] fmt_ts_point: Converter to format time-series points.
Configuration Options:
     - type: Type of the plugin.
     - measurement: Key name to be used as measurement name in TSDB.
     - name: [Optional] Name of the plugin.
        default: Same as type
     - field_name: [Optional] Key name to be used as field name in TSDB.
        default: None
     - field_value: [Optional] Key name to be used as value in TSDB.
        default: value
```

How to use

Steps:

1. (Optional) Write or extend a connector
2. (Optional) Write some converters
3. Write configuration files
4. Deploy

Example: fetching data from opc-ua servers

Writing Plugins

1. Connector: OPC-UA

```
class OPCUAPullConnector(PullConnectorBase):
    async def fetch_data(self, client: Client, entities):
        nodes = [client.get_node(node_id) for node_id in entities]
        data = []
        values = await client.read_values(nodes)
        for node_id, value in zip(entities, values):
            if not isinstance(value, Number):
                value = str(value)
            data.append(dict(
                ma_id=node_id,
                value=value
            ))
        return data
```

Writing Plugin

2. Converter: fmt-ts-points

```
class Converter(ConverterBase):
    def convert(self, payload):
        payload["_measurement"] = self.config.measurement
        field_name = self.config.field_name
        field_value = self.config.field_value
        if field_name:
            payload[payload[field_name]] = payload[field_value]
            del payload[field_name]
            del payload[field_value]
        return payload
```

Configuration

Use the CLI tool to help configure

```
$ karez config converter fmt_ts_point -p ../plugins/
[converter] fmt_ts_point: Converter to format time-series points.
Configuration Options:
      - type: Type of the plugin.
      - measurement: Key name to be used as measurement name in TSDB.
      - name: [Optional] Name of the plugin.
         default: Same as type
      - field_name: [Optional] Key name to be used as field name in TSDB.
         default: None
      - field_value: [Optional] Key name to be used as value in TSDB.
         default: value
```

Configuration

One or several configuration files in YAML, TOML, JSON or Python.

```yaml
dispatchers:
  - type: default
    connector: opcua_conn
    batch_size: 100
    interval: 10
    entity_file: config/opcua_points.json
```

```yaml
connectors:
  - name: opcua_conn
    type: opcua
    url: opc.tcp://opcuaserver.com:48010
    converter:
      - fix_timestamp
      - fmt_ts_point
```

```yaml
converters:
  - type: fix_timestamp
    tz_infos:
      SGT: Aisa/Singapore
  - type: fmt_ts_point
    measurement: opcua_ma
    field_name: ma_name
    field_value: value
```

Deployment

Option 1

All in one (for testing or simple scenario)

```
$ karez deploy -c config/opcua.yaml -p ../plugins/ -l INFO
INFO:root:Configurations: [PosixPath('config/opcua.yaml')].
INFO:root:NATS address: nats://localhost:4222.
INFO:root:Launched 2 Converters.
INFO:root:Launched 1 Connector.
INFO:root:Launched 1 Dispatcher.
```

Deployment

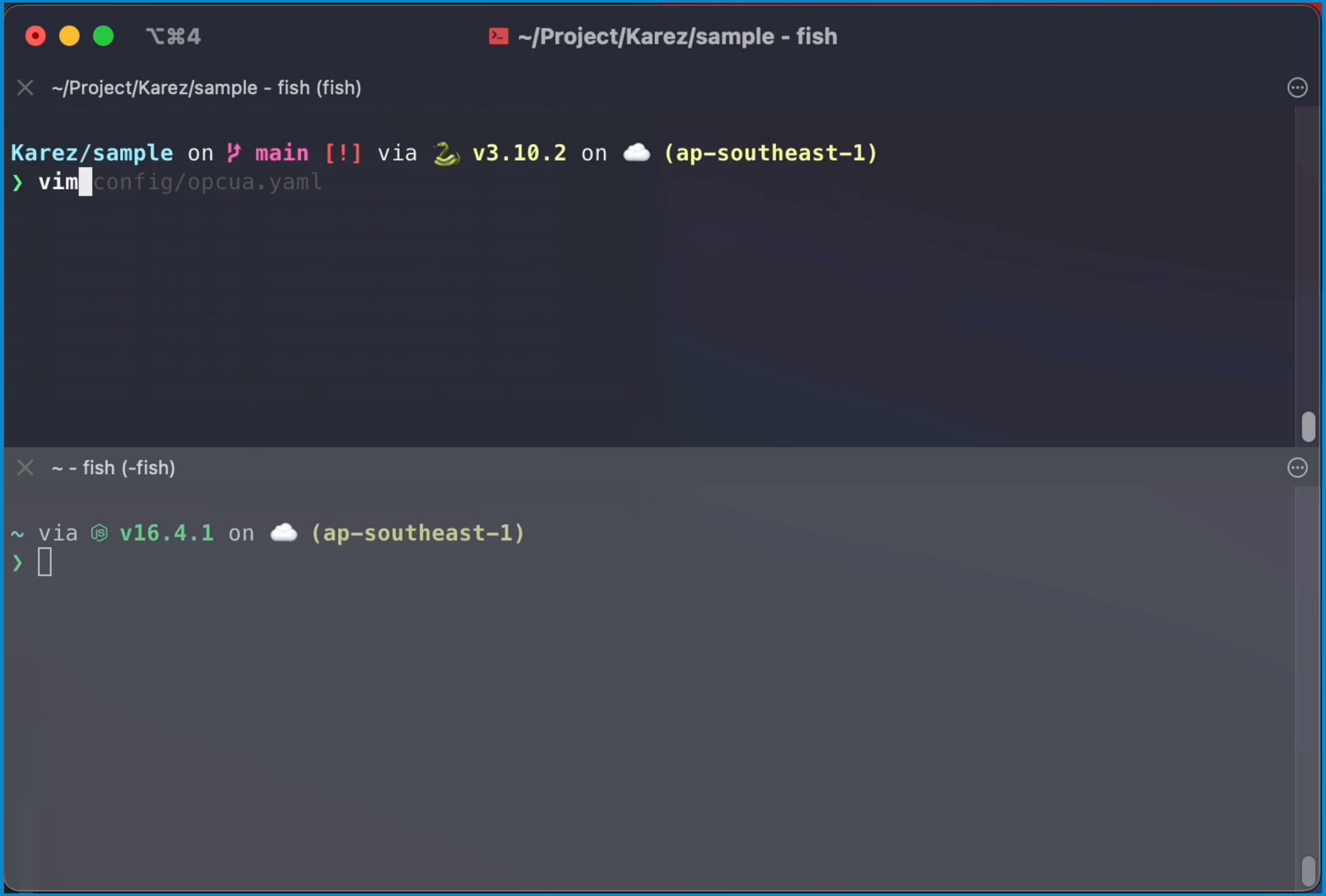
Option 2

Using docker compose or other container orchestration platforms

```
docker compose up -d --scale karze-connectors=2
[+] Running 8/8
 ⠿ Network deploy_default               Created                              0.2s
 ⠿ Container deploy-karze-converters-1   Started                              4.1s
 ⠿ Container deploy-karze-dispatchers-1  Started                              4.9s
 ⠿ Container deploy-karze-connectors-2   Started                              3.5s
 ⠿ Container deploy-storage-influxdb-1   Started                              2.9s
 ⠿ Container deploy-telegraf-1           St...                                4.8s
 ⠿ Container deploy-nats-server-1        Started                              2.8s
 ⠿ Container deploy-karze-connectors-1   Started                              3.7s
```

Checking Outputs

```
$ nats sub "karez.telemetry.>"
23:22:27 Subscribing on karez.telemetry.>
[#1] Received on "karez.telemetry.opcua_conn"
{
    "ma_id": "ns=3;s=AirConditioner_1.State",
    "dev_name": "AirConditioner_1",
    "dev_type": "AirConditioner_1",
    "dev_id": "AirConditioner_1",
    "timestamp": 1648538550.276894,
    "_measurement": "opcua_ma",
    "State": 1
}

[#2] Received on "karez.telemetry.opcua_conn"
......
```


⌥⌘4
~/Project/Karez/sample - fish
~/Project/Karez/sample - fish (fish)
Karez/sample on main [!] via v3.10.2 on (ap-southeast-1)
❯ vim config/opcua.yaml
~ - fish (-fish)
~ via v16.4.1 on (ap-southeast-1)
❯

Next Steps

1. Benchmarking
2. Docs & tests
3. More plugins
4. Integration with the platforms

Utinni: The Data Toolkit

Initial Motivations

1. **Abstraction of the InfluxDB query interface**
   - So don't need to write *similar* FLUX queries everywhere
2. **Extraction of common data post-processing procedures**
   - Data interpolation, time-zone correction, etc.
3. **Central management of the metric/indicator definitions**
   - Lazy Evaluation
   - Pandas

First version: dcwiz-tscc (time-series column calculator)

Howerver...

- More and more **data types** (not only time-series data)
- More and more **data sources** (influxdb, mocked data, local file, etc.)
- More and more **complicated operations** (not only op along rows)
- More and more **use scenarios** (Low-code/no-code, etc.)

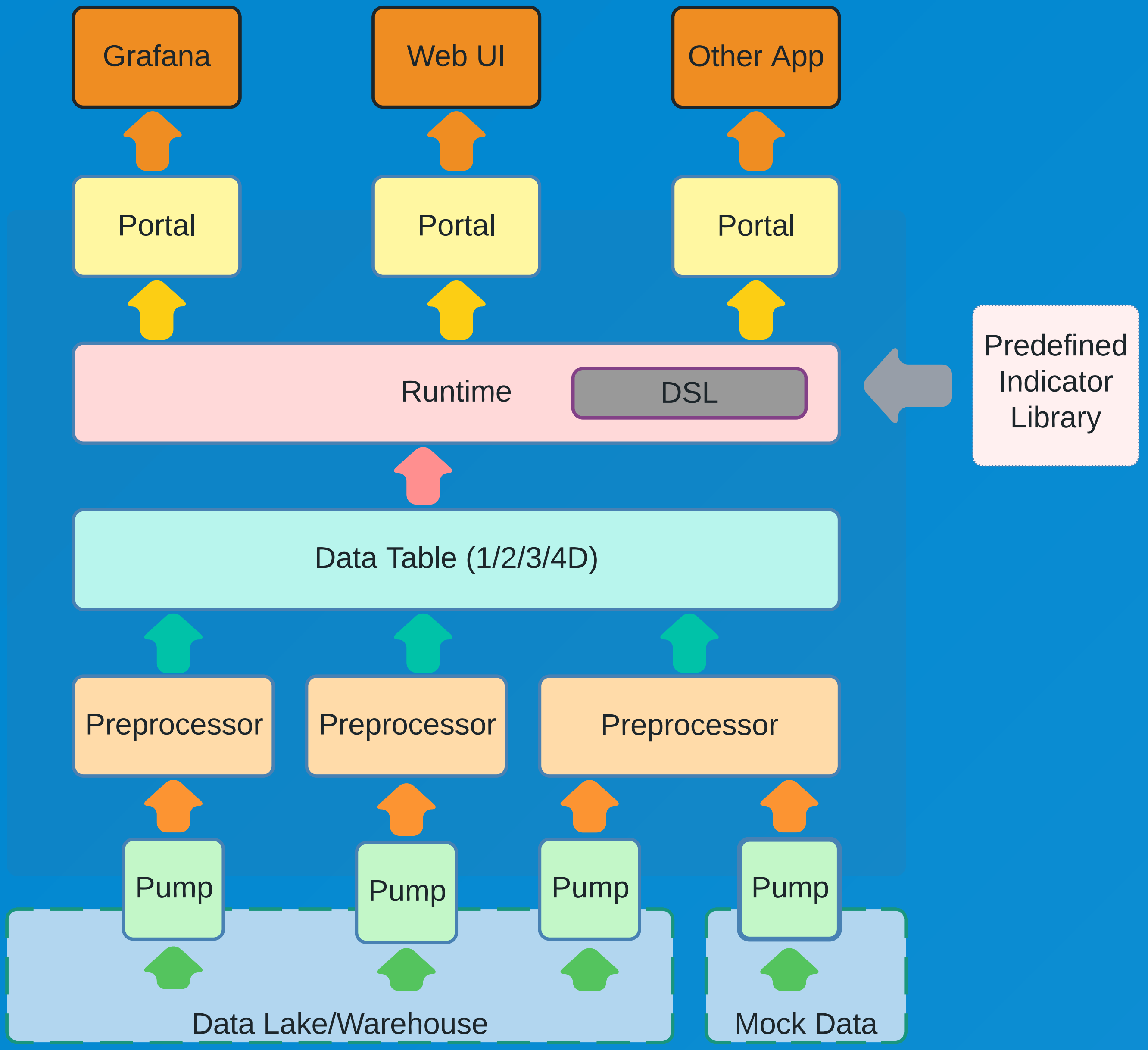
Features

Complete toolchain from data lake to applications

1. Abstraction
2. Optimisation
3. Preparation
4. Transformation
5. Providing

Architecture

1. Data Pumps
2. Preprocessors
3. Table
4. Runtime
   - Transforming data
   - Managing definitions
5. Portals
   - AI/DT engines
   - LCNC platforms



Data Pump

To deal with **storage heterogeneity.**

```
from utinni.pump import ConstantTSDataPump, \
                        RandomTSDataPump, \
                        WrappedDataPump, \
                        InfluxDBDataPump
from utinni.pump.extensions import DCWizTelemetryPump

context.add_pump("tsdb", InfluxDBDataPump(**conf.influxdb)) \
       .add_pump("dc", DCWizTelemetryPump(**conf.influxdb)) \
       .add_pump("const", ConstantTSDataPump()) \
       .add_pump("rand", RandomTSDataPump()) \
       .add_pump("wrap", WrappedDataPump())
```

Data Pump

Abstraction: to provide unified data access/generation interfaces

```
rack_table_md = context.tsdb_table(column="dev_name", dev_type="Sensor")
ups_table_md = context.dc_pump["UPS"]

crac_info = dict(crac_names = [f"ACCPU 4-{i}" for i in range(1, 6)])

crac_power_md = context.const_table(5.26, fields=crac_names) * 1000
crac_supply_air_flow_rate_md = context.const_table(25100, *crac_info)
crac_supply_temperature_md = context.rand_table("normal",
    rand_args=dict(loc=12.0, scale=1.0), *crac_info)
crac_return_temperature_md = context.rand_table("normal",
    rand_args=dict(loc=20.0, scale=1.0), *crac_info)

wrap_table = context.wrap_table(pd.Series([1,2,3,4]))
```

Preprocessor

- To deal with **data heterogeneity.**
- **Prepare** the data

Example:

For time-series data, the preprocessor needs to do

- Data interpolation
- Data resampling & reshaping
- Handling timezone issues
- ……

Preprocessor

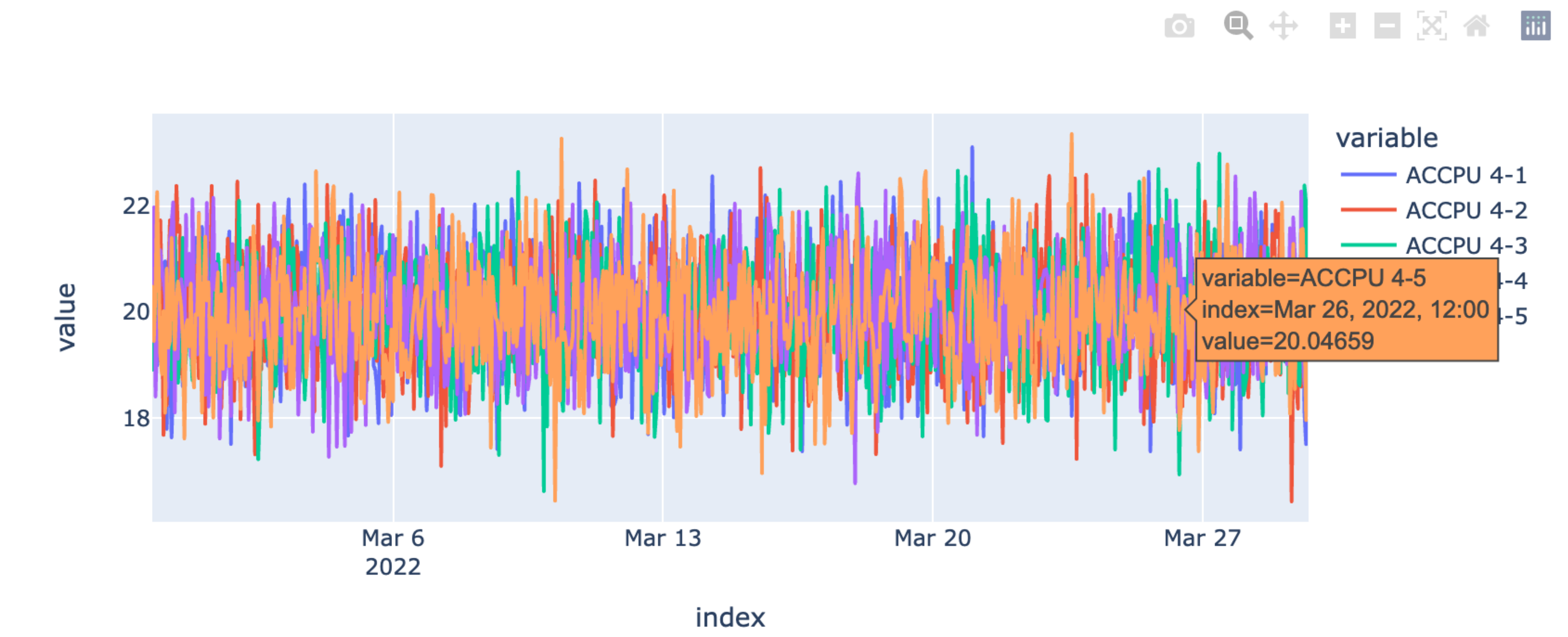
```
context.bind(start=-timedelta(hours=4),
             step=timedelta(minutes=60))

crac_return_temperature_md.value
```

	ACCPU 4-1	ACCPU 4-2	ACCPU 4-3	ACCPU 4-4	ACCPU 4-5
2022-03-29 13:00:00+00:00	20.288349	20.672196	19.747211	21.462398	21.241952
2022-03-29 14:00:00+00:00	20.265066	19.037901	19.763592	19.919092	21.999154
2022-03-29 15:00:00+00:00	21.210437	21.841045	21.038708	18.893409	18.599474
2022-03-29 16:00:00+00:00	18.793384	18.912498	19.676935	19.777637	18.602298
2022-03-29 17:00:00+00:00	20.116312	20.774355	21.149603	20.709414	19.576563

Notes: late binding & lazy evaluation

Preprocessor

```
crac_return_temperature_md.plot()
```



Use any tool in Python data science ecosystem!

Table

The unified / core data structure. It wraps

- 1D: a single value (`int`, `float`, `str`, etc.)
- 2D: `pandas.Series`
- 3D: `pandas.DataFrame`
- 4D: `dict[str, pandas.DataFrame]`

xD => (x+1)D: `ascent` / `aggregate`
(x+1)D => xD: `extract`

All the existing methods on `Series`, `DataFrame` ... can also be applied

Runtime

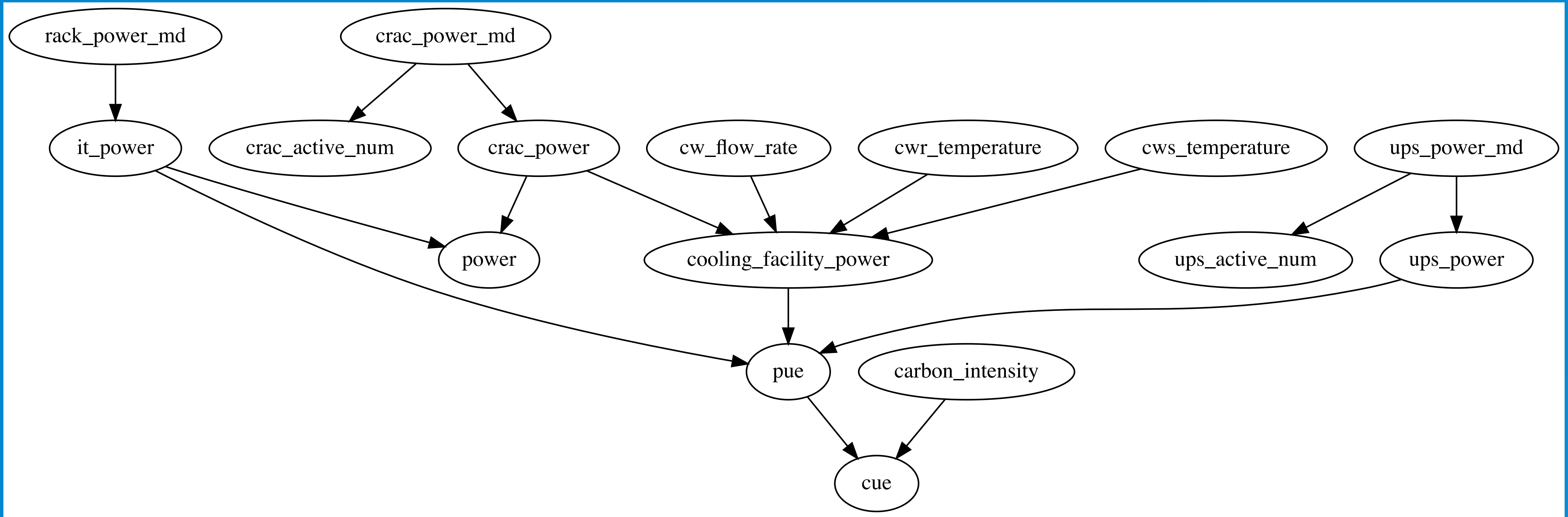
- Define and perform `table` **transformations**
- **Optimise** the data fetching procedures
- Lazy evaluation applied
- A tiny DSL (Domain-Specific Language) to support **dynamic evaluation**

Runtime

- Define and perform `table` **transformations**
  - **Purpose**: shaping the data to fit applications' needs
  - Use data from different data sources
  - Operations:
    - Arithmetic Calculations (1/2/3D)
    - Pandas methods (2/3D)
    - Self defined functions/lambdas (all)
    - Extract, aggregate, concat, ascent, filter ...

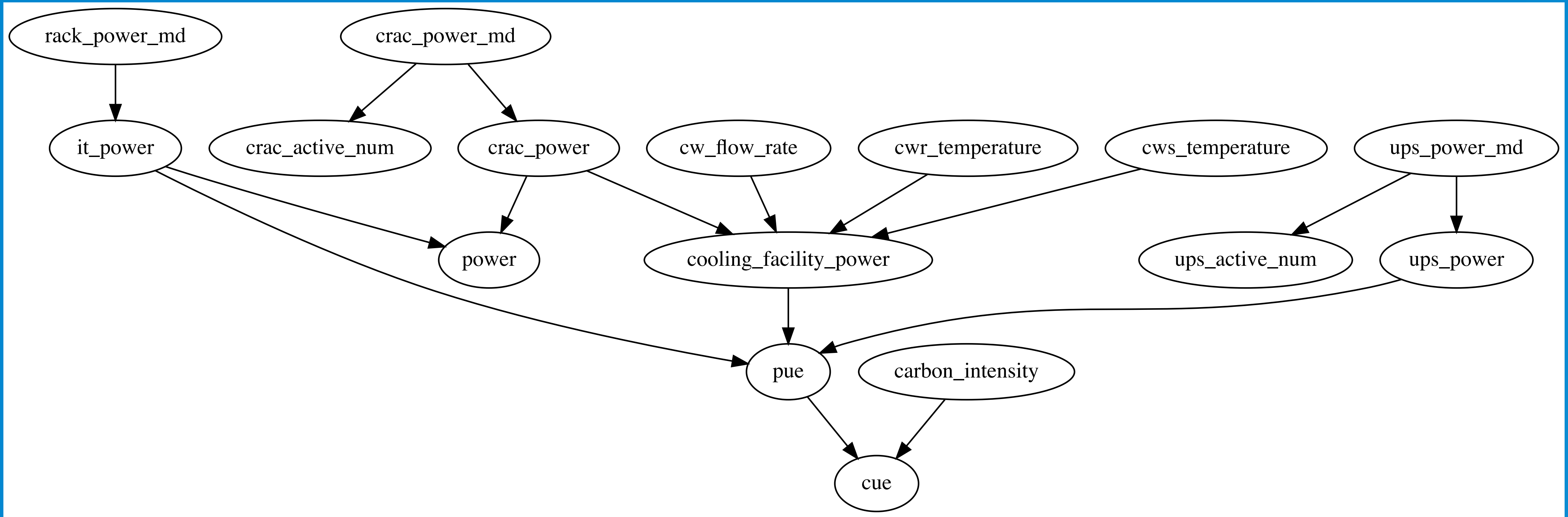
```
cws_temperature = context.const_table(8)
cwr_temperature = context.const_table(13)
cw_flow_rate = context.const_table(4.25 * 5)

air_flow_rate = crac_supply_air_flow_rate_md.sum(axis=1)
crac_power = crac_power_md.sum(axis=1)

_cooling_water_power = cw_flow_rate * (cwr_temperature - cws_temperature) * 4.18 / 5.5
cooling_facility_power = crac_power + _cooling_water_power

crac_active_num = crac_power_md.apply_tf(lambda x: x > 100).astype(int).sum()
ups_active_num = ups_power_md.apply_tf(lambda x: x > 100).astype(int).sum()

it_power = rack_power_md.sum(axis=1)
ups_power = ups_power_md.sum(axis=1)
power = it_power + crac_power
_lighting_power = context.const_table(0)
pue = (it_power + cooling_facility_power + _lighting_power + ups_power) / it_power

carbon_intensity = context.rand_table("normal", rand_args=dict(loc=0.20, scale=0.005))
cue = pue * carbon_intensity
```

Runtime



- **Optimise** the data fetching procedures
  - Only needed data will be retrieved

Runtime



- Lazy Evaluation => **Central management of transformations**

Runtime

- Dynamic evalutaion
  - Not using the `eval` function => **Not safe!**
  - Instead, using `pyparsing` to define and implemente a tiny DSL
  - **Purpose**: support low-code platform that allow users to define their own formulas.
  - *Still In Progress:* to support more features

```
context.parse("(it_power + cooling_facility_power) / it_power")
```

Portals

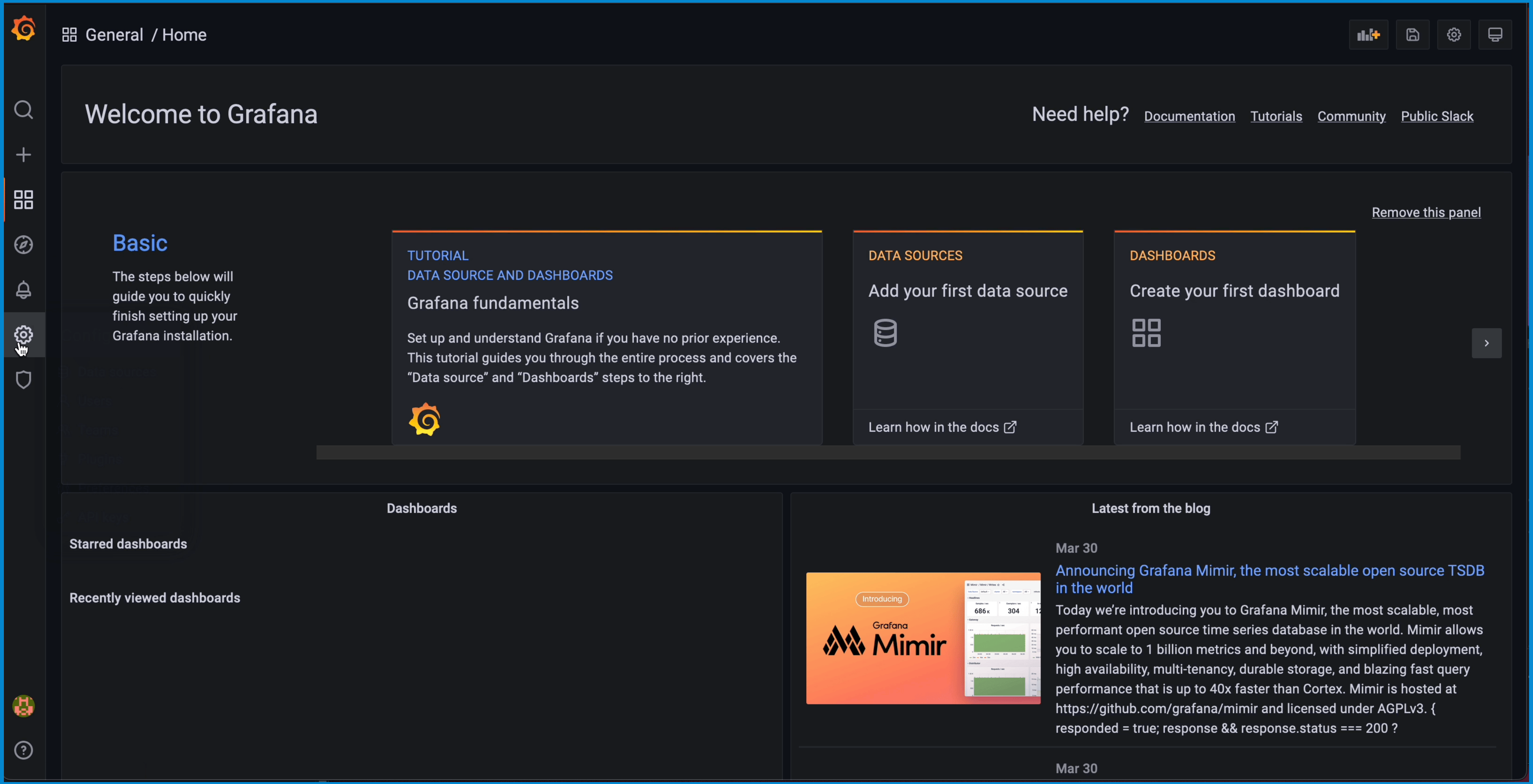
- **Providing** the data to different apps

Example: Grafana Data Source

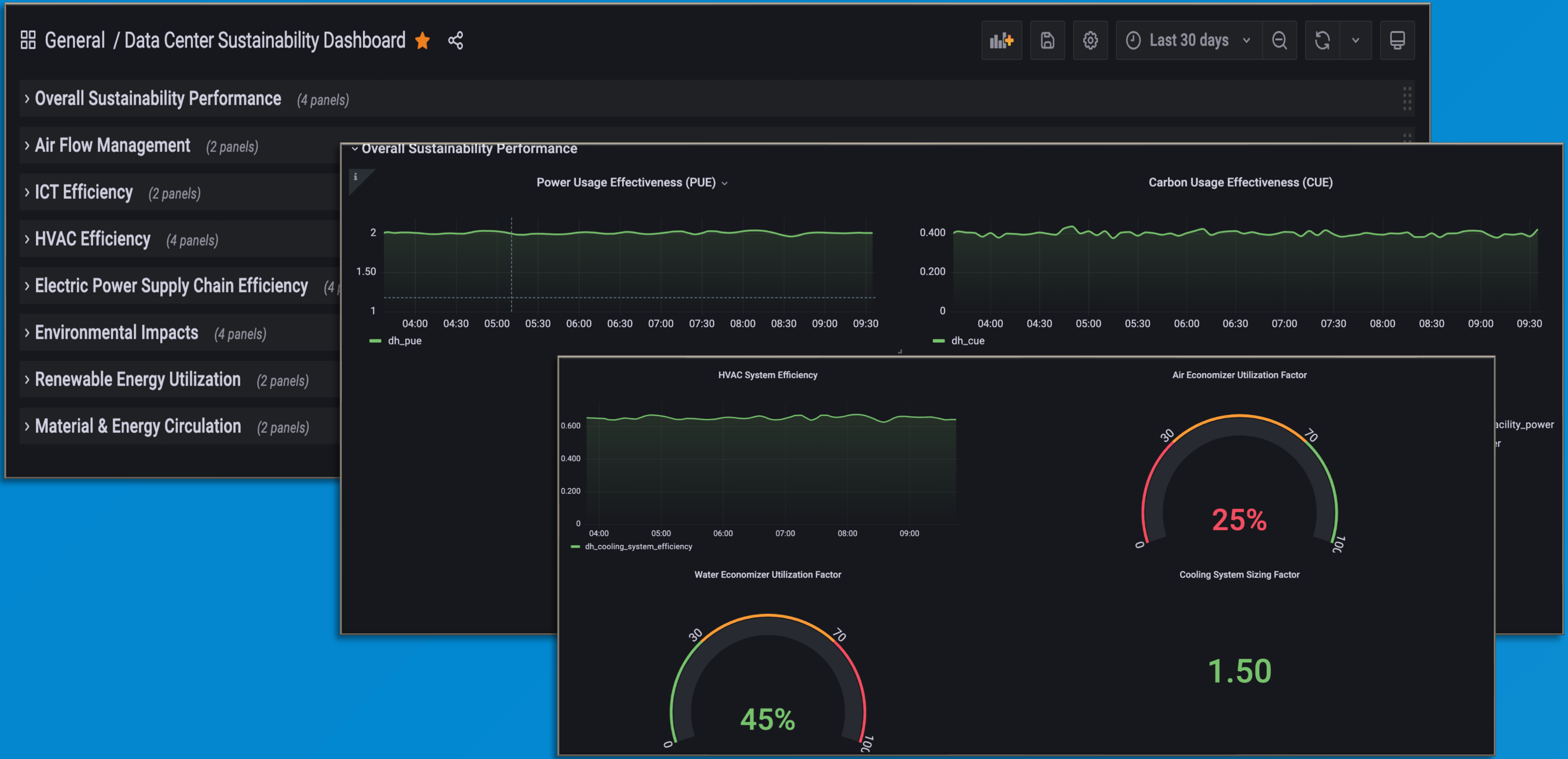
```
$ utinni-grafana-ds grafana/cookbook:demo
INFO:     Started server process [17496]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
```

A CLI tool has been provided.

Grafana


General / Home
Welcome to Grafana
Need help? Documentation Tutorials Community Public Slack
Remove this panel
Basic
The steps below will guide you to quickly finish setting up your Grafana installation.
TUTORIAL
DATA SOURCE AND DASHBOARDS
Grafana fundamentals
Set up and understand Grafana if you have no prior experience. This tutorial guides you through the entire process and covers the "Data source" and "Dashboards" steps to the right.
DATA SOURCES
Add your first data source
Learn how in the docs
DASHBOARDS
Create your first dashboard
Learn how in the docs
Dashboards
Starred dashboards
Recently viewed dashboards
Latest from the blog
Mar 30
Announcing Grafana Mimir, the most scalable open source TSDB in the world
Today we're introducing you to Grafana Mimir, the most scalable, most performant open source time series database in the world. Mimir allows you to scale to 1 billion metrics and beyond, with simplified deployment, high availability, multi-tenancy, durable storage, and blazing fast query performance that is up to 40x faster than Cortex. Mimir is hosted at https://github.com/grafana/mimir and licensed under AGPLv3. { responded = true; response && response.status === 200 ?
Introducing
Grafana Mimir
Mar 30

Grafana


General / Data Center Sustainability Dashboard
Last 30 days
Overall Sustainability Performance (4 panels)
Air Flow Management (2 panels)
ICT Efficiency (2 panels)
HVAC Efficiency (4 panels)
Electric Power Supply Chain Efficiency
Environmental Impacts (4 panels)
Renewable Energy Utilization (2 panels)
Material & Energy Circulation (2 panels)
Overall Sustainability Performance
Power Usage Effectiveness (PUE)
dh_pue
Carbon Usage Effectiveness (CUE)
dh_cue
HVAC System Efficiency
dh_cooling_system_efficiency
Air Economizer Utilization Factor
25%
Water Economizer Utilization Factor
45%
Cooling System Sizing Factor
1.50

Next Steps

- Protocols to write back to the data lake
- Benchmarking, docs and tests
- Integration with the DCWiz Platform
- Improve DSL
- How to providing models
  - Model definition format and tools

Summary

- **Karez**: pluggable, scalable data collection framework
- **Utinni**: toolkit bridging heterogeneous storages and applications

Roadmap

- Cloud native architecture (in progress)
  - Multi-cloud/hybrid-cloud
  - Container orchestration & Microservices
- Data ecosystem (in progress)
  - Inlet: Karez / Outlet: Utinni
  - Standard DC model format & toolchains
- Task management framewrok (todo)
- Security framework (todo)
- Asset and user management framework (todo)

Thank you!

Q & A

https://github.com/cap-dcwiz/Karez (Opensourced)
(https://cap-dcwiz.github.io/Karez/, under construction)

https://github.com/cap-dcwiz/Utinni