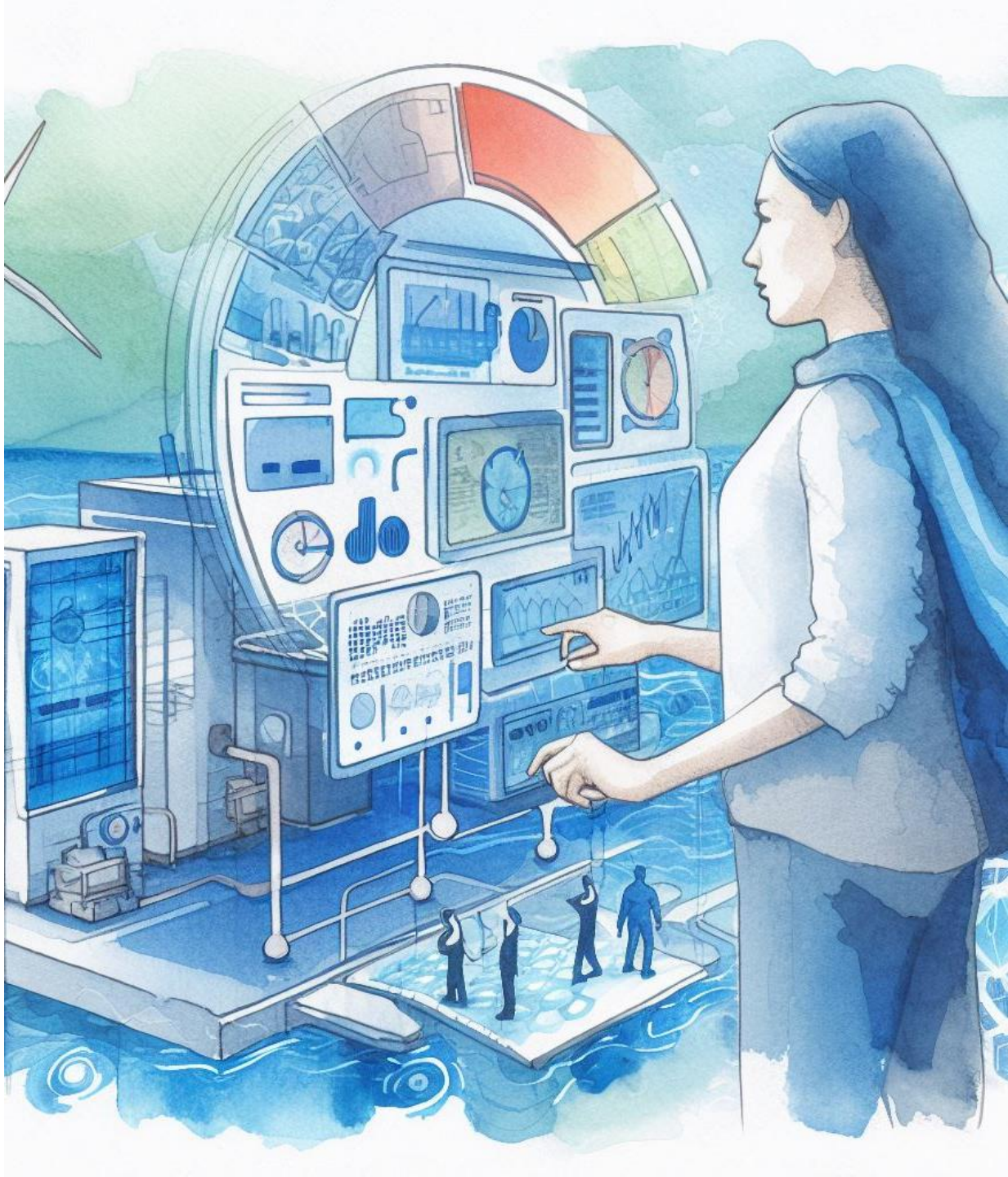
SuperSHOP

Nils Ræder

Trondheim / 2023-11-29

Contents

- Goals
- How we (plan to) do it
- Current state of things
- Lessons learned
- Examples



Goals

We want SHOP to make decisions on which we will perform automated trading and planning processes

We need

- Trust - to run SHOP with confidence
  - Consistent quality
  - Only good quality data enters production environments
- Reproducibility
  - To debug and improve
- Transparency
  - All input data are available to answer questions
  - Results easily available and comparable across time and model permutations
- Monitoring and alerting

In SHOP we trust!

- The tolerance for errors is very low once we use the results for automated trading, planning
- The system architecture, design and technologies must
  - Limit the potential for errors
  - Encourage data validation
  - Limit overall complexity

- A big source of confusion and errors can be *too many degrees of freedom*
- Sources of errors include
  - Unintended variation of software versions (SHOP, license, CPLEX, Python, python dependencies, operating system, …)
  - Ability to make changes that are not a part of a consistent whole

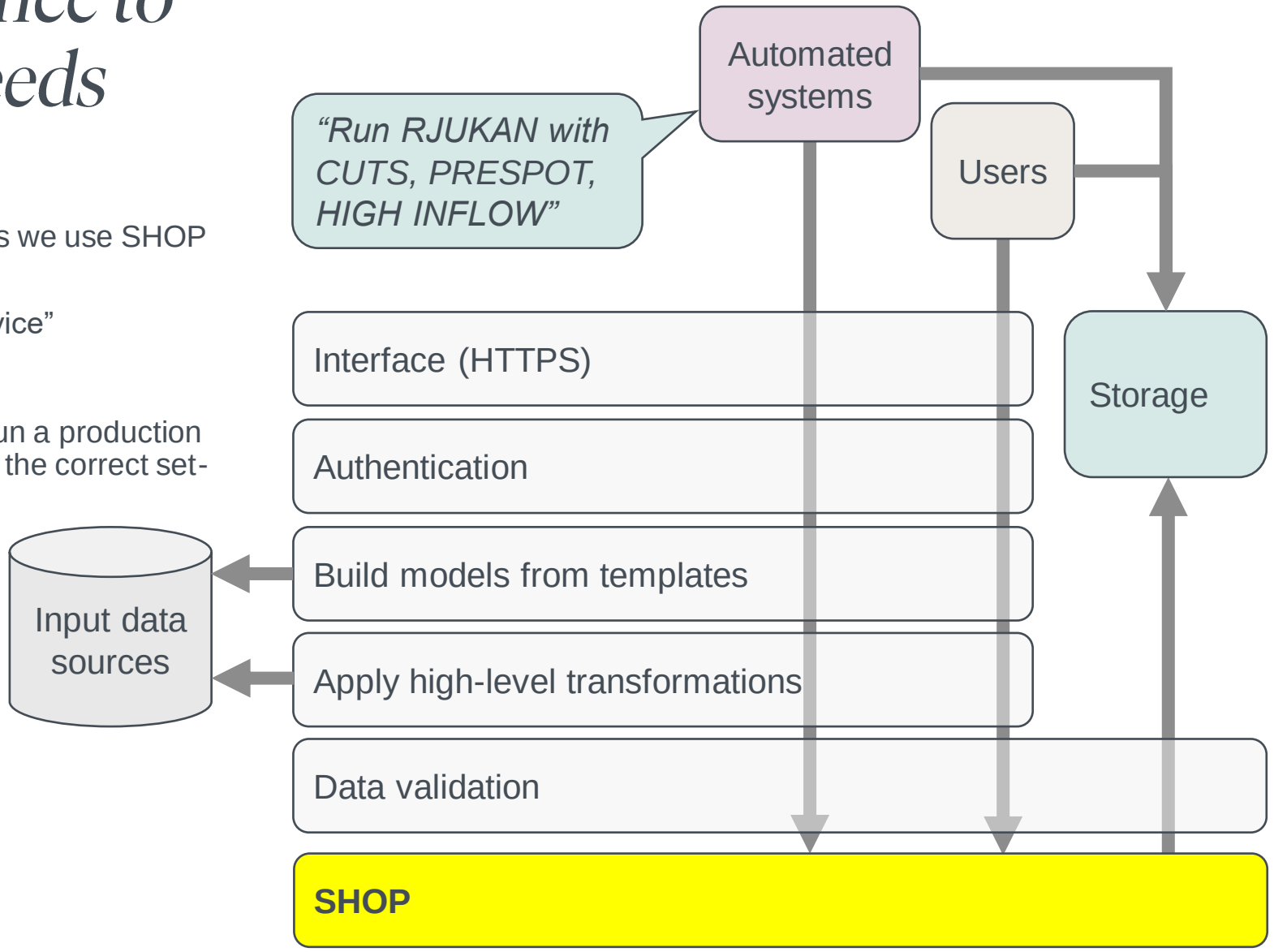
✓ *A single SHOP instance to serve all production needs*

✓ *High-level functionality for mutating models*

✓ *A single SHOP instance to serve all production needs*

- During development, testing and ad-hoc analyses we use SHOP any way we choose to
- In production setting we employ “SHOP as a service”

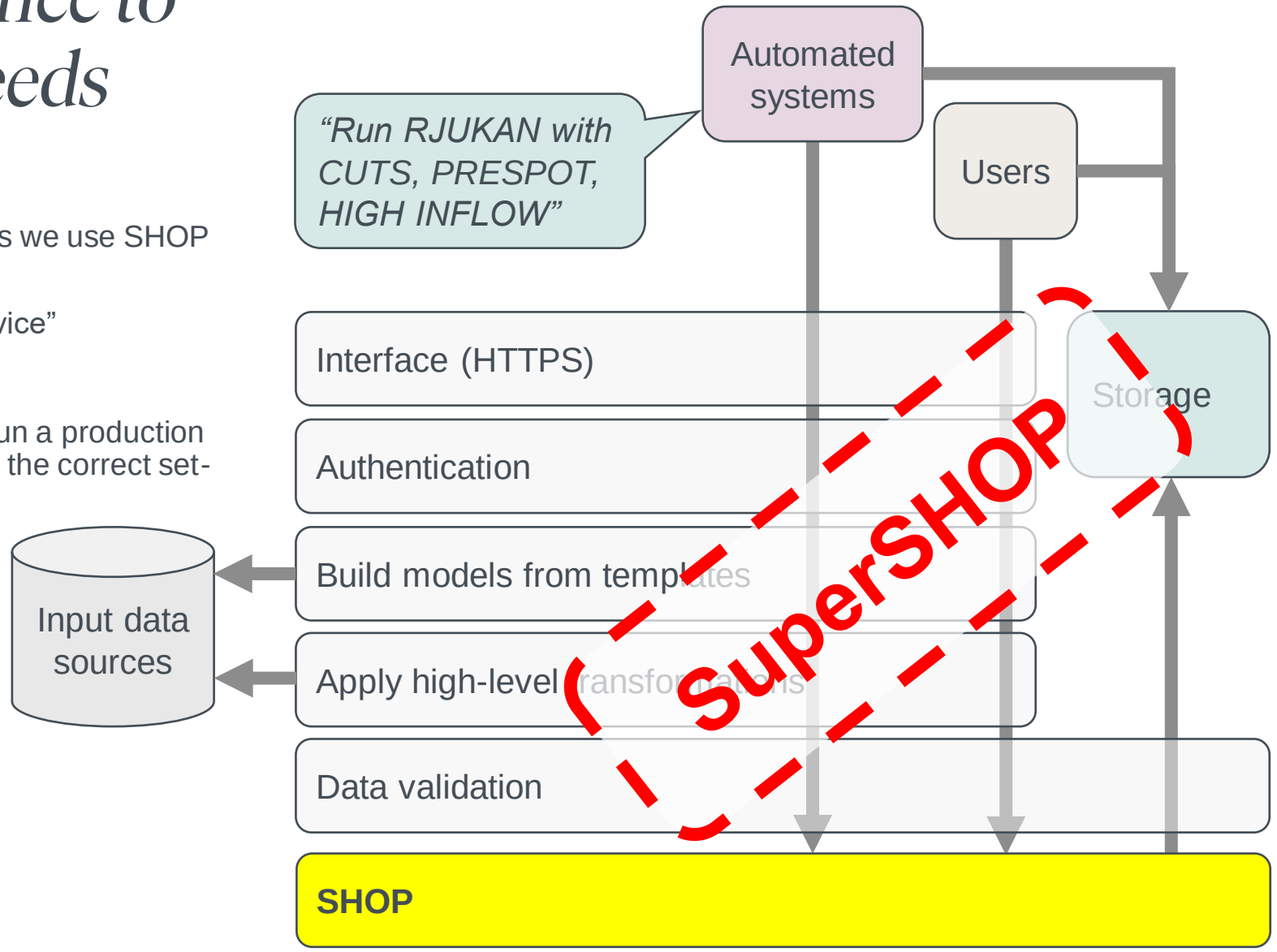
- Any automated system or person who wants to run a production grade optimization can call this service to ensure the correct set-up is used
- The service can be called to:
  - Build and execute a model
  - Build and return a model
  - Receive and execute a model



✓ *A single SHOP instance to serve all production needs*

- During development, testing and ad-hoc analyses we use SHOP any way we choose to
- In production setting we employ “SHOP as a service”

- Any automated system or person who wants to run a production grade optimization can call this service to ensure the correct set-up is used
- The service can be called to:
  - Build and execute a model
  - Build and return a model
  - Receive and execute a model

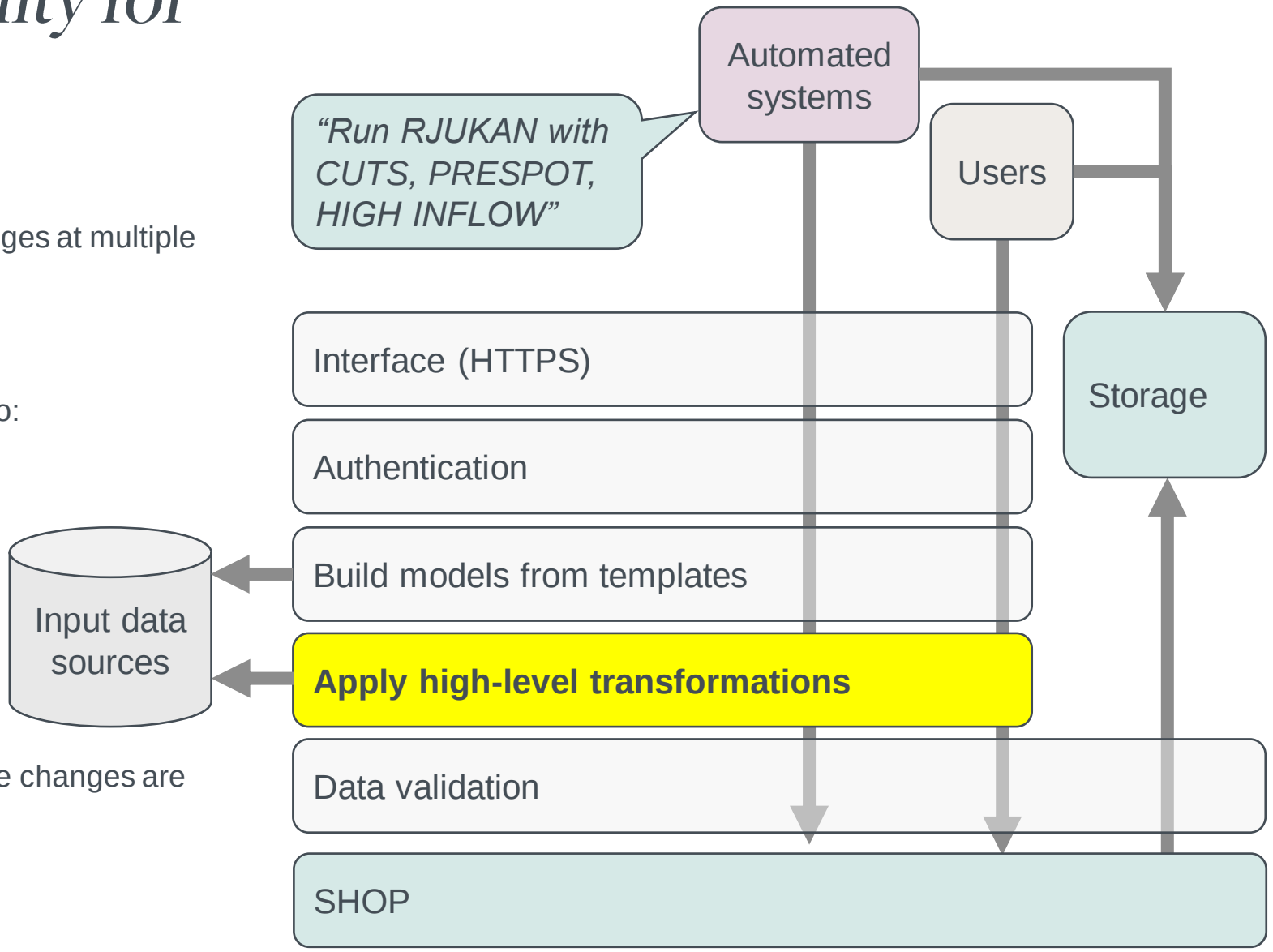


✓ *High-level functionality for mutating models*

- Meaningful updates to a model may require changes at multiple locations

- To go from a static `energy_value_input` to a `water_value_input` based on cuts we need to:
  - Remove `energy_value_input` from reservoirs
  - Add `water_value_input` to reservoirs
  - Create the `cut_groups`
  - Add `inflow_series` to the model
  - Connect `cut_groups` to reservoirs
  - Connect `inflow_series` to reservoirs
  - Connect `inflow_series` to `cut_groups`

- This is not too hard, but we want certainty that the changes are applied as a consistent whole

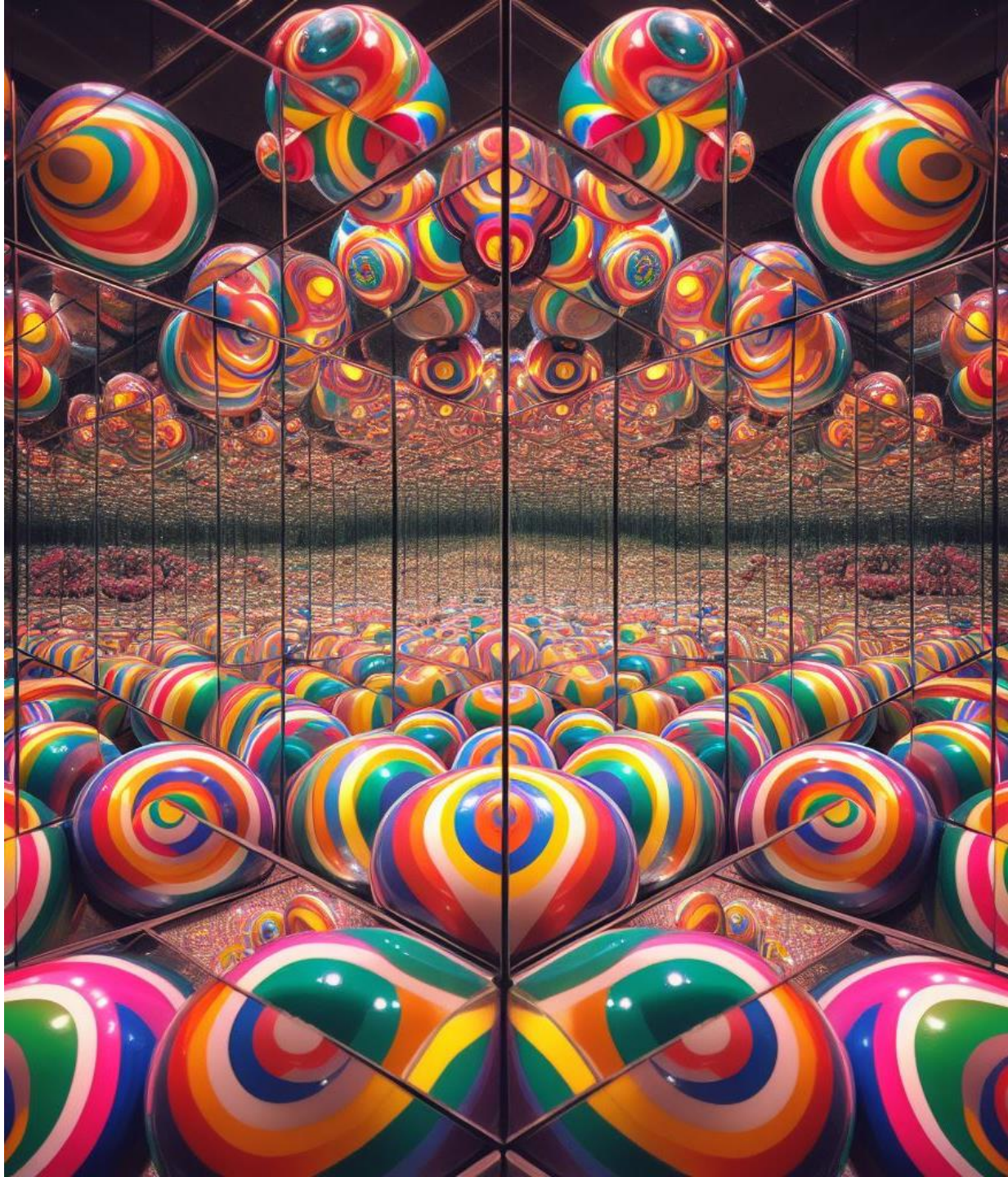


Reproducibility

- We want to reproduce SHOP cases for debugging, improvements, validation when changing SHOP version etc.

- We want to compare input data across time, and trace any problems back to the data sources

✓ *Save the input data*



✓ *Save the input data*

- We want the stored input data to represent what entered the SHOP kernel

- The interactive interface of `PySHOP` is great, but …
  - `PySHOP` transforms data
  - Inserting data using the `ModelBuilder` framework is a gradual additive process where order may be important

✓ *We build the complete input data before it ever enters SHOP*

✓ *We build the complete input data before it ever enters SHOP*

- We only need two function in PySHOP* ** *** ****
  - `ShopSession.load_yaml(…)`
  - `ShopSession.dump_yaml(…)`

** In reality we also use the `execute_full_command` and `get_messages` methods since we inspect the logs and results between each iteration of the optimization*

*** In the first versions of SuperSHOP we used the `ModelBuilder` interface to get and set data. The addition of a YAML spec has made the set up easier and less error prone.*

**** We also use the `GetAttributeInfo` and `GetObjectInfo` methods to build a pure python version of the SHOP data structure. It's about 20k lines of python models.*

***** We use an extensive set of PySHOP functionality for testing, debugging, development, …*

```yaml
model:
  reservoir:
    Reservoir1:
      max_vol: 12.0
      lrl: 90.0
      hrl: 100.0
      vol_head:
        ref: 0
        x:
        - 0.0
        - 12.0
        - 14.0
        y:
        - 90.0
        - 100.0
        - 101.0
      start_head: 92.0
      inflow:
        2023-03-31 00:00:00: 17.916153110203407
        2023-03-31 01:00:00: 15.942096263507963
        2023-03-31 02:00:00: 13.905825185115457
        2023-03-31 03:00:00: 12.15583466820391
        2023-03-31 04:00:00: 11.529288479311207
        2023-03-31 05:00:00: 10.193658631704283
        2023-03-31 06:00:00: 8.113891021832284
        2023-03-31 07:00:00: 5.977161940253971
        2023-03-31 08:00:00: 1.2621934711921958
        2023-03-31 09:00:00: 2.4394019663799114
        2023-03-31 10:00:00: 7.943245133292623
        2023-03-31 11:00:00: 12.388503361264817
        ...
      flow_descr:
        ref: 0
        x:
        - 100.0
        - 101.0
        y:
        - 0.0
        - 1000.0
      energy_value_input: 39.7
    Reservoir2:
  plant:
    Plant1:
      less_distribution_eps: 0.001
      ownership: 100.0
```

20k lines of python models? 😳

- Having a pure python model of the SHOP data structure allow us to maintain the first goal

  ✓*A single SHOP instance to serve all production needs*

- Client applications and end user applications have a working data model that is independent of PySHOP, SHOP, CPLEX, license files etc.
  - Easier to debug
  - Easier to maintain

- We run a (much shorter) script to update all the models every time we migrate to a new SHOP version

```python
@dataclass(repr=False)
class Plant(BaseObject):
    num_gen: Int = field(
        default=None,
        metadata=dict(
            isInput=False,
            isOutput=True,
            datatype="int",
            xUnit="NO_UNIT",
            yUnit="NO_UNIT",
            licenseName="SHOP_OPEN",
            fullName=None,
            dataFuncName="internal",
            description="Number of generators in the plant",
            documentationUrl="https://docs.shop.sintef.energy/attribute-table.html",
            exampleUrlPrefix="https://shop.sintef.energy/documentation/examples/",
            example=None,
            defaultValue=None,
        ),
    )
    num_pump: Int = field(
        default=None,
        metadata=dict(
            isInput=False,
            isOutput=True,
            datatype="int",
            xUnit="NO_UNIT",
            yUnit="NO_UNIT",
            licenseName="SHOP_OPEN",
            fullName=None,
            dataFuncName="internal",
            description="Number of pumps in the plant",
            documentationUrl="https://docs.shop.sintef.energy/attribute-table.html",
            exampleUrlPrefix="https://shop.sintef.energy/documentation/examples/",
            example=None,
            defaultValue=None,
        ),
    )
    less_distribution_eps: Double = field(
        default=None,
        metadata=dict(
            isInput=True,
            isOutput=False,
            datatype="double",
            xUnit="NO_UNIT",
```

20k lines of python models? 😳

- Having a pure python model of the SHOP data structure allow us to maintain the first goal

  ✓*A single SHOP instance to serve all production needs*

- Client applications and end user applications have a working data model that is independent of PySHOP, SHOP, CPLEX, license files etc.
  - Easier to debug
  - Easier to maintain

- We run a (much shorter) script to update all the models every time we migrate to a new SHOP version

```
135
136
137  def get_object_info() -> dict[str, dict[str, str]]:
138      """Get the data returned by `.info()` for all pyshop model o
139      s = ShopSession(
140          license_path=LICENCE_PATH, solver_path=SOLVER_PATH, supp
141      )
142      data = dict()
143      for objtype in s.shop_api.GetObjectTypeNames():
144          data[objtype] = dict()
145          for key in OBJECT_INFO_KEYS:
146              data[objtype][key] = parse_info_value(
147                  s.shop_api.GetObjectInfo(objtype, key)
148              )
149      return data
150
151
152  def get_attribute_info() -> dict[str, dict[str, dict[str, str]]]
153      """Get the data returned by `.info()` for all pyshop model a
154      s = ShopSession(
155          license_path=LICENCE_PATH, solver_path=SOLVER_PATH, supp
156      )
157      data = dict()
158      for objtype in s.shop_api.GetObjectTypeNames():
159          data[objtype] = dict()
160          for attrname in s.shop_api.GetObjectTypeAttributeNames(o
161              data[objtype][attrname] = dict()
162              for key in ATTRIBUTE_INFO_KEYS:
163                  data[objtype][attrname][key] = parse_info_value(
164                      s.shop_api.GetAttributeInfo(objtype, attrnam
165                  )
166      return data
167
168
```

Validation

- The data model validates and transforms data →→

- Using this framework, we can easily build, change and interrogate SHOP models independently of PySHOP, SHOP etc.
- Model data can be submitted for execution to

  ✓*A single SHOP instance to serve all production needs*

- The data model also simplifies data storage since the serialized YAML document is interchangeable with SHOP YAML

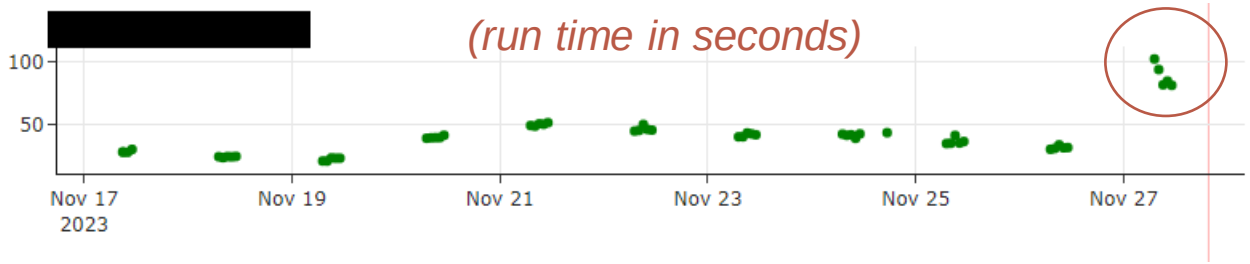
```python
def validate_array_has_no_na(x: Iterable):
    for y in x:
        validate_is_not_na(y)
    return x


def validate_first_txy_value_is_not_na(x: pd.Series | pd.DataFrame):
    if x.empty:
        raise ValueError("The timeseries is empty")
    if isinstance(x, pd.Series):
        validate_is_not_na(x.iloc[0])
    if isinstance(x, pd.DataFrame):
        x.iloc[0].apply(validate_is_not_na)
    return x


def validate_xy_has_no_na(x: pd.Series):
    x.apply(validate_is_not_na)
    x.index.map(validate_is_not_na)
    validate_is_not_na(x.name)
    return x


class ShopType:
    ...


class Double(float, ShopType):
    @classmethod
    def _supershop_parse(cls, v):
        validate_is_not_na(v)
        return float(v)

    @classmethod
    def _supershop_dump(cls, v):
        return float(v)


class Double_array(list, ShopType):
    @classmethod
    def _supershop_parse(cls, v):
        validate_array_has_no_na(v)
        return [float(i) for i in v]
```

SuperSHOP storage

- Results, along with inputs, logs and metadata is uploaded to a centralized storage
- The storage solution can be used for documenting and sharing non-production models

- Complete model data is stored as JSON documents in cold storage
- The metadata is stored in a hot database

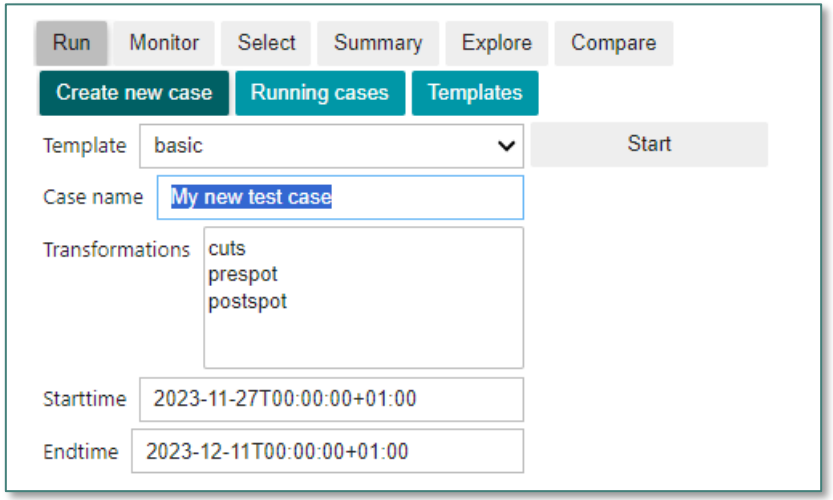
- We can query the metadata to find models of interest and use the *filename* to load the complete data set from cold storage
- The metadata table is also the source for monitoring, which allow us to detect performance regressions and problems early ⬇

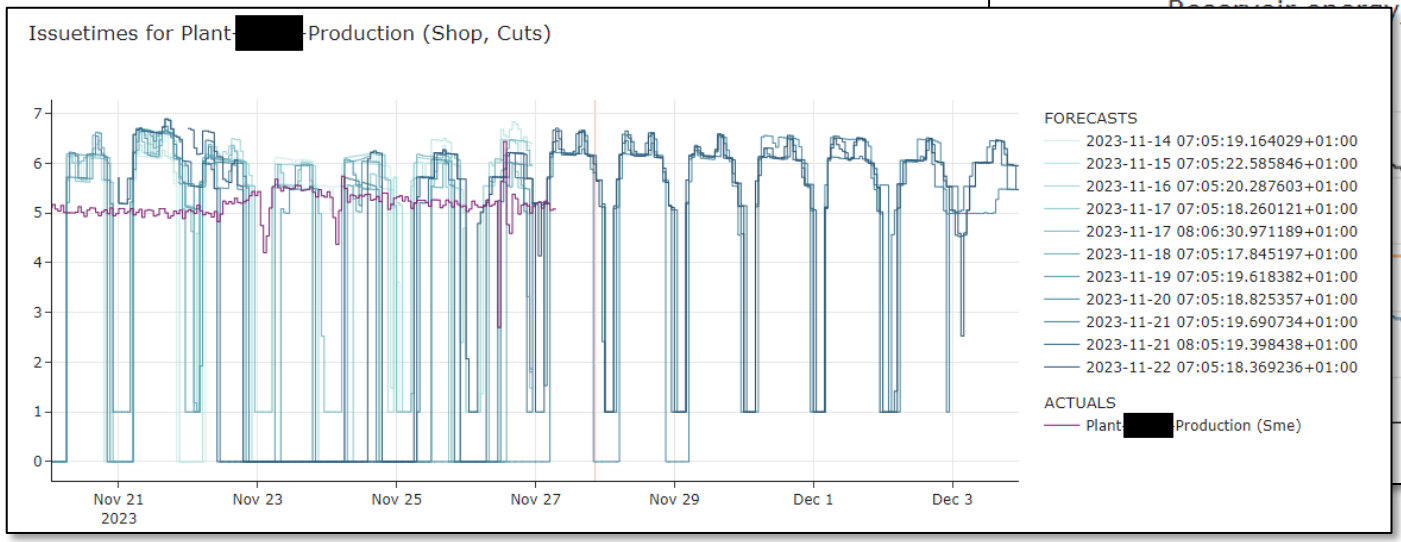


(a compressed YAML model)


```
{
    "data": "H4sIAKT7/WMC/+y96bJdt5GF+b+fQv87eALz4Kfo6BdgsGTaVjRNqkm
    "metadata": {
        "initialization_time": "2023-02-28 13:03:07.252370",
        "basemodel": "basic",
        "name": "high_inflow",
        "id": "3f50b017",
        "username": "a165963",
        "filename": "20230228_1303_basic_high_inflow_3f50b017.json",
        "starttime": "2023-02-28 00:00:00",
        "endtime": "2023-03-13 00:00:00",
        "transformations": [],
        "solver_status": "Optimal solution is available",
        "run_starttime": "2023-02-28 13:03:16.426327+00:00",
        "run_seconds": "6.772239",
        "run_seconds_per_iteration": [
            0.198807,
            0.16426,
            0.150423,
            0.096469,
            0.088603,
            0.088354
        ],
        "n_warnings": 3,
        "shop_version": "15.0.0.0",
        "supershop_version": "0.7.0",
        "python_version": "3.10.1",
        "comment": null
    }
}
```

User interfaces





User interfaces

```
s = SuperShop.from_remote_archive('20231127_1932_basic_My new test case_7709c770.json')
```

•••

```
s.metadata
```

	20231127_1932_basic_My new test case_7709c770.json
basemodel	basic
comment	
endtime	2023-12-11 00:00:00
filename	20231127_1932_basic_My new test case_7709c770.json
id	7709c770
initialization_time	2023-11-27 19:32:29.014526
n_warnings	0
name	My new test case
python_version	3.11.6
run_seconds	5.447923
run_seconds_per_iteration	[0.116723, 0.098458, 0.090219, 0.056048, 0.056518, 0.055785]
run_starttime	2023-11-27 19:32:29.162260
shop_version	15.4.0.1
solver_status	Optimal solution is available
starttime	2023-11-27 00:00:00
supershop_version	0.10.0
transformations	[]
username	Nils Flaten Ræder

User interfaces

```
s = SuperShop.from_remote_archive('20231127_1932_basic_My new test case_7709c770.json')
```

```
...
```

```
s.metadata
```

	20231127_1932_basic_My new test case_7709c770.json
basemodel	basic
comment	
endtime	2023-12-11 00:00:00
filename	20231127_1932_basic_My new test case_7709c770.json
id	7709c770
initialization_time	2023-11-27 19:32:29.014526
n_warnings	0
name	My new test case
python_version	3.11.6
run_seconds	5.447923
run_seconds_per_iteration	[0.116723, 0.098458, 0.090219, 0.056048, 0.056518, 0.055785]
run_starttime	2023-11-27 19:32:29.162260
shop_version	15.4.0.1
solver_status	Optimal solution is available
starttime	2023-11-27 00:00:00
supershop_version	0.10.0
transformations	[]
username	Nils Flaten Ræder

```
s.objective
```

	average_objective	scen_1
solver_status	Optimal solution is available	Optimal solution is available
grand_total	-451329.625578	-451329.625578
total	-451329.625578	-451329.625578
rsv_end_value	-56760.152209	-56760.152209
rsv_end_value_relative	-30543.388004	-30543.388004
market_sale_buy	-409569.473369	-409569.473369
startup_costs	15000.0	15000.0

User interfaces

```
s = SuperShop.from_remote_archive('20231127_1932_basic_My new test case_7709c770.json')
```

```
...
```

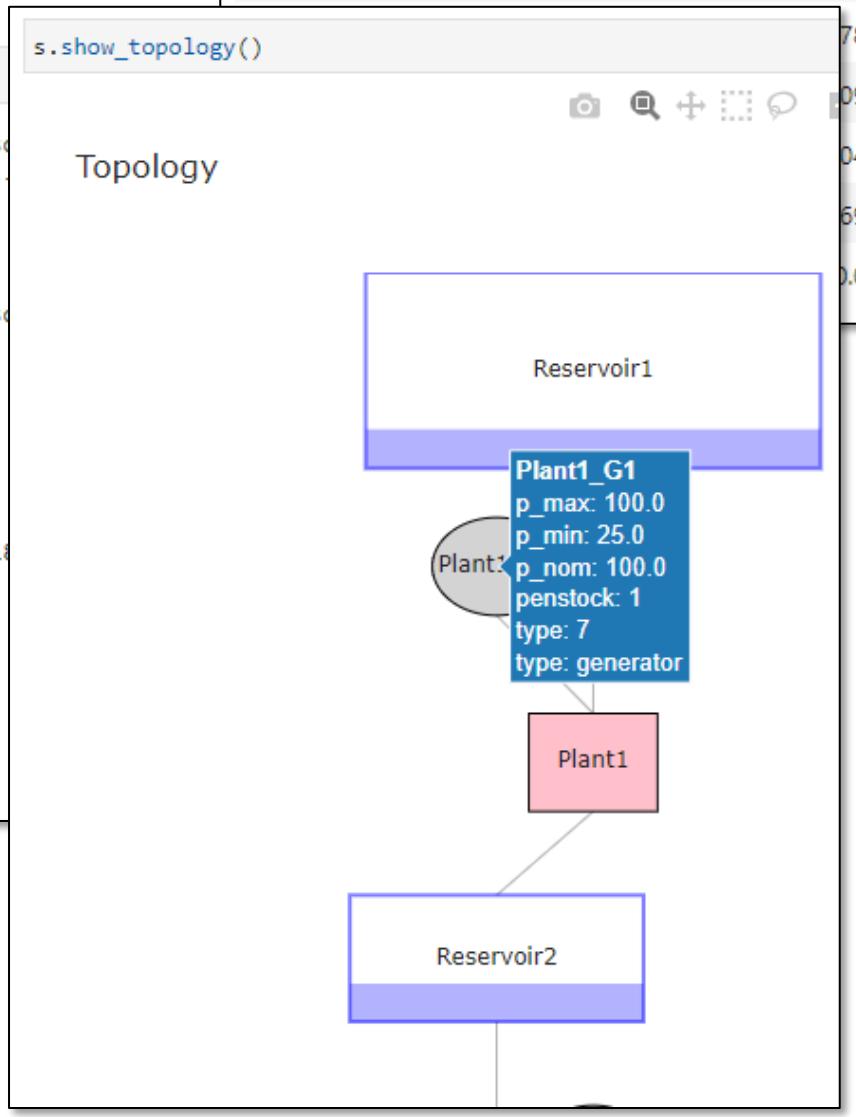
```
s.metadata
```

	20231127_1932_basic_My new test case_7709c770.js
basemodel	basic
comment	
endtime	2023-12-11 00:00:00
filename	20231127_1932_basic_My new test case_7709c770.js
id	7709c770
initialization_time	2023-11-27 19:32:29.014526
n_warnings	0
name	My new test case
python_version	3.11.6
run_seconds	5.447923
run_seconds_per_iteration	[0.116723, 0.098458, 0.090219, 0.056048, 0.05651
run_starttime	2023-11-27 19:32:29.162260
shop_version	15.4.0.1
solver_status	Optimal solution is available
starttime	2023-11-27 00:00:00
supershop_version	0.10.0
transformations	[]
username	Nils Flaten Ræder

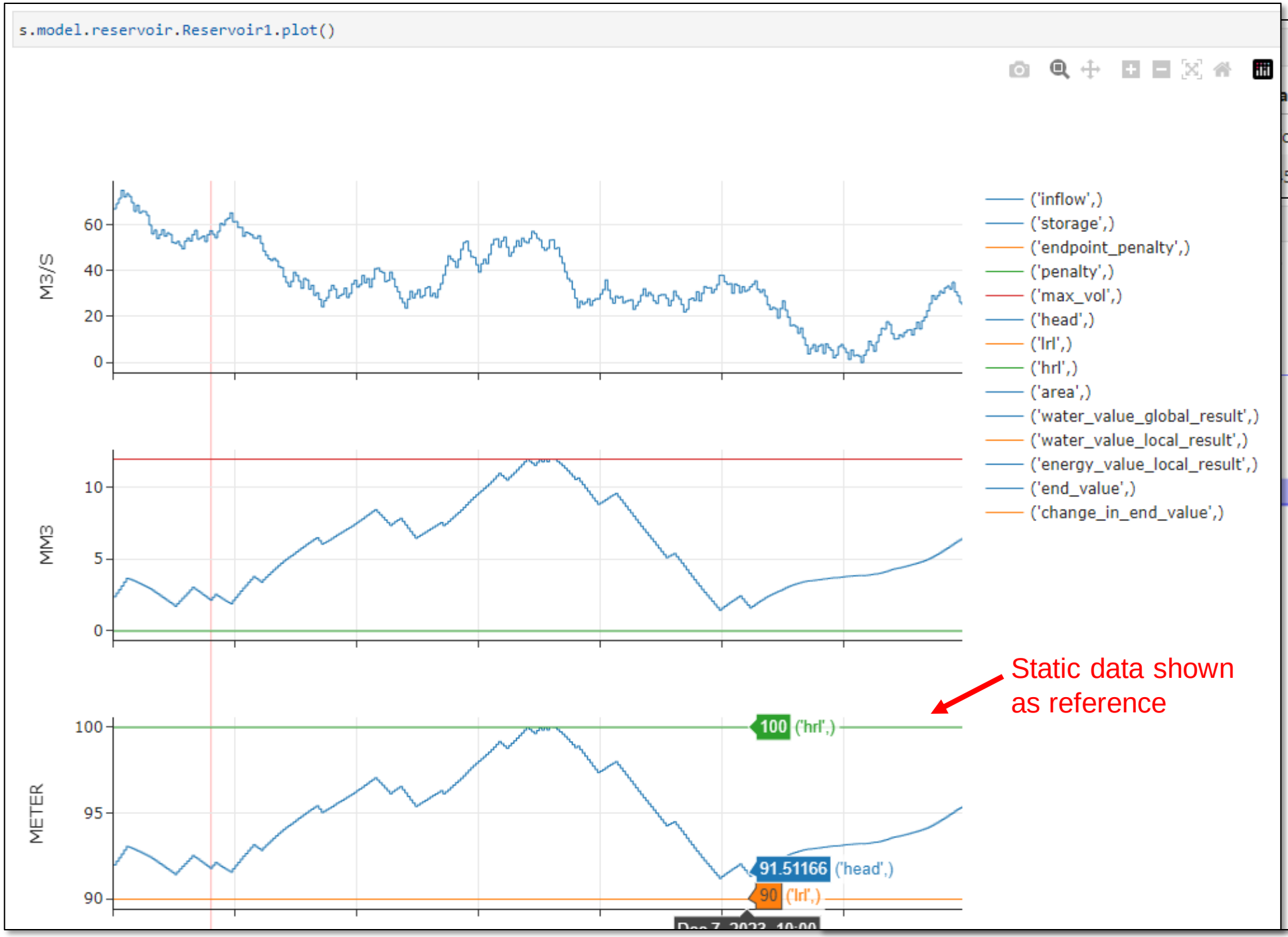
```
s.objective
```

	average_objective	scen_1
solver_status	Optimal solution is available	Optimal solution is available
grand_total	-451329.625578	-451329.625578
	78	-451329.625578
	09	-56760.152209
	04	-30543.388004
	69	-409569.473369
	0.0	15000.0

```
s.show_topology()
```



```
s.model.reservoir.Reservoir1.plot()
```



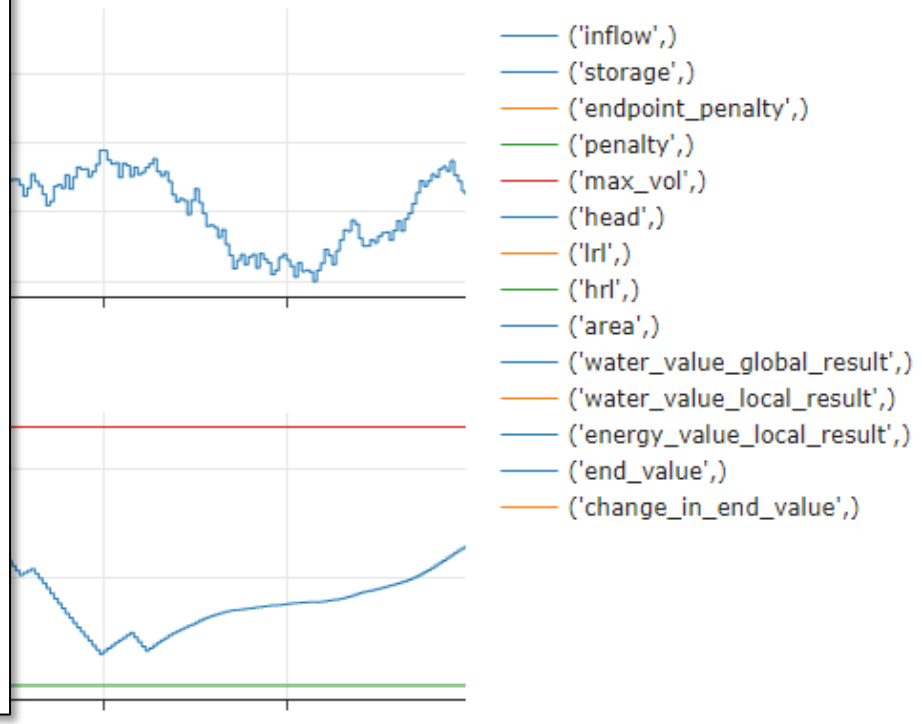
Static data shown as reference

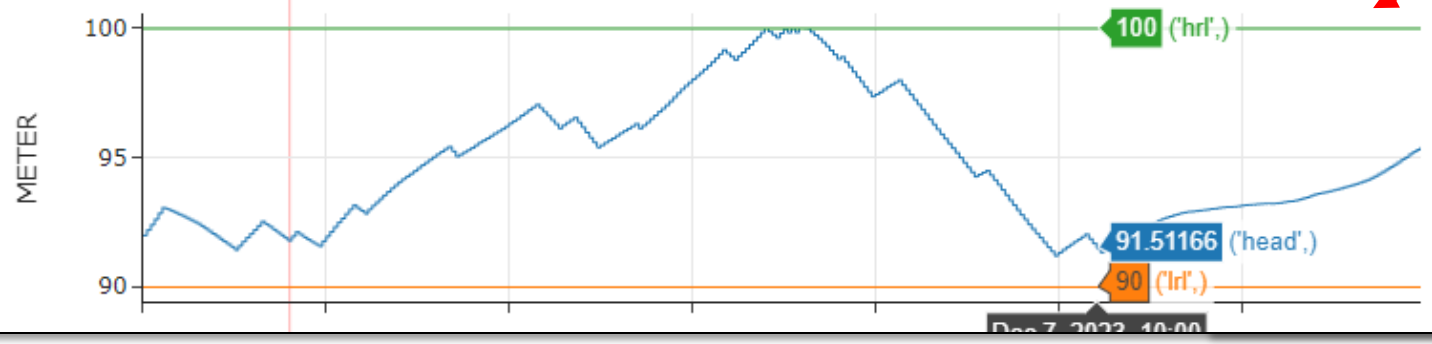
...age_objective	scen_1
...on is available	Optimal solution is available
...51329.625578	-451329.625578
...78	-451329.625578
...09	-56760.152209
...04	-30543.388004
...69	-409569.473369
...0.0	15000.0

```
s.model.reservoir.Reservoir1.plot()
```

```
s.model.reservoir
```

	Reservoir1	Reservoir2
max_vol	12.0	5.0
lrl	90.0	40.0
hrl	100.0	50.0
vol_head	<Xy>	<Xy>
start_head	92.0	43.0
inflow	<Txy>	
flow_descr	<Xy>	<Xy>
energy_value_input	39.7	38.6
storage	<Txy>	<Txy>
head	<Txy>	<Txy>
area	0.0	0.0
endpoint_penalty	0.0	0.0
penalty	0.0	0.0
water_value_global_result	<Txy>	<Txy>
water_value_local_result	<Txy>	<Txy>
energy_value_local_result	<Txy>	<Txy>
end_value	<Txy>	<Txy>
change_in_end_value	<Txy>	<Txy>
calc_global_water_value	8491.7707415225	3891.0096168763
energy_conversion_factor	115.88818953769	100.80335795016





Static data shown as reference

	age_objective	scen_1
	on is available	Optimal solution is available
	51329.625578	-451329.625578
	78	-451329.625578
	09	-56760.152209
	04	-30543.388004
	69	-409569.473369
	.0	15000.0

```
s.model.reservoir.Reservoir1.plot()
```

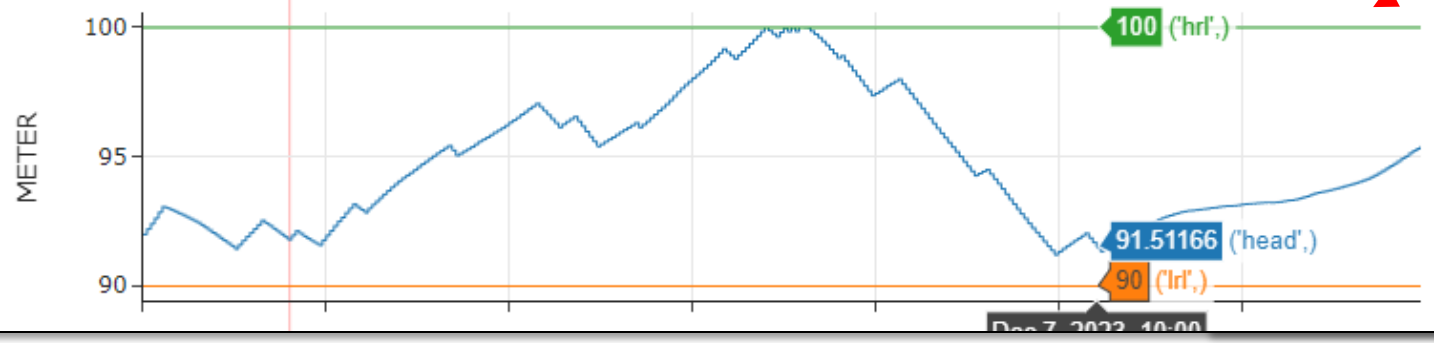
```
s.model.reservoir
```

	Reservoir1	Reservoir2
max_vol	12.0	5.0
lrl	90.0	40.0
hrl	100.0	50.0
vol_head	<Xy>	<Xy>
start_head	92.0	43.0
inflow	<Txy>	
flow_descr	<Xy>	<Xy>
energy_value_input	39.7	38.6
storage	<Txy>	<Txy>
head	<Txy>	<Txy>
area	0.0	0.0
endpoint_penalty	0.0	0.0
penalty	0.0	0.0
water_value_global_result	<Txy>	<Txy>
water_value_local_result	<Txy>	<Txy>
energy_value_local_result	<Txy>	<Txy>
end_value	<Txy>	<Txy>
change_in_end_value	<Txy>	<Txy>
calc_global_water_value	8491.7707415225	3891.0096168763
energy_conversion_factor	115.88818953769	100.80335795016

```
s.sh
```

- f show_gen_eff_curves function
- f show_highlights function
- f show_multiple_txy function
- f show_topology function
- f show_turb_eff_curves function
- f show_txy function
- f show_volhead function

('inflow',)
('storage',)
('endpoint_penalty',)
('penalty',)
('max_vol',)
('head',)
('lrl',)
('hrl',)
('area',)
('water_value_global_result',)
('water_value_local_result',)
('energy_value_local_result',)
('end_value',)
('change_in_end_value',)



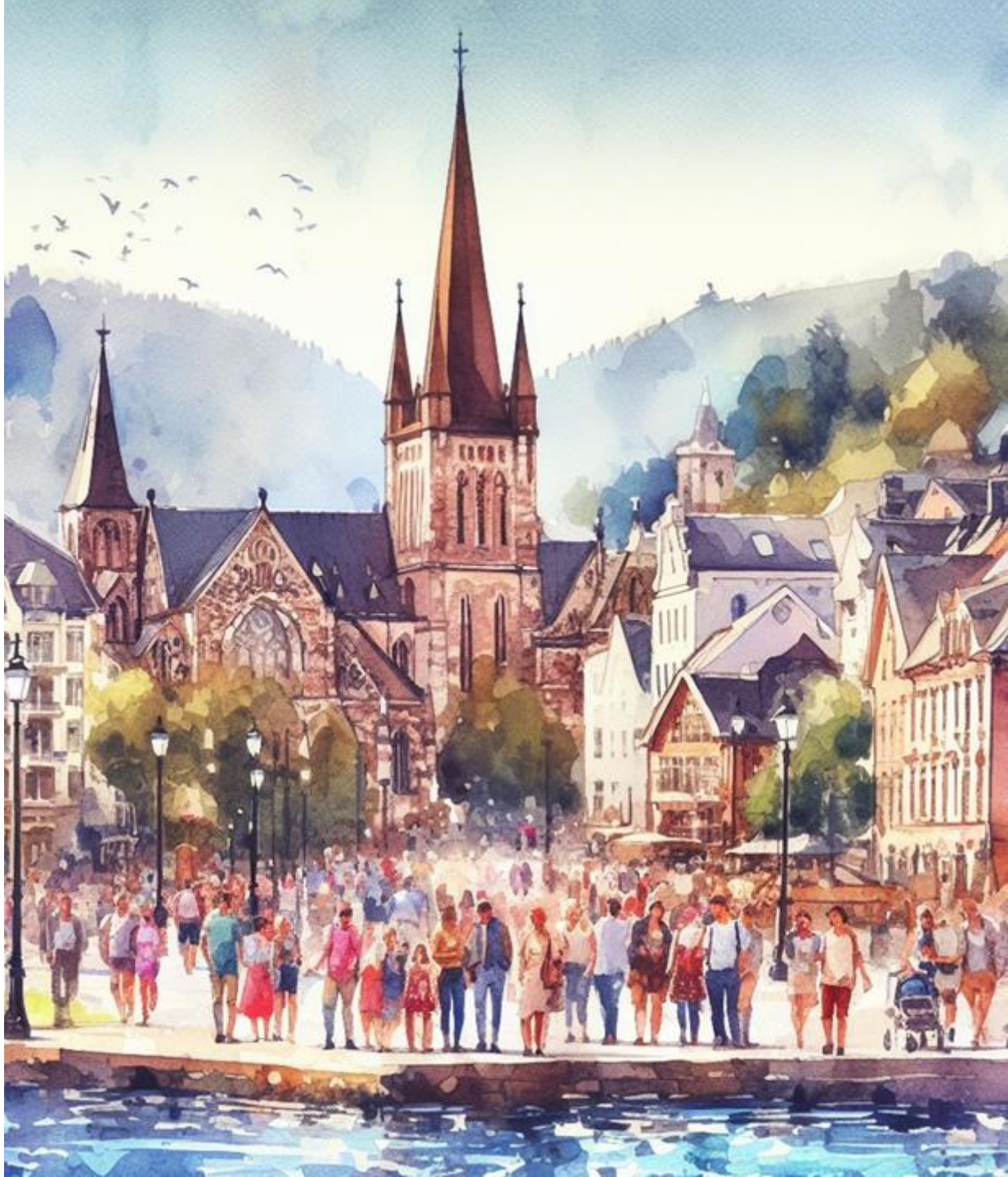
Static data shown as reference

	scen_1
on is available	Optimal solution is available
51329.625578	-451329.625578
78	-451329.625578
09	-56760.152209
04	-30543.388004
69	-409569.473369
.0	15000.0

Lessons learned

- The most important interface of SuperSHOP is the python API which allow interactive data discovery and prototyping
- `Streamlit` (demoed previously) didn't scale well with when when deployed to the cloud, we have switched frontend framework
- A data model that is independent of SHOP simplifies the creation of dashboards, client applications and data pipelines
- Metadata is important!
- Previous versions of SuperSHOP were built on `pydantic` models
  - Transition from v1 to v2 was a pain
  - We still use `pydantic` for some data validation/transformation
  - I highly recommend `pydantic`!

Questions?



Source


Hydro
ENERGY