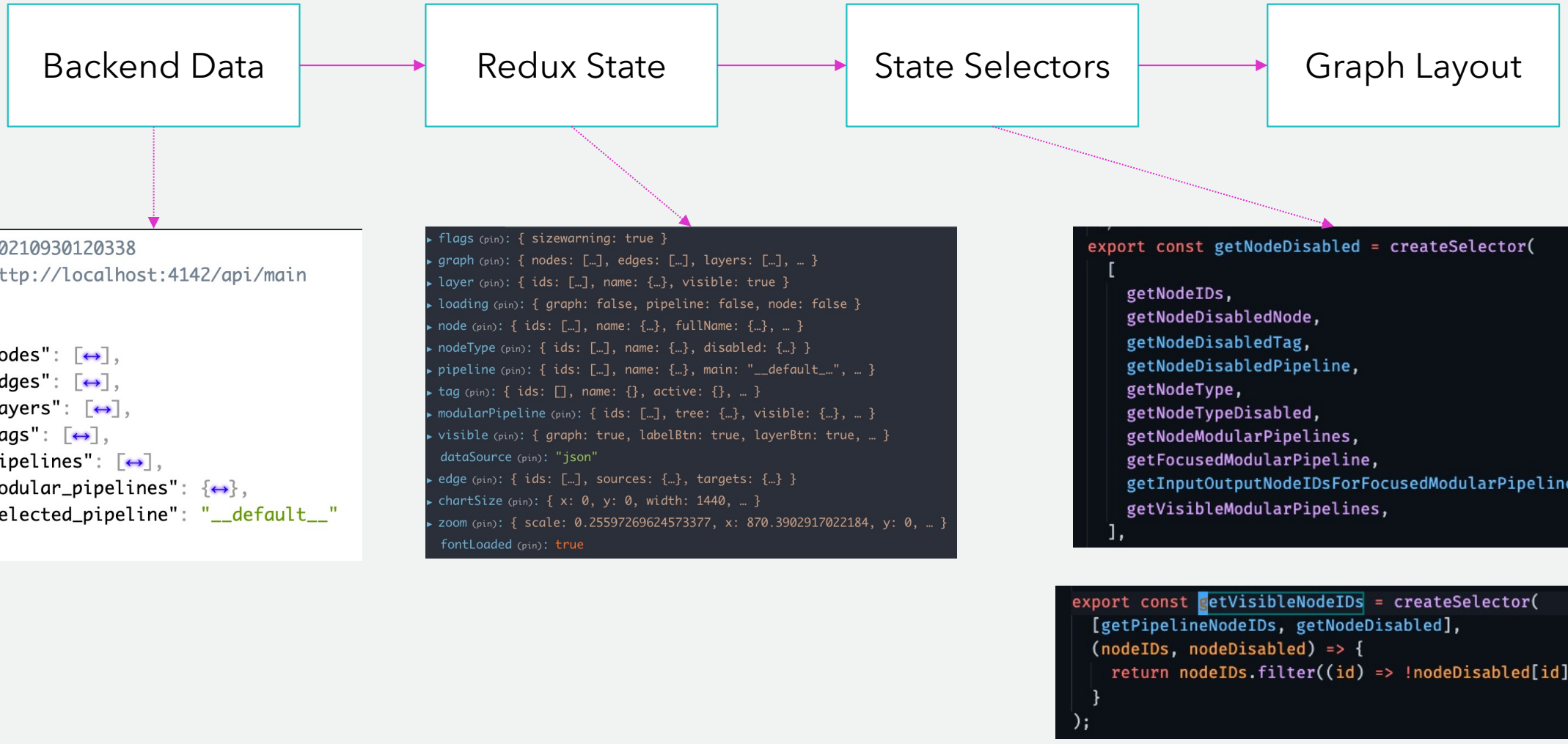
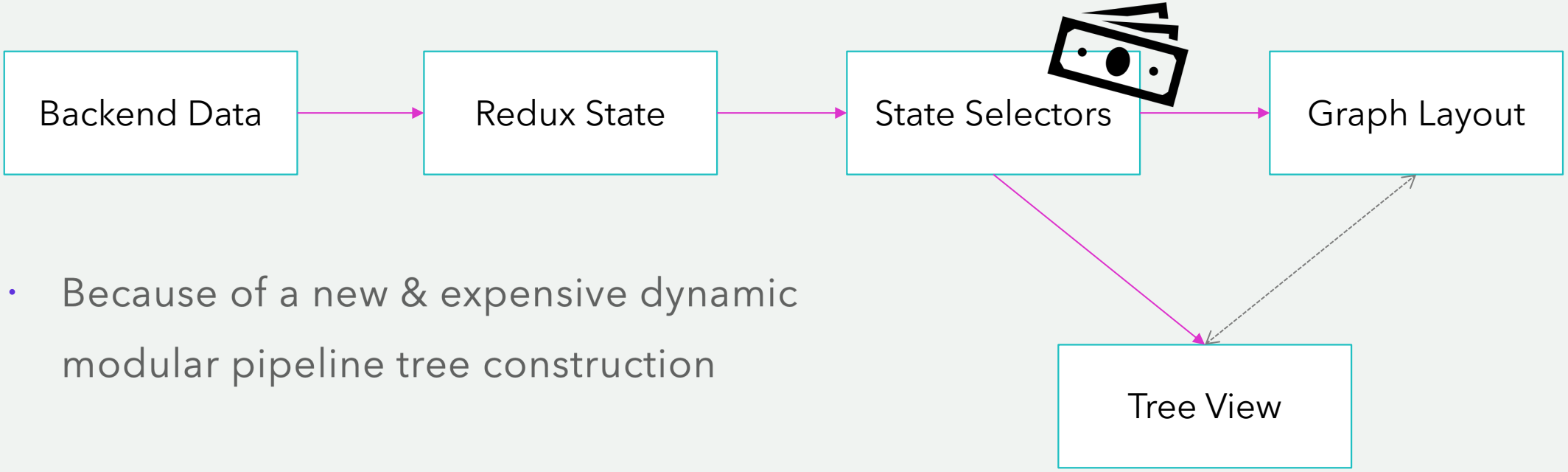
Feature Demo

- Expand & collapse modular pipelines on the graph
- With layers
- With search
- With filters
- With hover state
- With metadata side panel
- With registered pipeline dropdown
- With focused mode (will discuss further)
- Some stress testing

How Kedro Viz works



Why did it get so complex



- Because of a new & expensive dynamic modular pipeline tree construction

Why is the dynamic tree construction complex & expensive

- Or rather, why do we have to dynamically construct this tree in the first place?
- In Kedro's core, modular pipeline trees are represented as materialized path

```
db.categories.insertMany( [
   { _id: "Books", path: null },
   { _id: "Programming", path: ",Books," },
   { _id: "Databases", path: ",Books,Programming," },
   { _id: "Languages", path: ",Books,Programming," },
   { _id: "MongoDB", path: ",Books,Programming,Databases," },
   { _id: "dbm", path: ",Books,Programming,Databases," }
] )
```

https://docs.mongodb.com/manual/tutorial/model-tree-structures-with-materialized-paths/

Why is the dynamic tree construction complex & expensive

- Easy-to-use for Kedro user: You can declare your node 3 levels deep just by declaring `namespace="one.two.three"`
- Efficient for Kedro runtime: At runtime, a Kedro pipeline is a flat set of nodes, so the materialized path structure is superfluous and can be discarded without traversing cost.
- Also enable neat querying syntax for nodes in the tree, e.g. pipeline.only_node_with_namespace("")
- In other words, Kedro core optimizes for the leaves in the tree.

Why is the dynamic tree construction complex & expensive

- On the other hand, a visualization of the tree structure is (probably) better served by a child-references structure

```
db.categories.insertMany( [
   { _id: "MongoDB", children: [] },
   { _id: "dbm", children: [] },
   { _id: "Databases", children: [ "MongoDB", "dbm" ] },
   { _id: "Languages", children: [] },
   { _id: "Programming", children: [ "Databases", "Languages" ] },
   { _id: "Books", children: [ "Programming" ] }
] )
```

https://docs.mongodb.com/manual/tutorial/model-tree-structures-with-child-references/

Why is the dynamic tree construction complex & expensive

- On the other hand, a tree visualization is (probably) better served by a child-references structure: search is a simple DFS and render is just an in-order traversal, both can be implemented recursively.

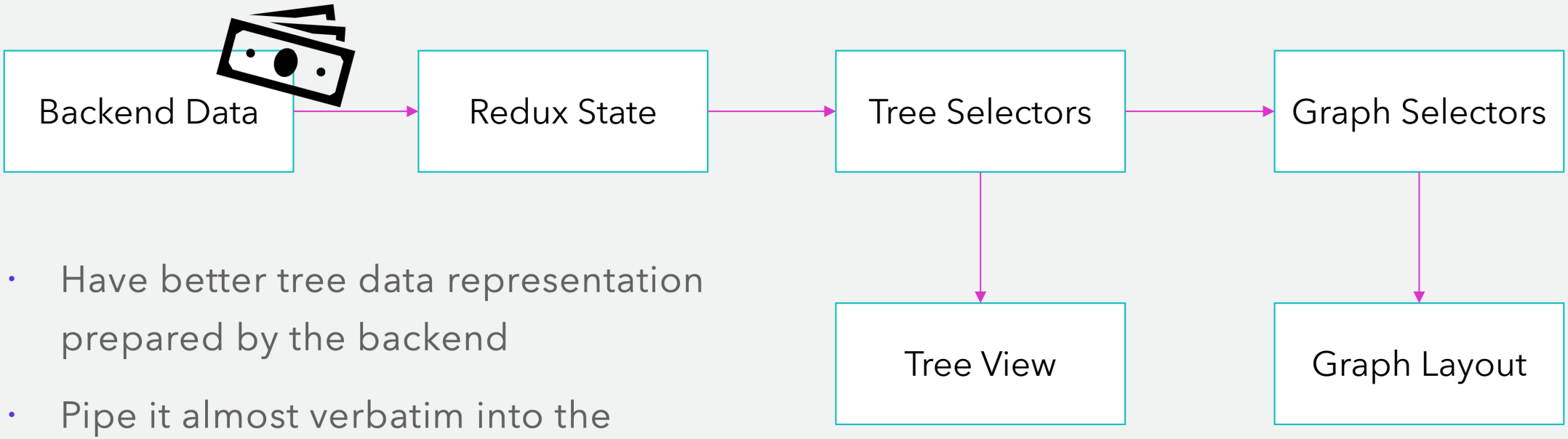
```
db.categories.insertMany( [
   { _id: "MongoDB", children: [] },
   { _id: "dbm", children: [] },
   { _id: "Databases", children: [ "MongoDB", "dbm" ] },
   { _id: "Languages", children: [] },
   { _id: "Programming", children: [ "Databases", "Languages" ] },
   { _id: "Books", children: [ "Programming" ] }
] )
```

https://docs.mongodb.com/manual/tutorial/model-tree-structures-with-child-references/

Why is the dynamic tree construction complex & expensive

- The problem, though, is that viz uses neither of these methods to represent and transfer modular pipeline tree data in the frontend itself and between frontend and backend.
- It uses a flat list of modular pipelines and work out the tree dynamically from the graph's node namespace… in the selectors… on every render with state changes!

Problem solving strategy



- Have better tree data representation prepared by the backend
- Pipe it almost verbatim into the collapsible tree view
- The tree's state becomes the source of truth for what's visible on the graph

Kedro-Viz Backend

materialised paths (namespace) representation of modular pipelines tree

Kedro Core

Data Access Manager

Modular Pipelines Repository

child-references representation of modular pipelines tree

API


```
"modular_pipelines": {
  "__root__": {
    "id": "__root__",
    "name": "Root",
    "inputs": [

    ],
    "outputs": [

    ],
    "children": [
      {
        "id": "de",
        "type": "modularPipeline"
      },
      {
        "id": "f1f1425b",
        "type": "parameters"
      },
      {
        "id": "23c94afb",
        "type": "data"
      },
      {
        "id": "ds",
        "type": "modularPipeline"
      },
      {
        "id": "966b9734",
        "type": "data"
      }
    ]
  },
  "de.level_one.level_two.level_three": {↔},
  "de": {↔},
  "de.level_one": {↔},
  "de.level_one.level_two": {↔},
  "ds": {↔}
},
```


example API response

Kedro-Viz Frontend

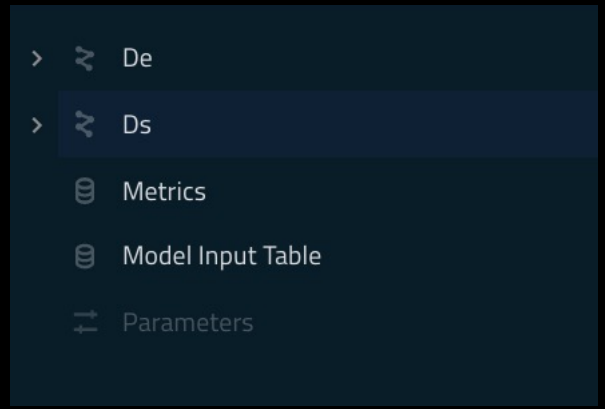
Redux State

Tree Selectors

Graph Selectors

Tree View

Graph Layout



example modular pipelines tree view

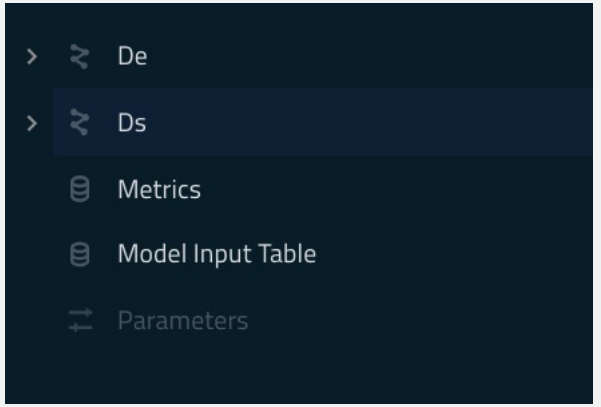
Modular Pipelines Tree API Response

- Child-references representation
- Map directly to the data structure required by Material UI TreeView with little extra processing -> tree render faster.
- Search is a tree filtering procedure, not construction -> search faster.
- Modular Pipelines' inputs & outputs are pre-calculated by the server, so no graph traversal to calculate focus mode -> focus mode faster (and more "correct" – more on this later)
- Lots of complex selectors could be removed.
- Strictly unidirectional data & concept flow.


```
"modular_pipelines": {
  "__root__": {
    "id": "__root__",
    "name": "Root",
    "inputs": [

    ],
    "outputs": [

    ],
    "children": [
      {
        "id": "de",
        "type": "modularPipeline"
      },
      {
        "id": "f1f1425b",
        "type": "parameters"
      },
      {
        "id": "23c94afb",
        "type": "data"
      },
      {
        "id": "ds",
        "type": "modularPipeline"
      },
      {
        "id": "966b9734",
        "type": "data"
      }
    ]
  },
  "de.level_one.level_two.level_three": {↔},
  "de": {↔},
  "de.level_one": {↔},
  "de.level_one.level_two": {↔},
  "ds": {↔}
},
```

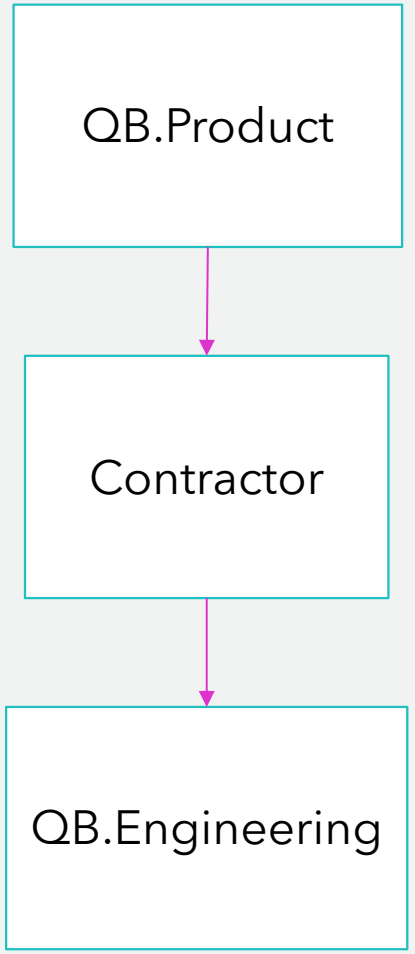



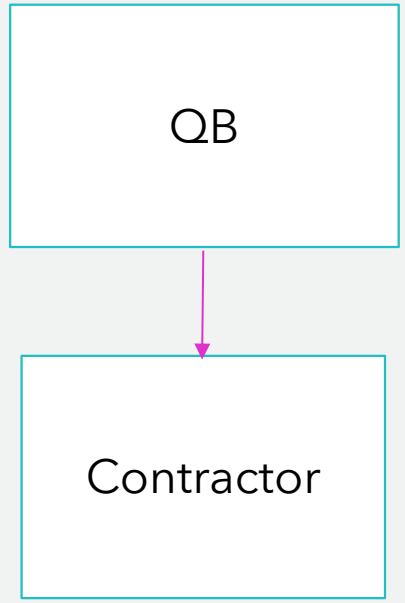
Tradeoff: Compatibility

- The new feature requires a different JSON structure for modular pipelines, so won't be compatible with the JSON exported by older kedro viz.
- It's still compatible with older Kedro version.

Edge case: Cycle introduced by modular pipeline nodes

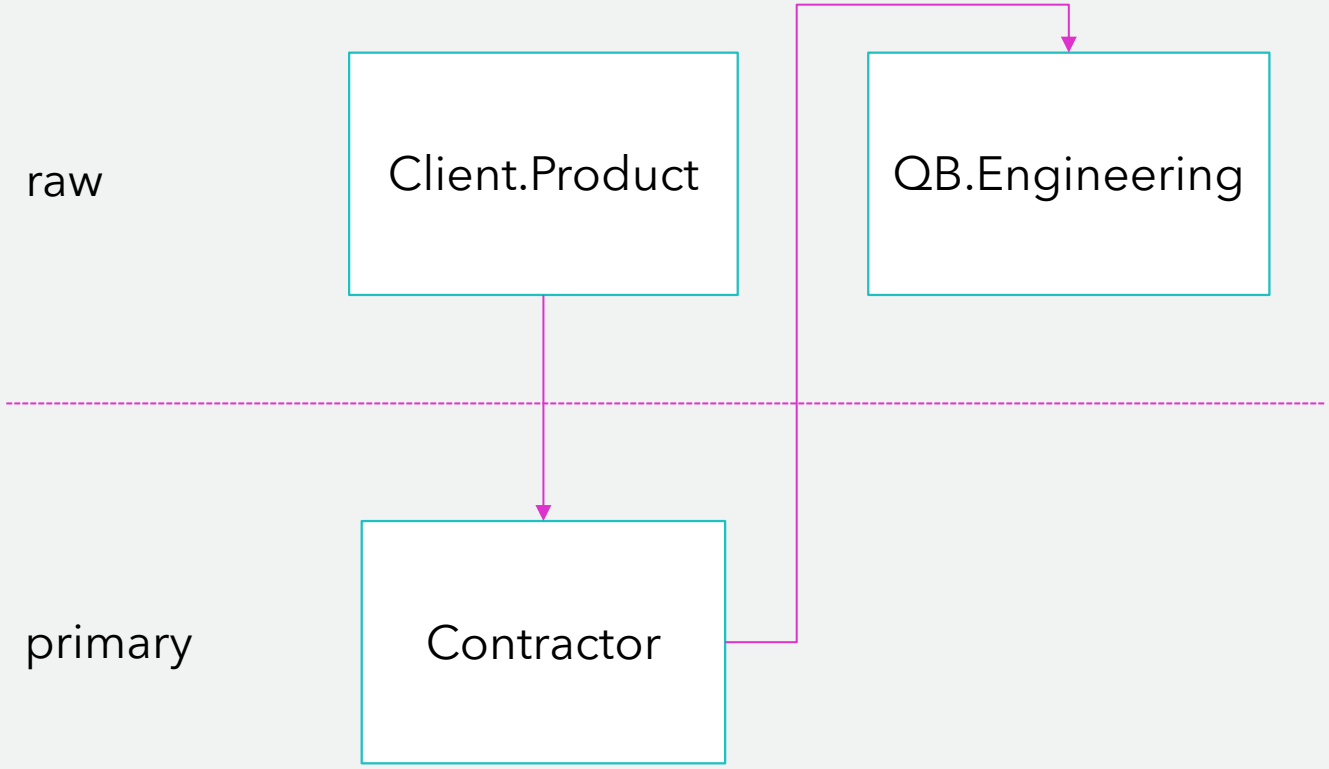
- Kedro nodes don't form cycle but modular pipeline can
- Detect cycle based on an observation: a cycle can only form when a *reachable descendant* of a modular pipeline is also an *input* into that pipeline
- Since we know what the inputs of a mod pipe are up-front and only have to check the descendants of each mod pipe node, not all nodes, this is quick (definitely quicker than doing generic cycle detection on the graph on every render)
- Then we throw away the edge between the input and the modular pipeline
- Can potentially show an error message along the line of: hey, when collapsed your modular pipeline has a dependency between input X and modular pipeline Y so we have removed that edge.





Edge case: Cycle introduced by layers

- Detect cycle in layer
- MVP: Disable layers in case there is a circular dependency.
- Log a warning saying there is a cycle in your graph. Fix it.
- Other approach: throw away the cycle layer edge, but that's a bit more work. Could be a further improvement.



Discussion: Collapsed structure vs. focused structure

There are still lots of improvement to be made

- The way we load and cache nodes for registered pipelines is too complicated in the name of pre-mature optimization, so we should fix that.
- Still writing tests for PR. Should be done soon ™