The PDI Data Interface

Architecture of a data coupling library

February 22nd, 2021 – EXA2PRO-EoCoE joint workshop

Julien Bigot[2], Amal Gueroudji[2], Karim Hasnaoui[3], Yacine Ould Rouis[3], Karol Sierocinski[7], Kacper Sinkiewicz[7]

Thanks to:
Leonardo Bautista Gomez[1], Sebastian Friedemann[6], Virginie Grandgirard[2], Francesco Iannone[4], Kai Keller[1], Guillaume Latu[2], Bruno Raffin[6], Benedikt Steinbusch[5], Christian Witzler[5]

[1] BSC, [2] CEA, [3] CNRS, [4] ENEA, [5] FZJ, [6] Inria, [7] PSNC

Initial Motivation: the I/O Issue

- We want it easy to use
- We want it fast
- We want a portable library
- We want large language support
- We want parallelization independent file format
- We want a portable file format
- We want to leverage the underlying hardware
- We want…

Initial Motivation: the I/O Issue

- We want it easy to use
- We wan
- We wan
- We want large language support
- We want parallelization independent file format
- We want a portable file format
- We want to leverage the underlying hardware
- We want…

Handling I/O is complex
Optimizing I/O is a job on its own

Initial Motivation: the I/O Issue

- We want it easy to use
- We war
- We war
- We want large language support
- W
- We want a portable file format
- We want to
- We want…

Handling I/O is complex
Optimizing I/O is a job on its own

Complex but common problem,
A community with dedicated expert

Let’s use **libraries**

The I/O Issue: the library ecosystem

FTI

MPI I/O

pNetCDF

SIONlib

FlowVR

Adios

Hercule

NetCDF

HDF5

XIOS

IME

POSIX

SCR

pHDF5

Damaris

PaDaWAn

The I/O issue: Choosing a library

Choosing the best library: a problem on its own

The best library depends on…

- The code specifics, the type of I/O
  - Parallelism level, replicated / distributed data, I/O frequency, …
  - Initialization data reading, result writing (small or large), checkpoint writing, coupling related I/O
- The specific execution
  - Small case / large case, debug / production, …
- The specific hardware available
    - I/O bandwidth, intermediate storage, …

The I/O issue: Choosing a library

Choosing the best library: a problem on its own

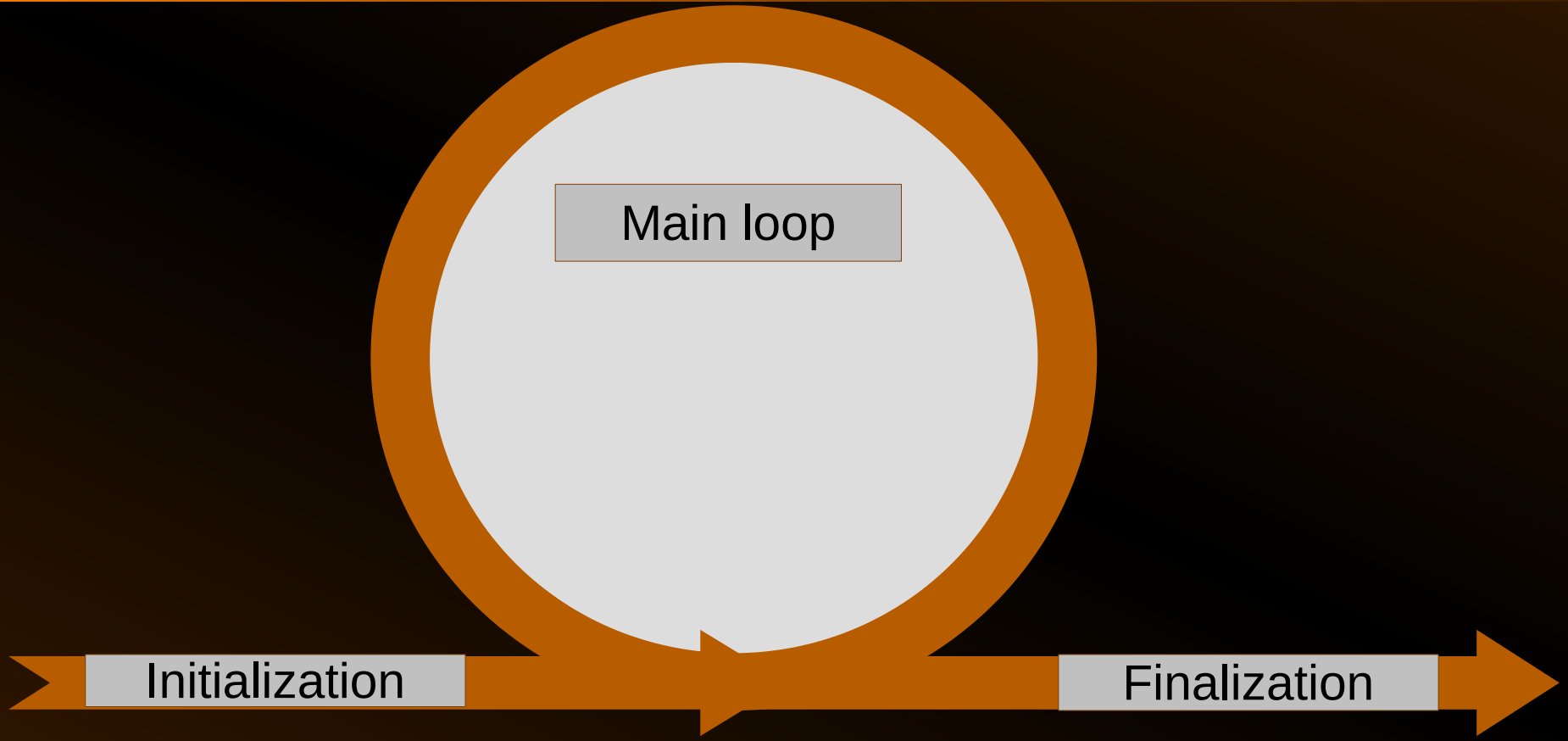
The best library depends on…

- The code specifics, the type of I/O
  - Parallelism lev quency, …
  - Initialization data reading, result writing (small or large), checkpoint writing, coupling related I/O
- The specific execution
  -
- T
  - I/O bandwidth, intermediate storage, …

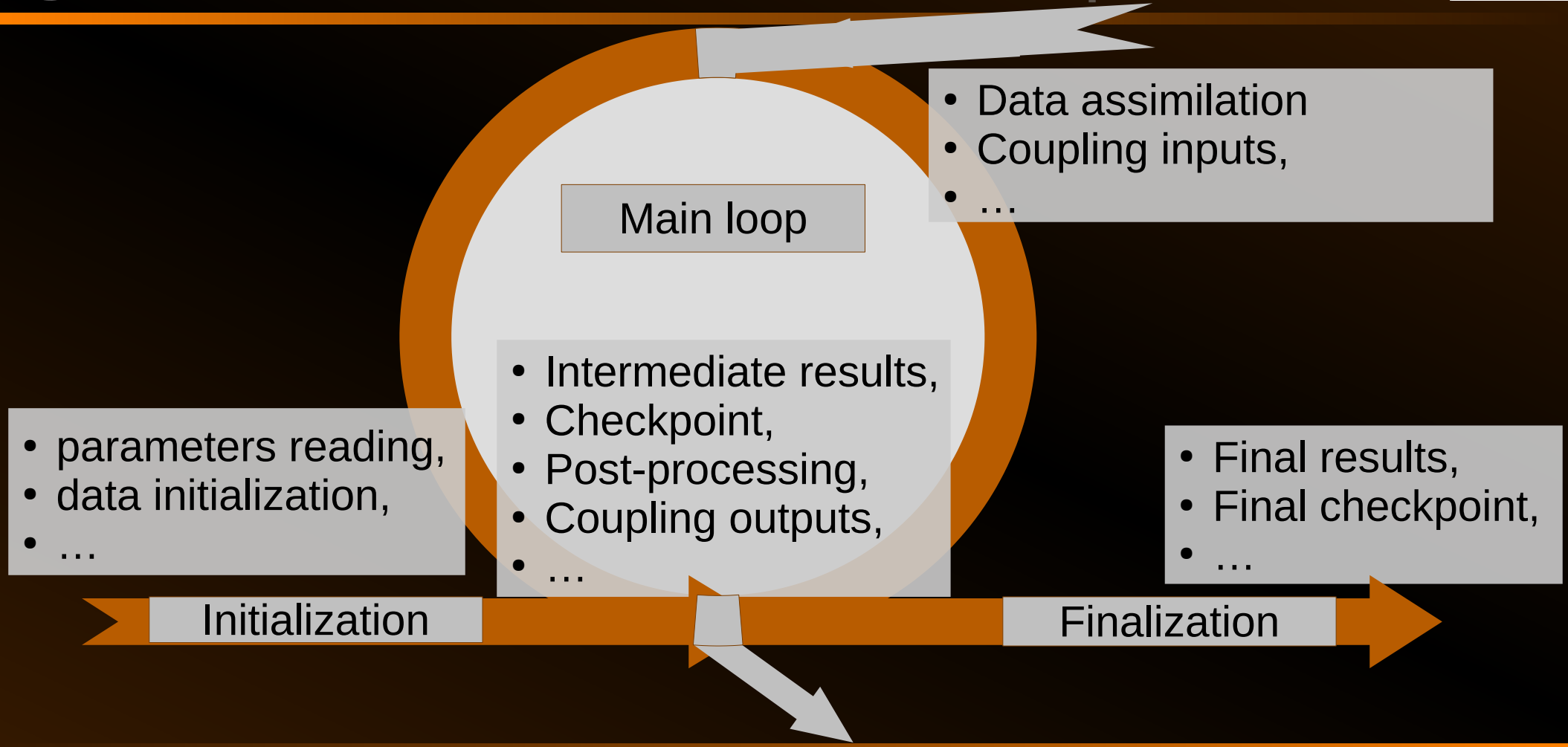
Not one-size fits all library

Many codes end-up with an IO abstraction layer

I/O in codes: let's take a step back



I/O in codes: let's take a step back

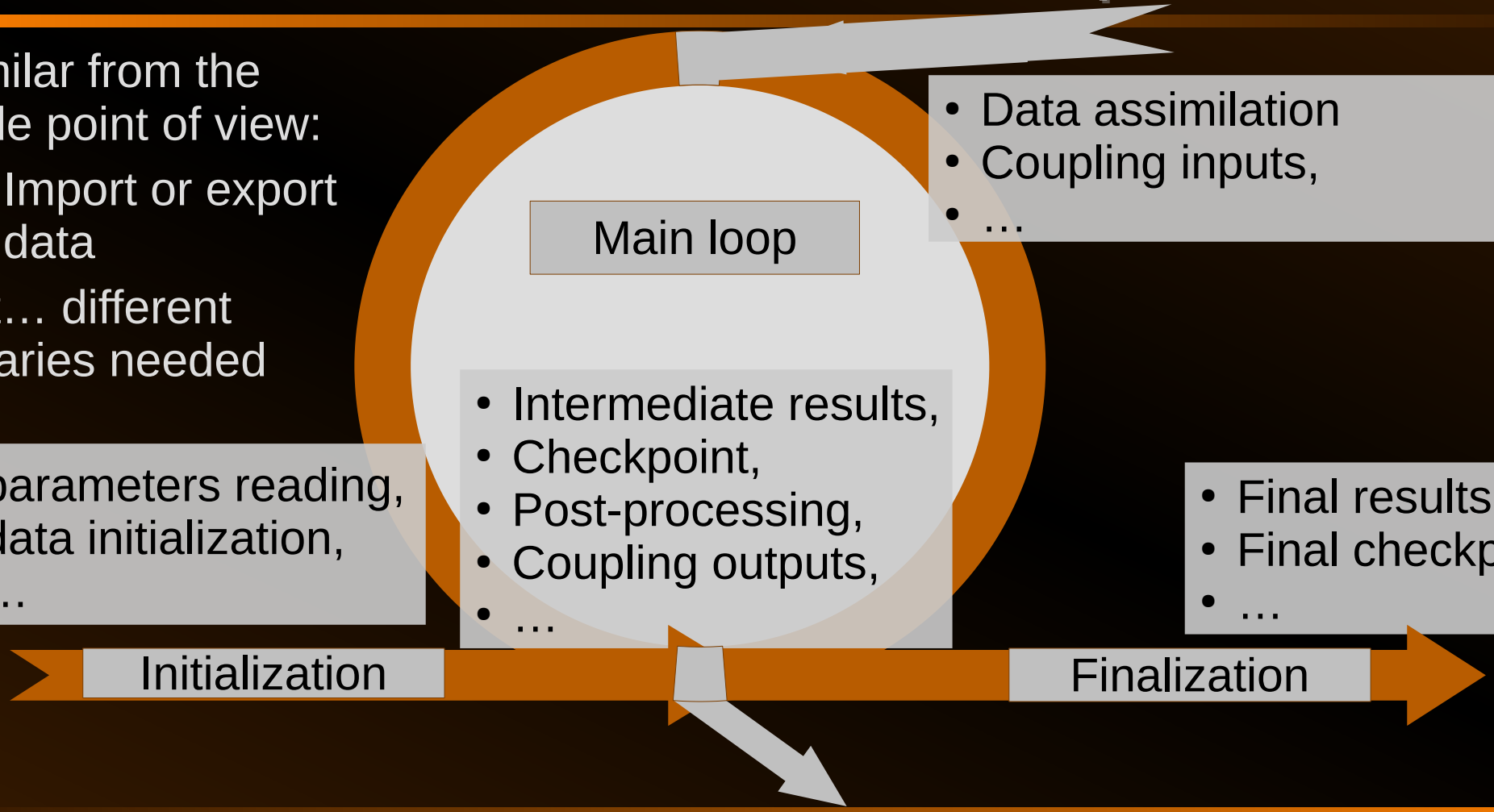


I/O in codes: let's take a step back

Similar from the code point of view:

- Import or export data

But… different libraries needed



I/O in codes: let's take a step back

Similar from the code point of view:

- Import or export data

But… different libraries needed

- parameters rea
- data initializati
- …

Initialization

Main loop

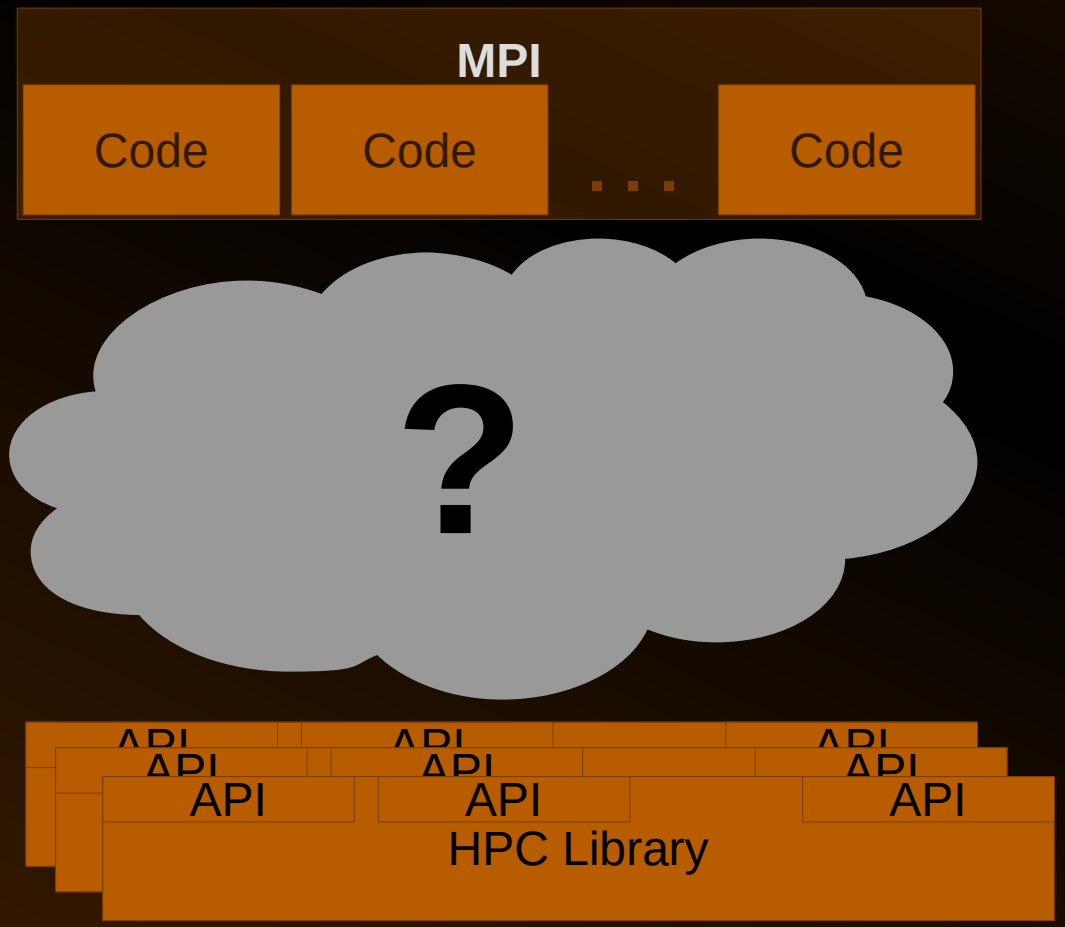
- Data assimilation
- Coupling inputs,
- …

The data-coupling problem

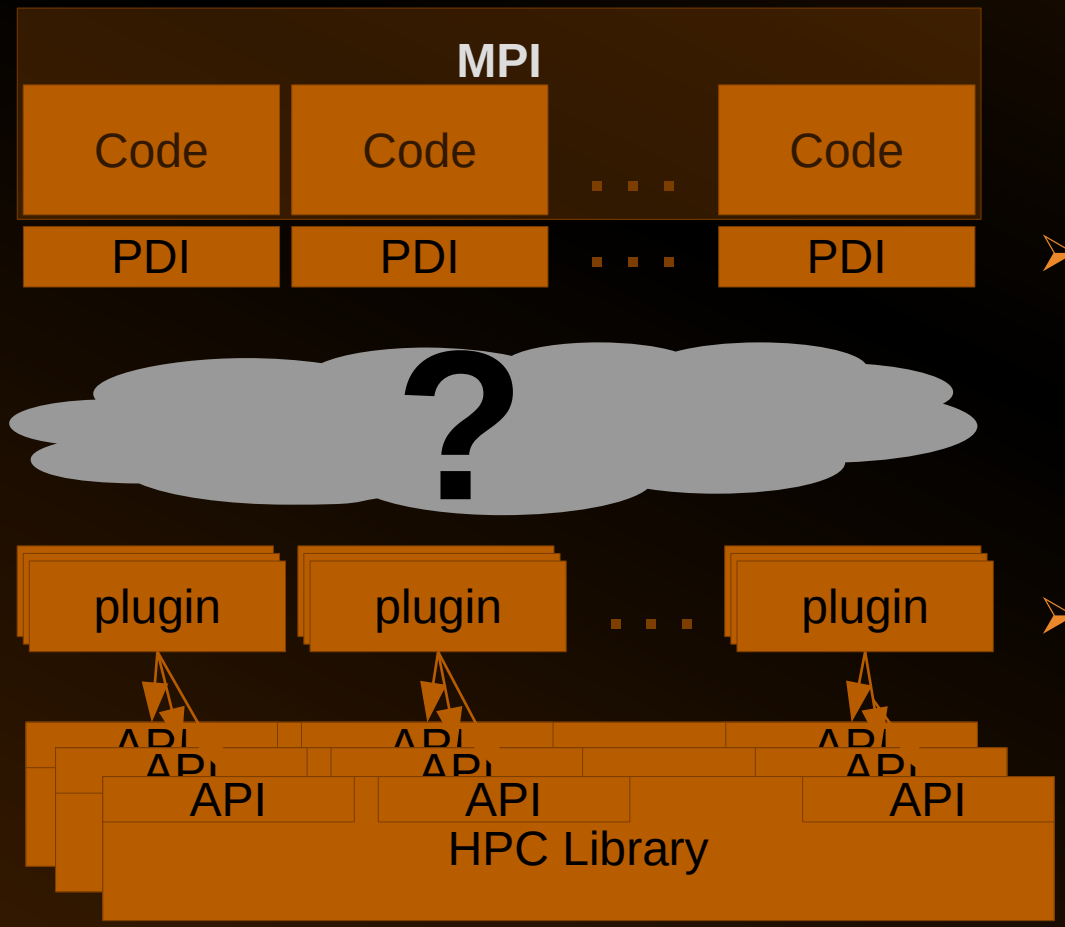
Finalization

- Final results,
- Final checkpoint,
- …

What is PDI?

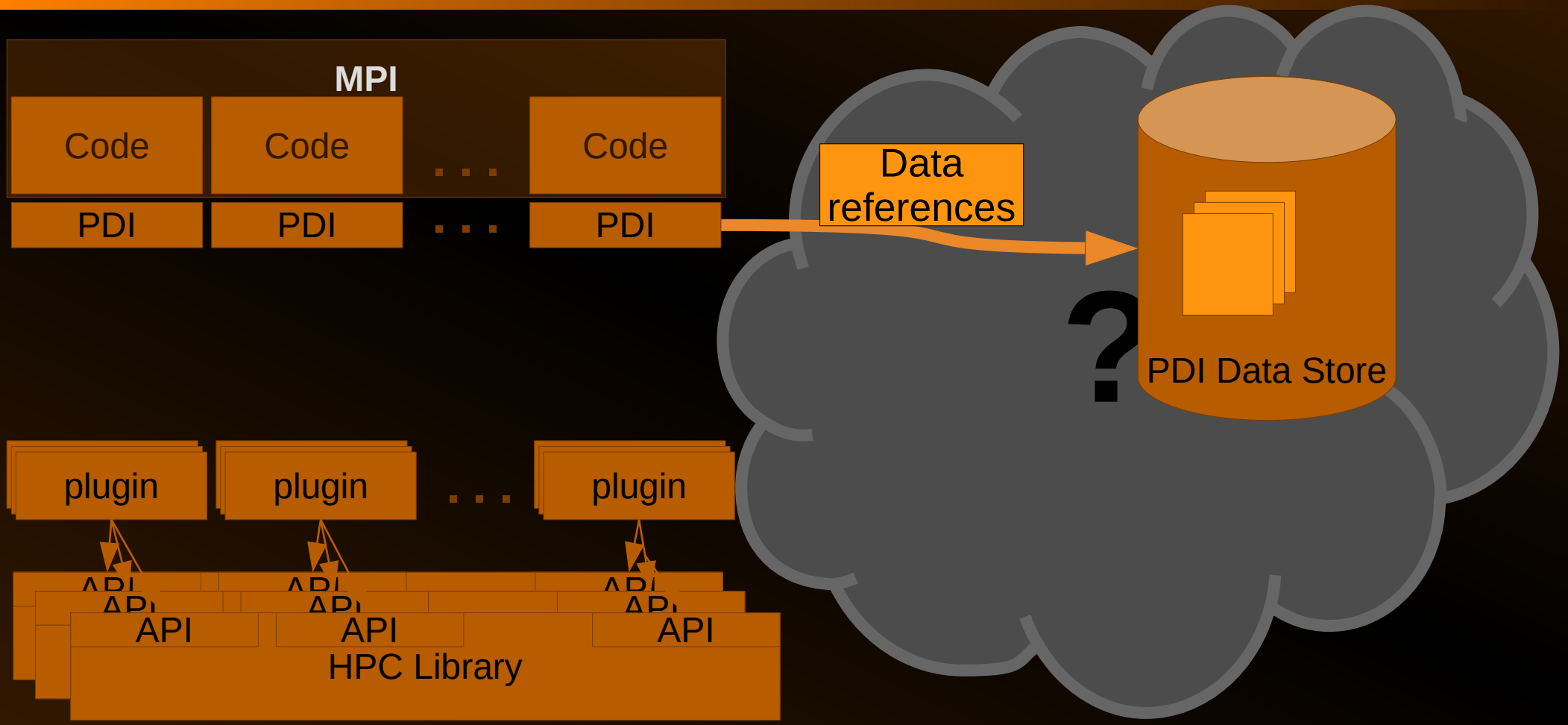

MPI
Code
Code
. . .
Code
?
API
API
API
API
API
API
API
API
API
HPC Library

What is PDI?

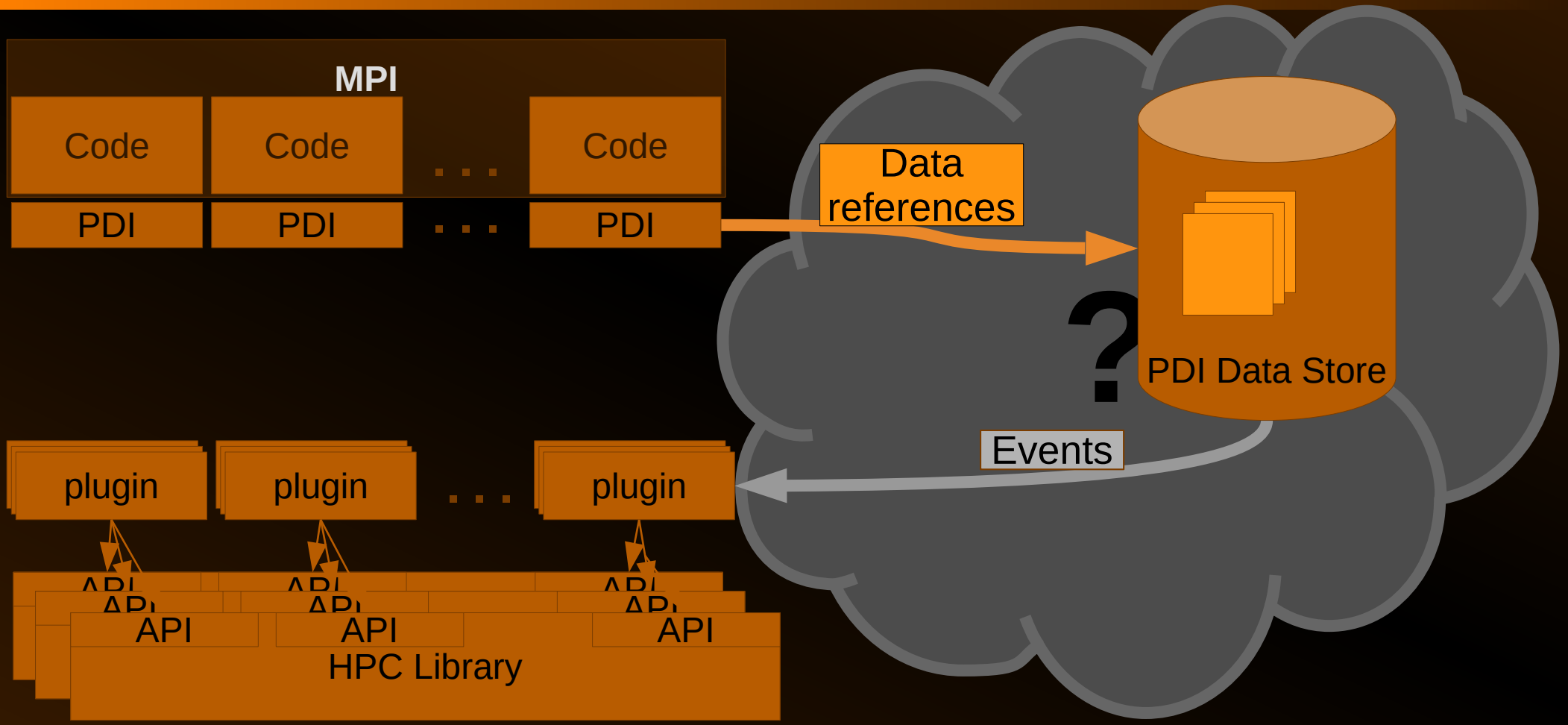


- PDI annotations: a purely declarative API
- Plugins for access to existing libraries

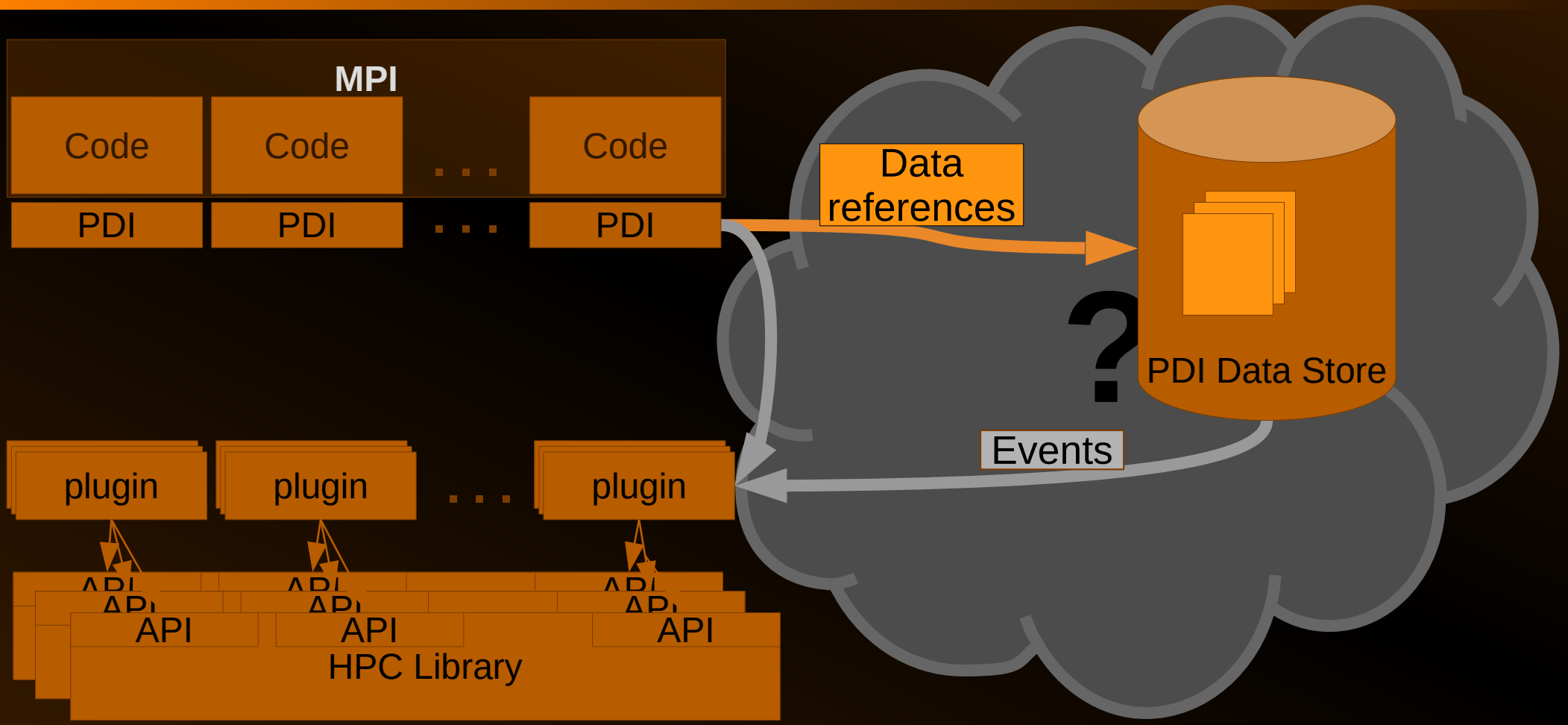
What is PDI?


MPI
Code
Code
. . .
Code
PDI
PDI
. . .
PDI
Data references
?
PDI Data Store
plugin
plugin
. . .
plugin
API
API
API
API
API
API
HPC Library

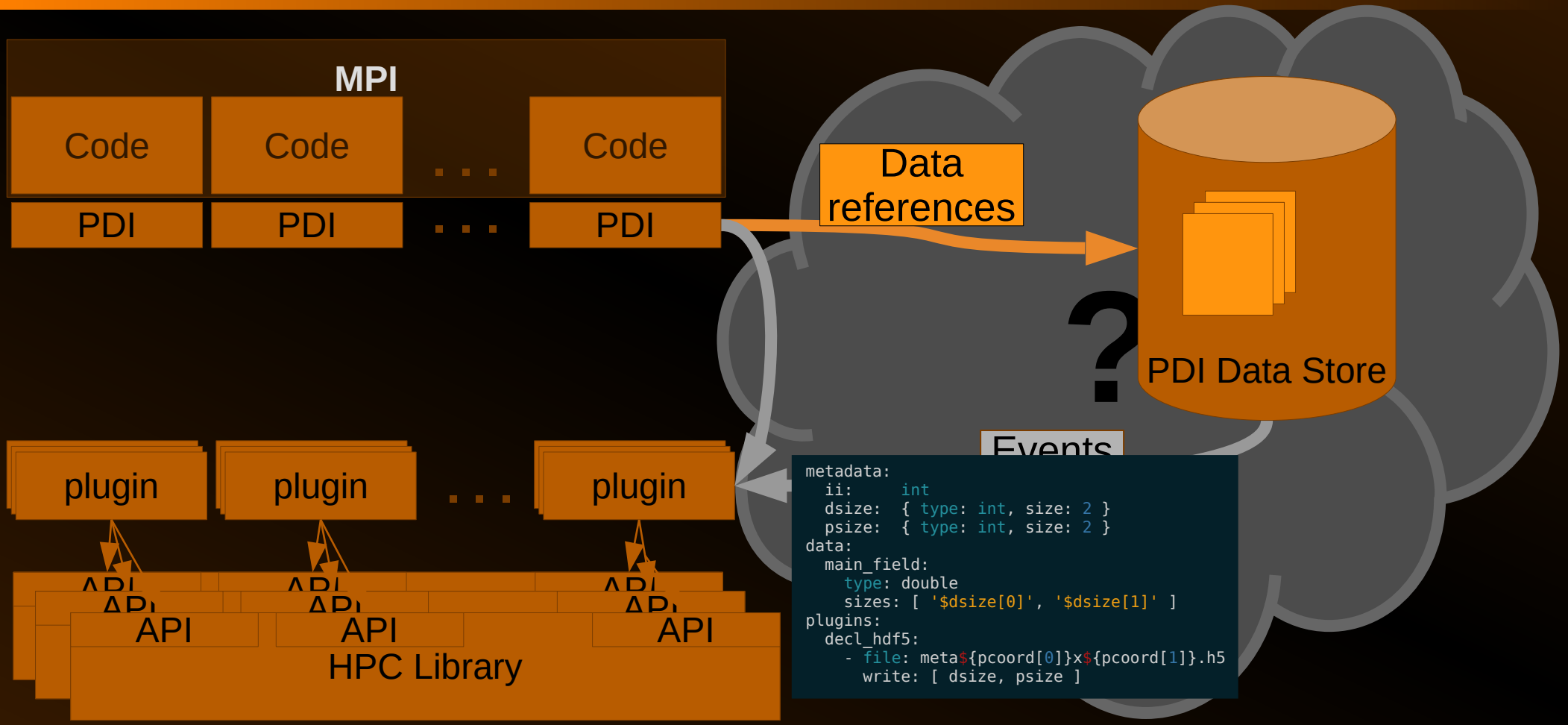
What is PDI?


MPI
Code
Code
Code
PDI
PDI
PDI
Data references
PDI Data Store
?
Events
plugin
plugin
plugin
API
API
API
HPC Library

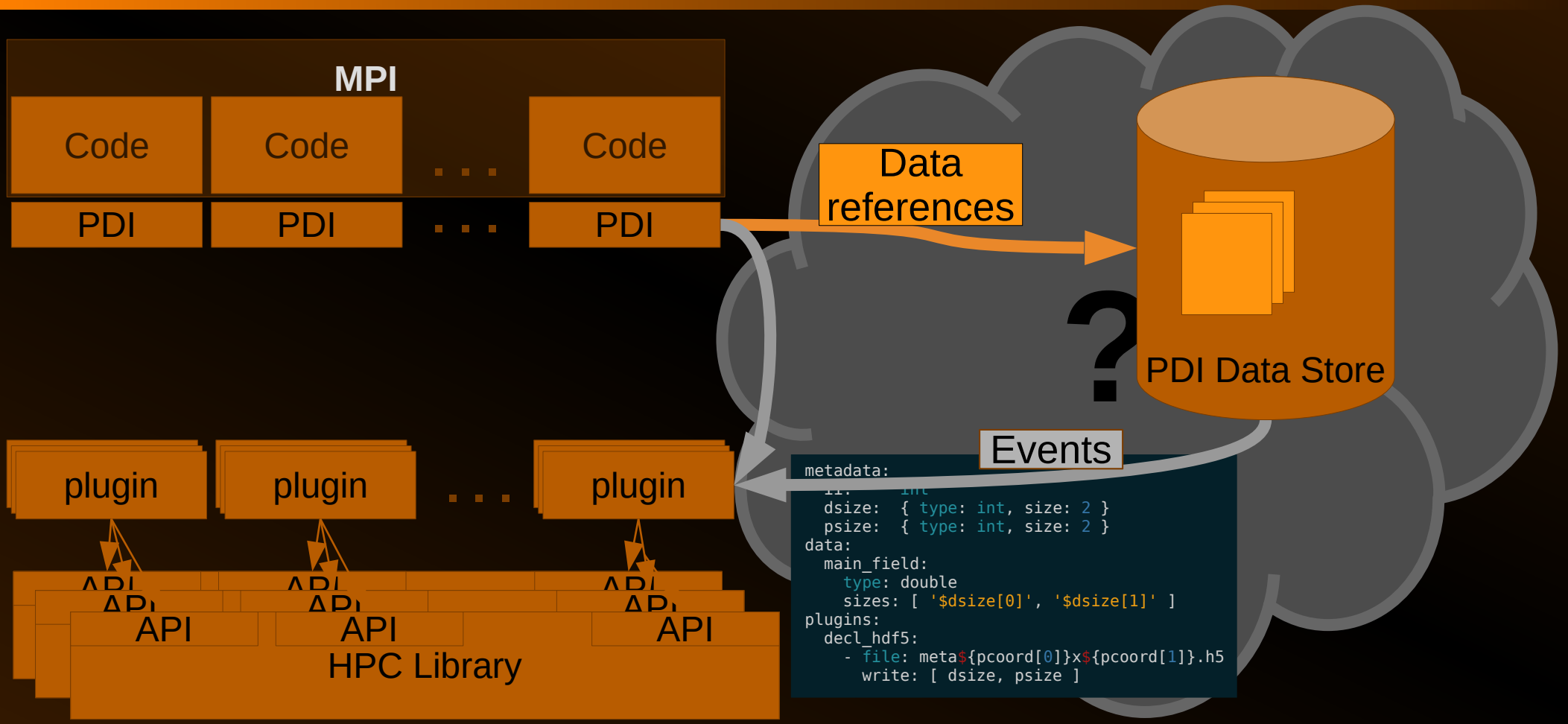
What is PDI?


MPI
Code
Code
Code
PDI
PDI
PDI
Data references
PDI Data Store
?
Events
plugin
plugin
plugin
API
API
API
HPC Library

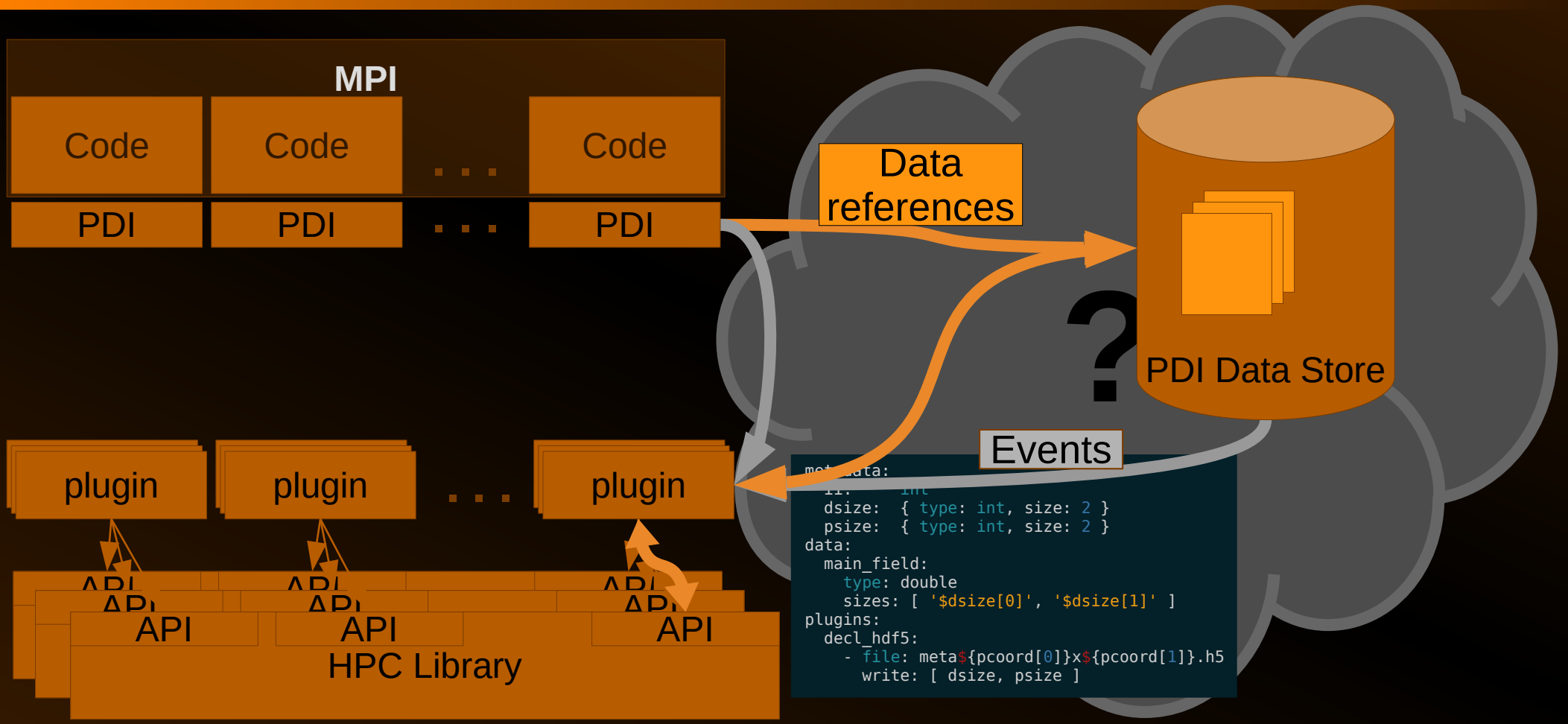
What is PDI?


MPI
Code
Code
Code
PDI
PDI
PDI
Data references
PDI Data Store
?
Events
plugin
plugin
plugin
API
API
API
HPC Library
metadata:
ii: int
dsize: { type: int, size: 2 }
psize: { type: int, size: 2 }
data:
main_field:
type: double
sizes: ['$dsize[0]', '$dsize[1]']
plugins:
decl_hdf5:
- file: meta${pcoord[0]}x${pcoord[1]}.h5
write: [dsize, psize]

What is PDI?

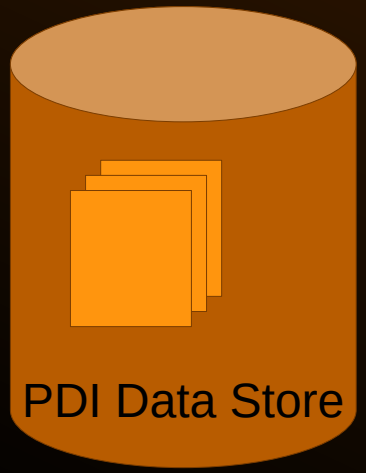

MPI
Code
Code
Code
PDI
PDI
PDI
Data references
PDI Data Store
?
Events
plugin
plugin
plugin
API
API
API
HPC Library
metadata:
dsize: { type: int, size: 2 }
psize: { type: int, size: 2 }
data:
main_field:
type: double
sizes: ['$dsize[0]', '$dsize[1]']
plugins:
decl_hdf5:
- file: meta${pcoord[0]}x${pcoord[1]}.h5
write: [dsize, psize]

What is PDI?


MPI
Code
Code
Code
PDI
PDI
PDI
Data references
PDI Data Store
?
Events
plugin
plugin
plugin
API
API
API
HPC Library
dsize: { type: int, size: 2 }
psize: { type: int, size: 2 }
data:
main_field:
type: double
sizes: ['$dsize[0]', '$dsize[1]']
plugins:
decl_hdf5:
- file: meta${pcoord[0]}x${pcoord[1]}.h5
write: [dsize, psize]

Inside PDI: The shared data store

- PDI data store: a map of buffer references
  - Name ⇒ unique identifier
  - Reference
    - Ownership & locking information
      - RW-lock: Single Writer / Multiple Readers
      - Memory ownership : Strong or Semi-weak
    - Type ⇒ memory layout and interpretation
    - Buffer address ⇒ pointer to user memory



Inside PDI: The data model

- Scalar
  - A size
  - An interpretation
- Array
  - Size
  - Padding
  - Subtype
- Record
  - Members
    - Type
    - offset
- Absolute reference
  - An address
  - A referenced type

- Covers C, Fortran, HDF5, MPI type systems
- Dynamic versions for python / C++ opaque types

PDI Annotation API

```
typedef enum { PDI_IN, PDI_OUT, PDI_INOUT } PDI_inout_t;

// A data buffer is ready (filled)
PDI_status_t PDI_share(const char *name, void *data, PDI_inout_t access);

// A buffer will be reused
PDI_status_t PDI_reclaim(const char *name);

// An interesting location in the code has been reached
PDI_status_t PDI_event(const char* event);
```

a C / C++ API
Also available for:

- Fortran
- Python

- Share
  - A buffer is in a coherent consistent state
  - Reference in PDI store
- Reclaim
  - The buffer will be reused for a different use
  - Un-reference in Store

PDI Annotation API usage

```
double* data_buffer = malloc( buffer_size*sizeof(double) );

while ( !computation_finished )
{
    compute_the_value_of( data_buffer, /*...*/ );
    PDI_share("main_buffer", data_buffer, PDI_OUT);
    do_something_without_data_buffer();
    do_something_reading( data_buffer, /*...*/ );
    PDI_reclaim("main_buffer");
    update_the_value_of( data_buffer, /*...*/ );
}
```

buffer is shared
- between here

…
- and here

- Creates a “shared region” in code where
  - Data referenced in PDI store
  - Plugins can use it
- Code should refrain from
  - modifying it (**PDI_IN|OUT**)
  - accessing it (**PDI_IN**)

PDI type specification in YAML

```yaml
metadata:
  buffer_size: int
data:
  main_buffer: { type: array,  subtype: double, size: $buffer_size } }
```

- Data type is specified in YAML
  - For “static” languages (C/C++/Fortran)
    - For dynamic languages (Python) type info is auto-extracted
- Type expressed as a template
  - Values expressed as $-expressions
- Metadata: tell PDI to keep a copy

Inside PDI: $-expressions

- A simple expression language that supports:
  - literals
  - references to the store
  - operations
    - the same operators as C (+, -, /, *, %, ==, !=, …)
- A $-expression can represent
  - Scalars: strings, integers, floating-point
  - Array / list, Record / mapping

Inside PDI: In summary

- PDI data store: a map of buffer references
  - Name ⇒ unique identifier
  - Reference
    - RW-lock & Ownership information
    - Type ⇒ memory layout and interpretation
    - Buffer address ⇒ pointer to user memory
- The user interface
  - Type templates
  - $-expressions

PDI approach: wrap-up

In code

- Write code
- Annotate buffers availability (share / reclaim)
- Compile and… DONE! (on the code side)

In YAML

- Describe shared data
- Use pre-made plugins or write your own code to choose I/O libraries, describe behavior
  - React to events
  - Access data in the store

PDI: Existing plugins

- IO libraries
  - HDF5 / parallel HDF5, NetCDF4 / pNetCDF4, SIONlib
- Special purpose IO
  - FTI (see next talk), SENSEI (WIP)
- Workflow integration
  - Dask (WIP), FlowVR, Melissa (WIP)
- Your own code
  - $-expressions based language, Python, C, C++, Fortran

PDI in practice: Decl'HDF5

```
PDI_share("buffer_size", &buffer_size, PDI_OUT);
double* data_buffer = malloc( buffer_size*sizeof(double) );
PDI_reclaim("buffer_size");

while ( iteration_id < max_iteration_id )
{
    compute_the_value_of( data_buffer, /*...*/ );
    PDI_share("main_buffer", data_buffer, PDI_OUT);
    do_something_reading( data_buffer, /*...*/ );
    PDI_reclaim("main_buffer");
}
```

- Write data in the HDF5 format
- Heavily relies on
  - $-expressions
  - default configuration values
- Makes
  - *Simple* things **easy**
  - *Complex* things **possible**

Decl'HDF5: the YAML

```yaml
metadata: { buffer_size: int, iteration_id: int, rank: int }
data: { main_buffer: { type: array,  subtype: double, size: $buffer_size } }
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x${rank}.h5'
    write: main_buffer
```

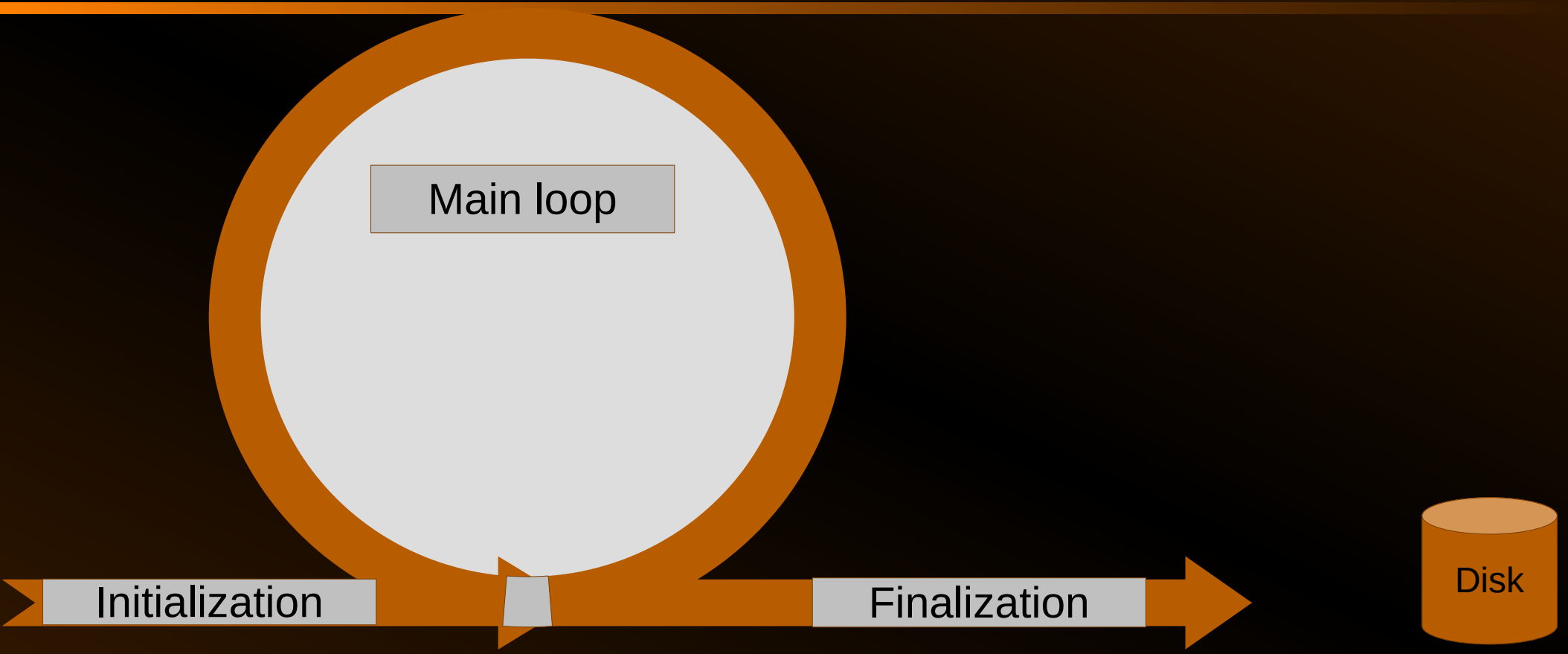
- Simple to just dump data as HDF5

Decl'HDF5: a complex YAML example

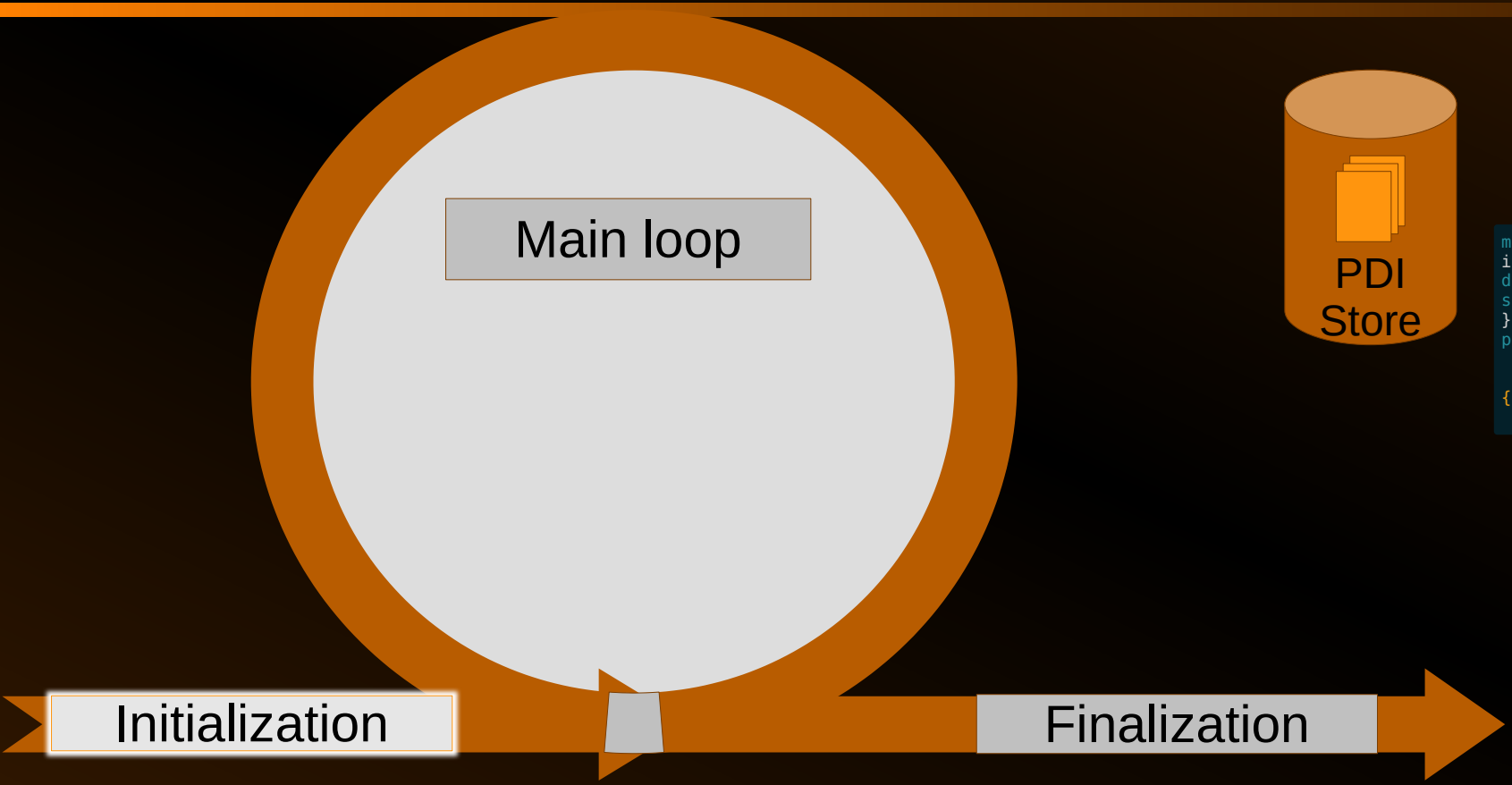
```yaml
metadata: { buffer_size: int, iteration_id: int, rank: int, np: int }
data: { main_buffer: { type: array,  subtype: double, size: $buffer_size } }
plugins:
  decl_hdf5:
    file: 'my_file.h5'
    when: '$iteration_id % 100 = 0 & $iteration_id < 10000'
    datasets:
      main_dset:
        type: array
        subtype: double
        Size: [ '($buffer_size - 2) * $np', 100 ]
    write:
      main_buffer:
        memory_selection: { start: 1, size: '$buffer_size - 2' }
        dataset: main_dset
        dataset_selection:
          start: [ '($buffer_size - 2) * $iteration_id', '$iteration_id/100' ]
          size: [ '$buffer_size - 2', 1 ]
    communicator: $MPI_COMM_WORLD
  mpi:
```

- Possible to do complex rearranging of data in parallel

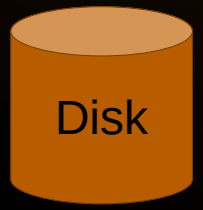
PDI: behind the scene



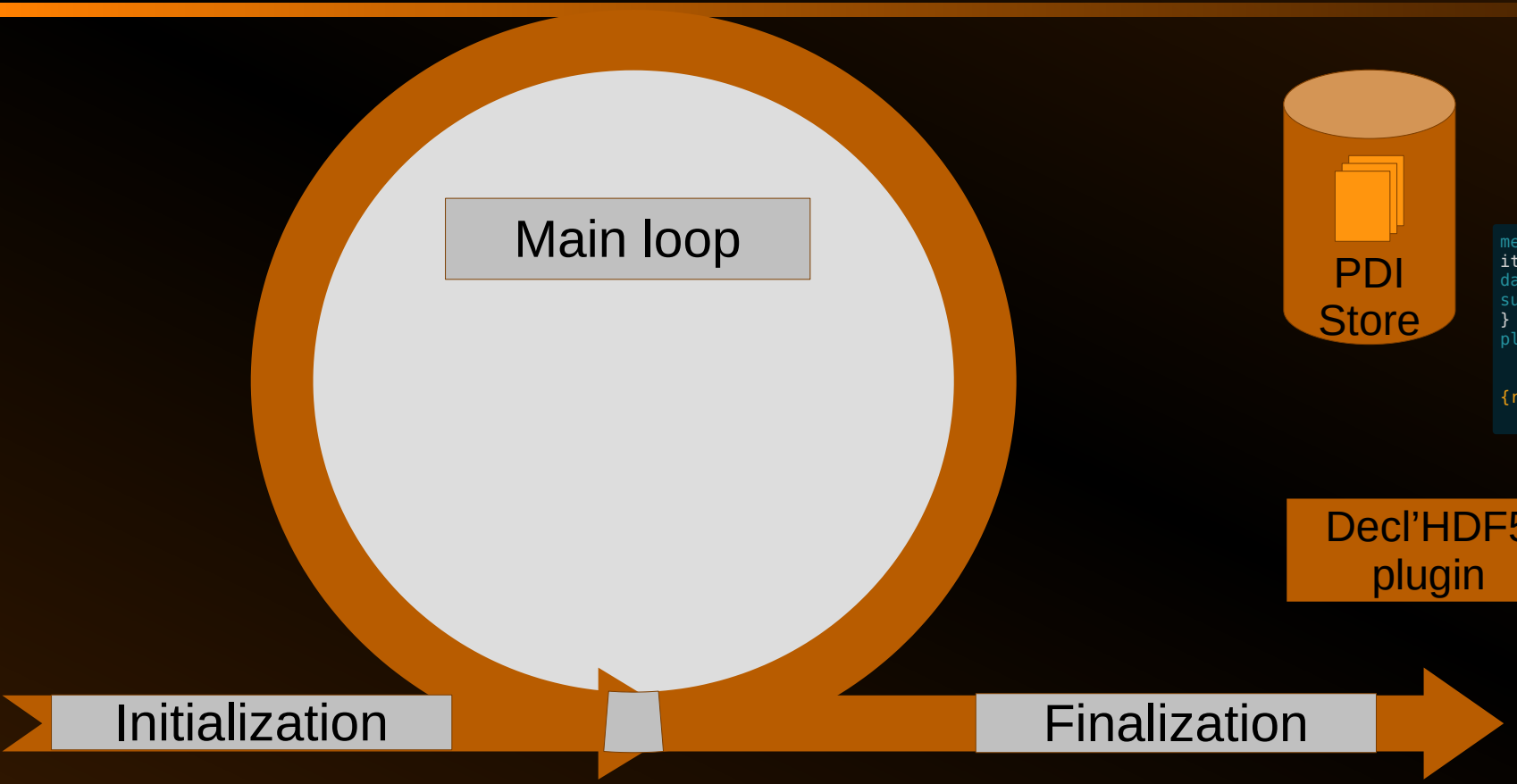
PDI: behind the scene




```
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

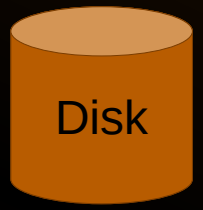


PDI: behind the scene

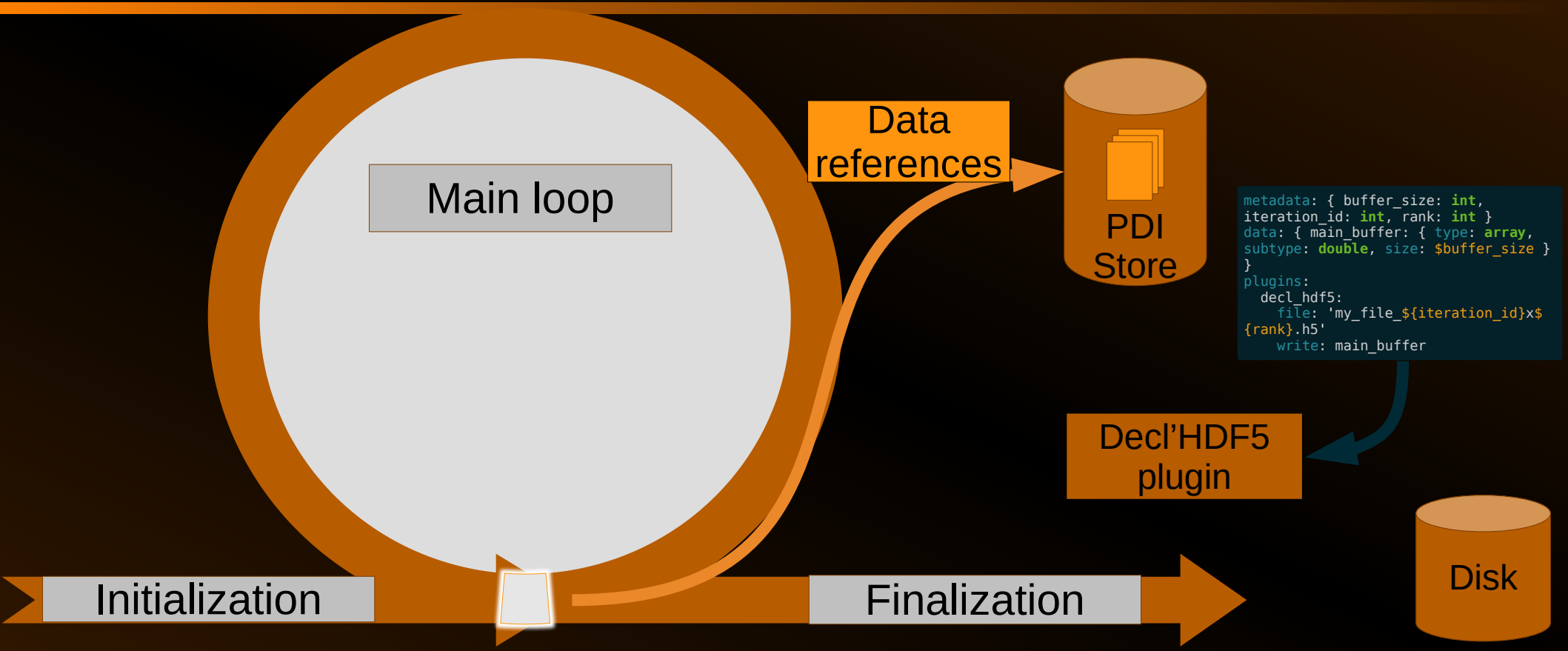



```
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

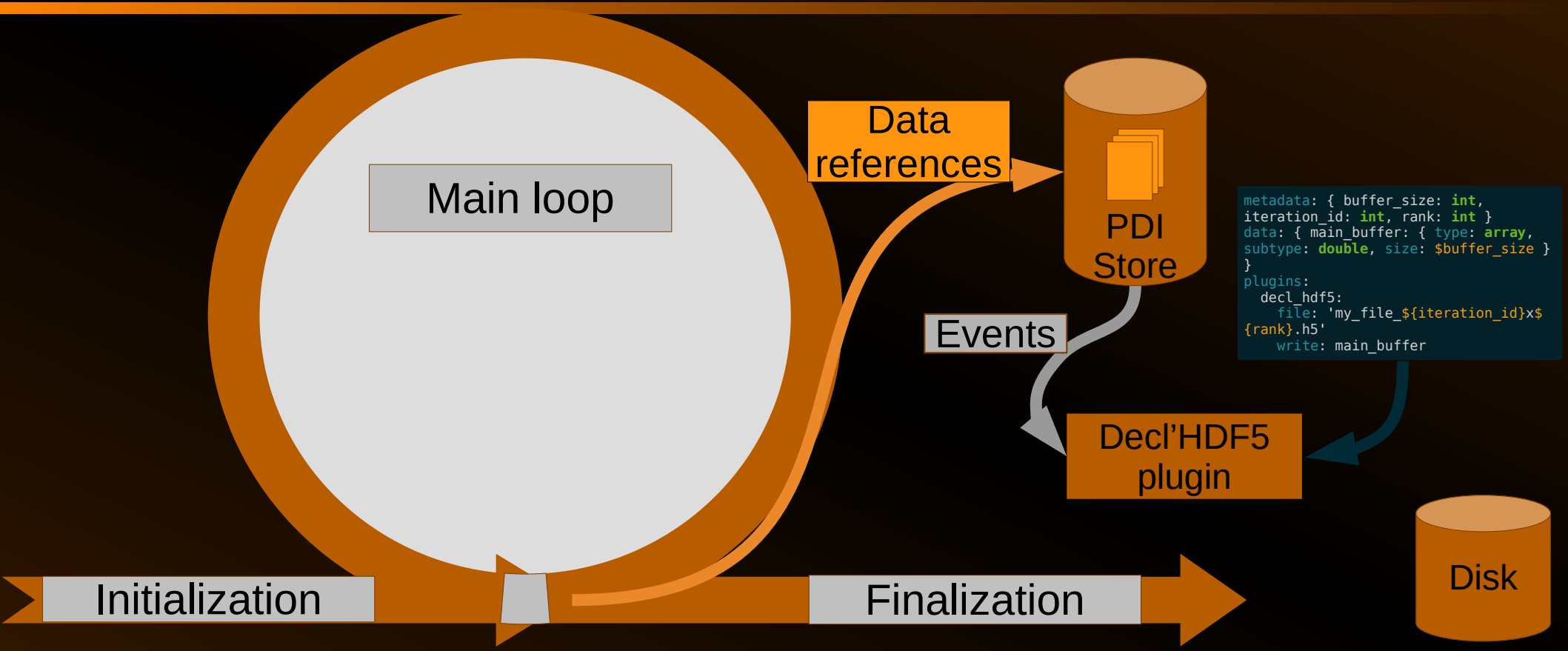
Decl'HDF5 plugin



PDI: behind the scene

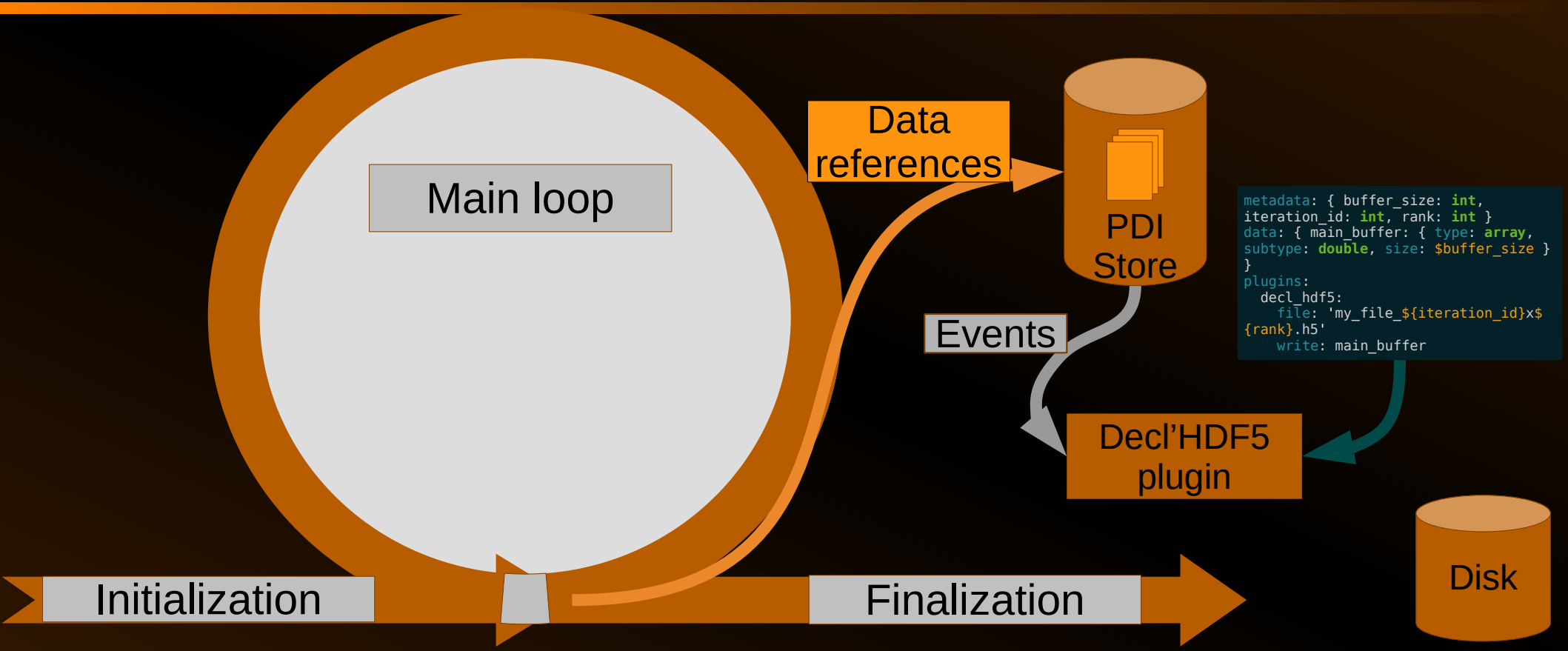


PDI: behind the scene



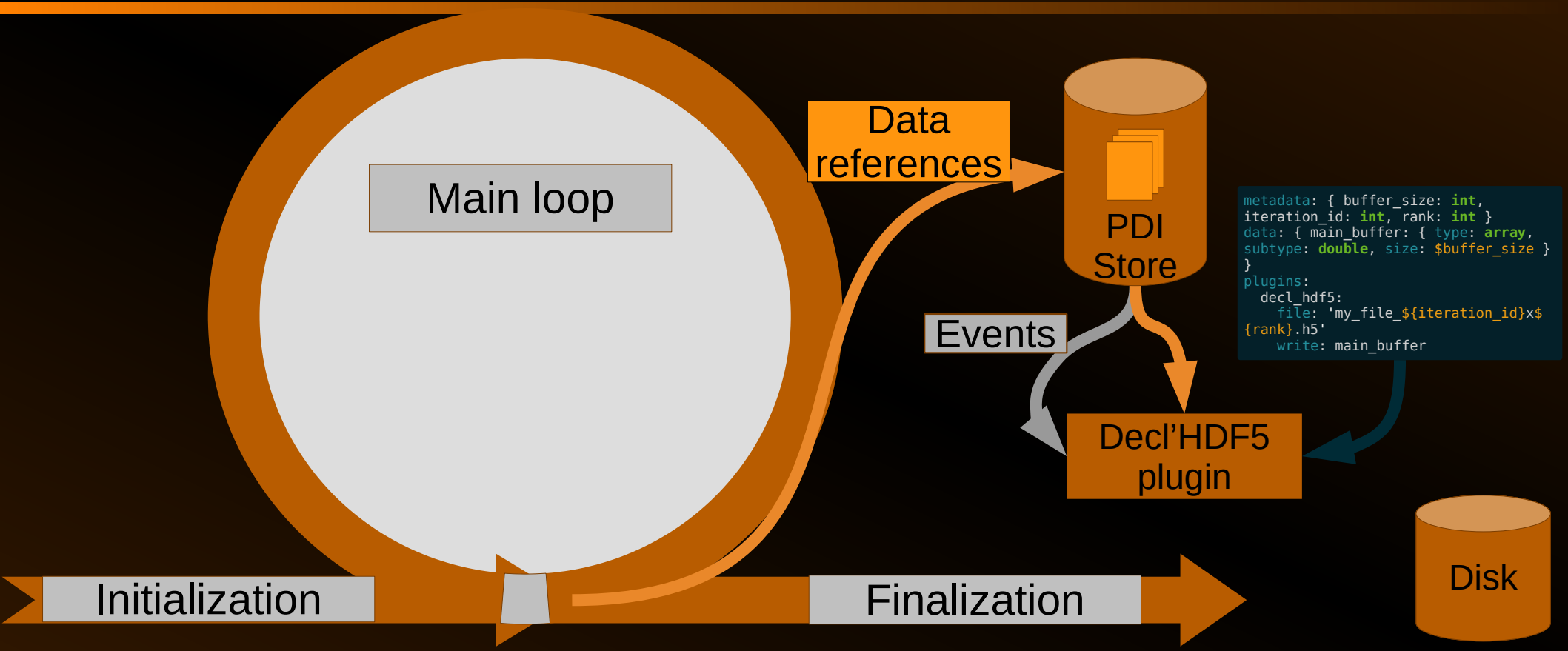

```
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

PDI: behind the scene



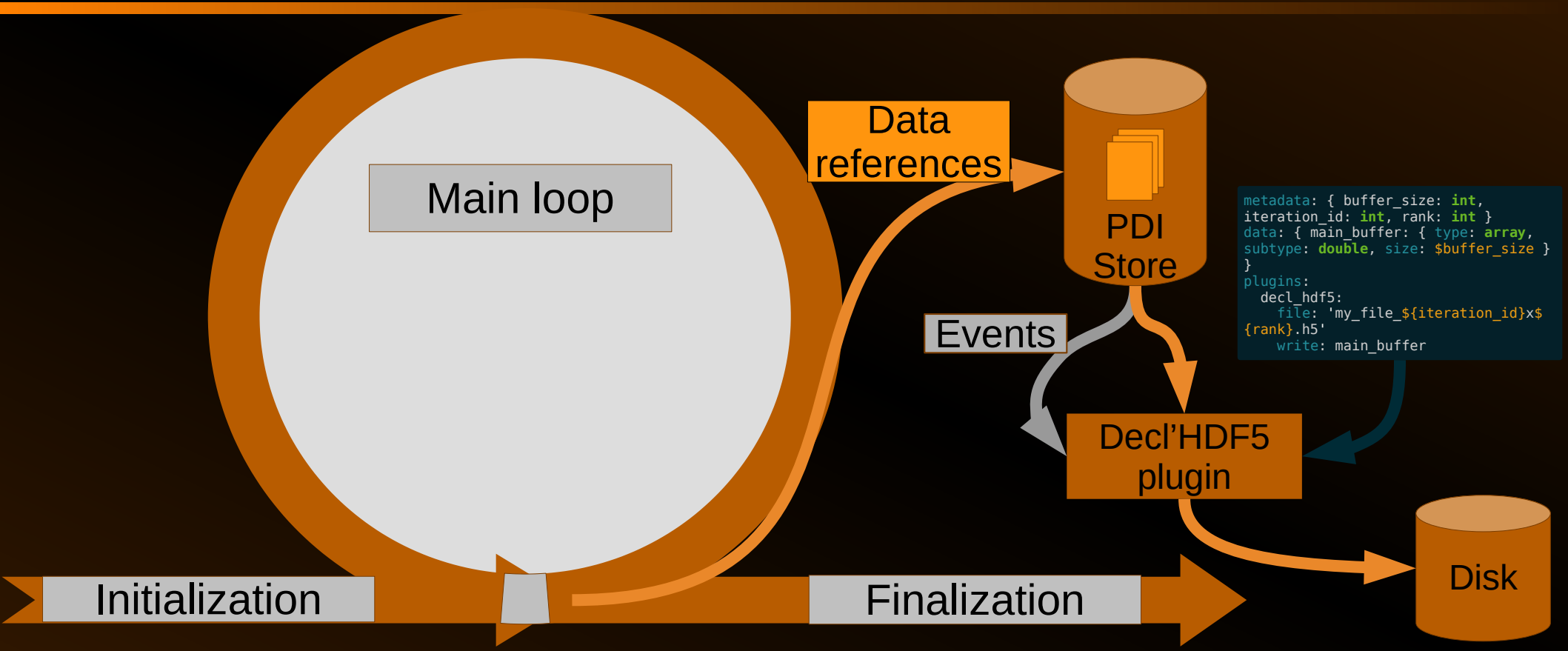

```
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

PDI: behind the scene




```
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

PDI: behind the scene




```
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```

Data coupling with PDI: pycall

```
plugins:
  pycall:
    on_event:
      trigger_event_name: # event that triggers the call
        with: { iter: $iteration_id, original_data: $main_field }
        exec: |
          if iter<1000:
               new_data = original_data*4 # uses numpy
               pdi.expose('new_data', new_data, pdi.OUT);
```

- Let you call your own Python code
  - Data is exposed as numpy arrays
  - Numpy arrays can be re-exposed

    ⇒ In-process post-processing and data transformation

Data coupling with PDI: user-code

```
plugins:
  user_code:
    on_event:
      trigger_event_name: # event that triggers the call
        function_name { in1: $iteration_id, in2: $main_field }
```

```
void function_name(void)
{
     int* iter = NULL; PDI_access("in1", &iter, PDI_IN);
     double* main_field = NULL; PDI_access("in2", &iter, PDI_IN);
     // …
     PDI_release("in2");
     PDI_release("in1");
}
```

- Let you call your own (C/Fortran) functions
  - When performance matters
  - To call library APIs not covered by plugins

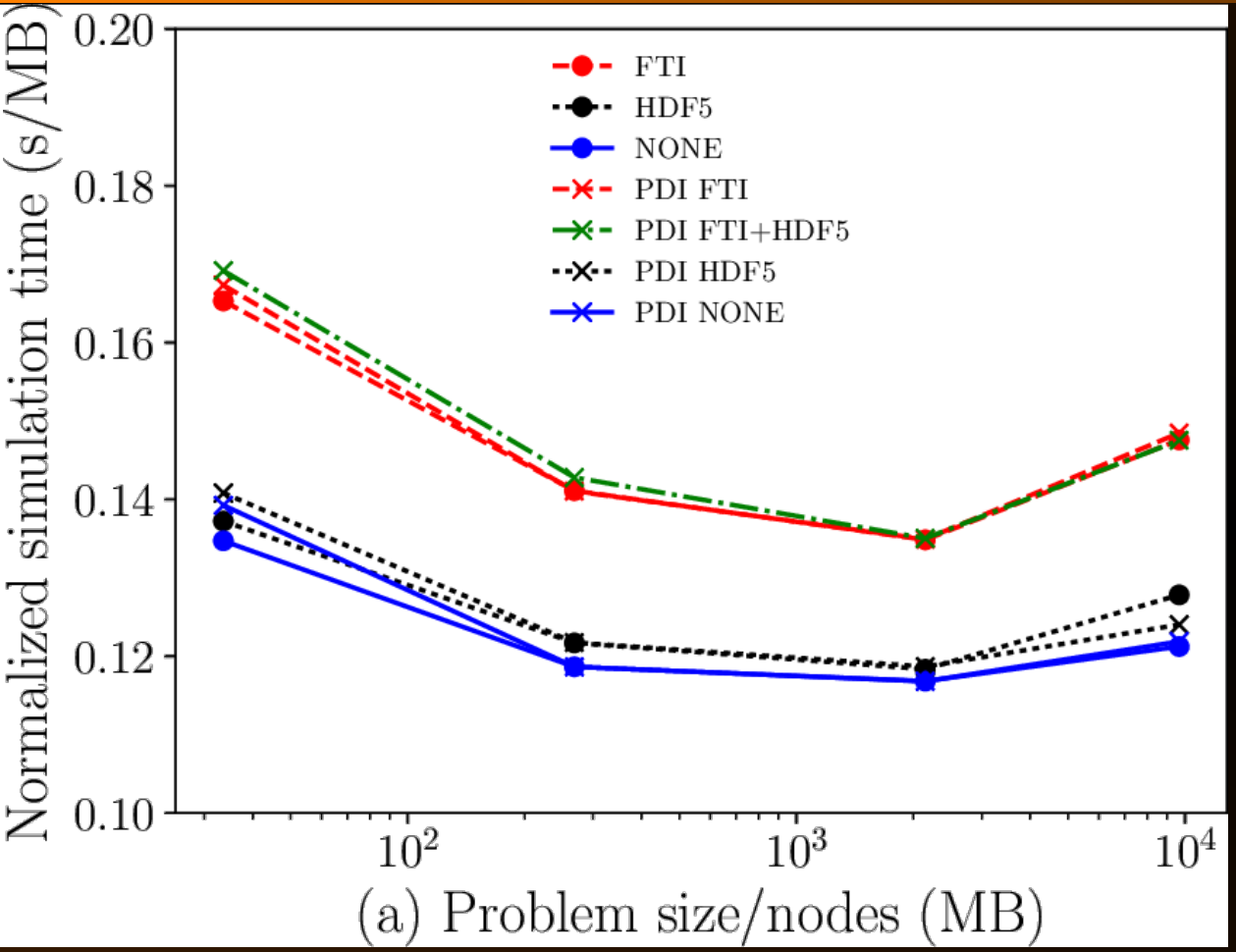
Writing a plugin is easy

1 Class + 1 Macro

```
struct <plugin_name>_Plugin: Plugin {
  // Implement the plugin
};
// Declare the plugin to PDI
PDI_PLUGIN(<plugin_name>)
```

- Register on events
  - Share
  - Reclaim
  - Named events
- Read your plugin sub-tree in YAML
  - A simple API for $-expression access
- Access data references as smart-pointers

PDI: Perf. Evaluation 1/2



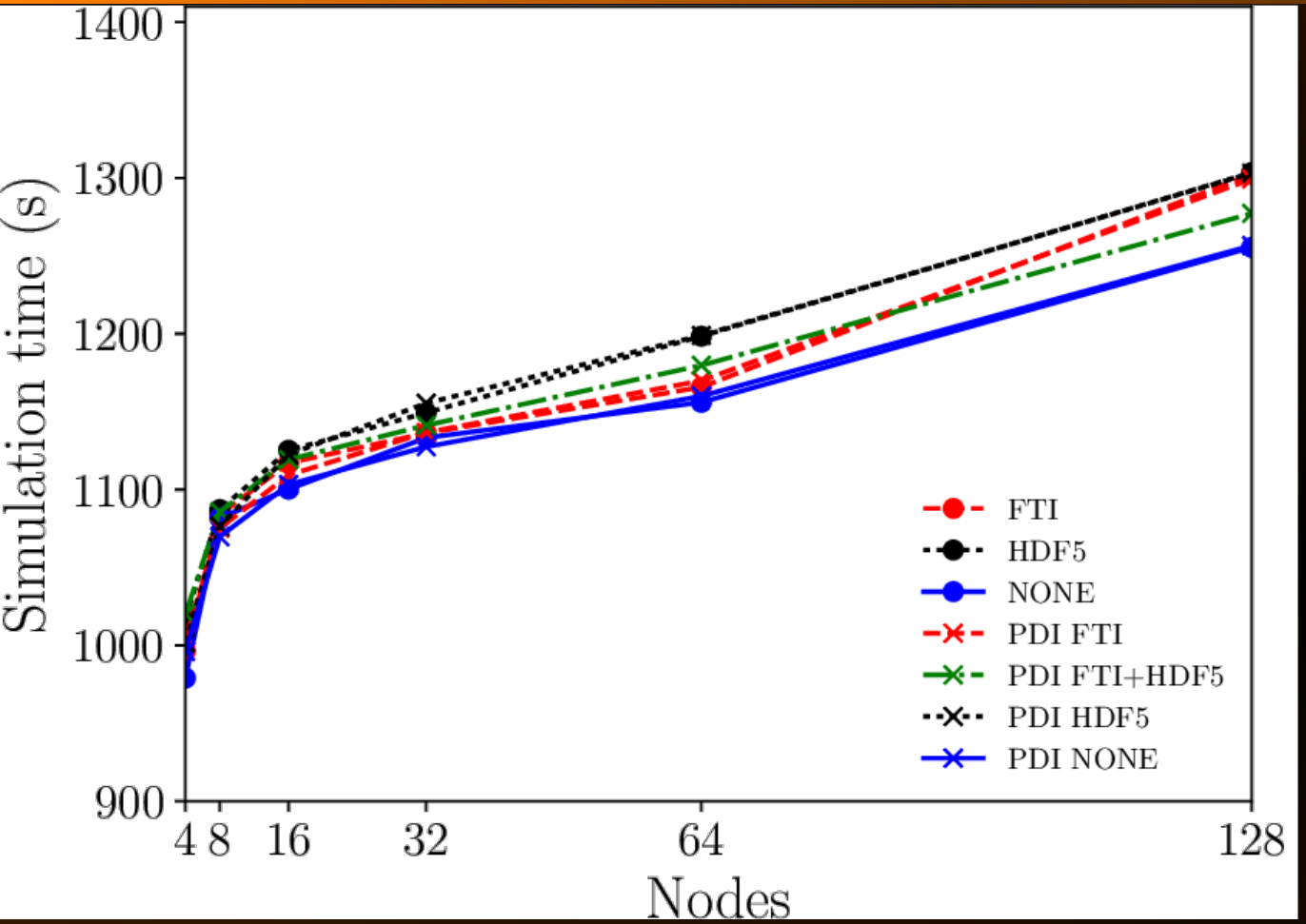
Corentin Roussel (MdlS)
Kai Keller (BSC)

- 4 versions of Gysela
  - No checkpoint
  - HDF5 checkpoints
  - FTI fault-tolerance
  - PDI (none / HDF5 / FTI / HDF5+FTI)

Execution time by MB of checkpointed data on 4 MareNostrum Nodes with and without PDI

PDI: Perf. Evaluation 2/2

Corentin Roussel (MdlS)
Kai Keller (BSC)

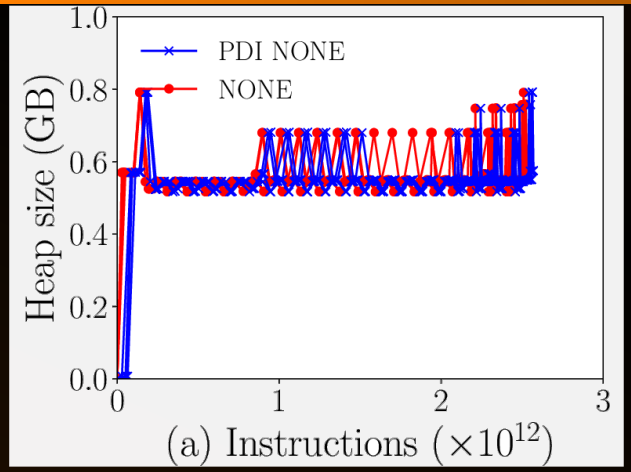


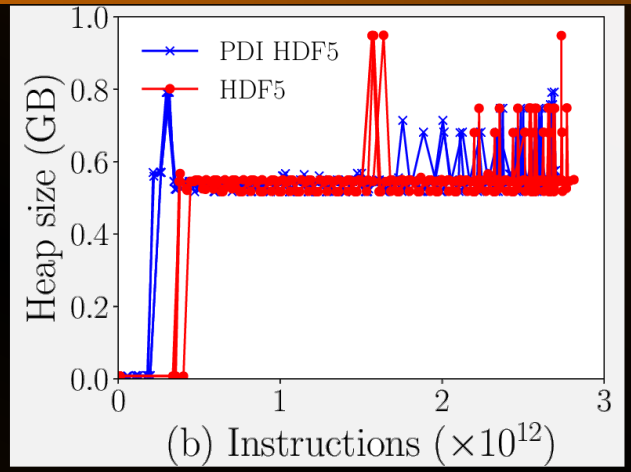
Gysela Wallclock time in weak scaling on Curie (TGCC – France) with and without PDI

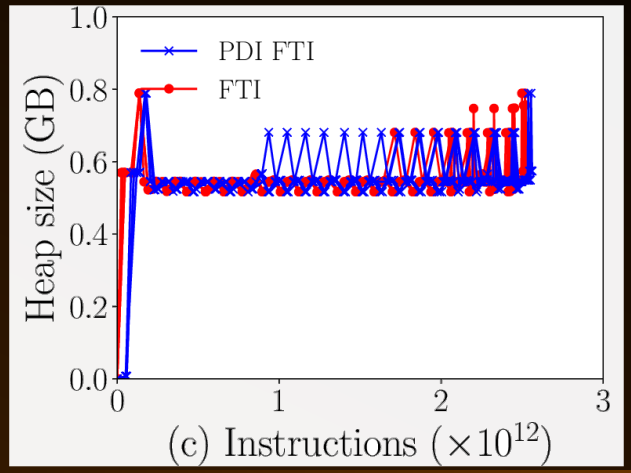
Checkpointed data ~2.1GB/node

PDI: Memory overhead

Corentin Roussel (MdlS)
Kai Keller (BSC)

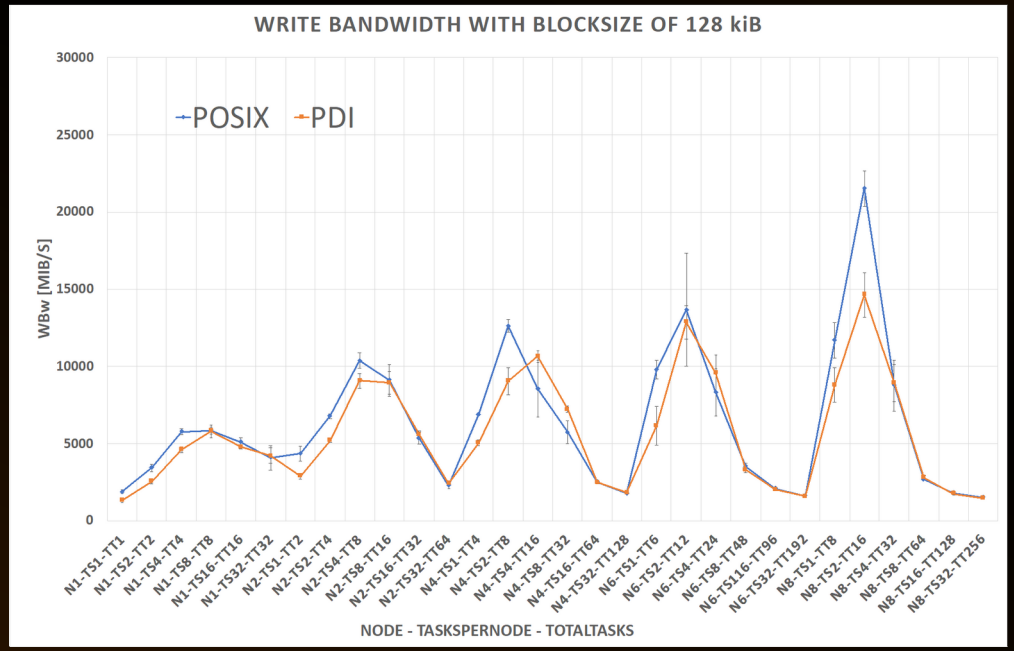


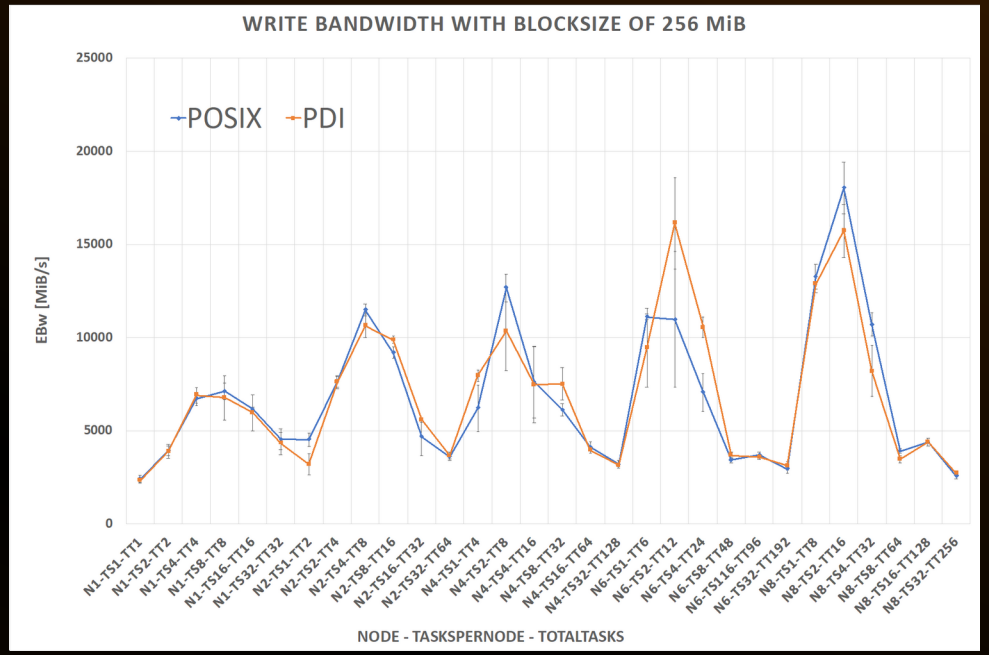




Memory usage during a Gysela execution with and without PDI on 4 nodes of MareNostrum (BSC – Spain)

Perf evaluation: IOR





IOR IO Benchmark PDI integration

Scaling with small (128k) & large (256M) data blocks on CRESCO6 @ ENEA

Francesco Iannone (ENEA)

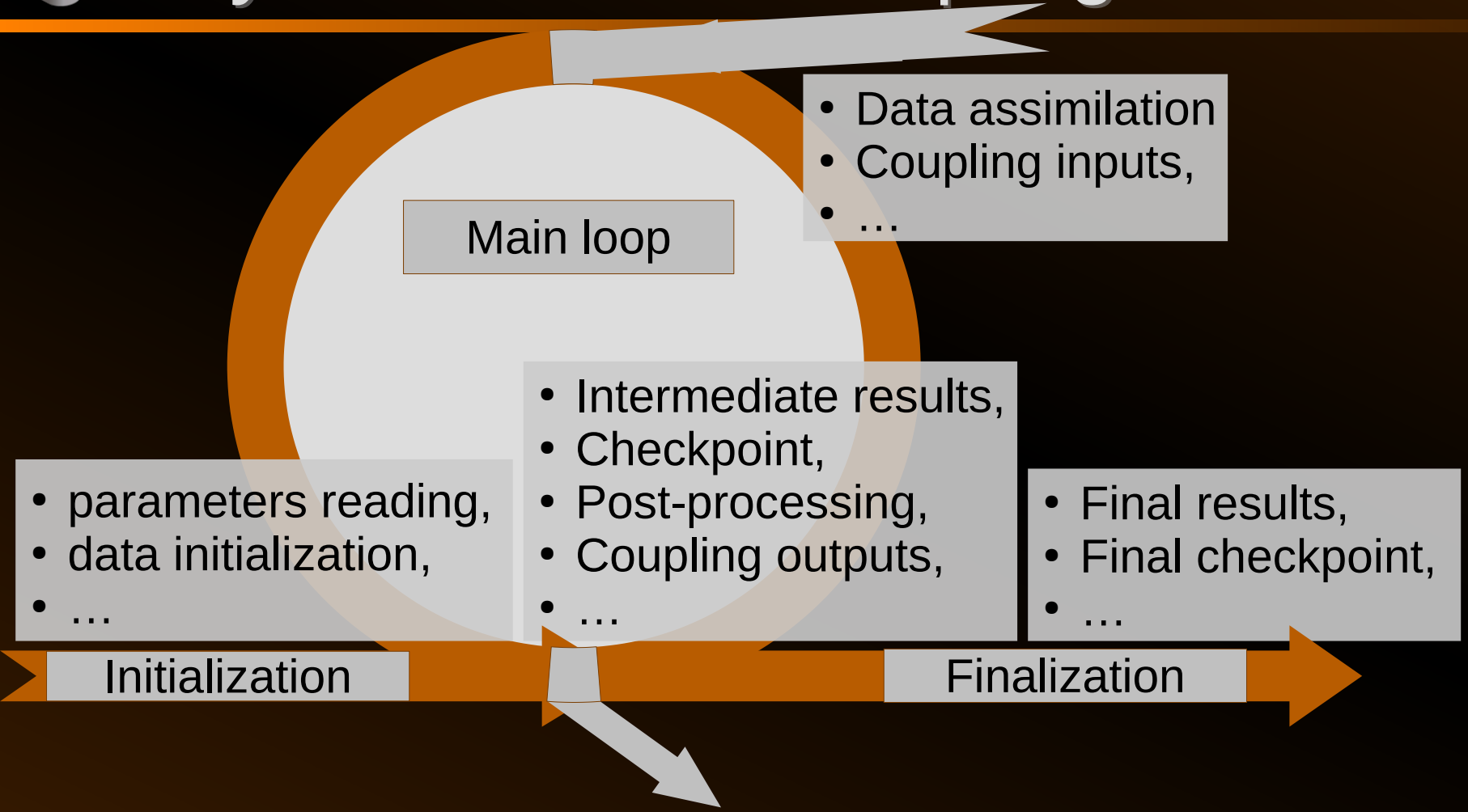
PDI In practice

- PDI is publicly available (BSD 3-clause license)
  - Version 1.0.0 just released (2021-02-02)
  - Packages available for Debian, Fedora, Ubuntu, Spack
  - Documentation available @ https://pdidev.github.io/1.0/
  - Heavily tested & validated
    - more than 700 tests
    - more than 14 platforms
  - PRACE Training offered next week
    - https://events.prace-ri.eu/event/1121/
- Integration in production codes
  - Gysela, Parflow, ESIAS, Metalwalls (Planned & funded)

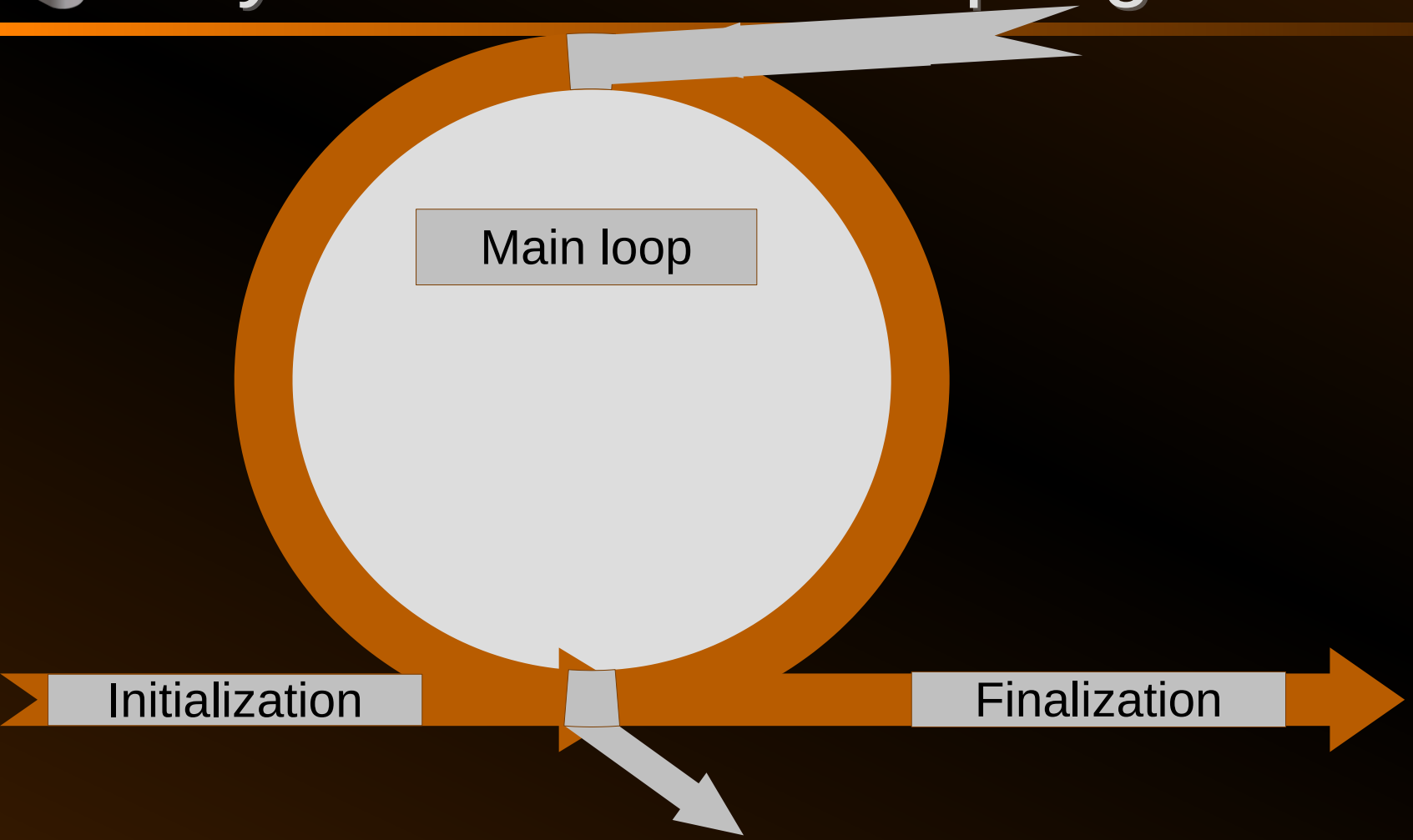
Conclusion

- A library for Data Coupling
  - Not an IO library
- Supports separation of concern w. negligible overheads
- Current work in progress
  - More plugins, especially coupling-related
    - Dask (Amal Gueroudji PhD. thesis), Melissa (Sebastian Friedemann), SENSEI (Christian Witzler)
  - Extraction of the type-related library
    - Needed in many libraries (FTI, HDF5, MPI, …) usually read-only
    - Can we get most of the information from the compiler?

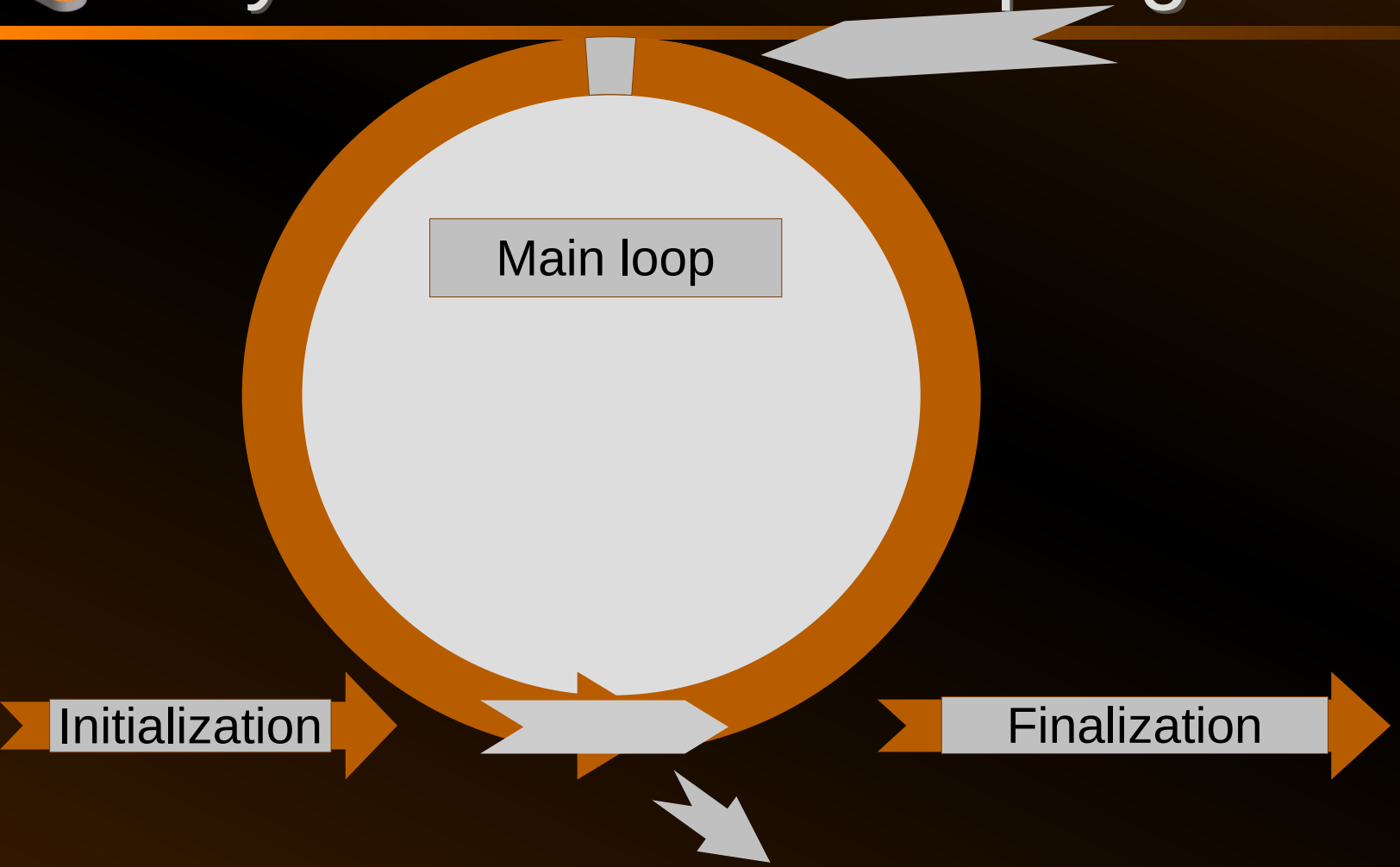
Beyond I/O: data coupling



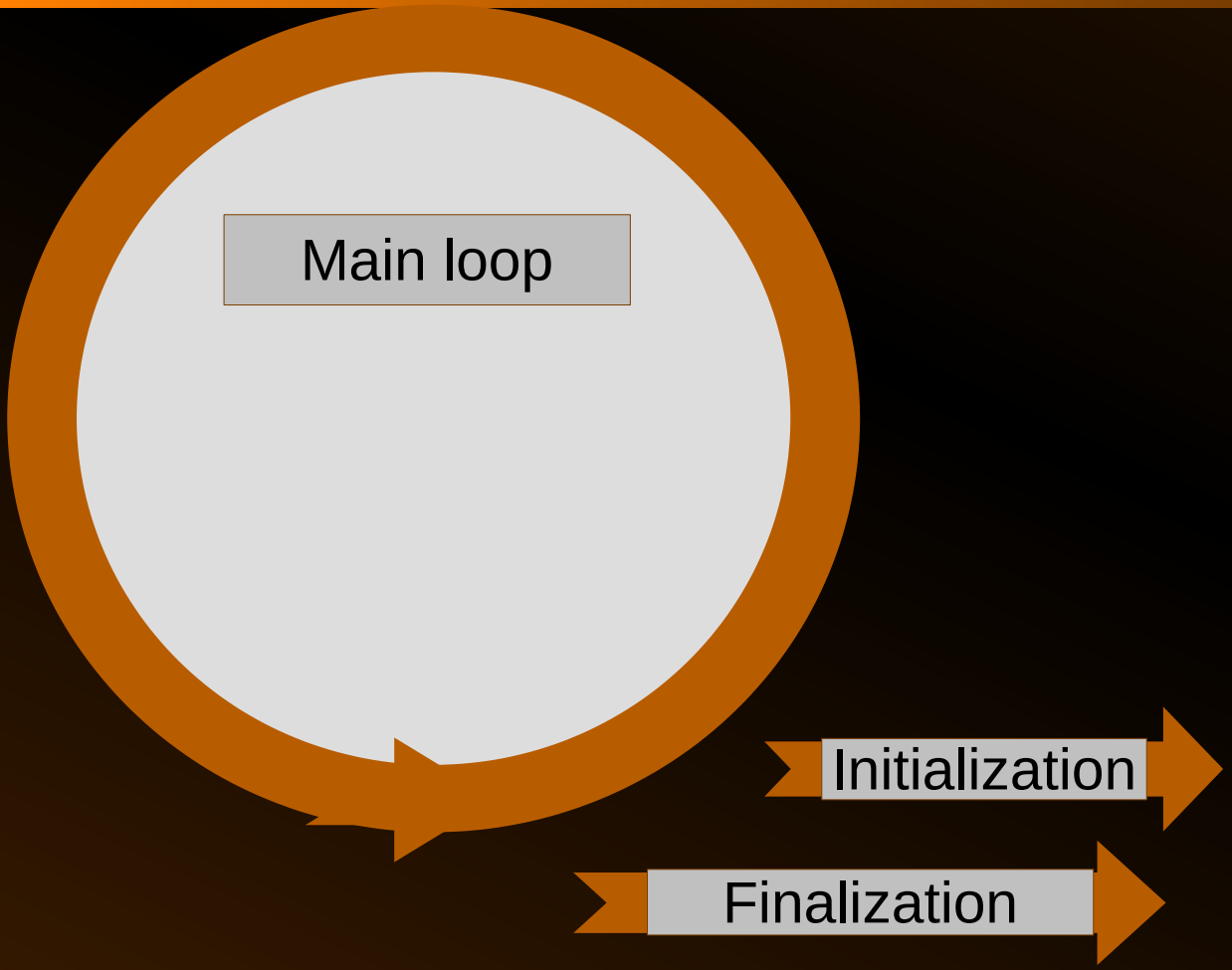
Beyond I/O: data coupling



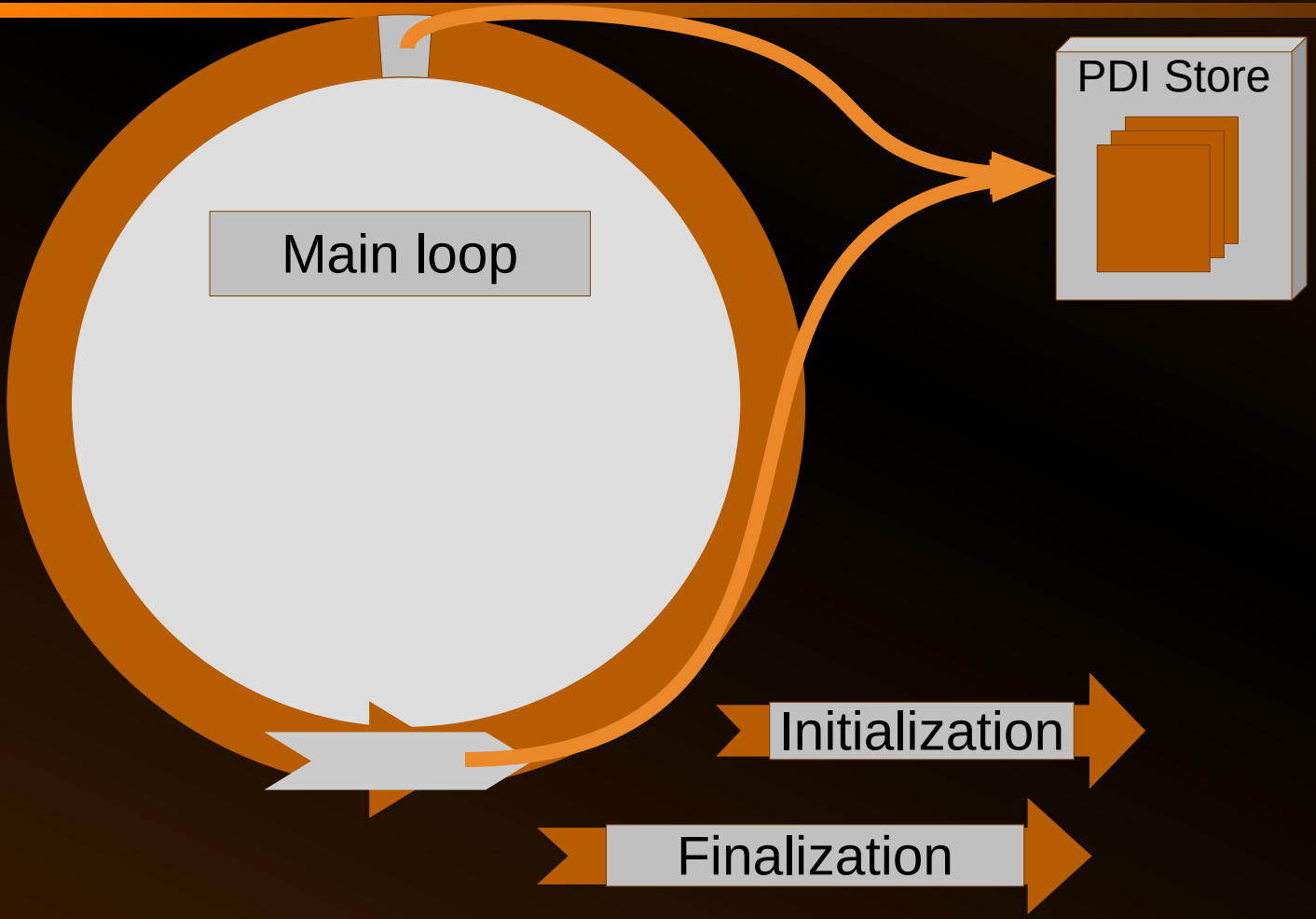
Beyond I/O: data coupling



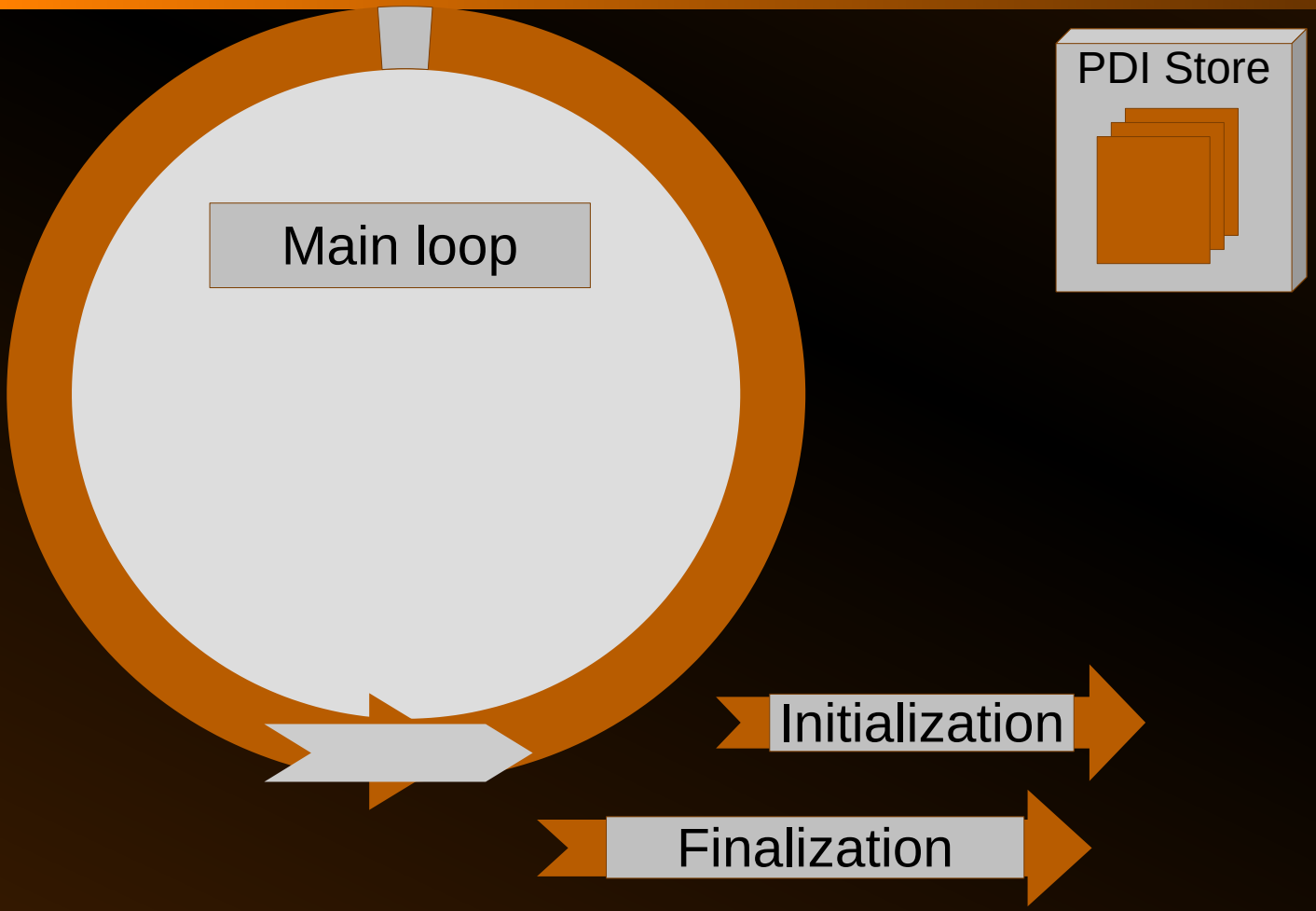
Beyond I/O: data coupling



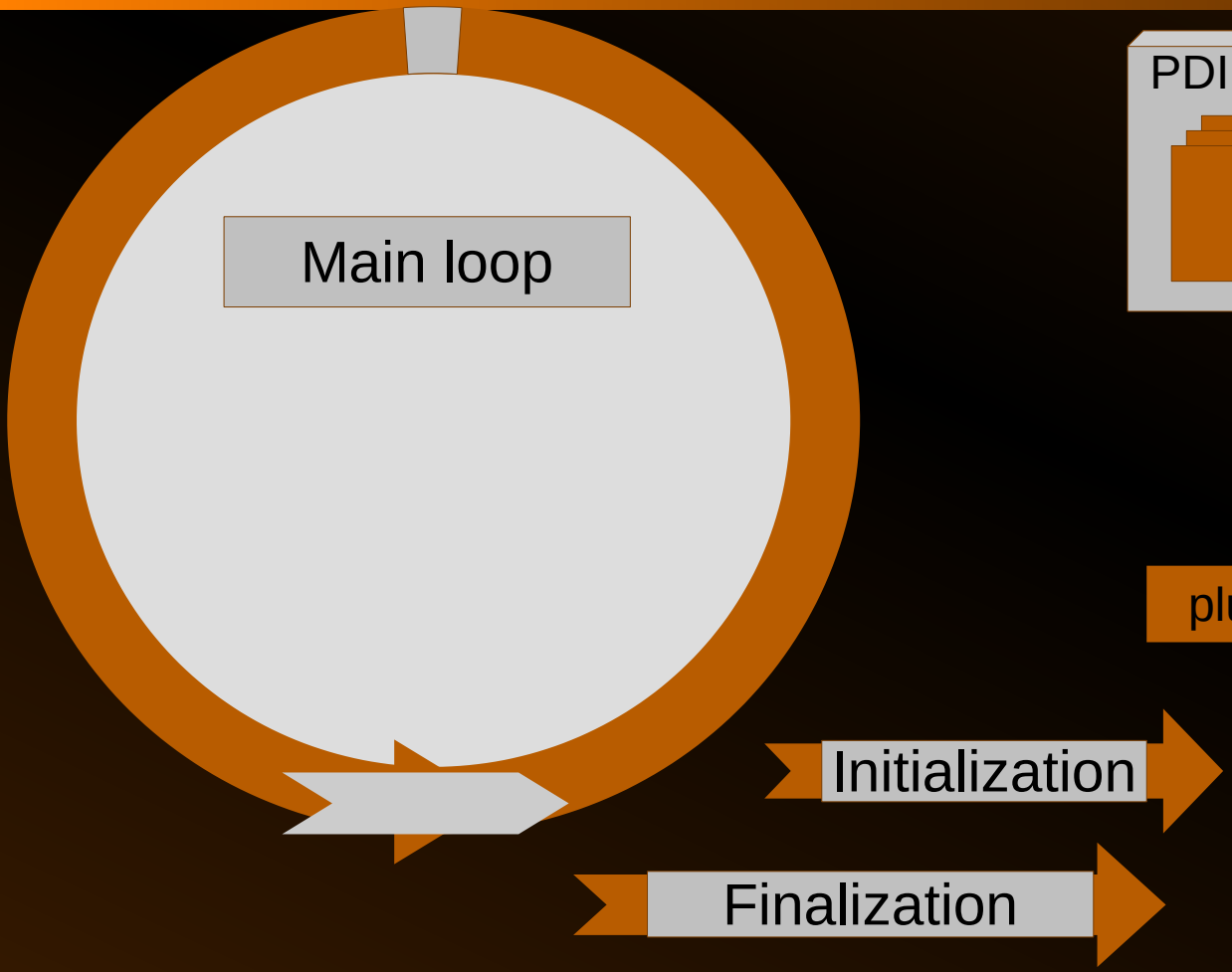
Beyond I/O: data coupling



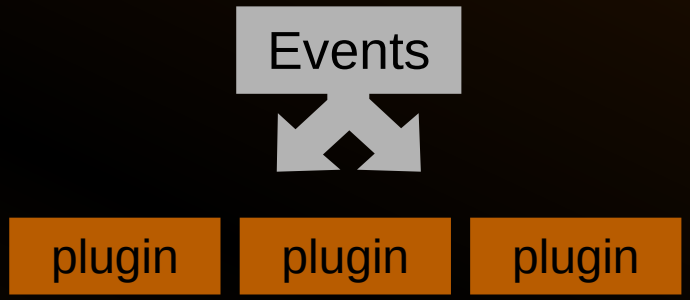
Beyond I/O: data coupling



Beyond I/O: data coupling




```
metadata: { buffer_size: int,
iteration_id: int, rank: int }
data: { main_buffer: { type: array,
subtype: double, size: $buffer_size }
}
plugins:
  decl_hdf5:
    file: 'my_file_${iteration_id}x$
{rank}.h5'
    write: main_buffer
```



Beyond I/O: data coupling