



Handling IO as a data-coupling problem with PDI



Sept 21st, 2022 – PRACE Training



Julien Bigot², Yacine Ould Rouis³,
Karol Sierocinski⁷, Kacper Sinkiewicz⁷

Thanks to:

Leonardo Bautista Gomez¹, Sebastian Friedemann⁶, Virginie Grandgirard², Amal Gueroudji², Karim Hasnaoui³, Francesco Iannone⁴, Kai Keller¹, Guillaume Latu², Bruno Raffin⁶, Benedikt Steinbusch⁵, Christian Witzler⁵

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...



- 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 want

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

➤ We want

➤ We want large language support

➤ We want
Complex but common problem,
A community with dedicated expert

➤ We want a portable file format

➤ We want to

Let's use **libraries**

➤ We want...



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

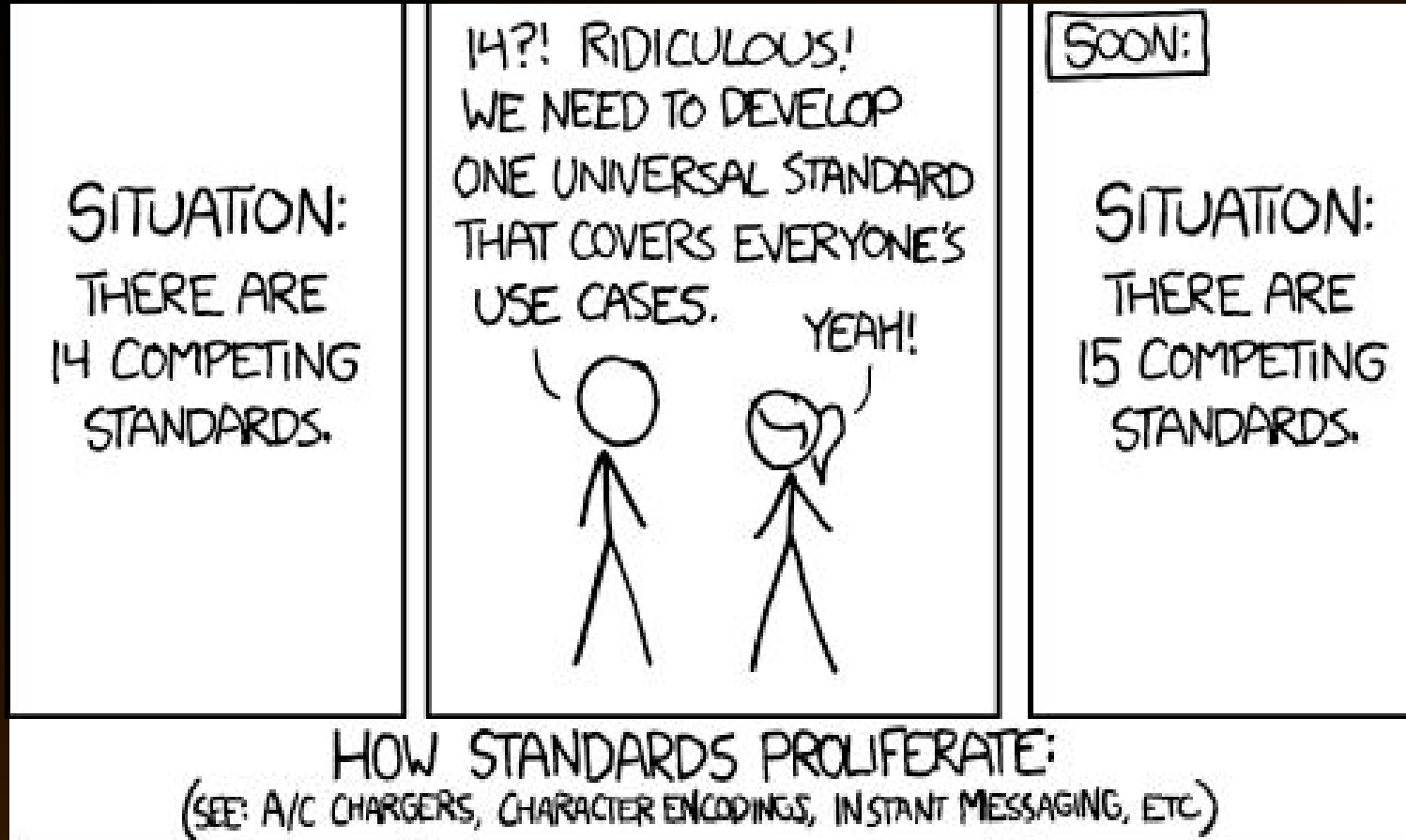
Choosing the best library: a problem on its own

The best library depends on...

- The code specifics, the type of I/O
 - Parallelism level, frequency, ...
 - Initialization data reading, result writing (small or large), checkpoint writing, coupling related I/O
- The specific execution
 - Many codes end-up with an IO abstraction layer
- The specific hardware available
 - I/O bandwidth, intermediate storage, ...



Introducing PDI

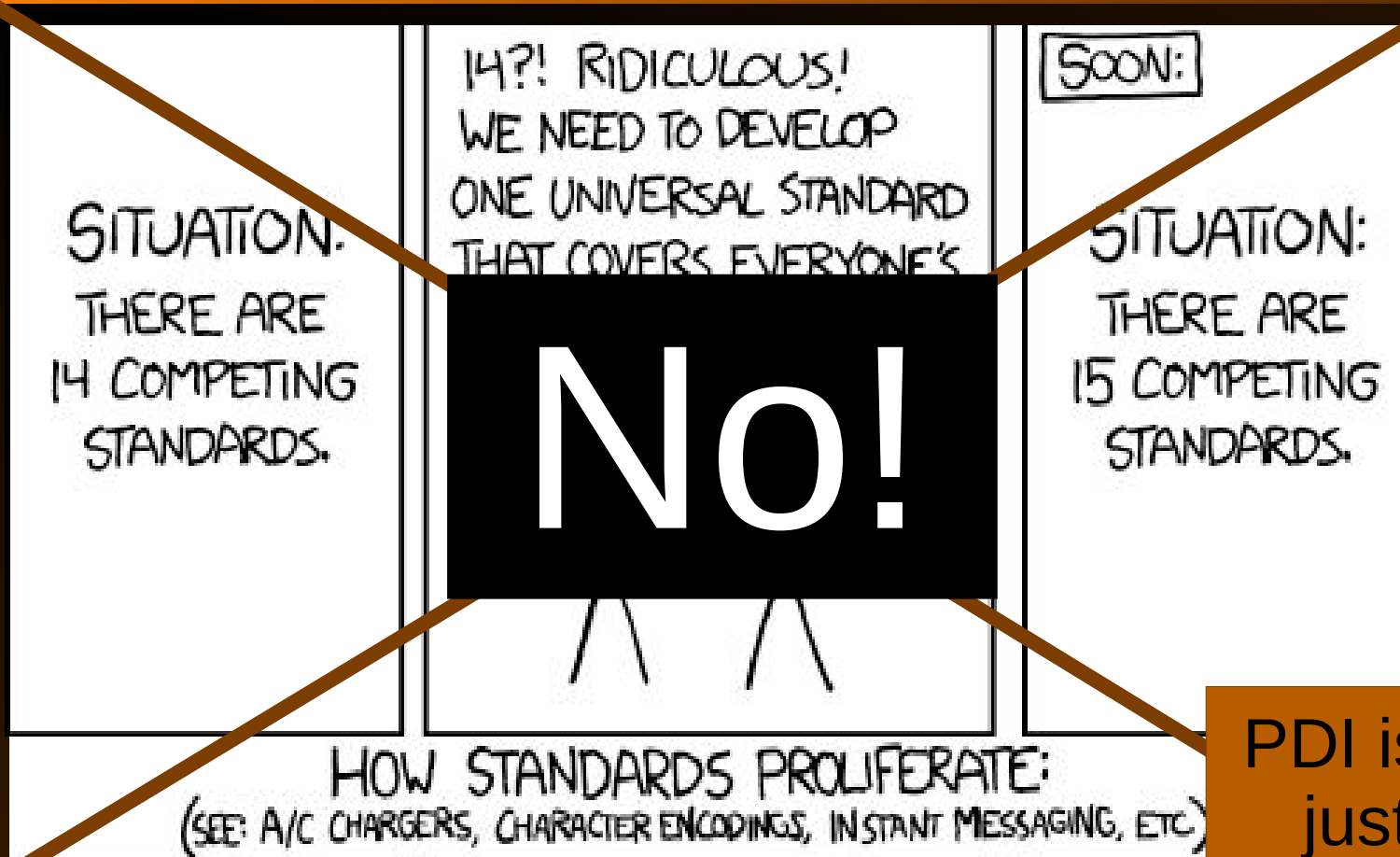


© XKCD

<https://xkcd.com/927/>



Introducing PDI



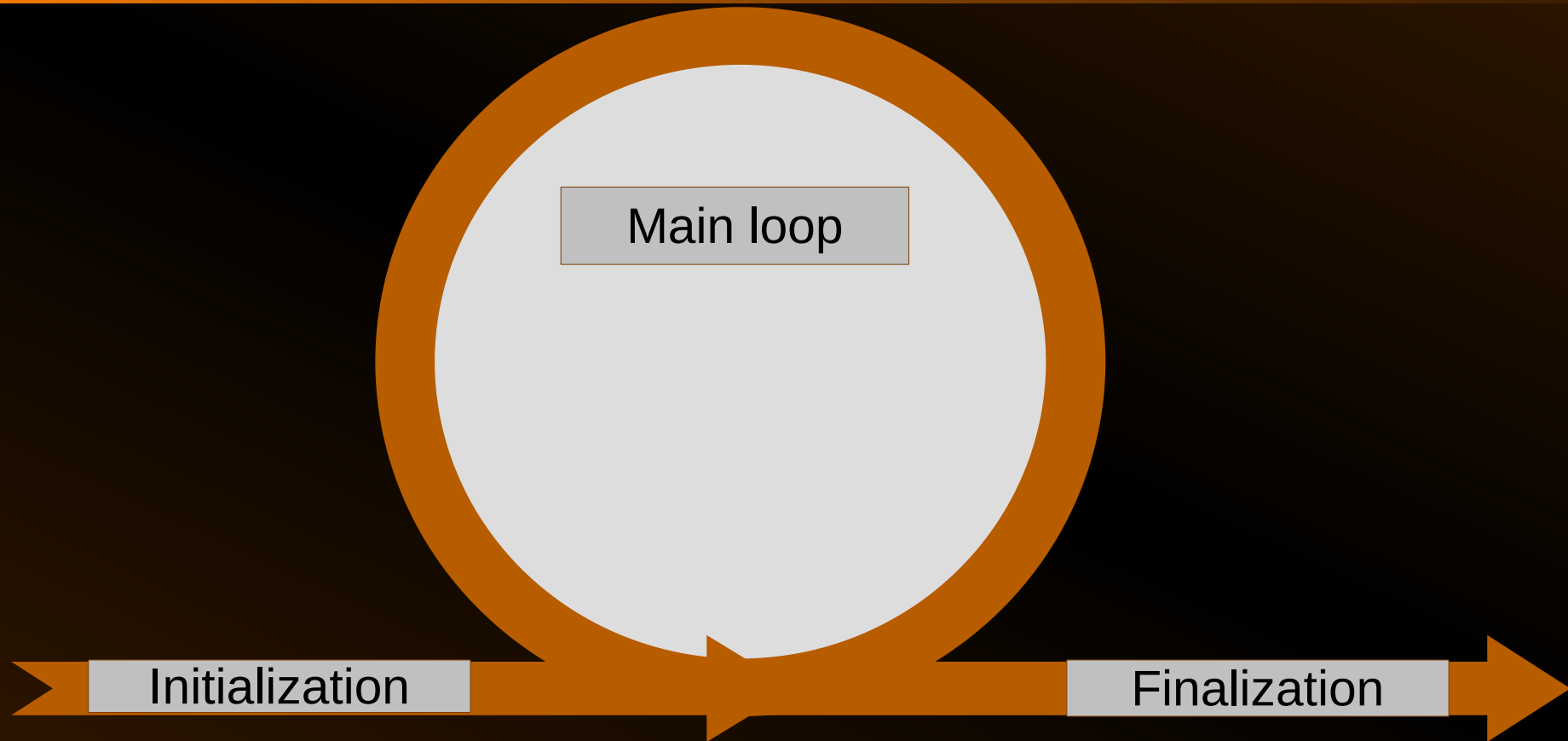
© XKCD

<https://xkcd.com/927/>

PDI is an Interface...
just an interface!

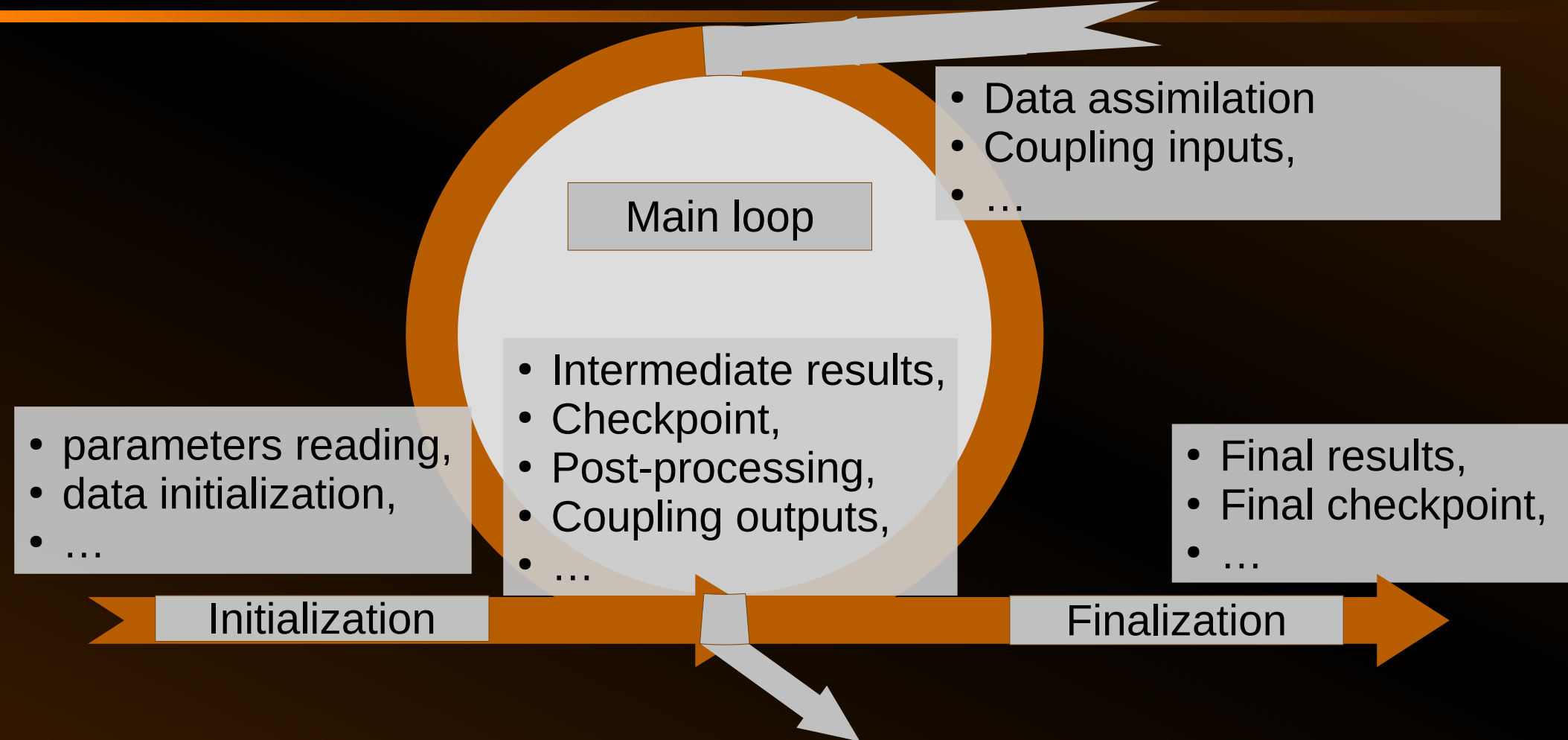


I/O in codes: let's take a step back





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

- parameters reading,
- data initialization,
- ...

Main loop

- Intermediate results,
- Checkpoint,
- Post-processing,
- Coupling outputs,
- ...

- Data assimilation
- Coupling inputs,
- ...

- Final results,
- Final checkpoint,
- ...

Initialization

Finalization



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 read
- data initialization
- ...

Main loop

- Data assimilation
- Coupling inputs,
- ...

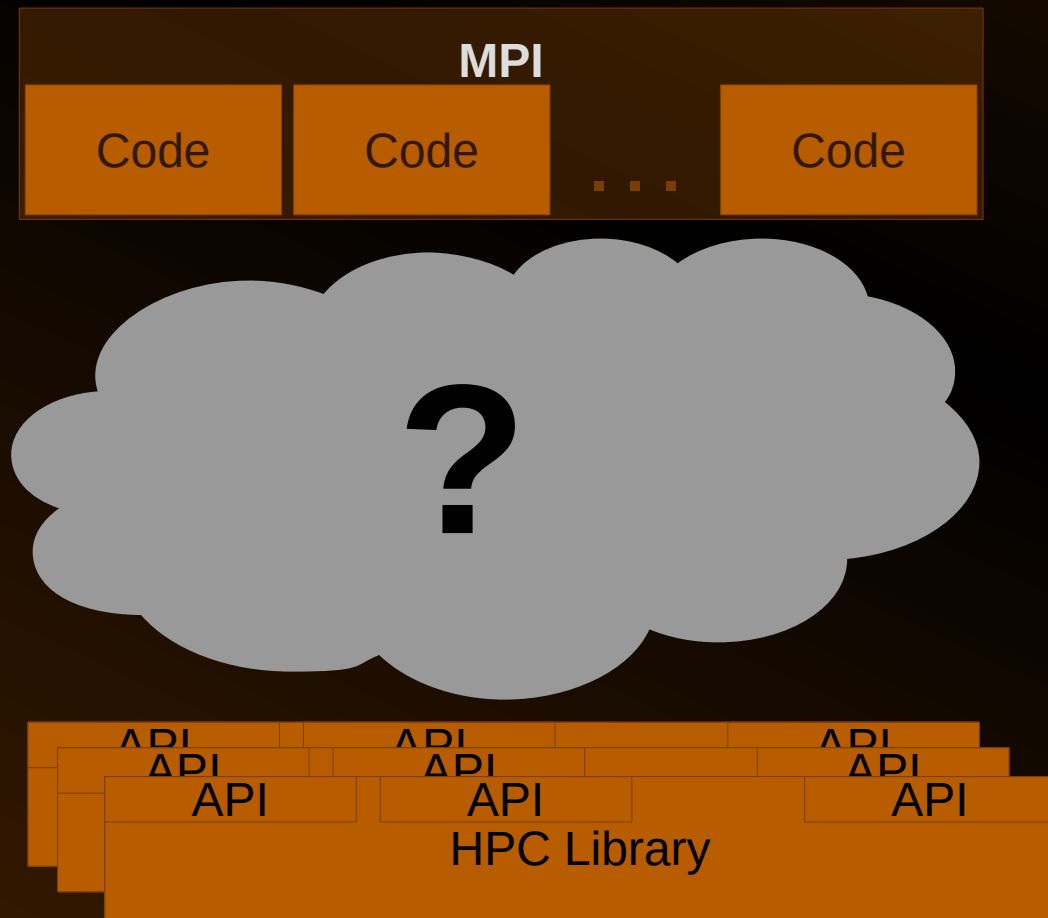
**The
data-coupling
problem**

- Final results,
- Final checkpoint,
- ...

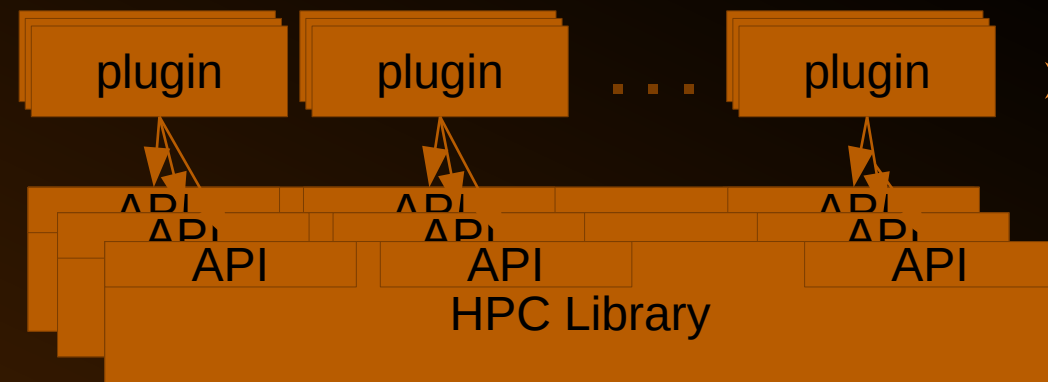
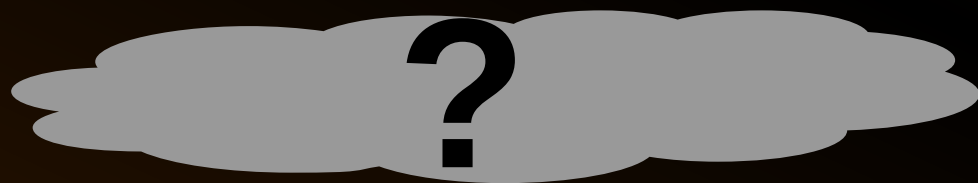
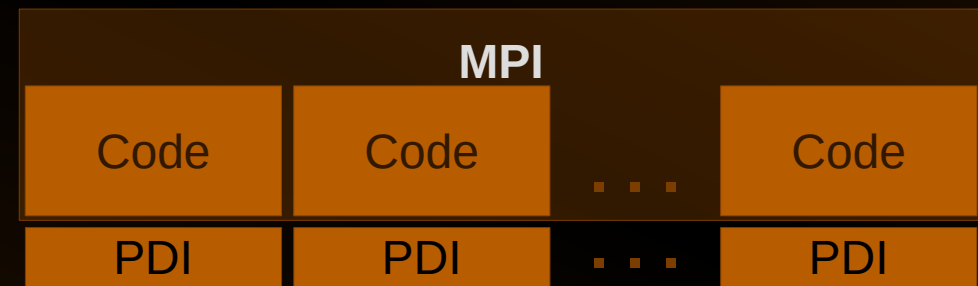
Initialization

Finalization

What is PDI?



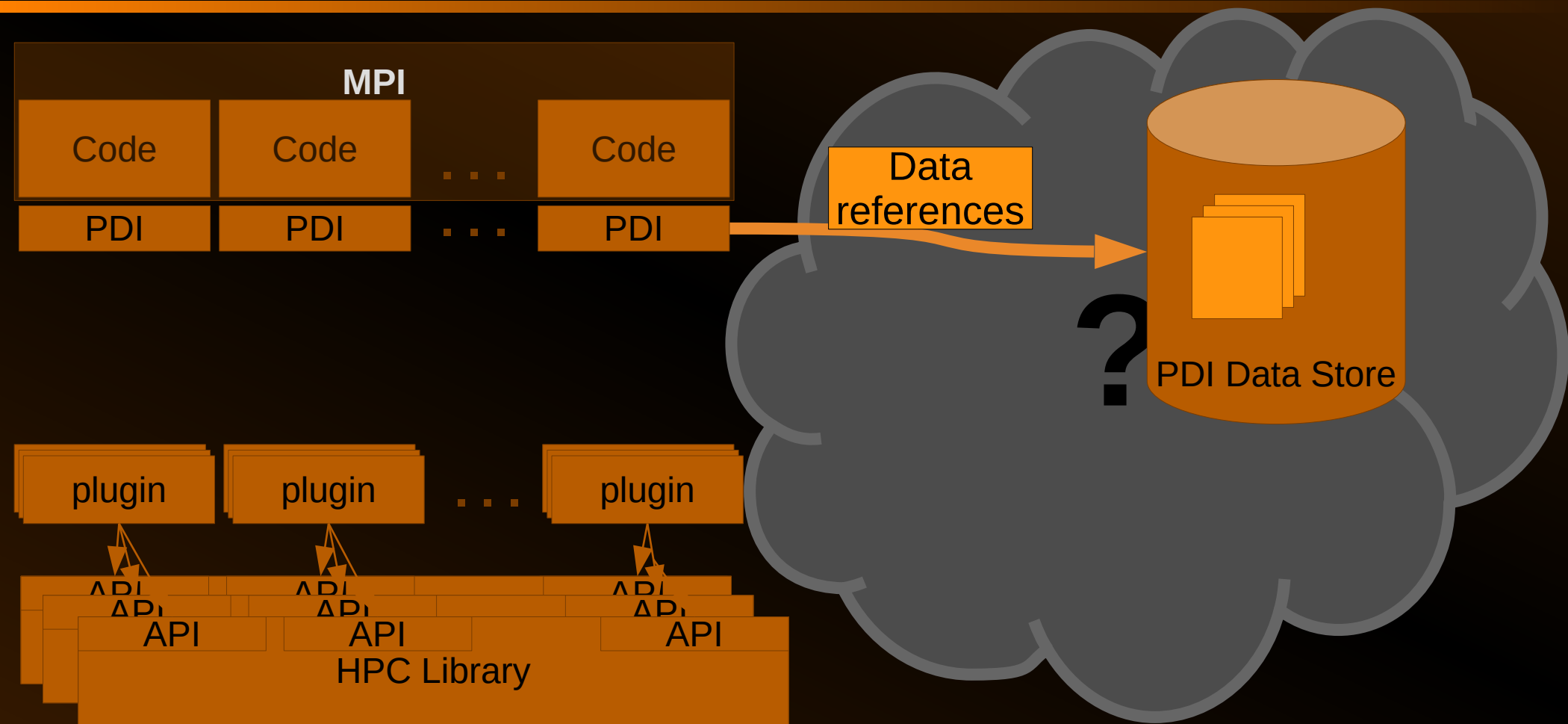
What is PDI?



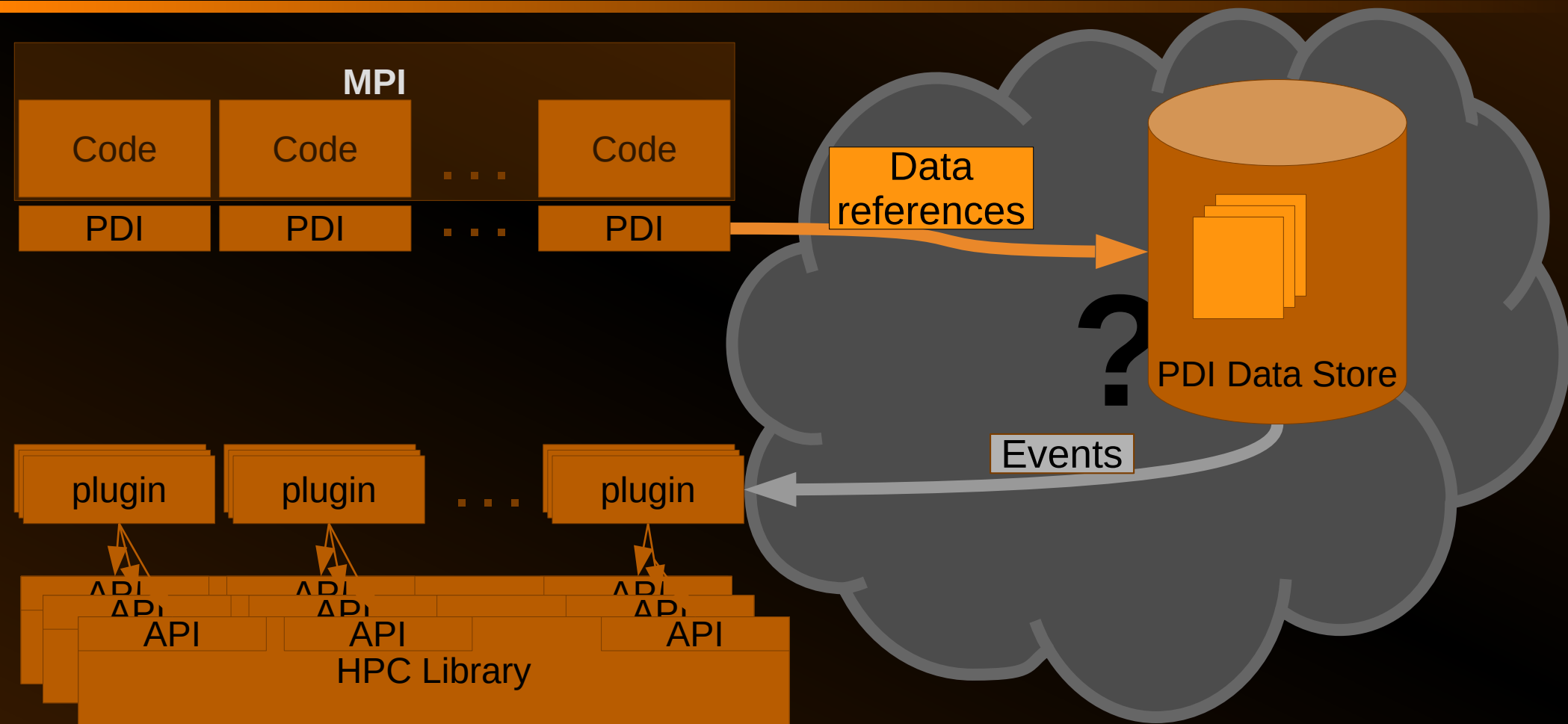
➤ PDI annotations: a purely declarative API

➤ Plugins for access to existing libraries

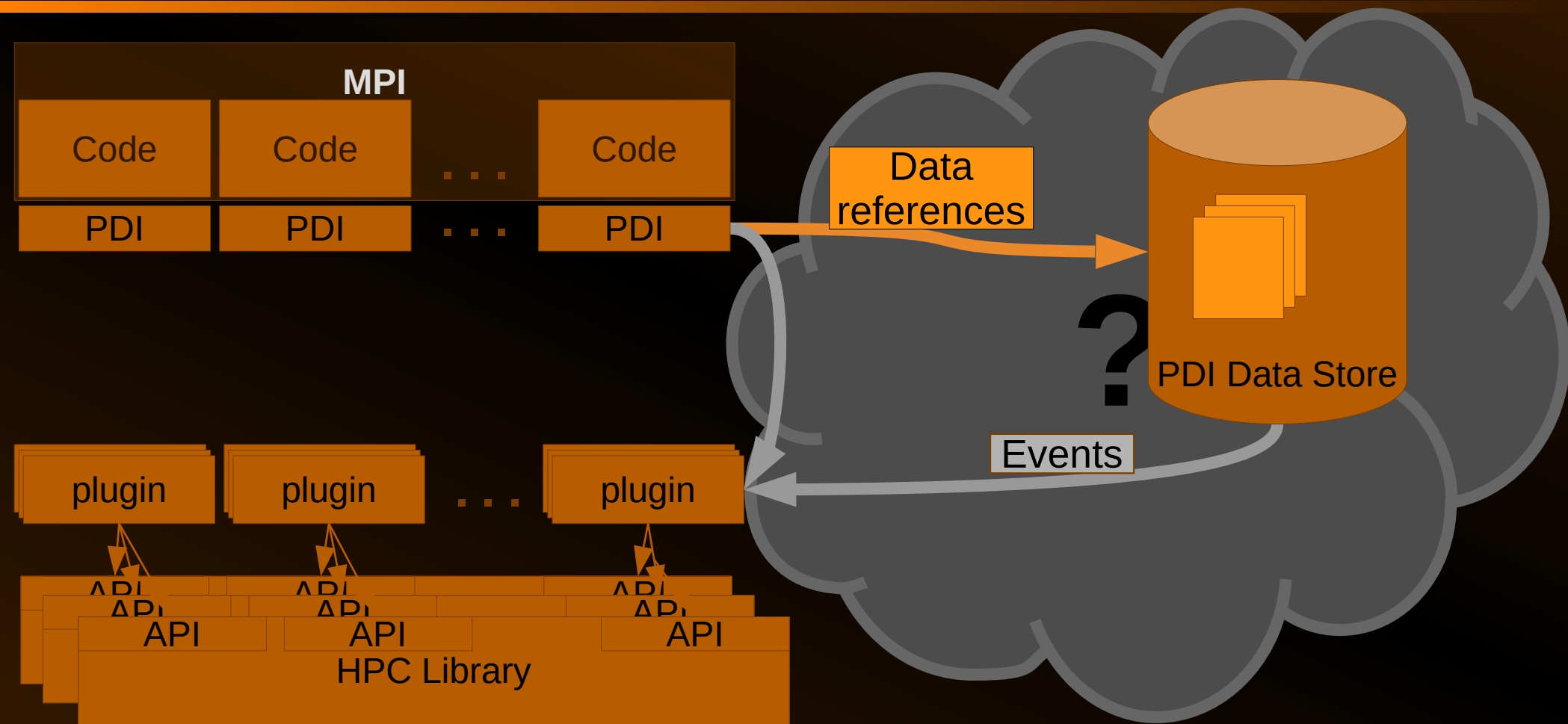
What is PDI?



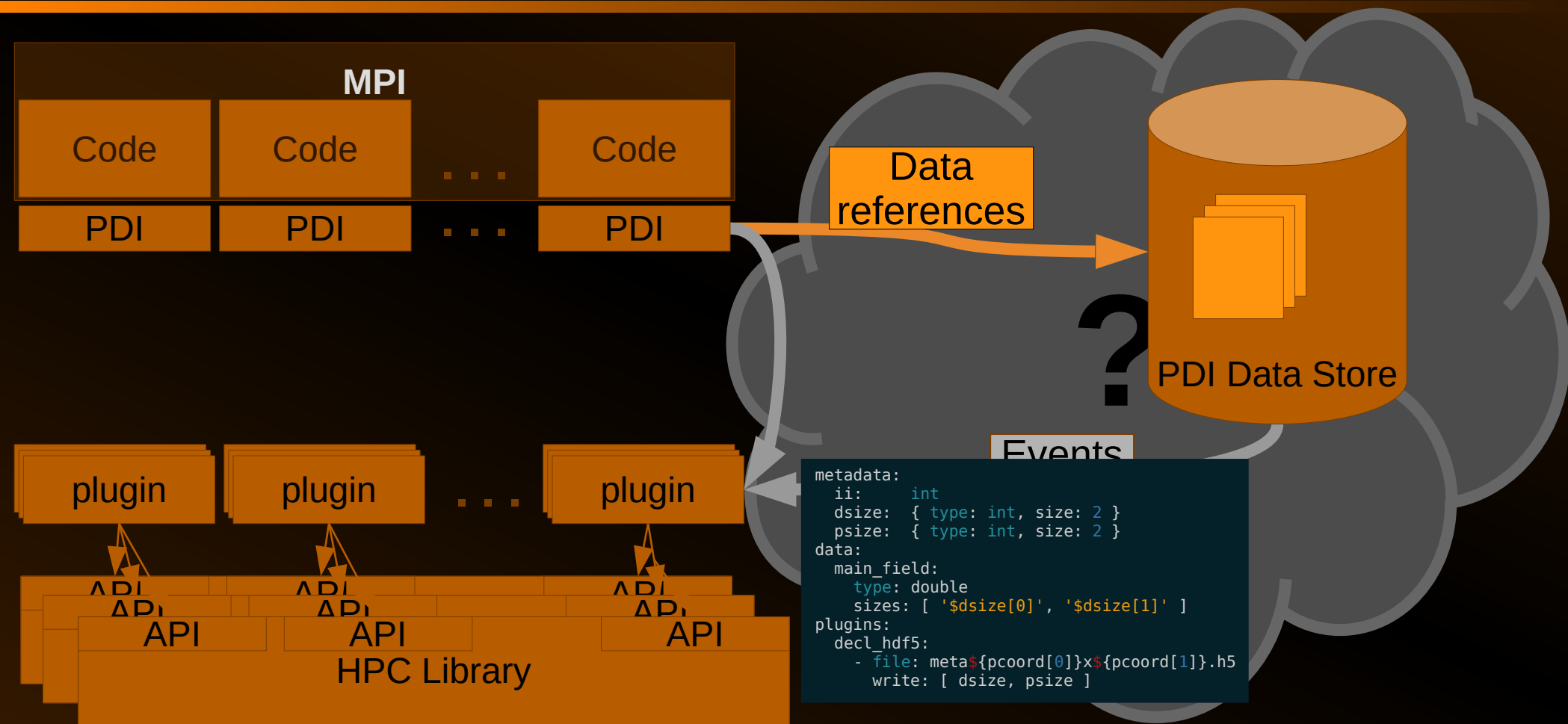
What is PDI?



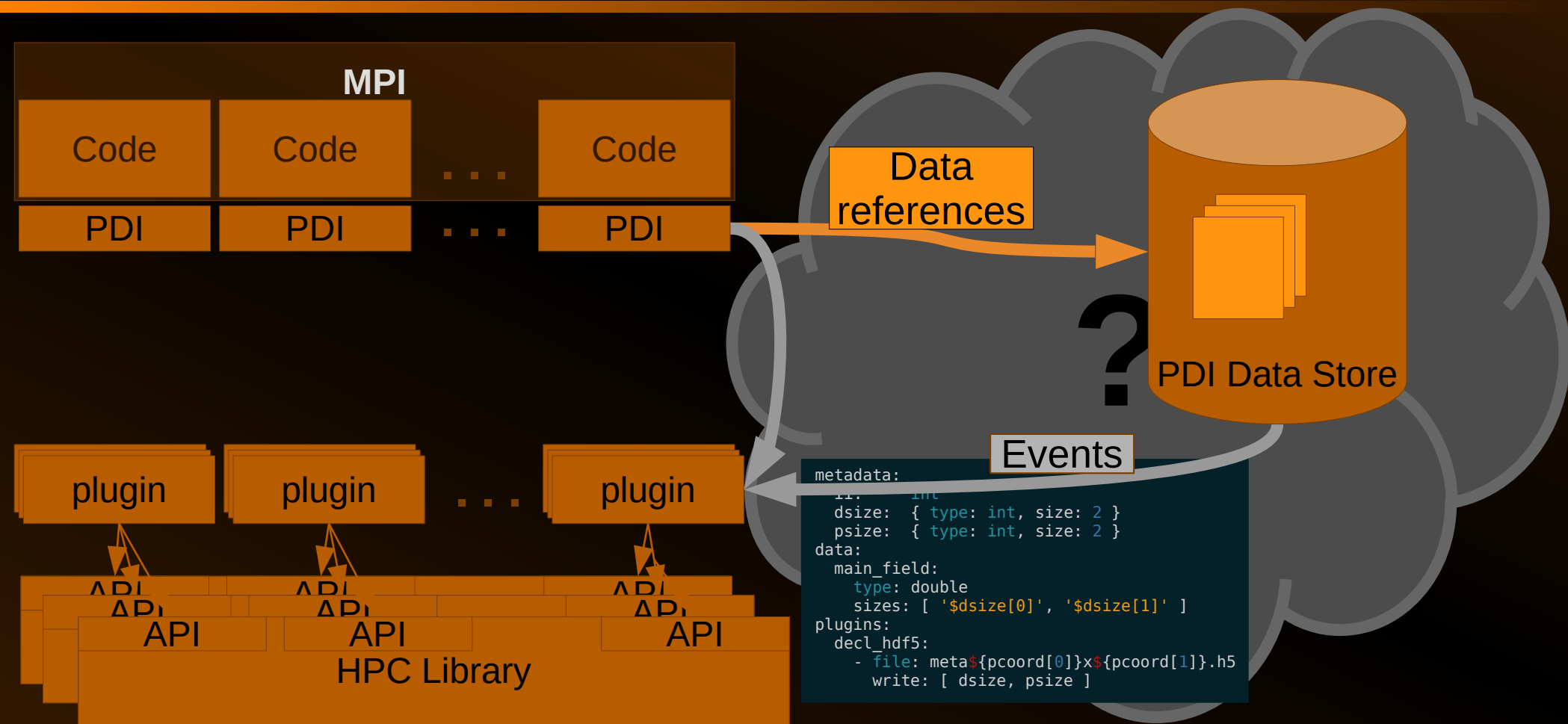
What is PDI?



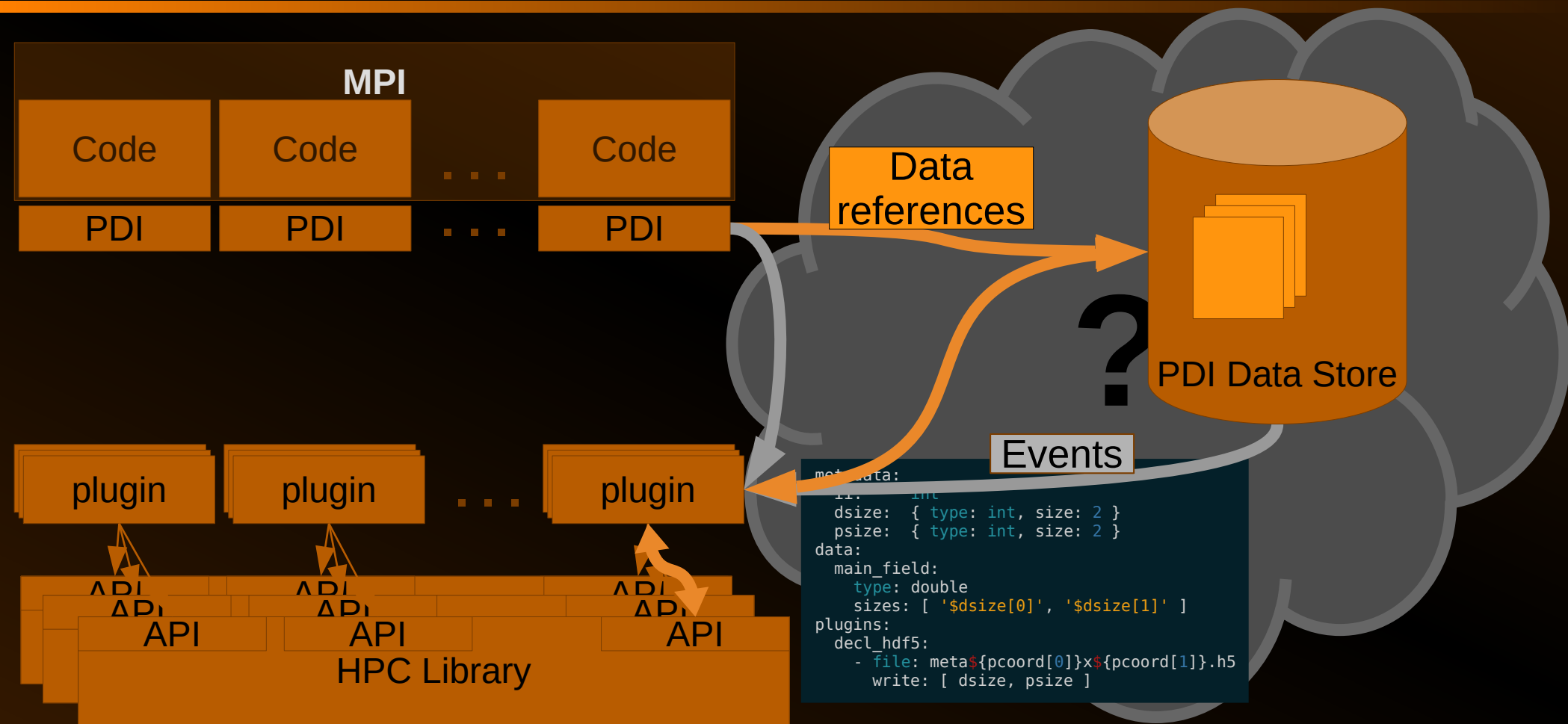
What is PDI?



What is PDI?



What is PDI?





PDI: A simple API

```
/** Initializes PDI */  
PDI_status_t PDI_init(PC_tree_t yaml_conf);  
  
/** Finalizes PDI */  
PDI_status_t PDI_finalize();
```

a C / C++ API
Also available for:

- Fortran
- Python

- **Init** takes the specification tree as parameter
 - The YAML is parsed using the *paraconf* library
- **Finalize** releases all PDI-related resources



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);
```

a C / C++ API
Also available for:

- Fortran
- Python

➤ Share

- A buffer is in a coherent consistent state
- Reference the buffer in PDI store

➤ Reclaim

- The buffer will be reused for a different use
- Un-reference the buffer in PDI 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 Additional API

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

// Combine share & expose for 1 piece of data
PDI_status_t PDI_expose(const char *name, void *data, PDI_inout_t access);

// When there is no data... (an interesting location has been reached)
PDI_status_t PDI_event(const char* event);

// And when there is multiple data
PDI_status_t PDI_multi_expose(const char *event_name,
    void *data, PDI_inout_t access,
    ...,
    NULL );
```

a C / C++ API
Also available for:

- Fortran
- Python

- Expose = share + reclaim
- Events: similar to exposing empty data

- Multi-expose:
 - All share
 - An event
 - All reclaims



Inside PDI: The shared data store

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





PDI data model

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

- Data type is specified in YAML
 - Required for static languages (C/C++/Fortran) for now
 - Optional for dynamic languages (Python)
 - MPI / HDF5 inspired model: scalar / array / record
- “Data” vs. “Metadata”
 - PDI only handles the pointer for “data”
 - Minimal overhead
 - PDI keeps a copy of “metadata”
 - Can be used in \$-expressions



Inside PDI: the store + notifications

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





Inside PDI: the store + notifications

➤ PDI data store: a map of buffer references

- Name \Rightarrow unique identifier
- Reference
 - Ownership & locking information
 - RW-lock: Single Writer / Multiple Readers
 - Memory ownership : Strong or Semi-weak
 - Type \Rightarrow memory layout and interpretation
 - Buffer address \Rightarrow pointer to user memory



➤ Notification system: register to be called

- On data share / access
- On arbitrary locations in code (named “events”)





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 in practice: Decl'HDF5

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

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

```
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



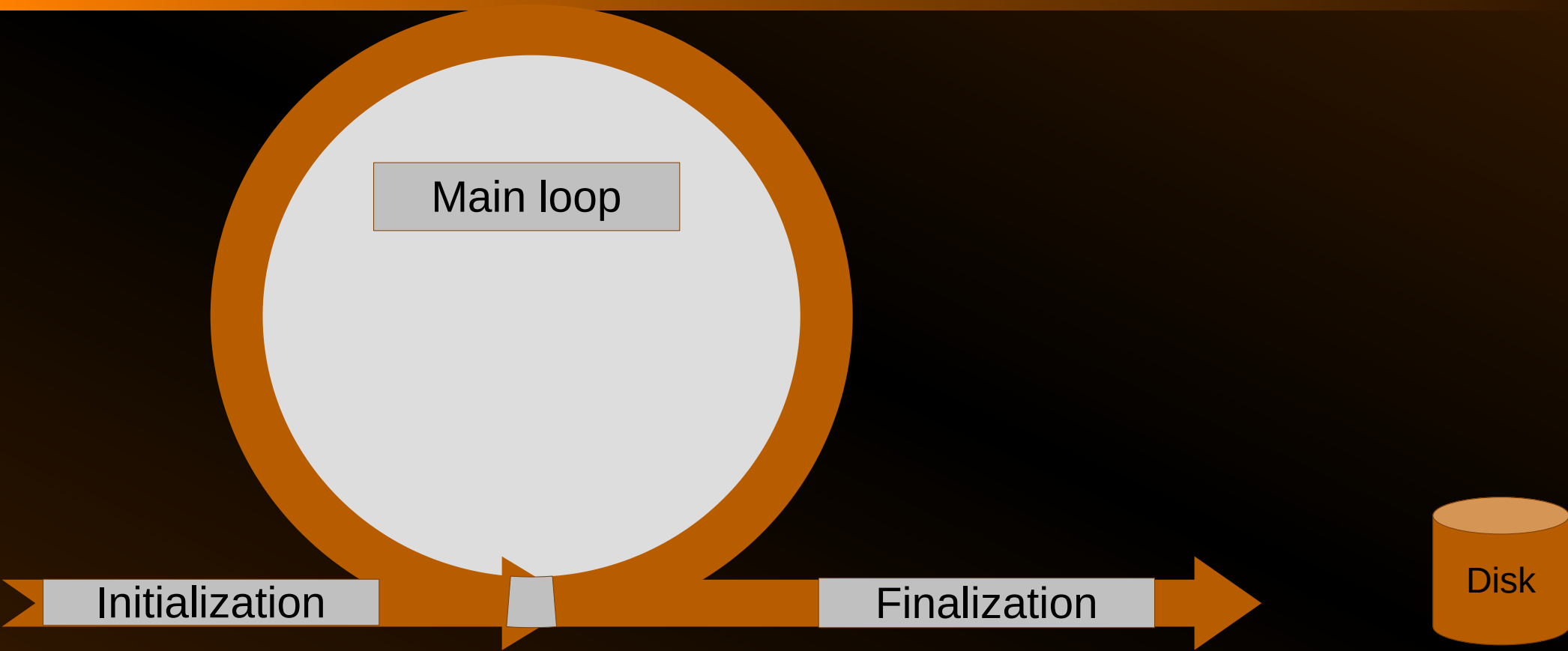
Decl'HDF5: a complex YAML example

```
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

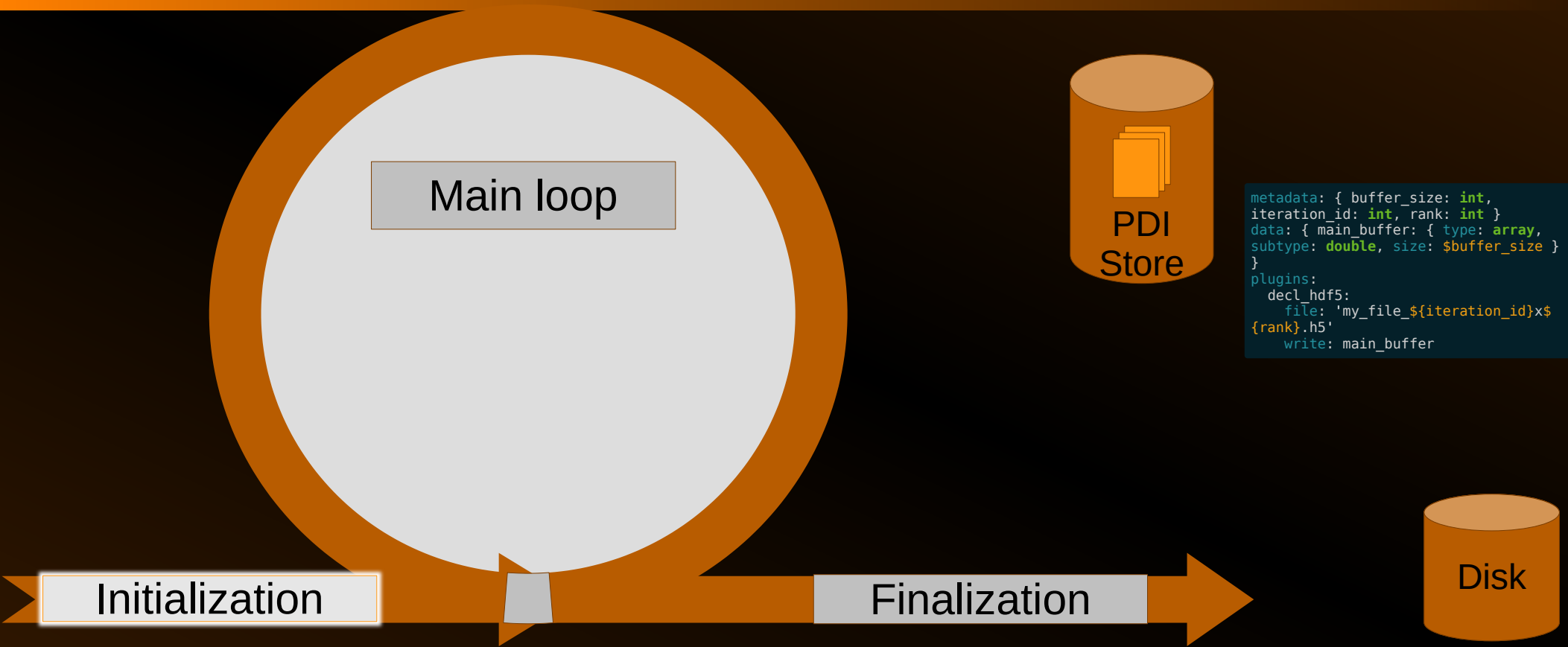


PDI: behind the scene



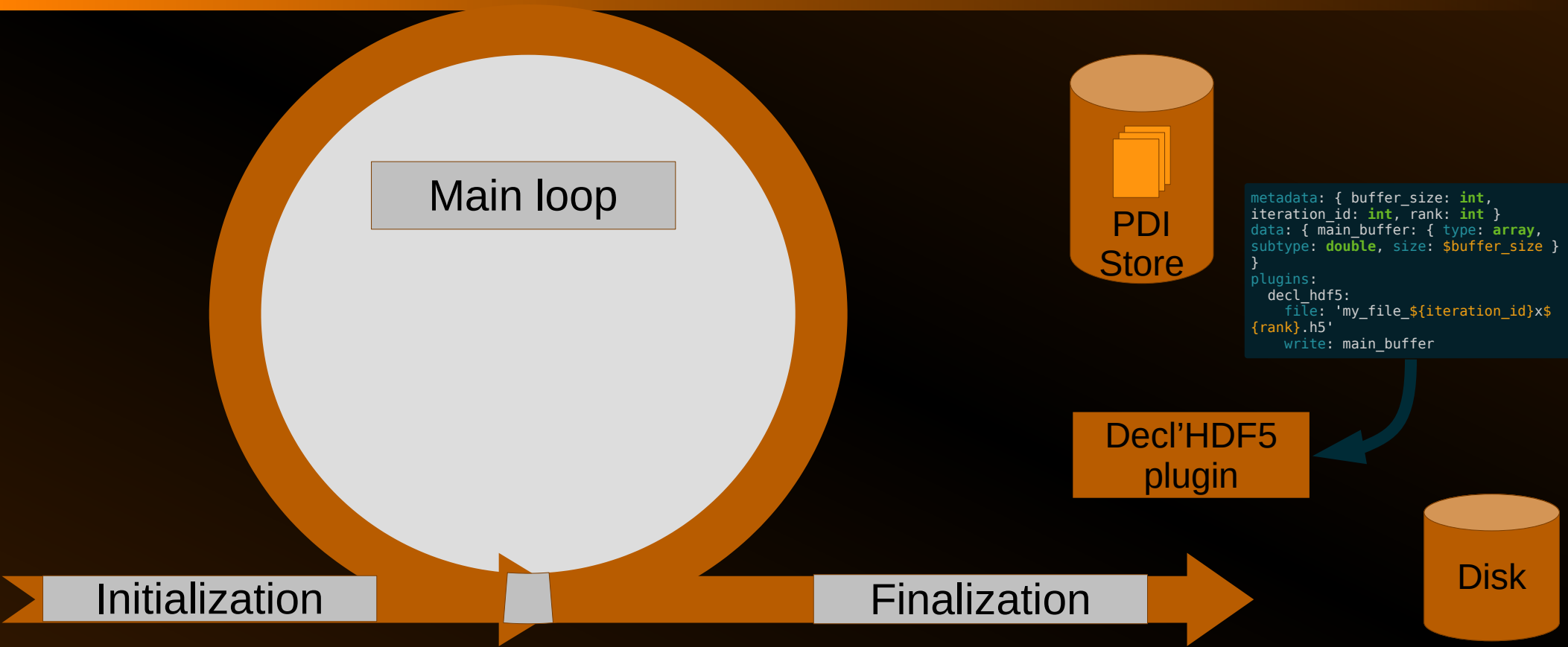


PDI: behind the scene



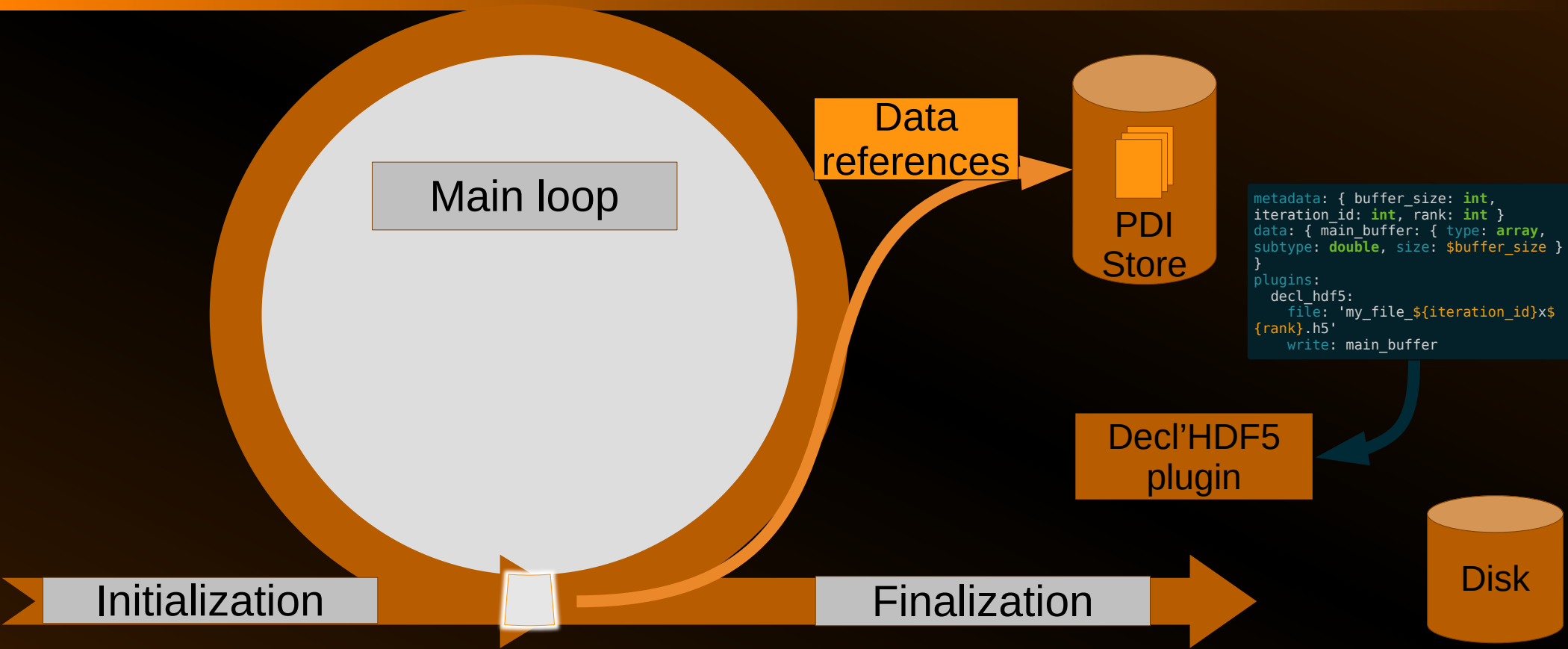


PDI: behind the scene



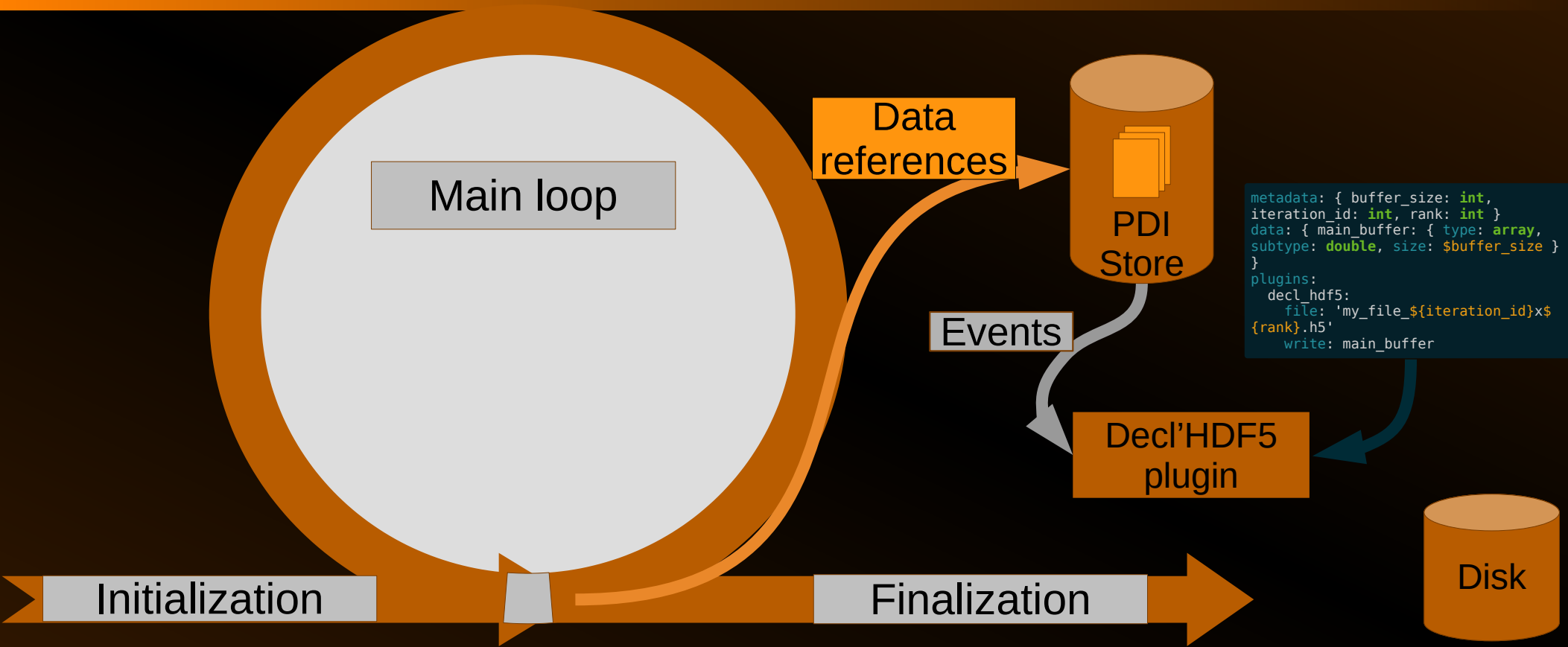


PDI: behind the scene



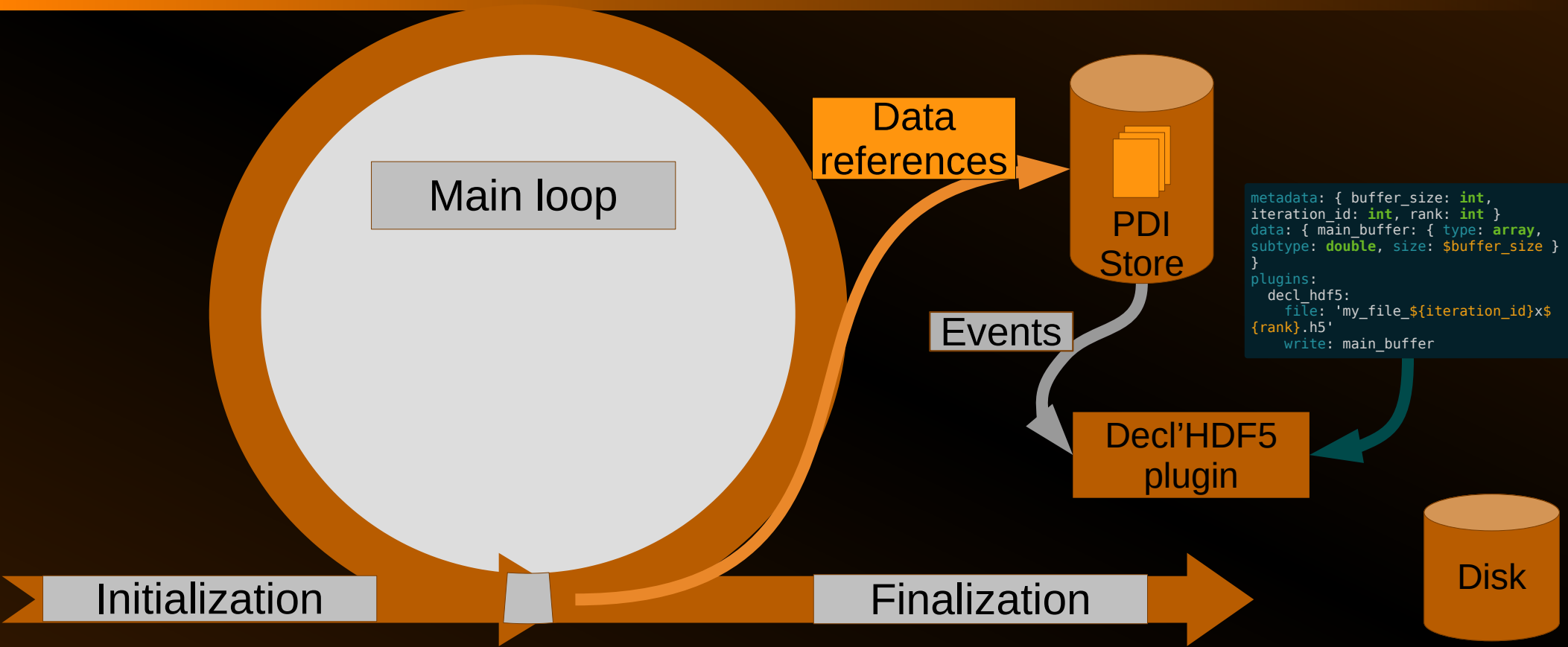


PDI: behind the scene



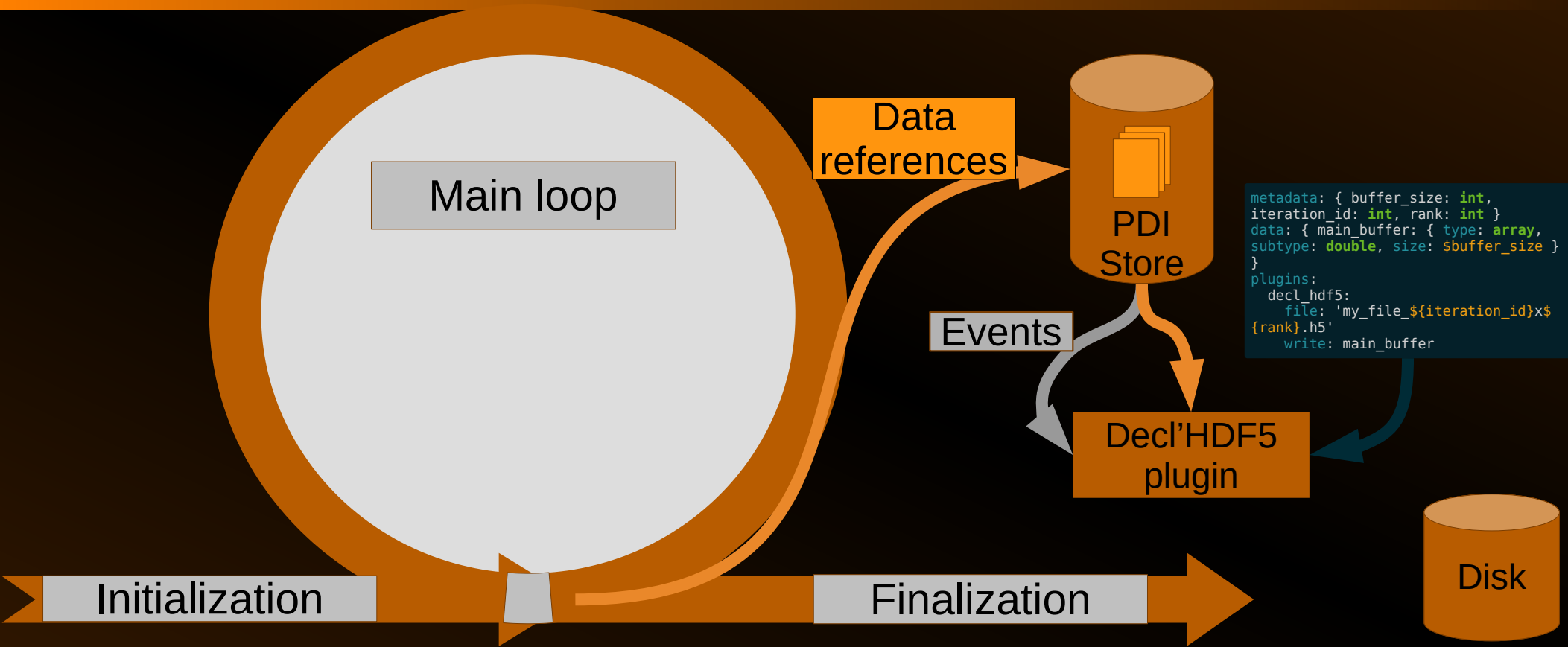


PDI: behind the scene



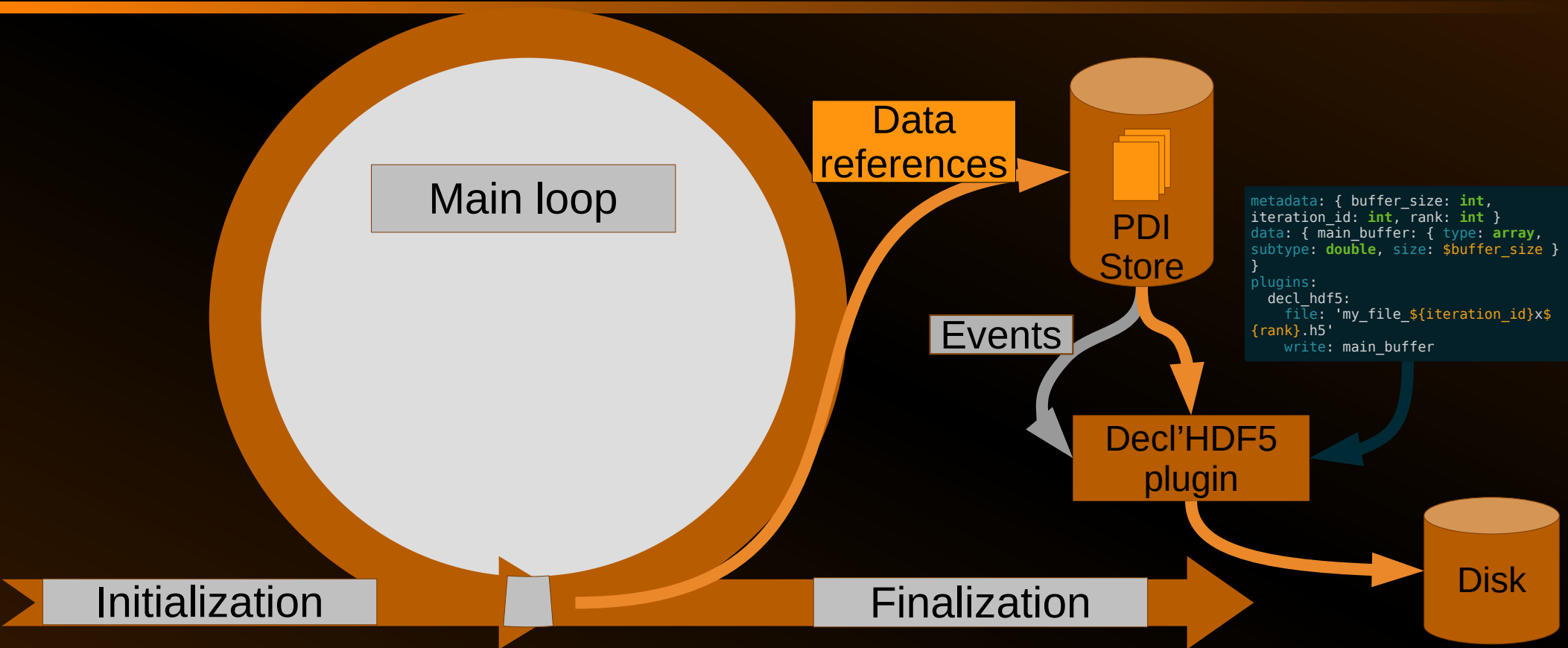


PDI: behind the scene





PDI: behind the scene

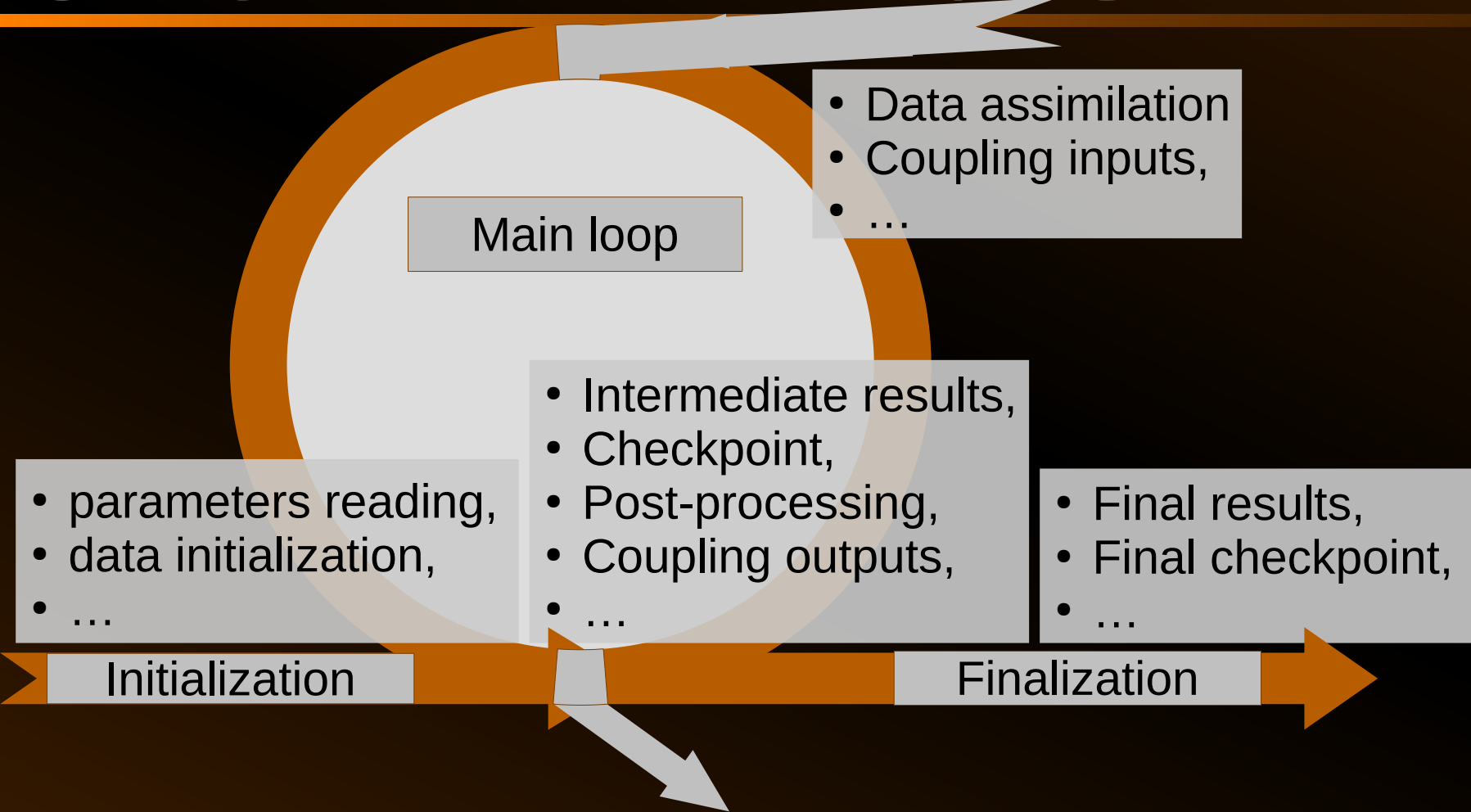




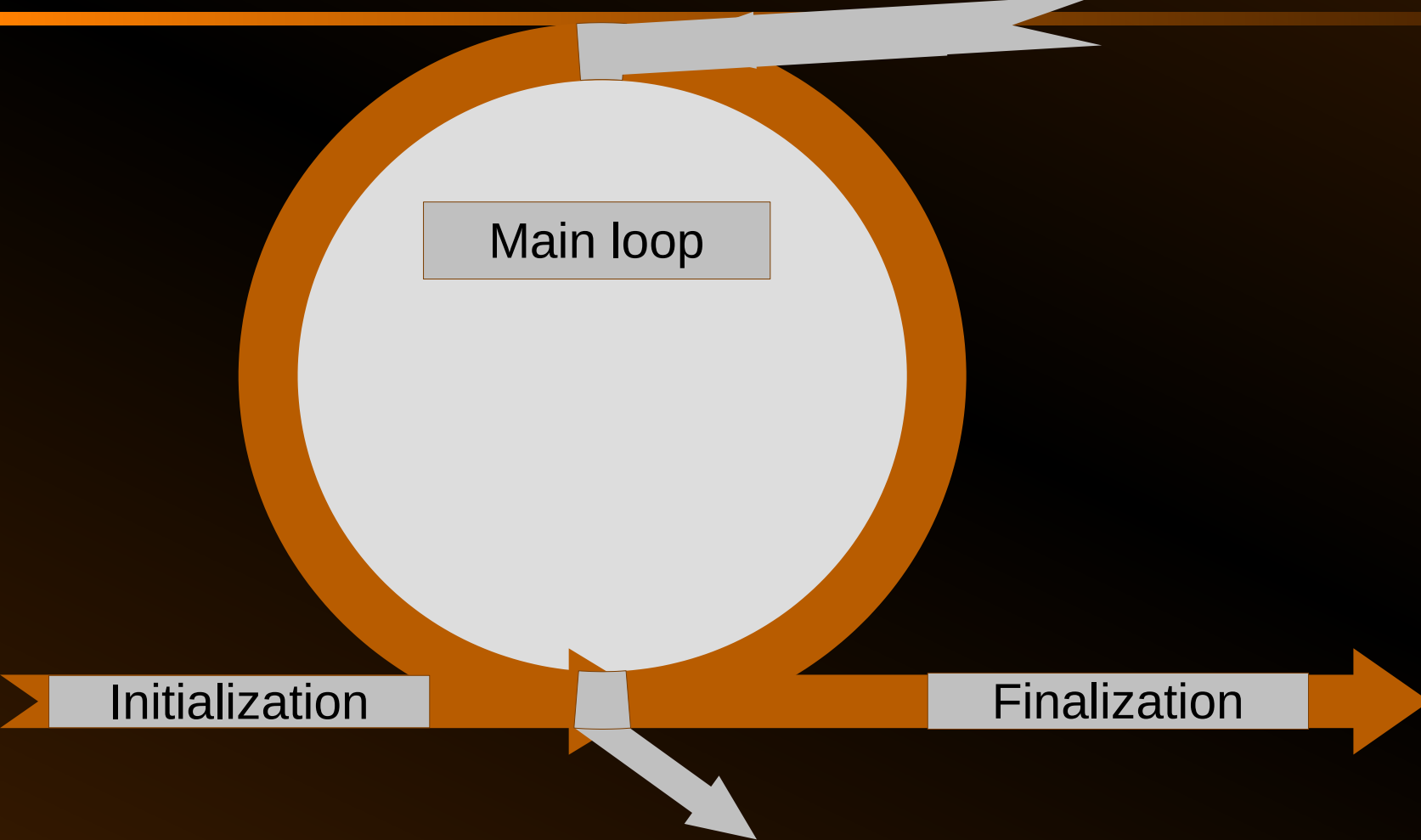
PDI: The plugins

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

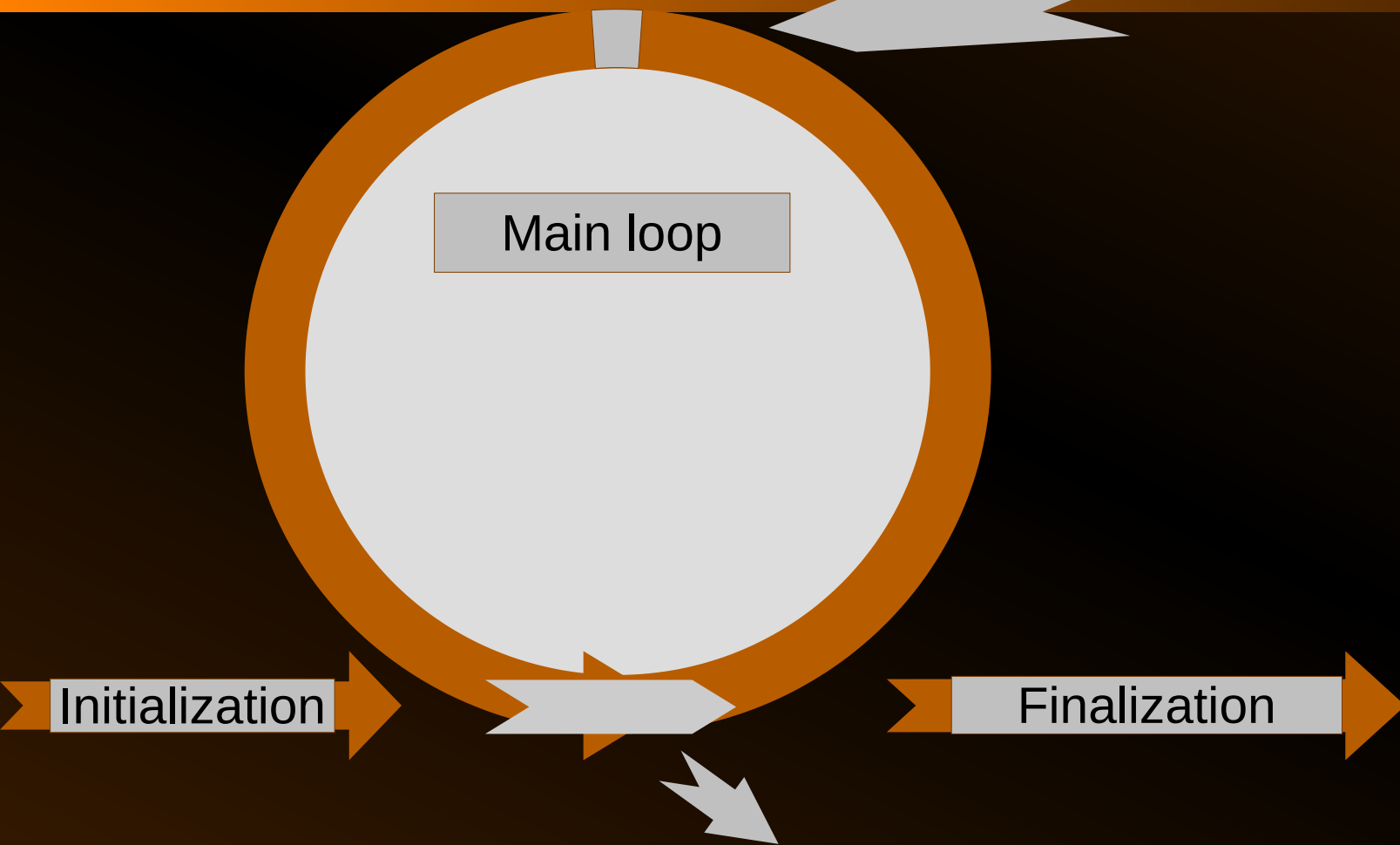
Beyond I/O: data coupling



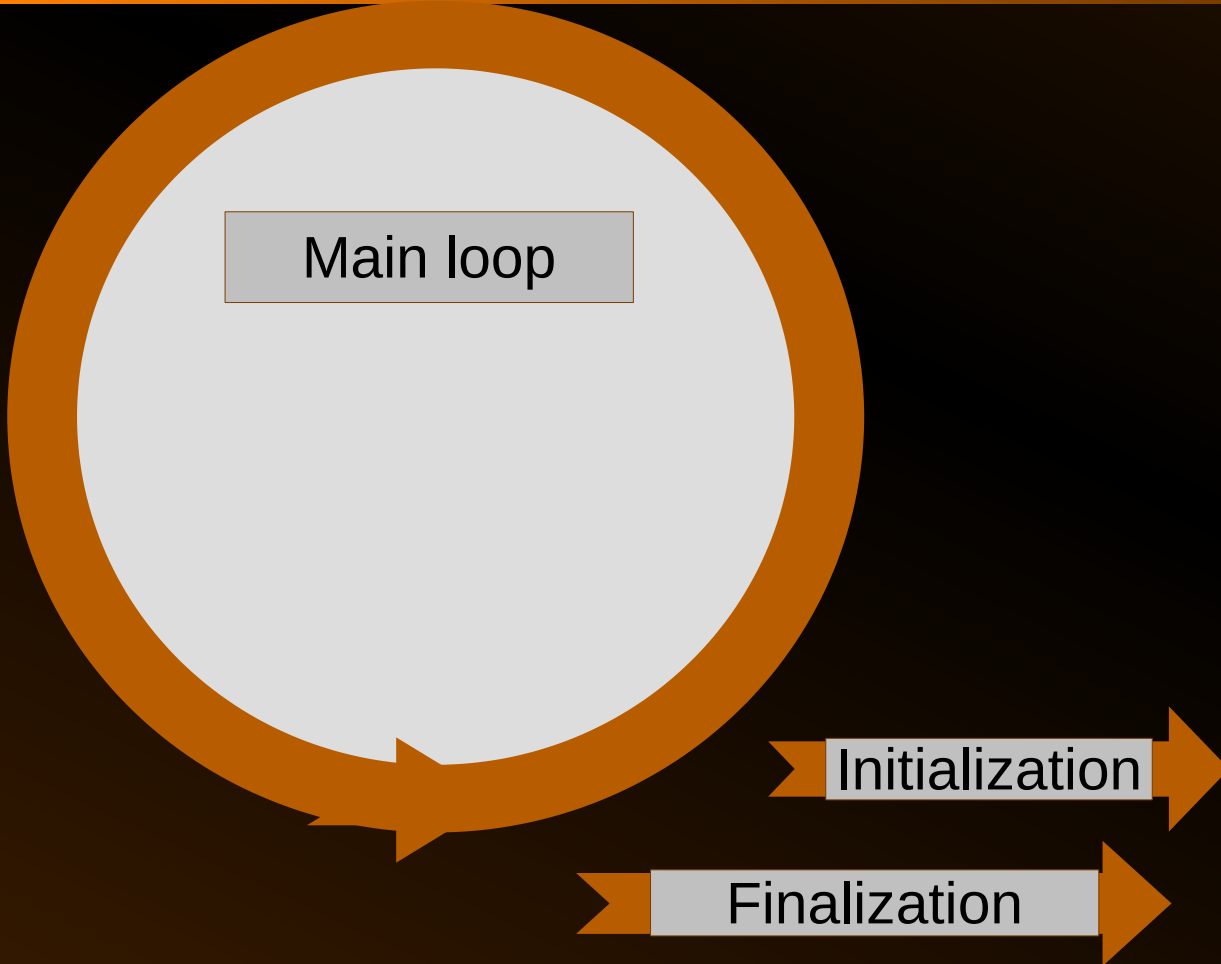
Beyond I/O: data coupling



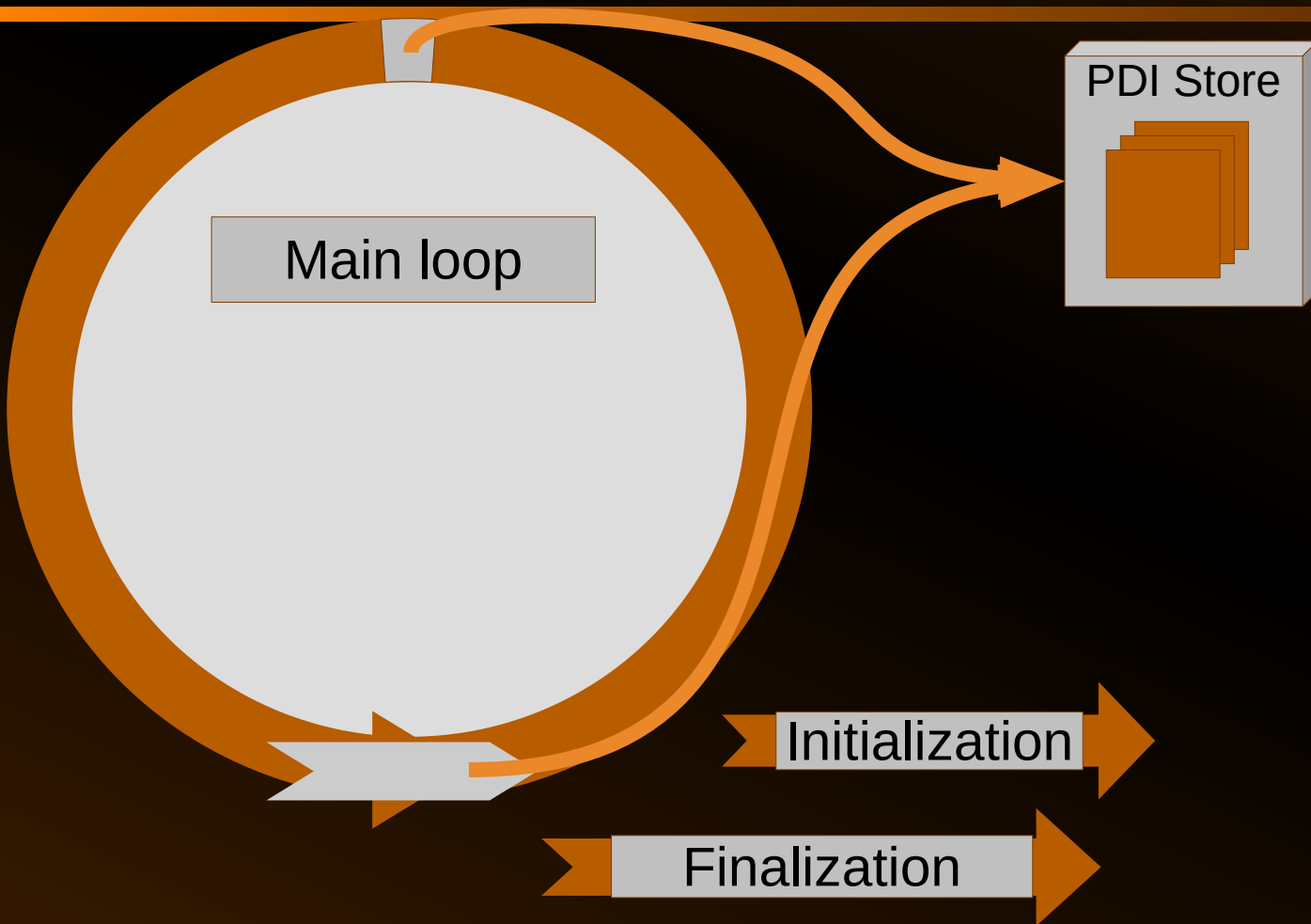
Beyond I/O: data coupling



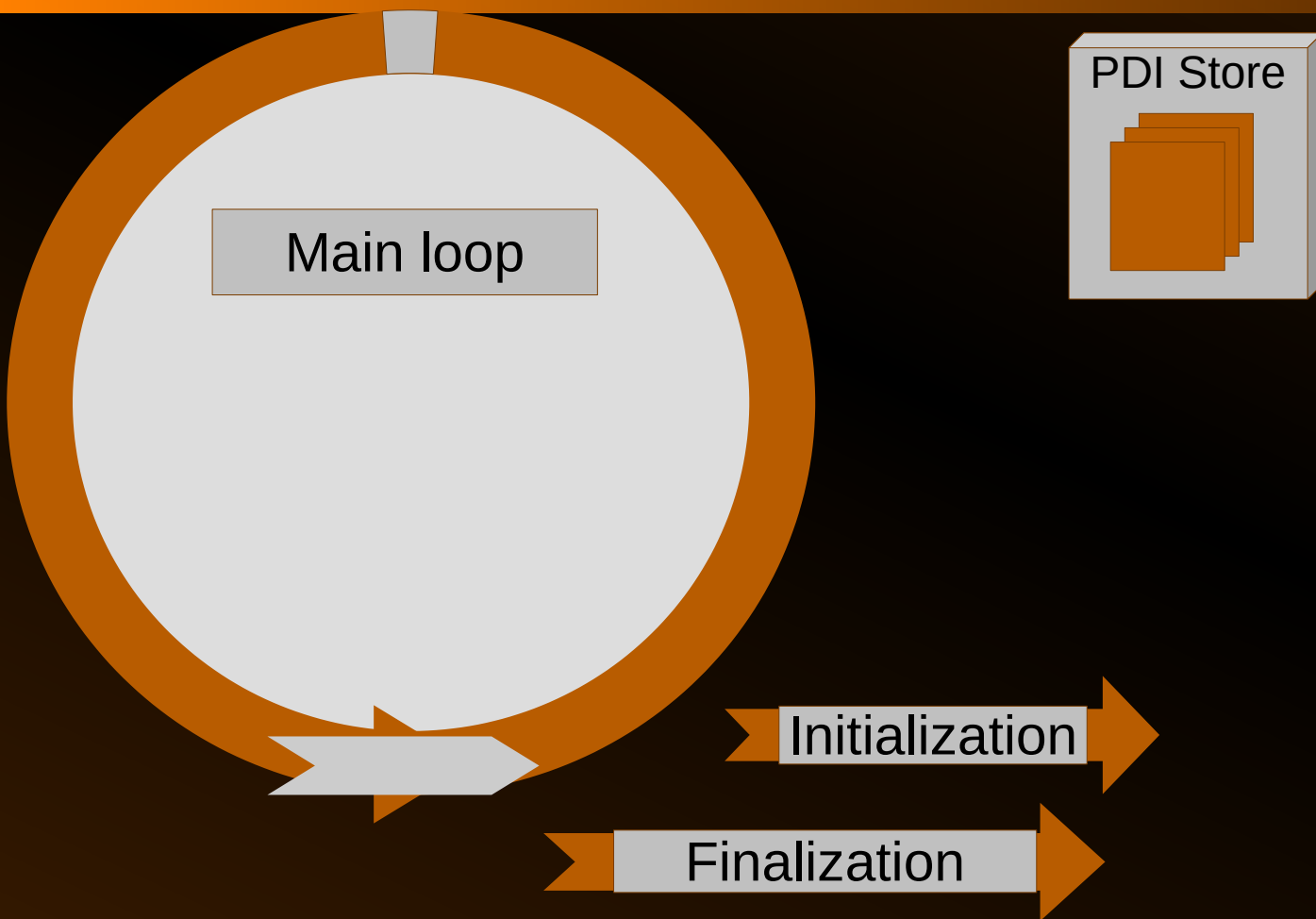
Beyond I/O: data coupling



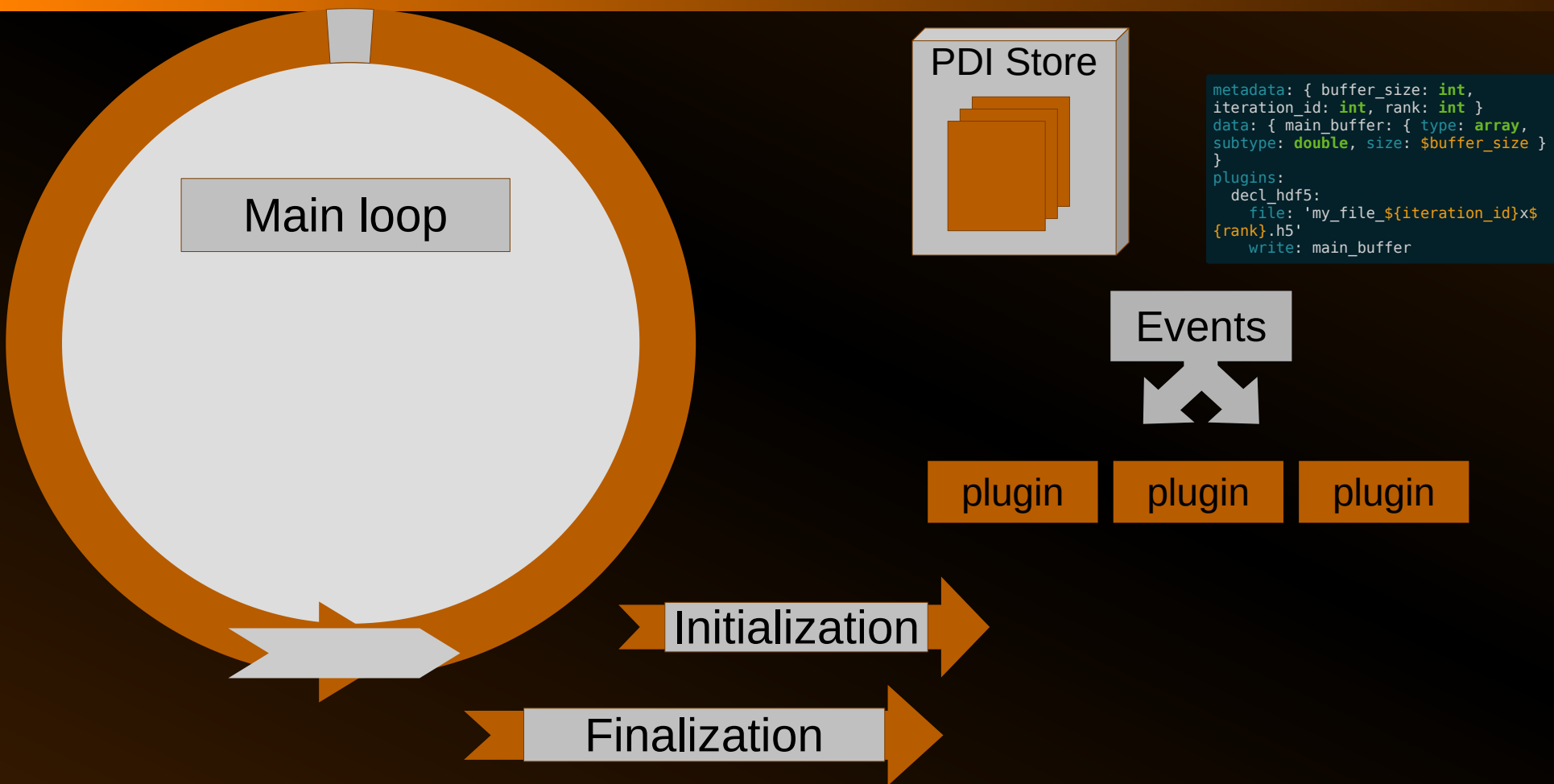
Beyond I/O: data coupling



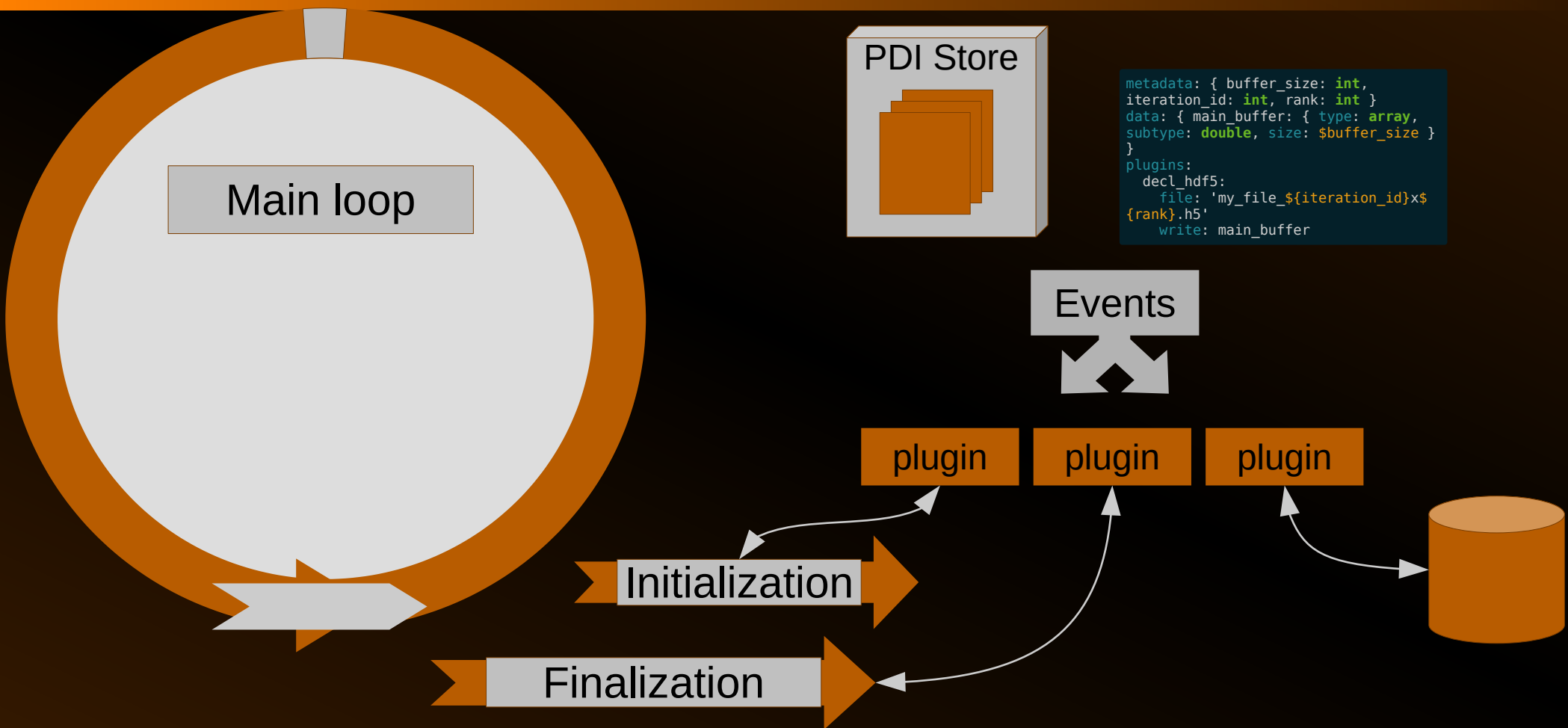
Beyond I/O: data coupling



Beyond I/O: data coupling



Beyond I/O: data coupling





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

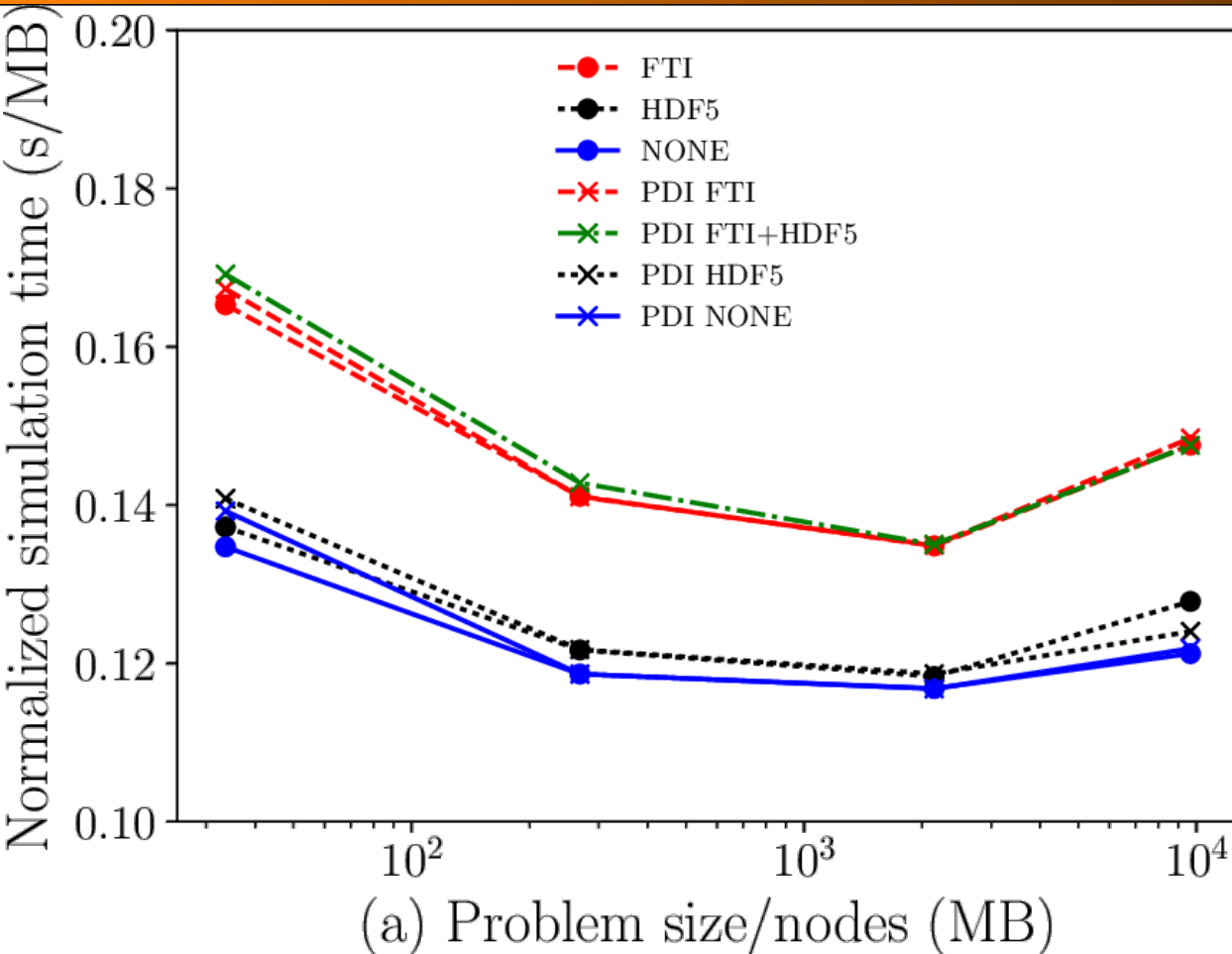
```
plugins:
  user_code:
    on_event:
      trigger_event_name: # event that triggers the call
      function_name { in1: $iteration_id, in2: $main_field }
```

```
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

PDI: Perf. Evaluation 1/2

Corentin Roussel (MdlS)
Kai Keller (BSC)



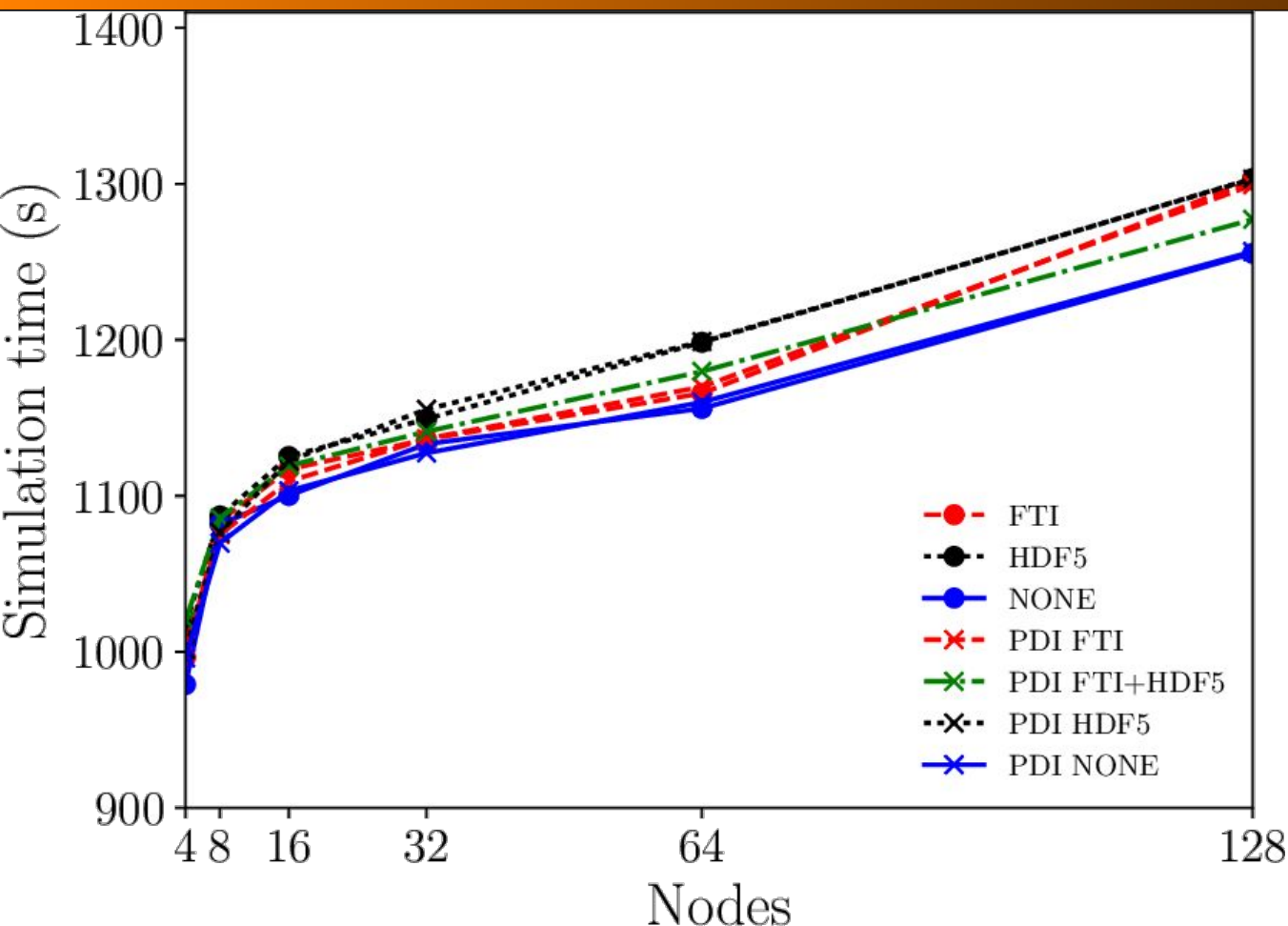
➤ 4 versions of Gysela

- 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)



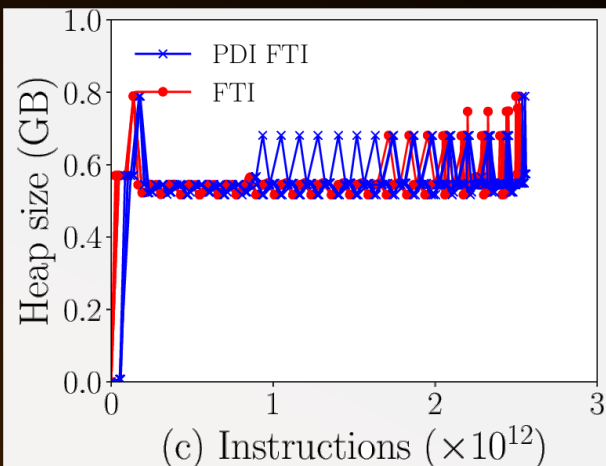
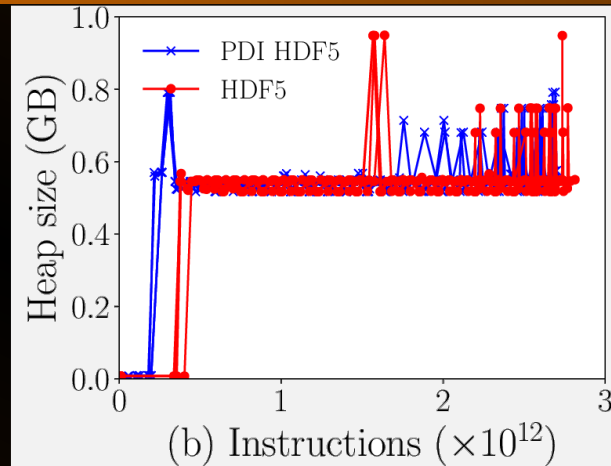
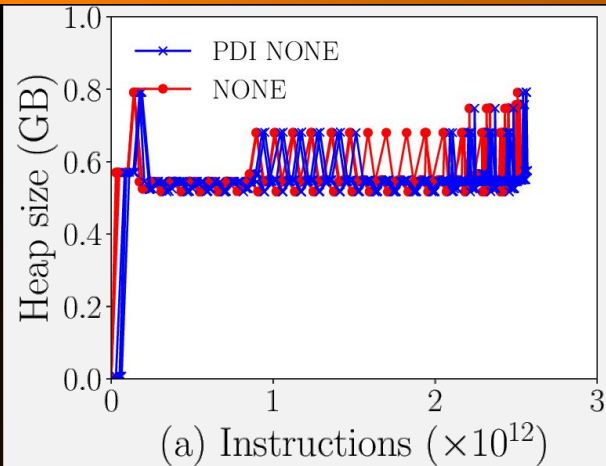
Gysela Wallclock time
in weak scaling on
Curie (TGCC –
France) with and
without PDI

Checkpointed data
~2.1GB/node



PDI: Memory overhead

Corentin Roussel (MdIS)
Kai Keller (BSC)

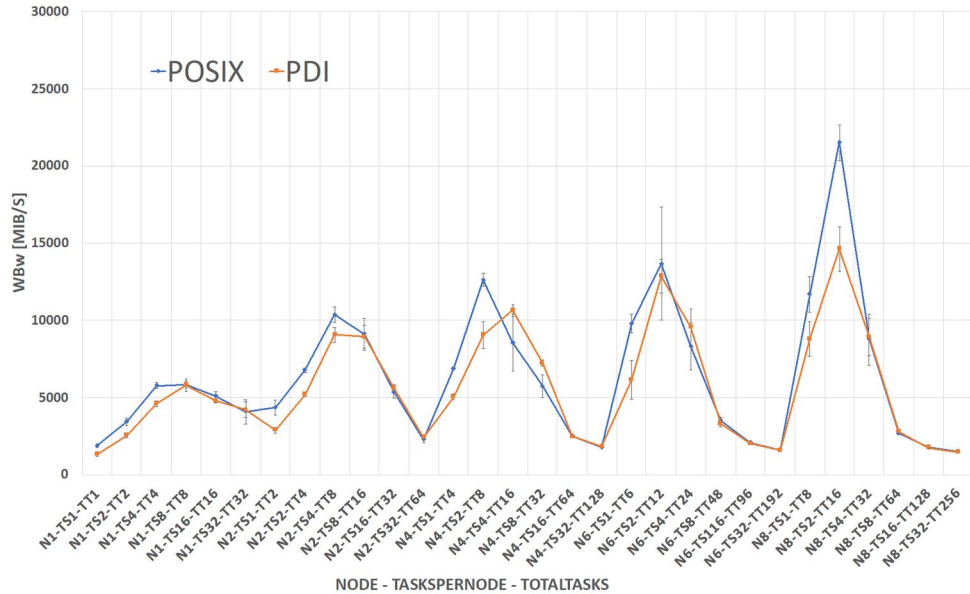


Memory usage during a Gysela execution with and without PDI on 4 nodes of MareNostrum (BSC – Spain)

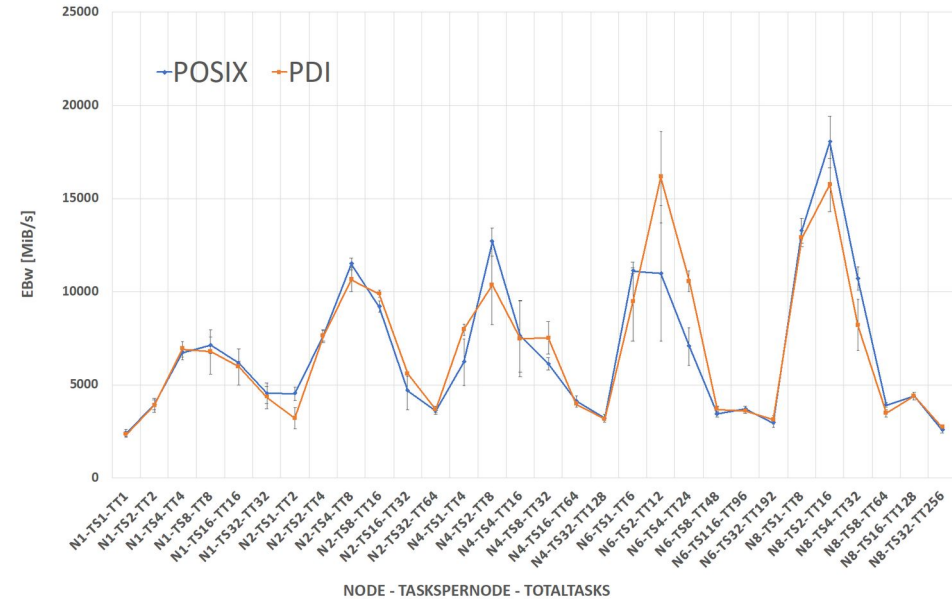


Perf evaluation: IOR

WRITE BANDWIDTH WITH BLOCKSIZE OF 128 kiB



WRITE BANDWIDTH WITH BLOCKSIZE OF 256 MiB



IOR IO Benchmark PDI integration

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

Francesco Iannone
(ENEA)



PDI In practice

- PDI is publicly available (BSD 3-clause license)
 - Version 1.5.4 released
 - Packages available for Debian, Fedora, Ubuntu, Spack
 - Documentation available @ <https://pdi.dev/1.5/>
 - Heavily tested & validated
 - more than 1500 tests
 - more than 14 platforms
- Integration in production codes
 - Gysela, Parflow, ESIAS, Metalwalls (Planned & funded)



Conclusion

- A library for Data Coupling, Not an IO library
 - A declarative annotation API
 - Describe your data in YAML
 - Multiple plugins for actual IO and data processing
 - Describe your IO from YAML
- Your turn now!
 - Get the tutorial: https://pdi.dev/master/Hands_on.html
 - Join the fun on <https://bit.ly/2OPmhA9>

