DUNE Software Framework Requirements Taskforce Report Annotated for Non-DUNE Review

Executive Summary

This taskforce report was commissioned by the DUNE computing consortium. The scope of the report was to provide an enumeration of the needs of the DUNE experiment, as driven by its physics mission, in regards to a software framework for data processing and analysis.

The taskforce was composed of representatives from the different physics missions of DUNE, scientists with extensive experience with large scale data processing and analysis from outside of DUNE[1] and technical experts in software framework design, including the current conveners of the HSF[2] frameworks working group.

In this report we have included additional annotations regarding the underlying rationale or driving principles that have lead to the information that is presented here. Annotations are provided as textual callouts to distinguish them from the report findings and requirements.

`This is an example of an annotation providing additional information.`

Task force members

Co-chairs - Andrew Norman (FNAL) and Paul Laycock (BNL)

DUNE members - David Adams (BNL), Adam Aurisano (U. Cinc), Chris Backhouse (UCL), Mary Bishai (BNL), Claire David (York), Tom Junk (FNAL), Tom LeCompte (ANL), Chris Marshall (LBL), Brett Viren (BNL)

Advisors - Brian Bockelman (Madison), Chris Jones (FNAL), Kyle Knoepfel (FNAL), Liz Sexton-Kennedy (FNAL), Vakho Tsulaia (LBL), Peter Van Gemmeren (ANL)

General framework requirements

For ease of reference, this executive summary lists the enumerated framework requirements defined by the Software Framework Requirements Task Force, more info on the task force can be found here:
https://wiki.dunescience.org/wiki/Software_Framework_Requirements_Task_Force

Brave readers are encouraged to read the full document to understand the context and nuances of each of the requirements, the wording here is the same as the full text. While there may be overlap, the complete set of requirements as derived from various considerations is presented

[1] Included scientists from Atlas, CMS, Belle II, NOvA, MicroBooNE, CDF and D0
[2] High Energy Physics Software Foundation

and no attempt at reducing this list is made here, rather that is left to framework designers when drawing up specifications. Considerations on “Utilities” and “Desired Features” are also presented at the end of the document to capture useful discussions and provide additional context.

We list in the following the formal requirements determined by the taskforce.
(1) The framework must separate data and algorithms.

(2) The framework must separate the persistent data representation from the in-memory representation seen by algorithms.

Configuration requirements

(3) The framework should provide a Turing complete configuration language as a foundational component so that it can ensure coherence of its configuration.

(4) The framework should provide a suitable API so that algorithm writers can ensure their required parameters are self-describing and validatable.

(5) The framework configuration system needs to have a robust persistency and versioning system that makes it easy to document and reproduce previous results. It must be possible to create, tag, check-sum, store and compare configurations. This configuration management system should be external to the framework or data files so that configurations can reliably be reused and audited.

Strong versioning and persistency are required to be provided at the framework level, so that the framework can track the versioning of data products, perform needed bookkeeping, and strongly enforce version consistency between algorithms and data. In addition versioning is required to be provided for the configurations that are used at runtime within the framework, so that any given data is fully reproducible. Moreover the neutrino community has a culture of embedding this information into their data so that the conditions under which a given set of data was produced can be determined from the persisted objects themselves. This becomes important in neutrino analyses which often deal with many “systematically” varied or derived versions of data and having the data be self describing is essential for the automation of the analysis tools.

(6) The resulting state of the configured framework and its components should be deterministic and precisely reproducible given a set of environmental conditions which include the available hardware, operating system, input data, etc.

It is recognized that in future heterogeneous computing environments, that results of algorithms may differ based upon the underlying platforms and architectures that they were run on, either due to the machine precision of the computational environment (i.e. reduced precision GPUs, which have significant computational speed benefits) or the concurrency [width] that they are run under. This is very different from previous generations of computation which were effectively homogeneous, taking place on the x86 architecture. We therefore require that the framework be aware of the platform and conditions under which is running, and that it bookkeep the data accordingly, so that differences between platforms can be enumerated and accounted for.

(7) The ensemble of the framework+environmental conditions must give reproducible results.

(8) It is desirable that it should be possible to only configure those framework components required for a particular data processing use case**.**

(9) It must be possible to derive the input data requirements for any algorithm, in order to define the sequence of all algorithms needed for a particular use case.

Concurrency and Multithreading

(10) It is (therefore) desirable that the framework should help facilitate the use of multi-threaded data processing and facilitate access to co-processors in an efficient manner.

The framework needs to be able to operate in both a multi-threaded environment and in a co-processor offload environment. More specifically it needs to provide facilities for either enforcing safety under multi-threaded concurrency, or for tagging/forcing regions into a safe serial operations mode. Similarly for offloading of computations to co-processors, the framework must either manage and enforce the data/memory coherency, or allow for the imposing of locks on data regions and co-processor states, and adjust execution flow accordingly. More generally we have NOT required in this specific requirement that the framework maintain coherency between non-local execution ranks of computations (i.e. if the framework were to utilize an MPI like execution model, each rank is required to maintain coherence but data exchange and reduction is not explicitly something that the framework would handle, rather we would rely on the MPI memory model to provide this coherence)

(11) It is highly desirable that it be possible to write algorithmic code independently of the framework.

(12) *It is highly desirable that the framework be sufficiently modular in design to allow re-use of framework services and functionality both within and outside of the framework context, as far as that is possible.*

(13) The framework must be able to schedule thread-safe and non-thread-safe work appropriately.

It is recognized that not all algorithms or data processing techniques are capable of being done in a thread safe manner. As a result, we need the ability to pass information to the framework to give it hints regarding scheduling. In particular we need the ability to tell the framework that a given code block is only safe for execution in a serial environment, or that a given block is safe for parallel execution up to some level. This can then be translated down into the actual scheduling algorithms which the framework uses.

Reproducibility and provenance

(14) The framework must provide a full provenance chain for any and all data products which must include enough information to reproduce identically every persistent data product. By definition, the chain will also need to include sufficient information to reproduce the transient data passed between algorithm modules, even though the product is not persisted in the final output.

(15) The framework must provide full provenance information and all of the metadata required to ensure reproducibility.

We assume in DUNE, as was done for previous neutrino experiments, that the event data model includes meta information which allows for the data to be fully self describing. This means that each data product carries with it a provenance chain which is tracked and propagated by the framework. In particular the I/O layers of the framework are expected to read and write this information in a manner that is transparent to the users and their algorithms. In prior frameworks, and this is an implementation detail, an SQLite database was embedded directly into the output files that the framework wrote. This database contained metadata and other related provenance information. The advantage of these self describing files approach are that they then do not require external information (e.g. a metadata database) to be available at runtime. It also prevents the loss of “data identity” when individuals copy files around and potentially change their names (i.e. the files know who they actually are and retain that identity even if someone manually renames them) and allows for easier extraction and aggregation of data from files (i.e. skimming) without loss of identity since the meta information can be carried at the event level. It is important to note that this is a viable strategy in part because of the large native event sizes of neutrino data, which then yield data to metadata ratios that are very favorable.

Random numbers, machine learning and conditions

(16) It is highly desirable that the framework broker access to random number generators and seeds in order to guarantee reproducibility.

(17) The framework should give special attention to machine learning inference in the design, both to allow simple exchanges of inference backends and to record the provenance of those backends and all necessary versioning information.

Machine learning inference is called out here as an example of an external calculation which is called out to by the framework. The framework must be aware that a portion of its scheduling is making an external call, which may have restrictions that it then consequently needs to impose back on the execution flow and data model. Moreover, the framework needs to be able to bookkeep the provenance of these styles of external calls to services which may have versioning information or configuration information that needs to be embedded back into the data. As a concrete example, it can be envisioned that the experiment has trained and configured two significantly different forms of neutrino interaction classification. The frame, at run time, is configured to call out to one of these and then the results are written to the data. Alternatively the configuration could have pointed to the other classification setup, and those results written. It is important for the framework to be able to track this, even though it is an external computation.

(18) The framework must provide a conditions service that is a single point of access to conditions data.

(19) The configuration of the framework conditions service should ideally be via one configuration parameter (a global tag).

(20) It must be possible to override a subset of global tag configured conditions for testing purposes.

Data and I/O layer

(21) The framework must support reading and writing different persistent data formats.

(22) The framework I/O functionality must be backward compatible across versions.

(23) A mechanism for user-defined schema evolution of data products needs to be provided.

(24) The framework must provide a mechanism to register/associate and to run/apply custom serialization/deserialization and compression/decompression algorithms to data on write/read of that data from a persistable form.

The nature of the liquid argon TPC data, makes it very amenable to specific types of compression in some forms, and to other compression methods in other forms. When this is combined with the very large nature of specific data products, it makes it advantageous to be able to specify how individual data products are treated when they are being written or restored from a data store. This should not be thought of as a traditional “data format converter”, but rather as an association of an algorithm that is applied to the data during its store/restore operations. In particular, for data that is being read, the algorithm for its deserialization and decompression needs to be embedded with the data so that the framework can perform the operations without intervention or manual configuration. We are trying to avoid situations where the data becomes unreadable because the end user of the data does not have (or can not determine) how a given data product was stored. This should be thought of as auto-magical read/write of data.

(25) The framework should support compression on output data in a manner that is transparent to users and is configurable. It must be possible to disable the automatic compression of output data or provide compression transforms that are effectively identity transforms.

(26) The framework should allow a configurable maximum output file size and provide appropriate file-handling functionality.

We have made an assumption that the framework is responsible for driving the I/O layer but retains control over the I/O layer. This is different from some past frameworks which delegated complete control to the I/O layer for reads/write. We believe that we need to have the framework retain control due to our parallelism/concurrency requirements, and our needs to interact with multiple I/O layers simultaneously.

Memory management

(27) The framework must be able to operate on subsets of a trigger record. Specifically, it must be possible to break trigger records down into smaller chunks (e.g. one APA) and be able to stitch those chunks back together. For supernovae, it must also be possible to reuse a fraction (nominally 100 us) of the previous chunk (nominally 5ms) of data to allow stitching in time.

This requirement should be interpreted as the framework being able to natively subset its primary data atom into a collection of new secondary data atoms which represent the same information, but can be processed or iterated on by the framework's "event loop" and then reconstituted back into the primary data atom either for subsequent processing or for recording by the I/O layers. The second part of this requirement, is that data from an adjacent data atom (specifically the time windows in a supernovae readout) be able to be stitched together with the current data atom to provide "edge effect" coverage in these types of extended time readouts (this applies to more than just supernovae detection). This is both an artifact of how the spatial/temporal degeneracies work in the TPC detector and the need to have contiguous time window coverage for the physical process of interest (i.e. the supernova). One effect of this requirement is that individual data atoms are not truly independent of each other in the event loop of a framework. Rather there is a weak form of hysteresis that needs to be accounted for, although it is thought that his can be mapped logically into a data overlay problem.

(28) Data products should not occupy memory beyond their useful lifetimes.

(29) The framework must manage memory of data products in the Data Store.

(30) The framework needs to support skimming/slimming/thinning for data reduction.

(31) The framework needs to be able to read and write several parallel data streams (including friend trees). Labelling of data objects across streams should be intuitive and not error prone. Provenance information should support correlating related data objects across streams.

This requirement arises from a number of different problems specific to the LArTPC data sizes and analysis techniques. In general, at the analysis level there is a desire to not carry around the full readout data (wire waveforms and derived hits) for the majority of the analysis. However in the late stages of the analysis there is a need/desire to reintroduce the full information content of the neutrino interaction candidate for operations like classification, or even mundane tasks like data visualization and display. As a result, the neutrino community needs a way to divide up the data objects that are part of a single event, subset them, and then later reassociate them back together. The framework needs to have the ability to deal with this, and provide the needed linkage and accounting facilities so that the provenance chain, normalizations, exposures and other information that are required to ensure that the final events records are coherent, are maintained. Essentially the framework needs to be able to split an input stream in a one-to-many operation, and then be able to perform the inversion of that operation as a many-to-one mapping.

(32) The framework needs to allow experiment code to mix simulation and (overlay) data.

Physics analysis

(33) It must be possible to define arbitrary units of execution that are independent of trigger records. It must be possible to correlate these units to trigger records for exposure accounting, and experiment conditions.

(34) The framework should make minimal assumptions about the data model (e.g. event-by-event or particle-candidate-based).

This does not mean that we are abandoning an event data model or an event loop. Rather it means that we don't make an assumption about what the base data atom is. In neutrino physics we often need to make contextual switches between what the unit of interest is, where examples are the subsetting of extended accelerator spills structures into smaller time windows, or the subsetting of time windows into disconnected regions of concentrated activity, or the subsetting of activity regions into shower and particle track trajectories and objects. We need our framework to be able to operate over these units in a native manner, instead of having everything below the highest organizational unit be looped over manually by a user module.

(35) Analysis must be able to use particle-candidate-based control flow, without any constraints arising from event-based control flow.

This is an outgrowth of the need to switch data atom context. Specifically this applies mainly to the exposure accounting that is needed when down shifting between different contexts. This can be complicated, but it is an essential concept in neutrino analyses.

(36) The framework must support partial reading of the persistent data and must not require reading an entire trigger record unless required (i.e. it must not force the entire trigger record to be read).

(37) Calibration, reconstruction, and selection algorithms must be framework-agnostic, i.e. able to run transparently in any official DUNE framework where equivalent requisite data products exist.

This is a bit odd. Basically we have collaborators who don't want to use the official DUNE framework, so they want it in writing that our framework will allow for them to not use our framework.

(38) The framework should be easily portable and capable of running on local resources.

We specifically did not specify a programming language or packaging system in the requirements. We are looking for a framework that is portable between architectures and environments. In particular we are NOT assuming that the framework is strictly tied to an x86 architecture, rather we assume that the framework and associated build system will be able to support running on the future equivalent of today’s single node x86 platforms as well as more exotic platforms as will be present in the DoE’s leadership computing facilities. This means that we also need the framework to be supportable across different flavors of GPU or other accelerators.

While this may seem like a stretch compared to what was done for the LHC or previous generations of neutrino analysis, we believe that the direction of the high performance computing landscape and the associated tools, compilers, and performance tuned library suites will make it possible to support this operational requirement.

(39) The same code developed and tested on local resources must scale to large resources. This should include HPC resources as far as possible.

The scale of resources we wish to support running on varies from based on the specific workflow in question. For DUNE it is important to remember that event data can be subsetted or partitioned easily on an APA (Anode Plane Assembly) by APA basis, and the large side of the data makes this division highly favorable both in terms of memory footprints and computational scaling. This gives a very natural 169 way parallelism, where for many of the algorithms there are then further divisions of that data that make sense.

When we talk about our scaling needs, we then consider that our framework needs to support a) Running on a single physical node with an N-way parallelism based on the core and memory configuration of the hardware (i.e. we don’t know what the machines will look like in 2026+ but if there is a machine with X cores, we want the framework to be able to support running on at least some of those in an efficient way). b) Running on a collection of nodes colocated on a common network fabric, each running one or more ranks of a computation (e.g. we would map each of 169 APA based divisions of the data to 169 separate ranks in the computation which would be spread over 169+1 separate nodes. This would allow each node to process one APA of data with a rank 0 handling reduction operations and allowing for further exploitation of parallelism on the individual nodes to support specific algorithm's needs.).

On today's machines we have successfully used this model with O(1-4k) concurrent colocated physical nodes, each with 40-68 cores per node. This has allowed us to run computations with in excess of 300k concurrent ranks per computation. While DUNE will not need quite this level of parallelism for our reconstruction or simulation workflows, we already explicitly do this for our fitting workflows, so we would want to be able to scale at or beyond this level in the future.

Our general goal for the DUNE framework and workflows is to be able to support running on machines with a million computational ranks plus offloads to dedicated accelerators.

(40) Analysis files must record their parent framework files, but no event-by-event provenance is required. The full provenance information need not be retained in analysis files as this could easily become larger than the data itself.

Our model in neutrino physics has always been to provide a full provenance chain which is reconstructable from a single data file. The way our analysis chains work, it is often more important to know "how" a number was derived than what the number is. This is mainly because of the way that we treat systematic variations of the data, and the way that we need to do our exposure counting on a spill by spill basis.

(41) The framework must have native support for exposure accounting (POT and live-time), so as to make errors of this sort difficult.

Our exposure counting is difficult due to the unobserved nature of the neutrino flux and the "live nature" of the detectors. As a result we need to have ways of accounting for our real exposure spill by spill, and adjusting it appropriately when we are forced to cut a spill. At the same time we often have a problem of "double counting" if we get data with more than one interaction during a spill (as happens with the near detectors) which needs to be handled properly when doing cross section measurements.

There is also the issue of “spill matched Monte Carlo” which we use to do our flux estimates. This means that the analysis techniques do not lend themselves to “averaged” exposures as are often used in external luminosity style databases.

(42) The framework should provide some means of cross-referencing (labelling) multiple input streams to correlate them in order to facilitate evaluation of systematic uncertainties.

This is a need of our methods for computing systematics. In an ideal system, we would compute a given value as its “nominal” (i.e. the nominal incident neutrino energy) and then we would have the same value computed simultaneously under some set of variations. This allows us to then do side-by-side comparisons of the

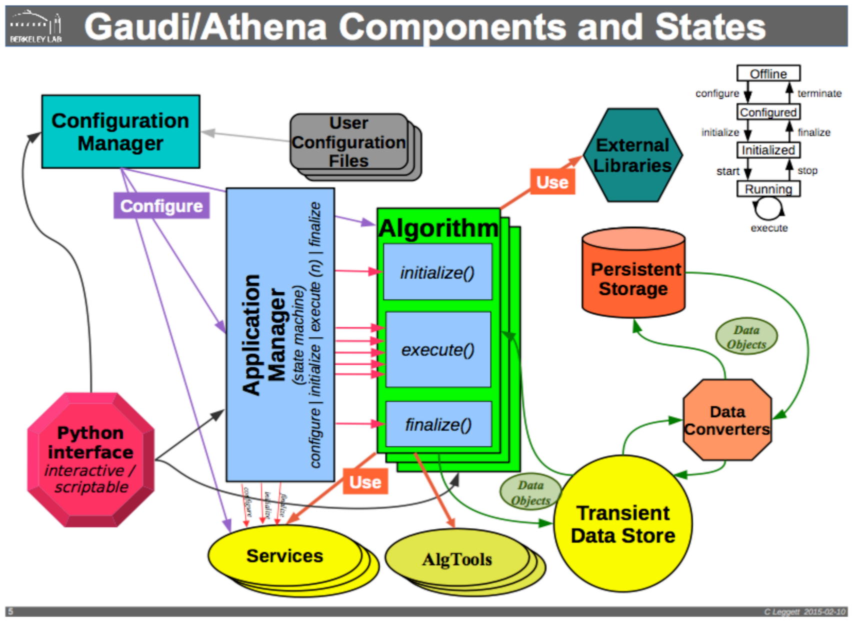
(43) The framework should be able to work with both ND and FD data on an equal footing, and within the same job.

DUNE Software Framework Requirements Taskforce Report
Full Report

<u>Document Scope</u>

Many experiments use data processing frameworks to process their data in a reliable way and to provide a structure for many authors to contribute algorithmic code. The main aim of this document is ***to define the physics-based requirements of the software framework*** that DUNE will use to process data, herein referred to as "the framework". Requirements are highlighted in ***bold italic*** for the executive reader and are enumerated in parentheses. The primary use cases considered relate to simulation and reconstruction of DUNE data in production runs using DUNE's distributed computing resources. Substantial and unique algorithmic functionality exists in established code and providing support for their continued exploitation is critical.

In the classic implementation, the *framework* is an executable program that loops over events and executes physics code that resides in *modules* (called *algorithms* in Gaudi) that are scheduled by that framework. The code in these modules is provided the means to read and write data from the current event and has access to other information (e.g. conditions data) via *services*. Services, tools and modules are all configurable and dynamically loaded. In the model envisioned here, most of the framework code resides in supporting services and the term "framework" refers to those services and the supporting libraries as well as the framework executable(s).



Schematic of the Gaudi/Athena framework components and states. Physics code resides in modules in (proto)DUNE, which are called Algorithms in Gaudi/Athena.

Late-stage processing, referred to here as “analysis”, could in principle use the same framework and that may be very advantageous, or even necessary, in certain use cases. Unlike production runs, analysis is characterised by rapid R&D cycles and an expectation of fast turnaround, often using ad hoc data and metadata. Therefore general framework design is an important topic considered here as it can easily, although unintentionally, effectively exclude the analysis use case by making rapid R&D very difficult. These design considerations are discussed with the aim of providing guidance for the final design, while the final design choices must be taken by the framework design team. The design of the framework will impose constraints on developers and these will need to be accepted by the developer community and documented with the framework. To the extent that is possible, it is highly desirable that the same services and tools can be configured and accessed inside and outside the framework executable. This has the implication that much of the functionality traditionally provided by the framework executable (scheduling, reading and writing of event data) be accessible outside of the framework context.

Heterogeneous computing and concurrency in general already plays an important role in scientific computing, promising greater speed and efficiency so long as we can utilise the disparate resources well. Again, general design considerations are presented here to provide guidance for the framework design. Technical design choices of e.g. whether developers should have direct access to concurrency tools like TBB, or whether they should be brokered by the framework is left to the framework design team.

An important use case not considered in this document is the use of the data processing framework in the online environment, particularly for a high-level trigger (HLT) which will present its own unique requirements (e.g. early termination of a trigger chain, forced accept, etc.). The most striking feature of the HLT usage pattern is the need to process a subset of data, usually corresponding to some region of interest. This pattern arises naturally in the use cases considered here.

General framework requirements

Frameworks typically use state transitions similar to those used in data acquisition and run control, including configuration, initialization, execution (where algorithmic code is typically run) and finalization. These state transitions are used by the framework to guarantee that all framework components have their work scheduled coherently. The components of a framework depend on the particular design but key requirements are

(1) the framework must separate data and algorithms

(2) the framework must separate the persistent data representation from the in-memory representation seen by algorithms.

For ease of reference, the “Data Store” is defined as the framework’s repository for storing data in memory beyond the lifetime of particular algorithms. Both “trigger record” and “event” are used to refer to the data belonging to one trigger record. More detail on these broad requirements follow in later sections.

The framework needs to perform many different data processing steps with potentially very many different variations. As the code base will be rather large, targeted (re)compilation for particular purposes is not desirable, especially if e.g. only a handful of parameters for one algorithm need to be changed. Instead, the components of the framework are configured to perform a particular task in a particular way, e.g. performing signal processing on raw input data.

Configuration requirements

Configuration is distinct from initialization of the framework objects; configuration happens first. Given that RAII is an important concept for multi-threaded design, it is best to have a fully configured framework instance in the initialization phase. Given the abundance of variations that make up HEP workflows a robust and easily programmable configuration system is a foundational component of all modern frameworks. Some use a strictly declarative language and some use a Turing complete language. The former must be augmented with scripts that write the declarations in a Turing complete language (usually it is part of the workflow management system, WMS) because it turns out that control flow is a requirement. Given this, there is a requirement that:

(3) the framework should provide a Turing complete configuration language as a foundational component so that it can ensure coherence of its configuration.

The WMS needs to be able to supply framework configuration parameters such as input file(s) or random number seeds to each framework application instance, which it should do using the framework’s configuration language. To minimise errors, these parameters should be self-describing and validatable, and so:

(4) the framework should provide a suitable API so that algorithm writers can ensure their required parameters are self-describing and validatable.

The framework should provide the concept of parameter sets that are nestable. The set of all parameter sets that define a framework application instance should be identifiable, referred to here as a FrameworkConfigID. Tracking that identity is one of the ingredients necessary to ensure scientific reproducibility. However some parameters do not (and should not) change the algorithmic results, such as a debug print flag. Independent of the state of such a flag it should be possible to define equivalence between FrameworkConfigIDs.

(5) The framework configuration system needs to have a robust persistency and versioning system that makes it easy to document and reproduce previous results. It must be possible to create, tag, check-sum, store and compare configurations. This configuration management system should be external to the framework or data files so that configurations can reliably be reused and audited.

Further fundamental requirements related to the framework configuration to guarantee reproducibility are:

(6) The resulting state of the configured framework and its components should be deterministic and precisely reproducible given a set of environmental conditions which include the available hardware, operating system, input data, etc.

(7) the ensemble of the framework+environmental conditions must give reproducible results.

Following on from the discussion of supporting fast analysis R&D:

***(8) it is desirable that it should be possible to only configure those framework components required for a particular data processing use case*.**

This generates a further requirement that

(9) it must be possible to derive the input data requirements for any algorithm, in order to define the sequence of all algorithms needed for a particular use case.

<u>Concurrency and Multithreading</u>

The arrival of concurrency and heterogeneous architectures has added a further level of complication for framework designers and developers alike. Much of the existing code-base still relies heavily on serial programming for CPU architectures, meanwhile the co-processor market is evolving rapidly resulting in a diverse hardware landscape. Both multi-threading and co-processors present challenges for both frameworks and developers.

(10) It is therefore desirable that the framework should help facilitate the use of multi-threaded data processing and facilitate access to co-processors in an efficient manner.

For developers, highly modular code must be encouraged, allowing evolution or replacement of sub-algorithms that lend themselves to particular approaches. It is assumed that algorithmic code will be organized in “algorithm modules” and this term is used in this document. The codebase will therefore likely contain several alternatives for (sub)algorithms, the choice of which would depend on the available hardware. The framework will need to run in heterogeneous and potentially dynamic environments where the availability and type of co-processors may be known late. While the design of this technically challenging aspect is left to framework developers, it is worth pointing out the added challenge of developing algorithms for diverse hardware if the framework does not easily allow it. Therefore:

(11) it is highly desirable that it be possible to write algorithmic code independently of the framework.

This also helps keep the gap between analysis and production code as low as possible. Furthermore,

(12) ***it is highly desirable that the framework be sufficiently modular in design to allow re-use of framework services and functionality both within and outside of the framework context, as far as that is possible.***

Multi-threading presents additional, well-documented challenges, particularly given that many important libraries are not thread-safe. Summarising briefly: algorithmic code, including sub-algorithms, should be thread-safe and must declare their compatibility to the framework.

(13) The framework must be able to schedule thread-safe and non-thread-safe work appropriately.

Thread-safety implicitly includes a general requirement on developers that algorithms and their sub-algorithms do not store state information in a thread-unsafe manner. Further, data exchange should be done in controlled ways to ensure thread-safety, e.g. via some service which manages transient data - again the implementation of this challenging aspect is left to the framework designers.

Reproducibility and provenance

The reproducibility of physics results and the knowledge of how physics results were obtained is essential to DUNE and to the neutrino community as a whole. It must be possible both to replicate physics results using identical input data, or to repeat an analysis using a different set of input data with an identical sequence of identically configured algorithms. Therefore,

(14) the framework must provide a full provenance chain for any and all data products which must include enough information to reproduce identically every persistent data product. By definition, the chain will also need to include sufficient information to reproduce the transient data passed between algorithm modules, even though the product is not persisted in the final output.

The need for transient data products is driven by the large event sizes that can be encountered in the DUNE data, and whose transformation may be required, but for storage considerations are not written out.

The use of highly parallel and heterogeneous computing environments leads to additional provenance requirements regarding the execution ordering and computing architectures on which the algorithms were executed. The need for this information arises because of the possibility of different computing architectures producing different results, and of accelerators and other computing offload mechanisms producing different results than serially executed or non-accelerated codes (i.e. GPU accelerated code producing different results than the same code executed on the host processor). Therefore

(15) the framework must provide full provenance information and all of the metadata required to ensure reproducibility.

This will include the computing architecture, including any specialized hardware used, on which the application is run as well as the runtime environment, execution model and concurrency level that the application used. It is likely that a full picture of the necessary metadata will also require information only known to the workflow management system. In such a complex environment, it is highly desirable that the framework provide support to allow the effect of configuration changes and computing environments to be easily understood.

Random numbers, machine learning and conditions

DUNE will use several libraries outside of the framework software stack, and it will also be necessary to record the precise versions of all of these libraries to guarantee reproducibility of the physics results. It is noted that containers (docker et al) could potentially make provenance tracking easier in this respect. Random number generation is an important aspect of code and given the additional complications of multi-threading and co-processors:

(16) it is highly desirable that the framework broker access to random number generators and seeds in order to guarantee reproducibility.

One important source of external libraries relates to machine learning, and machine learning inference is expected to play a significant role in all stages of data processing including analysis.

(17) The framework should give special attention to machine learning inference in the design, both to allow simple exchanges of inference backends and to record the provenance of those backends and all necessary versioning information.

In addition to the trigger record data, data processing requires access to non-event data from various sources, for example slow controls, detector status, beam component status. Such data is referred to generically as “conditions data” and also includes e.g. detector calibrations and any data external to the event data. The time granularity or “interval of validity” of this data varies by source and is typically of much coarser granularity than the event data, e.g. calibrations may be valid for months of data taking. Meanwhile there are often several versions of conditions data and correlations between conditions is not uncommon, making the coherent management of conditions data a challenge in itself. For this reason, conditions data management should be external to the framework.

Access to the external conditions data should preferably proceed via REST interfaces that support loose coupling of the framework and the external conditions management system. As conditions data may need to be transformed from its persistent format into a format required by an algorithm, and as multi-threading makes the cache validity of conditions data complicated:

(18) the framework must provide a conditions service that is a single point of access to conditions data.

(19) The configuration of the framework conditions service should ideally be via one configuration parameter (a global tag).

Developers must not hard-code conditions data in their algorithms, although

(20) it must be possible to override a subset of global tag configured conditions for testing purposes.

Developers usually find it convenient if such alternative conditions payloads can be provided outside of the main managed conditions system, e.g. via a local file.

Data and I/O layer

The main aim of this document is to describe the data processing steps for simulation and reconstruction of DUNE data. The first stage of processing in offline jobs, after job configuration and initialization, is reading in detector data. Offline jobs must read in data produced directly by the data acquisition system and also data produced by Monte Carlo simulation. Input data files may be retrieved from a persistent storage system, delivered over a network, or reside on local storage.

(21) The framework must support reading and writing different persistent data formats.

Every version of the framework must be able to read data files written by that version of the framework and all previous versions, with no loss in functionality or change in meaning of the data elements.

(22) The framework I/O functionality must be backward compatible across versions.

We do not require forward compatibility, in which data written with newer versions of the framework are also readable by older versions. Cases in which forward compatibility is broken need to be documented as far as possible, however, as these are breaking changes.

Experimenters may change their minds about the contents of data products. For example, data members may be added because they were initially not included in the design but later found to be necessary, and rather than create a new data product, an expansion of an old one is more convenient. Framework programs reading old and new data files need to behave seamlessly if the data product has changed definition.

(23) A mechanism for user-defined schema evolution of data products needs to be provided.

The DUNE data model allows for event data to be stored in persistable representations which are generated by customized hardware or which are optimized for specific acceleration hardware or computing systems. As a result, the data model expects that data will have custom “packed” representations that do not conform to 32-bit or 64-bit little-endian words. Furthermore, compression of the raw waveform data will be performed in the DAQ, though some data may arrive uncompressed. Some highly compressible data products may benefit from dedicated compression algorithms scheduled to run before output. Therefore,

(24) the framework must provide a mechanism to register/associate and to run/apply custom serialization/deserialization and compression/decompression algorithms to data on write/read of that data from a persistable form.

Data products that do not have dedicated compression algorithms associated with them can still benefit from automatic compression that is enabled by default.

(25) The framework should support compression on output data in a manner that is transparent to users and is configurable. It must be possible to disable the automatic compression of output data or provide compression transforms that are effectively identity transforms.

Experience shows that it is highly desirable to be able to configure a maximum file size such that output files are the correct size for efficient storage and for units of data processing; currently a file size of several GBs is considered optimal. As this requires the closure of an existing output file and creation and opening of a new file (with sensible filename) then this needs to be addressed at the framework level.

(26) The framework should allow a configurable maximum output file size and provide appropriate file-handling functionality.

Memory management

An issue that arises during data read-in and decompression and unpacking is the memory footprint used. The DUNE Far Detector is big. Supernova-burst (SNB) processing in the DUNE Far Detector presents unique challenges due to the large volume of data that are produced in each trigger. An uncompressed SNB readout for 100 seconds will take about 120TB of storage for one single-phase far detector module for just the TPC wire data, and DUNE will have four detector modules. These data will be divided into smaller chunks both in time and by detector component. For single-phase detector modules, these components are likely to be the anode plane assemblies (APA) due to the granularity of the data preparation processing.

In the first stage of offline processing, waveforms from the channels are de-noised and deconvolved, and pulses that are approximately Gaussian appear in the processed waveforms. Because the 100s of SNB readout from each channel must be artificially broken into small chunks in time, there is the potential to introduce edge effects at the chunk boundaries. In order to avoid this, about 100 microseconds of data spanning the boundary on both sides must be used in processing the chunks.

A common ratio of RAM to CPU cores on existing grids is 2 GB/core. Memory usage beyond this results in poor performance and can lead to job eviction depending on the resource configuration. An uncompressed DUNE Far Detector module trigger record will be larger than this, about 6 GB. A supernova trigger record of 100s will be more than five orders of magnitude bigger again. Clearly the framework will need to be able to act on subsets of trigger records, while respecting the overlap criteria noted above in order to avoid creating artefacts.

(27) The framework must be able to operate on subsets of a trigger record. Specifically, it must be possible to break trigger records down into smaller chunks (e.g. one APA) and be able to stitch those chunks back together. For supernovae, it must also be possible to reuse a fraction (nominally 100 us) of the previous chunk (nominally 5ms) of data to allow stitching in time.

Data unpacking and initial processing can be arranged to operate on these subsets. In order to realize the benefit from operating in this way, however, intermediate data products that are no longer needed must no longer occupy space in memory. Data products that have been written out and are no longer needed in memory are also good examples of those that can be evicted from memory, but there are also cases of intermediate products which must be flushed instead of written out.

(28) Data products should not occupy memory beyond their useful lifetimes.

As noted in "General Considerations", the framework must be aware of which algorithms need which data products, while the algorithmic code, by nature of its modularity and re-use/reconfigurability, is expected to be unaware of what other components may be run in the same job. Therefore, the framework must be responsible for garbage collection, capable of freeing up memory at the earliest possible time. Furthermore, the framework will need to respect the memory constraints imposed by the processing environment which will entail supporting partial reading of data objects into memory and potentially purging any data objects or partial data objects not immediately required.

(29) The framework must manage memory of data products in the Data Store.

Frameworks need to provide configurable, flexible I/O access so that experiments can control the output of their jobs in fine-grained detail. This is needed to save on storage and also experimenter time, as smaller datasets take less time to analyze than larger ones.

(30) The framework needs to support skimming/slimming/thinning for data reduction.

Similarly, processing and analysis is made much more convenient (or even possible) if the skimmed/slimmed/thinned output streams can be associated with information in other streams that may be stored separately. Some analyzers may need the auxiliary data streams while others may not, and so a framework job that produces outputs for collaboration use would need to read all of the necessary streams. This also allows efficient use of storage, as data does not need to be co-located in the same file to be available for processing. In the case of writing, I/O cost can be very efficiently amortized if several output streams can be written based on one input file.

(31) The framework needs to be able to read and write several parallel data streams (including friend trees). Labelling of data objects across streams should be intuitive and not error prone. Provenance information should support correlating related data objects across streams.

A common offline job need that goes beyond the 1->1 input to output data model is event mixing. Monte Carlo simulation is often not sufficiently realistic, perhaps it fails to capture the time dependence of detector conditions or the generator or detector simulation simply lacks sufficient accuracy for the physics use case. In this case, Monte Carlo simulation can be augmented by adding actual detector data, e.g. to embed single tracks or entire events into data trigger records and reconstruct them as if they were data. This can lead to a 2->many input output situation with asynchrony in both input and output.

(32) The framework needs to allow experiment code to mix simulation and (overlay) data.

<u>Physics analysis</u>

A SNB trigger record may have thousands of interesting physics interactions in it. They will be small tracks ("stubs"), of order of tens of wires hit in each plane, distributed through the detector and in time during the long trigger record. A convenient analysis workflow will save regions of interest to smaller files containing only data needed to analyze the small tracks and not the large amounts of waveform data containing only electronics noise and radiologicals. Most other trigger records, initiated by cosmic rays, beam neutrino interactions and atmospheric neutrino interactions, will have only one interaction in a Far Detector module. Some cosmic rays arrive in bundles with other cosmic rays, even at the 4850' level, and these are interesting to read out.

The near detector, on the other hand, will have many neutrino interactions per LBNF spill. The Gaseous Argon Near Detector Component (ND-GAr) expects 60 overlaid interactions per spill, mostly originating in its calorimeter, which has fast timing capabilities and thus can be used to separate one event's particles from another. The liquid-argon TPC near detector will have order of 20 interactions per spill. Some downstream analyses will benefit from and expect upstream analyses to divide a trigger record's data into subsets based on classification algorithms that are intended to separate one interaction from another. These physics regions of interest, usually called “slices”, are then what physicists expect to use as a unit of execution, i.e. they loop over slices in their analysis code.

(33) It must be possible to define arbitrary units of execution that are independent of trigger records. It must be possible to correlate these units to trigger records for exposure accounting, and experiment conditions.

Here, experiment conditions refers to data collected outside of the trigger-record data stream. It consists of monitoring data from slow controls, detector status, and beam component status.

Data analysis presents its own challenges, and requires different tradeoffs to the preceding simulation/reconstruction/particle identification stages. Analysis includes the extraction of oscillation parameters, but is not limited to that, encompassing, as a minimum, comparisons between data and Monte Carlo, extraction of calibration and detector performance parameters, cross-section measurements, measurements of atmospheric, solar, and supernova neutrinos, and searches for non-standard phenomena.

The event-by-event paradigm is not a good match here. Spectra are filled by looping over neutrino candidates (or cosmic rays, or candidate exotic events) but then may undergo substantial processing in their own right. For example, the main work of an oscillation fit is the evaluation of many different combinations of oscillation and nuisance parameters. While slices provide sub-event control flow, particle candidate control flow ignores the event structure entirely.

Efficient data access for typical analysis workflows is also very different to event-by-event processing. It is very common to require access to only a small subset of the event information (the variables required to make a cut, or reconstruct a quantity), and for the amount of information to vary from event to event. For example ROOT TTrees use a “column-wise” data layout to make this access pattern efficient, but any event-by-event serialization process will produce “row-wise” data. This may have implications on the persistent and transient Data Models and corresponding I/O layer implementations. Taking the above into consideration, the requirements for the framework are:

(34) The framework should make minimal assumptions about the data model (e.g. event-by-event or particle-candidate-based).

(35) Analysis must be able to use particle-candidate-based control flow, without any constraints arising from event-based control flow.

(36) The framework must support partial reading of the persistent data and must not require reading an entire trigger record unless required (i.e. it must not force the entire trigger record to be read).

If these requirements cannot be reasonably satisfied then it is highly likely analysis would need to use a different framework to that used for data processing, but it is noted that e.g. Belle II has a similar dichotomy satisfied by one framework.

In the case where the data-processing and analyses frameworks are separated, it is desirable for a level of compatibility to be maintained between the frameworks. In previous experiments, there is a common pattern where, as the analysis framework matures, graduate students cease to be able to develop in the data-processing framework. To avoid this, compatibility layers may be required to allow algorithms (such as calibration, reconstruction, and selection methods) to transparently plug into either framework, to the extent it is possible.

(37) Calibration, reconstruction, and selection algorithms must be framework-agnostic, i.e. able to run transparently in any official DUNE framework where equivalent requisite data products exist.

Analysis work will be undertaken by a large number of collaborators, with varying levels of experience. In many cases, rapid feedback and iteration will be required to make progress. Meaningful analysis work must be possible with resources available locally to collaborators (single CPU, few gigabytes of memory, <100GB of disk). It must be possible to produce relevant histograms on interactive timescales (ie minutes). This re-emphasises the earlier requirements in General Considerations and in addition:

(38) The framework should be easily portable and capable of running on local resources.

Experience shows that oscillation fits accounting for large numbers of systematic uncertainties are resource intensive, while analysts will likely only have access to modest local resources for prototyping and development. Therefore the framework should make scaling and concurrency transparent both to the analyst and the developer as far as possible. The use of declarative analysis techniques should be strongly encouraged to support this even when co-processors (and low-level implementation) changes.

(39) The same code developed and tested on local resources must scale to large resources. This should include HPC resources as far as possible.

Analysis files, of course, are derived from the data-processing framework files, and it must be possible to reconstruct this history. Due to the very large number of events expected to be summarized in a single analysis file, the size requirements, and the fact that per event information remains available in the parent files, we require:

(40) Analysis files must record their parent framework files, but no event-by-event provenance is required. The full provenance information need not be retained in analysis files as this could easily become larger than the data itself.

One common and insidious class of mistakes is errors in exposure accounting and normalization. This is also a problem that is entirely solvable at the technical level. Each individual event (beam spill or other trigger) has exposure associated with it, whether POT or livetime or both. When filling a summary histogram from events the exposure should be calculated and stored as an integral part of the histogram, and operations between histograms should take correct notice of the exposure, e.g. ratio of one large exposure sample to a smaller exposure sample should produce a dimensionless ratio that has allowed for the differing exposures.

(41) The framework must have native support for exposure accounting (POT and livetime), so as to make errors of this sort difficult.

All but the simplest analyses require a treatment of systematic uncertainties. There are three main technical means by which these systematics can be introduced. The most common, and most convenient, is reweighting. For example, the effect of various cross-section and flux uncertainties may be encapsulated by applying weights to events of certain categories, to increase or decrease their representation in the final spectra. Secondly, events may be shifted. For example, an energy scale uncertainty may be most conveniently represented by rewriting of event records to increase or decrease reconstructed energies by a certain amount. Finally, the least convenient method is alternate simulation samples. The profusion of files requiring processing and bookkeeping makes this a heavyweight option, but in the case of uncertainties early in the analysis chain with complex effects, it may be the only way to handle them accurately. The treatment of systematics is cross-cutting across all analyses, it is important it is handled correctly, and the framework is able to offer substantive technical assistance.

In addition to being able to handle multiple input data streams:

(42) The framework should provide some means of cross-referencing (labelling) multiple input streams to correlate them in order to facilitate evaluation of systematic uncertainties.

For oscillation analysis, it will be important to work with both Near and Far detector data. Whether in an explicit joint fit, or where extracting constraints from the ND to apply to the FD analysis, there must be a uniformity in the treatment of various systematics. In general, experience gained with the Near Detector (where the majority of analysis work is likely to happen) should be transferable to the Far Detector. This re-emphasises the importance of the framework making minimal assumptions about the Data Model.

(43) The framework should be able to work with both ND and FD data on an equal footing, and within the same job.

While most of the focus at this stage of the experiment is on other tasks, in the long run analysis work will dominate. The outsized influence of decisions about data formats and analysis infrastructure mean that they should be afforded specially-careful consideration. This is also a very dynamic area, with rapid timescales and the possibility of important new ideas, so we must also remain flexible.

Utilities

It should be possible to read/write framework-format data outside of the framework.

Utilities must be supplied that list data product names, sizes, and compressed sizes in files written by the framework.

Desired features

Care must be taken by the experiments to label mixtures of real data and simulated data as simulations. Experiment code may expect MC truth labels on all objects when in fact only some may be available, or possibly all are absent by choice. A method or a flag that returns a binary “isRealData” value may be too coarse-grained. Issues that need to be addressed are the handling of bookkeeping: trigger record (“event”) numbering, handling of MC truth information in what otherwise is a data trigger record, and any code that performs different actions on MC and data. Analyzing mixed events will almost certainly require access to values from run conditions and calibration databases, and these have to be understood that they are the ones the user desires.

Another goal to think about without formally requiring it is to allow and encourage the overlapping of computation and I/O operations. I/O operations often take significant wall-clock time that can be used for computation if possible. The framework should support concurrent execution of both I/O and algorithm components and in a manner that allows for multiple units of data, not limited to just trigger records, to be processed concurrently (data pipelining).

Production processing usually involves one or more subsequent reprocessing passes, either after calibration constants have been calculated from the data or from external sources, or to accommodate new algorithms. Sometimes these reconstruction passes start with raw data as input, and sometimes they can take advantage of previous passes initial data processing stages (such as signal processing) and work on higher-level objects, such as hits. The framework needs to allow for this sort of workflow to be configured and run by the experiments. Sometimes the results of an earlier processing pass need to be compared with those of a later pass, and the option to keep both sets of results in the output for comparison is useful.

It should be possible to reprocess data produced by the framework, both in re-running some or all previously run processing steps, and to add new processing steps. The output must be properly labeled and configurable so that the output of the earlier processing steps may be distinguished, dropped or retained.

The fully processed data and Monte Carlo from the experiment are likely to run to multiple terabytes. In order for analysis, which must necessarily consume the entire dataset, a hugely reduced data representation is required. It would be technologically feasible to do this by aggressive summarizing and slimming of the framework files from previous stages, and this does have some benefits, but due to the radically different demands of analysis and the benefits of alternate structurings of the data we explicitly do not make this a requirement or even a recommendation.

Experience has shown the importance of compatibility between disparate analysis efforts. This allows work on improving the data formats and analysis framework to be shared, allows experience and expertise from one area to carry over to another without retraining, reduces the opportunity for bugs, and saves huge amounts of time spent attempting to compare between or reconcile analyses.

There should be a common analysis data format, shared by virtually all analysis efforts.

Of course, this does not mean all analyses must use precisely the same files. Different aspects of data analysis will require radically different event selections, and use non-overlapping properties of the data. We refer only to technical compatibility. This skimming work is expected to be common, and analysis is expected to iterate substantially more rapidly than data processing, including the inevitable respins for mistakes.

It should be possible to create a new iteration of the analysis files from the data-processing framework files on a reduced timescale (weeks) and to create skim files rapidly (days).